.NET 기본 사항 설명서

다양한 유형의 애플리케이션을 빌드하기 위한 오픈 소스 개발자 플랫폼인 .NET 기본 사항을 알아봅니다.

.NET 대해 알아보기

다운로드

.NET 다운로드

개요

.NET란?

.NET 소개

.NET 언어

개념

.NET Standard

CLR(공용 언어 런타임)

.NET Core 지원 정책

새로운 기능

.NET 11의 새로운 기능

.NET 10의 새로운 기능

.NET 9의 새로운 기능

.NET 8의 새로운 기능

설치 .NET

개요

사용할 .NET 버전 선택

방법 가이드

.NET SDK 설치

.NET 런타임 설치

Linux 패키지 관리자를 사용하여 설치

설치된 버전 확인

참조

.NET SDK 및 런타임 종속성

.NET 시작하기

시작하기

.NET 사용 시작하기

ASP.NET Core 시작하기

.NET Q&A에서

.NET 기술 커뮤니티 포럼

VIDEO

자습서: 10분 만에 Hello World

자습서

Hello World 앱 만들기

.NET Core 앱 컨테이너화

개념

.NET Framework에서 .NET으로 마이그레이션

배포

앱 게시

GitHub Actions 사용하여 .NET 앱 게시

데이터 직렬화

개념

JSON 직렬화 및 역직렬화

방법 가이드

C를 사용하여 JSON 직렬화 및 역직렬화#

Newtonsoft.Json에서 System.Text.Json으로 마이그레이션

JSON serialization에 대한 사용자 지정 변환기 작성

SAMPLE

XML serialization의 예

런타임 라이브러리.

개요

런타임 라이브러리 개요

개념

.NET에서의 종속성 주입

.NET 구성

.NET 로그인

제네릭 호스트 .NET

.NET 작업자 서비스

.NET 캐싱

.NET에서의 HTTP

.NET 지역화

.NET 파일 탐색

자습서

사용자 지정 구성 공급자 구현

컴파일 타임 로그 소스 생성하기

BackgroundService를 사용하여 Windows 서비스 만들기

날짜, 숫자 및 문자열 서식 지정 및 변환

개념

숫자 형식 문자열

날짜 및 시간 형식 문자열

복합 형식 지정

표준 시간대 간 시간 변환

문자열에서 문자 자르기 및 제거

.NET에서의 정규 표현식

방법 가이드

문자열을 DateTime으로 변환

숫자에 앞자리 0을 추가하기

날짜 및 시간 값의 밀리초 표시

참조

정규식 언어

이벤트 및 예외 사용

개념

예외에 대한 모범 사례

이벤트를 처리하고 발생시키기

방법 가이드

try-catch 블록을 사용하여 예외를 잡다

이벤트 발생 및 사용

파일 및 스트림 I/O

개념

파일 및 스트림 I/O

Windows 파일 경로 형식

방법 가이드

파일에 텍스트 쓰기

파일에서 텍스트 읽기

파일 압축 및 추출

로그 파일 열기 및 추가

.NET 시작

이 자습서에서는 파일 기반 앱을 사용하여 첫 번째 .NET 앱을 만들고 실행하는 방법을 설명합니다. 간단한 앱을 작성하고 코드 실행 결과를 확인합니다.

이 자습서에서는 다음을 수행합니다.

- ✓ .NET 개발 환경을 사용하여 GitHub Codespace를 시작합니다.
- ✓ 첫 번째 .NET 앱을 만듭니다.
- ✓ 앱을 실행합니다.

필수 조건

다음 옵션 중 하나가 있어야 합니다.

- GitHub Codespaces를 사용하는 GitHub 계정입니다. 아직 계정이 없는 경우 GitHub.com 체험 계정을 만들 수 있습니다.
- 다음 도구가 설치된 컴퓨터:
  - .NET 10 SDK입니다.
  - Visual Studio Code
  - C# DevKit입니다.

Codespaces 열기

자습서 환경에서 GitHub Codespace를 시작하려면 자습서 코드스페이스 리포지토리에 대한 브라우저 창을 엽니다. 녹색 **코드** 단추와 **Codespaces 탭을** 선택합니다. 그런 다음, `+` 이 환경을 사용하여 새 Codespace를 만드는 기호를 선택합니다.

첫 번째 앱 만들기 및 실행

1. 코드스페이스가 로드되면 `tutorials` 폴더에 새 `hello-world.cs` 파일을 만드세요.
2. 새 파일을 엽니다.
3. 다음 코드를 입력하거나 `hello-world.cs`에 복사합니다.

```C#
Console.WriteLine("Hello, World!");
```

4. 통합 터미널 창에서 폴더를 `tutorials` 현재 폴더로 만들고 앱을 실행합니다.

.NET CLI

```
cd tutorials
dotnet hello-world.cs
```

첫 번째 .NET 앱을 실행했습니다. "Hello, World!" 메시지를 인쇄하는 간단한 앱입니다. 메서드를 Console.WriteLine 사용하여 해당 메시지를 인쇄합니다. `Console` 콘솔 창을 나타내는 형식입니다. `WriteLine` 텍스트 콘솔에 텍스트 줄을 인쇄하는 `Console` 형식의 메서드입니다.

축하합니다! 간단한 .NET 애플리케이션을 만들었습니다.

리소스 정리

GitHub는 30일 동안 비활성 상태이면 Codespace를 자동으로 삭제합니다. .NET 자습서를 계속 진행하려는 경우 Codespace를 프로비전된 상태로 둘 수 있습니다. 컴퓨터에 .NET SDK를 다운로드할 준비가 되면 Codespace를 삭제할 수 있습니다. Codespace를 삭제하려면 브라우저 창을 열고 Codespaces 로 이동합니다. 창에 코드스페이스 목록이 표시됩니다. 자습서 코드스페이스의 항목에서 점 3개(`...`)를 선택하고 **삭제**를 선택합니다.

다음 단계

단계별 자습서를 수행하거나 YouTube에서 .NET 101 비디오를 시청하여 .NET 애플리케이션 개발을 시작합니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 22.

.NET 시작하기 위한 자습서

다음 단계별 자습서는 명시된 경우를 제외하고 Windows, Linux 또는 macOS에서 실행됩니다.

앱을 만들기 위한 자습서

- 콘솔 앱 만들기
- 웹앱 만들기
  - 서버 측 웹 UI 사용하여
  - 클라이언트 쪽 웹 UI 사용하여
- 웹 API 만들기
- 원격 프로시저 호출 웹앱 만들기
- 실시간 웹앱 만들기
- 서버리스 함수를 클라우드에 만들기
- .NET MAUI
- Windows 데스크톱 앱 만들기
  - WPF
  - Windows Forms
  - Universal Windows Platform (UWP)
- Unity 사용하여 게임 만들기
- Windows 서비스 만들기

클래스 라이브러리를 만들기 위한 자습서

- 클래스 라이브러리 만들기

.NET 언어를 학습하기 위한 리소스

- C# 시작하기
- F# 시작하기
- Visual Basic 시작하기

기타 시작 리소스

다음 리소스는 .NET 앱 개발을 시작하기 위한 것이지만 단계별 자습서는 아닙니다.

- IoT(사물 인터넷)
- 기계 학습

다음 단계

.NET에 대한 자세한 내용은 .NET 소개를 참조하세요.

Last updated on 2026. 03. 11.

Windows, Linux, macOS에 .NET 설치

Windows, Linux 및 macOS에 .NET을 설치하는 방법을 알아봅니다. .NET 앱을 개발, 배포, 실행하는 데 필요한 종속성을 검색합니다.

Windows

개요

Windows에 설치

지원되는 버전

Visual Studio를 사용하여 설치

Visual Studio Code로 설치

macOS

개요

macOS에 설치

지원되는 macOS 릴리스

Visual Studio Code와 함께 설치

Linux

개요

Linux 개요

Alpine

CentOS

Debian

Fedora

openSUSE

Red Hat Enterprise Linux 및 CentOS Stream

SUSE Linux Enterprise Server

Ubuntu

Q&A

시작하기

독립 실행형 설치 프로그램

Visual Studio 설치 관리자

Linux 피드

Docker

엔터프라이즈 배포

Windows .NET 설치

이 문서에서는 Windows 지원되는 .NET 버전, .NET 설치하는 방법 및 SDK와 런타임 간의 차이점을 설명합니다.

.NET Framework와 달리 .NET Windows 버전에 연결되지 않습니다. Windows 단일 버전의 .NET Framework만 설치할 수 있습니다. 그러나 .NET 독립 실행형이며 컴퓨터의 아무 곳에나 설치할 수 있습니다. 일부 앱에는 자체 .NET 복사본이 포함될 수 있습니다.

설치 방법이 다른 디렉터리를 선택하지 않는 한 기본적으로 .NET 컴퓨터의 *Program Files\dotnet* 디렉터리에 설치됩니다.

> **Important**
>
> 시스템 전체에서 .NET 설치하는 경우 관리자 권한으로 설치합니다.

.NET 런타임 및 SDK로 구성됩니다. 런타임은 .NET 앱을 실행하고 SDK는 앱을 만드는 데 사용됩니다.

올바른 런타임 선택

Windows 여러 유형의 앱을 실행할 수 있는 세 가지 런타임이 있습니다. SDK에는 세 개의 런타임이 모두 포함되며 런타임에 대한 설치 관리자에는 추가 런타임이 포함될 수 있습니다. 다음 표에서는 특정 .NET 설치 관리자에 포함되는 런타임에 대해 설명합니다.

테이블 확장

설치 프로그램	.NET 런타임 포함	.NET 데스크톱 런타임 포함	ASP.NET Core 런타임 포함
.NET 런타임	예	아니요	아니요
.NET 데스크톱 런타임	예	예	아니요
ASP.NET Core 런타임	아니요	아니요	예
.NET SDK	예	예	예

Windows 모든 .NET 앱을 실행할 수 있도록 하려면 ASP.NET Core 런타임과 .NET 데스크톱 런타임을 모두 설치합니다. ASP.NET Core 런타임은 웹 기반 앱을 실행하고 .NET 데스크톱 런타임은 Windows Presentation Foundation(WPF) 또는 Windows Forms 앱과 같은 데스크톱 앱을 실행합니다.

.NET 설치하는 방법 선택

.NET 설치하는 방법에는 여러 가지가 있으며 일부 제품은 자체 버전의 .NET 관리할 수 있습니다. 자체 버전의 .NET 관리하는 소프트웨어를 통해 .NET 설치하는 경우 시스템 전체에서 사용하도록 설정되지 않을 수 있습니다. 다른 소프트웨어를 통해 .NET 설치하는 것이 미치는 영향을 이해해야 합니다.

다음 섹션에서 목록을 검토한 후 어떤 방법을 선택해야 할지 잘 모르는 경우 .NET Installer 사용하려고 할 것입니다.

개발자

- Visual Studio

  Visual Studio 사용하여 .NET 앱을 개발하려는 경우 **Visual Studio**를 사용하여 .NET 설치합니다. Visual Studio 자체 .NET 복사본을 관리합니다. 이 메서드는 SDK, 런타임 및 Visual Studio 템플릿을 설치합니다.

- Visual Studio Code - C# 개발 키트

  Visual Studio Code **C# Dev Kit** 확장을 설치하여 .NET 앱을 개발합니다. 확장 기능은 이미 설치된 SDK를 사용하거나, 아직 설치되지 않았을 경우 설치해 줄 수 있습니다.

사용자 및 개발자

- .NET Installer

  실행하는 실행 파일인 Windows Installer 패키지를 사용하여 .NET 설치합니다. 이 방법을 사용하면 SDK와 런타임을 설치할 수 있습니다. 설치는 시스템 전체에서 수행됩니다.

- Windows Package Manager(WinGet)

  명령줄을 통해 .NET 관리하려는 경우 **WinGet**를 사용하여 .NET 설치합니다. 이 방법을 사용하면 SDK와 런타임을 설치할 수 있습니다. 설치는 시스템 전체에서 수행됩니다.

- PowerShell

  SDK 또는 런타임 설치를 자동화할 수 있는 PowerShell 스크립트입니다. 설치할 .NET 버전을 선택할 수 있습니다.

지원되는 버전

다음 표는 현재 지원되는 .NET 릴리스 및 지원되는 Windows 버전 목록입니다. 이러한 버전은 .NET 버전이 지원 종료에 도달하거나 Windows 버전이 수명 종료 도달할 때까지 계속 지원됩니다.

> **팁**
>
> 참고로, 이 테이블은 .NET Framework와 달리 최신 .NET에 적용됩니다. .NET Framework를 설치하려면 **.NET Framework 설치 가이드** 참조하세요.

Windows 10 버전의 서비스 종료 날짜는 버전별로 구분됩니다. **Home**, **Pro**, **Pro Education** 및 **Pro for Workstations** 버전만 다음 표에서 다룹니다. 자세한 내용은 Windows 수명 주기 팩트 시트를 확인하세요.

테이블 확장

운영 체제	.NET 10(아키텍처)	.NET 9(아키텍처)	.NET 8(아키텍처)
Windows 11 (24H2, 23H2, 22H2 엔터프라이즈/교육)	✔ x64, Arm64	✔ x64, Arm64	✔ x64, Arm64
Windows 10(22H2)	✔ x64, Arm64	✔ x64, Arm64	✔ x64, Arm64
Windows Server 2025년 Windows Server 2022 Windows Server 2019 Windows Server 버전 1903 이상 Windows Server 2016 Windows Server 2012 R2 Windows Server 2012	✔ x64, x86	✔ x64, x86	✔ x64, x86
Windows Server Core 2012(및 R2)	✔ x64, x86	✔ x64, x86	✔ x64, x86
Nano 서버(2025, 2022, 2019)	✔ x64	✔ x64	✔ x64
Windows 8.1	✘	✘	✘
WINDOWS 7 SP1 ESU	✘	✘	✘

> **팁**
>
> `+` 기호는 최소 버전을 나타냅니다.

Windows 7 / 8.1 / Server 2012

더 이상 **Windows 7 및 Windows 8.1** 지원되는 .NET 버전이 없습니다. 마지막으로 지원되는 릴리스는 .NET 6이며 지원은 2024년 11월 12일에 종료되었습니다.

Windows Server 2012 여전히 지원되는 모든 버전의 .NET 지원됩니다.

이러한 세 버전의 Windows 모두 추가 종속성을 설치해야 합니다.

테이블 확장

운영 체제	필수 조건
WINDOWS 7 SP1 ESU	- Microsoft Visual C++ 2015-2019 재배포 가능 64비트 / 32비트 - KB3063858 64비트 / 32비트 - Microsoft Root Certificate Authority 2011 (.NET Core 2.1 오프라인 설치 관리자만 해당)
Windows 8.1	Microsoft Visual C++ 2015-2019 재배포 가능 64비트 / 32비트
Windows Server 2012	Microsoft Visual C++ 2015-2019 재배포 가능 64비트 / 32비트
Windows Server 2012 R2	Microsoft Visual C++ 2015-2019 재배포 가능 64비트 / 32비트

다음 dll 중 하나와 관련된 오류를 수신하는 경우에도 이전 요구 사항이 필요합니다.

- *api-ms-win-crt-runtime-l1-1-0.dll*
- *api-ms-win-cor-timezone-l1-1-0.dll*
- *hostfxr.dll*

Arm 기반 Windows PC

.NET Arm 기반 Windows PC에서 지원됩니다. 다음 섹션에서는 .NET 설치할 때 고려해야 할 사항에 대해 설명합니다.

경로 차이

Arm 기반 Windows PC에서는 모든 Arm64 버전의 .NET 일반 *C:\Program Files\dotnet* 폴더에 설치됩니다. 그러나 .NET SDK의 **x64** 버전은 *C1 /Program Files\dotnet\x64* 폴더에 설치됩니다.

경로 변수

`PATH` 변수와 같이 시스템 경로에 .NET 추가하는 환경 변수는 x64 및 Arm64 버전의 .NET SDK가 모두 설치된 경우 변경해야 할 수 있습니다. 또한 일부 도구는 적절한 .NET SDK 설치 폴더를 가리키도록 업데이트해야 하는 `DOTNET_ROOT` 환경 변수를 사용합니다.

Visual Studio를 사용하여 설치

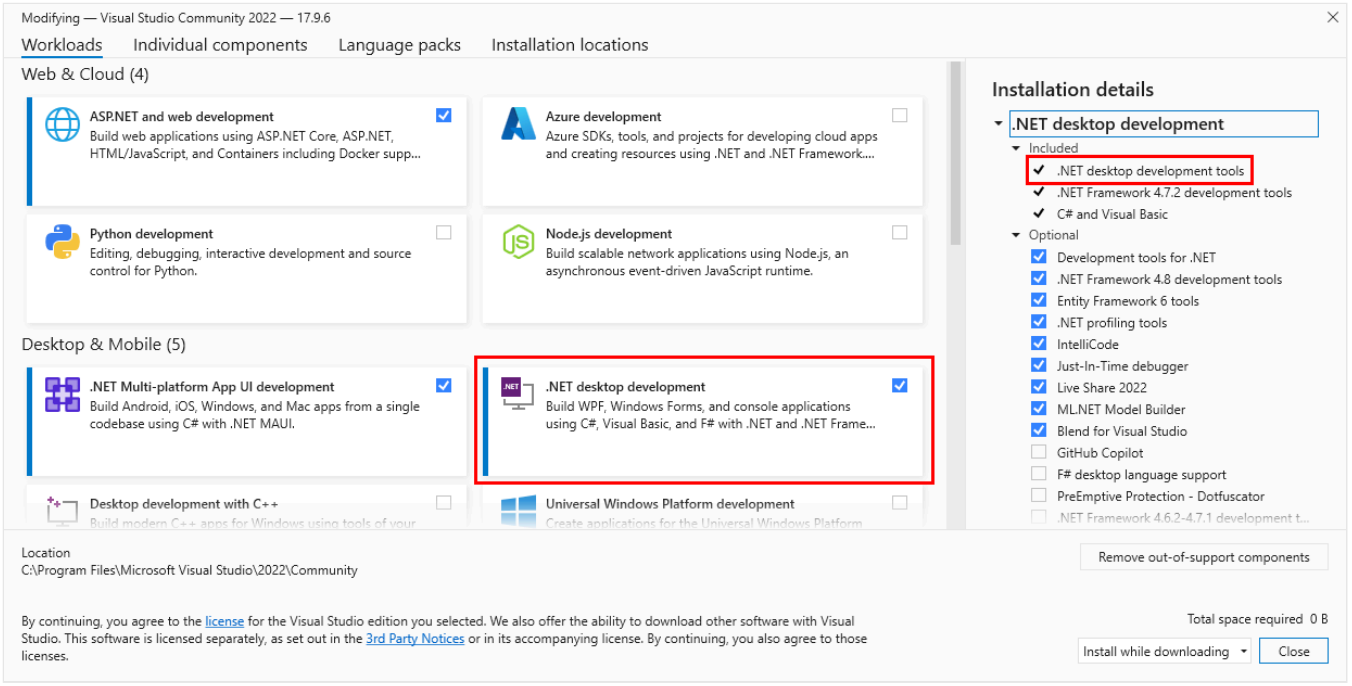
Visual Studio 다른 .NET 복사본과는 별도로 .NET 자체 복사본을 설치합니다. 다른 버전의 Visual Studio 다양한 버전의 .NET 지원합니다. Visual Studio 최신 버전은 항상 최신 버전의 .NET 지원합니다.

Visual Studio Installer 다운로드

Visual Studio 설치 관리자는 Visual Studio 설치하고 구성합니다. 일부 Visual Studio 워크로드에는 **ASP.NET 및 웹 개발** 및 **.NET 다중 플랫폼 앱 UI 개발** 같은 .NET 포함됩니다. 특정 버전의 .NET **분할 구성 요소** 탭을 통해 설치할 수 있습니다.

Visual Studio 설명서에서는 다음 방법에 대한 지침을 제공합니다.

- Visual Studio 설치.
- Visual Studio 워크로드 구성.



.NET 버전 및 Visual Studio

Visual Studio 사용하여 .NET 앱을 개발하는 경우 다음 표에서는 대상 .NET SDK 버전에 따라 필요한 최소 Visual Studio 버전을 설명합니다.

테이블 확장

.NET SDK 버전	Visual Studio 버전
10	Visual Studio 2026 버전 18.0 이상.

.NET SDK 버전	Visual Studio 버전
9	Visual Studio 2022 버전 17.12 이상.
8 (여덟)	Visual Studio 2022 버전 17.8 이상.
7	Visual Studio 2022 버전 17.4 이상.
6	Visual Studio 2022 버전 17.0 이상.
5	Visual Studio 2019 버전 16.8 이상.
3.1	Visual Studio 2019 버전 16.4 이상.
3.0	Visual Studio 2019 버전 16.3 이상.
2.2	Visual Studio 2017 버전 15.9 이상.
2.1	Visual Studio 2017 버전 15.7 이상.

이미 Visual Studio 설치되어 있는 경우 다음 단계를 사용하여 버전을 확인할 수 있습니다.

1. Visual Studio 엽니다.
2. **Help**>**About Microsoft Visual Studio**를 선택합니다.
3. **정보** 대화 상자에서 버전 번호를 확인합니다.

자세한 내용은 .NET SDK, MSBuild 및 Visual Studio 버전 관리 참조하세요.

Visual Studio Code 사용하여 설치

Visual Studio Code 데스크톱에서 실행되는 강력하고 간단한 소스 코드 편집기입니다. Visual Studio Code 시스템에 이미 설치된 SDK를 사용할 수 있습니다.

> **ⓘ Important**
>
> 시스템 전체에서 .NET 설치하는 경우 관리자 권한으로 설치합니다.

이 WinGet 구성 파일 최신 .NET SDK, Visual Studio Code 및 C# DevKit을 설치합니다. 이미 설치되어 있는 경우 WinGet은 해당 단계를 건너뜁니다.

1. 파일을 다운로드하고 두 번 클릭하여 실행합니다.
2. 사용권 계약을 읽고, `y` 입력하고, 동의하라는 메시지가 표시되면 Enter 키를 선택합니다.
3. 작업 표시줄에 깜박이는 UAC(사용자 계정 컨트롤) 프롬프트가 표시되면 설치를 계속하도록 허용합니다.

또한 C# Dev Kit 확장은 아직 설치되지 않은 경우 .NET 설치합니다.

Visual Studio Code 통해 .NET 설치하는 방법에 대한 지침은 Getting Started with C# in VS Code 참조하세요.

.NET 설치 관리자

.NET의 download 페이지 는 Windows 설치 관리자 실행 파일을 제공합니다.

> **Important**
>
> 시스템 전체에서 .NET 설치하는 경우 관리자 권한으로 설치합니다.

1. 웹 브라우저를 열고 https://dotnethttps://dotnet.microsoft.com/download/dotnet 이동합니다.
2. 다운로드하려는 .NET 버전(예: 10.0)을 선택합니다.
3. .NET 다운로드하기 위한 링크가 포함된 SDK 또는 런타임 상자를 찾습니다.
4. **인스톨러** 열 아래에서 **Windows** 행을 찾아 CPU 아키텍처에 대한 링크를 선택합니다. 확실하지 않은 경우 가장 일반적인 **x64**를 선택합니다.

   브라우저에서 설치 관리자를 자동으로 다운로드해야 합니다.

   > **팁**
   >
   > 다음 이미지는 SDK를 보여 주지만 런타임을 다운로드할 수도 있습니다.

   Build apps - SDK

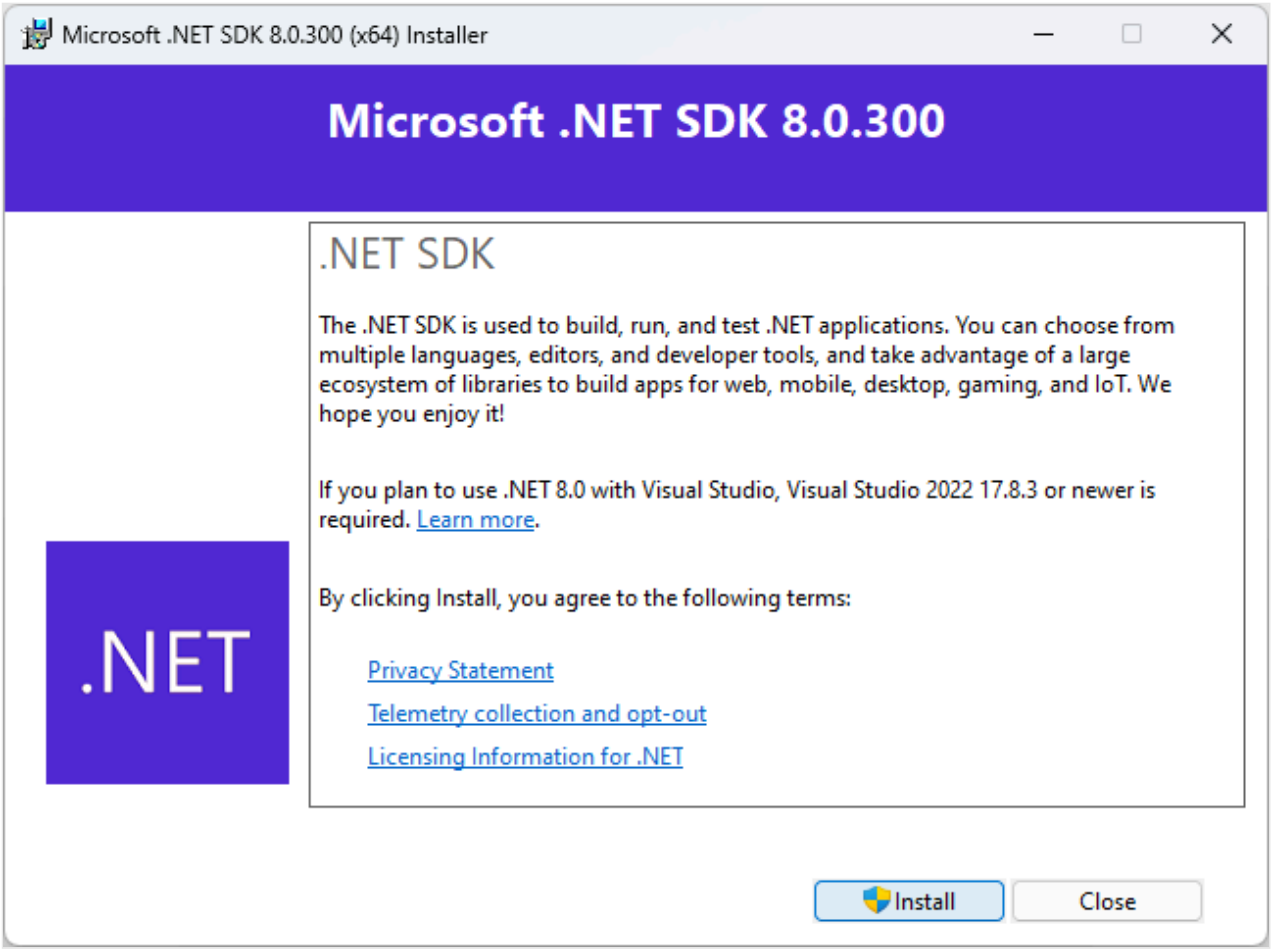
   SDK 8.0.300

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS	Arm64 | x64	Arm64 | x64
Windows	Arm64 | x64 | x86 | winget instructions	Arm64 | x64 | x86
All	dotnet-install scripts	

5. Windows 탐색기를 열고 파일이 다운로드된 위치로 이동합니다. **Downloads** 폴더일 가능성이 큽니다.

6. 파일을 두 번 클릭하여 .NET 설치합니다.

   Windows 설치 관리자 대화 상자가 열립니다.



7. **설치**를 선택하고 지침에 따라 .NET 설치합니다.

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

명령줄 옵션

`/?` 매개 변수를 사용하여 옵션 목록을 표시합니다.

프로덕션 환경과 같이 .NET 자동으로 설치하거나 지속적인 통합을 지원하려면 다음 옵션을 사용합니다.

- `/install`
  .NET 설치합니다.
- `/quiet`
  UI와 프롬프트가 표시되지 않도록 합니다.
- `/norestart`
  재시작을 시도하는 모든 시도를 억제합니다.

콘솔

```
dotnet-sdk-9.0.100-win-x64.exe /install /quiet /norestart
```

.NET 이미 설치한 경우 .NET 설치 관리자를 사용하여 설치를 관리합니다. `/install`대신 다음 옵션 중 하나를 사용합니다.

- `/uninstall`
  이 버전의 .NET 제거합니다.
- `/repair`
  설치 키 파일 또는 구성 요소가 손상되었는지 확인하고 복원합니다.

> **팁**
>
> 설치 관리자는 성공 시 종료 코드 **0**을 반환하고 다시 시작이 필요함을 나타내기 위해 종료 코드 **3010**을 반환합니다. 다른 값은 오류 코드일 가능성이 높습니다.

Microsoft 업데이트

.NET 설치 관리자 실행 파일은 Windows MU(Microsoft Update)를 사용하여 서비스할 수 있는 독립 제품입니다. MU는 .NET Framework와 같은 운영 체제 구성 요소를 서비스하는 데 사용되는 WU(Windows Update)와 다릅니다.

지원되는 .NET 버전에 대한 보안 및 비보안 수정 사항은 여러 배포 채널을 사용하여 MU를 통해 제공됩니다. 자동 업데이트(AU)는 최종 사용자 및 소비자와 관련이 있으며 WSUS(Window Server Update Services) 및 Windows Update 카탈로그는 IT 관리자와 관련이 있습니다.

.NET 설치 관리자 실행 파일은 런타임 및 SDK와 같은 다양한 아키텍처 및 구성 요소에 대한 주 및 부 릴리스에서 SxS(Side-By-Side) 설치를 지원합니다. 예를 들어 6.0.15(x64) 및 6.0.17(x86) 런타임을 모두 설치할 수 있습니다. MU가 트리거되면 두 설치 모두에 대한 최신 설치 관리자를 제공합니다.

업데이트 차단

대부분의 사용자는 최신 상태를 유지하는 것을 선호하지만 다음 표의 레지스트리 키를 사용하여 .NET 업데이트를 차단할 수 있습니다.

테이블 확장

.NET 버전	레지스트리 키	이름	타입	값
모두	HKLM\SOFTWARE\Microsoft\.NET	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 9	HKLM\SOFTWARE\Microsoft\.NET\9.0	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 8	HKLM\SOFTWARE\Microsoft\.NET\8.0	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 7	HKLM\SOFTWARE\Microsoft\.NET\7.0	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 6	HKLM\SOFTWARE\Microsoft\.NET\6.0	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 5	HKLM\SOFTWARE\Microsoft\.NET\5.0	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET Core 3.1	HKLM\SOFTWARE\Microsoft\.NET\3.1	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET Core 2.1	HKLM\SOFTWARE\Microsoft\.NET\2.1	BlockMU	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001

서버 OS에 대한 자동 업데이트

서버 운영 체제에 대한 업데이트는 WSUS 및 Microsoft 업데이트 카탈로그에서 지원되지만 AU는 지원하지 않습니다. 서버 운영 체제는 다음 레지스트리 키를 사용하여 AU를 통해 업데이트를 수신하도록 옵트인할 수 있습니다.

테이블 확장

.NET 버전	레지스트리 키	이름	타입	값
모두	HKLM\SOFTWARE\Microsoft\.NET	서버 운영 체제에서 AU 허용	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 9	HKLM\SOFTWARE\Microsoft\.NET\9.0	서버 운영 체제에서 AU 허용	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 8	HKLM\SOFTWARE\Microsoft\.NET\8.0	서버 운영 체제에서	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001

.NET 버전	레지스트리 키	이름	타입	값
		AU 허용		
.NET 7	HKLM\SOFTWARE\Microsoft\.NET\7.0	서버 운영 체제에서 AU 허용	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 6	HKLM\SOFTWARE\Microsoft\.NET\6.0	서버 운영 체제에서 AU 허용	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET 5	HKLM\SOFTWARE\Microsoft\.NET\5.0	서버 운영 체제에서 AU 허용	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001
.NET Core 3.1	HKLM\SOFTWARE\Microsoft\.NET\3.1	서버 운영 체제에서 AU 허용	프로그램 레지스트리 데이터 타입인 REG_DWORD	0x00000001

WSUS 및 업데이트 분류

WSUS는 분류에 따라 특정 업데이트를 제공하도록 구성할 수 있습니다. .NET의 업데이트는 *보안* 또는 *중요* 중 하나로 분류됩니다. 최신 업데이트가 위험으로 분류되는 경우 최신 보안 업데이트로 대체되는 이전 버전의 .NET 설치될 때 이전 *security* 업데이트가 제공될 수 있습니다. 이는 오프라인 CAB (Wsusscan2.cab) 를 사용하여 컴퓨터를 검사하는 데에도 적용됩니다.

> **참고 항목**
>
> 경우에 따라 WSUS는 설치한 .NET 버전보다 오래된 버전에 대한 누락된 업데이트를 보고할 수 있습니다. 예를 들어 사용자가 .NET 6의 최신 릴리스인 .NET 6.0.36을 설치하는 경우를 상상해 보세요. 이 버전은 중요(비보안) 업데이트로 분류됩니다. 그런 다음 애플리케이션에서 이전 버전인 6.0.33을 설치합니다. (애플리케이션에서 특정 버전의 .NET 필수 구성 요소로 포함하는 것은 드문 일이 아닙니다.) 관리자가 보안 업데이트만 제공하도록 WSUS를 구성한 경우 다음 검사에서 누락된 업데이트로 6.0.35를 보고합니다. AU 또는 WSUS를 통해 *보안* 업데이트를 받도록 구성된 컴퓨터는 6.0.36이 설치된 경우에도 6.0.35가 제공됩니다. 그 이유는 6.0.35가 6.0.33을 대체하고 최신 *보안* 업데이트이기 때문입니다.

이전 버전이 제거되는 시기 선택

설치 관리자 실행 파일은 이전 설치를 제거하기 전에 항상 새 콘텐츠를 설치합니다. 실행 중인 애플리케이션은 이전 런타임이 제거될 때 중단되거나 충돌할 수 있습니다. 업데이트 .NET 영향

을 최소화하려면 레지스트리 키를 사용하여 이전 .NET 설치를 제거할 시기를 지정할 수 있습니다.

테이블 확장

.NET 버전	레지스트리 키	이름	타입	값
모두	HKLM\SOFTWARE\Microsoft\.NET	이전 버전 제거	REG_SZ	`always`, `never` 또는 `nextSession`
.NET 10	HKLM\SOFTWARE\Microsoft\.NET\10.0	이전 버전 제거	REG_SZ	`always`, `never` 또는 `nextSession`
.NET 9	HKLM\SOFTWARE\Microsoft\.NET\9.0	이전 버전 제거	REG_SZ	`always`, `never` 또는 `nextSession`
.NET 8	HKLM\SOFTWARE\Microsoft\.NET\8.0	이전 버전 제거	REG_SZ	`always`, `never` 또는 `nextSession`

- `never` 이전 설치를 유지하고 이전 .NET 설치를 제거하려면 수동 개입이 필요합니다.
- `always` 는 새 버전이 설치된 후 이전 설치를 제거합니다. 이는 .NET 기본 동작입니다.
- `nextSession` 는 Administrators 그룹의 멤버에서 다음 로그온 세션까지 제거를 지연합니다.
- 값은 대/소문자를 구분하지 않으며 잘못된 값은 기본값입니다 `always`.

제거가 지연되면 설치 관리자는 이전 버전을 제거하는 명령을 RunOnce 레지스트리 키에 씁니다. 이 명령은 Administrators 그룹의 사용자가 컴퓨터에 로그온하는 경우에만 실행됩니다.

> **참고 항목**
>
> 이 기능은 .NET 8(8.0.11)부터만 사용할 수 있습니다. 독립 실행형 설치 관리자 실행 파일에만 적용되며 이를 사용하는 WinGet과 같은 배포에 영향을 줍니다.

Windows 패키지 관리자(WinGet) 설치

`winget.exe` 도구를 사용하여 Windows Package Manager 서비스를 통해 .NET 설치하고 관리할 수 있습니다. **WinGet** 설치 및 사용 방법에 대한 자세한 내용은 winget 도구를 사용하여 애플리케이션 설치 및 관리를 참조하세요.

> **Important**
>
> 시스템 전체에서 .NET 설치하는 경우 관리자 권한으로 설치합니다.

.NET WinGet 패키지는 다음과 같습니다.

- `Microsoft.DotNet.Runtime.10` — .NET 런타임 10.0
- `Microsoft.DotNet.AspNetCore.10` — ASP.NET Core 런타임 10.0
- `Microsoft.DotNet.DesktopRuntime.10` — .NET Desktop Runtime 10.0
- `Microsoft.DotNet.SDK.10` — .NET SDK 10.0
- `Microsoft.DotNet.Runtime.9` — .NET 런타임 9.0
- `Microsoft.DotNet.AspNetCore.9` — ASP.NET Core 런타임 9.0
- `Microsoft.DotNet.DesktopRuntime.9` — .NET Desktop Runtime 9.0
- `Microsoft.DotNet.SDK.9` — .NET SDK 9.0
- `Microsoft.DotNet.Runtime.8` — .NET 런타임 8.0
- `Microsoft.DotNet.AspNetCore.8` — ASP.NET Core 런타임 8.0
- `Microsoft.DotNet.DesktopRuntime.8` — .NET Desktop Runtime 8.0
- `Microsoft.DotNet.SDK.8` — .NET SDK 8.0

SDK 설치

SDK를 설치하면 해당 런타임을 설치할 필요가 없습니다.

1. WinGet을 설치합니다.
2. PowerShell 또는 명령 프롬프트와 같은 터미널을 엽니다.
3. `winget install` 명령을 실행하고 SDK 패키지 이름을 전달합니다.

Windows 명령 프롬프트

```
winget install Microsoft.DotNet.SDK.10
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

설치할 수 있는 다양한 런타임이 있습니다. 각 런타임에 포함된 내용을 알아보려면 올바른 런타임 선택 섹션을 참조하세요.

1. WinGet을 설치합니다.
2. **PowerShell** 또는 **명령 프롬프트**와 같은 터미널을 엽니다.
3. `winget install` 명령을 실행하고 SDK 패키지 이름을 전달합니다.

```
```

Windows 명령 프롬프트

```
winget install Microsoft.DotNet.DesktopRuntime.10
winget install Microsoft.DotNet.AspNetCore.10
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

버전 검색

설치하려는 패키지의 다른 버전을 검색하려면 `winget search` 명령을 사용합니다. 예를 들어 다음 명령은 WinGet을 통해 사용할 수 있는 모든 .NET SDK를 검색합니다.

Windows 명령 프롬프트

```
winget search Microsoft.DotNet.SDK
```

검색 결과는 각 패키지 식별자와 함께 테이블에 인쇄됩니다.

출력

```
Name                           Id                            Version
Source
-------------------------------------------------------------------------------
---------
Microsoft .NET SDK 3.1         Microsoft.DotNet.SDK.3_1      3.1.426
winget
Microsoft .NET SDK 5.0         Microsoft.DotNet.SDK.5        5.0.408
winget
Microsoft .NET SDK 6.0         Microsoft.DotNet.SDK.6        6.0.428
winget
Microsoft .NET SDK 7.0         Microsoft.DotNet.SDK.7        7.0.410
winget
Microsoft .NET SDK 8.0         Microsoft.DotNet.SDK.8        8.0.415
winget
Microsoft .NET SDK 9.0         Microsoft.DotNet.SDK.9        9.0.306
winget
Microsoft .NET SDK 10.0        Microsoft.DotNet.SDK.10       10.0.100
winget
```

미리 보기 버전 설치

미리 보기 버전을 사용할 수 있는 경우 ID**의 버전 번호를** 단어`Preview`로 바꿔야 합니다. 다음 예제에서는 .NET 데스크톱 런타임의 미리 보기 릴리스를 설치합니다.

Windows 명령 프롬프트

```
winget install Microsoft.DotNet.DesktopRuntime.Preview
```

PowerShell을 사용하여 설치

연속 통합 및 nonadmin 설치에는 `dotnet-install` PowerShell 스크립트를 통해 .NET 설치하는 것이 좋습니다. 시스템에서 정상적으로 사용하기 위해 .NET 설치하는 경우 .NET Installer 또는 Windows Package Manager 설치 방법을 사용합니다.

스크립트는 기본적으로 .NET 10인 최신 long term support(LTS) 버전을 설치합니다. `-Channel` 스위치를 지정하여 특정 릴리스를 선택할 수 있습니다. 런타임을 설치하려면 `-Runtime` 스위치를 포함합니다. 그렇지 않으면 스크립트가 SDK를 설치합니다. 스크립트는 https://dot.net/v1/dotnet-install.ps1 사용할 수 있으며 소스 코드는 GitHub 호스트됩니다.

스크립트 다운로드

스크립트에 대한 자세한 내용은 -install 스크립트 참조를 참조하세요.

런타임 설치

.NET 런타임은 `-Runtime` 스위치를 제공하여 설치됩니다.

1. https://dot.net/v1/dotnet-install.ps1 에서 설치 스크립트를 다운로드합니다.
2. PowerShell을 열고 스크립트가 포함된 폴더로 이동합니다.
3. 다음 명령을 실행하여 최대 호환성을 위해 데스크톱 런타임과 ASP.NET Core 런타임을 모두 설치합니다.

   PowerShell

   ```
   dotnet-install.ps1 -Runtime windowsdesktop
   dotnet-install.ps1 -Runtime aspnetcore
   ```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

SDK 설치

SDK를 설치하는 경우 런타임을 설치할 필요가 없습니다.

1. https://dot.net/v1/dotnet-install.ps1 에서 설치 스크립트를 다운로드합니다.
2. PowerShell을 열고 스크립트가 포함된 폴더로 이동합니다.

3. 다음 명령을 실행하여 .NET SDK를 설치합니다.

PowerShell

```
dotnet-install.ps1
```

> **ⓘ 참고 항목**
>
> SDK는 `-Runtime` 스위치를 생략하여 설치됩니다.

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

유효성 검사

설치 프로그램 또는 이진 릴리스를 다운로드한 후 파일이 변경되거나 손상되지 않았는지 확인합니다. 컴퓨터에서 체크섬을 확인한 다음 다운로드 웹 사이트에 보고된 내용과 비교할 수 있습니다.

공식 다운로드 페이지에서 파일을 다운로드하면 해당 파일의 체크섬이 텍스트 상자에 표시됩니다. 체크섬 값을 클립보드에 복사하려면 **복사** 단추를 선택합니다.

PowerShell 또는 **명령 프롬프트**를 사용하여 다운로드한 파일의 체크섬의 유효성을 검사할 수 있습니다. 예를 들어 다음 명령은-sdk-9.0.306-win-x64.exe파일의 체크섬을 *dotnet* 보고합니다.

Windows 명령 프롬프트

```
> certutil -hashfile dotnet-sdk-9.0.306-win-x64.exe SHA512
SHA512 hash of dotnet-sdk-9.0.306-win-x64.exe:
f048ddf80c0aa88e713070e66a0009435ad9a5f444adbde6edf2b17f8da562d494a5c37cbabaf63ee31
```

```
25fe1d2da735a397de9a38dd6ca638b8dc085adc01d4f
CertUtil: -hashfile command completed successfully.
```

PowerShell

```
> (Get-FileHash .\dotnet-sdk-9.0.306-win-x64.exe -Algorithm SHA512).Hash
f048ddf80c0aa88e713070e66a0009435ad9a5f444adbde6edf2b17f8da562d494a5c37cbabaf63ee31
25fe1d2da735a397de9a38dd6ca638b8dc085adc01d4f
```

체크섬을 다운로드 사이트에서 제공한 값과 비교합니다.

PowerShell 및 체크섬 파일을 사용하여 유효성을 검사합니다.

.NET 릴리스 정보에는 다운로드한 파일의 유효성을 검사하는 데 사용할 수 있는 체크섬 파일에 대한 링크가 포함되어 있습니다. 다음 단계에서는 체크섬 파일을 다운로드하고 .NET 설치 이진 파일의 유효성을 검사하는 방법을 설명합니다.

1. https://github.com/dotnet/core/tree/main/release-notes/9.0 GitHub .NET 9의 릴리스 정보 페이지에는 **Releases** 섹션이 포함되어 있습니다. 해당 섹션의 표는 각 .NET 9 릴리스의 다운로드 및 체크섬 파일에 연결됩니다. 다음 이미지는 .NET 8 릴리스 테이블을 참조로 보여줍니다.

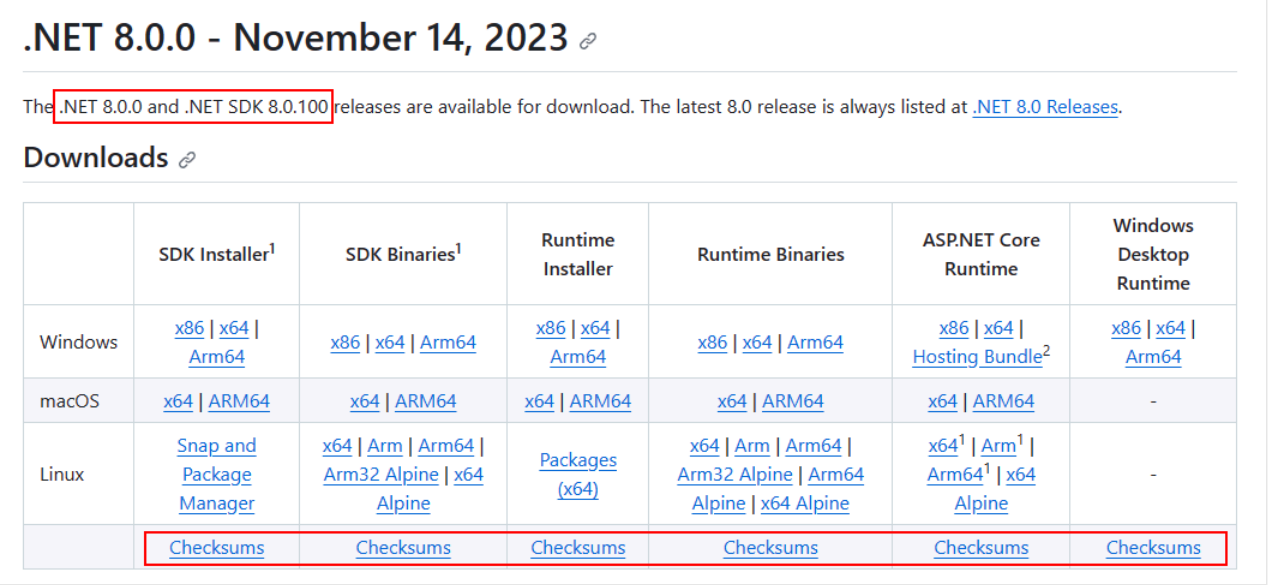
Releases

Date	Release
2023/11/14	8.0.0
2023/10/10	8.0.0 RC 2
2023/09/12	8.0.0 RC 1
2023/08/08	8.0.0 Preview 7
2023/07/11	8.0.0 Preview 6

2. 다운로드한 .NET 버전의 링크를 선택합니다. 이전 섹션에서는 .NET 9.0.10 릴리스의 .NET SDK 9.0.306을 사용했습니다.

> **팁**
>
> 체크섬 파일이 포함된 .NET 릴리스를 잘 모르는 경우 찾을 때까지 링크를 탐색합니다.

3. 릴리스 페이지에서 .NET 런타임 및 .NET SDK 버전과 체크섬 파일에 대한 링크를 볼 수 있습니다. 다음 이미지는 .NET 8 릴리스 테이블을 참조로 보여줍니다.



.NET 8.0.0 - November 14, 2023

The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at .NET 8.0 Releases.

Downloads

	SDK Installer[1]	SDK Binaries[1]	Runtime Installer	Runtime Binaries	ASP.NET Core Runtime	Windows Desktop Runtime
Windows	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Hosting Bundle[2]	x86 | x64 | Arm64
macOS	x64 | ARM64	x64 | ARM64	x64 | ARM64	x64 | ARM64	x64 | ARM64	-
Linux	Snap and Package Manager	x64 | Arm | Arm64 | Arm32 Alpine | x64 Alpine	Packages (x64)	x64 | Arm | Arm64 | Arm32 Alpine | Arm64 Alpine | x64 Alpine	x64[1] | Arm[1] | Arm64[1] | x64 Alpine	-
	Checksums	Checksums	Checksums	Checksums	Checksums	Checksums

4. 체크섬 파일에 대한 링크를 복사합니다.

5. 다음 스크립트를 사용하되 링크를 바꿔 적절한 체크섬 파일을 다운로드합니다.

```PowerShell
Invoke-WebRequest https://builds.dotnet.microsoft.com/dotnet/checksums/9.0.10-sha.txt -OutFile 9.0.10-sha.txt
```

6. 체크섬 파일과 .NET 릴리스 파일을 모두 동일한 디렉터리에 다운로드한 상태에서 체크섬 파일에서 .NET 다운로드의 체크섬을 검색합니다.

   유효성 검사를 통과하면 **True**가 인쇄된 것으로 표시됩니다.

```PowerShell
> (Get-Content .\9.0.10-sha.txt | Select-String "dotnet-sdk-9.0.306-win-x64.exe").Line -like (Get-FileHash .\dotnet-sdk-9.0.306-win-x64.exe -Algorithm SHA512).Hash + "*"
True
```

   False가 인쇄되면 다운로드한 파일이 유효하지 않은 것이므로 사용해서는 안 됩니다.

문제 해결

.NET SDK를 설치한 후 .NET CLI 명령을 실행하는 데 문제가 발생할 수 있습니다. 이 섹션에서는 이러한 일반적인 문제를 수집하고 솔루션을 제공합니다.

- .NET SDK를 찾을 수 없음
- 앱 빌드가 예상보다 느림
- hostfxr.dll / api-ms-win-crt-runtime-l1-1-0.dll / api-ms-win-cor-timezone-l1-1-0.dll이 누락됨

.NET SDK를 찾을 수 없습니다.

대부분의 경우 .NET SDK의 x86(32비트) 및 x64(64비트) 버전을 모두 설치했습니다. `dotnet` 명령을 실행하면 x64 버전으로 확인되어야 하는데 x86 버전으로 확인되기 때문에 이로 인해 충돌이 발생합니다. 이 문제는 x64 버전을 먼저 해결하기 위해 `%PATH%` 변수를 조정하여 해결됩니다.

1. `where.exe dotnet` 명령을 실행하여 두 버전이 모두 설치되어 있는지 확인합니다. 이 경우 *Program Files* 및 *Program Files(x86)* 폴더 모두에 대한 항목이 표시됩니다. 다음 예제에서 설명한 대로 *Program Files(x86)* 폴더가 먼저 있으면 올바르지 않으며 다음 단계로 계속 진행해야 합니다.

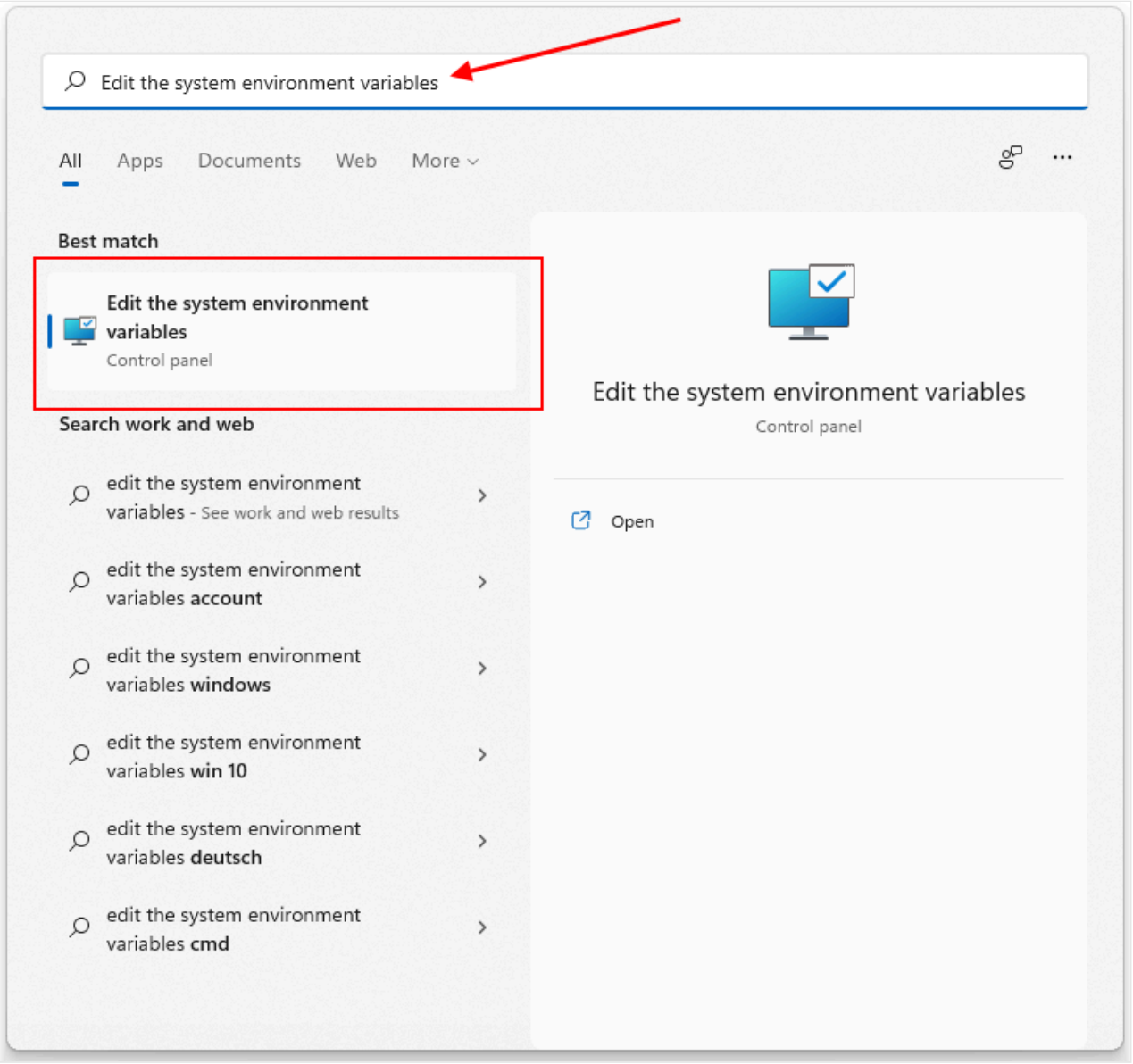
   Windows 명령 프롬프트

   ```
   > where.exe dotnet
   C:\Program Files (x86)\dotnet\dotnet.exe
   C:\Program Files\dotnet\dotnet.exe
   ```

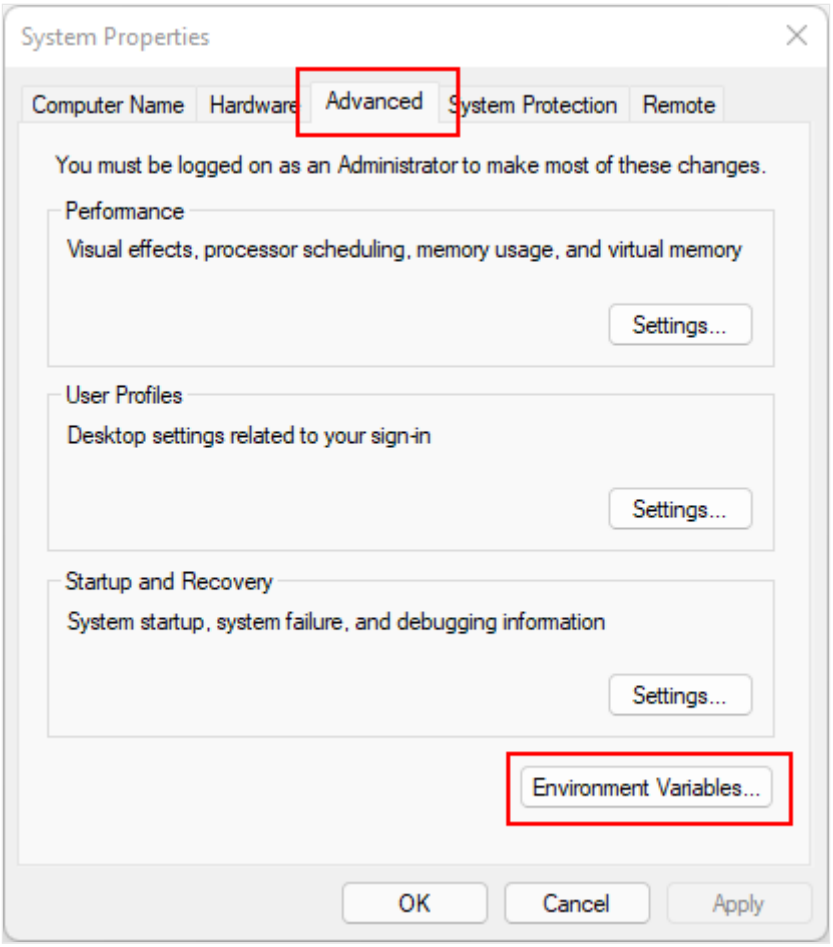
   > **팁**
   >
   > 이 예제에서는 *Program Files* 사용되지만 다른 *dotnet.exe* 복사본이 나열될 수 있습니다. 적절한 *dotnet.exe*이 먼저 해결되도록 조정합니다.

   기본값이 맞고 *Program Files*가 첫 번째로 설정되어 있으며, 이 섹션에서 논의하는 문제가 없는 경우, GitHub에서 .NET 도움 요청 문제 를 생성해야 합니다.

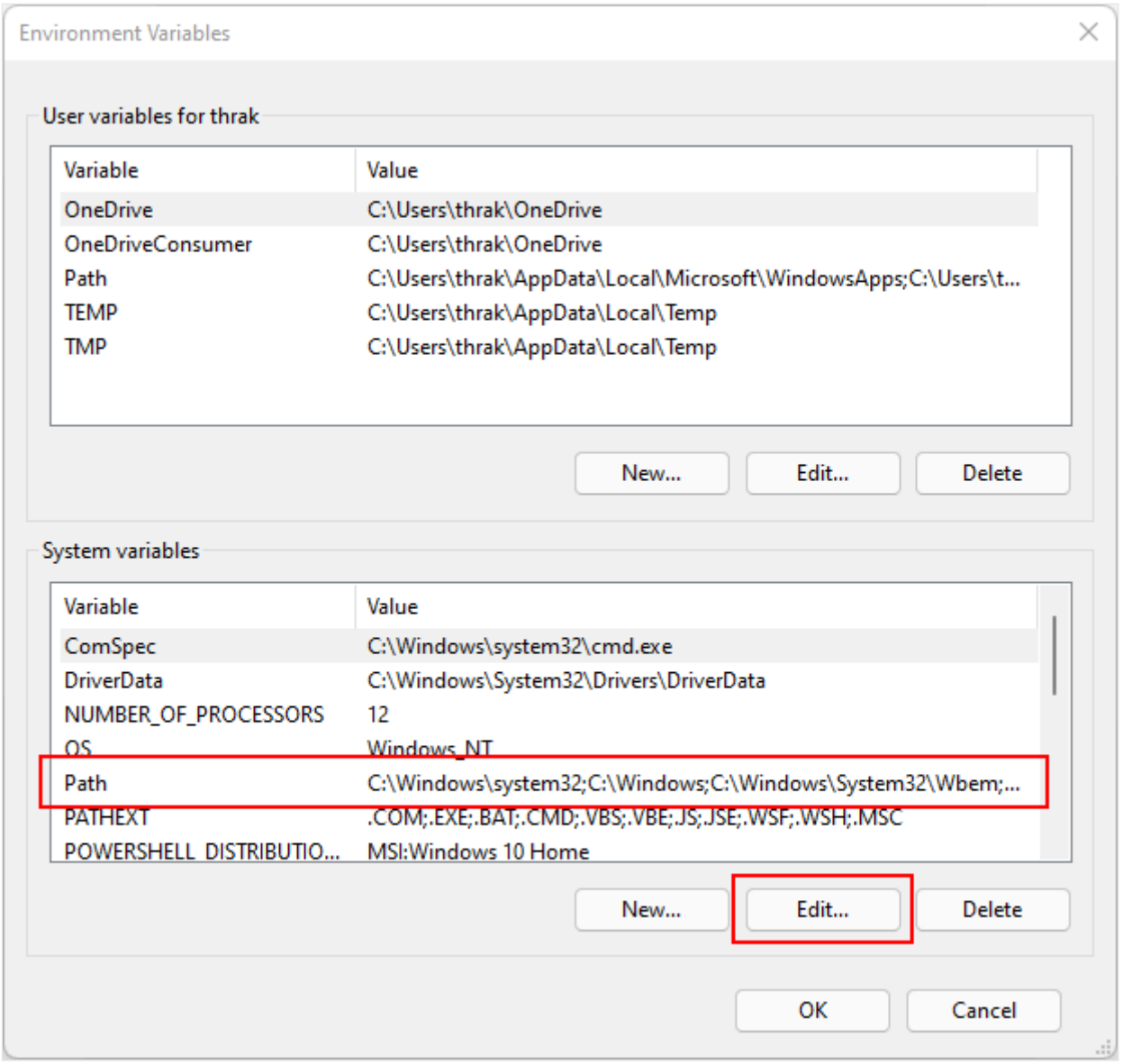
2. Windows 단추를 누르고 검색에 "시스템 환경 변수 편집"을 입력합니다. **시스템 환경 변수 편집**을 선택합니다.



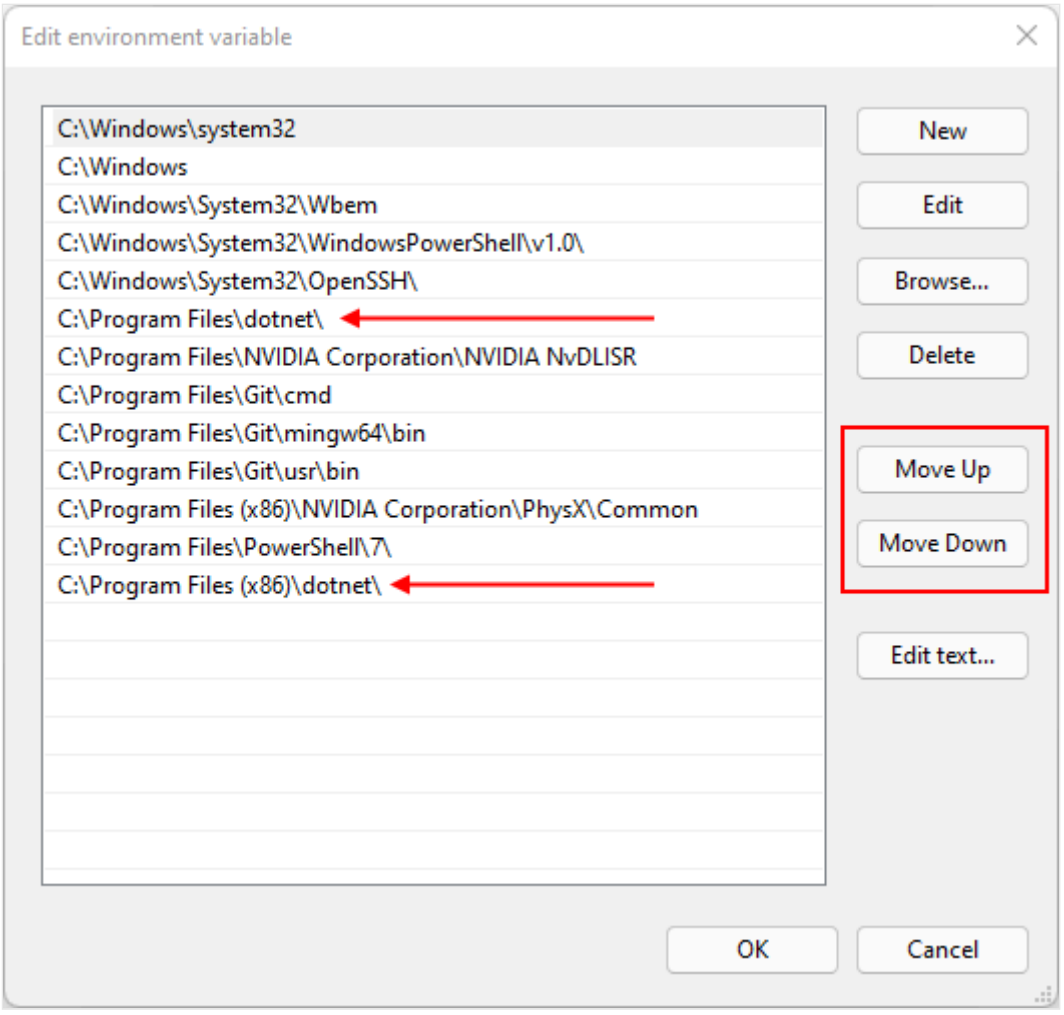
3. **시스템 속성** 창이 열리고 **고급 탭**이 나타납니다. **환경 변수**를 선택합니다.



4. **환경 변수** 창의 **시스템 변수** 그룹 아래에서 *경로** 행을 선택한 다음 **편집** 단추를 선택합니다.



5. **위로 이동아래로 이동** 단추를 사용하여 **C:\Program Files\dotnet** 항목을 **C:\Program Files(x86)\dotnet**위로 이동합니다.



앱 빌드가 예상보다 느림

Windows 기능인 스마트 앱 컨트롤이 꺼져 있는지 확인합니다. 개발에 사용되는 컴퓨터에서는 스마트 앱 제어를 사용하도록 설정하지 않는 것이 좋습니다. "off" 이외의 설정은 SDK 성능에 부정적인 영향을 미칠 수 있습니다.

`hostfxr.dll` / `api-ms-win-crt-runtime-l1-1-0.dll` / `api-ms-win-cor-timezone-l1-1-0.dll`가 누락됨

Microsoft Visual C++ 2015-2019 재배포 가능 패키지(64비트 또는 32비트)를 설치합니다.

관련 콘텐츠

- .NET CLI 개요
- 새 .NET 버전으로 업그레이드.
- .NET 이미 설치되어 있는지 확인하는 방법.
- 자습서: 새 콘솔 애플리케이션을 만듭니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 06.

macOS에 .NET 설치

이 문서에서는 macOS에서 지원되는 .NET 버전, .NET 설치하는 방법 및 SDK와 런타임 간의 차이점을 설명합니다.

.NET 최신 버전은 10입니다.

다운로드 .NET

지원되는 버전

다음 표에서는 지원되는 .NET 릴리스와 지원되는 macOS를 나열합니다. 이러한 버전은 .NET 버전이 지원 종료 또는 macOS 버전이 더 이상 지원되지 않을 때까지 계속 지원됩니다.

테이블 확장

macOS 버전	.NET
macOS 26 "Tahoe"	10.0, 9.0, 8.0
macOS 15 "Sequoia"	10.0, 9.0, 8.0
macOS 14 "Sonoma"	9.0, 8.0

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

런타임 또는 SDK

runtime은 .NET 사용하여 만든 앱을 실행하는 데 사용됩니다. 앱 작성자는 앱을 게시할 때 앱과 함께 런타임을 포함할 수 있습니다. 앱 작성자가 런타임을 포함하지 않는 경우, 사용자가 올바른 런타임을 설치할 수 있습니다.

macOS에 설치할 수 있는 두 개의 런타임이 있으며 둘 다 SDK에 포함되어 있습니다.

- *ASP.NET Core 런타임*
  ASP.NET Core 앱을 실행합니다. .NET 런타임을 포함합니다. **설치 관리자로 사용할 수 없습니다.**

- *.NET 런타임*
  이는 일반 .NET 앱을 실행하지만 ASP.NET Core 기반 앱과 같은 특수 앱은 실행하지 않습니다.

SDK는 .NET 앱 및 라이브러리를 빌드하고 게시하는 데 사용됩니다. 최신 SDK는 이전 버전의 .NET 대한 앱 빌드를 지원합니다. 일반적인 상황에서는 최신 SDK만 설치하면 됩니다.

SDK 설치에는 표준 .NET 런타임과 ASP.NET Core 런타임이 모두 포함됩니다. 예를 들어 .NET SDK 9.0이 설치되어 있는 경우 .NET 런타임 9.0 및 ASP.NET Core 9.0 런타임이 모두 설치됩니다. 그러나 다른 런타임 버전은 SDK와 함께 설치되지 않으며 별도로 설치해야 합니다.

.NET 설치하는 방법 선택

.NET 설치하는 방법에는 여러 가지가 있으며 일부 제품은 자체 버전의 .NET 관리할 수 있습니다. 자체 버전의 .NET 관리하는 소프트웨어를 통해 .NET 설치하는 경우 시스템 전체에서 사용하도록 설정되지 않을 수 있습니다. 다른 소프트웨어를 통해 .NET 설치하는 것이 미치는 영향을 이해해야 합니다.

다음 섹션의 목록을 검토한 후 어떤 방법을 선택해야 할지 잘 모르는 경우 .NET Installer 패키지 사용하려고 할 것입니다.

개발자

- Visual Studio Code - C# 개발 키트

  Visual Studio Code **C# Dev Kit** 확장을 설치하여 .NET 앱을 개발합니다. 확장은 이미 설치된 SDK를 사용하거나 새 SDK를 설치할 수 있습니다.

사용자 및 개발자

- .NET Installer

  독립 실행형 설치 관리자를 사용하여 .NET 설치합니다. 이 방법은 개발자 또는 사용자 컴퓨터에 .NET 설치하는 일반적인 방법입니다.

- 스크립트를 사용하여 .NET 설치

  SDK 또는 런타임의 설치를 자동화할 수 있는 bash 스크립트입니다. 설치할 .NET 버전을 선택할 수 있습니다.

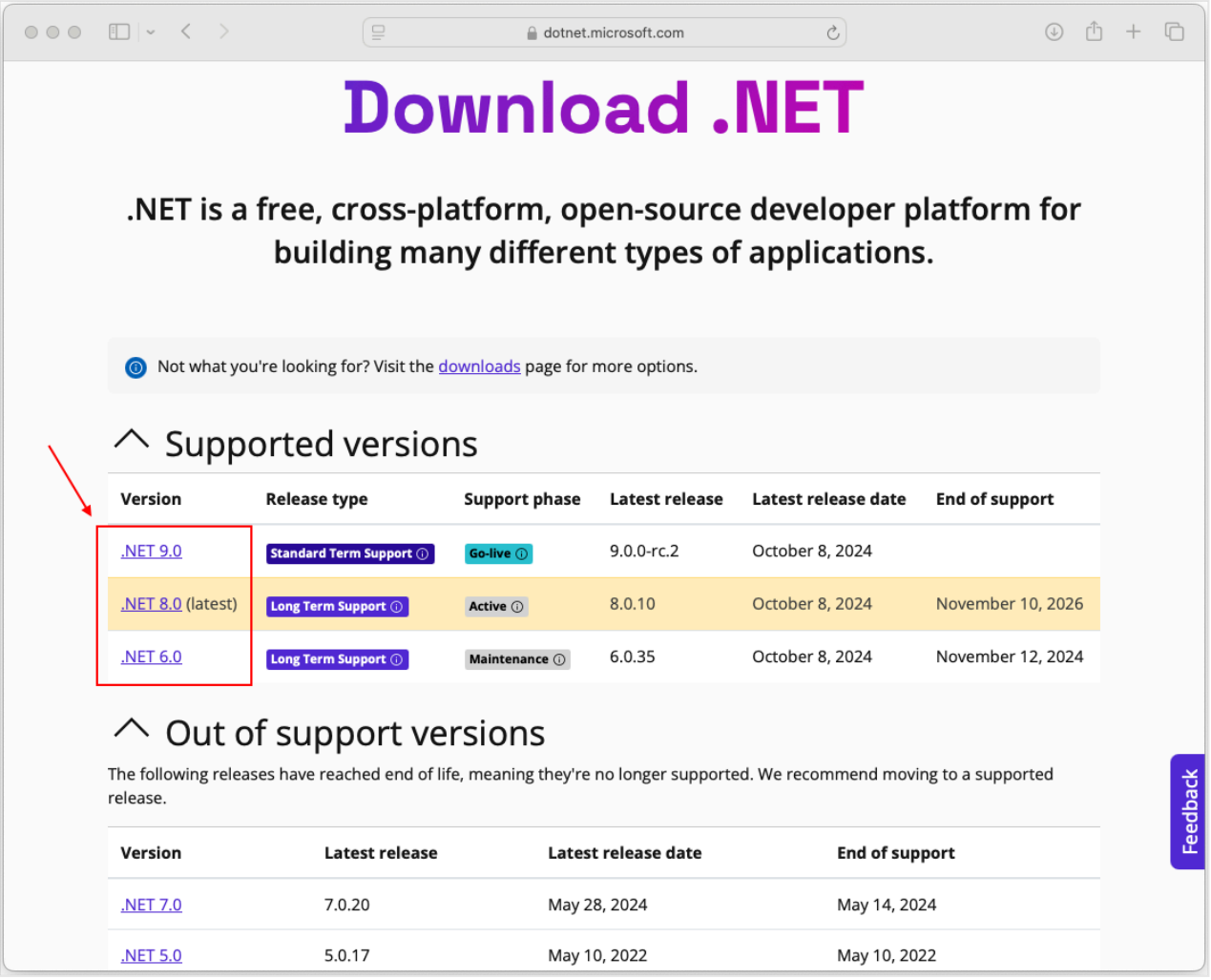
- 수동으로 .NET 설치

  특정 폴더에 .NET 설치하고 다른 .NET 복사본과 별도로 실행해야 하는 경우 이 설치 방법을 사용합니다.

설치 .NET

설치 관리자 패키지는 .NET 설치하는 쉬운 방법인 macOS에서 사용할 수 있습니다.

1. 브라우저를 열고 https://dotnet.microsoft.com/download/dotnet 로 이동합니다.
2. 설치하려는 .NET 버전(예: **.NET 10.0** 링크를 선택합니다.



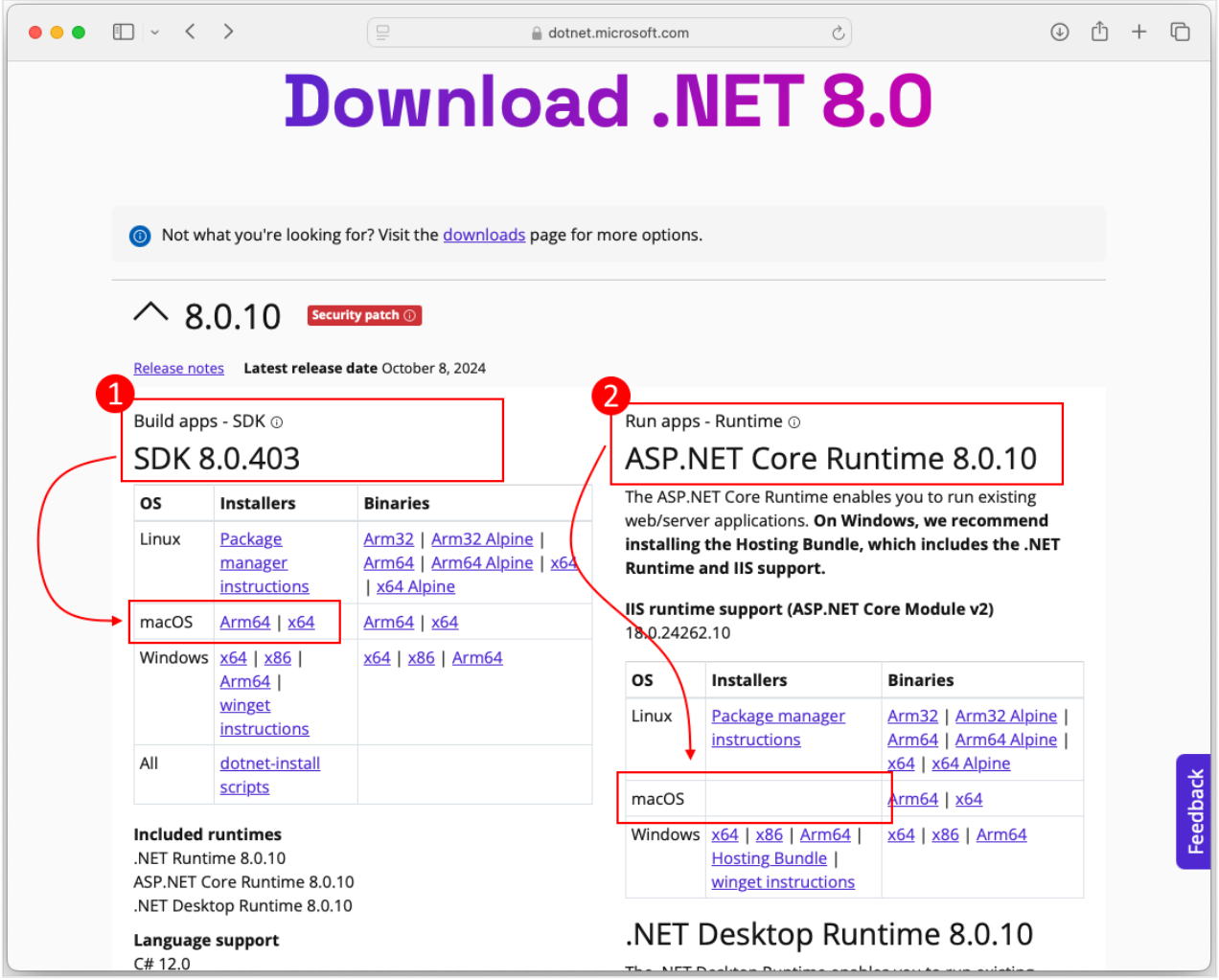
이 링크를 사용하면 해당 버전의 .NET 다운로드할 수 있는 링크가 있는 페이지로 이동됩니다.

SDK를 설치하려는 경우 최신 .NET 버전을 선택합니다. SDK는 이전 버전의 .NET 대한 앱 빌드를 지원합니다.

> **팁**

다운로드할 버전을 잘 모르는 경우 **최신** 버전으로 표시된 버전을 선택합니다.

3. 이 페이지에서는 SDK 및 런타임에 대한 다운로드 링크를 제공합니다. 여기서는 .NET SDK 또는 .NET 런타임을 다운로드합니다.



이전 이미지에는 두 개의 섹션이 강조 표시되어 있습니다. SDK를 다운로드하는 경우 섹션 1을 참조하세요. .NET 런타임의 경우 섹션 2를 참조하세요.

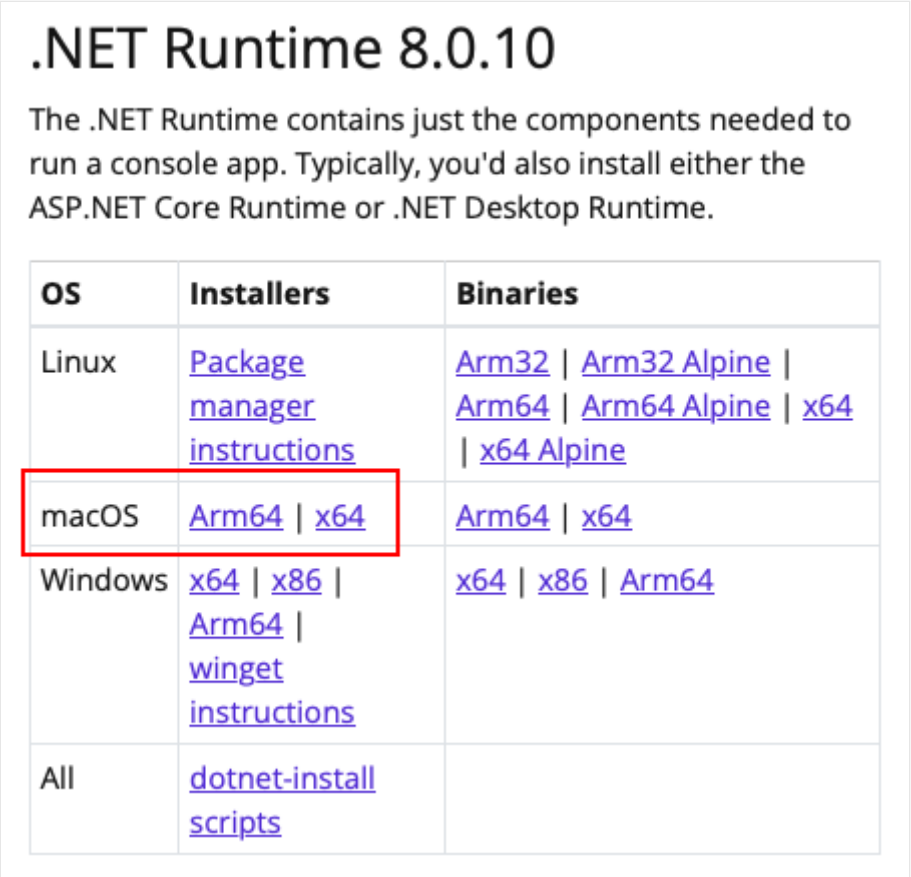
- 섹션 1(SDK)

  이 섹션은 SDK 다운로드 영역입니다. **macOS** 행의 **설치 관리자** 열 아래에는 **Arm64** 및 **x64** 두 아키텍처가 나열됩니다.
  - **M1** 또는 **M3 Pro**와 같은 Apple 프로세서를 실행하는 경우 **Arm64**를 선택합니다.
  - Intel 프로세서를 실행하는 경우 **x64**를 선택합니다.

- 섹션 2 - 실행 시간

  이 섹션에는 런타임 다운로드가 포함되어 있습니다. **macOS** 행의 **설치 관리자** 열에 대한 링크가 비어 있습니다! 이 섹션은 **ASP.NET Core 런타임** SDK 또는 binary 설치 통해서만 제공되므로 비어 있습니다.

  표준 **.NET 런타임**을 다운로드하려면 더 아래로 스크롤하십시오.



.NET Runtime 8.0.10

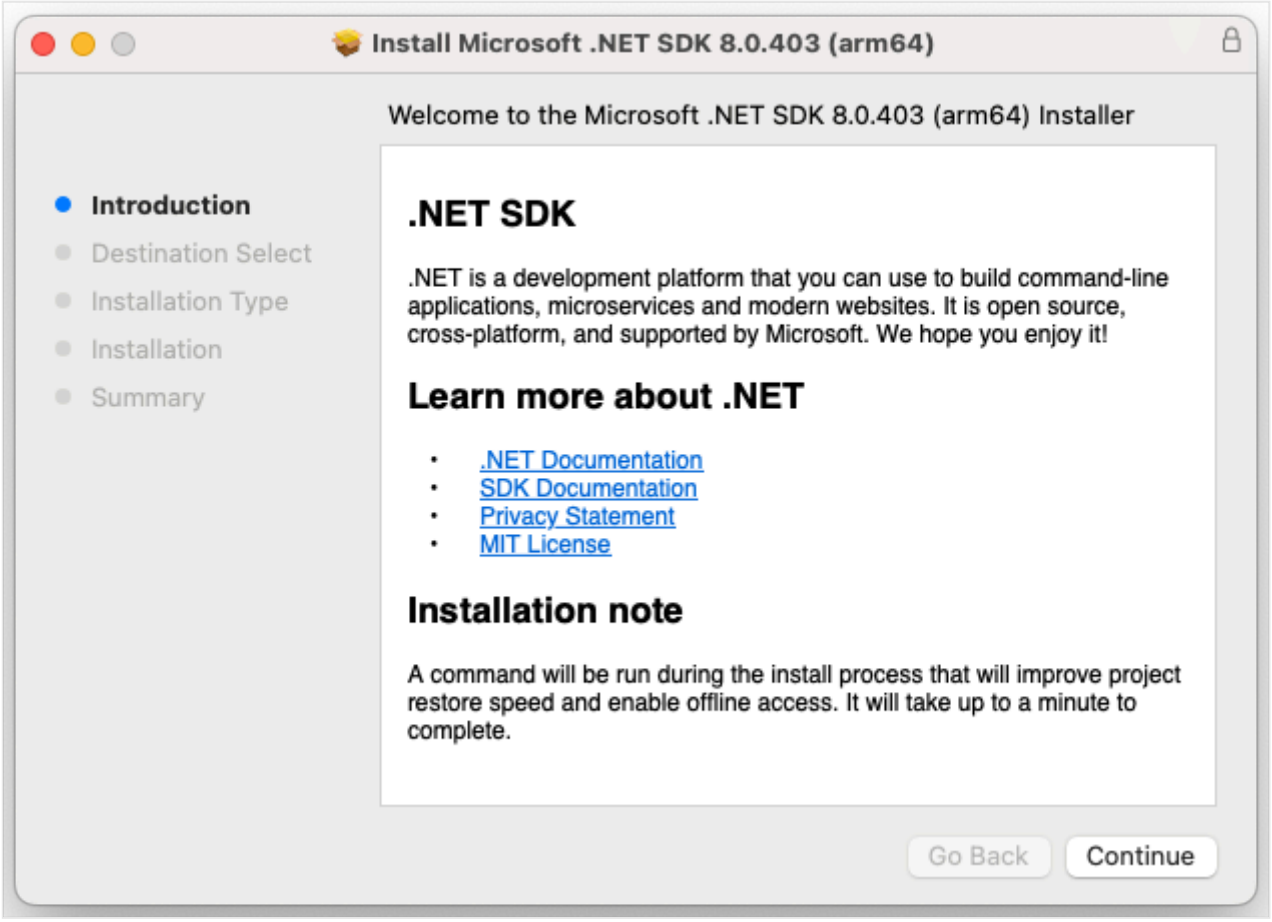
The .NET Runtime contains just the components needed to run a console app. Typically, you'd also install either the ASP.NET Core Runtime or .NET Desktop Runtime.

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS	Arm64 | x64	Arm64 | x64
Windows	x64 | x86 | Arm64 | winget instructions	x64 | x86 | Arm64
All	dotnet-install scripts	

- **M1** 또는 **M3 Pro**와 같은 Apple 프로세서를 실행하는 경우 **Arm64**를 선택합니다.
- Intel 프로세서를 실행하는 경우 **x64**를 선택합니다.

4. 다운로드가 완료되면 엽니다.

5. 설치 관리자의 단계를 따릅니다.



수동으로 .NET 설치

macOS 설치 관리자 대신 SDK와 런타임을 다운로드하여 수동으로 설치할 수 있습니다. 수동 설치는 일반적으로 연속 통합 시나리오에서 일부 자동화로 수행됩니다. 일반적으로 개발자와 사용자는 설치 관리자를 사용해야 합니다.

> **팁**
>
> install-dotnet.sh 스크립트를 사용하여 이러한 단계를 자동으로 수행합니다.

1. 브라우저를 열고 https://dotnet.microsoft.com/download/dotnet 로 이동합니다.
2. 설치하려는 .NET 버전(예: **.NET 8.0** 링크를 선택합니다.

   이 링크를 사용하면 해당 버전의 .NET 다운로드할 수 있는 링크가 있는 페이지로 이동됩니다.

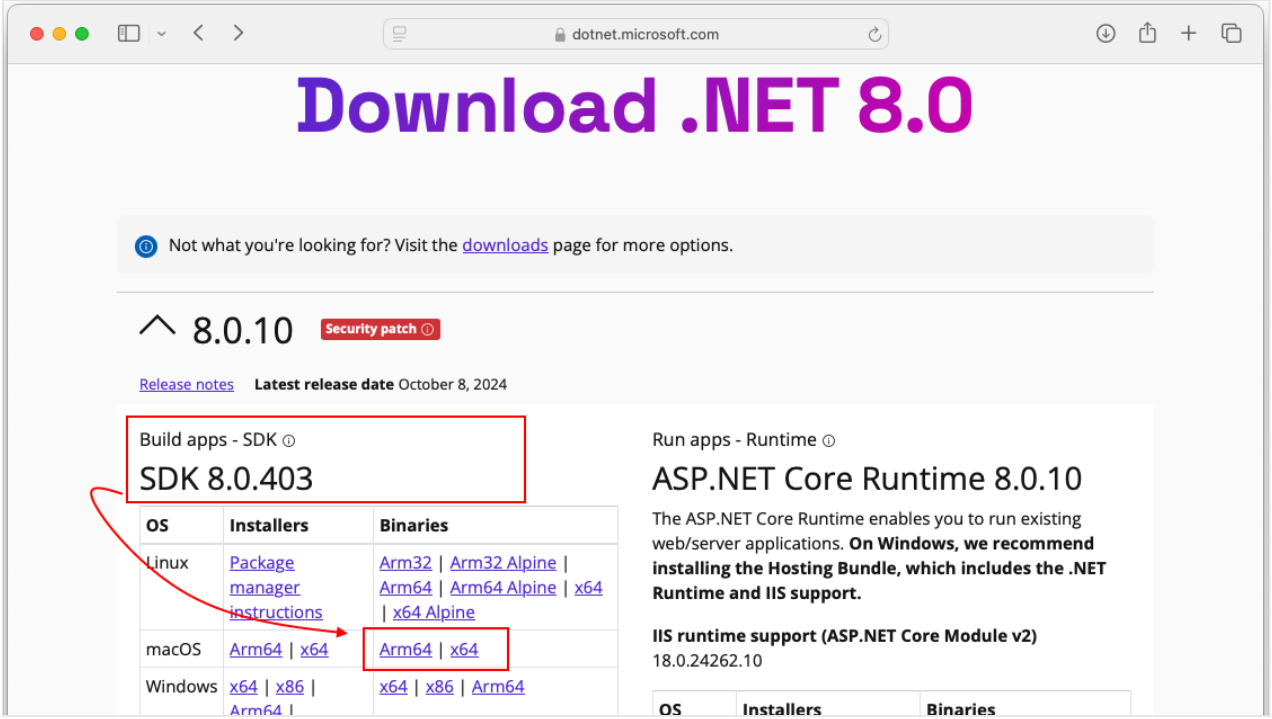
   SDK를 설치하려는 경우 최신 .NET 버전을 선택합니다. SDK는 이전 버전의 .NET 대한 앱 빌드를 지원합니다.

   > **팁**

다운로드할 버전을 잘 모르는 경우 **최신** 버전으로 표시된 버전을 선택합니다.

.NET 다운로드 웹 사이트입니다. 버전 6.0부터 9.0까지 나열됩니다. 빨간색 상자는 해당 다운로드 링크를 강조 표시합니다.

3. 설치하려는 SDK 또는 런타임에 대한 링크를 선택합니다. **Binaries** 열을 **macOS** 행에서 찾습니다.



   - **M1** 또는 **M3 Pro**와 같은 Apple 프로세서를 실행하는 경우 **Arm64**를 선택합니다.
   - Intel 프로세서를 실행하는 경우 **x64**를 선택합니다.

4. 터미널을 열고 .NET 이진 파일이 다운로드된 위치로 이동합니다.

5. 시스템에서 원하는 위치에 tarball을 추출하여 .NET을 설치합니다. 다음 예제에서는 **HOME** 디렉터리 `~/Applications/.dotnet`을 사용합니다.

   Bash

   ```
   mkdir -p ~/Applications/.dotnet
   tar -xf "dotnet-sdk-9.0.100-rc.2.24474.11-osx-arm64.tar" -C
   ~/Applications/.dotnet/
   ```

디렉터리를 .NET 설치된 위치로 변경하여 .NET 작동하는지 테스트하고 `dotnet --info` 명령을 실행합니다.

Bash

```
chdir ~/Applications/.dotnet/
```

```
./dotnet --info
```

스크립트를 사용하여 .NET 설치

dotnet-install 스크립트는 자동화 및 런타임의 관리자 권한이 없는 설치를 수행하는 데 사용됩니다. https://dot.net/v1/dotnet-install.sh에서 스크립트를 다운로드할 수 있습니다.

스크립트는 기본적으로 .NET 8인 최신 long term support(LTS) 버전을 설치합니다. `channel` 스위치를 지정하여 특정 릴리스를 선택할 수 있습니다. 런타임을 설치하려면 `runtime` 스위치를 포함합니다. 그렇지 않으면 스크립트가 SDK를 설치합니다.

> **팁**
>
> 이러한 명령은 이 절차의 마지막에 스크립트 코드 조각으로 제공됩니다.

1. 터미널을 엽니다.
2. *~/Downloads*와 같이 스크립트를 다운로드하려는 폴더로 이동합니다.
3. `wget` 명령이 없는 경우 **Brew**를 사용하여 설치합니다.

   Bash

   ```
   brew install wget
   ```

4. 스크립트를 다운로드하려면 다음 명령을 실행합니다.

   Bash

   ```
   wget https://dot.net/v1/dotnet-install.sh
   ```

5. 스크립트에 실행 권한 부여

   Bash

   ```
   chmod +x dotnet-install.sh
   ```

6. 스크립트를 실행하여 .NET 설치합니다.

   스크립트는 기본적으로 `~/.dotnet` 디렉터리에 최신 SDK를 설치합니다.

   Bash

```
./dotnet-install.sh
```

다음의 모든 명령은 단일 bash 스크립트입니다.

Bash

```
chdir ~/Downloads
brew install wget
wget https://dot.net/v1/dotnet-install.sh
chmod +x dotnet-install.sh
./dotnet-install.sh
```

`~/.dotnet` 폴더로 이동하고 `dotnet --info` 명령을 실행하여 .NET 테스트합니다.

Bash

```
chdir ~/.dotnet
./dotnet --info
```

> ⓘ **Important**
>
> 일부 프로그램은 환경 변수를 사용하여 시스템에서 .NET 찾을 수 있으며 `dotnet` 명령을 사용하면 새 터미널을 열 때 작동하지 않을 수 있습니다. 이 문제를 해결하기 위해 **시스템 전역에서 .NET을 사용 가능하게 설정** 섹션을 참조하세요.

Visual Studio Code용 .NET 설치

Visual Studio Code 데스크톱에서 실행되는 강력하고 간단한 소스 코드 편집기입니다. Visual Studio Code 시스템에 이미 설치된 SDK를 사용할 수 있습니다. 또한 C# Dev Kit 확장은 아직 설치되지 않은 경우 .NET 설치합니다.

Visual Studio Code 통해 .NET 설치하는 방법에 대한 지침은 Getting Started with C# in VS Code 참조하세요.

공증

개발자 ID로 배포되는 macOS용으로 만든 소프트웨어는 .NET 사용하여 만든 앱을 포함하여 공증되어야 합니다.

공증되지 않은 앱을 실행하면 다음 이미지와 비슷한 오류 창이 표시됩니다.

강제 표시가 .NET(및 .NET 앱)에 미치는 영향에 대한 자세한 내용은 macOS Catalina Notarization 작업 참조하세요.

유효성 검사

설치 프로그램 또는 이진 릴리스를 다운로드한 후 파일이 변경되거나 손상되지 않았는지 확인합니다. 컴퓨터에서 체크섬을 확인한 다음 다운로드 웹 사이트에 보고된 내용과 비교할 수 있습니다.

공식 다운로드 페이지에서 파일을 다운로드하면 해당 파일의 체크섬이 텍스트 상자에 표시됩니다. 체크섬 값을 클립보드에 복사하려면 **복사** 단추를 선택합니다.

`shasum -a 512` 명령을 사용하여 다운로드한 파일의 체크섬을 인쇄합니다. 예를 들어 다음 명령은 *dotnet-sdk-9.0.306-osx-x64.tar.gz* 파일의 체크섬을 보고합니다.

Bash

```
$ shasum -a 512 dotnet-sdk-9.0.306-osx-x64.tar.gz
a9700f98e5aa4f70b2a08ddba2b1c6085106b0d17828bd719fdcef460b06c890b32d752fbff8e4659cd
1ca4174b4b211b301fe682439ea9a24b6521ca5a64c69  dotnet-sdk-9.0.306-osx-x64.tar.gz
```

체크섬을 다운로드 사이트에서 제공한 값과 비교합니다.

체크섬 파일을 사용하여 유효성 검사

.NET 릴리스 정보에는 다운로드한 파일의 유효성을 검사하는 데 사용할 수 있는 체크섬 파일에 대한 링크가 포함되어 있습니다. 다음 단계에서는 체크섬 파일을 다운로드하고 .NET 설치 이진 파일의 유효성을 검사하는 방법을 설명합니다.

1. https://github.com/dotnet/core/tree/main/release-notes/9.0#releases GitHub .NET 9의 릴리스 정보 페이지에는 **Releases** 섹션이 포함되어 있습니다. 해당 섹션의 표는 각 .NET 9 릴리스의 다운로드 및 체크섬 파일에 연결됩니다. 다음 이미지는 .NET 8 릴리스 테이블을 참조로 보여줍니다.

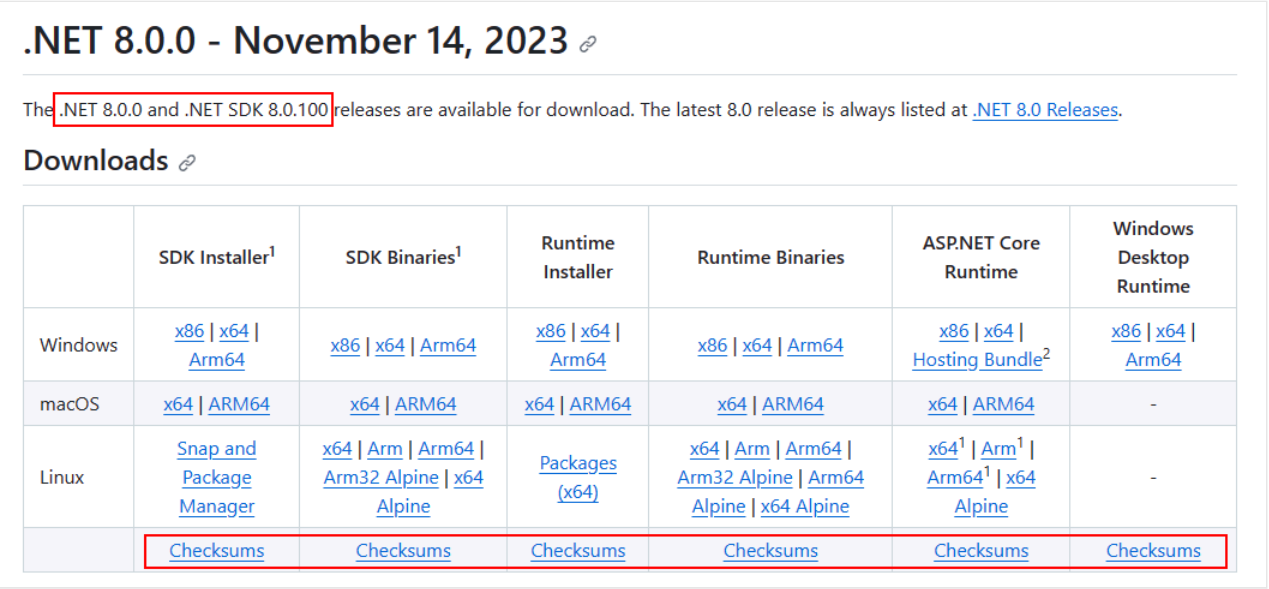
Releases

Date	Release
2023/11/14	8.0.0
2023/10/10	8.0.0 RC 2
2023/09/12	8.0.0 RC 1
2023/08/08	8.0.0 Preview 7
2023/07/11	8.0.0 Preview 6

2. 다운로드한 .NET 버전의 링크를 선택합니다.

   이전 섹션에서는 .NET 9.0.10 릴리스의 .NET SDK 9.0.306을 사용했습니다.

3. 릴리스 페이지에서 .NET 런타임 및 .NET SDK 버전과 체크섬 파일에 대한 링크를 볼 수 있습니다. 다음 이미지는 .NET 8 릴리스 테이블을 참조로 보여줍니다.



.NET 8.0.0 - November 14, 2023

The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at .NET 8.0 Releases.

Downloads

	SDK Installer[1]	SDK Binaries[1]	Runtime Installer	Runtime Binaries	ASP.NET Core Runtime	Windows Desktop Runtime
Windows	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Hosting Bundle[2]	x86 | x64 | Arm64
macOS	x64 | ARM64	x64 | ARM64	x64 | ARM64	x64 | ARM64	x64 | ARM64	-
Linux	Snap and Package Manager	x64 | Arm | Arm64 | Arm32 Alpine | x64 Alpine	Packages (x64)	x64 | Arm | Arm64 | Arm32 Alpine | Arm64 Alpine | x64 Alpine	x64[1] | Arm[1] | Arm64[1] | x64 Alpine	-
	Checksums	Checksums	Checksums	Checksums	Checksums	Checksums

4. **체크섬** 링크를 마우스 오른쪽 단추로 클릭하고 클립보드에 복사합니다.

5. 터미널을 엽니다.

6. `curl -O {link}`를 사용하여 체크섬 파일을 다운로드합니다.

   다음 명령의 링크를 복사한 링크로 바꿉니다.

   Bash

   ```
   curl -O https://builds.dotnet.microsoft.com/dotnet/checksums/9.0.10-sha.txt
   ```

7. 체크섬 파일과 .NET 릴리스 파일이 모두 동일한 디렉터리에 다운로드되면 `shasum -a 512 -c {file}` 명령을 사용하여 다운로드한 파일의 유효성을 검사합니다.

   유효성 검사를 통과하면 **확인** 상태로 인쇄된 파일이 표시됩니다.

   Bash

   ```
   $ shasum -a 512 -c 9.0.10-sha.txt
   dotnet-sdk-9.0.306-osx-x64.tar.gz: OK
   ```

   FAILED로 표시된 파일이 표시되면 다운로드한 파일이 유효하지 않으므로 사용하면 안 됩니다.

   Bash

   ```
   $ shasum -a 512 -c 9.0.10-sha.txt
   dotnet-sdk-9.0.306-osx-x64.tar.gz: FAILED
   shasum: WARNING: 1 computed checksum did NOT match
   ```

Arm 기반의 Mac

다음 섹션에서는 Arm 기반 Mac에 .NET 설치할 때 고려해야 할 사항에 대해 설명합니다.

경로 차이

Arm 기반 Mac에서는 모든 Arm64 버전의 .NET 일반 */usr/local/share/dotnet/* 폴더에 설치됩니다. 그러나 .NET SDK의 **x64** 버전을 설치하면 */usr/local/share/dotnet/x64/dotnet/* 폴더에 설치됩니다.

경로 변수

`PATH` 변수와 같이 시스템 경로에 .NET 추가하는 환경 변수는 x64 및 Arm64 버전의 .NET SDK가 모두 설치된 경우 변경해야 할 수 있습니다. 또한 일부 도구는 적절한 .NET SDK 설치 폴더를 가리키도록 업데이트해야 하는 `DOTNET_ROOT` 환경 변수를 사용합니다.

문제 해결

문제 해결을 위해 다음 섹션을 사용할 수 있습니다.

- Arm 기반의 Mac
- .NET을 시스템 전체에서 사용할 수 있게 만들기

.NET 시스템 전체에서 사용할 수 있도록 설정

경우에 따라 터미널을 포함한 시스템의 앱에서 .NET 설치되는 위치를 찾아야 합니다. .NET macOS 설치 관리자 패키지 자동으로 시스템을 구성해야 합니다. 그러나 관리 설치 방법 또는 .NET 설치 스크립트를 사용한 경우 .NET 설치된 디렉터리를 `PATH` 변수에 추가해야 합니다.

일부 앱은 .NET 설치된 위치를 확인하려고 할 때 `DOTNET_ROOT` 변수를 찾을 수 있습니다.

macOS에서는 다양한 셸을 사용할 수 있으며 각 셸에는 서로 다른 프로필이 있습니다. 예시:

- **Bash 셸**: *~/.profile*, */etc/profile*
- ko-KR: **Korn 셸**: *~/.kshrc* 또는 *.profile*
- **Z 셸**: *~/.zshrc* 또는 *.zprofile*

셸 프로필에서 다음 두 환경 변수를 설정합니다.

- `DOTNET_ROOT`

  이 변수는 .NET 설치된 폴더(예: `$HOME/.dotnet`)로 설정됩니다.

Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

- `PATH`

  이 변수에는 `DOTNET_ROOT` 폴더와 `DOTNET_ROOT/tools` 폴더가 모두 포함되어야 합니다.

  Bash

  ```
  export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
  ```

관련 콘텐츠

- .NET 이미 설치되어 있는지 확인하는 방법.
- macOS Catalina 공증 관련 사항
- Tutorial: .NET 콘솔 애플리케이션 만듭니다.
- Tutorial: .NET 앱 컨테이너화.

Last updated on 2026. 03. 06.

Linux에 .NET 설치

이 문서에서는 다양한 Linux 배포판에서 .NET을 사용할 수 있는 방법을 설명합니다. .NET은 패키지 관리자나 스냅에 의해 또는 수동으로 설치할 수 있습니다. .NET은 컨테이너 이미지로도 사용할 수도 있습니다.

패키지

패키지는 Microsoft 패키지 리포지토리에 https://packages.microsoft.com/ 게시되며 다음 Linux 배포판에 사용할 수 있습니다.

- Azure 리눅스
- Debian
- openSUSE Leap
- SUSE Enterprise Linux

> **참고 항목**
>
> 배포는 dotnet/core #9556 정의된 정책에 따라 선택됩니다.

다음 Linux 배포판은 자체 .NET 패키지를 게시합니다.

- 알파인
- CentOS 스트림
- 페도라
- Red Hat Enterprise Linux(RHEL)
- Ubuntu

찰칵

.NET SDK 스냅 패키지는 Canonical에서 제공하고 유지 관리합니다. Snap은 Linux 배포에 기본 제공되는 패키지 관리자의 좋은 대안입니다.

- 스냅을 사용하여 .NET 런타임 설치
- 스냅을 사용하여 .NET SDK 설치

수동 설치

다음과 같은 방법으로 .NET을 수동으로 설치할 수 있습니다.

- 수동 설치

- 스크립팅된 설치

.NET을 수동으로 설치하는 경우 .NET 종속성 설치해야 할 수도 있습니다.

추가 원본

.NET은 다른 원본에서도 사용할 수 있습니다. 패키지 및 컨테이너는 다음 이름 중 하나와 유사한 이름을 사용합니다.

- aspnet-runtime
- dotnet-runtime
- dotnet-sdk
- dotnet

패키지 관리자

- https://formulae.brew.sh/cask/dotnet-sdk
- https://ports.macports.org/port/dotnet-cli
- https://search.nixos.org/packages?query=dotnet
- https://archlinux.org
- https://aur.archlinux.org

컨테이너

- https://containers.dev/features
- https://images.chainguard.dev

Last updated on 2025. 11. 08.

Ubuntu에 .NET SDK 또는 .NET 런타임 설치

이 문서에서는 Ubuntu에 .NET 설치하는 방법을 설명합니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

> **Important**
>
> 패키지 관리자를 사용하여 **Microsoft 패키지 피드**에서 .NET 설치하면 **x64** 아키텍처만 지원됩니다. **Arm64**와 같은 다른 아키텍처는 **Microsoft 패키지 피드**에서 지원되지 않습니다. Ubuntu 피드를 사용하거나 .NET 수동으로 설치합니다. 여러 피드를 사용할 때 패키지 혼합 문제를 주의해야 합니다. 자세한 내용은 Linux에서 **.NET 패키지 혼합을 참조하세요**.

.NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.

- install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
- .NET.

우분투 24.04

.NET Ubuntu 패키지 관리자 피드에서 사용할 수 있습니다. Microsoft 패키지 리포지토리에는 더 이상 Ubuntu용 .NET 패키지가 포함되어 있지 않습니다.

다음 버전의 .NET Ubuntu 24.04에서 지원되거나 사용할 수 있습니다.

테이블 확장

지원되는 .NET 버전	사용 가능 기본 제공 Ubuntu 피드	다음에서 사용 가능 백포트 (기존 소프트웨어에 새로운 기능이나 수정을 추가하는 것) Ubuntu 피드	다음에서 사용 가능 Microsoft 피드
10.0, 9.0, 8.0	10.0, 8.0	9.0, 7.0, 6.0	없음

Ubuntu 버전 지원이 중단되면 .NET 해당 버전에서 더 이상 지원되지 않습니다.

.NET 10

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo apt-get install -y dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

종속성

패키지 관리자를 설치할 때 이러한 라이브러리가 설치됩니다. 그러나 .NET 수동으로 설치하거나 자체 포함된 앱을 게시하는 경우 다음 라이브러리가 설치되어 있는지 확인해야 합니다.

- CA 인증서
- libc6
- libgcc-s1

- libgssapi-krb5-2
- libicu74
- libssl3t64
- libstdc++6
- tzdata
- zlib1g

종속성은 `apt install` 명령을 사용하여 설치할 수 있습니다. 다음 코드 조각은 `zlib1g` 라이브러리의 설치를 보여 줍니다.

```Bash
sudo apt install zlib1g
```

지원되지 않는 버전

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

다른 버전을 설치하는 방법

.NET 패키지 이름은 모든 Linux 배포에서 표준화됩니다. 다음 표에는 패키지가 나열되어 있습니다.

모든 버전의 .NET은 https://dotnet.microsoft.com/download/dotnet 에서 다운로드할 수 있지만, 수동 설치가 필요합니다. 패키지 관리자를 사용하여 다른 버전의 .NET 설치할 수 있습니다. 그러나 요청된 버전을 사용하지 못할 수도 있습니다.

패키지 관리자 피드에 추가되는 패키지는 해킹 가능한 형식으로 명명됩니다(예: `{product}-{type}-{version}`).

- **제품**
  설치할 .NET 제품의 유형입니다. 유효한 옵션은 다음과 같습니다.

- `dotnet`
- `aspnetcore`

- **type**
  SDK와 런타임 중 선택합니다. 유효한 옵션은 다음과 같습니다.
  - `sdk`(**dotnet** 제품에만 사용 가능)
  - `runtime`

- **version**
  설치한 SDK 또는 런타임의 버전입니다. 유효한 옵션은 모든 릴리스된 버전입니다. 예:
  - `9.0`
  - `8.0`
  - `3.1`
  - `2.1`

  다운로드하려는 SDK/런타임을 Linux 배포판에서 사용할 수 없을 수 있습니다. 지원되는 배포 목록은 Linux에서 .NET 설치 참조하세요.

예제

- ASP.NET Core 9.0 런타임 설치: `aspnetcore-runtime-9.0`
- .NET Core 2.1 런타임 설치: `dotnet-runtime-2.1`
- .NET 5 SDK 설치: `dotnet-sdk-5.0`
- .NET Core 3.1 SDK 설치: `dotnet-sdk-3.1`

> ⓘ **참고 항목**
>
> 일부 패키지는 Linux 배포판에서 사용할 수 없을 수 있습니다.

패키지가 없음

패키지-버전 조합이 작동하지 않는다면 사용할 수 없는 것입니다. 예를 들어 ASP.NET Core SDK가 없습니다. ASP.NET Core SDK 구성 요소는 .NET SDK에 포함됩니다. 값 `aspnetcore-sdk-8.0`는 올바르지 않으며, 올바른 값은 `dotnet-sdk-8.0`입니다. .NET 지원하는 Linux 배포 목록은 .NET 종속성 및 요구 사항 참조하세요.

다음 단계

- .NET CLI 개요

- TAB 완성을 .NET CLI에서 사용할 수 있도록 설정하는 방법
- Tutorial: .NET.

Last updated on 2026. 03. 06.

Ubuntu 의사 결정 가이드에 .NET 설치

이 문서는 Ubuntu에 .NET 설치하는 방법을 결정하는 데 도움이 됩니다. Ubuntu 22.04부터 지원되는 대부분의 .NET 버전은 기본 제공 Ubuntu 피드에서 사용할 수 있습니다. Ubuntu .NET 백포트 패키지 리포지토리에는 지원되는 나머지 .NET 버전이 포함되어 있습니다.

캐노니컬이 우분투에서 .NET 출판을 인수했습니다. Ubuntu 22.04부터 Microsoft는 더 이상 Ubuntu용 .NET Microsoft 패키지 리포지토리에 배포하지 않습니다.

지원되는 배포

다음 표는 현재 지원되는 .NET 릴리스 및 지원되는 Ubuntu 버전 목록입니다. 각 링크는 해당 버전의 Ubuntu에 대한 .NET 설치하는 방법에 대한 지침과 함께 특정 Ubuntu 버전 페이지로 이동합니다.

테이블 확장

Ubuntu	지원되는 .NET 버전	다음에서 사용 가능 기본 제공 Ubuntu 피드	다음에서 사용 가능 .NET 백포트 Ubuntu 피드	다음에서 사용 가능 Microsoft 피드
25.10	10.0, 9.0, 8.0	10.0, 9.0, 8.0	없음	없음
25.04	10.0, 9.0, 8.0	10.0, 9.0, 8.0	없음	없음
24.04(LTS)	10.0, 9.0, 8.0	10.0, 8.0	9.0, 7.0, 6.0	없음
22.04(LTS)	10.0, 9.0, 8.0	8.0, 7.0, 6.0	10.0, 9.0	8.0, 7.0, 6.0, 3.1

Ubuntu 버전 지원 기간이 끝나면 해당 특정 Ubuntu 버전에서 .NET 더 이상 지원되지 않습니다.

Canonical은 Microsoft에서 제공하는 지원 수명을 초과하더라도 해당 Ubuntu 버전의 수명 동안 기본 제공 Ubuntu 피드의 .NET 버전을 지원하며 Microsoft에서 제공하는 지원 수명을 초과하지 않는 .NET 백포트 패키지 리포지토리의 .NET 버전에 대한 최상의 지원을 제공합니다.

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1

- .NET Core 2.0

Ubuntu 업그레이드 시 고려 사항

Ubuntu를 22.04 이상으로 업그레이드하시겠습니까? 먼저 .NET 제거하는 것이 좋습니다.

패키지 관리자를 사용하여 Microsoft 패키지 리포지토리에서 .NET 설치한 경우 Ubuntu를 업그레이드한 후 패키지 혼합 문제가 발생합니다. Canonical이 Ubuntu 22.04 이상 버전의 패키지 피드에 .NET 게시했으므로 패키지 관리자는 이전에 설치된 .NET 버전에 대해 알지 못합니다. 패키지를 최신 .NET 업그레이드할 수 없습니다. 먼저 제거한 다음 Ubuntu 패키지 리포지토리에서 다시 설치합니다.

.NET 설치하는 방법 결정

Ubuntu 버전이 기본 제공 또는 .NET 백포트 Ubuntu 피드를 통해 .NET 지원하는 경우 이러한 .NET 빌드에 대한 지원이 Canonical에서 제공되고 빌드가 다른 워크로드에 맞게 최적화될 수 있습니다. Microsoft는 Microsoft 패키지 리포지토리 피드의 패키지에 대한 지원을 제공합니다.

> ⚠ **Warning**
>
> 원본 .NET 패키지에 대한 Ubuntu 또는 Microsoft 피드 중에서 선택하는 것이 좋습니다. 앱이 특정 버전의 .NET 해결하려고 할 때 문제가 발생하므로 여러 패키지 리포지토리의 .NET 패키지를 혼합하지 마세요.

테이블 확장

메서드	장점	단점
패키지 관리자 (내장 Ubuntu 피드)	• 일반적으로 최신 버전을 사용할 수 있습니다. • 패치는 즉시 사용할 수 있습니다. • 종속성이 포함되어 있습니다. • 쉽게 제거할 수 있습니다. • 사용 가능한 .NET 버전은 특정 Ubuntu 버전의 지원 기간 동안 지원됩니다. • .NET 8 이상용 IBM System Z 및 Power 플랫폼 지원	• Ubuntu 16.04, 18.04, 20.04에는 사용할 수 없습니다. • 사용 가능한 .NET 버전은 Ubuntu 버전에 따라 다릅니다. • 미리 보기 릴리스는 사용할 수 없습니다.
패키지 관리자 (.NET 백포트 Ubuntu 피드)	• 기본 제공 Ubuntu 피드에는 포함되지 않은 지원되는 버전을 포함합니다. • 패치는 즉시 사용할 수 있습니다. • 종속성이 포함되어 있습니다. • 쉽게 제거할 수 있습니다.	• Ubuntu 16.04, 18.04, 20.04에는 사용할 수 없습니다. • Ubuntu .NET 백포트 패키지 리포지토리를 등록해야 합니다.

메서드	장점	단점
	• 기본 제공 Ubuntu 피드와 호환됩니다.	• 미리 보기 릴리스는 사용할 수 없습니다.
패키지 관리자(Microsoft 피드)	• 지원되는 버전은 언제든지 사용할 수 있습니다. • 패치는 즉시 사용할 수 있습니다. • 종속성이 포함되어 있습니다. • 쉽게 제거할 수 있습니다.	• Ubuntu 24.04 이상에는 사용할 수 없습니다. • Microsoft 패키지 리포지토리 등록이 필요합니다. • 미리 보기 릴리스는 사용할 수 없습니다. • x64 Ubuntu만 지원합니다.
스크립트 \ 수동 추출	• .NET 설치되는 위치를 제어합니다. • 미리 보기 릴리스를 사용할 수 있습니다.	• 업데이트를 수동으로 설치합니다. • 종속성을 수동으로 설치합니다. • 수동으로 제거합니다.

다음 섹션을 사용하여 .NET 설치하는 방법을 결정합니다.

- Ubuntu 22.04 이상을 사용하고 있으며 .NET
- Ubuntu 22.04 이전 버전을 사용하고 있습니다
- 다른 Microsoft 패키지(예: powershell, mdatp, mssql) 사용하고 있습니다
- .NET 앱을 만들려고 합니다
- 컨테이너, 클라우드 또는 연속 통합 시나리오에서 .NET 앱을 실행하려고 합니다
- My Ubuntu 배포에는 원하는 .NET 버전이 포함되지 않거나 지원되지 않는 .NET 버전이 필요합니다
- 미리 보기 버전을 설치하려고 합니다
- APT를 사용하고 싶지 않습니다
- Arm 기반 CPU를 사용하고 있습니다
- IBM System Z 또는 Power 플랫폼을 사용하고 있습니다.

Ubuntu 22.04 이상을 사용하고 있으며, .NET만 필요합니다.

`powershell`, `mdatp` 또는 `mssql` 같은 다른 Microsoft 패키지가 필요하지 않은 경우 Ubuntu 피드를 통해 .NET 설치합니다. 자세한 내용은 다음 페이지를 참조하세요.

- Ubuntu 25.10 .NET 설치합니다.
- Ubuntu 25.04 .NET 설치합니다.
- Ubuntu 24.04 .NET 설치합니다.
- Ubuntu 22.04 .NET 설치합니다.

> ⓘ **Important**

.NET 8 SDK 및 Ubuntu 22.04를 사용하는 경우 Canonical에서 제공하는 SDK 버전은 항상 **.1xx 기능 대역** 있음을 이해합니다. 최신 기능 밴드 릴리스를 사용하려면 **Microsoft 피드를 사용하여 SDK를 설치합니다**. 리포지토리 피드 간 전환의 의미를 이해하기 위해서는 **Linux에서의 .NET 패키지 혼동 문제** 문서의 정보를 반드시 검토해야 합니다.

`powershell`, `mdatp` 또는 `mssql` 같은 다른 Microsoft 패키지를 사용하도록 Microsoft 리포지토리를 설치하려는 경우 Microsoft 리포지토리에서 제공하는 .NET 패키지의 우선 순위를 해제해야 합니다. 패키지의 우선 순위를 해제하는 방법에 대한 지침은 My Linux 배포판에서 .NET 패키지를 제공하며 사용하려는 경우 참조하세요.

Ubuntu 22.04 이전 버전을 사용하고 있습니다

Ubuntu에서 .NET SDK 또는 .NET 런타임을 설치 버전별 섹션의 지침을 사용하세요.

Ubuntu 버전에 대해 지원되는 .NET 버전에 대한 자세한 내용은 지원 배포 섹션을 검토하세요. 지원되지 않는 버전을 설치하려는 경우 Microsoft 패키지 리포지토리 등록을 참조하세요.

다른 Microsoft 패키지(예: `powershell`, `mdatp`, `mssql`)를 사용하고 있습니다

Ubuntu 버전이 Ubuntu 피드를 통해 .NET 지원하는 경우 .NET 설치해야 하는 피드를 결정해야 합니다. 지원 배포 섹션에서는 패키지 피드에서 사용할 수 있는 .NET 버전을 나열하는 테이블을 제공합니다.

Ubuntu 피드에서 .NET 패키지를 원본하려는 경우 Microsoft 리포지토리에서 제공하는 .NET 패키지의 우선 순위를 취소해야 합니다. 패키지의 우선 순위를 해제하는 방법에 대한 지침은 My Linux 배포판에서 .NET 패키지를 제공하며 사용하려는 경우 참조하세요.

.NET 앱을 만들려고 합니다.

런타임에 사용하는 것과 동일한 패키지 소스를 SDK에 사용합니다. Ubuntu 피드를 통해 .NET 설치하는 것이 좋습니다. 그러나 다른 원본(예: Microsoft 패키지 리포지토리에서 더 높은 SDK 기능 밴드에 액세스하기 위해 .NET 설치하려는 경우 .NET 제거하고 Ubuntu 피드에서 .NET 패키지를 무시하고 다른 원본에서 다시 설치하도록 패키지 관리자를 구성해야 합니다.

.NET 설치 방법 섹션의 다른 제안을 검토하세요.

컨테이너, 클라우드 또는 연속 통합 시나리오에서 .NET 앱을 실행하려고 합니다.

Ubuntu 버전에서 필요한 .NET 버전을 제공하는 경우 Ubuntu 피드에서 설치합니다. 그렇지 않으면 Microsoft 패키지 리포지토리를 등록하고 해당 리포지토리에서 .NET 설치합니다. 지원되는 배포 섹션에서 정보를 확인하세요.

원하는 .NET 버전을 사용할 수 없는 경우 dotnet 설치 스크립트 사용해 보세요.

내 Ubuntu 배포에는 원하는 .NET 버전이 포함되지 않거나 지원되지 않는 .NET 버전이 필요합니다.

APT 및 Microsoft 패키지 리포지토리를 사용하는 것이 좋습니다. 자세한 내용은 Microsoft 패키지 리포지토리로 등록 및 설치 섹션을 참조하세요.

미리 보기 버전을 설치하려고 합니다

다음 방법 중 하나를 사용하여 .NET 설치합니다.

- install-dotnet 스크립트를 사용하여 .NET을 설치합니다.
- 관리적으로 .NET

APT를 사용하고 싶지 않습니다

자동 설치를 원하는 경우 Linux 설치 스크립트를 사용합니다.

.NET 설치 환경을 완전히 제어하려면 tarball을 다운로드하고 .NET 수동으로 설치합니다. 자세한 내용은 수동 설치를 참조하세요.

Arm 기반 CPU를 사용하고 있습니다

Ubuntu 버전에서 필요한 .NET 버전을 제공하는 경우 기본 제공 피드에서 설치합니다. 지원되는 배포 섹션에서 정보를 확인하세요.

원하는 .NET 버전을 사용할 수 없는 경우 다음 방법 중 하나를 사용하여 .NET 설치해 보세요.

- 스크립트를 사용하여 .NET을 설치합니다install-dotnet.
- 관리적으로 .NET

IBM System Z 또는 Power 플랫폼을 사용하고 있습니다.

Ubuntu 22.04의 .NET 8부터 Canonical은 IBM System Z 및 Power 플랫폼에 대한 .NET 지원합니다. 이 지원은 앞으로 모든 .NET 릴리스에 대해 계속됩니다.

기본 제공 Ubuntu 피드를 통해 .NET 설치합니다. 자세한 내용은 다음 페이지를 참조하세요.

- Ubuntu 25.10 .NET 10을 설치합니다.

패키지 리포지토리 등록

Ubuntu 버전에 따라 Ubuntu 백포트 또는 Microsoft 패키지 리포지토리를 등록해야 할 수 있습니다.

> ⓘ **Important**
>
> **.NET 설치 방법 결정 섹션**의 정보를 고려해야 합니다.

- Ubuntu .NET 백포트 패키지 리포지토리 등록
- Microsoft 패키지 리포지토리 등록

Ubuntu .NET 백포트 패키지 리포지토리

Ubuntu .NET 백포트 패키지 리포지토리는 기본 제공 Ubuntu 피드에서 사용할 수 없는 .NET 버전을 제공합니다. Canonical은 이 패키지 리포지토리에 포함된 패키지를 유지 관리하며 Microsoft에서 제공하는 지원 수명 또는 특정 Ubuntu 버전의 지원 기간 이내에서 최상의 지원을 제공합니다.

이 패키지 리포지토리는 Ubuntu 24.04 LTS(Noble Numbat) 및 Ubuntu 22.04 LTS(Jammy 해파리)에서 지원됩니다. 지원 배포 섹션 패키지 피드에서 사용할 수 있는 .NET 버전을 나열하는 테이블을 제공합니다. 자세한 내용은 Ubuntu .NET 백포트 패키지 리포지토리 를 참조하세요.

이 패키지 리포지토리를 추가하려면 다음 명령을 실행합니다.

Bash

```
sudo add-apt-repository ppa:dotnet/backports
sudo apt update
```

Ubuntu .NET 백포트 패키지 리포지토리 등록

터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo add-apt-repository ppa:dotnet/backports
```

> ⓘ **참고 항목**

Ubuntu .NET 백포트 패키지 리포지토리는 기본 제공 Ubuntu 피드와 호환됩니다. 따라서 기본 제공 Ubuntu 피드에서 .NET 패키지를 무시하도록 패키지 관리자를 구성할 필요가 없습니다.

Ubuntu .NET 백포트 패키지 리포지토리 등록 취소

Ubuntu .NET 백포트 패키지 리포지토리에서 패키지를 더 이상 사용하지 않으려면 등록을 취소할 수 있습니다. 터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo add-apt-repository --remove ppa:dotnet/backports
```

> **Important**
>
> Ubuntu .NET 백포트 패키지 리포지토리를 등록 취소해도 패키지는 제거되지 않습니다.

add-apt-repository 명령을 찾을 수 없음

add-apt-repository(1) 유틸리티는 대부분의 Ubuntu 설치에 미리 설치되어 있습니다.

`add-apt-repository` 명령을 찾을 수 없다는 오류 메시지가 표시되면 이 명령을 제공하는 `software-properties-common` 패키지를 설치해야 합니다. 터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo apt update
sudo apt install software-properties-common
```

Microsoft 패키지 리포지토리 등록

> **Important**
>
> 이는 24.04 이전의 Ubuntu 버전에만 적용됩니다. Ubuntu 24.04부터 Microsoft는 더 이상 Microsoft 패키지 리포지토리에 패키지를 게시하지 않습니다. **지원되는 배포 테이블** 사용하여 .NET 설치하는 가장 좋은 방법을 결정합니다.

Microsoft 패키지 리포지토리에는 이전 버전이거나 현재 Ubuntu 지원된 모든 버전의 .NET 포함되어 있습니다. Ubuntu 버전에서 .NET 패키지를 제공하는 경우 Ubuntu 패키지의 우선 순위를 제거하고 Microsoft 리포지토리를 사용해야 합니다. 패키지의 우선 순위를 해제하는 방법에 대한 지침은 내 Linux 배포판에서 제공하지 않는 .NET 버전이 필요합니다 참조하세요.

> ⓘ **Important**
>
> Microsoft 패키지 리포지토리는 **x64** 아키텍처를 대상으로 하는 .NET 패키지만 지원합니다. **Arm** 같은 다른 아키텍처는 **installer 스크립트** 또는 **관리 설치** 같은 다른 방법을 통해 .NET 설치해야 합니다.

미리 보기 릴리스는 Microsoft 패키지 리포지토리에서 사용할 수 **없습니다**. 자세한 내용은 미리 보기 버전 설치를 참조하세요.

> ⊗ **주의**
>
> 리포지토리를 하나만 사용하여 모든 .NET 설치를 관리하는 것이 좋습니다. 이전에 Ubuntu 리포지토리를 사용하여 .NET 설치한 경우 .NET 패키지 시스템을 정리하고 Ubuntu 피드를 무시하도록 APT를 구성해야 합니다. 이 작업을 수행하는 방법에 대한 자세한 내용은 **내 Linux 배포판에서 제공하지 않는 .NET 버전이 필요합니다** 참조하세요.

몇 가지 명령을 사용하여 APT 설치를 완료할 수 있습니다. .NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 패키지 리포지토리를 추가합니다.

터미널을 열고 다음 명령을 실행합니다.

Bash

```
# Get OS version info which adds the $ID and $VERSION_ID variables
source /etc/os-release

# Download Microsoft signing key and repository
wget https://packages.microsoft.com/config/$ID/$VERSION_ID/packages-microsoft-
prod.deb -O packages-microsoft-prod.deb

# Install Microsoft signing key and repository
sudo dpkg -i packages-microsoft-prod.deb

# Clean up
rm packages-microsoft-prod.deb

# Update packages
sudo apt update
```

> **팁**
>
> 이전 스크립트는 Ubuntu용으로 작성되었으며 Linux Mint와 같은 파생 배포를 사용하는 경우 작동하지 않을 수 있습니다. `$ID` 및 `$VERSION_ID` 변수에 올바른 값이 할당되지 않을 가능성이 있어 `wget` 명령의 URI가 유효하지 않게 됩니다. `$ID`는 배포에 해당하고(예: `ubuntu`), `$VERSION_ID`는 22.04 또는 23.10과 같이 패키지를 가져올 특정 Ubuntu 버전에 매핑됩니다.
>
> 예를 들어 Ubuntu 22.04 `$ID` `ubuntu` `$VERSION_ID` `22.04`에서는 다음과 같습니다. URL은 다음과 `https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb` 같습니다.
>
> 웹 브라우저에서 https://packages.microsoft.com/config/ubuntu/ (으)로 이동하여 `$repo_version` 값으로 사용할 수 있는 Ubuntu 버전을 확인할 수 있습니다.

.NET 설치, 제거 또는 업데이트

다음 섹션에서는 패키지 관리자를 통해 .NET 관리하는 방법을 설명합니다.

설치 .NET

`sudo apt install <package-name>` 명령을 사용하여 패키지 관리자를 통해 .NET 설치합니다. `<package-name>` 설치하려는 .NET 패키지의 이름으로 바꿉 있습니다. 예를 들어 .NET SDK 10.0을 설치하려면 `sudo apt install dotnet-sdk-10.0` 명령을 사용합니다. 다음 표에서는 현재 지원되는 .NET 패키지를 (Ubuntu 버전에 따라 달라질 수 있음) 나열합니다.

테이블 확장

	제품	유형	패키지
10.0	ASP.NET Core	실행 시간	`aspnetcore-runtime-10.0`
10.0	.NET	실행 시간	`dotnet-runtime-10.0`
10.0	.NET	SDK (소프트웨어 개발 키트)	`dotnet-sdk-10.0`
9.0	ASP.NET Core	실행 시간	`aspnetcore-runtime-9.0`
9.0	.NET	실행 시간	`dotnet-runtime-9.0`
9.0	.NET	SDK (소프트웨어 개발 키트)	`dotnet-sdk-9.0`
8.0	ASP.NET Core	실행 시간	`aspnetcore-runtime-8.0`

	제품	유형	패키지
8.0	.NET	실행 시간	`dotnet-runtime-8.0`
8.0	.NET	SDK (소프트웨어 개발 키트)	`dotnet-sdk-8.0`

> **팁**
>
> .NET 앱을 만들지 않는 경우 .NET 런타임을 포함하고 ASP.NET Core 앱을 지원하므로 ASP.NET Core 런타임을 설치합니다.

일부 환경 변수는 .NET 설치 후 실행되는 방식에 영향을 줍니다. 자세한 내용은 .NET SDK 및 CLI 환경 변수 참조하세요.

.NET 제거

패키지 관리자를 통해 .NET 설치한 경우 `apt-get remove` 명령을 사용하여 동일한 방식으로 제거합니다.

Bash

```
sudo apt-get remove dotnet-sdk-6.0
```

자세한 내용은 .NET 제거(c0)를 참조하세요.

.NET 업데이트하기

패키지 관리자를 통해 .NET 설치한 경우 `apt upgrade` 명령을 사용하여 패키지를 업그레이드할 수 있습니다. 예를 들어 다음 명령은 `dotnet-sdk-10.0` 패키지를 최신 버전으로 업그레이드합니다.

Bash

```
sudo apt update
sudo apt upgrade dotnet-sdk-10.0
```

> **팁**
>
> .NET 설치한 후 Linux 배포를 업그레이드한 경우 Microsoft 패키지 리포지토리를 다시 구성해야 할 수 있습니다. 현재 배포 버전에 대한 설치 지침을 실행하여 .NET 업데이트에 적합한 패키지 리포지토리로 업그레이드합니다.

미리 보기 버전 관리

다음 섹션에서는 .NET 미리 보기 릴리스를 설치하고 제거하는 방법을 설명합니다.

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

문제 해결

Ubuntu 22.04부터는 .NET 일부만 사용할 수 있는 상황이 발생할 수 있습니다. 예를 들어 런타임과 SDK를 설치했지만 `dotnet --info`을(를) 실행할 때 런타임만 나열됩니다. 이 상황은 서로 다른 두 패키지 소스를 사용하는 경우와 관련이 있을 수 있습니다. 기본 제공 Ubuntu 22.04 및 Ubuntu 22.10 패키지 피드에는 일부 버전의 .NET 포함되어 있지만 전부는 아니며 Microsoft 피드에서 .NET 설치했을 수도 있습니다. 이 문제를 해결하는 방법에 대한 자세한 내용은 Linux에서 누락된 파일과 관련된 오류 .NET 문제 해결 참조하세요.

APT 문제

이 섹션에서는 APT를 사용하여 .NET 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

패키지를 찾을 수 없음

> **Important**
>
> 패키지 관리자를 사용하여 **Microsoft 패키지 피드**에서 .NET 설치하면 **x64** 아키텍처만 지원됩니다. **Arm64**와 같은 다른 아키텍처는 **Microsoft 패키지 피드**에서 지원되지 않습니다.

Ubuntu 피드를 사용하거나 .NET 수동으로 설치합니다. 여러 피드를 사용할 때 패키지 혼합 문제를 주의해야 합니다. 자세한 내용은 Linux에서 .NET 패키지 혼합을 참조하세요.

.NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.

- install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
- .NET.

찾을 수 없음\일부 패키지를 설치할 수 없음

> ⓘ **참고 항목**
>
> 이 정보는 Microsoft 패키지 피드에서 .NET 설치된 경우에만 적용됩니다.

{netcore-package} 패키지를 찾을 수 없음 또는 **일부 패키지를 설치할 수 없음**과 같은 오류 메시지가 표시되는 경우 다음 명령을 실행합니다.

다음 명령 집합에는 두 개의 자리 표시자가 있습니다.

- `{dotnet-package}`
  이는 설치하려는 .NET 패키지(예: `aspnetcore-runtime-8.0`)를 나타냅니다. 다음 `sudo apt-get install` 명령에 사용됩니다.

먼저, 다음 패키지 목록을 제거해 봅니다.

Bash

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
```

그런 다음 .NET 다시 설치해 봅니다. 이 방법으로 문제가 해결되지 않으면 다음 명령을 사용하여 수동 설치를 실행할 수 있습니다.

Ubuntu 23.10 이상을 사용하는 경우 다음 명령을 사용해 보세요.

Bash

```
# Get OS version info which adds the $ID and $VERSION_ID variables
source /etc/os-release

# Download the Microsoft keys
sudo apt-get install -y gpg wget
```

```
wget https://packages.microsoft.com/keys/microsoft.asc
cat microsoft.asc | gpg --dearmor -o microsoft.asc.gpg

# Add the Microsoft repository to the system's sources list
wget https://packages.microsoft.com/config/$ID/$VERSION_ID/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list

# Move the key to the appropriate place
sudo mv microsoft.asc.gpg $(cat /etc/apt/sources.list.d/microsoft-prod.list | grep
-oP "(?<=signed-by=).*(?=\])")

# Update packages and install .NET
sudo apt-get update && \
  sudo apt-get install -y {dotnet-package}
```

23.10 이전의 Ubuntu 버전을 사용하는 경우 다음 명령을 사용해 보세요.

Bash

```
# Define the OS version, name, and codename
source /etc/os-release

# Download the Microsoft keys
sudo apt-get install -y gpg wget
wget https://packages.microsoft.com/keys/microsoft.asc
cat microsoft.asc | gpg --dearmor -o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/

# Add the Microsoft repository to the system's sources list
wget https://packages.microsoft.com/config/$ID/$VERSION_ID/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list

# Set ownership
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list

# Update packages and install .NET
sudo apt-get update && \
  sudo apt-get install -y {dotnet-package}
```

가져오지 못함

.NET 패키지를 설치하는 동안 `Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 유사한 오류가 표시 될 수 있습니다. 이 오류는 .NET 대한 패키지 피드가 최신 패키지 버전으로 업그레이드되고 있으며 나중에 다시 시도해야 한다는 것을 의미할 수 있습니다. 업그레이드하는 동안 30분 이상 패키지 피드를 사용할 수 없습니다. 이 오류 메시지가 30분 이상 계속 표시되는 경우 https://github.com/dotnet/core/issues 에서 문제를 제출하세요.

종속성

패키지 관리자를 설치할 때 이러한 라이브러리가 설치됩니다. 그러나 .NET 수동으로 설치하거나 자체 포함된 앱을 게시하는 경우 다음 종속성을 설치하여 앱을 실행해야 합니다.

- CA 인증서
- libc6
- libgcc1(16.x 및 18.x용)
- libgcc-s1(20.x 이상용)
- libgssapi-krb5-2
- libicu55(16.x용)
- libicu60(18.x용)
- libicu66(20.x용)
- libicu70(22.04용)
- libicu72(버전 23.10)
- libicu74(24.04의 경우)
- libicu76(25.04 이상)
- libssl1.0.0(16.x용)
- libssl1.1(18.x용, 20.x용)
- libssl3(22.x 이상)
- libstdc++6
- tzdata
- zlib1g

종속성은 `apt install` 명령을 사용하여 설치할 수 있습니다. 다음 코드 조각은 `zlib1g` 라이브러리의 설치를 보여 줍니다.

Bash

```
sudo apt install zlib1g
```

다음 단계

- .NET CLI 개요
- .NET CLI에 대해 Tab 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 06.

Alpine에 .NET SDK 또는 .NET 런타임 설치

.NET Alpine에서 지원되며 이 문서에서는 Alpine에 .NET 설치하는 방법을 설명합니다. Alpine 버전이 지원되지 않는 경우 .NET 해당 버전에서 더 이상 지원되지 않습니다.

Docker를 사용하는 경우 .NET 직접 설치하는 대신 official .NET Docker 이미지를 사용하는 것이 좋습니다.

> **ⓘ Important**
>
> .NET 10은 2025년 11월 11일에 릴리스되었습니다. 패키지 관리자 피드에 패키지가 표시되거나 특정 Linux 배포판에 패키지가 포함되는 데 시간이 걸릴 수 있습니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

지원되는 배포

다음 표는 현재 지원되는 .NET 릴리스 및 지원되는 Alpine 버전 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 알핀이 수명 종료 도달할 때까지 계속 지원됩니다.

테이블 확장

알파인	지원되는 버전	Package Manager에서 이용 가능
3.23	10, 9, 8	10, 9, 8
3.22	10, 9, 8	9, 8
3.21	9, 8	9, 8
3.20	9, 8	8, 6

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

설치 .NET

.NET 10

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo apk add dotnet10-sdk
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo apk add aspnetcore10-runtime
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore10-runtime` `dotnet10-runtime` 바꿉니다.

Bash

```
sudo apk add dotnet10-runtime
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

지원되는 아키텍처

다음 표는 현재 지원되는 .NET 릴리스 및 지원되는 Alpine 아키텍처 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 알핀이 지원됨# 아키텍처에 도달할 때까지 계속 지원됩니다. `x86_64`, `armv7`, `aarch64`는 Microsoft에서 공식적으로 지원됩니다. 다른 아키텍처는 배포 유지 관리자에서 지원되며 해당 아키텍처에 패키지를 사용할 수 있는 경우 `apk` 패키지 관리자를 사용하여 설치할 수 있습니다.

테이블 확장

아키텍처	.NET 10	.NET 9	.NET 8
x86_64	3.22	3.20, 3.21, 3.22	3.20, 3.21, 3.22
x86	없음	없음	없음
aarch64	3.22	3.20, 3.21, 3.22	3.20, 3.21, 3.22
armv7	3.22	3.20, 3.21, 3.22	3.20, 3.21, 3.22
armhf	없음	없음	없음
s390x	없음	없음	없음
ppc64le	없음	없음	없음
riscv64	없음	없음	없음

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

종속성

패키지 관리자를 설치할 때 이러한 라이브러리가 설치됩니다. 그러나 .NET 수동으로 설치하거나 자체 포함된 앱을 게시하는 경우 다음 라이브러리가 설치되어 있는지 확인해야 합니다.

3.20 이상

- CA 인증서
- libgcc
- libssl3
- libstdc++
- zlib(.NET 8만 해당)
- icu-libs 및 icu-data-full(.NET 앱이 글로발화 고정 모드에서 실행되지 않는 한)
- tzdata
- krb5

`apk add` 명령을 사용하여 종속성을 설치합니다.

다음 단계

- .NET CLI 개요
- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 06.

RHEL 및 CentOS Stream에 .NET SDK 또는 .NET 런타임 설치

.NET RHEL(Red Hat Enterprise Linux)에서 지원됩니다. 이 문서에서는 RHEL 및 CentOS Stream에 .NET 설치하는 방법을 설명합니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

Red Hat 구독 등록

RHEL의 Red Hat에서 .NET 설치하려면 먼저 Red Hat 구독 관리자를 사용하여 등록해야 합니다. 시스템에서 이 작업이 수행되지 않았거나 확실하지 않은 경우 .NETRed Hat 제품 설명서를 참조하세요>.

> **Important**
>
> 이전 문은 CentOS Stream에 적용되지 않습니다.

지원되는 배포

다음 표는 RHEL 및 CentOS Stream 모두에서 현재 지원되는 .NET 릴리스 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 Linux 배포가 더 이상 지원되지 않을 때까지 계속 지원됩니다.

테이블 확장

배포	.NET
RHEL 10	10, 9, 8
RHEL 9	10, 9, 8
RHEL 8	10, 9, 8
CentOS Stream 10	10, 9, 8
CentOS Stream 9	10, 9, 8

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

RHEL 10

.NET RHEL 10의 AppStream 리포지토리 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

RHEL 9

.NET RHEL 9용 AppStream 리포지토리 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

RHEL 8

.NET RHEL 8용 AppStream 리포지토리 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

CentOS Stream 10

.NET CentOS Stream 10용 AppStream 리포지토리에 포함됩니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

CentOS Stream 9

.NET CentOS Stream 9의 AppStream 리포지토리에 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

CentOS Linux는 어디에 있나요?

.NET CentOS Linux에서 더 이상 지원되지 않습니다. 2024년 6월 30일을 기준으로 CentOS Linux는 수명이 다했습니다. 자세한 내용은 CentOS Stream 8 및 CentOS Linux 7의 종료 날짜 를 참조하세요.

종속성

RHEL 및 CentOS Stream에서 .NET 실행하려면 다음 라이브러리가 필요합니다. 패키지 관리자를 `dnf` 사용하여 설치합니다.

- glibc
- libgcc
- CA 인증서
- openssl-libs
- libstdc++
- libicu
- tzdata
- krb5-libs
- zlib(.NET 8에만 필요)

예를 들어 모든 종속성을 설치하려면 다음을 수행합니다.

Bash

```
sudo dnf install glibc libgcc ca-certificates openssl-libs libstdc++ libicu tzdata
krb5-libs
```

.NET 8의 경우 다음을 설치합니다.

Bash

```
sudo dnf install zlib
```

다른 버전을 설치하는 방법

.NET의 다른 릴리스를 설치하는 데 필요한 단계를 알아보려면 Red Hat의 .NET 문서를 참조하세요.

패키지 관리자 문제 해결

이 섹션에서는 패키지 관리자를 사용하여 .NET 또는 .NET Core를 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

누락된 `fxr`, `libhostfxr.so` 또는 `FrameworkList.xml` 관련 오류

이 문제를 해결하는 방법에 대한 자세한 내용은 fxr, libhostfxr.so, 및 FrameworkList.xml 오류를 해결하는 방법을 참조하세요.

다음 단계

- .NET CLI 개요

- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 06.

Debian에 .NET SDK 또는 .NET 런타임 설치

이 문서에서는 Debian에 .NET 설치하는 방법을 설명합니다. Debian 버전이 지원되지 않는 경우 .NET 해당 버전에서 더 이상 지원되지 않습니다. 그러나 이러한 지침은 지원되지 않더라도 해당 버전에서 .NET 실행하는 데 도움이 될 수 있습니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

지원되는 배포

다음 표는 현재 지원되는 .NET 릴리스 및 지원되는 Debian 버전 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 Debian 버전이 수명 종료 도달할 때까지 계속 지원됩니다.

테이블 확장

데비안	.NET
13	10, 9, 8
12	10, 9, 8

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치

- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

Debian 13

몇 가지 명령을 사용하여 APT 설치를 완료할 수 있습니다. .NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 패키지 리포지토리를 추가합니다.

터미널을 열고 다음 명령을 실행합니다.

Bash

```
wget https://packages.microsoft.com/config/debian/13/packages-microsoft-prod.deb -O
packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

.NET 10

> ⓘ **Important**
>
> **Microsoft 패키지 피드**는 .NET 10에 대한 **x64** 및 **Arm64** 패키지만 게시합니다. **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo apt-get install -y dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

Debian 12

몇 가지 명령을 사용하여 APT 설치를 완료할 수 있습니다. .NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 패키지 리포지토리를 추가합니다.

터미널을 열고 다음 명령을 실행합니다.

Bash

```
wget https://packages.microsoft.com/config/debian/12/packages-microsoft-prod.deb -O
packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

.NET 10

> **ⓘ Important**
>
> **Microsoft 패키지 피드**는 .NET 10에 대한 **x64** 및 **Arm64** 패키지만 게시합니다. **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo apt-get install -y dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

APT를 사용하여 .NET 업데이트

새 .NET 패치 릴리스가 사용 가능하면, 다음 명령을 통해 APT로 .NET을 업그레이드할 수 있습니다.

Bash

```
sudo apt-get update
sudo apt-get upgrade
```

.NET 설치한 후 Linux 배포를 업그레이드한 경우 Microsoft 패키지 리포지토리를 다시 구성해야 할 수 있습니다. 현재 배포 버전에 대한 설치 지침을 실행하여 .NET 업데이트에 적합한 패키지 리포지토리로 업그레이드합니다.

문제 해결

이 섹션에서는 APT를 사용하여 .NET 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

패키지를 찾을 수 없음

ⓘ Important

Microsoft 패키지 피드는 .NET 버전에 따라 다른 아키텍처에 대한 패키지를 게시합니다.

- **.NET 10**: **x64** 및 **Arm64** 패키지만 해당합니다.
- **.NET 9**: **x64** 전용 패키지입니다.
- **.NET 8**: **x64** 패키지만.

Arm32 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.

- install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
- .NET.

찾을 수 없음\일부 패키지를 설치할 수 없음

{netcore-package} 패키지를 찾을 수 없음 또는 **일부 패키지를 설치할 수 없음**과 같은 오류 메시지가 표시되는 경우 다음 명령을 실행합니다.

다음 명령 집합에는 두 개의 자리 표시자가 있습니다.

- `{dotnet-package}`
  이는 설치하려는 .NET 패키지(예: `aspnetcore-runtime-8.0`)를 나타냅니다. 다음 `sudo apt-get install` 명령에 사용됩니다.

먼저, 다음 패키지 목록을 제거해 봅니다.

Bash

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
```

그런 다음 .NET 다시 설치해 봅니다. 이 방법으로 문제가 해결되지 않으면 다음 명령을 사용하여 수동 설치를 실행할 수 있습니다.

Debian 12 이상을 사용하는 경우 다음 명령을 시도합니다.

Bash

```
# Get OS version info which adds the $ID and $VERSION_ID variables
source /etc/os-release

# Download the Microsoft keys
sudo apt-get install -y gpg wget
wget https://packages.microsoft.com/keys/microsoft.asc
cat microsoft.asc | gpg --dearmor -o microsoft.asc.gpg

# Add the Microsoft repository to the system's sources list
wget https://packages.microsoft.com/config/$ID/$VERSION_ID/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list

# Move the key to the appropriate place
sudo mv microsoft.asc.gpg $(cat /etc/apt/sources.list.d/microsoft-prod.list | grep -oP "(?<=signed-by=).*(?=\])")

# Update packages and install .NET
sudo apt-get update && \
  sudo apt-get install -y {dotnet-package}
```

12보다 오래된 Debian 버전을 사용하는 경우 다음 명령을 시도합니다.

Bash

```
# Define the OS version, name, and codename
source /etc/os-release

# Download the Microsoft keys
sudo apt-get install -y gpg wget
wget https://packages.microsoft.com/keys/microsoft.asc
cat microsoft.asc | gpg --dearmor -o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/

# Add the Microsoft repository to the system's sources list
wget https://packages.microsoft.com/config/$ID/$VERSION_ID/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list

# Set ownership
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list

# Update packages and install .NET
sudo apt-get update && \
  sudo apt-get install -y {dotnet-package}
```

가져오지 못함

.NET 패키지를 설치하는 동안 `Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 유사한 오류가 표시 될 수 있습니다. 이 오류는 .NET 대한 패키지 피드가 최신 패키지 버전으로 업그레이드되고 있으며 나중에 다시 시도해야 한다는 것을 의미할 수 있습니다. 업그레이드하는 동안 30분 이상 패키지 피드를 사용할 수 없습니다. 이 오류 메시지가 30분 이상 계속 표시되는 경우 https://github.com/dotnet/core/issues 에서 문제를 제출하세요.

종속성

패키지 관리자를 설치할 때 이러한 라이브러리가 설치됩니다. 그러나 .NET 수동으로 설치하거나 자체 포함된 앱을 게시하는 경우 다음 라이브러리가 설치되어 있는지 확인해야 합니다.

13.x

- libc6
- libgcc-s1
- libgssapi-krb5-2
- libicu72
- libssl3
- libstdc++6
- zlib1g

12.x

- libc6
- libgcc-s1
- libgssapi-krb5-2
- libicu72
- libssl3
- libstdc++6
- zlib1g

기타 참고 사항

종속성은 `apt install` 명령을 사용하여 설치할 수 있습니다. 다음 코드 조각은 `libc6` 라이브러리의 설치를 보여 줍니다.

Bash

```
sudo apt install libc6
```

다음 단계

- .NET CLI 개요
- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 11.

Fedora에 .NET SDK 또는 .NET 런타임 설치

.NET Fedora에서 지원되며 이 문서에서는 Fedora에 .NET 설치하는 방법을 설명합니다. Fedora 버전이 지원되지 않는 경우 .NET 해당 버전에서 더 이상 지원되지 않습니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

패키지 관리자 없이 .NET 설치하는 방법에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.

- 스크립트를 사용하여 .NET SDK 또는 .NET 런타임을 설치합니다.
- .NET SDK 또는 .NET 런타임을 수동으로 설치합니다.

지원되는 배포

다음 표는 현재 지원되는 .NET 릴리스 및 지원되는 Fedora 버전 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 Fedora 버전이 수명 종료 도달할 때까지 계속 지원됩니다.

테이블 확장

페도라	.NET
43	10, 9, 8
42	10, 9, 8
41	10, 9, 8

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

.NET 10

> **Important**
>
> .NET 10은 2025년 11월 11일에 릴리스되었습니다. 패키지 관리자 피드에 패키지가 표시되거나 특정 Linux 배포판에 패키지가 포함되는 데 시간이 걸릴 수 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

종속성

.NET은 다양한 기능을 위해 여러 Linux 패키지에 의존합니다. 다음 패키지가 필요합니다.

- glibc
- libgcc
- CA 인증서
- openssl-libs
- libstdc++
- libicu
- tzdata
- krb5-libs
- zlib(.NET 8에만 필요)

다음 명령을 사용하여 필요한 모든 패키지를 설치할 수 있습니다.

Bash

```
sudo dnf install -y glibc libgcc ca-certificates openssl-libs libstdc++ libicu
tzdata krb5-libs zlib
```

다른 버전을 설치하는 방법

모든 버전의 .NET은 https://dotnet.microsoft.com/download/dotnet 에서 다운로드할 수 있지만, 수동 설치가 필요합니다. 패키지 관리자를 사용하여 다른 버전의 .NET 설치할 수 있습니다. 그러나 요청된 버전을 사용하지 못할 수도 있습니다.

패키지 관리자 피드에 추가되는 패키지는 해킹 가능한 형식으로 명명됩니다(예: `{product}-{type}-{version}`).

- **제품**
  설치할 .NET 제품의 유형입니다. 유효한 옵션은 다음과 같습니다.
  - `dotnet`
  - `aspnetcore`

- **type**
  SDK와 런타임 중 선택합니다. 유효한 옵션은 다음과 같습니다.
  - `sdk` (**dotnet** 제품에만 사용 가능)
  - `runtime`

- **version**
  설치한 SDK 또는 런타임의 버전입니다. 유효한 옵션은 모든 릴리스된 버전입니다. 예:
  - `9.0`
  - `8.0`
  - `3.1`
  - `2.1`

  다운로드하려는 SDK/런타임을 Linux 배포판에서 사용할 수 없을 수 있습니다. 지원되는 배포 목록은 Linux에서 .NET 설치 참조하세요.

예제

- ASP.NET Core 9.0 런타임 설치: `aspnetcore-runtime-9.0`
- .NET Core 2.1 런타임 설치: `dotnet-runtime-2.1`
- .NET 5 SDK 설치: `dotnet-sdk-5.0`
- .NET Core 3.1 SDK 설치: `dotnet-sdk-3.1`

> ⓘ **참고 항목**
>
> 일부 패키지는 Linux 배포판에서 사용할 수 없을 수 있습니다.

패키지가 없음

패키지-버전 조합이 작동하지 않는다면 사용할 수 없는 것입니다. 예를 들어 ASP.NET Core SDK가 없습니다. ASP.NET Core SDK 구성 요소는 .NET SDK에 포함됩니다. 값 `aspnetcore-sdk-8.0`는 올바르지 않으며, 올바른 값은 `dotnet-sdk-8.0`입니다. .NET 지원하는 Linux 배포 목록은 .NET 종속성 및 요구 사항 참조하세요.

패키지 관리자 문제 해결

이 섹션에서는 패키지 관리자를 사용하여 .NET 또는 .NET Core를 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

패키지를 찾을 수 없음

패키지 관리자 없이 .NET 설치하는 방법에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.

- 스크립트를 사용하여 .NET SDK 또는 .NET 런타임을 설치합니다.
- .NET SDK 또는 .NET 런타임을 수동으로 설치합니다.

가져오지 못함

.NET 패키지를 설치하는 동안 `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 유사한 오류가 표시 될 수 있습니다. 일반적으로 이 오류는 .NET 대한 패키지 피드가 최신 패키지 버전으로 업그레이드되고 있으며 나중에 다시 시도해야 한다는 것을 의미합니다. 업그레이드하는 동안 2시간 이상 패키지 피드를 사용할 수 없습니다. 이 오류 메시지가 2시간 이상 계속 표시되는 경우 https://github.com/dotnet/core/issues 에서 문제를 제출하세요.

누락된 `fxr`, `libhostfxr.so`, `FrameworkList.xml` 또는 `/usr/share/dotnet` 관련 오류

이 문제를 해결하는 방법에 대한 자세한 내용은 fxr, libhostfxr.so, 및 FrameworkList.xml 오류를 해결하는 방법을 참조하세요.

다음 단계

- .NET CLI 개요
- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 11.

openSUSE Leap에 .NET SDK 또는 .NET 런타임 설치

.NET openSUSE Leap에서 지원됩니다. 이 문서에서는 openSUSE Leap에 .NET 설치하는 방법을 설명합니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

지원되는 배포

다음 표는 openSUSE Leap 15에서 현재 지원되는 .NET 릴리스 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 openSUSE Leap 버전이 더 이상 지원되지 않을 때까지 계속 지원됩니다.

테이블 확장

openSUSE Leap	.NET
16	10, 9, 8
15.6	10, 9, 8

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

openSUSE Leap 16

.NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 Microsoft 패키지 리포지토리를 추가합니다. 터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo zypper install libicu
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
wget https://packages.microsoft.com/config/opensuse/16/prod.repo
sudo mv prod.repo /etc/zypp/repos.d/microsoft-prod.repo
sudo chown root:root /etc/zypp/repos.d/microsoft-prod.repo
```

.NET 10

> ⓘ Important
>
> **Microsoft 패키지 피드**는 .NET 10에 대한 **x64** 및 **Arm64** 패키지만 게시합니다. **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo zypper install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo zypper install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo zypper install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

openSUSE Leap 15

.NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 Microsoft 패키지 리포지토리를 추가합니다. 터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo zypper install libicu
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
wget https://packages.microsoft.com/config/opensuse/15/prod.repo
sudo mv prod.repo /etc/zypp/repos.d/microsoft-prod.repo
sudo chown root:root /etc/zypp/repos.d/microsoft-prod.repo
```

.NET 10

> ⓘ Important
>
> **Microsoft 패키지 피드**는 .NET 10에 대한 **x64** 및 **Arm64** 패키지만 게시합니다. **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo zypper install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo zypper install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo zypper install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

다른 버전을 설치하는 방법

모든 버전의 .NET은 https://dotnet.microsoft.com/download/dotnet 에서 다운로드할 수 있지만, 수동 설치가 필요합니다. 패키지 관리자를 사용하여 다른 버전의 .NET 설치할 수 있습니다. 그러나 요청된 버전을 사용하지 못할 수도 있습니다.

패키지 관리자 피드에 추가되는 패키지는 해킹 가능한 형식으로 명명됩니다(예: `{product}-{type}-{version}`).

- **제품**
  설치할 .NET 제품의 유형입니다. 유효한 옵션은 다음과 같습니다.
  - `dotnet`
  - `aspnetcore`

- **type**
  SDK와 런타임 중 선택합니다. 유효한 옵션은 다음과 같습니다.
  - `sdk` (**dotnet** 제품에만 사용 가능)
  - `runtime`

- **version**
  설치한 SDK 또는 런타임의 버전입니다. 유효한 옵션은 모든 릴리스된 버전입니다. 예:
  - `9.0`
  - `8.0`
  - `3.1`
  - `2.1`

  다운로드하려는 SDK/런타임을 Linux 배포판에서 사용할 수 없을 수 있습니다. 지원되는 배포 목록은 Linux에서 .NET 설치 참조하세요.

예제

- ASP.NET Core 9.0 런타임 설치: `aspnetcore-runtime-9.0`
- .NET Core 2.1 런타임 설치: `dotnet-runtime-2.1`
- .NET 5 SDK 설치: `dotnet-sdk-5.0`
- .NET Core 3.1 SDK 설치: `dotnet-sdk-3.1`

> **참고 항목**
>
> 일부 패키지는 Linux 배포판에서 사용할 수 없을 수 있습니다.

패키지가 없음

패키지-버전 조합이 작동하지 않는다면 사용할 수 없는 것입니다. 예를 들어 ASP.NET Core SDK가 없습니다. ASP.NET Core SDK 구성 요소는 .NET SDK에 포함됩니다. 값 `aspnetcore-sdk-8.0`는 올바르지 않으며, 올바른 값은 `dotnet-sdk-8.0`입니다. .NET 지원하는 Linux 배포 목록은 .NET 종속성 및 요구 사항 참조하세요.

패키지 관리자 문제 해결

이 섹션에서는 패키지 관리자를 사용하여 .NET 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

패키지를 찾을 수 없음

> ⓘ Important
>
> **Microsoft 패키지 피드**는 .NET 버전에 따라 다른 아키텍처에 대한 패키지를 게시합니다.
>
> - **.NET 10**: **x64** 및 **Arm64** 패키지만 해당합니다.
> - **.NET 9**: **x64** 전용 패키지입니다.
> - **.NET 8**: **x64** 패키지만.
>
> **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

가져오지 못함

.NET 패키지를 설치하는 동안 `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 유사한 오류가 표시 될 수 있습니다. 일반적으로 이 오류는 .NET 대한 패키지 피드가 최신 패키지 버전으로 업그레이드되고 있으며 나중에 다시 시도해야 한다는 것을 의미합니다. 업그레이드하는 동안 2시간 이상 패키지 피드를 사용할 수 없습니다. 이 오류 메시지가 2시간 이상 계속 표시되는 경우 https://github.com/dotnet/core/issues 에서 문제를 제출하세요.

종속성

패키지 관리자를 설치할 때 이러한 라이브러리가 설치됩니다. 그러나 .NET 수동으로 설치하거나 자체 포함된 앱을 게시하는 경우 다음 라이브러리가 설치되어 있는지 확인해야 합니다.

- krb5
- libicu
- libopenssl3(OpenSSL 3.x)

> **ⓘ Important**
>
> .NET 8부터 openSUSE에 대한 .NET 패키지는 OpenSSL 3.x(libopenssl3)에 따라 달라집니다. 이 변경 내용은 .NET 6 및 .NET 7 패키지에도 적용됩니다. 자세한 내용은 **openSUSE 및 SLES에 대한 .NET 패키지는 OpenSSL 3.x에 의존합니다**를 참조하세요.

종속성은 `zypper install` 명령을 사용하여 설치할 수 있습니다. 다음 코드 조각은 `krb5` 라이브러리의 설치를 보여 줍니다.

Bash

```
sudo zypper install krb5
```

종속성에 대한 자세한 내용은 Self-contained Linux apps (자체 포함 Linux 앱)를 참조하세요.

다음 단계

- .NET CLI 개요
- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 06.

RHEL 및 CentOS Stream에 .NET SDK 또는 .NET 런타임 설치

.NET RHEL(Red Hat Enterprise Linux)에서 지원됩니다. 이 문서에서는 RHEL 및 CentOS Stream에 .NET 설치하는 방법을 설명합니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

Red Hat 구독 등록

RHEL의 Red Hat에서 .NET 설치하려면 먼저 Red Hat 구독 관리자를 사용하여 등록해야 합니다. 시스템에서 이 작업이 수행되지 않았거나 확실하지 않은 경우 .NETRed Hat 제품 설명서를 참조하세요>.

> **Important**
>
> 이전 문은 CentOS Stream에 적용되지 않습니다.

지원되는 배포

다음 표는 RHEL 및 CentOS Stream 모두에서 현재 지원되는 .NET 릴리스 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 Linux 배포가 더 이상 지원되지 않을 때까지 계속 지원됩니다.

테이블 확장

배포	.NET
RHEL 10	10, 9, 8
RHEL 9	10, 9, 8
RHEL 8	10, 9, 8
CentOS Stream 10	10, 9, 8
CentOS Stream 9	10, 9, 8

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

RHEL 10

.NET RHEL 10의 AppStream 리포지토리 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

RHEL 9

.NET RHEL 9용 AppStream 리포지토리 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

RHEL 8

.NET RHEL 8용 AppStream 리포지토리 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

CentOS Stream 10

.NET CentOS Stream 10용 AppStream 리포지토리에 포함됩니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

CentOS Stream 9

.NET CentOS Stream 9의 AppStream 리포지토리에 포함되어 있습니다.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo dnf install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo dnf install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo dnf install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

CentOS Linux는 어디에 있나요?

.NET CentOS Linux에서 더 이상 지원되지 않습니다. 2024년 6월 30일을 기준으로 CentOS Linux는 수명이 다했습니다. 자세한 내용은 CentOS Stream 8 및 CentOS Linux 7의 종료 날짜 를 참조하세요.

종속성

RHEL 및 CentOS Stream에서 .NET 실행하려면 다음 라이브러리가 필요합니다. 패키지 관리자를 `dnf` 사용하여 설치합니다.

- glibc
- libgcc
- CA 인증서
- openssl-libs
- libstdc++
- libicu
- tzdata
- krb5-libs
- zlib(.NET 8에만 필요)

예를 들어 모든 종속성을 설치하려면 다음을 수행합니다.

Bash

```
sudo dnf install glibc libgcc ca-certificates openssl-libs libstdc++ libicu tzdata
krb5-libs
```

.NET 8의 경우 다음을 설치합니다.

Bash

```
sudo dnf install zlib
```

다른 버전을 설치하는 방법

.NET의 다른 릴리스를 설치하는 데 필요한 단계를 알아보려면 Red Hat의 .NET 문서를 참조하세요.

패키지 관리자 문제 해결

이 섹션에서는 패키지 관리자를 사용하여 .NET 또는 .NET Core를 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

누락된 `fxr`, `libhostfxr.so` 또는 `FrameworkList.xml` 관련 오류

이 문제를 해결하는 방법에 대한 자세한 내용은 fxr, libhostfxr.so, 및 FrameworkList.xml 오류를 해결하는 방법을 참조하세요.

다음 단계

- .NET CLI 개요

- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 06.

SLES에 .NET SDK 또는 .NET 런타임 설치

.NET SLES(SUSE Enterprise Linux)에서 지원됩니다. 이 문서에서는 SLES에 .NET 설치하는 방법을 설명합니다.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

지원되는 배포

다음 표는 SLES에서 현재 지원되는 .NET 릴리스 목록입니다. 이러한 버전은 .NET 버전이 지원 종료 또는 SLES 버전이 더 이상 지원되지 않을 때까지 계속 지원됩니다.

테이블 확장

SLES	.NET
16.0	10, 9, 8
15.7	10, 9, 8
15.6	10, 9, 8

다음 버전의 .NET ❌ 더 이상 지원되지 않습니다.

- .NET 7
- .NET 6
- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

미리 보기 버전 설치

패키지 리포지토리에서는 .NET 미리 보기 및 릴리스 후보 버전을 사용할 수 없습니다. 다음 방법 중 하나로 .NET 미리 보기 및 릴리스 후보를 설치할 수 있습니다.

- *install-dotnet.sh* 스크립트를 통한 설치
- 수동 이진 추출

미리 보기 버전 제거

패키지 관리자를 사용하여 .NET 설치를 관리하는 경우 이전에 미리 보기 릴리스를 설치한 경우 충돌이 발생할 수 있습니다. 패키지 관리자는 미리 보기가 아닌 릴리스를 .NET의 이전 버전으로 해석할 수 있습니다. 미리 보기가 아닌 릴리스를 설치하려면 먼저 미리 보기 버전을 제거합니다. .NET 제거하는 방법에 대한 자세한 내용은 .NET 런타임 및 SDK를 제거하는 방법 참조하세요.

SLES 16

.NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 Microsoft 패키지 리포지토리를 추가합니다. 터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/16/packages-microsoft-prod.rpm
```

.NET 10

> ⓘ Important
>
> **Microsoft 패키지 피드**는 .NET 10에 대한 **x64** 및 **Arm64** 패키지만 게시합니다. **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo zypper install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo zypper install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo zypper install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

SLES 15

.NET 설치하기 전에 다음 명령을 실행하여 신뢰할 수 있는 키 목록에 Microsoft 패키지 서명 키를 추가하고 Microsoft 패키지 리포지토리를 추가합니다. 터미널을 열고 다음 명령을 실행합니다.

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/15/packages-microsoft-prod.rpm
```

현재 SLES 15 Microsoft 리포지토리 설치 패키지는 `microsoft-prod.repo` 파일을 잘못된 디렉터리에 설치하므로 zypper가 .NET 패키지를 찾을 수 없습니다. 이 문제를 해결하려면 올바른 디렉터리에 symlink를 만듭니다.

Bash

```
sudo ln -s /etc/yum.repos.d/microsoft-prod.repo /etc/zypp/repos.d/microsoft-prod.repo
```

.NET 10

> ⓘ Important
>
> **Microsoft 패키지 피드**는 .NET 10에 대한 **x64** 및 **Arm64** 패키지만 게시합니다. **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

SDK 설치

.NET SDK를 사용하면 .NET 사용하여 앱을 개발할 수 있습니다. .NET SDK를 설치하는 경우 해당 런타임을 설치할 필요가 없습니다. .NET SDK를 설치하려면 다음 명령을 실행합니다.

Bash

```
sudo zypper install dotnet-sdk-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

런타임 설치

ASP.NET Core 런타임을 사용하면 런타임을 제공하지 않은 .NET 사용하여 만든 앱을 실행할 수 있습니다. 다음 명령은 .NET 가장 호환되는 런타임인 ASP.NET Core 런타임을 설치합니다. 터미널에서 다음 명령을 실행합니다.

Bash

```
sudo zypper install aspnetcore-runtime-10.0
```

ASP.NET Core 런타임 대신 ASP.NET Core 지원을 포함하지 않는 .NET 런타임을 설치할 수 있습니다. 이전 명령의 `aspnetcore-runtime-10.0` `dotnet-runtime-10.0` 바꿉니다.

Bash

```
sudo zypper install dotnet-runtime-10.0
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

다른 버전을 설치하는 방법

모든 버전의 .NET은 https://dotnet.microsoft.com/download/dotnet 에서 다운로드할 수 있지만, 수동 설치가 필요합니다. 패키지 관리자를 사용하여 다른 버전의 .NET 설치할 수 있습니다. 그러나 요청된 버전을 사용하지 못할 수도 있습니다.

패키지 관리자 피드에 추가되는 패키지는 해킹 가능한 형식으로 명명됩니다(예: `{product}-{type}-{version}`).

- **제품**
  설치할 .NET 제품의 유형입니다. 유효한 옵션은 다음과 같습니다.
  - `dotnet`
  - `aspnetcore`

- **type**
  SDK와 런타임 중 선택합니다. 유효한 옵션은 다음과 같습니다.
  - `sdk` (**dotnet** 제품에만 사용 가능)
  - `runtime`

- **version**
  설치한 SDK 또는 런타임의 버전입니다. 유효한 옵션은 모든 릴리스된 버전입니다. 예:
  - `9.0`
  - `8.0`
  - `3.1`
  - `2.1`

  다운로드하려는 SDK/런타임을 Linux 배포판에서 사용할 수 없을 수 있습니다. 지원되는 배포 목록은 Linux에서 .NET 설치 참조하세요.

예제

- ASP.NET Core 9.0 런타임 설치: `aspnetcore-runtime-9.0`
- .NET Core 2.1 런타임 설치: `dotnet-runtime-2.1`
- .NET 5 SDK 설치: `dotnet-sdk-5.0`
- .NET Core 3.1 SDK 설치: `dotnet-sdk-3.1`

> ⓘ **참고 항목**
>
> 일부 패키지는 Linux 배포판에서 사용할 수 없을 수 있습니다.

패키지가 없음

패키지-버전 조합이 작동하지 않는다면 사용할 수 없는 것입니다. 예를 들어 ASP.NET Core SDK가 없습니다. ASP.NET Core SDK 구성 요소는 .NET SDK에 포함됩니다. 값 `aspnetcore-sdk-8.0`는 올바르지 않으며, 올바른 값은 `dotnet-sdk-8.0`입니다. .NET 지원하는 Linux 배포 목록은 .NET 종속성 및 요구 사항 참조하세요.

패키지 관리자 문제 해결

이 섹션에서는 패키지 관리자를 사용하여 .NET 설치하는 동안 발생할 수 있는 일반적인 오류에 대한 정보를 제공합니다.

패키지를 찾을 수 없음

> ⓘ **Important**
>
> **Microsoft 패키지 피드**는 .NET 버전에 따라 다른 아키텍처에 대한 패키지를 게시합니다.
>
> - **.NET 10**: **x64** 및 **Arm64** 패키지만 해당합니다.
> - **.NET 9**: **x64** 전용 패키지입니다.
> - **.NET 8**: **x64** 패키지만.
>
> **Arm32** 같은 다른 아키텍처에 .NET 설치해야 하는 경우 Microsoft 패키지 피드와 함께 패키지 관리자를 사용하지 마세요. .NET을 패키지 관리자 **없이 설치하는 방법**에 대한 자세한 내용은 다음 문서 중 하나를 참조하세요.
>
> - install-dotnet 스크립트를 사용하여 .NET를 설치합니다.
> - .NET.

가져오지 못함

.NET 패키지를 설치하는 동안 `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 유사한 오류가 표시 될 수 있습니다. 일반적으로 이 오류는 .NET 대한 패키지 피드가 최신 패키지 버전으로 업그레이드되고 있으며 나중에 다시 시

도해야 한다는 것을 의미합니다. 업그레이드하는 동안 2시간 이상 패키지 피드를 사용할 수 없습니다. 이 오류 메시지가 2시간 이상 계속 표시되는 경우 https://github.com/dotnet/core/issues 에서 문제를 제출하세요.

종속성

패키지 관리자를 설치할 때 이러한 라이브러리가 설치됩니다. 그러나 .NET 수동으로 설치하거나 자체 포함된 앱을 게시하는 경우 다음 라이브러리가 설치되어 있는지 확인해야 합니다.

- krb5
- libicu
- libopenssl3(OpenSSL 3.x)

> **Important**
>
> .NET 8부터 SLES용 .NET 패키지는 OpenSSL 3.x(libopenssl3)에 따라 달라집니다. 이 변경 내용은 .NET 6 및 .NET 7 패키지에도 적용됩니다. 자세한 내용은 **openSUSE 및 SLES에 대한 .NET 패키지는 OpenSSL 3.x에 의존합니다**를 참조하세요.

종속성은 `zypper install` 명령을 사용하여 설치할 수 있습니다. 다음 코드 조각은 `krb5` 라이브러리의 설치를 보여 줍니다.

Bash

```
sudo zypper install krb5
```

종속성에 대한 자세한 내용은 Self-contained Linux apps (자체 포함 Linux 앱)를 참조하세요.

다음 단계

- .NET CLI 개요
- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 12.

설치 스크립트를 사용하거나 이진 파일을 추출하여 Linux에 .NET 설치

이 문서에서는 설치 스크립트를 사용하거나 이진 파일을 추출하여 linux에서 .NET SDK 또는 .NET 런타임을 설치하는 방법을 보여 줍니다. 기본 제공 패키지 관리자를 지원하는 배포 목록은 Linux에서 .NET 설치를 참조하세요.

.NET 앱을 개발하려는 경우 SDK(런타임 포함)를 설치합니다. 또는 앱을 실행하기만 하려면 런타임을 설치합니다. 런타임을 설치하는 경우 .NET 및 ASP.NET Core 런타임을 모두 포함하므로 **ASP.NET Core 런타임** 설치하는 것이 좋습니다.

`dotnet --list-sdks` 및 `dotnet --list-runtimes` 명령을 사용하여 설치된 버전을 확인합니다. 자세한 내용은 .NET 이미 설치되어 있는지 확인하는 방법 참조하세요.

.NET 릴리스

지원되는 릴리스에는 LTS(장기 지원) 및 STS(표준 기간 지원)의 두 가지 유형이 있습니다. 모든 릴리스의 품질은 동일합니다. 유일한 차이점은 지원 기간입니다. LTS 릴리스는 3년 동안 무료 지원 및 패치를 받습니다. STS 릴리스는 2년 동안 무료 지원 및 패치를 받습니다. 자세한 내용은 .NET 지원 정책 참조하세요.

다음 표에서는 각 버전의 .NET(및 .NET Core)의 지원 상태를 나열합니다.

테이블 확장

✔ 지원됨	✘ 지원 부족
10(LTS)	7
9(STS)	6
8(LTS)	5
	3.1
	3.0
	2.2
	2.1
	2.0
	1.1
	1.0

종속성

.NET을 설치할 때, 수동으로 설치하는 경우와 같이 특정 종속성이 설치되지 않을 수 있습니다. 다음 목록에서는 Microsoft에서 지원되고 설치해야 할 수 있는 종속성이 있는 Linux 배포판에 대해 자세히 설명합니다. 자세한 내용은 배포 페이지를 확인하세요.

- 알파인
- Debian
- 페도라
- RHEL 및 CentOS Stream
- SLES
- Ubuntu

종속성에 대한 일반적인 내용은 Self-contained Linux apps (자체 포함 Linux 앱)를 참조하세요.

RPM 종속성

배포가 이전에 나열되지 않았고 RPM 기반인 경우 다음 종속성이 필요할 수 있습니다.

- glibc
- libgcc
- CA 인증서
- openssl-libs
- libstdc++
- libicu
- tzdata
- krb5-libs

DEB 종속성

배포가 이전에 나열되지 않았고 debian 기반인 경우 다음 종속성이 필요할 수 있습니다.

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu70
- libssl3
- libstdc++6
- zlib1g

스크립팅된 설치

dotnet-install 스크립트는 자동화와 **SDK** 및 **런타임**의 관리자 설치가 아닌 일반 설치를 수행하는 데 사용됩니다. https://dot.net/v1/dotnet-install.sh 스크립트를 다운로드할 수 있습니다. 이러한 방식으로 .NET 설치되는 경우 Linux 배포에 필요한 종속성을 설치해야 합니다. 특정 Linux 배포판에 맞는 링크는 Linux에 .NET 설치 문서를 참조하세요.

> **ⓘ Important**
>
> 스크립트를 실행하려면 Bash가 필요합니다.

`wget`을 사용하여 스크립트를 다운로드할 수 있습니다.

Bash

```
wget https://dot.net/v1/dotnet-install.sh -O dotnet-install.sh
```

또는 `curl`를 사용하여:

Bash

```
curl -L https://dot.net/v1/dotnet-install.sh -o dotnet-install.sh
```

이 스크립트를 실행하기 전에 이 스크립트를 실행 파일로 실행할 수 있는 권한을 부여해야 합니다.

Bash

```
chmod +x ./dotnet-install.sh
```

스크립트는 기본적으로 .NET 10인 최신 long term support(LTS) SDK 버전을 설치합니다. (LTS) 버전이 아닐 수도 있는 최신 릴리스를 설치하려면 `--version latest` 매개 변수를 사용합니다.

Bash

```
./dotnet-install.sh --version latest
```

SDK 대신 .NET 런타임을 설치하려면 `--runtime` 매개 변수를 사용합니다.

Bash

```
./dotnet-install.sh --version latest --runtime aspnetcore
```

특정 버전을 나타내는 `--channel` 매개 변수를 사용하여 특정 주 버전을 설치할 수 있습니다. 다음 명령은 .NET 9.0 SDK를 설치합니다.

Bash

```
./dotnet-install.sh --channel 9.0
```

자세한 내용은 dotnet-install 스크립트 참조를 참조하세요.

명령줄에서 .NET 사용하도록 설정하려면 시스템 전체 환경 변수를 설정합니다.

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

수동 설치

패키지 관리자의 대안으로, SDK와 런타임을 다운로드하여 수동으로 설치할 수 있습니다. 수동 설치는 일반적으로 연속 통합 테스트의 일부로서 사용되거나 지원되지 않는 Linux 배포판에서 사용됩니다. 개발자 또는 사용자의 경우 패키지 관리자를 사용하는 것이 좋습니다.

다음 사이트 중 한 곳에서 SDK 또는 런타임의 **binary** 릴리스를 다운로드합니다. .NET SDK에는 해당 런타임이 포함됩니다.

- ✔ .NET 10 다운로드
- ✔ .NET 9 다운로드
- ✔ .NET 8 다운로드
- 모든 .NET Core 다운로드

다운로드한 파일을 추출하고 `export` 명령을 사용하여 `DOTNET_ROOT` 추출된 폴더의 위치로 설정한 다음 .NET PATH에 있는지 확인합니다. `DOTNET_ROOT` 내보내면 터미널에서 .NET CLI 명령을 사용할 수 있습니다. .NET 환경 변수에 대한 자세한 내용은 .NET SDK 및 CLI 환경 변수 참조하세요.

서로 다른 버전의 .NET 동일한 폴더로 추출할 수 있습니다. 이 폴더는 나란히 공존합니다.

예시

다음 명령은 Bash를 사용하여 환경 변수 `DOTNET_ROOT`를 현재 작업 디렉터리로 설정하고 다음으로 `.dotnet`으로 설정합니다. 해당 디렉터리가 존재하지 않을 경우 생성됩니다. `DOTNET_FILE` 환경 변수는 설치하려는 .NET 이진 릴리스의 파일 이름입니다. 이 파일은 `DOTNET_ROOT` 디렉터리로 추출됩니다. `DOTNET_ROOT` 디렉터리와 해당 `tools` 하위 디렉터리가 모두 `PATH` 환경 변수에 추가됩니다.

> ⓘ **Important**

이러한 명령을 실행하는 경우 `DOTNET_FILE` 값을 다운로드한 .NET 이진 파일의 이름으로 변경해야 합니다.

Bash

```
DOTNET_FILE=dotnet-sdk-9.0.306-linux-x64.tar.gz
export DOTNET_ROOT=$(pwd)/.dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"

export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

동일한 폴더에 둘 이상의 .NET 버전을 설치할 수 있습니다.

`HOME` 변수 또는 `~` 경로로 식별되는 홈 디렉터리에 .NET 설치할 수도 있습니다.

Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

다운로드한 이진 파일 확인

설치 프로그램 또는 이진 릴리스를 다운로드한 후 파일이 변경되거나 손상되지 않았는지 확인합니다. 컴퓨터에서 체크섬을 확인한 다음 다운로드 웹 사이트에 보고된 내용과 비교할 수 있습니다.

공식 다운로드 페이지에서 파일을 다운로드하면 해당 파일의 체크섬이 텍스트 상자에 표시됩니다. 체크섬 값을 클립보드에 복사하려면 **복사** 단추를 선택합니다.

`sha512sum` 명령을 사용하여 다운로드한 파일의 체크섬을 인쇄합니다. 예를 들어 다음 명령은 *dotnet-sdk-9.0.306-linux-x64.tar.gz* 파일의 체크섬을 보고합니다.

Bash

```
$ sha512sum dotnet-sdk-9.0.306-linux-x64.tar.gz
bbb6bdc3c8048e7cc189759b406257839e7d4bd6b8b1ba4bcdaeea8f92340e6855231043dd73f902130
ca5357af72b810bb51a4da4d1315a2927ff85f831f1d5  dotnet-sdk-9.0.306-linux-x64.tar.gz
```

체크섬을 다운로드 사이트에서 제공한 값과 비교합니다.

체크섬 파일을 사용하여 유효성 검사

.NET 릴리스 정보에는 다운로드한 파일의 유효성을 검사하는 데 사용할 수 있는 체크섬 파일에 대한 링크가 포함되어 있습니다. 다음 단계에서는 체크섬 파일을 다운로드하고 .NET 설치 이진 파일의 유효성을 검사하는 방법을 설명합니다.

1. https://github.com/dotnet/core/tree/main/release-notes/9.0#releases GitHub .NET 9의 릴리스 정보 페이지에는 **Releases** 섹션이 포함되어 있습니다. 해당 섹션의 표는 각 .NET 9 릴리스의 다운로드 및 체크섬 파일에 연결됩니다. 다음 이미지는 .NET 8 릴리스 테이블을 참조로 보여줍니다.

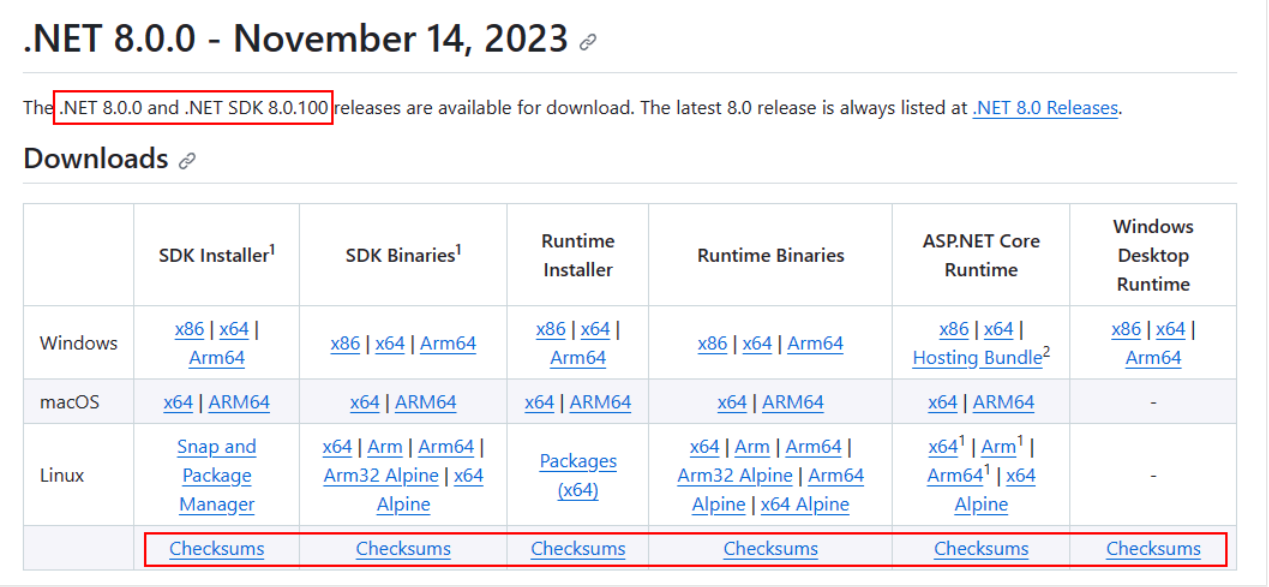
   
   Releases

Date	Release
2023/11/14	8.0.0
2023/10/10	8.0.0 RC 2
2023/09/12	8.0.0 RC 1
2023/08/08	8.0.0 Preview 7
2023/07/11	8.0.0 Preview 6

2. 다운로드한 .NET 버전의 링크를 선택합니다.

   이전 섹션에서는 .NET 9.0.10 릴리스의 .NET SDK 9.0.306을 사용했습니다.

3. 릴리스 페이지에서 .NET 런타임 및 .NET SDK 버전과 체크섬 파일에 대한 링크를 볼 수 있습니다. 다음 이미지는 .NET 8 릴리스 테이블을 참조로 보여줍니다.



.NET 8.0.0 - November 14, 2023

The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at .NET 8.0 Releases.

Downloads

	SDK Installer[1]	SDK Binaries[1]	Runtime Installer	Runtime Binaries	ASP.NET Core Runtime	Windows Desktop Runtime
Windows	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Arm64	x86 | x64 | Hosting Bundle[2]	x86 | x64 | Arm64
macOS	x64 | ARM64	x64 | ARM64	x64 | ARM64	x64 | ARM64	x64 | ARM64	-
Linux	Snap and Package Manager	x64 | Arm | Arm64 | Arm32 Alpine | x64 Alpine	Packages (x64)	x64 | Arm | Arm64 | Arm32 Alpine | Arm64 Alpine | x64 Alpine	x64[1] | Arm[1] | Arm64[1] | x64 Alpine	-
	Checksums	Checksums	Checksums	Checksums	Checksums	Checksums

4. **체크섬** 링크를 마우스 오른쪽 단추로 클릭하고 클립보드에 복사합니다.

5. 터미널을 엽니다.

6. `curl -O {link}`를 사용하여 체크섬 파일을 다운로드합니다.

   다음 명령의 링크를 복사한 링크로 바꿉니다.

   Bash

   ```
   curl -O https://builds.dotnet.microsoft.com/dotnet/checksums/9.0.10-sha.txt
   ```

7. 체크섬 파일과 .NET 릴리스 파일이 모두 동일한 디렉터리에 다운로드되면 `sha512sum -c {file} --ignore-missing` 명령을 사용하여 다운로드한 파일의 유효성을 검사합니다.

   유효성 검사를 통과하면 **확인** 상태로 인쇄된 파일이 표시됩니다.

   Bash

   ```
   $ sha512sum -c 9.0.10-sha.txt --ignore-missing
   dotnet-sdk-9.0.306-linux-x64.tar.gz: OK
   ```

   FAILED로 표시된 파일이 표시되면 다운로드한 파일이 유효하지 않으므로 사용하면 안 됩니다.

   Bash

   ```
   $ sha512sum -c 9.0.10-sha.txt --ignore-missing
   dotnet-sdk-9.0.306-linux-x64.tar.gz: FAILED
   sha512sum: WARNING: 1 computed checksum did NOT match
   sha512sum: 9.0.10-sha.txt: no file was verified
   ```

시스템 전체 환경 변수 설정

이전 설치 스크립트를 사용한 경우 설정된 변수는 현재 터미널 세션에만 적용됩니다. 셸 프로필에 추가합니다. Linux에서는 다양한 셸을 사용할 수 있으며 각 셸에는 서로 다른 프로필이 있습니다. 예시:

- **Bash 셸**: *~/.bash_profile* 또는 *~/.bashrc*
- ko-KR: **Korn 셸**: *~/.kshrc* 또는 *.profile*
- **Z 셸**: *~/.zshrc* 또는 *.zprofile*

셸 프로필에서 다음 두 환경 변수를 설정합니다.

- `DOTNET_ROOT`

  이 변수는 .NET 설치된 폴더(예: `$HOME/.dotnet`)로 설정됩니다.

  ```
  Bash
  export DOTNET_ROOT=$HOME/.dotnet
  ```

- `PATH`

  이 변수에는 `DOTNET_ROOT` 폴더와 `DOTNET_ROOT/tools` 폴더가 모두 포함되어야 합니다.

  ```
  Bash
  export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
  ```

다음 단계

- .NET CLI 개요
- .NET CLI에 TAB 완성을 사용하도록 설정하는 방법
- Tutorial: .NET

Last updated on 2026. 03. 11.

스냅을 사용하여 .NET 런타임 설치

이 문서에서는 .NET 런타임 스냅 패키지를 설치하는 방법을 설명합니다. .NET 런타임 스냅 패키지는 Canonical에서 제공하고 유지 관리합니다. Snap은 Linux 배포에 기본 제공되는 패키지 관리자의 좋은 대안입니다. SDK를 설치해야 하는 경우 Snap을 사용하여 .NET SDK 설치를 참조하세요.

스냅은 다양한 Linux 배포판에서 작동하는 앱과 해당 종속성의 번들입니다. Snap은 Snap 스토어에서 검색해서 설치할 수 있습니다. 스냅에 대한 자세한 내용은 시작하기를 참조하세요.

> ⊗ **주의**
>
> .NET 스냅 설치에서 **.NET 도구**를 실행하는 데 문제가 있을 수 있습니다. .NET 도구를 사용하려는 경우 **dotnet-install 스크립트** 또는 특정 Linux 배포에 대한 패키지 관리자를 사용하여 .NET을 설치하는 것이 좋습니다.

필수 조건

- 스냅을 지원하는 Linux 배포판.
- `snapd` 스냅 디먼.

Linux 배포판에 스냅이 이미 포함되어 있을 수 있습니다. 터미널에서 `snap`을 실행하여 명령이 작동하는지 확인하세요. 지원되는 Linux 배포판 목록 및 스냅을 설치하는 방법에 대한 지침은 설치snapd를 참조하세요.

.NET 릴리스

Microsoft는 LTS(장기 지원) 및 STS(표준 기간 지원)라는 두 가지 지원 정책에 따라 .NET을 게시합니다. 모든 릴리스의 품질은 동일합니다. 유일한 차이점은 지원 기간입니다. LTS 릴리스는 3년 동안 무료 지원과 패치를 가져옵니다. STS 릴리스는 2년 동안 무료 지원 및 패치를 받습니다. 자세한 내용은 .NET 지원 정책을 참조하세요.

현재 Microsoft에서 지원하는 .NET 버전은 다음과 같습니다.

- 10.0(LTS) - 지원은 **2028년 11월 14**일에 종료됩니다.
- 9.0(STS) - 지원은 **2026년 11월 10**일에 종료됩니다.
- 8.0(LTS) - **2026년 11월 10일**에 지원이 종료됩니다.

.NET을 빌드하고 릴리스하는 다른 엔터티는 다른 지원 정책을 도입할 수 있습니다. .NET 지원 방식을 이해하기 위해 그들과 확인해야 합니다.

1. 런타임 설치

다음 단계에서는 .NET 9 런타임 스냅 패키지를 설치합니다.

1. 터미널을 엽니다.
2. `snap install`을 사용하여 .NET SDK 런타임 패키지를 설치합니다. 예를 들어 다음 명령은 .NET 9 런타임을 설치합니다.

```
Bash
sudo snap install dotnet-runtime-90
```

각 .NET 런타임은 개별 스냅 패키지로 게시됩니다. 다음 표에는 패키지가 나열되어 있습니다.

테이블 확장

.NET 버전	Snap 패키지	Microsoft에서 지원하는 .NET 버전
10(LTS)	`dotnet-runtime-100`	예
STS(9)	`dotnet-runtime-90`	예
8(LTS)	`dotnet-runtime-80`	예
7(STS)	`dotnet-runtime-70`	아니요
6(LTS)	`dotnet-runtime-60`	아니요
5	`dotnet-runtime-50`	아니요
3.1	`dotnet-runtime-31`	아니요
3.0	`dotnet-runtime-30`	아니요
2.2	`dotnet-runtime-22`	아니요
2.1	`dotnet-runtime-21`	아니요

2. dotnet 명령 사용

.NET 런타임 스냅 패키지가 설치되면 `dotnet` 명령이 자동으로 구성되지 않습니다. `snap alias` 명령을 사용하여 터미널에서 `dotnet` 명령을 사용합니다. 이 명령의 형식은 `sudo snap alias {package}.{command} {alias}`로 지정됩니다. 다음 예는 `dotnet` 명령을 매핑합니다.

```
Bash
```

```
sudo snap alias dotnet-runtime-90.dotnet dotnet
```

3. 설치 위치 내보내기

`DOTNET_ROOT` 환경 변수는 도구에서 .NET이 설치된 위치를 확인하는 데 자주 사용됩니다. .NET이 Snap을 통해 설치되면 이 환경 변수는 구성되지 않습니다. 프로필에서 *DOTNET_ROOT* 환경 변수를 구성해야 합니다. 스냅 경로는 `/snap/{package}/current` 형식을 사용합니다. 예를 들어 `dotnet-runtime-90` 스냅을 설치한 경우 다음 명령을 사용하여 환경 변수를 .NET이 있는 위치로 설정합니다.

Bash

```
export DOTNET_ROOT=/snap/dotnet-runtime-90/current
```

환경 변수를 영구적으로 내보내기

앞의 `export` 명령은 실행된 터미널 세션에 대한 환경 변수만 설정합니다.

셸 프로필을 편집하여 명령을 영구적으로 추가할 수 있습니다. 몇 가지 Linux용 셸이 있으며, 각각 다른 프로필을 갖습니다. 예를 들어:

- **Bash 셸**: *~/.bash_profile*, *~/.bashrc*
- **Korn 셸**: *~/.kshrc* 또는 *.profile*
- **Z 셸**: *~/.zshrc* 또는 *.zprofile*

셸에 적절한 원본 파일을 편집하고 `export DOTNET_ROOT=/snap/dotnet-runtime-90/current`를 추가합니다.

문제 해결

- dotnet 터미널 명령이 작동하지 않음
- WSL2에서 스냅을 설치할 수 없음

dotnet 터미널 명령이 작동하지 않음

스냅 패키지는 패키지에서 제공하는 명령에 별칭을 매핑할 수 있습니다. .NET 런타임 스냅 패키지는 `dotnet` 명령과 자동으로 연결되지 않습니다. `dotnet` 명령의 별칭을 스냅 패키지에 지정하려면 다음 명령을 사용합니다.

Bash

```
sudo snap alias dotnet-runtime-90.dotnet dotnet
```

`dotnet-runtime-90`을 런타임 패키지의 이름으로 대체합니다.

WSL2에서 스냅을 설치할 수 없음

스냅을 설치하려면 먼저 WSL2 인스턴스에서 `systemd`를 활성화해야 합니다.

1. 선택한 텍스트 편집기에서 `/etc/wsl.conf`를 엽니다.
2. 다음 구성을 붙여넣으세요.

   ini

   ```ini
   [boot]
   systemd=true
   ```

3. 파일을 저장하고 PowerShell을 통해 WSL2 인스턴스를 다시 시작합니다. `wsl.exe --shutdown` 명령 사용

4. .NET CLI 사용

터미널을 열고 `dotnet`을 입력합니다.

.NET CLI

```
dotnet
```

다음과 유사한 출력이 표시됩니다.

출력

```
Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help         Display help.
  --info            Display .NET information.
  --list-sdks       Display the installed SDKs.
  --list-runtimes   Display the installed runtimes.

path-to-application:
  The path to an application .dll file to execute.
```

.NET CLI 사용 방법을 알아보려면 .NET CLI 개요를 참조하세요.

관련 콘텐츠

- .NET CLI 개요
- .NET CLI에 대해 탭 완성 기능을 사용하도록 설정하는 방법입니다.

Last updated on 2026. 03. 16.

Snap을 사용하여 .NET SDK 설치

이 문서에서는 .NET SDK 스냅 패키지를 설치하는 방법을 설명합니다. .NET SDK 스냅 패키지는 Canonical에서 제공하고 유지 관리합니다. Snap은 Linux 배포에 기본 제공되는 패키지 관리자의 좋은 대안입니다.

스냅은 다양한 Linux 배포판에서 작동하는 앱과 해당 종속성의 번들입니다. Snap은 Snap 스토어에서 검색해서 설치할 수 있습니다. Snap에 대한 자세한 내용은 빠른 시작 둘러보기를 참조하세요.

> ⊗ **주의**
>
> .NET 스냅 설치 시 .NET 도구 실행하는 데 문제가 있을 수 있습니다. .NET 도구를 사용하려는 경우 dotnet-install 스크립트 또는 특정 Linux 배포에 대한 패키지 관리자를 사용하여 .NET 설치하는 것이 좋습니다.
>
> snap을 통해 .NET 설치될 때 `dotnet watch` 명령이 작동하지 않는 알려진 문제입니다.
>
> .NET 도구 또는 `dotnet watch` 명령을 사용하려는 경우 dotnet-install 스크립트를 사용하여 .NET 설치하는 것이 좋습니다.

필수 조건

- 스냅을 지원하는 Linux 배포판.
- `snapd` 스냅 디먼.

Linux 배포판에 스냅이 이미 포함되어 있을 수 있습니다. 터미널에서 `snap`을 실행하여 명령이 작동하는지 확인하세요. 지원되는 Linux 배포판 목록 및 스냅을 설치하는 방법에 대한 지침은 설치snapd를 참조하세요.

.NET 릴리스

Microsoft는 두 가지 지원 정책인 LTS(장기 지원) 및 STS(표준 기간 지원)에 따라 .NET 게시합니다. 모든 릴리스의 품질은 동일합니다. 유일한 차이점은 지원 기간입니다. LTS 릴리스는 3년 동안 무료 지원과 패치를 가져옵니다. STS 릴리스는 2년 동안 무료 지원 및 패치를 받습니다. 자세한 내용은 .NET 지원 정책 참조하세요.

현재 Microsoft에서 지원되는 .NET 버전은 다음과 같습니다.

- 10.0(LTS) - 지원은 **2028년 11월 14**일에 종료됩니다.
- 9.0(STS) - 지원은 **2026년 11월 10**일에 종료됩니다.

- 8.0(LTS) - **2026년 11월 10일**에 지원이 종료됩니다.

.NET 빌드하고 릴리스하는 다른 엔터티는 다른 지원 정책을 도입할 수 있습니다. .NET 지원되는 방법을 이해하려면 해당 사용자와 함께 확인해야 합니다.

1. SDK 설치

> **ⓘ Important**
>
> .NET 10은 2025년 11월 11일에 릴리스되었습니다. 패키지 관리자 피드에 패키지가 표시되거나 특정 Linux 배포판에 패키지가 포함되는 데 시간이 걸릴 수 있습니다.

.NET 9부터 .NET SDK에 대한 스냅 패키지는 버전별 식별자(예: .NET 9의 경우 `dotnet-sdk-90` 및 .NET 10의 경우 `dotnet-sdk-100`)에 게시됩니다. .NET 9 이전에는 모든 SDK 버전이 동일한 식별자 `dotnet-sdk` 아래에 게시되었으며 채널을 통해 버전을 지정했습니다. 또한 .NET 9 이상 스냅 패키지는 x64 및 Arm64 아키텍처를 모두 지원하지만 이전 버전은 x64만 지원합니다. SDK에는 SDK에 버전이 지정된 ASP.NET Core 및 .NET 런타임이 모두 포함됩니다.

> **팁**
>
> Snapcraft .NET SDK 패키지 페이지 에는 Snapcraft 및 .NET 설치하는 방법에 대한 배포 관련 지침이 포함되어 있습니다.

1. 터미널을 엽니다.
2. `snap install` 사용하여 .NET SDK 스냅 패키지를 설치합니다.

   `--classic` 매개 변수는 필수입니다.

   - **.NET 9 이상**

     버전별 패키지를 설치합니다. 예를 들어, 다음 명령은 .NET SDK 10을 설치합니다.

     ```
     Bash
     sudo snap install dotnet-sdk-100 --classic
     ```

   - **.NET 8 및 이전 버전**

     패키지에서 `dotnet-sdk` 설치하고 채널을 지정합니다. 이 매개 변수를 생략하면 `latest/stable`이 사용됩니다. 예를 들어 다음 명령은 .NET SDK 8을 설치합니다.

Bash

```
sudo snap install dotnet-sdk --classic --channel 8.0/stable
```

`dotnet` 스냅 별칭이 자동으로 만들어지고 스냅 패키지의 `dotnet` 명령에 매핑됩니다.

다음 표에서는 설치할 수 있는 스냅 패키지 및 채널을 나열합니다.

테이블 확장

.NET 버전	Snap 패키지 또는 채널
10(LTS)	`dotnet-sdk-100`(미리 보기)
9(STS)	`dotnet-sdk-90`
8(LTS)	`dotnet-sdk --channel 8.0/stable`
7	`dotnet-sdk --channel 7.0/stable`(지원되지 않음)
6	`dotnet-sdk --channel 6.0/stable`(지원되지 않음)
5	`dotnet-sdk --channel 5.0/stable`(지원되지 않음)
3.1	`dotnet-sdk --channel 3.1/stable`(지원되지 않음)
2.1	`dotnet-sdk --channel 2.1/stable`(지원되지 않음)

2. 설치 위치 내보내기

`DOTNET_ROOT` 환경 변수는 도구에서 .NET 설치되는 위치를 결정하는 데 자주 사용됩니다. Snap을 통해 .NET 설치되면 이 환경 변수가 구성되지 않습니다. 프로필에서 *DOTNET_ROOT* 환경 변수를 구성해야 합니다. 스냅 경로는 `/snap/{package}/current` 형식을 사용합니다.

.NET 9 이상에서는 버전별 패키지 이름을 사용합니다.

Bash

```
export DOTNET_ROOT=/snap/dotnet-sdk-100/current
```

.NET 8 이하의 경우 공유 패키지 이름을 사용합니다.

Bash

```
export DOTNET_ROOT=/snap/dotnet-sdk/current
```

환경 변수를 영구적으로 내보내기

앞의 `export` 명령은 실행된 터미널 세션에 대한 환경 변수만 설정합니다.

셸 프로필을 편집하여 명령을 영구적으로 추가할 수 있습니다. Linux에서는 다양한 셸을 사용할 수 있으며 각 셸에는 서로 다른 프로필이 있습니다. 예시:

- **Bash 셸**: *~/.bash_profile*, *~/.bashrc*
- ko-KR: **Korn 셸**: *~/.kshrc* 또는 *.profile*
- **Z 셸**: _~/.zshrc* 또는 *.zprofile*

셸에 적합한 원본 파일을 편집하고 설치된 .NET 버전에 대한 내보내기 명령을 추가합니다. .NET 9 이상에서는 `export DOTNET_ROOT=/snap/dotnet-sdk-100/current`(필요에 따라 버전 번호 조정)를 사용합니다. .NET 8 이하의 경우 `export DOTNET_ROOT=/snap/dotnet-sdk/current` 사용합니다.

3. .NET CLI 사용

터미널을 열고 `dotnet`을 입력합니다.

.NET CLI

```
dotnet
```

다음 출력이 표시됩니다.

출력

```
Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help         Display help.
  --info            Display .NET information.
  --list-sdks       Display the installed SDKs.
  --list-runtimes   Display the installed runtimes.

path-to-application:
  The path to an application .dll file to execute.
```

.NET CLI를 사용하는 방법을 알아보려면 .NET CLI 개요 참조하세요.

문제 해결

- dotnet 터미널 명령이 작동하지 않음

- WSL2에서 스냅을 설치할 수 없음
- dotnet 명령 또는 SDK를 확인할 수 없음
- TLS/SSL 인증서 오류

dotnet 터미널 명령이 작동하지 않음

스냅 패키지는 패키지에서 제공하는 명령에 별칭을 매핑할 수 있습니다. 기본적으로 .NET SDK 스냅 패키지는 `dotnet` 명령에 대한 별칭을 만듭니다. 별칭이 만들어지지 않았거나 이전에 제거된 경우 다음 명령을 사용하여 별칭을 매핑합니다.

.NET 9 이상:

```bash
sudo snap alias dotnet-sdk-100.dotnet dotnet
```

.NET 8 이하의 경우:

```bash
sudo snap alias dotnet-sdk.dotnet dotnet
```

WSL2에서 Snap을 설치할 수 없음

스냅을 설치하려면 먼저 WSL2 인스턴스에서 `systemd`를 활성화해야 합니다.

1. 선택한 텍스트 편집기에서 `/etc/wsl.conf`를 엽니다.
2. 다음 구성을 붙여넣으세요.

   ```ini
   [boot]
   systemd=true
   ```

3. 파일을 저장하고 PowerShell을 통해 WSL2 인스턴스를 다시 시작합니다. `wsl.exe --shutdown` 명령 사용

dotnet 명령 또는 SDK를 해결할 수 없음

코드 IDE 또는 Visual Studio Code 확장과 같은 다른 앱에서 .NET SDK의 위치를 확인하는 것이 일반적입니다. 일반적으로 검색은 `DOTNET_ROOT` 환경 변수를 확인하거나 `dotnet` 실행 파일이 있

는 위치를 파악하는 방식으로 수행됩니다. 스냅 설치 .NET SDK는 이러한 앱을 혼동할 수 있습니다. 이러한 앱이 .NET SDK를 해결할 수 없는 경우 다음 메시지 중 하나와 유사한 오류가 표시됩니다.

- 지정한 SDK 'Microsoft.NET.Sdk'을(를) 찾을 수 없습니다.
- SDK 'Microsoft.NET.Sdk.Web'을 찾을 수 없습니다.
- 지정된 SDK 'Microsoft.NET.Sdk.Razor'을(를) 찾을 수 없습니다.

문제를 해결하려면 다음 단계를 완료합니다.

1. DOTNET_ROOT 환경 변수를 영구적으로 내보내는지 확인합니다.
2. 스냅 `dotnet` 실행 파일을 프로그램이 찾고 있는 위치에 기호 링크로 연결해 봅니다.

   `dotnet` 명령에서 찾고 있는 두 가지 일반적인 경로는 다음과 같습니다.

   - `/usr/local/bin/dotnet`
   - `/usr/share/dotnet`

   다음 명령을 사용하여 스냅 패키지에 대한 기호 링크를 만듭니다. .NET 9 이상에서는 버전별 패키지 이름을 사용합니다.

   Bash

   ```
   ln -s /snap/dotnet-sdk-100/current/dotnet /usr/local/bin/dotnet
   ```

   .NET 8 이하의 경우:

   Bash

   ```
   ln -s /snap/dotnet-sdk/current/dotnet /usr/local/bin/dotnet
   ```

TLS/SSL 인증서 오류

Snap을 통해 .NET 설치되는 경우 일부 배포에서 .NET TLS/SSL 인증서를 찾을 수 없으며 `restore` 중에 오류가 발생할 수 있습니다.

Bash

```
Processing post-creation actions...
Running 'dotnet restore' on /home/myhome/test/test.csproj...
  Restoring packages for /home/myhome/test/test.csproj...
/snap/dotnet-sdk/27/sdk/2.2.103/NuGet.targets(114,5): error : Unable to load the
service index for source https://api.nuget.org/v3/index.json.
[/home/myhome/test/test.csproj]
```

```
/snap/dotnet-sdk/27/sdk/2.2.103/NuGet.targets(114,5): error :   The SSL connection
could not be established, see inner exception. [/home/myhome/test/test.csproj]
/snap/dotnet-sdk/27/sdk/2.2.103/NuGet.targets(114,5): error :   The remote
certificate is invalid according to the validation procedure.
[/home/myhome/test/test.csproj]
```

이 문제를 해결하려면 다음과 같은 몇 가지 환경 변수를 설정합니다.

Bash

```
export SSL_CERT_FILE=[path-to-certificate-file]
export SSL_CERT_DIR=/dev/null
```

인증서 위치는 배포판에 따라 다릅니다. 문제가 관찰된 배포판의 위치는 다음과 같습니다.

테이블 확장

배포	위치
페도라	`/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem`
OpenSUSE	`/etc/ssl/ca-bundle.pem`
Solus	`/etc/ssl/certs/ca-certificates.crt`

관련 콘텐츠

- .NET CLI 개요
- TAB 완성을 .NET CLI에서 사용할 수 있도록 설정하는 방법
- Tutorial: .NET

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

새 .NET 버전으로 업그레이드

새 .NET 버전은 매년 릴리스됩니다. 많은 개발자가 새 버전을 사용할 수 있도록 즉시 업그레이드 프로세스를 시작하지만 다른 개발자는 사용 중인 버전이 더 이상 지원되지 않을 때까지 기다립니다. 업그레이드 프로세스에는 고려해야 할 여러 측면이 있습니다.

새 .NET 버전으로 업그레이드하는 일반적인 이유:

- 현재 사용되는 .NET 버전은 더 이상 지원되지 않습니다.
- 새 버전은 새 운영 체제를 지원합니다.
- 새 버전에는 중요한 API, 성능 또는 보안 기능이 있습니다.

개발 환경 업그레이드

새 .NET 버전으로 업그레이드하기 위해 .NET SDK는 설치할 기본 구성 요소입니다. 업데이트된 .NET CLI, 빌드 시스템 및 런타임 버전이 포함됩니다.

.NET 웹 사이트는 지원되는 운영 체제 및 아키텍처에서 다운로드하고 사용할 수 있는 설치 관리자 및 보관 파일을 제공합니다.

일부 운영 체제에는 새 .NET 버전을 설치하는 데 사용할 수 있는 패키지 관리자가 있습니다.

- 맥OS
- 리눅스
- Windows

Visual Studio는 새 .NET SDK 버전을 자동으로 설치합니다. Visual Studio 사용자의 경우 최신 Visual Studio 버전으로 업그레이드하는 것으로 충분합니다.

소스 코드 업그레이드

앱을 업그레이드하는 데 필요한 유일한 변경은 프로젝트 파일의 `TargetFramework` 속성을 최신 .NET 버전으로 업데이트하는 것입니다.

그 방법은 다음과 같습니다.

- 프로젝트 파일(`*.csproj`, `*.vbproj` 또는 `*.fsproj` 파일)을 엽니다.
- 예를 들어 `<TargetFramework>` 속성 값을 `net6.0`에서 `net8.0`(을)로 변경합니다.
- 사용 중인 경우 `<TargetFrameworks>` 속성에 동일한 패턴이 적용됩니다.

> **팁**

GitHub Copilot 앱 현대화 - 업그레이드 기능은 이러한 변경을 자동으로 수행할 수 있습니다.

다음 단계는 새 SDK를 사용하여 프로젝트(또는 솔루션)를 빌드하는 것입니다. 추가 변경이 필요한 경우 SDK는 사용자를 안내하는 경고 및 오류를 제공합니다.

새 SDK 버전으로 워크로드를 복원하려면 `dotnet workload restore`(을)를 실행해야 할 수 있습니다.

추가 리소스:

- .NET 9의 호환성이 손상되는 변경
- ASP.NET Core 앱 마이그레이션
- .NET MAUI를 .NET 7에서 .NET 8로 업그레이드

버전 고정 설정

.NET SDK, Visual Studio 또는 기타 구성 요소와 같은 개발 도구를 업그레이드할 때 새 동작, 분석기 경고 또는 빌드 프로세스에 영향을 주는 호환성이 손상되는 변경이 발생할 수 있습니다. 버전에 고정하면 프로젝트에서 특정 구성 요소가 업데이트되는 시기를 제어하면서 개발 환경을 업그레이드할 수 있습니다.

버전 고정은 다음과 같은 몇 가지 이점을 제공합니다.

- **예측 가능한 빌드**: 여러 컴퓨터 및 CI/CD 환경에서 일관된 빌드 결과를 보장합니다.
- **점진적 채택**: 새로운 기능을 한 번에 채택하는 대신 증분 방식으로 채택할 수 있습니다.
- **예기치 않은 변경 방지**: 새 분석기 규칙, SDK 동작 또는 패키지 버전에서 빌드 오류가 발생하지 않도록 방지합니다.
- **팀 조정**: 도구를 업데이트할 때 팀이 개별적으로 업그레이드하지 않고 계획된 시간에 함께 업그레이드할 수 있습니다.
- **디버깅 및 문제 해결**: 변경된 버전을 제어할 때 문제를 보다 쉽게 격리할 수 있습니다.

다음 섹션에서는 .NET 프로젝트에서 다양한 구성 요소의 버전을 제어하기 위한 다양한 메커니즘을 설명합니다.

- global.json사용하여 SDK 버전 제어
- 제어 분석기 동작
- NuGet 패키지 버전 제어
- MSBuild 버전 제어

global.json 사용하여 SDK 버전 제어

global.json 파일을 사용하여 프로젝트 또는 솔루션에 대한 .NET SDK 버전을 고정할 수 있습니다. 이 파일은 .NET CLI 명령을 실행할 때 사용할 SDK 버전을 지정하며 프로젝트가 대상으로 하는 런타임 버전과 독립적입니다.

솔루션 루트 디렉터리에 *global.json* 파일을 만듭니다.

.NET CLI

```
dotnet new globaljson --sdk-version 9.0.100 --roll-forward latestFeature
```

이 명령은 9.0 주 버전 내에서 SDK를 버전 9.0.100 이상 패치 또는 기능 밴드에 고정하는 다음 *global.json* 파일을 만듭니다.

JSON

```
{
  "sdk": {
    "version": "9.0.100",
    "rollForward": "latestFeature"
  }
}
```

정책은 `rollForward` 정확한 버전을 사용할 수 없을 때 SDK 버전을 선택하는 방법을 제어합니다. 이 구성을 사용하면 Visual Studio를 업그레이드하거나 새 SDK를 설치할 때 *global.json파일을 명시적으로* 업데이트할 때까지 프로젝트에서 SDK 9.0.x를 계속 사용할 수 있습니다.

자세한 내용은 global.json 개요를 참조하세요.

제어 분석기 동작

코드 분석기는 새 경고를 도입하거나 버전 간에 동작을 변경할 수 있습니다. 속성을 사용하여 일관된 빌드를 유지하도록 분석기 버전을 제어할 AnalysisLevel수 있습니다. 이 속성을 사용하면 특정 버전의 분석기 규칙을 잠글 수 있으므로 SDK를 업그레이드할 때 새 규칙이 도입되지 않습니다.

XML

```
<PropertyGroup>
  <AnalysisLevel>9.0</AnalysisLevel>
</PropertyGroup>
```

로 `9.0`설정하면 .NET 10 SDK를 사용하는 경우에도 .NET 9와 함께 제공되는 분석기 규칙만 사용하도록 설정됩니다. 이렇게 하면 문제를 해결할 준비가 될 때까지 새 .NET 10 분석기 규칙이 빌드에 영향을 주지 않습니다.

자세한 내용은 AnalysisLevel을 참조하세요.

NuGet 패키지 버전 제어

프로젝트 간에 패키지 버전을 일관되게 관리하면 예기치 않은 업데이트를 방지하고 신뢰할 수 있는 빌드를 유지할 수 있습니다.

- 패키지 잠금 파일
- 중앙 패키지 관리
- 패키지 원본 매핑

패키지 잠금 파일

패키지 잠금 파일은 패키지 복원 작업에서 서로 다른 환경에서 정확히 동일한 패키지 버전을 사용하도록 합니다. 잠금 파일(`packages.lock.json`)은 모든 패키지의 정확한 버전과 해당 종속성을 기록합니다.

프로젝트 파일에서 잠금 파일을 사용하도록 설정합니다.

XML

```xml
<PropertyGroup>
  <RestorePackagesWithLockFile>true</RestorePackagesWithLockFile>
</PropertyGroup>
```

잠금 파일이 만료된 경우 빌드가 실패하도록 하려면 다음을 수행합니다.

XML

```xml
<PropertyGroup>
  <RestorePackagesWithLockFile>true</RestorePackagesWithLockFile>
  <RestoreLockedMode>true</RestoreLockedMode>
</PropertyGroup>
```

잠금 파일을 사용하도록 설정한 후 실행 `dotnet restore` 하여 *packages.lock.json* 파일을 생성합니다. 이 파일을 소스 제어에 커밋합니다.

중앙 패키지 관리

CPM(중앙 패키지 관리)을 사용하면 솔루션의 모든 프로젝트에 대해 단일 위치에서 패키지 버전을 관리할 수 있습니다. 이 방법은 버전 관리를 간소화하고 프로젝트 간에 일관성을 보장합니다.

솔루션 루트에 *Directory.Packages.props* 파일을 만듭니다.

XML

```xml
<Project>
  <PropertyGroup>
    <ManagePackageVersionsCentrally>true</ManagePackageVersionsCentrally>
  </PropertyGroup>

  <ItemGroup>
    <PackageVersion Include="Azure.Identity" Version="1.17.0" />
    <PackageVersion Include="Microsoft.Extensions.AI" Version="9.10.1" />
  </ItemGroup>
</Project>
```

프로젝트 파일에서 버전을 지정하지 않고 패키지를 참조합니다.

XML

```xml
<ItemGroup>
  <PackageReference Include="Azure.Identity" />
  <PackageReference Include="Microsoft.Extensions.AI" />
</ItemGroup>
```

패키지 소스 매핑

패키지 원본 매핑을 사용하면 특정 패키지에 사용되는 NuGet 피드를 제어하여 보안 및 안정성을 향상시킬 수 있습니다.

nuget.config 파일에서 원본 매핑을 구성합니다.

XML

```xml
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json" />
    <add key="contoso" value="https://contoso.com/packages/" />
  </packageSources>

  <packageSourceMapping>
    <packageSource key="nuget.org">
      <package pattern="*" />
    </packageSource>
    <packageSource key="contoso">
      <package pattern="Contoso.*" />
    </packageSource>
  </packageSourceMapping>
</configuration>
```

이 구성을 사용하면 `Contoso.`로 시작하는 모든 패키지는 `contoso` 피드에서만 복원되고, 다른 패키지는 `nuget.org`에서 복원됩니다.

자세한 내용은 NuGet 패키지 복원을 참조하세요.

MSBuild 버전 제어

Visual Studio는 여러 버전의 병렬 설치를 지원합니다. 예를 들어 동일한 컴퓨터에 Visual Studio 2026 및 Visual Studio 2022를 설치할 수 있습니다. 각 Visual Studio 버전에는 해당 .NET SDK가 포함됩니다. Visual Studio를 업데이트하면 포함된 SDK 버전도 업데이트됩니다. 그러나 .NET 다운로드 페이지에서 별도로 설치하여 이전 SDK 버전을 계속 사용할 수 있습니다.

MSBuild 버전은 Visual Studio 버전에 해당합니다. 예를 들어 Visual Studio 2022 버전 17.8에는 MSBuild 17.8이 포함됩니다. .NET SDK에는 MSBuild도 포함됩니다. `dotnet build`를 사용할 때, *global.json*에 지정된 SDK 또는 최신으로 설치된 SDK에 포함된 MSBuild 버전을 사용하게 됩니다.

특정 MSBuild 버전을 사용하려면 다음을 수행합니다.

- `dotnet build`고정된 SDK 버전과 함께 사용합니다.
- 해당 Visual Studio 버전의 MSBuild에 대한 환경을 설정하는 적절한 Visual Studio 개발자 명령 프롬프트를 시작합니다.
- 특정 Visual Studio 설치(예 `"C:\Program Files\Microsoft Visual Studio\2022\Enterprise\MSBuild\Current\Bin\MSBuild.exe"`:)에서 MSBuild를 직접 호출합니다.

자세한 내용은 .NET SDK, MSBuild 및 Visual Studio 버전 관리를 참조하세요.

CI(지속적인 통합) 업데이트

CI 파이프라인은 프로젝트 파일 및 Dockerfiles와 유사한 업데이트 프로세스를 따릅니다. 일반적으로 버전 값만 변경하여 CI 파이프라인을 업데이트할 수 있습니다.

호스팅 환경 업데이트

애플리케이션 호스팅에 사용되는 많은 패턴이 있습니다. 호스팅 환경에 .NET 런타임이 포함된 경우 새 버전의 .NET 런타임을 설치해야 합니다. Linux에서는 종속성을 설치해야 하지만 일반적으로 .NET 버전 간에는 변경되지 않습니다.

컨테이너의 경우 새 버전 번호를 포함하도록 FROM문을 변경해야 합니다.

다음 Dockerfile 예제에서는 ASP.NET Core 9.0 이미지를 당기는 방법을 보여 줍니다.

Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:9.0
```

Azure App Service와 같은 클라우드 서비스에서는 구성 변경이 필요합니다.

참고하십시오

- .NET 10의 주요 변경 내용
- ASP.NET Core 앱 마이그레이션

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 11. 07.

.NET 런타임 및 SDK를 제거하는 방법

시간이 지남에 따라 업데이트된 버전의 .NET 런타임 및 SDK를 설치할 때 머신에서 오래된 버전의 .NET을 제거하는 것이 좋습니다. 이전 버전의 런타임을 제거하면 .NET 버전 선택문서에 설명된 대로 공유 프레임워크 애플리케이션을 실행하도록 선택한 런타임이 변경될 수 있습니다.

버전을 제거해야 하나요?

.NET 버전 선택 동작 및 .NET의 런타임 호환성을 업데이트하여 이전 버전을 안전하게 제거할 수 있습니다. .NET 런타임 업데이트는 8.x 및 7.x와 같은 주 버전 **대역** 내에서 호환됩니다. 또한 .NET SDK의 최신 릴리스는 호환 가능한 방식으로 이전 버전의 런타임을 대상으로 하는 애플리케이션을 빌드하는 기능을 일반적으로 유지합니다.

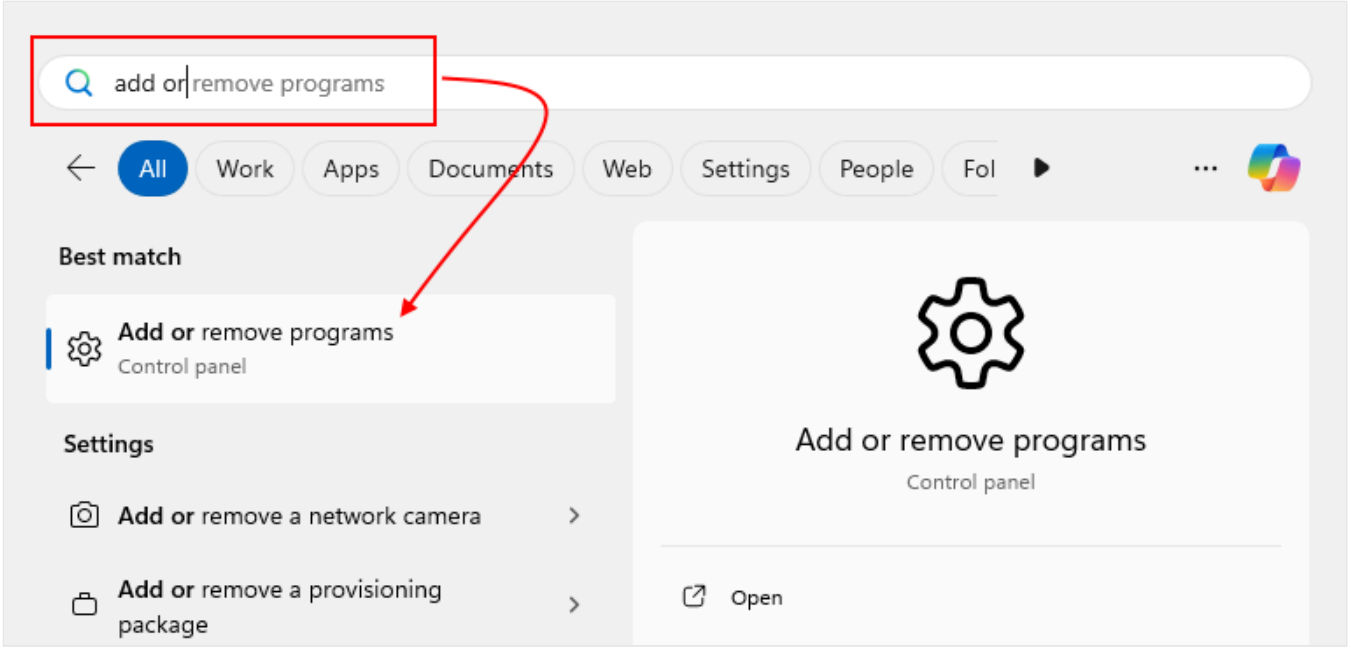
일반적으로 애플리케이션에 필요한 최신 SDK 및 런타임의 최신 패치 버전만 있으면 됩니다. 이전 SDK 또는 런타임 버전을 유지하려는 인스턴스에는 *project.json* 기반 애플리케이션 유지 관리가 포함됩니다. 애플리케이션에 이전 SDK 또는 런타임에 대한 특정 이유가 없는 경우 이전 버전을 안전하게 제거할 수 있습니다.

설치된 버전 확인

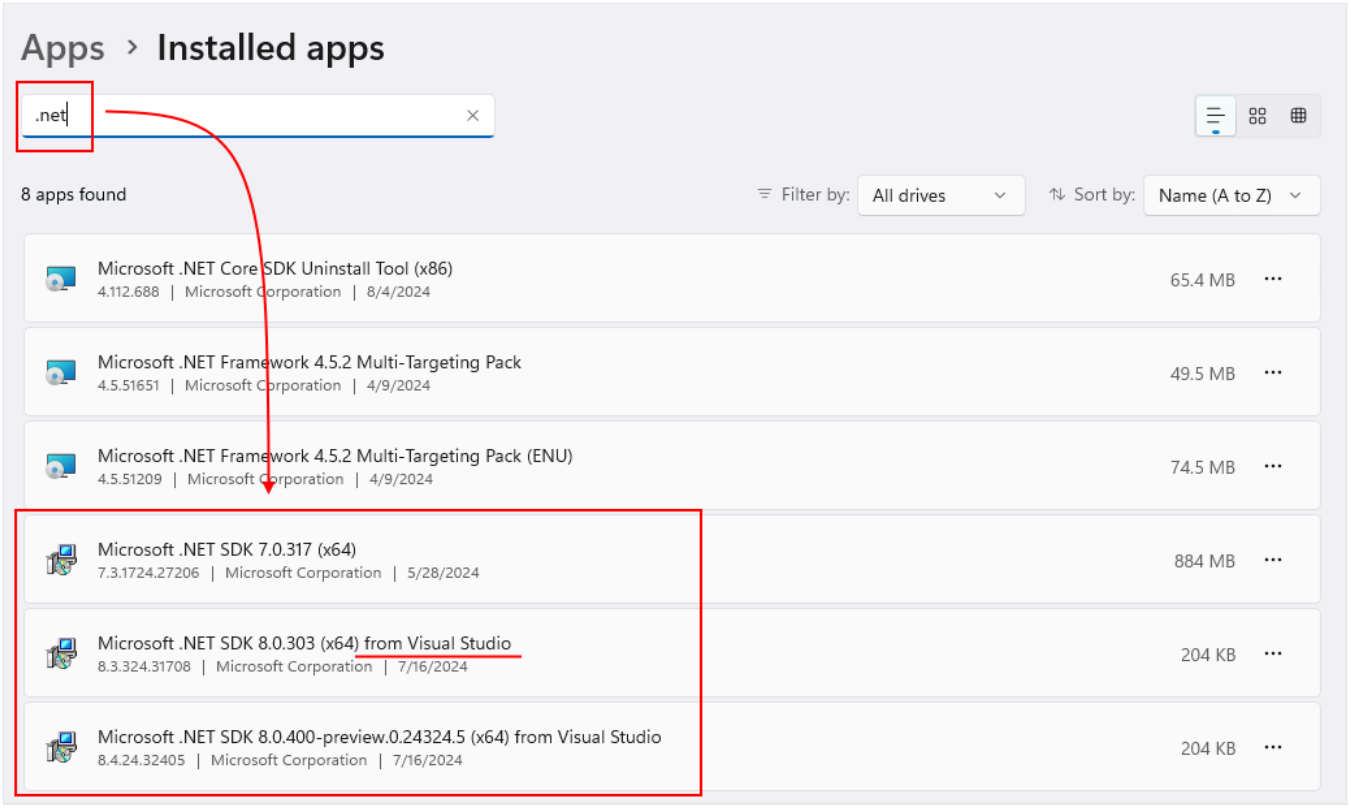
.NET CLI에는 컴퓨터에 설치된 SDK 및 런타임 버전을 나열하는 데 사용할 수 있는 옵션이 있습니다. dotnet --list-sdks를 사용하여 설치된 SDK의 목록을 확인하고 dotnet --list-runtimes를 사용하여 런타임의 목록을 확인합니다. 자세한 내용은 .NET이 이미 설치되어 있는지 확인하는 방법을 참조하세요.

.NET 제거

NET은 .NET 런타임 및 SDK 버전을 제거하기 위해 Windows **앱 & 기능** 또는 **앱** > **설치된 앱** 설정 페이지를 사용합니다. 설정 페이지를 열기 위해 다음 이미지와 같이 시작 메뉴를 사용하여 **프로그램 추가 또는 제거**를 검색합니다. **앱 & 기능** 대화 상자가 그림에 나와 있습니다. **core** 또는 **.net**을 검색하여 설치된 .NET 버전을 필터링하고 표시할 수 있습니다.



컴퓨터에 설치된 버전을 찾기 위해 설정 페이지에서 **.net**을 검색합니다. 항목을 제거하기 위해 ...>**제거**를 선택합니다. Windows 10 사용 시, 제거하려는 항목에 관한 **제거** 버튼을 선택합니다. Windows 11의 **설치된 앱** 설정 페이지를 다음 이미지에서 보여 줍니다.



> **ⓘ Important**
>
> 제거하고자 하는 항목의 출처가 Visual Studio임이 나타나는 경우, 해당 버전의 .NET을 제거하기 위해 Visual Studio 설치 관리자를 사용합니다.

.NET 제거 도구

.NET 제거 도구를 사용하면 시스템에서 .NET SDK 및 런타임을 제거할 수 있습니다. 제거해야 하는 버전을 지정할 수 있는 옵션 컬렉션이 제공됩니다. 자세한 내용은 .NET 제거 도구 개요를 참조하세요.

NuGet 대체 디렉터리 제거

.NET Core 3.0 SDK 이전에는 .NET Core SDK 설치 관리자가 *NuGetFallbackFolder*로 명명된 디렉터리를 사용하여 NuGet 패키지의 캐시를 저장했습니다. 이 캐시는 `dotnet restore`나 `dotnet build /t:Restore` 같은 작업 중에 사용되었습니다. *NuGetFallbackFolder*(은)는 .NET이 설치된 *sdk* 폴더 아래에 있습니다. 예를 들어 Windows에서 *C:\Program Files\dotnet\sdk\NuGetFallbackFolder*와 macOS의 */usr/local/share/dotnet/sdk/NuGetFallbackFolder*에 있을 수 있습니다.

다음 경우에는 이 디렉터리를 제거하고자 할 수 있습니다.

- .NET Core 3.0 SDK 또는 .NET 5 이상 버전을 사용하여 개발 중입니다.
- 3.0 이전의 .NET Core SDK 버전을 사용하여 개발하고 있지만 온라인으로 작업할 수 있습니다.

NuGet 대체 디렉터리를 제거하려면 삭제할 수 있지만 이렇게 하려면 관리자 권한이 필요합니다.

dotnet 디렉터리를 삭제하지 않는 것이 좋습니다. 이렇게 하면 이전에 설치한 모든 전역 도구가 제거됩니다. 또한 Windows에서

- Visual Studio 2019 버전 16.3 이상 버전의 호환성이 손상됩니다. **복구**를 실행하여 복구할 수 있습니다.
- **앱 및 기능** 대화 상자에 있는 .NET Core SDK 항목은 분리됩니다.

Last updated on 2025. 11. 08.

.NET 프로젝트 및 항목 템플릿 관리

.NET는 사용자가 NuGet, NuGet 패키지 파일 또는 파일 시스템 디렉터리에서 템플릿이 포함된 패키지를 설치하거나 제거할 수 있도록 하는 템플릿 시스템을 제공합니다. 이 문서에서는 .NET SDK CLI를 통해 .NET 템플릿을 관리하는 방법을 설명합니다.

템플릿 만들기에 대한 자세한 내용은 자습서: 템플릿 만들기를 참조하세요.

템플릿 설치

템플릿 패키지는 dotnet new install SDK 명령을 통해 설치됩니다. 템플릿 패키지의 NuGet 패키지 식별자를 제공하거나 템플릿 파일이 포함된 폴더를 제공할 수 있습니다.

NuGet 호스티드 패키지

.NET CLI 템플릿 패키지는 광범위한 배포를 위해 NuGet에 업로드됩니다. 템플릿 패키지는 프라이빗 피드에서도 설치할 수 있습니다. NuGet 피드에 템플릿 패키지를 업로드하는 대신 *로컬 NuGet 패키지* 섹션에 설명된 대로 nupkg 템플릿 파일을 배포하고 수동으로 설치할 수 있습니다.

NuGet 피드를 구성하는 방법에 대한 자세한 내용은 dotnet nuget add source를 참조하세요.

기본 NuGet 피드에서 템플릿 패키지를 설치하려면 `dotnet new install {package-id}` 명령을 사용합니다.

.NET CLI

```
dotnet new install Microsoft.DotNet.Web.Spa.ProjectTemplates
```

특정 버전으로 기본 NuGet 피드에서 템플릿 패키지를 설치하려면 `dotnet new install {package-id}::{version}` 명령을 사용합니다.

.NET CLI

```
dotnet new install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.2.6
```

로컬 NuGet 패키지

템플릿 패키지가 만들어지면 *nupkg* 파일이 만들어집니다. 템플릿을 포함하는 *nupkg* 파일이 있는 경우 `dotnet new install {path-to-package}` 명령을 사용하여 설치할 수 있습니다.

.NET CLI

```
dotnet new install c:\code\nuget-packages\Some.Templates.1.0.0.nupkg
```

폴더

nupkg 파일에서 템플릿을 설치하는 대신 `dotnet new install {folder-path}` 명령을 사용하여 폴더에서 직접 템플릿을 설치할 수도 있습니다. 지정된 폴더는 발견된 템플릿에 대한 템플릿 패키지 식별자로 처리됩니다. 지정된 폴더의 계층 구조에 있는 모든 템플릿이 설치됩니다.

.NET CLI

```
dotnet new install c:\code\nuget-packages\some-folder\
```

명령에 지정된 `{folder-path}`는 찾은 모든 템플릿의 템플릿 패키지 식별자가 됩니다. 템플릿 패키지 나열 섹션에 지정된 대로 `dotnet new uninstall` 명령을 사용하여 설치된 템플릿 패키지 목록을 가져올 수 있습니다. 이 예에서는 템플릿 패키지 식별자가 설치에 사용된 폴더로 표시됩니다.

콘솔

```
dotnet new uninstall
Currently installed items:

... cut to save space ...

  c:\code\nuget-packages\some-folder
    Templates:
      A Template Console Class (templateconsole) C#
      Project for some technology (contosoproject) C#
    Uninstall Command:
      dotnet new uninstall c:\code\nuget-packages\some-folder
```

템플릿 패키지 제거

템플릿 패키지는 dotnet new uninstall SDK 명령을 통해 제거됩니다. 템플릿 패키지의 NuGet 패키지 식별자를 제공하거나 템플릿 파일이 포함된 폴더를 제공할 수 있습니다.

NuGet 패키지

NuGet 템플릿 패키지를 설치한 후에는 NuGet 피드 또는 *nupkg* 파일에서 NuGet 패키지 식별자를 참조하여 해당 템플릿 팩을 제거할 수 있습니다.

템플릿 패키지를 제거하려면 `dotnet new uninstall {package-id}` 명령을 사용합니다.

.NET CLI

```
dotnet new uninstall Microsoft.DotNet.Web.Spa.ProjectTemplates
```

폴더

템플릿이 폴더 경로를 통해 설치되면 폴더 경로는 템플릿 패키지 식별자가 됩니다.

템플릿 패키지를 제거하려면 `dotnet new uninstall {package-folder-path}` 명령을 사용합니다.

.NET CLI

```
dotnet new uninstall c:\code\nuget-packages\some-folder
```

템플릿 패키지 나열

패키지 식별자 없이 표준 제거 명령을 사용하면 각 템플릿 패키지를 제거하는 명령과 함께 설치된 템플릿 패키지 목록을 볼 수 있습니다.

콘솔

```
dotnet new uninstall
Currently installed items:

... cut to save space ...

  c:\code\nuget-packages\some-folder
    Templates:
      A Template Console Class (templateconsole) C#
      Project for some technology (contosoproject) C#
    Uninstall Command:
      dotnet new uninstall c:\code\nuget-packages\some-folder
```

다른 SDK에서 템플릿 패키지 설치

SDK 6.0, SDK 7.0 등을 차례로 설치하는 것과 같이 각 버전의 SDK를 순차적으로 설치한 경우 SDK의 모든 템플릿이 설치됩니다. 그러나 7.0과 같은 최신 SDK 버전으로 시작하는 경우 이 버전의 템플릿만 포함됩니다. 다른 릴리스용 템플릿은 포함되지 않습니다.

.NET 템플릿은 NuGet에서 사용할 수 있으며 다른 템플릿처럼 설치할 수 있습니다. 자세한 내용은 NuGet 호스티드 패키지 설치를 참조하세요.

테이블 확장

SDK (소프트웨어 개발 키트)	NuGet 패키지 식별자
.NET Core 2.1	Microsoft.DotNet.Common.ProjectTemplates.2.1
.NET Core 2.2	Microsoft.DotNet.Common.ProjectTemplates.2.2
.NET Core 3.0	Microsoft.DotNet.Common.ProjectTemplates.3.0
.NET Core 3.1	Microsoft.DotNet.Common.ProjectTemplates.3.1
.NET 5.0	Microsoft.DotNet.Common.ProjectTemplates.5.0
.NET 6.0	Microsoft.DotNet.Common.ProjectTemplates.6.0
.NET 7.0	Microsoft.DotNet.Common.ProjectTemplates.7.0
.NET 8.0	Microsoft.DotNet.Common.ProjectTemplates.8.0
.NET 9.0	Microsoft.DotNet.Common.ProjectTemplates.9.0
.NET 10.0	Microsoft.DotNet.Common.ProjectTemplates.10.0
ASP.NET Core 2.1	Microsoft.DotNet.Web.ProjectTemplates.2.1
ASP.NET Core 2.2	Microsoft.DotNet.Web.ProjectTemplates.2.2
ASP.NET Core 3.0	Microsoft.DotNet.Web.ProjectTemplates.3.0
ASP.NET Core 3.1	Microsoft.DotNet.Web.ProjectTemplates.3.1
ASP.NET Core 5.0	Microsoft.DotNet.Web.ProjectTemplates.5.0
ASP.NET Core 6.0	Microsoft.DotNet.Web.ProjectTemplates.6.0
ASP.NET Core 7.0	Microsoft.DotNet.Web.ProjectTemplates.7.0
ASP.NET Core 8.0	Microsoft.DotNet.Web.ProjectTemplates.8.0
ASP.NET Core 9.0	Microsoft.DotNet.Web.ProjectTemplates.8.0
ASP.NET Core 10.0	Microsoft.DotNet.Web.ProjectTemplates.10.0

예를 들어 .NET 9 SDK에는 .NET 9를 대상으로 하는 콘솔 앱에 대한 템플릿이 포함되어 있습니다. .NET Core 3.1을 대상으로 하려면 3.1 템플릿 패키지를 설치해야 합니다.

1. .NET Core 3.1을 대상으로 하는 앱을 만들어 보세요.

   .NET CLI

   ```
   dotnet new console --framework netcoreapp3.1
   ```

   오류 메시지가 표시되면 다음 단계에서 템플릿을 설치합니다.

2. .NET Core 3.1 프로젝트 템플릿을 설치합니다.

.NET CLI

```
dotnet new install Microsoft.DotNet.Common.ProjectTemplates.3.1
```

3. 앱을 다시 만들어 보세요.

.NET CLI

```
dotnet new console --framework netcoreapp3.1
```

프로젝트가 생성되었음을 나타내는 메시지가 표시됩니다.

출력

```
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on path-to-project-file.csproj...
  Determining projects to restore...
  Restore completed in 1.05 sec for path-to-project-file.csproj.

Restore succeeded.
```

참고 항목

- 자습서: 항목 템플릿 만들기
- dotnet new
- dotnet nuget add source

Last updated on 2025. 11. 08.

macOS Catalina 공증과 이것이 .NET 다운로드 및 프로젝트에 미치는 영향

macOS Catalina(버전10.15)부터, 2019년 6월 1일 이후에 빌드되어 개발자 ID로 배포되는 모든 소프트웨어는 공증을 받아야 합니다. 이 요구 사항은 .NET 런타임, .NET SDK, .NET으로 만든 소프트웨어에 적용됩니다. 이 문서에서는 .NET 및 macOS 공증과 관련하여 자주 발생하는 시나리오에 대해 설명합니다.

.NET 설치

.NET(런타임 및 SDK 둘 다)용 설치 프로그램은 2020년 2월 18일부터 공증되었습니다. 그 전에 릴리스된 버전은 공증되지 않았습니다. 공증되지 않은 버전의 .NET은 먼저 설치 프로그램을 다운로드한 다음, 다운로드한 설치 프로그램을 통해 `sudo installer` 명령을 사용하여 수동으로 설치할 수 있습니다.

네이티브 appHost

앱이 컴파일되면 **네이티브 Mach-O 실행 파일인 appHost**가 생성됩니다. 이 실행 파일은 일반적으로 프로젝트가 `dotnet run` 명령을 통해 컴파일, 게시 또는 실행될 때 .NET에 의해 호출됩니다. 앱의 **appHost**가 아닌 버전은 명령으로 호출할 수 있는 `dotnet <app.dll>` 파일입니다.

로컬로 실행하면 SDK는 임시 서명을 사용하여 apphost에 서명합니다. 그러면 앱이 로컬로 실행될 수 있습니다. 앱을 배포할 때 Apple 지침에 따라 앱에 올바르게 서명해야 합니다.

apphost 없이 앱을 배포하고 사용자가 `dotnet`을 사용하여 앱을 실행할 수도 있습니다. **appHost** 생성을 끄려면 프로젝트 파일에 UseAppHost 부울 설정을 추가하고 이를 `false`로 설정합니다. 명령줄에서 `-p:UseAppHost` 매개 변수를 사용하여 실행하는 특정 `dotnet` 명령에 대해 appHost를 켜거나 끌 수도 있습니다.

- 프로젝트 파일

  XML

  ```xml
  <PropertyGroup>
    <UseAppHost>false</UseAppHost>
  </PropertyGroup>
  ```

- 명령줄 매개 변수

  .NET CLI

```
dotnet run -p:UseAppHost=false
```

앱 **자체 포함**을 게시할 때 appHost가 필요하며 사용하지 않도록 설정할 수 없습니다.

`UseAppHost` 설정에 대한 자세한 내용은 MSBuild properties for Microsoft.NET.Sdk(Microsoft.NET.Sdk의 MSBuild 속성)를 참조하세요.

appHost의 컨텍스트

프로젝트에서 appHost가 사용하도록 설정된 상태에서 `dotnet run` 명령을 사용하여 앱을 실행하면 앱이 기본 호스트(기본 호스트는 `dotnet` 명령임)가 아닌 appHost의 컨텍스트에서 호출됩니다. 프로젝트에서 appHost를 사용하지 않도록 설정된 경우 `dotnet run` 명령이 기본 호스트의 컨텍스트에서 앱을 실행합니다. appHost를 사용하지 않도록 설정된 경우에도 앱을 자체 포함 방식으로 게시하면 appHost 실행 파일이 생성되고, 사용자가 해당 실행 파일을 사용하여 앱을 실행할 수 있습니다. `dotnet <filename.dll>`을 사용하여 앱을 실행하면 기본 호스트인 공유 런타임으로 앱이 호출됩니다.

appHost를 사용하는 앱이 호출될 경우, 앱에서 액세스하는 인증서 파티션은 공증된 기본 호스트와 다릅니다. 앱에서 기본 호스트를 통해 설치된 인증서에 액세스해야 하는 경우, `dotnet run` 명령을 사용하여 앱을 프로젝트 파일에서 실행하거나 `dotnet <filename.dll>` 명령을 사용하여 앱을 직접 시작하세요.

ASP.NET Core 및 macOS와 인증서 섹션에서 이 시나리오에 대한 자세한 내용을 확인할 수 있습니다.

ASP.NET Core, macOS 및 인증서

.NET은 macOS 키체인에서 System.Security.Cryptography.X509Certificates 클래스로 인증서를 관리하는 기능을 제공합니다. macOS 키체인에 대한 액세스에서는 어느 파티션을 고려해야 할지 결정할 때 애플리케이션 ID를 기본 키로 사용합니다. 예를 들어, 서명되지 않은 애플리케이션은 비밀을 서명되지 않은 파티션에 저장하지만, 서명된 애플리케이션은 비밀을 해당 애플리케이션에서만 액세스할 수 있는 파티션에 저장합니다. 앱을 호출하는 실행 파일의 소스가 어느 파티션을 사용할지를 결정합니다.

.NET은 appHost, 기본 호스트(`dotnet` 명령), 사용자 지정 호스트와 같은 3가지 소스의 실행을 제공합니다. 각 실행 모델에는 서명되었거나 서명되지 않은 서로 다른 ID를 가질 수 있으며, 키체인 내의 서로 다른 파티션에 대한 액세스를 갖습니다. 하나의 모드로 가져온 인증서는 다른 모드에서 액세스하지 못할 수 있습니다. 예를 들어, .NET의 공증된 버전에는 서명된 기본 호스트가 있습니다. 인증서는 그 ID에 따라 보안 파티션으로 가져와집니다. appHost가 임시로 서명되었으므로 생성된 appHost에서 이러한 인증서에 액세스할 수 없습니다.

또 다른 예로, 기본적으로 ASP.NET Core는 기본 호스트를 통해 기본 SSL 인증서를 가져옵니다. appHost를 사용하는 ASP.NET Core 애플리케이션은 이 인증서에 액세스할 수 없으며, .NET에서 이 인증서에 액세스할 수 없다는 사실을 감지하면 오류가 발생합니다. 오류 메시지는 이 문제를 해결하는 방법에 대한 지침을 제공합니다.

인증서 공유가 필요한 경우, macOS는 `security` 유틸리티를 통해 구성 옵션을 제공합니다.

ASP.NET Core 인증서 문제를 해결하는 방법에 대한 자세한 내용은 Enforce HTTPS in ASP.NET Core(ASP.NET Core에서 HTTPS 강제 적용)를 참조하세요.

기본 자격

.NET의 기본 호스트(`dotnet` 명령)에는 기본 자격 세트가 있습니다. 이러한 자격은 .NET이 올바르게 작동하는 데 필요합니다. 애플리케이션에 그 밖의 자격이 필요한 경우가 있는데, 이 경우 appHost를 생성하여 사용한 다음 로컬에서 필수 자격을 추가해야 합니다.

.NET의 기본 자격 세트:

- `com.apple.security.cs.allow-jit`
- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.allow-dyld-environment-variables`
- `com.apple.security.cs.disable-library-validation`

.NET 앱 공증

애플리케이션을 macOS Catalina(버전 10.15) 이상에서 실행하려면 앱을 공증해야 합니다. 공증을 위해 애플리케이션과 함께 제출하는 appHost에는 .NET Core의 기본 자격과 같거나 높은 기본 자격을 사용해야 합니다.

다음 단계

- macOS에 .NET을 설치합니다.

Last updated on 2026. 01. 16.

Linux에서 누락된 파일과 관련된 .NET 오류 문제 해결

Linux에서 .NET을 사용하려고 하면 `dotnet new`, `dotnet run` 등의 명령이 실패하고 *fxr*, *libhostfxr.so* 또는 *FrameworkList.xml* 등의 파일을 찾을 수 없다는 메시지가 표시될 수 있습니다. 오류 메시지 몇 가지는 다음 항목과 유사할 수 있습니다.

- **System.IO.FileNotFoundException**

  > System.IO.FileNotFoundException: '/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList.xml' 파일을 찾을 수 없습니다.

- **오류가 발생했습니다.**

  > 오류가 발생했습니다. 필수 라이브러리 *libhostfxr.so*를 찾을 수 없습니다.

  또는

  > 오류가 발생했습니다. [/usr/share/dotnet/host/fxr] 폴더가 없습니다.

  또는

  > 오류가 발생했습니다. [/usr/share/dotnet/host/fxr] 폴더에 버전 번호가 매겨진 자식 폴더가 없습니다.

- **dotnet을 찾을 수 없음에 관한 일반 메시지**

  SDK를 찾을 수 없거나 패키지가 이미 설치되어 있음을 나타내는 일반 메시지가 표시될 수 있습니다.

이러한 문제의 한 가지 증상은 `/usr/lib64/dotnet` 및 `/usr/share/dotnet` 폴더가 모두 시스템에 있다는 것입니다.

> **팁**
>
> 설치된 SDK 및 런타임을 나열하려면 `dotnet --info` 명령을 사용합니다. 자세한 내용은 **.NET이 이미 설치되어 있는지 확인하는 방법**을 참조하세요.

문제의 원인

> **Important**
>
> .NET 9부터 Microsoft는 자체 패키지를 게시하지 않는 지원되는 Linux 배포판용 패키지만 게시합니다. 자세한 내용은 **Linux에 .NET 설치**를 참조하세요.

이러한 오류는 일반적으로 두 개의 Linux 패키지 리포지토리가 .NET 패키지를 제공할 때 발생합니다. Microsoft는 .NET 패키지를 제공하기 위해 하나의 Linux 패키지 리포지토리를 제공하지만, 일부 Linux 배포판은 .NET 패키지도 제공합니다. 이러한 배포판에는 다음이 포함됩니다.

- Alpine Linux
- 아치
- CentOS 스트림
- 페도라
- 레드햇 엔터프라이즈 리눅스 (RHEL)
- Ubuntu 22.04 이상

서로 다른 두 원본의 .NET 패키지를 혼합하면 문제가 발생할 가능성이 높습니다. 패키지는 항목을 다른 경로에 배치할 수 있으며 다르게 컴파일될 수 있습니다.

해결 방법

이 문제의 해결 방법은 패키지 리포지토리 하나의 .NET을 사용하는 것입니다. 선택할 리포지토리와 선택 방법은 사용 사례 및 Linux 배포판에 따라 다릅니다.

- 내 Linux 배포판은 .NET 패키지를 제공하며 이를 사용하려고 합니다.
- 내 Linux 배포판에서 제공하지 않는 .NET 버전이 필요합니다.

내 Linux 배포판은 .NET 패키지를 제공하며 이를 사용하려고 합니다.

- **PowerShell 및 MSSQL과 같은 다른 패키지에 대해 Microsoft 리포지토리를 사용하나요?**
  - **예**

    Microsoft 리포지토리의 .NET 패키지를 무시하도록 패키지 관리자를 구성합니다. 두 리포지토리 모두에서 .NET을 설치했을 수 있으므로 둘 중 하나를 선택하려고 합니다.

    1. 배포에서 기존 .NET 패키지를 제거합니다. 다시 시작하고 잘못된 리포지토리에서 설치하지 않았는지 확인하려고 합니다.

```
Bash
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. .NET 패키지를 무시하도록 Microsoft 리포지토리를 구성합니다.

```
Bash
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a
/etc/yum.repos.d/microsoft-prod.repo
```

3. 배포 패키지 피드에서 .NET을 다시 설치합니다. 자세한 내용은 Linux에 .NET 설치를 참조하세요.

- **문제**

1. 배포에서 기존 .NET 패키지를 제거합니다. 다시 시작하고 잘못된 리포지토리에서 설치하지 않았는지 확인하려고 합니다.

```
Bash
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. 배포에서 Microsoft 리포지토리 피드를 삭제합니다.

```
Bash
sudo dnf remove packages-microsoft-prod
```

3. 배포 패키지 피드에서 .NET을 다시 설치합니다. 자세한 내용은 Linux에 .NET 설치를 참조하세요.

내 Linux 배포판에서 제공하지 않는 .NET 버전이 필요합니다.

배포 리포지토리에서 .NET 패키지를 무시하도록 패키지 관리자를 구성합니다. 두 리포지토리 모두에서 .NET을 설치했을 수 있으므로 둘 중 하나를 선택하려고 합니다.

1. 배포에서 기존 .NET 패키지를 제거합니다. 다시 시작하고 잘못된 리포지토리에서 설치하지 않았는지 확인하려고 합니다.

```
Bash
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. .NET 패키지를 무시하도록 Linux 리포지토리를 구성합니다.

```bash
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a
/etc/yum.repos.d/<your-package-source>.repo
```

`<your-package-source>`를 배포판의 패키지 원본으로 바꿉니다.

3. 배포 패키지 피드에서 .NET을 다시 설치합니다. 자세한 내용은 Linux에 .NET 설치를 참조하세요.

온라인 참조

다른 많은 사용자들이 이러한 문제를 보고했습니다. 다음은 이러한 문제의 목록입니다. 발생할 수 있는 일에 대한 인사이트를 얻으려면 다음을 참조하세요.

- System.IO.FileNotFoundException 및 '/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList.xml'
  - SDK #15785: 5.0.3으로 업그레이드한 후 새 프로젝트를 빌드할 수 없음
  - SDK #15863: 5.0.103으로 업데이트한 후 "MSB4018 ResolveTargetingPackAssets 작업이 예기치 않게 실패했습니다."가 발생함
  - SDK #17411: dotnet 빌드에서 항상 오류가 throw됨
  - SDK #12075: Fedora 32의 dotnet 3.1.301에서 없어서 FrameworkList.xml을 찾을 수 없음
- 오류: *libhostfxr.so*를 찾을 수 없음
  - SDK #17570: Fedora 33에서 34로, dotnet 5.0.5에서 5.0.6으로 업데이트한 후 libhostfxr.so 관련 오류가 발생함:
- 오류: */host/fxr* 폴더가 없음
  - Core #5746: packages.microsoft.com 리포지토리가 사용하도록 설정된 상태에서 CentOS 8에 3.1을 설치할 때 해당 폴더가 없음
  - SDK #15476: 치명적인 오류가 발생했습니다. '/usr/share/dotnet/host/fxr' 폴더가 없습니다.:
- 오류: */host/fxr* 폴더에 버전 번호가 매겨진 자식 폴더가 없음
  - 설치 관리자 #9254: CentOS 8에 dotnet/core/aspnet:3.1을 설치할 때 오류 발생 - 폴더에 버전 번호가 매겨진 자식 폴더가 없음
  - StackOverflow: CentOS 8에 dotnet/core/aspnet:3.1을 설치할 때 오류 발생 - 폴더에 버전 번호가 매겨진 자식 폴더가 없음
- 명확한 메시지가 없는 일반 오류
  - Core #4605: "dotnet new console"을 실행할 수 없음
  - Core #4644: Fedora 32에 .NET Core SDK 2.1을 설치할 수 없음

- 런타임 #49375: 패키지 관리자를 사용하여 5.0.200-1로 업데이트한 후 SDK가 설치된 것 같지 않음
- 설치 프로그램 #16438: 애플리케이션 '--version'이 존재하지 않습니다.

참고 항목

- .NET이 설치되어 있는지 확인하는 방법
- .NET 런타임 및 SDK를 제거하는 방법
- Linux에 .NET 설치

Last updated on 2025. 11. 08.

.NET이 이미 설치되어 있는지 확인하는 방법

이 문서에서는 컴퓨터에 설치된 .NET 런타임 및 SDK 버전을 확인하는 방법을 설명합니다. Visual Studio와 같은 통합 개발 환경이 있는 경우 .NET이 이미 설치되었을 수 있습니다.

SDK를 설치하면 해당 런타임이 설치됩니다.

이 문서의 명령이 실패하면 런타임 또는 SDK가 설치되어 있지 않습니다. 자세한 내용은 Windows, macOS 또는 Linux에 대한 설치 문서를 참조하세요.

SDK 버전 확인

터미널과 함께 현재 설치되어 있는 .NET SDK 버전을 확인할 수 있습니다. 터미널을 열고 다음 명령을 실행합니다.

.NET CLI

```
dotnet --list-sdks
```

다음과 유사한 출력이 표시됩니다.

콘솔

```
7.0.404 [C:\program files\dotnet\sdk]
8.0.100 [C:\program files\dotnet\sdk]
9.0.306 [C:\program files\dotnet\sdk]
10.0.100 [C:\program files\dotnet\sdk]
```

런타임 버전 확인

다음 명령을 사용하여 현재 설치된 .NET 런타임 버전을 확인할 수 있습니다.

.NET CLI

```
dotnet --list-runtimes
```

다음과 유사한 출력이 표시됩니다.

콘솔

```
Microsoft.AspNetCore.App 7.0.5 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 8.0.0 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 9.0.10 [C:\Program
```

```
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 10.0.0 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 7.0.5 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 8.0.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 9.0.10 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 10.0.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.WindowsDesktop.App 7.0.5 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 8.0.0 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 9.0.10 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 10.0.0 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
```

설치 폴더 확인

.NET이 설치되어 있지만 운영 체제 또는 사용자 프로필의 `PATH` 변수에 추가되지 않을 수 있습니다. 이 경우 이전 섹션의 명령이 작동하지 않을 수 있습니다. 또는 .NET 설치 폴더가 있는지 확인할 수 있습니다.

설치 관리자 또는 스크립트에서 .NET을 설치하면 표준 폴더에 설치됩니다. .NET을 설치하는 데 사용하는 설치 관리자 또는 스크립트는 대부분 다른 폴더에 설치할 수 있는 옵션을 제공합니다. 다른 폴더에 설치하도록 선택한 경우 폴더 경로의 시작을 조정합니다.

- **dotnet 실행 파일**
  C:\program files\dotnet\dotnet.exe
- .NET SDK
  C:\program files\dotnet\sdk\{version}
- **.NET 런타임**
  C:\프로그램 파일\dotnet\shared\{runtime-type}\{version}

추가 정보

명령을 `dotnet --info`사용하여 SDK 버전과 런타임 버전을 모두 볼 수 있습니다. 운영 체제 버전 및 RID(런타임 식별자)와 같은 다른 환경 관련 정보도 가져옵니다.

다음 단계

- Windows용 .NET 런타임 및 SDK를 설치합니다.
- macOS용 .NET 런타임 및 SDK를 설치합니다.

- Linux용 .NET 런타임 및 SDK를 설치합니다.

참고하십시오

- 설치된 .NET Framework 버전 확인

Last updated on 2025. 11. 07.

.NET 소개

.NET은 다양한 종류의 애플리케이션을 빌드하기 위한 무료 플랫폼 간 오픈 소스 개발자 플랫폼입니다. C#이 가장 인기 있는 여러 언어로 작성된 프로그램을 실행할 수 있습니다. 많은 대규모 앱에서 프로덕션에 사용되는 고성능 런타임에 의존합니다.

.NET을 다운로드하고 첫 번째 앱 작성을 시작하는 방법을 알아보려면 시작하기를 참조하세요.

.NET 플랫폼은 생산성, 성능, 보안 및 안정성을 제공하도록 설계되었습니다. 가비지 수집기(GC)를 통해 자동 메모리 관리를 제공합니다. GC 및 엄격한 언어 컴파일러를 사용하기 때문에 형식이 안전하며 메모리가 안전합니다. 동시성을 `async` 기본 형식을 통해 `await` / `Task` 제공합니다. 여기에는 광범위한 기능이 있고 여러 운영 체제 및 칩 아키텍처의 성능에 최적화된 대규모 라이브러리 집합이 포함되어 있습니다.

.NET에는 다음과 같은 디자인 포인트가 있습니다.

- **생산성은** 런타임, 라이브러리, 언어 및 도구가 모두 개발자 사용자 환경에 기여하는 전체 스택입니다.
- **안전 코드** 는 기본 컴퓨팅 모델이지만 안전하지 않은 코드 는 더 많은 수동 최적화를 가능하게 합니다.
- **정적 코드와 동적 코드** 가 모두 지원되어 광범위한 고유 시나리오 집합을 사용할 수 있습니다.
- **네이티브 코드 상호운용 및 하드웨어 본질 기능**은 비용이 낮고 원시 API 및 명령에 대한 액세스 충실도가 높습니다.
- **코드는 플랫폼** (OS 및 칩 아키텍처)에서 이식 가능하지만 플랫폼 대상을 지정하면 특수화 및 최적화가 가능합니다.
- **프로그래밍 도메인(클라우드, 클라이언트, 게임)에서의 적응성**은 범용 프로그래밍 모델의 특수한 구현으로 실현됩니다.
- OpenTelemetry 및 gRPC와 같은 **업계 표준**은 맞춤형 솔루션보다 선호됩니다.

.NET은 Microsoft와 글로벌 커뮤니티가 공동으로 유지 관리합니다. 정기적인 업데이트를 통해 사용자는 안전하고 안정적인 애플리케이션을 프로덕션 환경에 배포할 수 있습니다.

구성 요소

.NET에는 다음 구성 요소가 포함됩니다.

- 런타임 - 애플리케이션 코드를 실행합니다.
- 라이브러리 - JSON 구문 분석과 같은 유틸리티 기능을 제공합니다.
- 컴파일러 - C# (및 기타 언어) 소스 코드를 (런타임) 실행 코드로 컴파일합니다.
- SDK 및 기타 도구 - 최신 워크플로를 사용하여 앱을 빌드하고 모니터링할 수 있습니다.
- 앱 스택 - ASP.NET Core 및 Windows Forms와 같이 앱을 작성할 수 있습니다.

런타임, 라이브러리 및 언어는 .NET 스택의 핵심 요소입니다. .NET 도구와 같은 상위 수준 구성 요소 및 앱 스택(예: ASP.NET Core)은 이러한 핵심 요소를 기반으로 합니다. C#은 .NET의 기본 프로그래밍 언어이며 대부분의 .NET은 C#으로 작성됩니다.

C#은 개체 지향이며 런타임은 개체 방향을 지원합니다. C#에는 가비지 수집이 필요하며 런타임은 추적 가비지 수집기를 제공합니다. 라이브러리(및 앱 스택)는 개발자가 직관적인 워크플로에서 알고리즘을 생산적으로 작성할 수 있도록 하는 개념 및 개체 모델로 이러한 기능을 형성합니다.

핵심 라이브러리는 수천 가지 형식을 노출하며, 그 중 상당수는 C# 언어와 통합되고 연료가 됩니다. 예를 들어 C#의 `foreach` 문을 사용하면 임의의 컬렉션을 열거할 수 있습니다. 패턴 기반 최적화를 사용하면 컬렉션을 `List<T>` 간단하고 효율적으로 처리할 수 있습니다. 리소스 관리가 가비지 수집에 맡겨질 수도 있지만, `IDisposable` 및 `using` 명령문에서의 직접적인 언어 지원을 통해 신속한 정리가 가능합니다.

동시에 여러 작업을 수행할 수 있도록 지원하는 것은 거의 모든 워크로드의 기본 사항입니다. UI 응답성을 유지하면서 백그라운드 처리를 수행하는 클라이언트 애플리케이션, 수천 개의 동시 요청을 처리하는 서비스, 수많은 동시 자극에 응답하는 디바이스 또는 컴퓨팅 집약적 작업의 처리를 병렬화하는 고성능 컴퓨터가 될 수 있습니다. 비동기 프로그래밍 지원은 C# 프로그래밍 언어의 일류 기능으로, 비동기 작업을 쉽게 작성하고 작성할 수 있는 키워드와 언어가 제공하는 모든 제어 흐름 구문의 모든 이점을 누릴 수 있는 키워드를 제공합니다 `async` `await` .

형식 시스템은 안전, 설명성, 역동성 및 네이티브 상호 운용을 균등하게 지원하는 널리 제공됩니다. 무엇보다도 형식 시스템은 개체 지향 프로그래밍 모델을 사용하도록 설정합니다. 여기에는 형식, (단일 기본 클래스) 상속, 인터페이스(기본 메서드 구현 포함) 및 개체 방향에서 허용하는 모든 형식 계층화에 대한 합리적인 동작을 제공하는 가상 메서드 디스패치가 포함됩니다. 제네릭 형식 은 유비쿼터스이며 하나 이상의 형식으로 클래스를 특수화할 수 있습니다.

.NET 런타임은 가비지 수집기를 통해 자동 메모리 관리를 제공합니다. 모든 언어의 경우 메모리 관리 모델이 가장 정의되는 특성일 수 있습니다. .NET 언어의 경우 마찬가지입니다. .NET에는 자체 튜닝 및 추적 GC가 포함되어 있습니다. 일반적인 경우에는 "자동으로" 사용하도록 하면서, 보다 극단적인 워크로드에는 구성 옵션을 제공하는 것을 목표로 합니다. 현재 GC는 수많은 워크로드에서 수년간의 투자와 학습의 결과입니다.

값 형식 및 스택 할당 메모리 블록은 .NET의 GC 관리형 형식과는 달리, 데이터와 네이티브 플랫폼 상호 운용성에 대해 보다 직접적이고 낮은 수준의 제어를 제공합니다. 정수 형식과 같은 .NET의 기본 형식 대부분은 값 형식이며 사용자는 비슷한 의미 체계를 사용하여 고유한 형식을 정의할 수 있습니다. 값 형식은 을 통해 완전히 지원됩니다. NET의 제네릭 시스템은 제네릭 형식과 같은 `List<T>` 값 형식 컬렉션의 오버헤드가 없는 플랫 메모리 표현을 제공할 수 있음을 의미합니다.

리플렉션 은 "데이터로서의 프로그램" 패러다임으로, 프로그램의 한 부분이 어셈블리, 형식 및 멤버 측면에서 다른 부분을 동적으로 쿼리하고 호출할 수 있도록 합니다. 지연 바인딩된 프로그

래밍 모델 및 도구에 특히 유용합니다.

예외는 .NET의 기본 오류 처리 모델입니다. 예외는 오류 정보를 메서드 서명에 표시하거나 모든 메서드에서 처리할 필요가 없다는 이점이 있습니다. 적절한 예외 처리는 애플리케이션 안정성에 필수적입니다. 앱이 충돌하지 않도록 하려면 코드에서 예상되는 예외를 의도적으로 처리할 수 있습니다. 크래시된 앱은 정의되지 않은 동작을 가진 앱보다 더 안정적이고 진단 가능합니다.

ASP.NET Core 및 Windows Forms와 같은 앱 스택은 하위 수준 라이브러리, 언어 및 런타임을 기반으로 빌드하고 활용합니다. 앱 스택은 앱이 생성되는 방식과 실행 수명 주기를 정의합니다.

SDK 및 기타 도구를 사용하면 개발자 데스크톱과 CI(연속 통합) 모두에서 최신 개발자 환경을 사용할 수 있습니다. 최신 개발자 환경에는 코드 빌드, 분석 및 테스트가 포함됩니다. .NET 프로젝트는 NuGet 패키지 복원 및 종속성 빌드를 오케스트레이션하는 단일 `dotnet build` 명령으로 빌드할 수 있습니다.

NuGet은 .NET의 패키지 관리자입니다. 여기에는 많은 시나리오에 대한 기능을 구현하는 수십만 개의 패키지가 포함되어 있습니다. 대부분의 앱은 일부 기능에 대해 NuGet 패키지를 사용합니다. NuGet 갤러리 는 Microsoft에서 유지 관리합니다.

무료 및 오픈 소스

.NET은 무료 오픈 소스이며 .NET Foundation 프로젝트입니다. .NET은 Microsoft와 GitHub의 커뮤니티가 여러 리포지토리 에서 유지 관리합니다.

.NET 원본 및 이진 파일은 MIT 라이선스로 라이선스 가 부여됩니다. 다른 라이선스는 Windows에 적용됩니다 .

지원

.NET은 여러 운영 체제 에서 .NET을 실행할 수 있고 최신 상태로 유지되도록 하는 여러 조직에서 지원됩니다. Arm64, x64 및 x86 아키텍처에서 사용할 수 있습니다.

.NET의 새 버전은 릴리스 및 지원 정책에 따라 매년 11월에 릴리스됩니다. 패치 화요일(두 번째 화요일)에 매월 업데이트 됩니다. 일반적으로 태평양 표준시 오전 10시에 업데이트됩니다.

.NET 에코시스템

.NET의 여러 변형이 존재하며, 각 변형은 기록 및 기술적 이유로 서로 다른 유형의 앱을 지원합니다.

.NET 구현:

- **.NET(Core)** - 최신 .NET. .NET의 플랫폼 간 및 오픈 소스 구현은 클라우드 시대에 맞춰 새롭게 구상되었으며, 기존과의 높은 호환성을 유지합니다. 진화하고 적극적으로 지원됩니다.
- **.NET Framework** - 원래 .NET입니다. Windows 및 Windows Server의 광범위한 기능에 액세스할 수 있습니다. 유지 관리에서 적극적으로 지원됩니다.
- **Mono** - 원래 커뮤니티 및 오픈 소스 .NET입니다. .NET Framework의 플랫폼 간 구현입니다.

다음 단계

- .NET 자습서 선택
- C# 둘러보기
- F# 둘러보기

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 14.

.NET으로 앱 빌드

2025. 10. 11.

.NET은 클라이언트, 클라우드 및 게임을 비롯한 다양한 종류의 앱을 빌드할 수 있습니다.

클라우드 앱

- Aspire
- 서버리스 함수
- 웹 및 마이크로 서비스

클라이언트 앱

- 모바일
- 게임
- 데스크톱 앱

기타 앱 유형

- 콘솔 앱
- IoT(사물 인터넷)
- 기계 학습
- Windows 서비스

Microsoft .NET 언어 전략

2025. 06. 17.

Microsoft는 .NET 플랫폼에서 C#, F# 및 Visual Basic의 3개 언어를 제공합니다. 이 문서에서는 각 언어에 대한 전략에 대해 알아봅니다. 이러한 전략이 우리를 안내하는 방법과 각 언어에 대해 자세히 알아볼 수 있는 방법에 대한 추가 문서에 대한 링크를 찾습니다.

C# (프로그래밍 언어)

C#은 성능이 뛰어난 코드를 작성하면서 개발자의 생산성을 높이는 플랫폼 간 범용 언어입니다. 수백만 명의 개발자가 있는 C#은 가장 인기 있는 .NET 언어입니다. C#은 에코시스템 및 모든 .NET 워크로드를 광범위하게 지원합니다. 개체 지향 원칙에 기반하여 함수형 프로그래밍을 비롯한 다른 패러다임의 많은 기능을 통합합니다. 하위 수준 기능은 안전하지 않은 코드를 작성하지 않고도 고효율 시나리오를 지원합니다. 대부분의 .NET 런타임 및 라이브러리는 C#으로 작성되며 C#의 발전은 모든 .NET 개발자에게 도움이 되는 경우가 많습니다.

우리의 C#을 위한 전략

개발자의 변화하는 요구를 충족하고 최첨단 프로그래밍 언어로 남도록 C#을 계속 발전시킬 것입니다. 우리는 .NET 라이브러리, 개발자 도구 및 워크로드 지원을 담당하는 팀과 협력하여 열렬하고 광범위하게 혁신할 것이며 언어의 정신에 주의를 기울이게 될 것입니다. C#이 사용되는 도메인의 다양성을 인식하면 모든 개발자 또는 대부분의 개발자에게 도움이 되는 언어 및 성능 향상을 선호하며 이전 버전과의 호환성에 대한 높은 노력을 유지합니다. 우리는 디자인 결정의 관리를 유지하면서 더 넓은 .NET 에코시스템의 역량을 지속적으로 강화하고 C#의 미래에 그 역할을 확장할 것입니다.

이 전략이 C# 가이드에서 우리를 안내하는 방법에 대해 자세히 읽을 수 있습니다.

F#

F#은 식 기반이며 기본적으로 변경할 수 없는 간결하고 강력하며 성능이 좋은 언어입니다. 표현력, 단순성 및 우아함에 중점을 두고 있으며.NET에 대한 실용적인 함수 우선 접근 방식을 높이 평가하는 수천 명의 개발자가 사용합니다. F#은 .NET의 모든 기능을 제공하며 혼합 언어 솔루션에 대한 C#과 잘 작동합니다. 커뮤니티는 컴파일러 및 런타임뿐만 아니라 광범위한 F# 도구 및 프레임워크에 상당한 기여를 합니다.

F에 대한 전략#

F# 진화를 주도하고 언어 리더십 및 거버넌스를 통해 F# 에코시스템을 지원할 것입니다. F# 언어 및 개발자 환경을 개선하기 위해 커뮤니티 기여를 장려할 것입니다. 우리는 중요한 라이브러리, 개발자 도구 및 워크로드 지원을 제공하기 위해 커뮤니티에 계속 의존할 것입니다. 언어가 발전함에 따라 F#은 .NET 플랫폼 개선을 지원하고 새로운 C# 기능과의 상호 운용성을 유지합니다. 새 개발자 및 조직의 F# 진입 장벽을 낮추고 새 도메인으로의 범위를 넓히기 위해 언어, 도구 및 설명서 전반에 걸쳐 작업할 것입니다.

이 전략이 F# 가이드에서 우리를 안내하는 방법에 대해 자세히 읽을 수 있습니다.

Visual Basic (비주얼 베이직 언어)

VB(Visual Basic)는 간결성보다 명확성을 선호하는 접근 가능한 언어로 오랜 역사를 가지고 있습니다. 수십만 명의 개발자가 VB가 오랫동안 뛰어난 도구와 사용 편의성을 개척해 온 기존의 Windows 기반 클라이언트 워크로드 에 집중되어 있습니다. 오늘날의 VB 개발자는 증가하는 .NET 에코시스템 및 지속적인 도구 개선과 결합된 안정적이고 성숙한 개체 지향 언어의 이점을 누릴 수 있습니다. 일부 .NET 워크로드는 VB에서 지원되지 않으며 VB 개발자는 이러한 시나리오에 C#을 사용하는 것이 일반적입니다.

Visual Basic에 대한 전략

Visual Basic은 안정적인 디자인으로 간단하고 접근 가능한 언어로 유지됩니다. .NET의 핵심 라이브러리는 Visual Basic을 지원하며, .NET 런타임 및 라이브러리의 많은 개선 사항이 Visual Basic에 자동으로 도움이 됩니다. C# 또는 .NET 런타임에서 언어 지원이 필요한 새로운 기능을 도입하는 경우 Visual Basic은 일반적으로 소비 전용 접근 방식을 채택하고 새 구문을 사용하지 않습니다. Visual Basic은 새 워크로드로 확장되지 않습니다. 특히 Windows Forms 및 라이브러리와 같은 핵심 Visual Basic 시나리오에서 Visual Studio의 환경과 C#과의 상호 운용성에 계속 투자할 것입니다.

이 전략이 Visual Basic 가이드에서 안내하는 방법에 대해 자세히 확인할 수 있습니다.

.NET 구현체

.NET 앱은 하나 이상의 .NET 구현을 위해 개발되었습니다. .NET 구현에는 .NET Framework, .NET 5+(및 .NET Core), Mono가 포함됩니다.

.NET의 각 구현체에는 다음 구성 요소가 포함됩니다.

- 하나 이상의 런타임(예:.NET Framework CLR 및 .NET 8 CLR).
- 클래스 라이브러리(예: .NET Framework 기본 클래스 라이브러리 및 .NET 8 기본 클래스 라이브러리).
- 필요에 따라 ASP.NET, Windows Forms및 WPF(Windows Presentation Foundation) 하나 이상의 애플리케이션 프레임워크가 .NET Framework 및 .NET 5 이상에 포함됩니다.
- 선택 사항: 개발 도구 일부 개발 도구는 여러 구현체에서 공통적으로 사용할 수 있음.

세 가지 주요 .NET 구현이 있습니다.

- .NET(Core)
- .NET Framework
- 모노

.NET(Core)

이전에 .NET Core라고도 하는 .NET은 현재 기본 구현입니다. .NET은 Windows 데스크톱 앱, 플랫폼 간 콘솔 앱, 클라우드 서비스 및 웹 사이트와 같은 여러 플랫폼과 많은 워크로드를 지원하는 단일 코드 베이스를 기반으로 합니다. .NET WebAssembly 빌드 도구와 같은 일부 워크로드는 선택적 설치로 사용할 수 있습니다.

.NET 10은 이 .NET 구현의 최신 버전입니다. .NET Standard를 대상으로 하는 코드가 .NET에서 실행되도록 .NET Standard를 구현합니다. ASP.NET Core, Windows Forms 및 WPF(Windows Presentation Foundation) 모두 .NET에서 실행됩니다.

자세한 내용은 다음 리소스를 참조하세요.

- .NET 소개
- 서버 앱 용 .NET 및 .NET Framework
- .NET 5+ 및 .NET Standard

.NET Framework

.NET Framework는 2002년부터 있었던 원래 .NET 구현입니다. 버전 4.5 이상은 .NET Standard를 구현하므로 .NET Standard를 대상으로 하는 코드는 .NET Framework의 해당 버전에서 실행할 수 있습니다. Windows Forms 및 WPF를 사용하는 Windows 데스크톱 개발용 API와 같이 Windows

관련 추가 API가 포함되어 있습니다. .NET Framework는 Windows 데스크톱 애플리케이션을 구축을 위해 최적화되어 있습니다.

자세한 내용은 .NET Framework 가이드를 참조하세요.

모노

원조 커뮤니티와 오픈 소스 .NET. Mono는 .NET Framework의 플랫폼 간 구현입니다. Android, macOS, iOS, tvOS 및 watchOS에서 Xamarin 애플리케이션(현재 지원되지 않음)을 구동하고 주로 작은 공간을 중심으로 하는 런타임입니다. 또한 Mono에서는 Unity 엔진으로 만든 게임이 동작합니다.

사용 가능한 모든 .NET Standard 버전을 지원합니다.

지금까지 Mono는 .NET Framework의 더 큰 API를 구현했으며 Unix에서 가장 인기 있는 기능 중 일부를 에뮬레이트했습니다. 경우에 따라 Unix에서 해당 기능을 사용하는 .NET 애플리케이션을 실행하는 데도 사용됩니다.

자세한 내용은 Mono 설명서를 참조하세요.

Last updated on 2026. 02. 24.

.NET 클래스 라이브러리

2025. 06. 17.

클래스 라이브러리는 .NET의 공유 라이브러리 개념입니다 . 이 기능을 사용하면 여러 애플리케이션에서 사용할 수 있는 모듈로 유용한 기능을 구성할 수 있습니다. 애플리케이션 시작 시 필요하지 않거나 알 수 없는 기능을 로드하는 수단으로 사용할 수도 있습니다. 클래스 라이브러리는 .NET 어셈블리 파일 형식을 사용하여 설명됩니다.

사용할 수 있는 클래스 라이브러리에는 다음 세 가지 유형이 있습니다.

- **플랫폼별** 클래스 라이브러리는 지정된 플랫폼의 모든 API(예: Windows의 .NET Framework)에 액세스할 수 있지만 해당 플랫폼을 대상으로 하는 앱 및 라이브러리에서만 사용할 수 있습니다.
- **이식 가능한** 클래스 라이브러리는 API의 하위 집합에 액세스할 수 있으며 여러 플랫폼을 대상으로 하는 앱 및 라이브러리에서 사용할 수 있습니다.
- **.NET Standard** 클래스 라이브러리는 플랫폼별 및 이식 가능한 라이브러리 개념을 둘 다 최상으로 제공하는 단일 모델로 병합한 것입니다.

플랫폼별 클래스 라이브러리

플랫폼별 라이브러리는 단일 .NET 플랫폼(예: Windows의 .NET Framework)에 바인딩되므로 알려진 실행 환경에 상당한 종속성을 사용할 수 있습니다. 이러한 환경은 알려진 API 집합(.NET 및 OS API)을 노출하고 예상 상태(예: Windows 레지스트리)를 유지 관리하고 노출합니다.

플랫폼별 라이브러리를 만드는 개발자는 기본 플랫폼을 완전히 활용할 수 있습니다. 라이브러리는 지정된 플랫폼에서만 실행되므로 플랫폼 검사 또는 기타 형태의 조건부 코드가 필요하지 않습니다(여러 플랫폼에 대한 모듈로 단일 소싱 코드).

플랫폼별 라이브러리는 .NET Framework의 기본 클래스 라이브러리 형식이었습니다. 다른 .NET 구현이 등장하더라도 플랫폼별 라이브러리는 여전히 주요 라이브러리 유형입니다.

이식 가능한 클래스 라이브러리

이식 가능한 라이브러리는 여러 .NET 구현에서 지원됩니다. 알려진 실행 환경에 대한 종속성을 여전히 사용할 수 있지만 환경은 구체적인 .NET 구현 집합의 교집합에 의해 생성되는 가상 환경입니다. 노출된 API 및 플랫폼 가정은 플랫폼별 라이브러리에서 사용할 수 있는 항목의 하위 집합입니다.

이식 가능한 라이브러리를 만들 때 플랫폼 구성을 선택합니다. 플랫폼 구성은 지원해야 하는 플랫폼 집합입니다(예: .NET Framework 4.5 이상, Windows Phone 8.0 이상). 지원할 플랫폼을 더 많이 선택할수록 지원할 수 있는 API와 플랫폼 가정이 줄어들어 가장 기본적인 수준으로 맞춰지

게 됩니다. 사람들이 종종 "더 낫다"고 생각하지만 지원되는 플랫폼이 많을수록 사용 가능한 API가 적기 때문에 처음에는 이러한 특성이 혼동될 수 있습니다.

많은 라이브러리 개발자가 하나의 원본(조건부 컴파일 지시문 사용)에서 이식 가능한 라이브러리로 여러 플랫폼별 라이브러리를 생성하는 것으로 전환했습니다. 이식 가능한 라이브러리 내에서 플랫폼별 기능에 액세스하는 방법에는 여러 가지가 있으며, 이 시점에서는 bait-and-switch가 가장 널리 받아들여지는 기술입니다.

.NET 표준 클래스 라이브러리

.NET Standard 라이브러리는 플랫폼별 이식 가능한 라이브러리 개념을 대체합니다. 기본 플랫폼(가상 플랫폼 또는 플랫폼 교집합 없음)의 모든 기능을 노출한다는 점에서 플랫폼별로 다릅니다. 모든 지원 플랫폼에서 작동한다는 점에서 이식 가능합니다.

.NET Standard는 라이브러리 *계약 집합을 노출합니다*. .NET 구현은 각 계약을 완전히 지원하거나 전혀 지원하지 않아야 합니다. 따라서 각 구현은 .NET Standard 계약 집합을 지원합니다. 각 .NET Standard 클래스 라이브러리는 계약 종속성을 지원하는 플랫폼에서 지원됩니다.

.NET Standard는 .NET Framework의 전체 기능을 노출하지 않지만(목표도 아님) 라이브러리는 이식 가능한 클래스 라이브러리보다 더 많은 API를 노출합니다.

다음 구현에서는 .NET Standard 라이브러리를 지원합니다.

- .NET 코어
- .NET Framework
- 모노
- UWP(유니버설 Windows 플랫폼)

자세한 내용은 .NET Standard 참조하세요.

Mono 클래스 라이브러리

클래스 라이브러리는 앞에서 설명한 세 가지 유형의 라이브러리를 포함하여 Mono에서 지원됩니다. Mono는 종종 .NET Framework의 플랫폼 간 구현으로 간주됩니다. 플랫폼별 .NET Framework 라이브러리는 수정 또는 다시 컴파일 없이 Mono 런타임에서 실행할 수 있기 때문입니다. 이 특성은 이식 가능한 클래스 라이브러리를 만들기 전에 적용되었으므로 .NET Framework와 Mono 간에 이진 이식성을 사용하도록 설정하는 것이 분명했습니다(한 방향으로만 작동함).

.NET Standard

.NET Standard는 여러 .NET 구현에서 사용할 수 있는 .NET API의 공식 사양입니다. .NET Standard의 동기는 .NET 생태계에서 더 큰 균일성을 확립하는 것이었습니다. .NET 5 이상 버전은 대부분의 시나리오에서 .NET 표준의 필요성을 없애는 균일성을 설정하는 다른 접근 방식을 채택합니다. 그러나 .NET Framework와 .NET Core와 같은 다른 .NET 구현 간에 코드를 공유하려는 경우 라이브러리는 .NET Standard 2.0을 대상으로 해야 합니다. .NET Standard의 새 버전은 릴리스되지 않지만 .NET 5 이상 버전은 .NET Standard 2.1 이하 버전을 계속 지원합니다.

.NET 5 이상과 .NET 표준 중에서 선택하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 .NET 5 이상 및 .NET Standard를 참조하세요.

.NET 표준 버전

.NET 표준에는 버전이 있습니다. 새로운 버전마다 더 많은 API가 추가됩니다. 라이브러리가 특정 버전의 .NET Standard에 대해 빌드되는 경우 해당 버전의 .NET Standard 이상을 구현하는 .NET 구현에서 실행할 수 있습니다.

더 높은 버전의 .NET Standard를 대상으로 지정하면 라이브러리에서 더 많은 API를 사용할 수 있지만 최신 버전의 .NET만 사용할 수 있습니다. 낮은 버전을 대상으로 하면 사용 가능한 API가 줄어들지만 라이브러리가 더 많은 곳에서 실행될 수 있음을 의미합니다.

.NET 표준 버전 선택

1.0

.NET Standard 1.0에는 37,118개의 사용 가능한 API 중 7,949가 있습니다.

테이블 확장

.NET 구현	버전 지원
.NET 및 .NET Core	1.0, 1.1, 2.0, 2.1, 2.2, 3.0, 3.1, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
모노	4.6, 5.4, 6.4
Xamarin.iOS	10.0, 10.14, 12.16
Xamarin. Mac	3.0, 3.8, 5.16
Xamarin. 안 드 로이드	7.0, 8.0, 10.0

.NET 구현	버전 지원
Universal Windows Platform	8.0, 8.1, 10.0, 10.0.16299, 추후 결정
유니티	2018.1

자세한 내용은 .NET Standard 1.0 참조하세요. 대화형 테이블은 .NET 표준 버전 참조하세요.

.NET Standard의 어느 버전을 대상으로 지정할지

.NET Standard를 대상으로 하는 경우 이전 버전을 지원하지 않는 한 .NET Standard 2.0을 대상으로 하는 것이 좋습니다. 대부분의 범용 라이브러리는 .NET Standard 2.0 이외의 API가 필요하지 않아야 하며 .NET Framework는 .NET Standard 2.1을 지원하지 않습니다. .NET Standard 2.0은 모든 최신 플랫폼에서 지원되며 하나의 대상으로 여러 플랫폼을 지원하는 것이 좋습니다.

.NET Standard 1.x를 지원해야 하는 경우 .NET Standard 2.0도 함께 대상으로 지정할 것을 권장합니다. .NET Standard 1.x는 큰 패키지 종속성 그래프를 만들고 프로젝트가 빌드될 때 많은 패키지가 다운로드되는 세분화된 NuGet 패키지 집합으로 배포됩니다. 자세한 내용은 이 문서의 뒷부분에 있는 Cross-platform targeting 및 .NET 5+ 및 .NET Standard를 참조하세요.

> ⓘ **참고 항목**
>
> .NET 9부터 프로젝트가 .NET Standard 1.x를 대상으로 하는 경우 빌드 경고가 내보내집니다. 자세한 내용은 .NET Standard 1.x 대상에 대해 내보내는 Warning 참조하세요.

.NET 표준 버전 관리 규칙

두 가지 기본 버전 관리 규칙이 있습니다.

- 가산: .NET 표준 버전은 논리적으로 동심원입니다. 상위 버전은 이전 버전의 모든 API를 통합합니다. 버전 간에 큰 차이는 없습니다.
- 변경할 수 없음: 배송되면 .NET 표준 버전이 고정됩니다.

2.1 이후 새로운 .NET 표준 버전은 없습니다. 자세한 내용은 이 문서의 뒷부분에 있는 .NET 5 이상 및 .NET Standard를 참조하세요.

규격

.NET 표준 사양은 표준화된 API 집합입니다. 사양은 .NET 구현자, 특히 Microsoft(.NET Framework, .NET Core 및 Mono 포함) 및 Unity에서 유지 관리됩니다.

공식 아티팩트

공식 규격은 표준의 일부인 API를 정의하는 *.cs* 파일 세트입니다. 현재 보관된 dotnet/standard 리포지토리의 ref 디렉터리는 .NET 표준 API를 정의합니다.

NETStandard.Library 메타패키지(source)는 하나 이상의 .NET 표준 버전을 정의하는 라이브러리 집합을 설명합니다.

`System.Runtime` 등의 지정된 구성 요소는 다음에 대해 설명합니다.

- .NET 표준의 일부(범위만 해당).
- .NET Standard의 다양한 버전이 해당 범위에 사용됩니다.

보다 편리하게 읽을 수 있고 특정 개발자 시나리오(예: 컴파일러 사용)를 지원할 수 있도록 파생 아티팩트가 제공됩니다.

- Markdown의 API 목록
- NuGet 패키지로 배포되고 NETStandard.Library 메타패키지에서 참조되는 참조 어셈블리입니다.

패키지 표현

.NET 표준 참조 어셈블리의 기본 배포 차량은 NuGet 패키지입니다. 구현은 각 .NET 구현에 적합한 다양한 방법으로 제공됩니다.

NuGet 패키지는 하나 이상의 프레임워크를 대상으로 합니다. .NET Standard 패키지는 ".NET 표준" 프레임워크를 대상으로 합니다. `netstandard` 컴팩트 TFM(대상 프레임워크 모니커)을 사용하여 .NET 표준 프레임워크를 대상으로 지정할 수 있습니다(예: `netstandard1.4`). .NET 여러 구현에서 실행하려는 라이브러리는 .NET Standard 프레임워크를 대상으로 해야 합니다. 가장 광범위한 API 집합의 경우 사용 가능한 API 수가 .NET Standard 1.6과 2.0 사이에서 두 배 이상 증가하므로 `netstandard2.0` 대상으로 합니다.

NETStandard.Library 메타패키지는 .NET 표준을 정의하는 NuGet 패키지의 전체 집합을 참조합니다. `netstandard`를 대상으로 지정하는 가장 일반적인 방법은 이 메타패키지를 참조하는 것입니다. .NET Standard를 정의하는 최대 40개의 .NET 라이브러리 및 관련 API에 대한 액세스를 설명하고 제공합니다. 추가 API에 대한 액세스를 얻기 위해 `netstandard`를 대상으로 하는 추가 패키지를 참조할 수 있습니다.

버전 관리

사양은 단수형이 아니라 선형적으로 버전이 지정되는 API 집합입니다. 첫 번째 버전의 표준에서는 API의 기준 집합을 설정합니다. 이후 버전에서는 API를 추가하고 이전 버전에서 정의한 API를 상속받습니다. 표준에서 API 제거와 관련하여 정해진 규정은 없습니다.

.NET 표준은 .NET 구현에만 해당되지 않으며 해당 구현의 버전 관리 체계와도 일치하지 않습니다.

앞에서 설명한 것처럼 2.1 이후의 새로운 .NET 표준 버전은 없습니다.

대상 .NET 표준

프레임워크와 `netstandard` 메타패키지의 조합을 사용하여 `NETStandard.Library` 빌드할 수 있습니다.

.NET 프레임워크 호환성 모드

.NET Standard 2.0부터 .NET Framework 호환성 모드가 도입되었습니다. 이 호환성 모드를 사용하면 .NET Standard 프로젝트가 .NET Standard용으로 컴파일된 것처럼 .NET Framework 라이브러리를 참조할 수 있습니다. .NET Framework 라이브러리 참조는 Windows Presentation Foundation(WPF) API를 사용하는 라이브러리와 같은 모든 프로젝트에서는 작동하지 않습니다.

자세한 내용은 .NET Framework 호환성 모드 참조하세요.

.NET 표준 라이브러리 및 Visual Studio

Visual Studio .NET 표준 라이브러리를 빌드하려면 Visual Studio 2019 이상 또는 Visual Studio 2017 버전 15.3 이상이 Windows 설치되어 있는지 확인합니다.

프로젝트에서 .NET Standard 2.0 라이브러리만 사용해야 하는 경우 Visual Studio 2015에서도 사용할 수 있습니다. 그러나 NuGet 클라이언트 3.6 이상을 설치해야 합니다. NuGet 다운로드 페이지에서 Visual Studio 2015용 NuGet 클라이언트를 다운로드할 수 있습니다.

.NET 5 이상 및 .NET Standard

.NET 5, .NET 6, .NET 7, .NET 8, .NET 9 및 .NET 10은 Windows 데스크톱 앱 및 플랫폼 간 콘솔 앱, 클라우드 서비스 및 웹 사이트에 사용할 수 있는 일관된 기능 및 API 집합이 있는 단일 제품입니다. 예를 들어 .NET 10 TFM은 다음과 같은 광범위한 시나리오를 반영합니다.

- `net10.0`

  이 TFM은 어디서나 실행되는 코드에 대한 것입니다. 몇 가지 예외를 제외하고 플랫폼 간에 작동하는 기술만 포함됩니다.

- `net10.0-windows`

이는 OS 특화 TFM의 예로서, `net10.0`이(OS 특정 기능성을 추가하는 모든 것에 대한) 참조입니다.

`netx.0`과 `netstandard`를 대상 지정하는 경우

.NET 표준 2.0 이상을 대상으로 하는 기존 코드의 경우 TFM을 `net8.0` 이상 TFM으로 변경할 필요가 없습니다. .NET 8, .NET 9 및 .NET 10은 .NET Standard 2.1 이하를 구현합니다. .NET Standard에서 .NET 8 이상으로 대상을 변경해야 하는 유일한 이유는 더 많은 런타임 기능, 언어 기능 또는 API에 대한 액세스 권한을 얻는 것입니다. 예를 들어 C# 9를 사용하려면 .NET 5 이상 버전을 대상으로 지정해야 합니다. 다중 대상 .NET 및 .NET Standard를 사용하여 최신 기능에 액세스할 수 있으며 다른 .NET 구현에서 라이브러리를 계속 사용할 수 있습니다.

> ⓘ **참고 항목**
>
> 프로젝트가 .NET Standard 1.x를 대상으로 하는 경우, .NET Standard 2.0 또는 .NET 8 이상으로 리타겟팅하는 것이 좋습니다. 자세한 내용은 .NET Standard 1.x 대상에 대해 내보내는 Warning 참조하세요.

.NET 5 이상에 대한 새 코드에 대한 몇 가지 지침은 다음과 같습니다.

- 앱 구성 요소

  라이브러리를 사용하여 애플리케이션을 여러 구성 요소로 세분화하는 경우 `net10.0`을 대상으로 지정하는 것이 좋습니다. 간단히 하기 위해 애플리케이션을 구성하는 모든 프로젝트를 동일한 버전의 .NET 유지하는 것이 가장 좋습니다. 그러면 어디에서나 동일한 BCL 기능을 사용할 수 있습니다.

- 재사용 가능한 라이브러리

  NuGet에 제공하려는 재사용 가능한 라이브러리를 구축하는 경우 도달 범위와 사용 가능한 기능 집합 간의 절충을 고려하세요. .NET Standard 2.0은 .NET Framework에서 지원하는 최신 버전이므로 상당히 큰 기능 집합을 사용하여 적절한 도달 범위를 제공합니다. 도달 범위를 최소화하기 위해 사용 가능한 기능 집합을 제한하므로 .NET Standard 1.x를 대상으로 지정하지 않는 것이 좋습니다.

  .NET Framework를 지원할 필요가 없는 경우 .NET Standard 2.1 또는 .NET 10을 대상으로 지정할 수 있습니다. 우리는 .NET 표준 2.1을 건너뛰고 .NET 10으로 바로 이동하는 것을 권장합니다. 가장 널리 사용되는 라이브러리는 .NET Standard 2.0 및 .NET 5 이상 모두에 대해 다중 대상입니다. .NET Standard 2.0을 지원하면 가장 많은 도달 범위를 제공하며, .NET 5+를 지원하면 이미 .NET 5 이상에 있는 고객을 위해 최신 플랫폼 기능을 활용할 수 있습니다.

.NET 표준 문제

.NET 5 이상 버전이 플랫폼 및 워크로드 간에 코드를 공유하는 더 나은 방법인 이유를 설명하는 데 도움이 되는 .NET Standard의 몇 가지 문제는 다음과 같습니다.

- 새 API 추가의 속도 저하

  .NET Standard는 모든 .NET 구현이 지원해야 하는 API 집합으로 만들어졌으므로 새 API를 추가하는 제안에 대한 검토 프로세스가 있었습니다. 목표는 모든 현재 및 미래의 .NET 플랫폼에서 구현할 수 있는 API만 표준화하는 것이었습니다. 그로 인해 기능이 특정 릴리스를 놓친 경우 표준 버전에 추가되기까지 몇 년을 기다려야 할 수도 있습니다. 그런 다음 .NET Standard의 새 버전이 널리 지원될 때까지 더 오래 기다립니다.

  .NET 5+의 해결책: 기능이 구현되면 공유 코드 베이스로 인해 모든 .NET 5+ 앱 및 라이브러리에서 이미 사용할 수 있습니다. 또한 API 사양과 해당 구현 간에 차이가 없으므로 .NET Standard보다 훨씬 빠르게 새로운 기능을 활용할 수 있습니다.

- 복잡한 버전 관리

  API 규격을 구현과 분리하면 API 규격 버전과 구현 버전 간의 매핑이 복잡해집니다. 이러한 복잡성은 이 문서의 앞부분에 나와 있는 표와 해석 방법에 대한 지침에서 분명히 확인할 수 있습니다.

  .NET 5+ 솔루션: .NET 5+ API 사양과 그 구현 사이에는 구분이 없습니다. 그 결과로 나온 것이 단순화된 TFM 체계입니다. 모든 워크로드에 대해 하나의 TFM 접두사가 있습니다. `net10.0`은 라이브러리, 콘솔 앱 및 웹앱에 사용됩니다. 유일한 변형은 특정 플랫폼의 API를 지정하는 접미사로, `net10.0-windows`와 같은 특정 플랫폼에 적용됩니다. 이 TFM 명명 규칙 덕분에 주어진 앱이 주어진 라이브러리를 사용할 수 있는지 쉽게 알 수 있습니다. .NET Standard와 같은 버전 번호에 해당하는 테이블은 필요하지 않습니다.

- 런타임 시 플랫폼 지원되지 않는 예외

  .NET Standard는 플랫폼별 API를 노출합니다. 코드는 오류 없이 컴파일될 수 있으며 이식 가능하지 않더라도 모든 플랫폼에 이식 가능한 것처럼 보일 수 있습니다. 지정된 API에 대한 구현이 없는 플랫폼에서 실행되면 런타임 오류가 발생합니다.

  .NET 5+의 솔루션: .NET 5 이상 SDK에는 기본적으로 활성화된 코드 분석기가 포함됩니다. 플랫폼 호환성 분석기는 실행하려는 플랫폼에서 지원되지 않는 API의 의도하지 않은 사용을 감지합니다. 자세한 내용은 플랫폼 호환성 분석기를 참조하세요.

.NET 표준이 더 이상 사용되지 않도록 권장되지 않음

.NET 표준은 여러 .NET 구현에서 사용할 수 있는 라이브러리에 여전히 필요합니다. 다음 시나리오에서 .NET Standard를 대상으로 하는 것이 좋습니다.

- `netstandard2.0` 사용하여 .NET Framework와 다른 모든 .NET 구현 간에 코드를 공유합니다.
- `netstandard2.1` 사용하여 Mono와 .NET Core 3.x 간에 코드를 공유합니다.

참고 항목

- .NET 표준 버전(원본)
- .NET 표준 버전(대화형 UI)
- .NET 표준 라이브러리 빌드
- 플랫폼 간 대상 지정

Last updated on 2026. 03. 11.

.NET에 대한 릴리스 및 지원

Microsoft는 .NET용 주요 릴리스, 부 릴리스 및 서비스 업데이트(패치)를 제공합니다. 이 문서에서는 릴리스 유형, 서비스 업데이트, SDK 기능 밴드, 지원 기간 및 지원 옵션에 대해 설명합니다.

> ⓘ **참고 항목**
>
> .NET Framework의 버전 관리 및 지원에 대한 자세한 내용은 **.NET Framework 수명 주기**를 참조하세요.

현재 지원되는 버전

현재 지원되는 .NET 버전은 다음과 같습니다.

- .NET 10(장기 지원) - 2028년 11월까지 지원됩니다.
- .NET 9(표준 용어 지원) - 2026년 11월까지 지원됩니다.
- .NET 8(장기 지원) - 2026년 11월까지 지원됩니다.

지원되는 버전 및 지원 종료 날짜의 전체 목록은 .NET 지원 정책을 참조하세요.

릴리스 유형

버전 번호는 *major.minor.patch* 형식의 각 릴리스 유형에 대한 정보를 인코딩합니다.

다음은 그 예입니다.

- .NET 8 및 .NET 9는 주요 릴리스입니다.
- .NET 9.0.1은 .NET 9의 첫 번째 패치입니다.

릴리스된 .NET 버전 목록 및 .NET 배송 빈도에 대한 자세한 내용은 지원 정책을 참조하세요.

주요 릴리스

주요 릴리스에는 새로운 기능, 새로운 공용 API 노출 영역 및 버그 수정이 포함됩니다. 예를 들어 .NET 8 및 .NET 9가 있습니다. 변경의 특성상 이러한 릴리스들은 호환성을 깨뜨리는 변경이 예상됩니다. 주 릴리스는 이전 주 릴리스와 나란히 설치됩니다.

소규모 릴리스

부 릴리스에는 새로운 기능, 공용 API 노출 영역 및 버그 수정도 포함되며 주요 변경 내용이 있을 수도 있습니다. 이러한 릴리스와 주요 릴리스의 차이점은 변경 내용의 크기가 더 작다는 것입니

다. 부 릴리스는 이전 부 릴리스와 함께 설치됩니다.

서비스 업데이트

서비스 업데이트(패치)는 거의 매달 제공되며, 이러한 업데이트는 보안 및 비보안 버그 수정을 모두 수행합니다. 예를 들어 .NET 9.0.1은 .NET 9의 첫 번째 업데이트입니다. 이러한 업데이트에 보안 수정 사항이 포함되면 항상 두 번째 화요일인 "패치 화요일"에 릴리스됩니다. 서비스 업데이트는 호환성을 유지합니다. 서비스 업데이트는 이전 업데이트를 제거합니다. 예를 들어 .NET 9에 대한 최신 서비스 업데이트는 설치에 성공하면 이전 .NET 9 업데이트를 제거합니다.

기능 밴드(SDK만 해당)

.NET SDK 버전 관리가 .NET 런타임과 다르게 작동합니다. 새 Visual Studio 릴리스에 맞게 .NET SDK 업데이트에는 MSBuild 및 NuGet과 같은 새 기능 또는 새 버전의 구성 요소가 포함되어 있는 경우가 있습니다. 이러한 새로운 기능 또는 구성 요소는 동일한 주 또는 부 버전에 대해 이전 SDK 업데이트에서 제공된 버전과 호환되지 않을 수 있습니다.

이러한 업데이트를 구분하기 위해 .NET SDK는 기능 밴드를 사용합니다. 예를 들어 첫 번째 .NET 9 SDK는 9.0.100이었습니다. 이 릴리스는 9.0.1xx *기능 밴드*에 해당합니다. 기능 대역은 버전 번호의 세 번째 섹션 내 100의 자리 그룹에 정의됩니다. 예를 들어 9.0.101 및 9.0.201은 서로 다른 두 기능 대역의 버전이며 9.0.101 및 9.0.199는 동일한 기능 밴드에 있습니다. .NET SDK 9.0.101이 설치되면 .NET SDK 9.0.100이 있는 경우 컴퓨터에서 제거됩니다. .NET SDK 9.0.200이 동일한 컴퓨터에 설치되어 있으면 .NET SDK 9.0.101이 제거되지 않습니다.

.NET SDK와 Visual Studio 버전 간의 관계에 대한 자세한 내용은 .NET SDK, MSBuild 및 Visual Studio 버전 관리를 참조하세요.

런타임 롤포워드 및 호환성

주 및 부 업데이트는 이전 버전과 함께 설치됩니다. 특정 *major.minor* 버전을 대상으로 빌드된 애플리케이션은 최신 버전을 설치하더라도 대상 런타임을 계속 사용합니다. 기본적으로 .NET 8을 대상으로 하는 앱은 자동으로 .NET 9(주 버전 변경)로 롤백되지 않지만 .NET 8.0을 사용할 수 없는 경우 .NET 8.1과 같은 최신 부 버전으로 롤백할 수 있습니다. 이 동작을 제어하는 방법에 대한 자세한 내용은 프레임워크 종속 앱 롤 포워드 및 자체 포함 배포 런타임 롤 포워드를 참조하세요.

패치 버전 롤 포워드는 자동으로 수행됩니다. .NET 9를 대상으로 빌드된 애플리케이션은 설치된 최신 패치 버전을 사용합니다. 예를 들어 프로젝트에서 .NET 9.0을 지정하고 .NET 9.0.3이 설치된 경우 앱은 .NET 9.0.3을 사용합니다. 이 자동 패치 롤 포워드는 사용 가능한 즉시 보안 수정을 사용해야 하므로 기본값입니다. 이 기본 롤 포워드 동작을 옵트아웃할 수 있습니다.

.NET 버전 수명 주기

.NET 버전은 .NET Framework 릴리스에서 사용하는 고정 수명 주기 대신 최신수명 주기 를 사용합니다. 최신 수명 주기를 사용하는 제품에는 지원 기간이 짧고 릴리스가 더 빈번하게 제공되는 서비스와 유사한 지원 모델이 있습니다.

업데이트 경로

릴리스에 대한 두 가지 지원 트랙이 있습니다.

- STS(*표준 용어 지원*) 릴리스

  이러한 버전은 2년(24개월) 동안 지원됩니다.

  예제:
  - .NET 9는 2024년 11월에 릴리스된 STS 릴리스입니다. 2026년 11월까지 2년 동안 지원합니다.

- LTS(*장기 지원*) 릴리스

  이러한 버전은 최소 3년 동안 지원되며, 해당 날짜가 나면 다음 LTS 릴리스가 출시된 후 1년 동안 지원됩니다.

  예제:
  - .NET 8은 2023년 11월에 릴리스된 LTS 릴리스입니다. 2026년 11월까지 3년간 지원합니다.

LTS와 STS를 번갈아 사용하는 릴리스입니다.

서비스 업데이트는 매월 제공되며 보안 및 비보안(안정성, 호환성 및 안정성) 수정 사항을 모두 포함합니다. 서비스 업데이트는 다음 서비스 업데이트가 릴리스될 때까지 지원됩니다. 서비스 업데이트에는 실행 시 롤 포워드 기능이 있습니다. 즉, 애플리케이션은 기본적으로 설치된 최신 런타임 서비스 업데이트에서 실행됩니다.

릴리스를 선택하는 방법

서비스를 빌드하고 정기적으로 업데이트해야 하는 경우 LTS 또는 STS와 같은 최신 릴리스를 사용하여 .NET에서 제공하는 최신 기능을 최신 상태로 유지합니다.

소비자에게 배포할 클라이언트 애플리케이션을 빌드하는 경우 최신 기능에 액세스하는 것보다 안정성이 더 중요할 수 있습니다. 소비자가 애플리케이션의 다음 버전으로 업그레이드하려면 애플리케이션에서 특정 기간 동안 지원이 필요할 수 있습니다. 이 경우 .NET 8 런타임과 같은 LTS 릴리스가 올바른 옵션이 될 수 있습니다.

> ⓘ **참고 항목**
>
> STS 릴리스인 경우에도 사용 가능한 모든 런타임을 대상으로 지정할 수 있으므로 최신 SDK 버전으로 업그레이드합니다.

서비스 업데이트 지원

.NET 서비스 업데이트는 다음 서비스 업데이트가 릴리스될 때까지 지원됩니다. 릴리스 주기는 매월입니다.

정기적으로 서비스 업데이트를 설치하여 앱이 안전하고 지원되는 상태인지 확인합니다. 예를 들어 .NET 9에 대한 최신 서비스 업데이트가 9.0.1이고 Microsoft가 9.0.2를 배송하는 경우 9.0.1은 더 이상 최신이 아닙니다. .NET 9에 대해 지원되는 서비스 수준은 9.0.2입니다.

각 주 버전 및 부 버전에 대한 최신 서비스 업데이트에 대한 자세한 내용은 .NET 다운로드 페이지를 참조하세요.

지원 종료

지원 종료는 Microsoft에서 제품 버전에 대한 수정, 업데이트 또는 기술 지원을 더 이상 제공하지 않는 날짜를 나타냅니다. 이 날짜 이전에 지원되는 버전으로 이동합니다. 지원되지 않는 버전은 더 이상 애플리케이션 및 데이터를 보호하는 보안 업데이트를 받지 않습니다. 각 버전의 .NET에 대해 지원되는 날짜 범위는 지원 정책을 참조하세요.

지원되는 운영 체제

다양한 운영 체제에서 .NET을 실행할 수 있습니다. 각 운영 체제에는 스폰서 조직(예: Microsoft, Red Hat 또는 Apple)에서 정의한 수명 주기가 있습니다. .NET은 운영 체제 버전에 대한 지원을 추가하고 제거할 때 이러한 수명 주기 일정을 고려합니다.

운영 체제 버전이 지원 종료에 도달하면 Microsoft는 해당 버전에 대한 테스트 및 지원 제공을 중지합니다. 지원을 받으려면 지원되는 운영 체제 버전으로 이동합니다.

자세한 내용은 .NET OS 수명 주기 정책을 참조하세요.

지원 받기

Microsoft 지원 지원과 커뮤니티 지원 중에서 선택할 수 있습니다.

Microsoft 지원

지원 지원은 Microsoft 지원 전문가에게 문의하세요.

지원되는 서비스 수준(사용 가능한 최신 서비스 업데이트)을 사용하여 지원을 받을 수 있습니다. 시스템에서 .NET 8을 실행하고 8.0.11 서비스 업데이트가 릴리스된 경우 첫 번째 단계로 8.0.11을 설치합니다.

커뮤니티 지원

커뮤니티 지원은 커뮤니티 페이지를 참조하세요.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 11. 20.

Ecma 표준

2025. 06. 17.

C# 언어 및 CLI(공용 언어 인프라) 사양은 Ecma International® 을 통해 표준화됩니다. 이러한 표준의 첫 번째 버전은 2001년 12월에 Ecma에 의해 출판되었습니다.

표준에 대한 후속 개정은 TC49-TG2(C#) 및 TC49-TG3(CLI) 작업 그룹이 TC49 (프로그래밍 언어 기술 위원회) 내에서 개발했으며, Ecma 총회에서 채택한 후 ISO Fast-Track 프로세스를 통해 ISO/IEC JTC 1에 의해 채택되었습니다.

최신 표준

C# 및 CLI (TR-84)에 사용할 수 있는 공식 Ecma 문서는 다음과 같습니다.

- **C# 언어 표준(버전 7)**: ECMA-334.pdf
- **공용 언어 인프라**: ECMA-335.pdf .
- **파티션 IV XML 파일:**ECMA TR/84 형식에서 파생된 정보입니다.

공식 ISO/IEC 문서는 ISO/IEC 공개적으로 사용 가능한 표준 페이지에서 확인할 수 있습니다 . 이러한 링크는 해당 페이지에서 직접 제공됩니다.

- **정보 기술 - 프로그래밍 언어 - C#**: ISO/IEC 23270:2018
- **정보 기술 - CLI(공용 언어 인프라) 파티션 I에서 VI로**: ISO/IEC 23271:2012
- **정보 기술 — CLI(공용 언어 인프라) - 파티션 IV XML 파일에서 파생된 정보에 대한 기술 보고서**: ISO/IEC TR 23272:2011

.NET 용어

아티클 • 2024. 09. 25.

이 용어집의 주 용도는 .NET 설명서에서 자주 나타나는 선택한 용어 및 머리글자어의 의미를 명확히 나타내는 것입니다.

AOT

Ahead-Of-Time 컴파일러입니다.

JIT와 비슷하게 이 컴파일러도 IL을 기계어 코드로 변환합니다. JIT 컴파일과는 달리 AOT 컴파일은 애플리케이션이 실행되기 전에 수행되며 일반적으로 다른 컴퓨터에서 수행됩니다. AOT 툴 체인은 런타임에 컴파일되지 않으므로 컴파일 시간을 최소화할 필요가 없습니다. 즉, 더 많은 시간을 최적화하는 데 사용할 수 있습니다. AOT의 컨텍스트는 전체 애플리케이션이므로 AOT 컴파일러는 모듈 간 연결 및 전체 프로그램 분석도 수행합니다. 즉, 모든 참조가 수행되고 단일 실행 파일이 생성된다는 의미입니다.

CoreRT와 .NET 네이티브를 참조하세요.

앱 모델

워크로드별 API. 다음 몇 가지 예를 참조하세요.

- .NET Aspire
- ASP.NET
- ASP.NET Web API
- EF(Entity Framework)
- WPF(Windows Presentation Foundation)
- WCF(Windows Communication Foundation)
- Windows Workflow Foundation(WF)
- Windows Forms(WinForms)

ASP.NET

ASP.NET 4.x 및 ASP.NET Framework라고도 하는 .NET Framework와 함께 제공되는 원래 ASP.NET 구현입니다.

경우에 따라 ASP.NET은 원래 ASP.NET 및 ASP.NET Core를 모두 나타내는 포괄적인 용어입니다. 지정된 인스턴스에서 용어가 전달하는 의미는 컨텍스트에 따라 결정됩니다.

ASP.NET을 사용하여 두 구현체를 모두 의미하는 것이 아님을 분명히 하려는 경우 ASP.NET 4.x를 참조하세요.

ASP.NET 설명서를 참조하세요.

ASP.NET Core

다양한 ASP.NET 플랫폼에서 사용할 수 있는 고성능 오픈 소스 구현입니다.

ASP.NET Core 설명서를 참조하세요.

assembly

애플리케이션이나 다른 어셈블리에서 호출할 수 있는 API 컬렉션을 포함할 수 있는 *.dll* 또는 *.exe* 파일입니다.

어셈블리에는 인터페이스, 클래스, 구조체, 열거형 및 대리자와 같은 형식이 포함됩니다. 프로젝트의 *bin* 폴더에 있는 어셈블리를 *바이너리*라고도 합니다. 라이브러리를 참조하세요.

BCL

기본 클래스 라이브러리입니다.

System.* (및 제한된 범위의 Microsoft.*) 네임스페이스를 구성하는 라이브러리 집합입니다. BCL은 ASP.NET Core 같은 상위 수준 애플리케이션 프레임워크의 기반이 되는 하위 수준의 범용 프레임워크입니다.

.NET용 BCL의 소스 코드는 .NET 런타임 리포지토리에 포함되어 있습니다. 대부분의 BCL API는 .NET Framework에서도 사용할 수 있으므로 이 소스 코드를 .NET Framework BCL 소스 코드의 포크로 간주할 수 있습니다.

다음 용어는 종종 BCL에서 참조하는 것과 동일한 API 컬렉션을 참조합니다.

- 핵심 .NET 라이브러리
- 프레임워크 라이브러리
- 런타임 라이브러리
- 공유 프레임워크

CLR

공용 언어 런타임입니다.

정확한 의미는 컨텍스트에 따라 달라집니다. 공용 언어 런타임은 일반적으로 .NET Framework의 런타임 또는 .NET 런타임을 나타냅니다.

CLR은 메모리 할당 및 관리를 처리합니다. 또한 CLR은 앱을 실행할 뿐만 아니라 JIT 컴파일러를 사용하여 즉시 코드를 생성하고 컴파일하는 가상 머신입니다.

.NET Framework의 CLR 구현은 Windows 전용입니다.

.NET(Core CLR이라고도 함)에 대한 CLR 구현은 .NET Framework CLR과 동일한 코드 베이스에서 빌드됩니다. 원래 Core CLR은 Silverlight의 런타임이고 여러 플랫폼, 특히 Windows 및 OS X에서 실행되도록 디자인되었습니다. 지금도 많은 Linux 배포 지원을 포함하는 플랫폼 간 런타임입니다.

런타임도 참조하세요.

Core CLR

.NET용 공용 언어 런타임입니다.

CLR을 참조하세요.

CoreRT

CLR과 달리 CoreRT는 가상 머신이 아닙니다. 즉, JIT를 포함하지 않으므로 즉시 코드를 생성하고 실행하는 기능을 포함하지 않습니다. 그러나 GC와 RTTI(런타임 형식 식별) 및 리플렉션에 대한 기능은 포함합니다. 그러나 해당 형식 시스템은 리플렉션에 대한 메타데이터가 필요하지 않도록 설계되었습니다. 메타데이터가 필요하지 않으면 불필요한 메타데이터를 분리하고 더 중요하게는 앱이 사용하지 않는 코드를 식별할 수 있는 AOT 도구 체인을 사용할 수 있습니다. CoreRT는 개발 중입니다.

Intro to CoreRT (CoreRT 소개) 및 .NET Runtime Lab (.NET 런타임 랩)을 참조하세요.

플랫폼 간

각 운영 체제에 맞게 다시 작성할 필요 없이 Linux, Windows 및 iOS와 같은 다양한 운영 체제에서 사용할 수 있는 애플리케이션을 개발하고 실행하는 기능입니다. 이를 통해 다양한 플랫폼에서 애플리케이션 간에 코드를 다시 사용하고 일관성을 유지할 수 있습니다.

플랫폼을 참조하세요.

에코시스템

특정 기술에 대한 애플리케이션을 빌드하고 실행하는 데 사용되는 모든 런타임 소프트웨어, 개발 도구 및 커뮤니티 리소스입니다.

".NET 에코시스템"이라는 용어는 타사 앱 및 라이브러리를 포함한다는 점에서 ".NET 스택"과 같은 유사한 용어와 다릅니다. 다음은 문장에서의 예제입니다.

- ".NET Standard는 .NET 에코시스템의 통일성을 높이기 위한 것이었습니다."

프레임워크

일반적으로 특정 기술을 기반으로 하는 애플리케이션의 개발 및 배포를 용이하게 하는 포괄적인 API 컬렉션입니다. 이 일반적인 의미에서 ASP.NET Core 및 Windows Forms는 애플리케이션 프레임워크의 예입니다. 프레임워크 및 라이브러리라는 단어는 종종 같은 뜻으로 사용됩니다.

"프레임워크"라는 단어는 다음 용어에서 다른 의미가 있습니다.

- 프레임워크 라이브러리
- .NET Framework
- 공유 프레임워크
- 대상 프레임워크
- TFM(대상 프레임워크 모니커)
- 프레임워크 종속 앱

경우에 따라 "프레임워크"는 .NET 구현을 나타냅니다.

프레임워크 라이브러리

의미는 컨텍스트에 따라 달라집니다. .NET용 프레임워크 라이브러리를 참조할 수 있습니다. 이 경우 BCL이 참조하는 동일한 라이브러리를 참조합니다. BCL을 기반으로 빌드하고 웹앱에 대한 추가 API를 제공하는 ASP.NET Core 프레임워크 라이브러리를 참조할 수도 있습니다.

GC

가비지 수집기입니다.

가비지 수집기는 자동 메모리 관리의 구현체입니다. GC는 개체가 사용한 메모리에서 더 이상 사용되지 않는 메모리를 해제합니다.

가비지 수집을 참조하세요.

IL

중간 언어입니다.

C#과 같은 상위 수준 .NET 언어가 IL(중간 언어)이라는 하드웨어 독립 명령 집합으로 컴파일됩니다. IL은 MSIL(Microsoft IL) 또는 CIL(공통 IL)이라고도 합니다.

JIT

Just-In-Time 컴파일러입니다.

AOT와 유사한 이 컴파일러는 IL을 프로세서에서 이해하는 기계어 코드로 변환합니다. AOT와 달리 JIT 컴파일은 요청 시 수행되며 코드가 실행되어야 하는 것과 동일한 컴퓨터에서 수행됩니다. JIT 컴파일은 애플리케이션을 실행하는 동안 발생하므로 컴파일 시간은 런타임의 일부입니다. 따라서 JIT 컴파일러는 결과 코드가 생성할 수 있는 시간 단축과 코드 최적화에 소요된 시간의 균형을 맞춰야 합니다. 그러나 JIT는 실제 하드웨어를 인식하므로 개발자가 다른 구현을 제공할 필요가 없도록 합니다.

.NET의 구현체

.NET의 구현체에는 다음이 포함됩니다.

- 하나 이상의 런타임. 예: CLR, CoreRT.
- .NET Standard 버전을 구현하고 추가 API를 포함할 수 있는 클래스 라이브러리입니다. 예: .NET Framework 및 .NET용 BCL.
- 필요에 따라 하나 이상의 애플리케이션 프레임워크. 예: ASP.NET, Windows Forms 및 WPF는 .NET Framework 및 .NET에 포함됩니다.
- 필요에 따라 개발 도구. 일부 개발 도구는 여러 구현체에서 공통적으로 사용할 수 있음.

.NET 구현의 예:

- .NET Framework
- .NET
- UWP(유니버설 Windows 플랫폼)
- Mono

자세한 내용은 .NET 구현을 참조 하세요.

라이브러리

앱이나 다른 라이브러리에서 호출할 수 있는 API 컬렉션입니다. .NET 라이브러리는 하나 이상의 어셈블리로 구성됩니다.

라이브러리 및 프레임워크라는 단어는 종종 같은 뜻으로 사용됩니다.

Mono

작은 런타임이 필요할 때 사용되는 오픈 소스 플랫폼 간 .NET 구현입니다. Android, Mac, iOS, tvOS 및 watchOS에서 Xamarin 애플리케이션을 구동하고 주로 작은 공간이 필요한 앱에 초점을 맞춘 런타임입니다.

Mono는 현재 게시된 .NET Standard 버전을 모두 지원합니다.

지금까지 Mono는 .NET Framework의 방대한 API를 구현하고 Unix에서 가장 인기 있는 기능의 일부를 따라서 만들었습니다. Unix에서 이러한 기능을 사용하는 .NET 애플리케이션을 실행하는 데 사용되는 경우도 있습니다.

일반적으로 Mono는 Just-In-Time 컴파일러에서 사용되지만 iOS 같은 플랫폼에 사용되는 전체 정적 컴파일러(Ahead-Of-Time 컴파일) 기능도 제공합니다.

자세한 내용은 Mono 설명서를 참조하세요.

네이티브 AOT

앱이 자체 포함되고 게시 시 네이티브 코드로 미리 컴파일되는 배포 모드입니다. 네이티브 AOT 앱은 런타임에 JIT 컴파일러를 사용하지 않습니다. .NET 런타임이 설치되지 않은 컴퓨터에서 실행할 수 있습니다.

자세한 내용은 네이티브 AOT 배포를 참조하세요.

.NET

.NET 에는 두 가지 의미가 있으며 의도된 의미는 컨텍스트에 따라 달라집니다.

- *.NET*은 *.NET* Standard 및 모든 .NET 구현 및 워크로드에 대한 우산 용어로 사용할 수 있습니다.
- *.NET*은 *.NET* Core라고 불렸던 .NET의 플랫폼 간 고성능 오픈 소스 구현을 더 자주 나타냅니다. .NET 5(및 .NET Core) 이상 버전 또는 *.NET 5 이상* 버전*이라고*도 합니다.

예를 들어 첫 번째 의미는 ".NET의 구현"과 같은 구로 의도된 것입니다. 두 번째 의미는 .NET SDK 및 .NET CLI와 같은 이름으로 의도된 것입니다. 첫 번째 의미를 나타내는 컨텍스트가 없는 경우 두 번째 의미가 의도된 것으로 가정합니다.

이전 버전의 .NET은 .NET Core 1~3.1이라고 합니다. 버전 번호는 4를 건너뛰고 3.1을 따르는 버전을 .NET 5라고 하며 이름에서 "Core"를 삭제합니다. 이 .NET 구현이 모든 새 개발에 권장되는 구현임을 강조하기 위해 "Core"를 삭제했습니다. 버전 4를 건너뛰면 .NET의 최신 구현과 .NET Framework라고 하는 이전 구현의 혼동을 방지하기 위해 수행되었습니다. .NET Framework의 현재 버전은 4.8.1입니다.

.NET은 항상 전체를 대문자로 표기하며 절대로 ".Net"을 사용하지 않습니다.

.NET 설명서를 참조하세요.

.NET CLI

.NET용 애플리케이션 및 라이브러리를 개발하기 위한 플랫폼 간 도구 체인입니다. .NET Core CLI*라고*도 합니다.

.NET CLI를 참조하세요.

.NET Core

.NET을 참조하세요.

.NET Framework

Windows에서만 실행되는 .NET의 구현입니다. CLR(공용 언어 런타임), BCL(기본 클래스 라이브러리) 및 ASP.NET, Windows Forms, WPF 등의 애플리케이션 프레임워크 라이브러리를 포함합니다.

.NET Framework 가이드를 참조하세요.

.NET 네이티브

JIT(Just-In-Time)와 반대로 네이티브 코드 AOT(Ahead-Of-Time)를 생성하는 컴파일러 툴 체인입니다.

컴파일은 C++ 컴파일러 및 링커가 작동하는 방식과 유사하게 개발자의 컴퓨터에서 수행되며, 사용되지 않는 코드를 제거하고 더 많은 시간을 최적화에 사용합니다. 라이브러리에서 코드를 추출하여 실행 파일에 병합합니다. 결과는 전체 앱을 나타내는 단일 모듈입니다.

UWP는 .NET 네이티브에서 지원하는 애플리케이션 프레임워크입니다.

.NET 네이티브 설명서를 참조하세요.

.NET SDK

개발자가 .NET용 애플리케이션 및 라이브러리를 만들 수 있도록 하는 라이브러리 및 도구 집합입니다. .NET Core SDK*라고*도 합니다.

앱을 빌드하기 위한 .NET CLI, 앱을 빌드하고 실행하기 위한 .NET 라이브러리 및 런타임, CLI 명령을 실행하고 애플리케이션을 실행하는 dotnet 실행 파일(*dotnet.exe*)을 포함합니다.

.NET SDK 개요를 참조하세요.

.NET Standard

각 .NET 구현에서 사용할 수 있는 .NET API의 공식 사양입니다.

.NET Standard 사양을 라이브러리라고도 합니다. 라이브러리는 API 구현체를 포함하므로 규격(인터페이스)뿐만 아니라 .NET Standard를 "라이브러리"라고 잘못 부르기도 합니다.

.NET Standard를 참조하세요.

NGen

네이티브(이미지) 생성입니다.

해당 기술을 영구 JIT 컴파일러로 생각할 수 있습니다. 일반적으로 코드가 실행되는 컴퓨터에서 코드를 컴파일하지만 컴파일은 대개 설치 시에 수행됩니다.

패키지

NuGet 패키지 또는 패키지 는 작성자 이름과 같은 추가 메타데이터와 함께 이름이 같은 하나 이상의 어셈블리가 있는 .zip 파일입니다.

.zip 파일에는 *.nupkg 확장이 있으며 여러 대상 프레임워크 및 버전에서 사용하기 위해 .dll 파일 및 .xml 파일과 같은 자산을 포함할 수 있습니다*. 앱 또는 라이브러리에 설치된 경우 앱 또는 라이브러리에서 지정한 대상 프레임워크에 따라 적절한 자산이 선택됩니다. 인터페이스를 정의하는 자산은 *ref* 폴더에 있으며 구현을 정의하는 자산은 *lib* 폴더에 있습니다.

플랫폼

Windows와 macOS, Linux, iOS, Android 같은 운영 체제와 해당 운영 체제가 실행되는 하드웨어입니다.

다음은 문장에서의 사용 예입니다.

- ".NET Core는 다양한 플랫폼에서 사용할 수 있는 .NET의 구현체입니다."
- "PCL 프로필은 Microsoft 플랫폼을 나타내지만 .NET Standard는 플랫폼에 독립적입니다."

레거시 .NET 설명서에서는 .NET 구현 또는 모든 구현을 포함하는 .NET 스택을 의미하는 용어로 ".NET 플랫폼"을 사용하기도 합니다. 해당 용어는 둘 다 기본적인(OS/하드웨어) 의미와 혼동되므로 여기서는 해당 용어를 사용하지 않습니다.

"플랫폼"은 앱 빌드 및 실행을 위한 도구 및 라이브러리를 제공하는 소프트웨어를 나타내는 "개발자 플랫폼"이라는 관용구에서 다른 의미를 가집니다. .NET은 다양한 유형의 애플리케이션을 빌드하기 위한 플랫폼 간 오픈 소스 개발자 플랫폼입니다.

POCO

POCO 또는 일반 이전 클래스/CLR 개체는 공용 속성 또는 필드만 포함하는 .NET 데이터 구조체입니다. POCO에는 다음과 같은 다른 멤버가 포함되어서는 안 됩니다.

- 메서드
- events
- 대리자

이러한 개체는 주로 DDO(데이터 전송 개체)로 사용됩니다. 순수 *POCO* 는 다른 개체를 상속하거나 인터페이스를 구현하지 않습니다. POCO는 직렬화와 함께 사용되는 것이 일반적입니다.

runtime

일반적으로 관리형 프로그램의 실행 환경입니다. OS는 런타임 환경의 일부이지만 .NET 런타임의 일부는 아닙니다. 해당 단어의 뜻 그대로 .NET 런타임의 몇 가지 예는 다음과 같습니다.

- CLR(공용 언어 런타임)
- .NET 네이티브(UWP)
- Mono 런타임

"런타임"이라는 단어는 컨텍스트에 따라 여러 가지 의미가 있습니다.

- .NET 5 다운로드 페이지의 '.NET 런타임'.

'.NET 런타임'이나 다른 런타임(예: 'ASP.NET Core 런타임')을 다운로드할 수 있습니다. 이러한 사용에 대한 '런타임'은 머신에서 프레임워크 종속 앱을 실행하기 위해 머신에 설치해야 하는 구성 요소 세트입니다. .NET 런타임에는 BCL을 제공하는 CLR 및 .NET 공유 프레임워크가 포함됩니다.

- '.NET 런타임 라이브러리'

  BCL이 참조하는 것과 동일한 라이브러리를 참조합니다. 그러나 ASP.NET Core 런타임과 같은 다른 런타임에는 BCL을 기반으로 하는 추가 라이브러리와 함께 다른 공유 프레임워크가 있습니다.

- RID(런타임 식별자).

  여기에서 '런타임'은 .NET 앱이 실행되는 OS 플랫폼 및 CPU 아키텍처를 의미합니다(예: `linux-x64`).

- 경우에 따라 "런타임"은 다음 예제와 같이 .NET 구현의 의미로 사용됩니다.
  - "다양한 .NET 런타임에서 특정 버전의 .NET Standard를 구현합니다. ... 각 .NET 런타임 버전은 지원하는 최신 .NET Standard 버전을 보급합니다."
  - "여러 런타임에서 실행되도록 의도된 라이브러리는 이 프레임워크를 대상으로 해야 합니다." (.NET Standard 참조)

공유 프레임워크

의미는 컨텍스트에 따라 달라집니다. '.NET 공유 프레임워크'는 .NET 런타임에 포함된 라이브러리를 참조합니다. 이 경우 .NET용 *공유 프레임워크*는 BCL이 참조하는 것과 동일한 라이브러리를 참조합니다.

다른 공유 프레임워크도 있습니다. 'ASP.NET Core 공유 프레임워크'는 BCL과 웹앱에서 사용할 추가 API를 포함하는 ASP.NET Core 런타임에 포함된 라이브러리를 참조합니다.

프레임워크 종속 앱의 공유 프레임워크는 앱을 실행하는 머신의 폴더에 설치된 어셈블리에 포함되는 라이브러리로 구성됩니다. 자체 포함 앱의 공유 프레임워크 어셈블리는 앱에 포함됩니다.

자세한 내용은 Deep-dive into .NET Core primitives, part 2: the shared framework (.NET Core 기본 형식 심층 분석, 2부: 공유 프레임워크)를 참조하세요.

stack

애플리케이션을 빌드하고 실행하는 데 함께 사용되는 프로그래밍 기술 집합입니다.

".NET 스택"은 .NET Standard 및 모든 .NET 구현을 나타냅니다. ".NET 스택"이라는 구는 .NET의 한 구현을 참조할 수 있습니다.

대상 프레임워크(target framework)

.NET 앱이나 라이브러리에서 사용하는 API 컬렉션입니다.

앱이나 라이브러리는 모든 .NET 구현에서 표준화된 API 세트의 사양인 .NET Standard의 버전(예: .NET Standard 2.0)을 대상으로 할 수 있습니다. 또한 앱이나는 라이브러리는 특정 .NET 구현체의 버전을 대상으로 할 수 있으며, 이 경우 구현체 관련 API에 액세스할 수 있습니다. 예를 들어 Xamarin.iOS를 대상으로 하는 앱은 Xamarin 제공 iOS API 래퍼에 액세스할 수 있습니다.

일부 대상 프레임워크(예: .NET Framework)의 경우 사용 가능한 API는 애플리케이션 프레임워크 API(예: ASP.NET, WinForms)를 포함할 수 있는 .NET 구현이 시스템에 설치하는 어셈블리에 의해 정의됩니다. 패키지 기반 대상 프레임워크에서 프레임워크 API는 앱이나 라이브러리에 설치된 패키지에 의해 정의됩니다.

대상 프레임워크를 참조하세요.

TFM

대상 프레임워크 모니커입니다.

.NET 앱이나 라이브러리의 대상 프레임워크를 지정하기 위한 표준화된 토큰 형식입니다. 대상 프레임워크는 일반적으로 `net462` 같은 짧은 이름으로 참조합니다. 긴 형식 TFM(예: `.NETFramework,Version=4.6.2`)이 존재하지만 일반적으로 대상 프레임워크를 지정하는 데 사용되지는 않습니다.

대상 프레임워크를 참조하세요.

UWP

범용 Windows 플랫폼

IoT(사물 인터넷)용 터치 가능 Windows 애플리케이션 및 소프트웨어를 빌드하는 데 사용되는 .NET 구현입니다. PC, 태블릿, 휴대폰 및 Xbox를 포함하여 대상으로 지정할 수 있는 다양한 유형의 장치를 통합하도록 설계되었습니다. UWP는 중앙 집중식 앱 스토어, 실행 환경(AppContainer), Win32를 대체할 Windows API(WinRT) 등 많은 서비스를 제공합니다. 앱은 C++과 C#, Visual Basic, JavaScript로 작성할 수 있습니다. C# 및 Visual Basic을 사용하는 경우 .NET API는 .NET에서 제공됩니다.

작업

다른 사람이 빌드 중인 앱의 유형. 앱 모델보다 일반적입니다. 예를 들어 이 문서를 포함한 모든 .NET 설명서 페이지의 맨 위에는 웹, **모바일**, **클라우드**, **클라우드 네이티브 및 데스크톱에 대한 설명서로 전환할 수 있는 워크로드에 대한** 드롭다운 목록이 있습니다.

일부 컨텍스트에서 *워크로드*는 특정 유형의 앱을 지원하기 위해 설치할 수 있는 Visual Studio 기능 컬렉션을 참조합니다. 예제는 Visual Studio 워크로드 구성을 참조 하세요.

참고 항목

- .NET 기본 사항
- .NET Framework 가이드
- ASP.NET 개요
- ASP.NET Core 개요

자습서: .NET 콘솔 애플리케이션 만들기

이 자습서에서는 Visual Studio .NET 콘솔 애플리케이션을 만들고 실행하는 방법을 보여줍니다.

이 자습서에서는 다음을 수행합니다.

- ✓ Visual Studio 솔루션 및 콘솔 앱 프로젝트를 만듭니다.
- ✓ "HelloWorld" .NET 콘솔 애플리케이션을 만듭니다.
- ✓ 사용자에게 이름을 묻는 메시지를 표시하고 콘솔 창에 표시하도록 앱을 개선합니다.

필수 조건

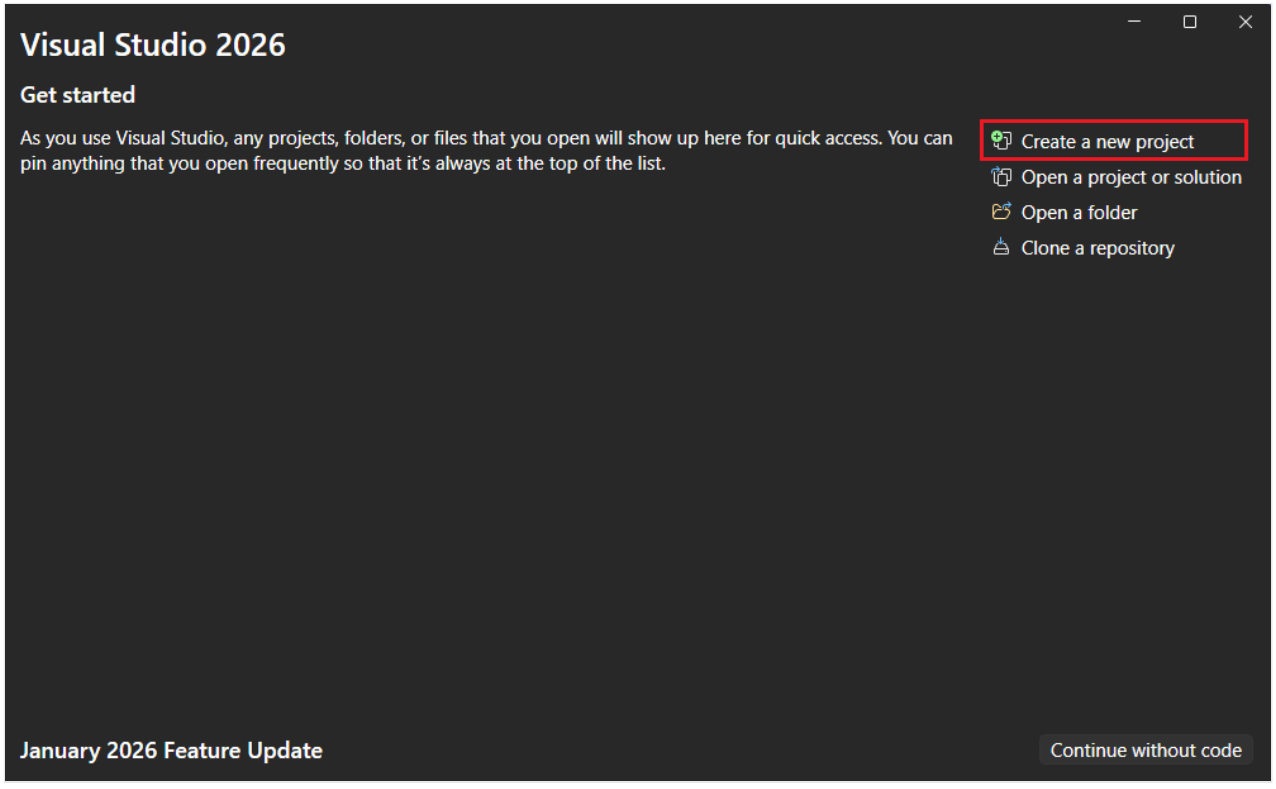
- Visual Studio **.NET 데스크톱 개발** 워크로드가 설치되어 있습니다. 이 워크로드를 선택하면 .NET SDK가 자동으로 설치됩니다.

  자세한 내용은 Visual Studio 참조하세요.

앱 만들기

"HelloWorld"라는 .NET 콘솔 앱 프로젝트를 만듭니다.

1. Visual Studio 시작합니다.
2. 시작 페이지에서 새 프로젝트 만들기를 선택합니다.



3. 새 프로젝트 만들기 페이지에 가서 검색 상자에 콘솔을 입력합니다. 그런 다음 언어 목록에서 **C#** 또는 **Visual Basic**을 선택한 다음 플랫폼 목록에서 **올 플랫폼**을 선택합니다. 콘솔 앱 템플릿을 선택한 다음, 다음을 선택합니다.

   필터가 선택된 새 프로젝트 창 만들기

   > **팁**
   >
   > .NET 템플릿이 표시되지 않으면 필요한 워크로드가 누락되었을 수 있습니다. 당신이 찾고있는 것을 찾을 수 없습니까? 메시지를추가 도구 및 기능 설치 링크를 선택합니다. Visual Studio 설치 관리자가 열립니다. **.NET 데스크톱 개발** 워크로드가 설치되어 있는지 확인합니다.

4. 새 프로젝트 구성 대화 상자의 프로젝트 이름 상자에 HelloWorld 입력합니다. 다음을 선택합니다.

   새 프로젝트 창을 프로젝트 이름, 위치 및 솔루션 이름 필드로 구성하세요.

5. 추가 정보 대화 상자에서 다음을 수행합니다.

   - **.NET 10.0(장기 지원)** 선택합니다.
   - 선택하고생성합니다.

   콘솔 앱에 대한 추가 정보를 입력합니다.

   템플릿은 콘솔 창에 "Hello, World!"를 표시하는 간단한 애플리케이션을 만듭니다. 코드는 Program.cs 또는 Program.vb 파일에 있습니다.

   C#

   ```
   // See https://aka.ms/new-console-template for more information
   Console.WriteLine("Hello, World!");
   ```

   사용하려는 언어가 표시되지 않으면 페이지 맨 위에 있는 언어 선택기를 변경합니다.

   C# 템플릿은 최상위 문을 사용하여 메서드를 호출하여 콘솔 창에 메시지를 표시합니다. Visual Basic 템플릿은 동일한 메서드를 호출하는 `Module Program` 메서드를 사용하여 `Sub Main` 정의합니다.

앱 실행

1. CtrlF5 눌러 디버깅하지 않고 프로그램을 실행합니다.

화면에 "Hello, World!" 텍스트가 인쇄된 콘솔 창이 열립니다. (또는 Visual Basic 프로젝트 템플릿에 쉼표가 없는 "헬로 월드!")

2. 아무 키나 눌러 콘솔 창을 닫습니다.

앱 향상

사용자에게 이름을 묻는 메시지를 표시하고 날짜 및 시간과 함께 표시하도록 애플리케이션을 향상시킵니다.

1. Program.cs 또는 Program.vb 내용을 다음 코드로 바꿉니다.

   C#

   ```
   Console.WriteLine("What is your name?");
   var name = Console.ReadLine();
   var currentDate = DateTime.Now;
   Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at
   {currentDate:t}!");
   Console.Write($"{Environment.NewLine}Press Enter to exit...");
   Console.Read();
   ```

   이 코드는 콘솔 창에 프롬프트를 표시하고 사용자가 문자열을 입력한 다음 Enter 키를 입력할 때까지 기다립니다. 이 문자열은 변수에 저장됩니다. 또한 현재 현지 시간을 포함하는 속성의 값을 검색하고 변수에 할당합니다. 콘솔 창에 이러한 값이 표시됩니다. 마지막으로 콘솔 창에 프롬프트를 표시하고 메서드를 호출하여 사용자 입력을 기다립니다.

   는 줄 바꿈을 나타내는 플랫폼 독립적이며 언어 독립적인 방법입니다. 대안은 C#의 `\n`, Visual Basic `vbCrLf`.

   문자열 앞에 달러 기호()를 사용하면 변수 이름과 같은 식을 문자열의 중괄호 안에 넣을 수 있습니다. 식 값은 식 대신 문자열에 삽입됩니다. 이 구문을 보간된 문자열이라고 합니다.

2. CtrlF5 눌러 디버깅하지 않고 프로그램을 실행합니다.

3. 이름을 입력하고 Enter 키를 눌러 프롬프트에 응답합니다.

   콘솔 창 (수정된 프로그램 출력)

4. 아무 키나 눌러 콘솔 창을 닫습니다.

추가 리소스

- STS(표준 지원 기간) 릴리스 및 LTS(장기 지원) 릴리스.

다음 단계:

이 자습서에서는 .NET 콘솔 애플리케이션을 만들었습니다. 다음 자습서에서는 앱을 디버그합니다.

.NET 콘솔 애플리케이션을 디버그합니다

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 06.

자습서: .NET 콘솔 애플리케이션 디버그

이 자습서에서는 Visual Studio 사용할 수 있는 디버깅 도구를 소개합니다.

> ⓘ **Important**
>
> 모든 바로 가기 키는 Visual Studio 기본값을 기반으로 합니다. 바로 가기 키는 다를 수 있습니다. 자세한 내용은 Visual Studio 참조하세요.

필수 조건

이 자습서는 .NET 콘솔 애플리케이션 만들기에서 만든 콘솔 앱에서 작동합니다.

디버그 빌드 구성 사용

Debug 및 *Release* Visual Studio 기본 제공 빌드 구성입니다. 디버깅에 디버그 빌드 구성을 사용하고 최종 릴리스 배포에 릴리스 구성을 사용합니다.

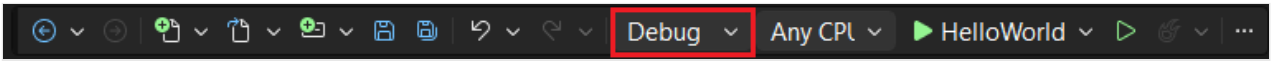
디버그 구성에서 프로그램은 전체 심볼릭 디버그 정보와 최적화 없이 컴파일됩니다. 소스 코드와 생성된 명령 간의 관계가 더 복잡하기 때문에 최적화로 인해 디버깅이 복잡해집니다. 프로그램의 릴리스 구성에는 기호화된 디버그 정보가 없으며 완전히 최적화되어 있습니다.

기본적으로 Visual Studio 디버그 빌드 구성을 사용하므로 디버깅하기 전에 변경할 필요가 없습니다.

1. Visual Studio 시작합니다.
2. .NET 콘솔 애플리케이션 만들기에서 만든 프로젝트를 엽니다.

   현재 빌드 구성이 도구 모음에 표시됩니다. 다음 도구 모음 이미지는 Visual Studio 앱의 디버그 버전을 컴파일하도록 구성되어 있음을 보여줍니다.

   디버그가 강조 표시된



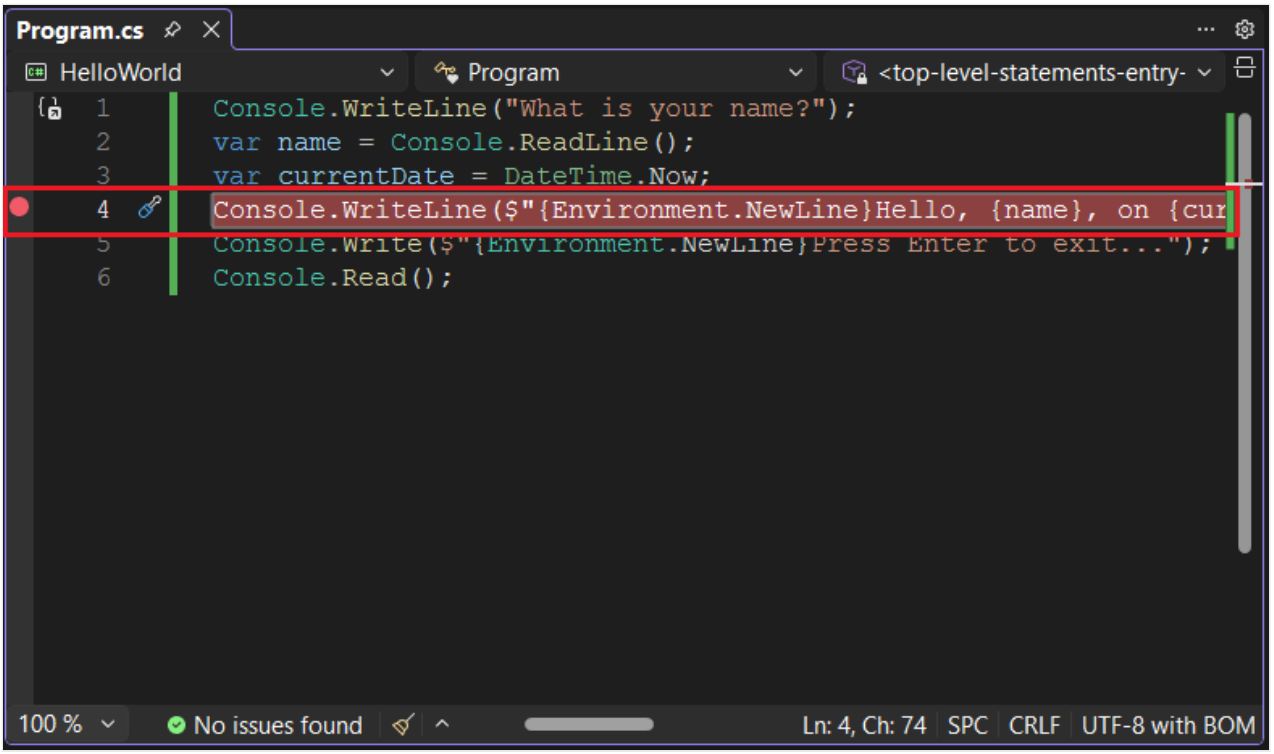
중단점 설정

중단점은 중단점이 있는 줄이 실행되기 전에 애플리케이션의 실행을 일시적으로 중단합니다.

1. 코드 창의 해당 줄 왼쪽 여백을 클릭하여 이름, 날짜, 시간 표시 줄에 *중단점*을 설정합니다. 왼쪽 여백은 줄 번호의 왼쪽에 있습니다. 중단점을 설정하는 다른 방법은 코드 줄에 커서를

놓은 다음 F9 누르거나 메뉴 모음에서 중단점 토글디버그를 선택하는 것입니다.

다음 이미지에서 볼 수 있듯이 Visual Studio 중단점을 강조 표시하고 왼쪽 여백에 빨간색 점을 표시하여 중단점이 설정된 선을 나타냅니다.



2. `F5` 눌러 디버그 모드에서 프로그램을 실행합니다. 디버깅을 시작하는 또 다른 방법은 메뉴에서 **디버그>디버깅 시작** 선택하는 것입니다.

3. 프로그램에서 이름을 묻는 메시지가 표시되면 콘솔 창에 문자열을 입력하고 `Enter 키를` 누릅니다.

4. 프로그램 실행은 중단점에 도달하고 `Console.WriteLine` 메서드가 실행되기 전에 중지됩니다. **Locals** 창에는 현재 실행 중인 메서드에 정의된 변수 값이 표시됩니다.

   Visual Studio의 중단점 스크린샷

디버깅 시작

이전 섹션에서 설명한 대로 중단점에 도달하면 프로그램 실행이 중지됩니다. **Locals** 창에는 현재 실행 중인 메서드에 정의된 변수 값이 표시됩니다.

즉시 창 사용

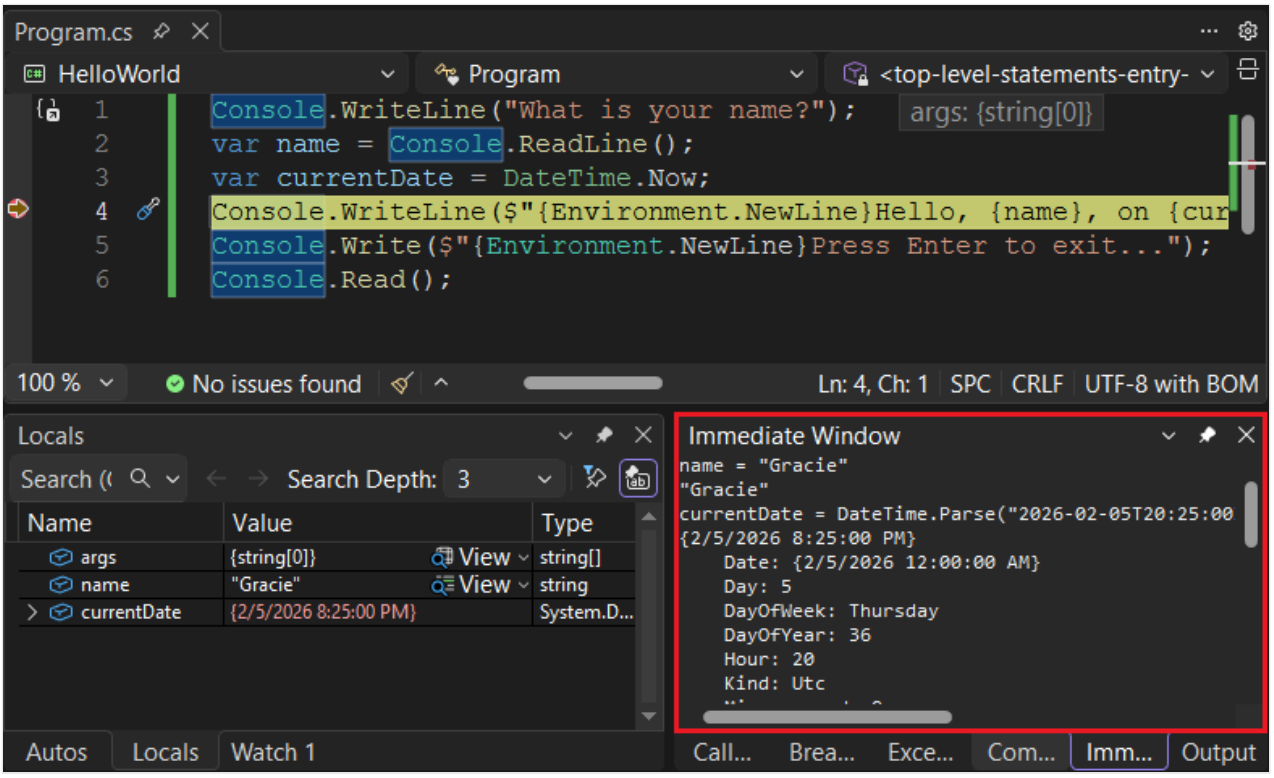
즉시 창을 사용하면 디버깅 중인 애플리케이션과 상호 작용할 수 있습니다. 변수 값을 대화형으로 변경하여 프로그램에 미치는 영향을 확인할 수 있습니다.

1. **임미디어** 창이 표시되지 않으면 **Debug>Windows>Immediate**를 선택하여 표시합니다.

2. `name = "Gracie"`을/를 **Immediate** 창에 입력하고 `Enter` 키를 누릅니다.

3. `currentDate = DateTime.Parse("2026-02-05T20:25:00Z").ToUniversalTime()`을/를 **Immediate** 창에 입력하고 `Enter` 키를 누릅니다.

   즉시 창에는 문자열 변수의 값과 DateTime 값의 속성이 표시됩니다. 또한 변수 값은 **지역** 창에서 업데이트됩니다.



4. `F5` 눌러 프로그램 실행을 계속합니다. 계속하는 또 다른 방법은 메뉴에서 **디버그**>**계속** 선택하는 것입니다.

   콘솔 창에 표시되는 값은 **즉시** 창에서 변경한 내용에 해당합니다.

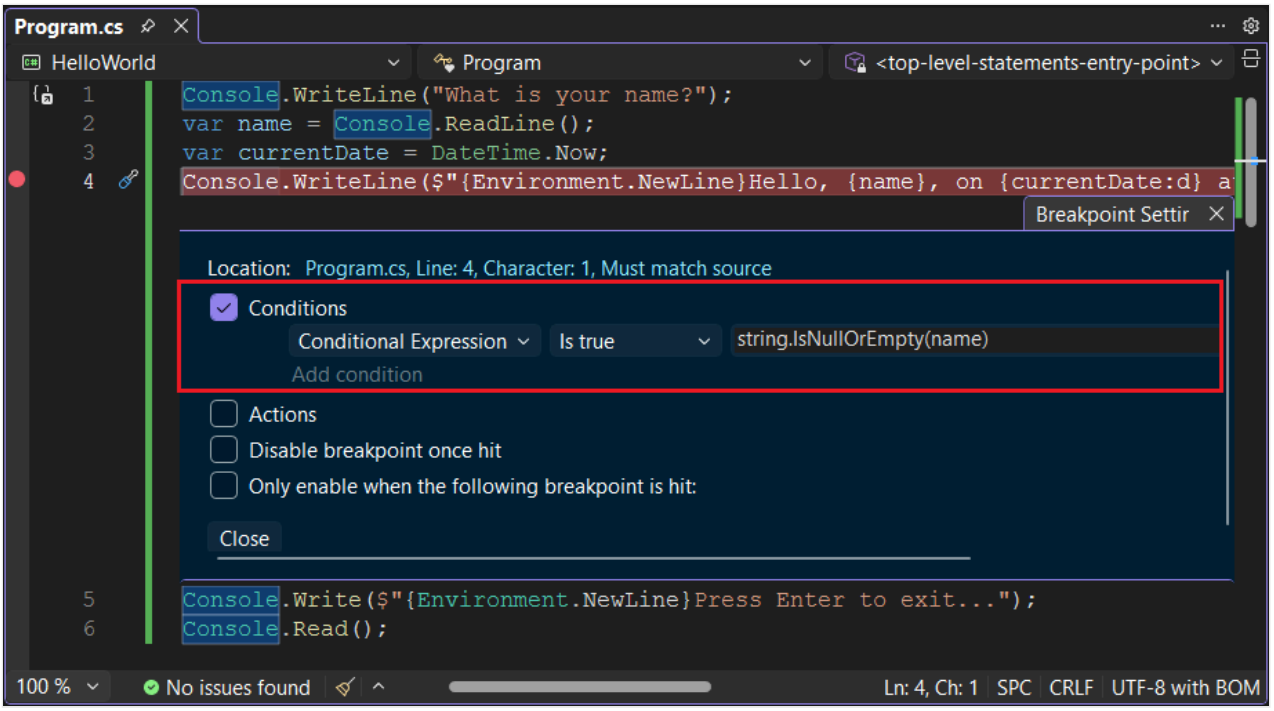
5. 아무 키나 눌러 애플리케이션을 종료하고 디버깅을 중지합니다.

조건부 중단점 설정

프로그램에는 사용자가 입력하는 문자열이 표시됩니다. 사용자가 아무것도 입력하지 않으면 어떻게 되나요? 유용한 디버깅 기능인 *조건부 중단점을 사용하여 이*을 테스트할 수 있습니다.

1. 중단점을 나타내는 빨간색 점을 마우스 오른쪽 단추로 클릭합니다. 상황에 맞는 메뉴에서 **조건** 선택하여 **중단점 설정** 대화 상자를 엽니다. 아직 선택되지 않은 경우 **조건** 상자를 선택합니다.

중단점 설정 패널을 보여 주는



2. **조건식**의 경우 텍스트 필드에 다음 코드를 입력합니다.

```C#
string.IsNullOrEmpty(name)
```

중단점이 적중될 때마다 디버거는 `String.IsNullOrWhiteSpace(name)` 메서드를 호출하고 메서드 호출이 `true`반환하는 경우에만 이 줄에서 중단됩니다.

조건식 대신 *적중 횟수*지정할 수 있습니다. 그러면 문이 지정된 횟수만큼 실행되기 전에 프로그램 실행을 중단합니다. 또 다른 옵션은 스레드 식별자, 프로세스 이름 또는 스레드 이름과 같은 특성에 따라 프로그램 실행을 중단하는 *필터 조건*지정하는 것입니다.

3. **닫기**를 선택하여 대화 상자를 닫습니다.

4. `F5`눌러 디버깅으로 프로그램을 시작합니다.

5. 콘솔 창에서 이름을 입력하라는 메시지가 표시되면 `Enter` 키를 누릅니다.

6. 지정한 조건(`name` `null` 또는 String.Empty)이 충족되었으므로 중단점에 도달하고 `Console.WriteLine` 메서드가 실행되기 전에 프로그램 실행이 중지됩니다.

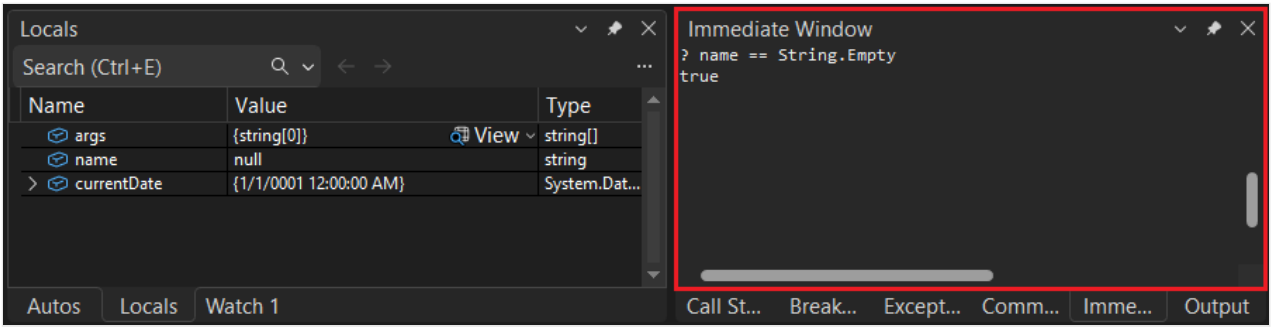
7. 현재 실행 중인 메서드에 로컬인 변수의 값을 보여 주는 **Locals** 창을 선택합니다. 이 경우 `Main` 현재 실행 중인 메서드입니다. 주목할 점은 `name` 변수의 값이 `""`또는 String.Empty이라는 것입니다.

8. **직접 실행** 창에 다음 문을 입력하고 `Enter`눌러 값이 빈 문자열임을 확인하십시오. 결과는 `true`입니다.

```
C#

? name == String.Empty
```

직접 실행 창에서 물음표는 식을 평가하도록 지시합니다 .

명령문이 실행된 후 true 값을 반환하는



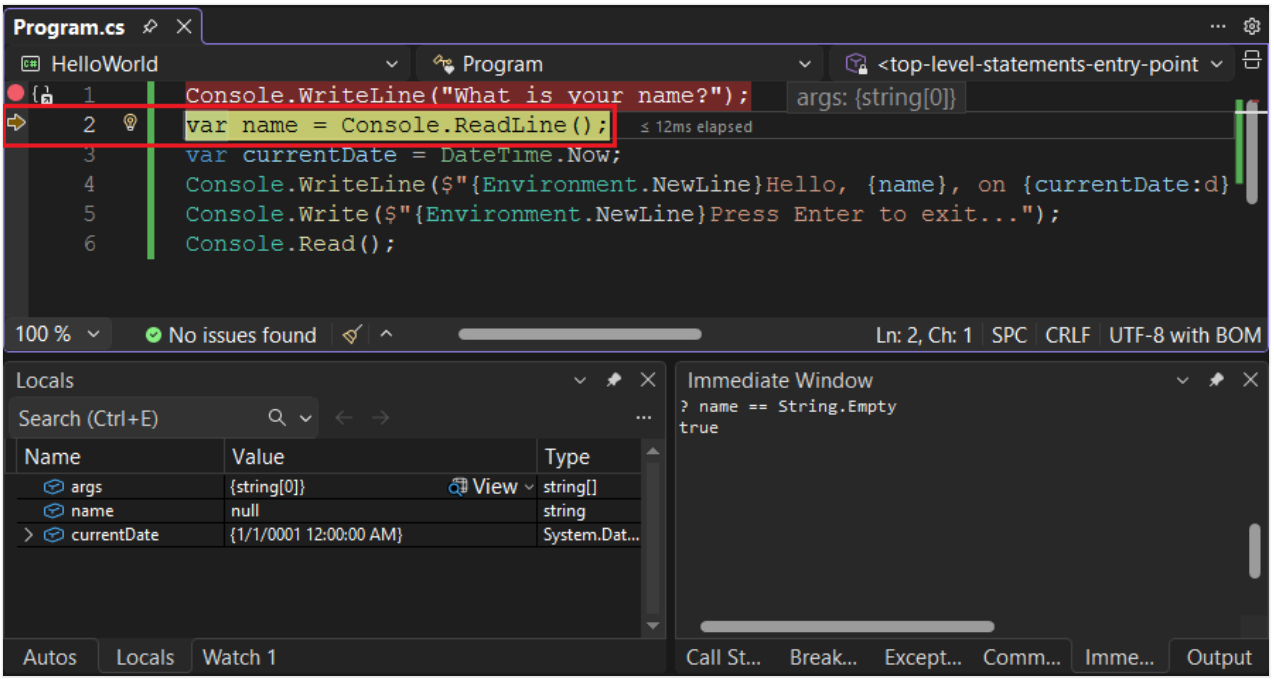
9. `F5` 눌러 프로그램 실행을 계속합니다.
10. 아무 키나 눌러 콘솔 창을 닫고 디버깅을 중지합니다.
11. 코드 창의 왼쪽 여백에 있는 점을 클릭하여 중단점을 지웁니다. 다른 방법으로 중단점을 제거하려면 코드 줄을 선택한 상태에서 `F9` 버튼을 누르거나 **디버그** > **중단점 설정/해제**를 선택하세요.

프로그램을 단계별로 실행

또한 Visual Studio 프로그램을 한 줄씩 단계별로 실행하고 실행을 모니터링할 수 있습니다. 일반적으로 중단점을 설정하고 프로그램 코드의 작은 부분을 통해 프로그램 흐름을 따릅니다. 이 프로그램은 작으므로 전체 프로그램을 단계별로 실행할 수 있습니다.

1. "이름이란?" 프롬프트를 표시하는 코드 줄에 중단점을 설정합니다.
2. **디버그**>**한 단계씩**선택합니다. 문을 하나씩 디버그하는 또 다른 방법은 `F11`누르는 것입니다.

   Visual Studio는 다음 실행 줄 옆에 있는 화살표를 강조 표시하고 나타냅니다.



이때 **Locals** 창에는 `args` 배열이 비어 있으며 `name` 및 `currentDate` 기본값이 있음을 보여 줍니다. 또한 Visual Studio 빈 콘솔 창을 열었습니다.

3. `F11 키를` 누릅니다. Visual Studio `name` 변수 할당을 포함하는 문을 강조 표시합니다. **Locals** 창에는 `name` `null`표시되고, 콘솔 창에는 "당신의 이름은 무엇입니까?"라는 문자열이 표시됩니다.

4. 콘솔 창에 문자열을 입력하고 `Enter`눌러 프롬프트에 응답합니다. 콘솔이 응답하지 않고 입력한 문자열이 콘솔 창에 표시되지 않지만 Console.ReadLine 메서드는 입력을 캡처합니다.

5. `F11 키를` 누릅니다. Visual Studio `currentDate` 변수 할당을 포함하는 문을 강조 표시합니다. **Locals** 창에는 Console.ReadLine 메서드 호출에서 반환된 값이 표시됩니다. 콘솔 창에는 프롬프트에 입력한 문자열도 표시됩니다.

6. `F11 키를` 누릅니다. **Locals** 창에는 `currentDate` 속성에서 할당된 후 DateTime.Now 변수의 값이 표시됩니다. 콘솔 창은 변경되지 않습니다.

7. `F11 키를` 누릅니다. Visual Studio Console.WriteLine(String, Object, Object) 메서드를 호출합니다. 콘솔 창에 서식이 지정된 문자열이 표시됩니다.

8. **디버그>을(를) 선택합니다**. 단계별 실행을 중지하는 또 다른 방법은 `Shift` + `F11`을(를) 누르는 것입니다.

   콘솔 창에 메시지가 표시되고 키를 누를 때까지 기다립니다.

9. 아무 키나 눌러 콘솔 창을 닫고 디버깅을 중지합니다.

릴리스 빌드 구성 사용

애플리케이션의 디버그 버전을 테스트한 후에는 릴리스 버전도 컴파일하고 테스트해야 합니다. 릴리스 버전은 애플리케이션의 동작에 부정적인 영향을 줄 수 있는 컴파일러 최적화를 통합합니다. 예를 들어 성능을 향상하도록 설계된 컴파일러 최적화는 다중 스레드 애플리케이션에서 경합 조건을 만들 수 있습니다.

콘솔 애플리케이션의 릴리스 버전을 빌드 및 테스트하려면 도구 모음의 빌드 구성을 **디버그**에서 **릴리스**로 변경합니다.

F5 키를 누르거나 Build 메뉴에서 Build Solution 선택하면 Visual Studio 애플리케이션의 릴리스 버전을 컴파일합니다. 디버그 버전처럼 테스트할 수 있습니다.

다음 단계:

이 자습서에서는 디버깅 도구를 사용했습니다. 다음 자습서에서는 배포 가능한 버전의 앱을 게시합니다.

.NET 콘솔 애플리케이션 게시

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

자습서: .NET 콘솔 애플리케이션 게시

이 자습서에서는 다른 사용자가 실행할 수 있도록 콘솔 앱을 게시하는 방법을 보여줍니다. 게시하면 애플리케이션을 실행하는 데 필요한 파일 집합이 만들어집니다. 파일을 배포하려면 대상 컴퓨터에 복사합니다.

필수 조건

- 이 자습서는 .NET 콘솔 애플리케이션 만들기에서 만든 콘솔 앱에서 작동합니다.

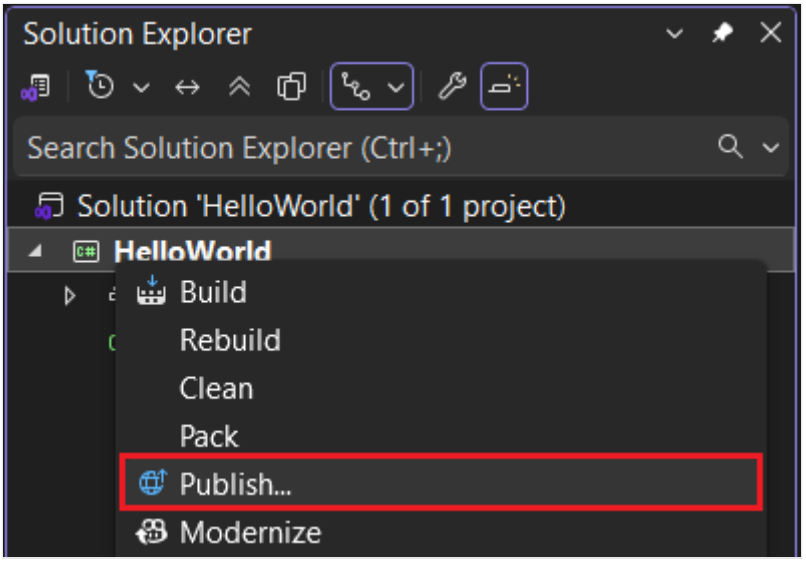
앱 게시

1. Visual Studio 시작합니다.
2. *HelloWorld* 프로젝트를 엽니다 .NET 콘솔 애플리케이션 만들기.
3. Visual Studio 릴리스 빌드 구성을 사용하고 있는지 확인합니다. 필요한 경우 도구 모음의 빌드 구성 설정을 **디버그릴리스**변경합니다.

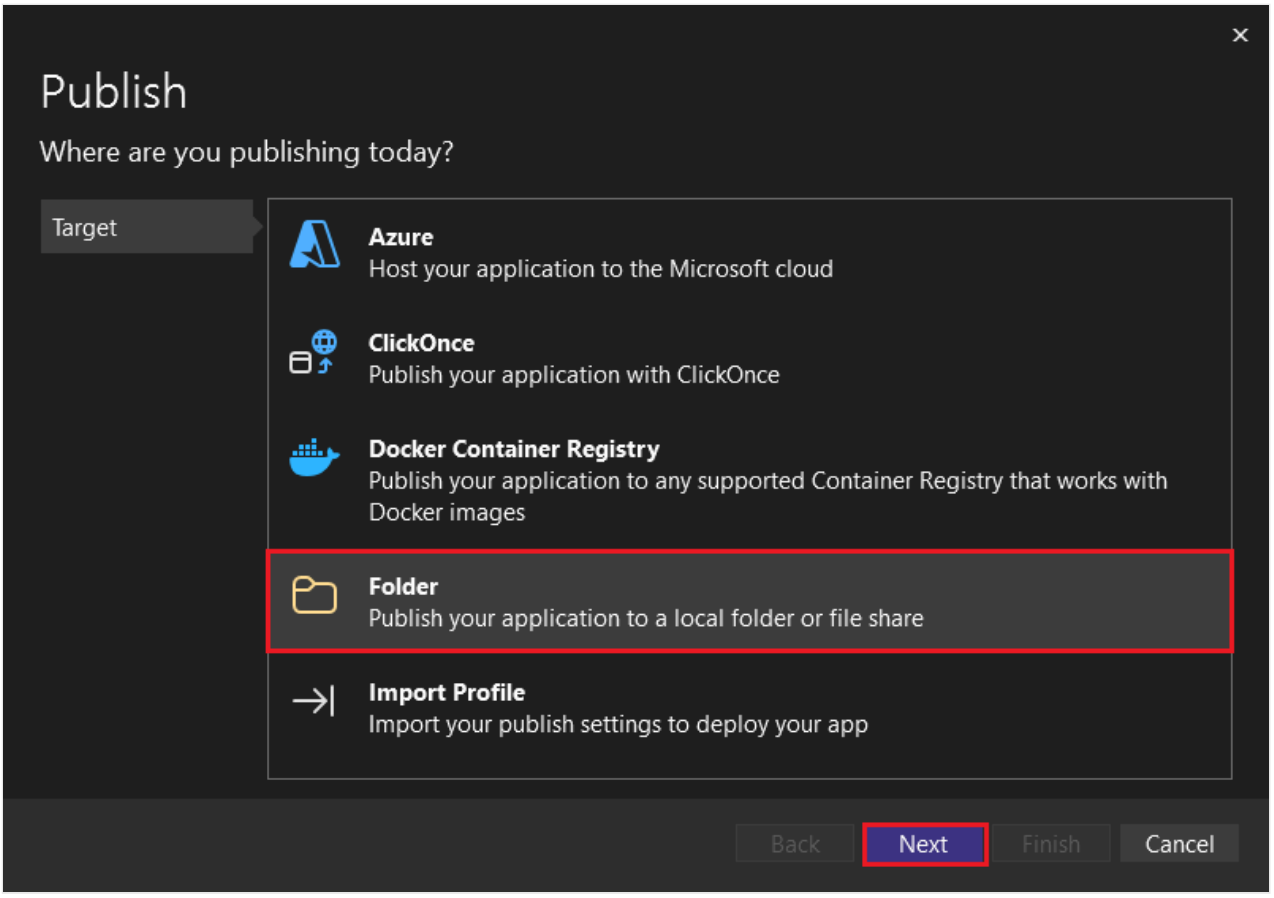
   릴리스 빌드가 선택된

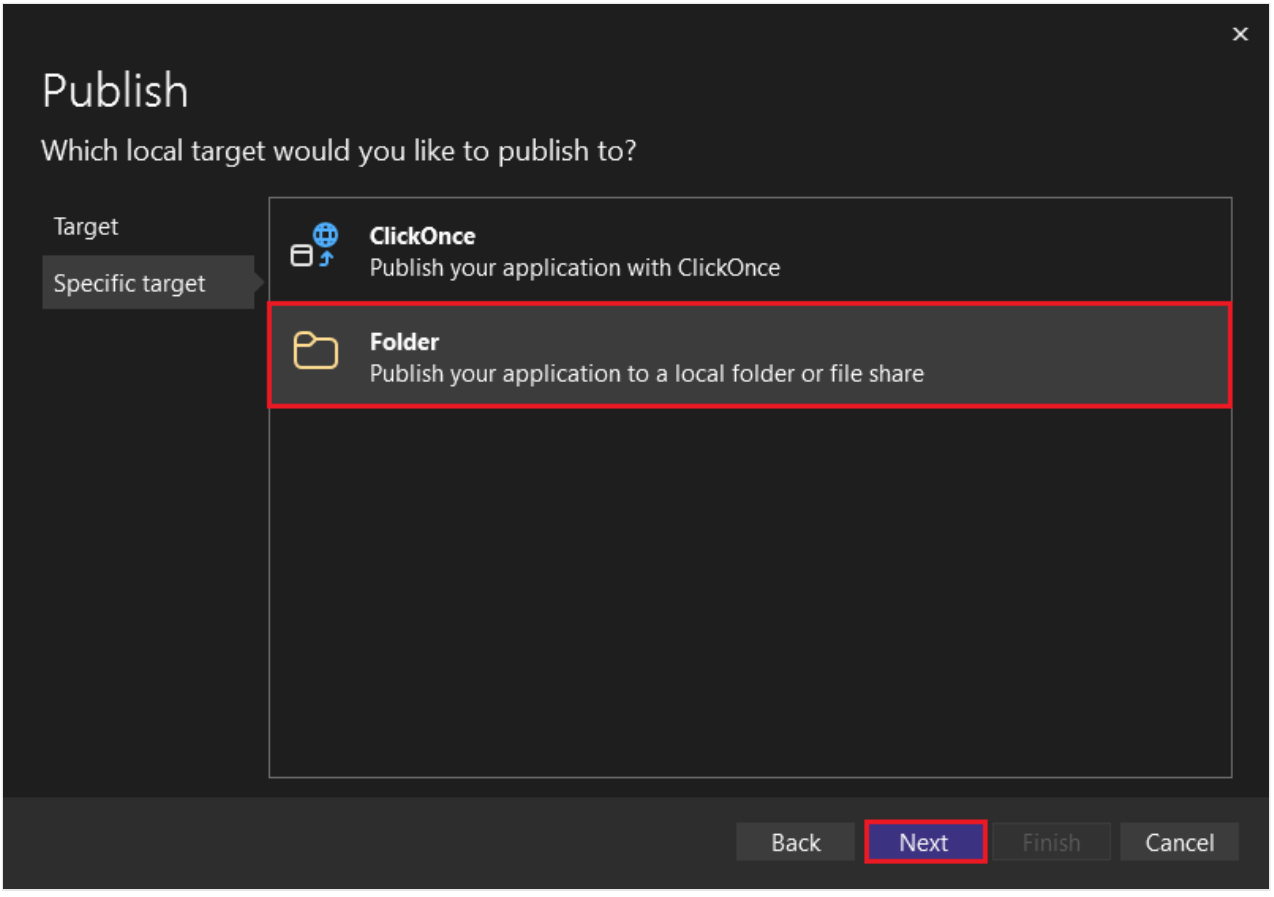
4. HelloWorld 솔루션이 아닌 **HelloWorld** 프로젝트를 마우스 오른쪽 단추로 클릭하고 메뉴에서 **게시**를 선택합니다.

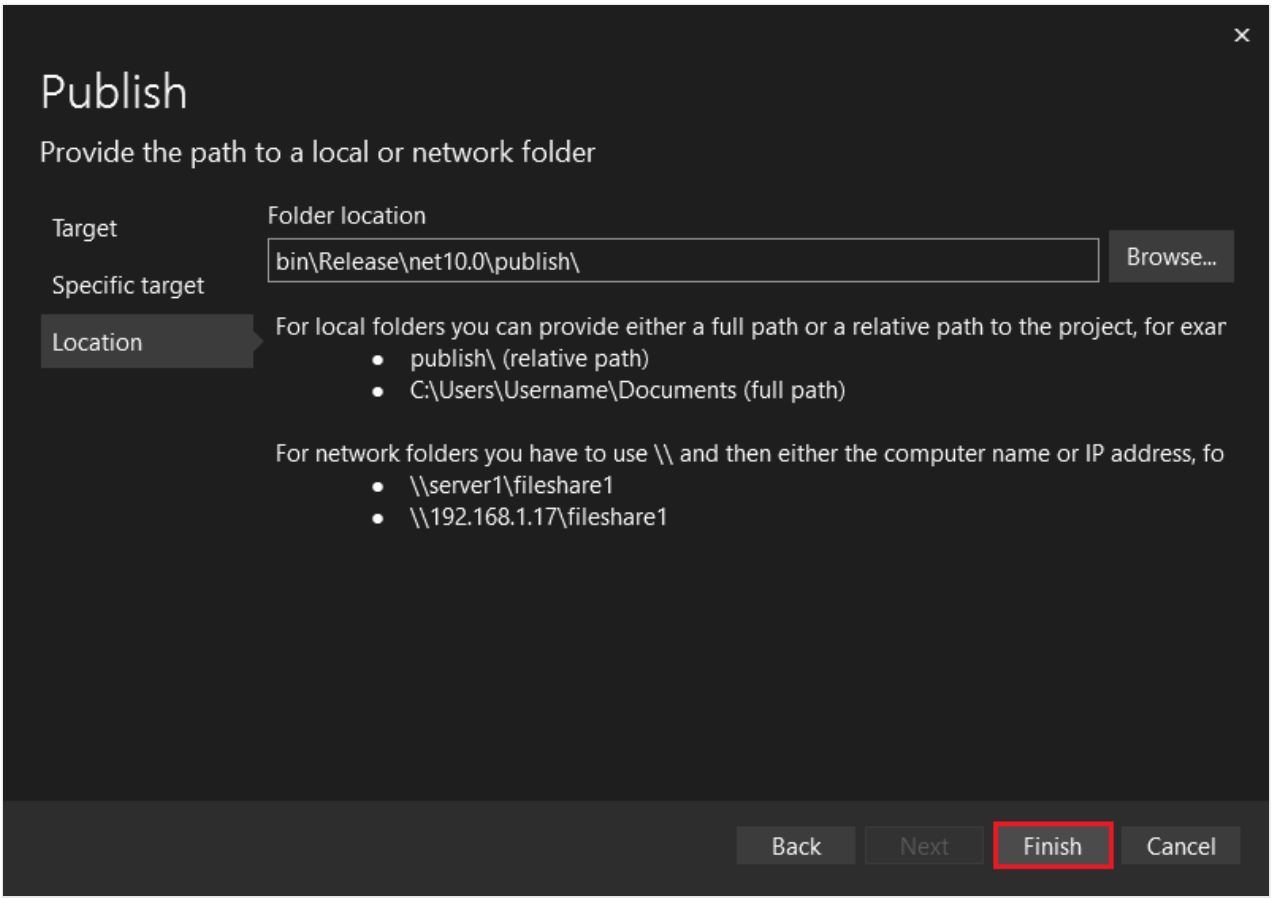
5. **대상** 탭의 **게시** 페이지에서 **폴더**를 선택한 다음, **다음**을 선택합니다.



6. **게시** 페이지의 **특정 대상** 탭에서 **폴더**를 선택한 후, **다음**을 선택합니다.

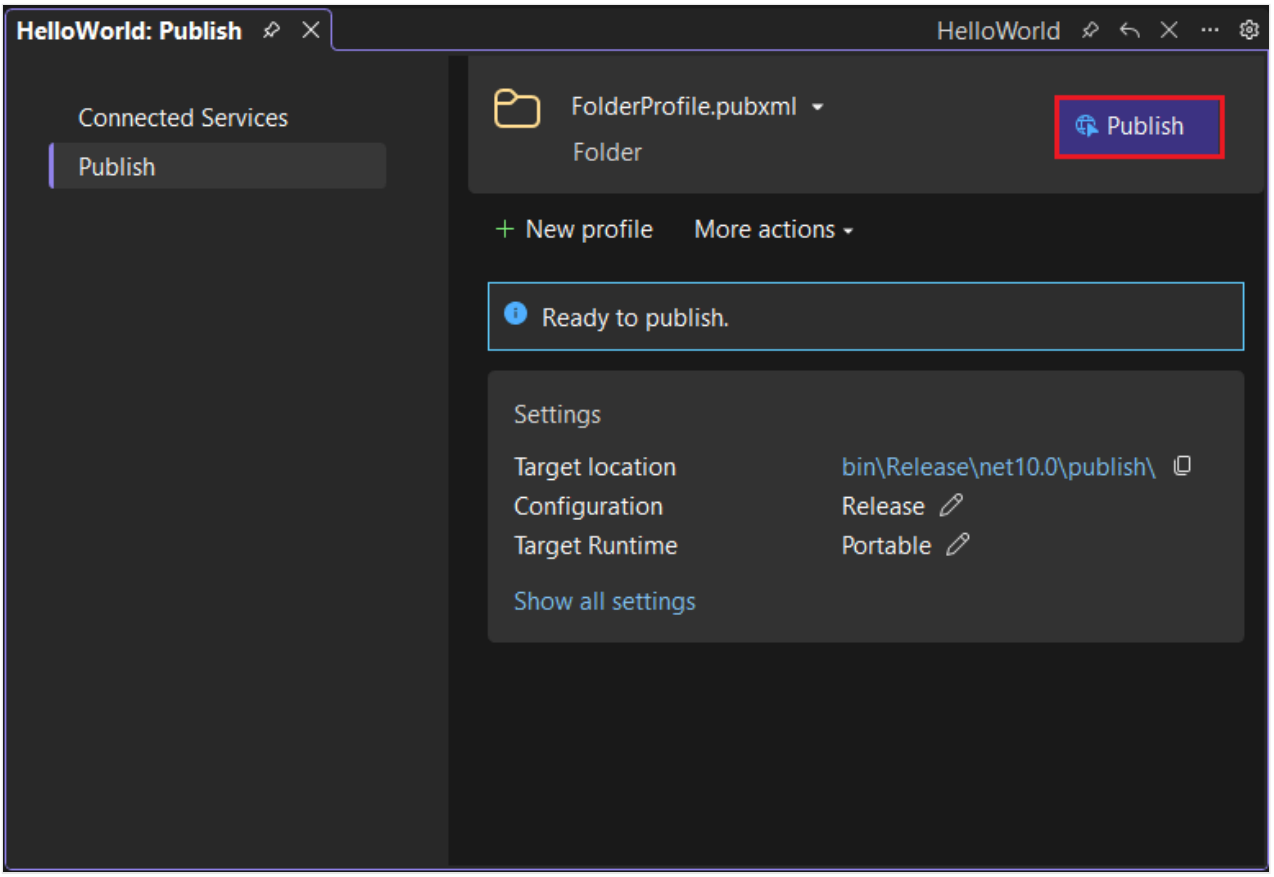


7. **게시** 페이지의 **위치** 탭에서 **마침**을 선택합니다.



8. **게시 프로필 만들기 진행률** 페이지에서 **닫기**선택합니다.

9. 게시 창의 게시 탭에서 게시선택합니다.

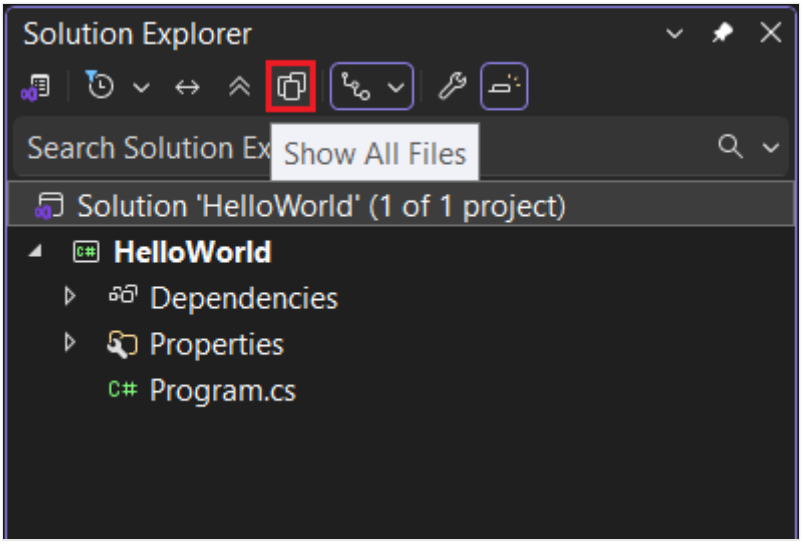


파일 검사

기본적으로 게시 프로세스는 게시된 애플리케이션이 .NET 런타임이 설치된 컴퓨터에서 실행되는 배포 유형인 프레임워크 종속 배포를 만듭니다. 사용자는 실행 파일을 두 번 클릭하거나 명령 프롬프트에서 `dotnet HelloWorld.dll` 명령을 실행하여 게시된 앱을 실행할 수 있습니다.

다음 단계에서는 게시 프로세스에서 만든 파일을 살펴보겠습니다.

1. **Solution Explorer**에서 **전체 파일 표시**를 선택합니다.



2. 프로젝트 폴더에서 *bin/Release/{net}/publish*를 확장합니다. (여기서 {net}은 *net10.0*과 같은 대상 프레임워크 폴더입니다.)

   솔루션 탐색기에 게시된 파일을 표시합니다.

   이미지에서 볼 수 있듯이 게시된 출력에는 다음 파일이 포함됩니다.

   - *HelloWorld.deps.json*

     애플리케이션의 런타임 종속성 파일입니다. 앱을 실행하는 데 필요한 .NET 구성 요소 및 라이브러리(애플리케이션을 포함하는 동적 링크 라이브러리 포함)를 정의합니다. 자세한 내용은 런타임 구성 파일 참조하세요.

   - *HelloWorld.dll*

     애플리케이션의 프레임워크 종속 배포 버전입니다. 이 동적 링크 라이브러리를 실행하려면 명령 프롬프트에 `dotnet HelloWorld.dll` 입력합니다. 이 앱 실행 방법은 .NET 런타임이 설치된 모든 플랫폼에서 작동합니다.

   - *HelloWorld.exe*

     애플리케이션의 프레임워크 종속 실행 파일 버전입니다. 실행하려면 명령 프롬프트에 `HelloWorld.exe` 입력합니다. 파일은 운영 체제에 따라 다릅니다.

- *HelloWorld.pdb*(배포의 경우 선택 사항)

  디버그 기호 파일입니다. 애플리케이션과 함께 이 파일을 배포할 필요는 없지만 게시된 버전의 애플리케이션을 디버그해야 하는 경우에 저장해야 합니다.

- *HelloWorld.runtimeconfig.json*

  애플리케이션의 런타임 구성 파일입니다. 애플리케이션이 실행되도록 빌드된 .NET 버전을 식별합니다. 구성 옵션을 추가할 수도 있습니다. 자세한 내용은 .NET 런타임 구성 설정 참조하세요.

게시된 앱 실행

1. **Solution Explorer***pblish* 폴더를 마우스 오른쪽 단추로 클릭하고 **Copy 전체 경로** 선택합니다.

2. 명령 프롬프트를 열고 *게시* 폴더로 이동합니다. 이렇게 하려면 `cd` 입력한 다음 전체 경로를 붙여넣습니다. 다음은 그 예입니다.

   콘솔

   ```
   cd C:\Projects\HelloWorld\bin\Release\net10.0\publish\
   ```

3. 실행 파일을 사용하여 앱을 실행합니다.

   a. `HelloWorld.exe` 입력하고 `를 누른 후`를 입력하세요.

   b. 프롬프트에 대한 응답으로 이름을 입력하고 아무 키나 눌러 종료합니다.

4. `dotnet` 명령을 사용하여 앱을 실행합니다.

   a. `dotnet HelloWorld.dll` 입력하고 `를 누른 후`를 입력하세요.

   b. 프롬프트에 대한 응답으로 이름을 입력하고 아무 키나 눌러 종료합니다.

추가 리소스

- .NET 애플리케이션 게시 개요
- dotnet publish
- CI(연속 통합) 환경에서 .NET SDK 사용

다음 단계:

이 자습서에서는 콘솔 앱을 게시했습니다. 다음 자습서에서는 클래스 라이브러리를 만듭니다.

.NET 클래스 라이브러리 만들기

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

자습서: .NET 클래스 라이브러리 만들기

이 자습서에서는 단일 문자열 처리 메서드를 포함하는 간단한 유틸리티 라이브러리를 만듭니다.

클래스 라이브러리 애플리케이션에서 호출되는 형식과 메서드를 정의합니다. 라이브러리가 .NET Standard 2.0을 대상으로 하는 경우 .NET Standard 2.0을 지원하는 .NET 구현(.NET Framework 포함)에서 호출할 수 있습니다. 라이브러리가 .NET 10을 대상으로 하는 경우 .NET 10을 대상으로 하는 애플리케이션에서 호출할 수 있습니다. 이 자습서에서는 .NET 10을 대상으로 하는 방법을 보여줍니다.

클래스 라이브러리를 만들 때 타사 구성 요소 또는 하나 이상의 애플리케이션이 있는 번들 구성 요소로 배포할 수 있습니다.

필수 조건

- Visual Studio **.NET 데스크톱 개발** 워크로드가 설치되어 있습니다. 이 워크로드를 선택하면 .NET SDK가 자동으로 설치됩니다.

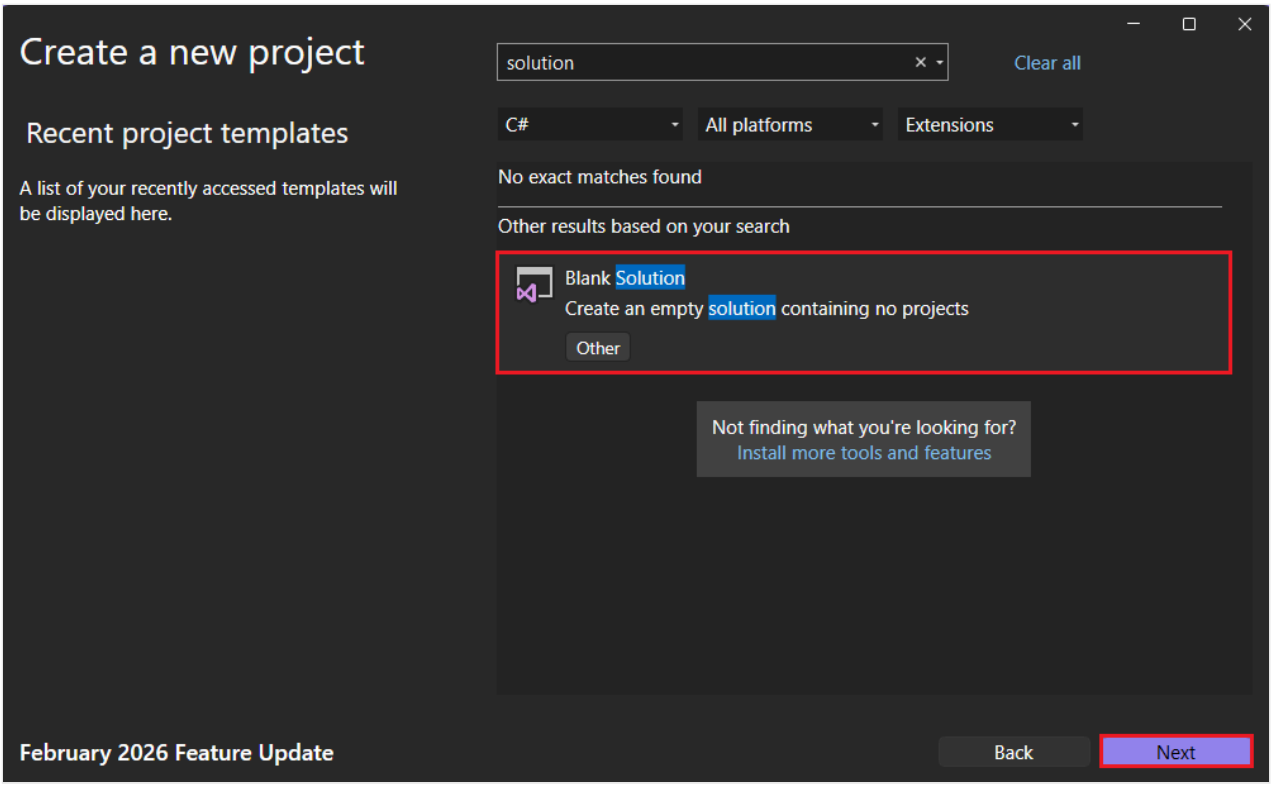
  자세한 내용은 Visual Studio 참조하세요.

솔루션 만들기

먼저 클래스 라이브러리 프로젝트를 저장할 빈 솔루션을 만듭니다. Visual Studio 솔루션은 하나 이상의 프로젝트에 대한 컨테이너 역할을 합니다. 동일한 솔루션에 관련 프로젝트를 추가합니다.

빈 솔루션을 만들려면 다음을 수행합니다.

1. Visual Studio 시작합니다.
2. 시작 창에서 **새 프로젝트 만들기**을 선택합니다.
3. **새 프로젝트** 만들기 페이지에서 검색 상자에 **솔루션** 입력합니다. **비어 있는 솔루션** 템플릿을 선택한 다음 **다음**으로 진행합니다.



4. **새 프로젝트** 구성 페이지에서 **솔루션 이름** 상자에 **ClassLibraryProjects**을 입력합니다. 그런 다음 **만들기**선택합니다.

클래스 라이브러리 프로젝트 만들기

1. 솔루션에 **StringLibrary**라는 새 .NET 클래스 라이브러리 프로젝트를 추가합니다.

   a. **Solution Explorer** 솔루션을 마우스 오른쪽 단추로 클릭하고 **Add**>**새로운 프로젝트** 선택합니다.

   b. **새 프로젝트** 추가 페이지에서 검색 상자에 **라이브러리**를 입력합니다. 언어 목록에서 **C#** 또는 **Visual Basic**을 선택한 다음 플랫폼 목록에서 **플랫폼** 선택합니다. **클래스 라이브러리** 템플릿을 선택한 다음 **다음**선택합니다.

   c. **새 프로젝트** 구성 페이지에서 **프로젝트 이름** 상자에 **StringLibrary** 입력한 다음, **다음**선택합니다.

   d. **기능 정보** 페이지에서 **.NET 10**을 선택한 다음, **Create** 선택합니다.

2. 라이브러리가 올바른 버전의 .NET 대상으로 지정하는지 확인합니다. **Solution Explorer** 라이브러리 프로젝트를 마우스 오른쪽 단추로 클릭한 다음 **Properties** 선택합니다. **Target Framework** 텍스트 상자에는 프로젝트가 .NET 10.0을 대상으로 하고 있음을 나타냅니다.

3. Visual Basic를 사용하는 경우 **Default 네임스페이스** 텍스트 상자에서 텍스트를 지웁니다.

   클래스 라이브러리의 프로젝트 속성

각 프로젝트에 대해 Visual Basic 프로젝트 이름에 해당하는 네임스페이스를 자동으로 만듭니다. 이 자습서에서는 코드 파일의 namespace 키워드를 사용하여 최상위 네임스페이스를 정의합니다.

4. *Class1.cs* 또는 *Class1.vb* 코드 창의 코드를 다음 코드로 바꾸고 파일을 저장합니다. 사용하려는 언어가 표시되지 않으면 페이지 맨 위에 있는 언어 선택기를 변경합니다.

   C#

   ```
   namespace UtilityLibraries;

   public static class StringLibrary
   {
       public static bool StartsWithUpper(this string? str)
       {
           if (string.IsNullOrWhiteSpace(str))
               return false;

           return char.IsUpper(str[0]);
       }
   }
   ```

   `UtilityLibraries.StringLibrary`클래스 라이브러리에는 `StartsWithUpper`메서드가 포함되어 있습니다. 이 메서드는 현재 문자열 인스턴스가 대문자로 시작하는지 여부를 나타내는 Boolean 값을 반환합니다. 유니코드 표준은 대문자를 소문자에서 구분합니다. Char.IsUpper(Char) 메서드는 문자가 대문자이면 `true` 반환합니다.

   클래스의 멤버인 것처럼 호출할 수 있도록 확장 메서드로 구현됩니다. C# 코드에서 `?` 후 물음표(`string`)는 문자열이 null일 수 있음을 나타냅니다.

5. 메뉴 모음에서 **빌드**>**빌드 솔루션**을 선택하거나 `Ctrl` + `Shift` + `B`를 눌러 프로젝트가 오류 없이 컴파일되는지 확인합니다.

솔루션에 콘솔 앱 추가

클래스 라이브러리를 사용하는 콘솔 애플리케이션을 추가합니다. 앱은 사용자에게 문자열을 입력하라는 메시지를 표시하고 문자열이 대문자로 시작하는지 여부를 보고합니다.

1. **ShowCase**라는 새 .NET 콘솔 애플리케이션을 솔루션에 추가합니다.

   a. **Solution Explorer** 솔루션을 마우스 오른쪽 단추로 클릭하고 **Add**>**새로운 프로젝트** 선택합니다.

   b. **새 프로젝트** 추가 페이지에서 검색 상자에 **콘솔** 입력하세요. 언어 목록에서 **C#** 또는 **Visual Basic**을 선택한 다음 플랫폼 목록에서 **플랫폼** 선택합니다.

c. **콘솔 앱** 템플릿을 선택한 다음, **다음**을 선택합니다.

d. **새 프로젝트** 구성 페이지에서 **프로젝트 이름** 상자에 ShowCase를 입력합니다. **다음**을 선택합니다.

e. **사용 정보** 페이지의 **프레임워크**> 상자에서 **.NET 10**을 선택합니다. 그런 다음 **만들기**선택합니다.

2. *Program.cs* 또는 *Program.vb* 파일의 코드 창에서 모든 코드를 다음 코드로 바꿉니다.

C#

```
using System;
using UtilityLibraries;

int row = 0;

do
{
    if (row == 0 || row >= 25)
        ResetConsole();

    string? input = Console.ReadLine();
    if (string.IsNullOrWhiteSpace(input)) break;
    Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                      $"{(input.StartsWithUpper() ? "Yes" : "No")}
{Environment.NewLine}");
    row += 3;
} while (true);
return;

// Declare a ResetConsole local method
void ResetConsole()
{
    if (row > 0)
    {
        Console.WriteLine("Press any key to continue...");
        Console.ReadKey();
    }
    Console.Clear();
    Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit;
otherwise, enter a string and press <Enter>:{Environment.NewLine}");
    row = 3;
}
```

이 코드는 `row` 변수를 사용하여 콘솔 창에 기록된 데이터 행 수의 수를 유지 관리합니다. 25보다 크거나 같을 때마다 코드는 콘솔 창을 지우고 사용자에게 메시지를 표시합니다.

프로그램에서 사용자에게 문자열을 입력하라는 메시지를 표시합니다. 문자열이 대문자로 시작하는지 여부를 나타냅니다. 사용자가 문자열을 입력하지 않고 `Enter` 키를 누르면 애

플리케이션이 종료되고 콘솔 창이 닫힙니다.

프로젝트 참조 추가

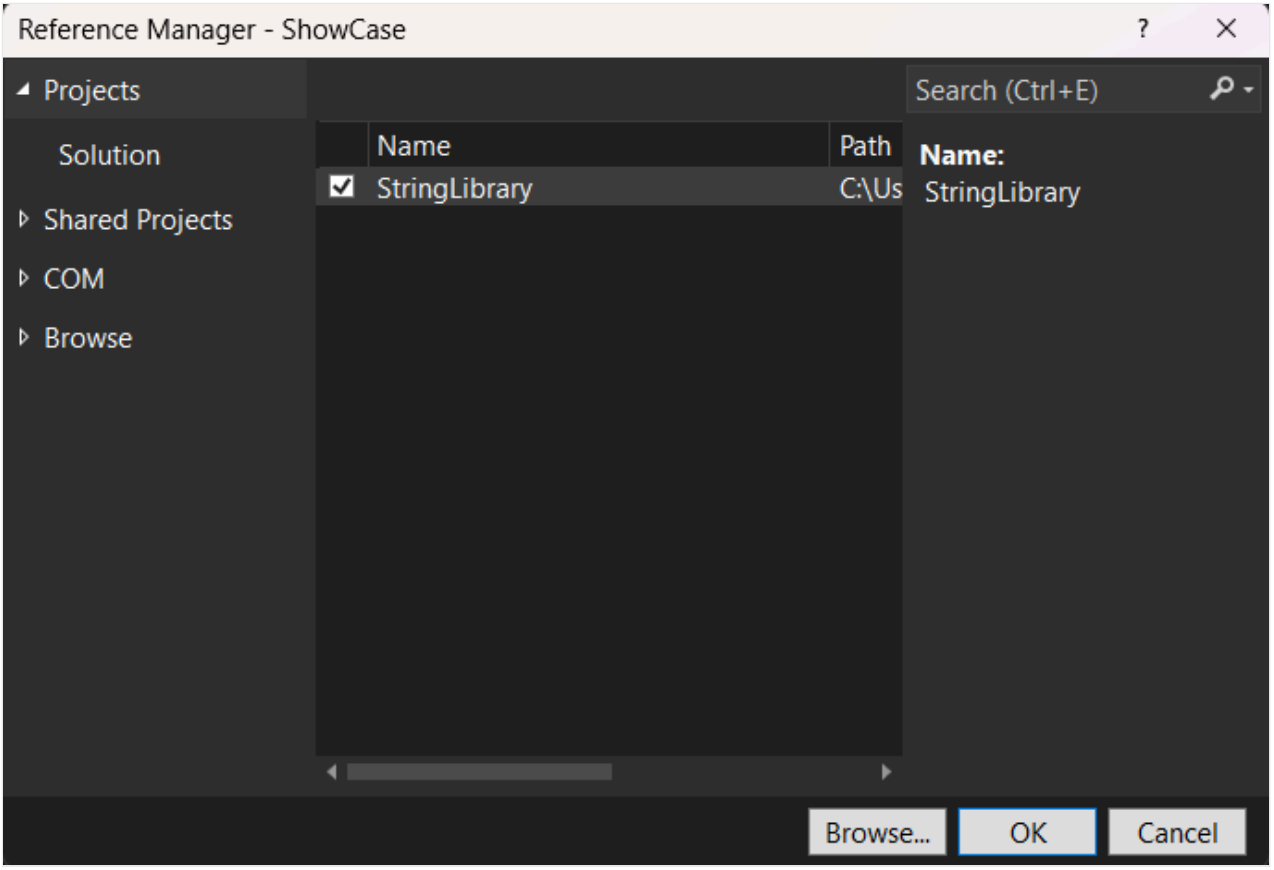
처음에는 새 콘솔 앱 프로젝트에서 클래스 라이브러리에 액세스할 수 없습니다. 클래스 라이브러리에서 메서드를 호출할 수 있도록 하려면 클래스 라이브러리 프로젝트에 대한 프로젝트 참조를 만듭니다.

1. **Solution Explorer**`ShowCase` 프로젝트의 **구성** 노드를 마우스 오른쪽 단추로 클릭하고 **프로젝트 참조 추가**를 선택합니다.

   Visual Studio에서 참조 메뉴 추가

2. **참조 관리자** 대화 상자에서 **StringLibrary** 프로젝트를 선택하고 **확인**을 선택합니다.

   StringLibrary가 선택된



앱 실행

1. **Solution Explorer**`ShowCase` 프로젝트를 마우스 오른쪽 단추로 클릭하고 상황에 맞는 메뉴에서 **시작 프로젝트로 설정** 선택합니다.

   Visual Studio 프로젝트 상황에 맞는 메뉴를 사용하여 시작 프로젝트로 설정합니다.

2. `Ctrl` + `F5` 눌러 디버깅하지 않고 프로그램을 컴파일하고 실행합니다.

3. 문자열을 입력한 후 `Enter`키를 누르고, 프로그램을 종료하려면 `Enter` 키를 누르세요.

ShowCase가 실행 중인

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:

Hello
Input: Hello        Begins with uppercase? : Yes

hello
Input: hello        Begins with uppercase? : No
```

추가 리소스

- .NET CLI를 사용하여 라이브러리를 개발합니다
- .NET 표준 버전 및 지원하는 플랫폼.

다음 단계:

이 자습서에서는 클래스 라이브러리를 만들었습니다. 다음 자습서에서는 클래스 라이브러리를 단위 테스트하는 방법을 알아봅니다.

.NET 클래스 라이브러리를 테스트합니다

또는 자동화된 단위 테스트를 건너뛰고 NuGet 패키지를 만들어 라이브러리를 공유하는 방법을 알아볼 수 있습니다.

Visual Studio

또는 콘솔 앱을 게시하는 방법을 알아봅니다. 이 자습서에서 만든 솔루션을 사용하여 콘솔 앱을 게시하면, 클래스 라이브러리가 *.dll* 파일로 함께 포함됩니다.

.NET 콘솔 애플리케이션 게시

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

자습서: .NET 클래스 라이브러리 테스트

이 자습서에서는 솔루션에 테스트 프로젝트를 추가하여 단위 테스트를 자동화하는 방법을 보여 줍니다.

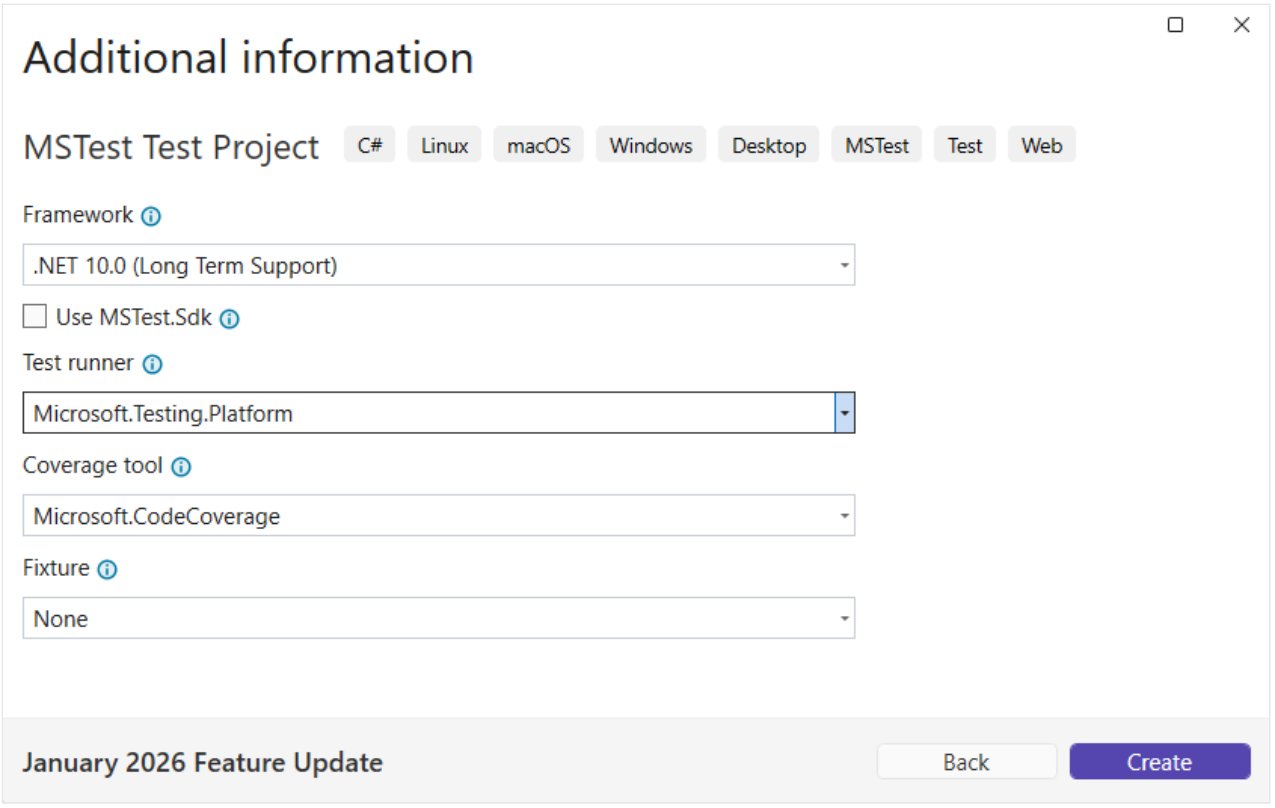
필수 조건

이 자습서는 .NET 클래스 라이브러리 만들기에서 만든 솔루션에서 작동합니다.

단위 테스트 프로젝트 만들기

단위 테스트는 개발 및 게시 중에 자동화된 소프트웨어 테스트를 제공합니다. MSTest 선택할 수 있는 세 가지 테스트 프레임워크 중 하나입니다. 다른 것들은 xUnit 과 nUnit 입니다.

1. Visual Studio 시작합니다.
2. `ClassLibraryProjects`에서 만든 솔루션을 엽니다.
3. 솔루션에 "StringLibraryTest"라는 새 단위 테스트 프로젝트를 추가합니다.
    a. **Solution Explorer** 솔루션을 마우스 오른쪽 단추로 클릭하고 **Add**>**새 프로젝트 새로 만들기**를 선택합니다.
    b. **새 프로젝트** 추가 페이지에서 검색 상자에 **mstest** 입력합니다. 언어 목록에서 **C#** 또는 **Visual Basic**을 선택한 다음 플랫폼 목록에서 **플랫폼** 선택합니다.
    c. **MSTest 테스트 프로젝트** 템플릿을 선택한 다음 **다음**선택하세요.
    d. **새 프로젝트** 구성 페이지의 **프로젝트 이름** 상자에 **StringLibraryTest** 입력합니다. **다음**을 선택합니다.
    e. **추가 정보** 페이지에서 **프레임워크** 상자에 있는 **.NET 10**을 선택하고, **테스트 실행기**에 대해 **Microsoft.Testing.Platform**을 선택한 다음, **생성**을 선택합니다.



4. Visual Studio 프로젝트를 만들고 다음 코드를 사용하여 코드 창에서 클래스 파일을 엽니다. 사용하려는 언어가 표시되지 않으면 페이지 맨 위에 있는 언어 선택기를 변경합니다.

C#

```
namespace StringLibraryTest
{

    [TestClass]
    public sealed class Test1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

단위 테스트 템플릿에서 만든 소스 코드는 다음을 수행합니다.

- C#의 StringLibraryTest 프로젝트 파일에 Microsoft.VisualStudio.TestTools.UnitTesting 포함하고 Visual Basic Microsoft.VisualStudio.TestTools.UnitTesting 가져옵니다.
- TestClassAttribute 특성을 `Test1` 클래스에 적용합니다.
- TestMethodAttribute 특성을 적용하여 C#에서 `TestMethod1` 또는 Visual Basic `TestSub` 정의합니다.

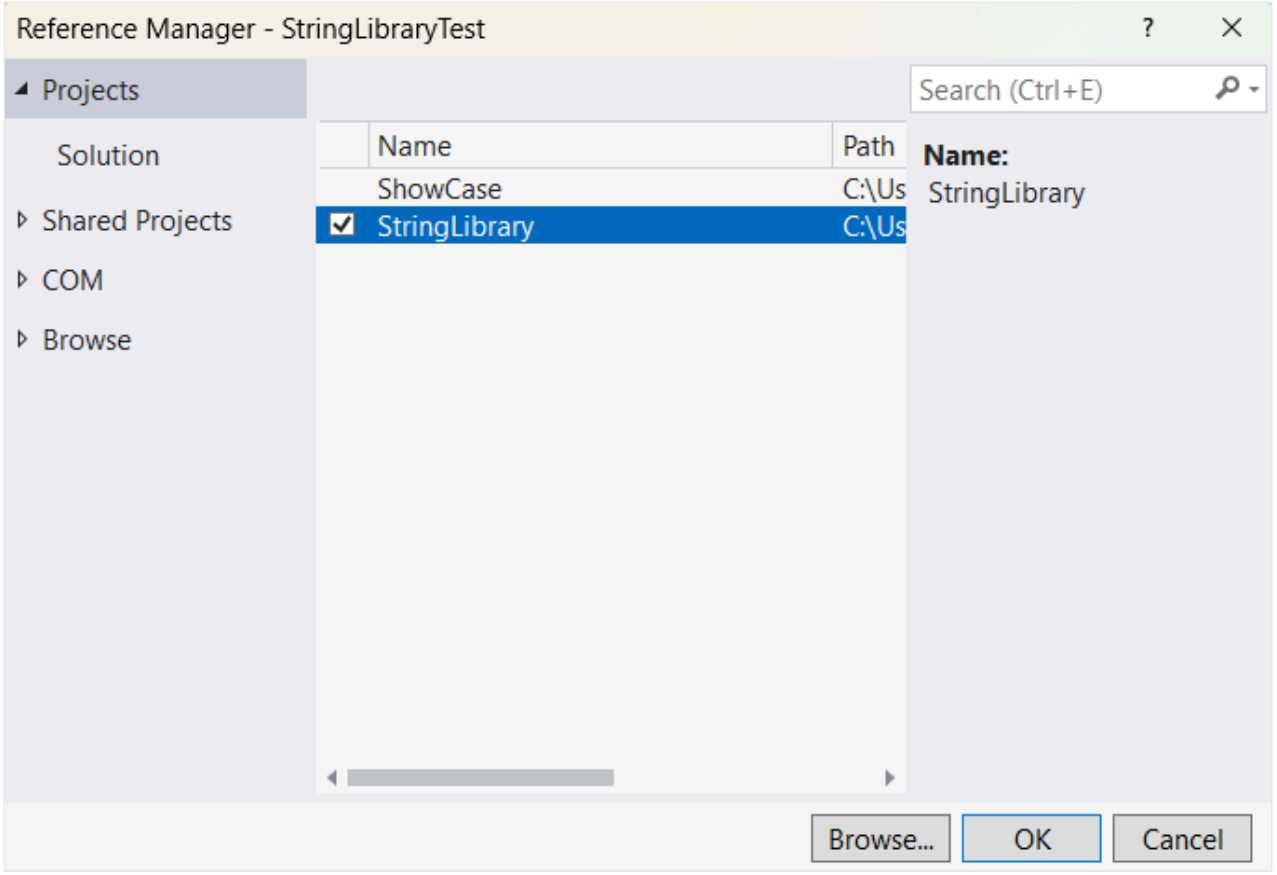
[TestClass]로 태그가 지정된 테스트 클래스에서 [TestMethod]로 태그가 지정된 각 메서드는 단위 테스트가 실행될 때 자동으로 실행됩니다.

프로젝트 참조 추가

테스트 프로젝트가 `StringLibrary` 클래스와 함께 작동하도록 하려면, 프로젝트 `StringLibraryTest`에서 프로젝트 `StringLibrary`에 대한 참조를 추가합니다.

1. Solution Explorer StringLibraryTest 프로젝트의 >Dependencies 노드를 마우스 오른쪽 단추로 클릭하고 상황에 맞는 메뉴에서 프로젝트 참조 추가를 선택합니다.

2. **참조 관리자** 대화 상자에서 **StringLibrary** 옆에 있는 상자를 선택합니다.



3. **확인**을 선택합니다.

단위 테스트 메서드 추가 및 실행

단위 테스트가 실행될 때, TestClassAttribute 특성으로 표시된 클래스 내의 각 메서드 중 TestMethodAttribute 특성으로 표시된 메서드는 자동으로 실행됩니다. 첫 번째 오류가 발견되거나 메서드에 포함된 모든 테스트가 성공하면 테스트 메서드가 종료됩니다.

가장 일반적인 테스트는 Assert 클래스의 멤버를 호출합니다. 많은 어설션 메서드에는 두 개 이상의 매개 변수가 포함되며, 그 중 하나는 예상 테스트 결과이고 다른 하나는 실제 테스트 결과입니다. `Assert` 클래스의 가장 자주 호출되는 메서드 중 일부는 다음 표에 나와 있습니다.

테이블 확장

단언 메서드	기능
`Assert.AreEqual`	두 값 또는 개체가 같은지 확인합니다. 값이나 개체가 같지 않으면 어설션이 실패합니다.
`Assert.AreSame`	두 개체 변수가 동일한 개체를 참조하는지 확인합니다. 변수가 다른 개체를 참조하는 경우 어설션이 실패합니다.
`Assert.IsFalse`	조건이 `false`인지 확인합니다. 조건이 `true`경우 어설션이 실패합니다.
`Assert.IsNotNull`	해당 개체가 `null`아닌지 확인합니다. 개체가 `null`경우 어설션이 실패합니다.

테스트 메서드에서 Assert.Throws 메서드를 사용하여 throw할 것으로 예상되는 예외 유형을 나타낼 수도 있습니다. 지정된 예외가 throw되지 않으면 테스트가 실패합니다.

메서드를 `StringLibrary.StartsWithUpper` 테스트할 때 대문자로 시작하는 여러 문자열을 제공하려고 합니다. 이러한 경우 메서드가 `true`를 반환할 것으로 기대하므로 Assert.IsTrue 메서드를 호출할 수 있습니다. 마찬가지로 대문자 이외의 문자열로 시작하는 여러 문자열을 제공하려고 합니다. 이러한 경우 메서드가 `false`를 반환할 것으로 기대하므로 Assert.IsFalse 메서드를 호출할 수 있습니다.

라이브러리 메서드가 문자열을 처리하므로 빈 문자열(String.Empty) 과 `null` 문자열을 성공적으로 처리해야 합니다. 빈 문자열은 문자가 없고 길이가 0인 문자열 Length입니다. `null` 문자열은 초기화되지 않은 문자열입니다. `StartsWithUpper`을(를) 정적 메서드로 직접 호출하고 단일 String 인수를 전달할 수 있습니다. 또는 `StartsWithUpper`을(를) 확장 메서드로 `null`에 할당된 `string` 변수에서 호출할 수 있습니다.

각각 문자열 배열의 각 요소에 대한 메서드를 Assert 호출하는 세 가지 메서드를 정의합니다. 테스트 실패 시 표시할 오류 메시지를 지정할 수 있는 메서드 오버로드를 호출합니다. 메시지는 오류를 발생시킨 문자열을 식별합니다.

테스트 메서드를 만들려면 다음을 수행합니다.

1. *Test1.cs* 또는 *Test1.vb* 코드 창에서 코드를 다음 코드로 바꿉다.

C#

```
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public sealed class Test1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
```

```
            // Tests that we expect to return true.
            string[] words = ["Alphabet", "Zebra", "ABC", "Αθήνα", "Москва"];
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result, $"Expected for '{word}': true; Actual:
{result}");
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = ["alphabet", "zebra", "abc",
"αυτοκινητοβιομηχανία", "государство",
                              "1234", ".", ";", " "];
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result, $"Expected for '{word}': false; Actual:
{result}");
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string?[] words = [string.Empty, null];
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result, $"Expected for '{word ?? "<null>"}':
false; Actual: {result}");
            }
        }
    }
}
```

`TestStartsWithUpper` 메서드의 대문자 테스트에는 그리스어 대문자 알파(U+0391) 및 키릴 자모 대문자 EM(U+041C)이 포함됩니다. `TestDoesNotStartWithUpper` 메서드의 소문자 테스트에는 그리스어 작은 문자 알파(U+03B1) 및 키릴 자모 작은 문자 Ghe(U+0433)가 포함됩니다.

2. 메뉴 모음에서 **파일**>**Test1.cs를 다른 이름으로 저장** 또는 **파일**>**Test1.vb를 다른 이름으로 저장**합니다. **파일** 저장 대화 상자에서 **저장** 단추 옆에 있는 화살표를 선택하고 **인코딩**사용하여 저장을 선택합니다.

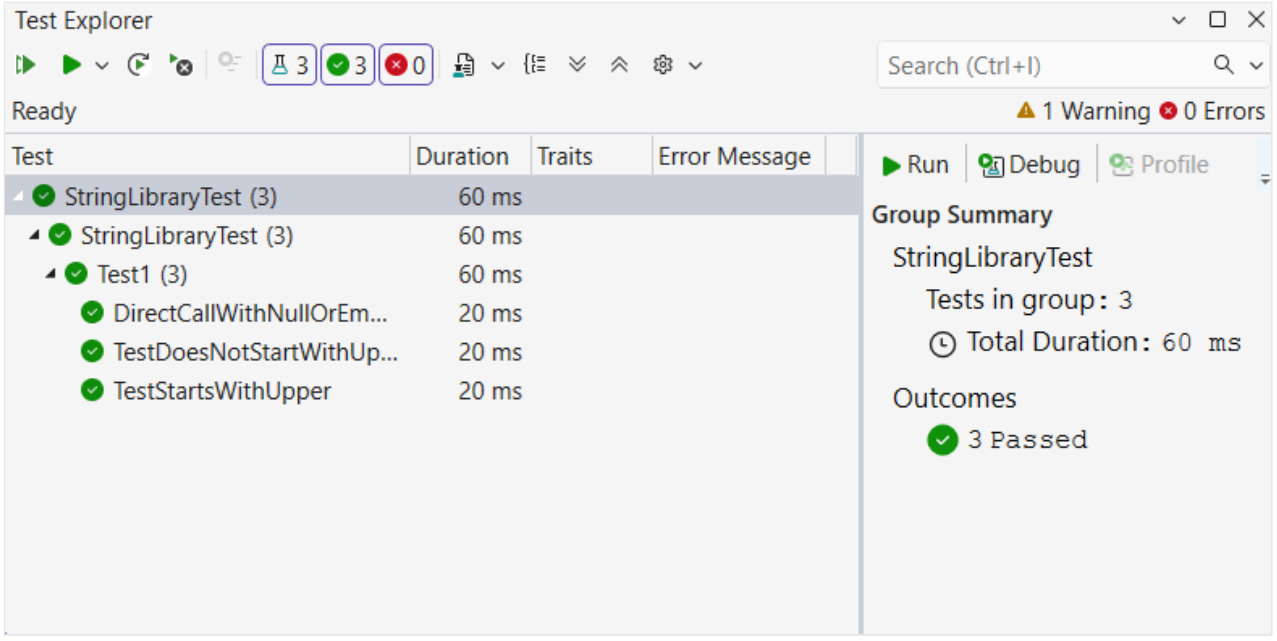
3. **다른 이름으로 저장 확인** 대화 상자에서 **예** 단추를 선택하여 파일을 저장합니다.

4. **고급 저장 옵션** 대화 상자에서 **인코딩** 드롭다운 목록에서 **유니코드(UTF-8 서명 포함) - Codepage 65001**을 선택하고 **확인**을 선택합니다.

   소스 코드를 UTF8로 인코딩된 파일로 저장하지 못하면 Visual Studio 이를 ASCII 파일로 저장할 수 있습니다. 이 경우 런타임은 ASCII 범위를 벗어난 UTF8 문자를 정확하게 디코딩하지 않으며 테스트 결과가 올바르지 않습니다.

5. 메뉴 모음에서 **테스트**>**모든 테스트 실행**선택합니다. **테스트 탐색기** 창이 열리지 않으면 **테스트**>**테스트 탐색기**선택하여 엽니다. 세 가지 테스트는 **통과한 테스트** 섹션에 나열되며, **요약** 섹션에서는 테스트 실행의 결과를 보고합니다.

   테스트가 통과된 테스트 탐색기 창



테스트 오류 처리

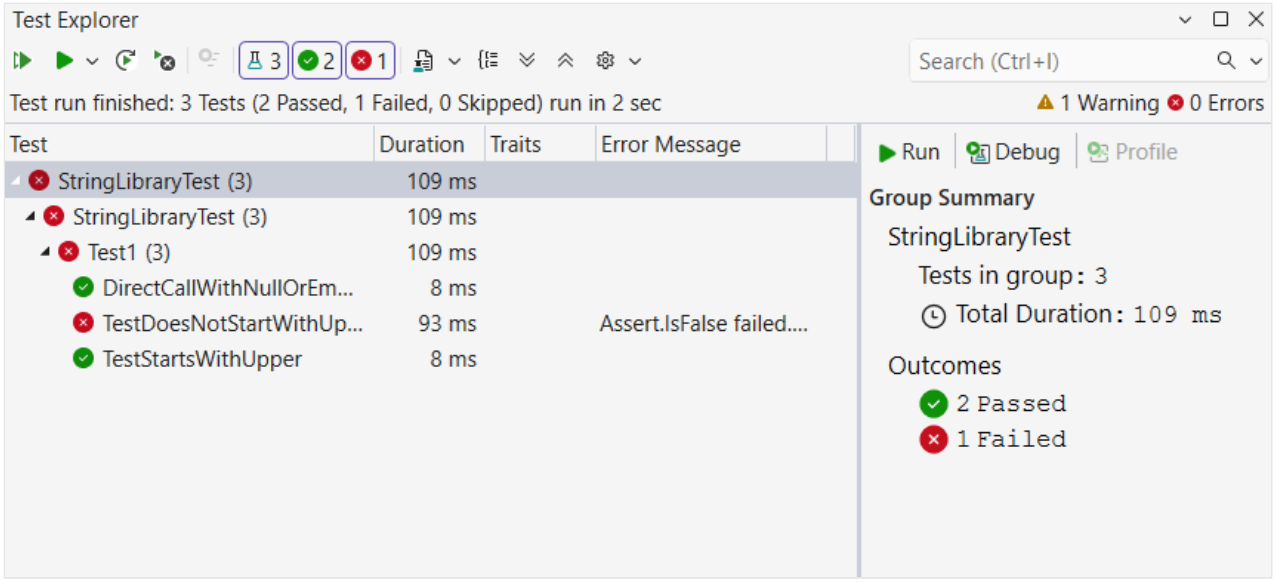
TDD(테스트 기반 개발)를 수행하는 경우 먼저 테스트를 작성하고 테스트를 처음 실행할 때 실패합니다. 그런 다음 테스트가 성공하도록 하는 코드를 앱에 추가합니다. 이 자습서에서는 유효성을 검사하는 앱 코드를 작성한 후에 테스트를 만들었기 때문에, 테스트가 실패하는 것을 보지 못했습니다. 테스트가 실패할 것으로 예상할 때 실패하는지 확인하려면 테스트 입력에 잘못된 값을 추가합니다.

1. "Error" 문자열을 포함하도록 `words` 메서드의 `TestDoesNotStartWithUpper` 배열을 수정합니다.

```C#
string[] words = { "alphabet", "Error", "zebra", "abc",
"αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

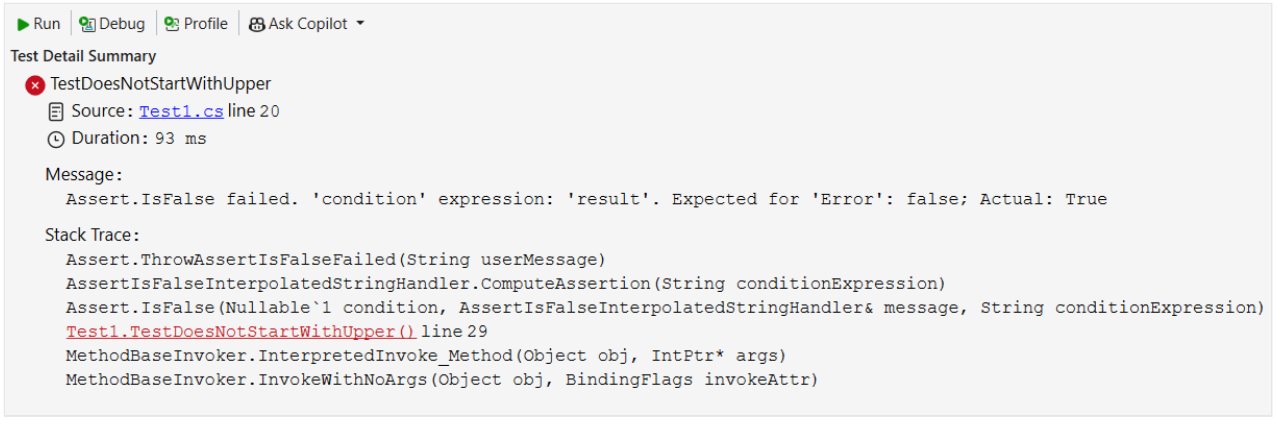
1. 메뉴 모음에서 **테스트**> 모든 테스트 실행을 선택하여 테스트를 실행하세요. **테스트 탐색기** 창은 두 개의 테스트가 성공했고 한 테스트가 실패했음을 나타냅니다.

   테스트 탐색기 창



2. 실패한 테스트, `TestDoesNotStartWith`을 선택하세요.

   테스트 탐색기 창에는 어설션에 의해 생성된 메시지가 표시됩니다. "Assert.IsFalse가 실패했습니다. 기대되는 'Error': false; 실제: True. 실패로 인해 "Error" 이후 배열의 문자열이 테스트되지 않았습니다.



1. 추가한 문자열 "Error"를 제거합니다.

2. 테스트를 다시 실행하고 테스트가 통과합니다.

라이브러리의 릴리스 버전을 테스트하세요

이제 라이브러리의 디버그 빌드를 실행할 때 테스트가 모두 통과되었으므로 라이브러리의 릴리스 빌드에 대해 테스트를 추가로 실행합니다. 컴파일러 최적화를 비롯한 다양한 요소는 디버그 빌드와 릴리스 빌드 간에 서로 다른 동작을 생성할 수 있습니다.

릴리스 빌드를 테스트하려면 다음을 수행합니다.

1. Visual Studio 도구 모음에서 빌드 구성을 **Debug**에서 **Release** 변경합니다.
2. **Solution ExplorerStringLibrary** 프로젝트를 마우스 오른쪽 단추로 클릭하고 상황에 맞는 메뉴에서 **Build**를 선택하여 라이브러리를 다시 컴파일합니다.
3. 메뉴 모음에서 **테스트**>**모든 테스트 실행** 선택하여 단위 테스트를 실행합니다. 테스트가 통과합니다.

디버그 테스트

Visual Studio IDE로 사용하는 경우 Tutorial: .NET 콘솔 애플리케이션 디버그에 표시된 것과 동일한 프로세스를 사용하여 단위 테스트 프로젝트를 사용하여 코드를 디버그할 수 있습니다. *ShowCase* 앱 프로젝트를 시작하는 대신 **StringLibraryTests** 프로젝트를 마우스 오른쪽 단추로 클릭하고 상황에 맞는 메뉴에서 **디버그 테스트** 선택합니다.

Visual Studio 디버거가 연결된 상태에서 테스트 프로젝트를 시작합니다. 테스트 프로젝트 또는 기본 라이브러리 코드에 추가한 중단점에서 실행이 중지됩니다.

추가 리소스

- 테스트 기본 사항 - Visual Studio
- .NET에서 단위 테스트

다음 단계:

이 자습서에서는 클래스 라이브러리를 단위 테스트를 수행했습니다. 라이브러리를 NuGet 에 패키지로 게시하여 다른 사용자가 사용할 수 있도록 만들 수 있습니다. 방법을 알아보려면 NuGet 자습서를 따릅니다.

dotnet CLI를 사용하여 패키지 만들기 및 게시

라이브러리를 NuGet 패키지로 게시하는 경우 다른 사용자가 라이브러리를 설치하고 사용할 수 있습니다. 방법을 알아보려면 NuGet 자습서를 따릅니다.

dotnet CLI를 사용하여 패키지 설치 및 사용

라이브러리는 패키지로 배포할 필요가 없습니다. 이를 사용하는 콘솔 앱과 함께 번들로 묶을 수 있습니다. 콘솔 앱을 게시하는 방법을 알아보려면 이 시리즈의 이전 자습서를 참조하세요.

.NET 콘솔 애플리케이션 게시

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

빠른 시작: dotnet CLI를 사용하여 패키지 설치 및 사용

아티클 • 2025. 04. 04.

NuGet 패키지에는 개발자가 다른 개발자가 프로젝트에서 사용할 수 있도록 하는 컴파일된 이진 코드가 포함되어 있습니다. 자세한 내용은 NuGet이란?을 참조하세요. 이 빠른 시작에서는 dotnet add package 명령을 사용하여 인기 있는 Newtonsoft.Json NuGet 패키지를 .NET 프로젝트에 설치하는 방법을 설명합니다.

코드에서 `using <namespace>` 지시문을 사용하여 설치된 패키지를 참조합니다. 여기서 `<namespace>`는 종종 패키지 이름입니다. 그런 다음 프로젝트에서 패키지의 API를 사용할 수 있습니다.

> **팁**
>
> nuget.org/packages 탐색하여 사용자 고유의 애플리케이션에서 다시 사용할 수 있는 패키지를 찾습니다. Visual Studio 내에서 https://nuget.org 직접 검색하거나 패키지를 찾아 설치할 수 있습니다. 자세한 내용은 **프로젝트에 대한 NuGet 패키지 찾기 및 계산을 참조하세요**.

필수 조건

- .NET SDK는 `dotnet` 명령줄 도구를 제공합니다. Visual Studio 2017부터 dotnet CLI는 .NET 또는 .NET Core 관련 워크로드와 함께 자동으로 설치됩니다.

프로젝트 만들기

.NET 프로젝트에 NuGet 패키지를 설치할 수 있습니다. 이 연습에서는 다음과 같이 dotnet CLI를 사용하여 간단한 .NET 콘솔 프로젝트를 만듭니다.

1. 프로젝트에 대한 *Nuget.Quickstart* 폴더를 만듭니다.
2. 명령 프롬프트를 열고 새 폴더로 전환합니다.
3. 다음 명령을 사용하여 프로젝트를 만듭니다.

.NET CLI

```
dotnet new console
```

4. 앱을 테스트하는 데 사용합니다 `dotnet run` . `Hello, World!` 출력이 표시됩니다.

Newtonsoft.Json NuGet 패키지 추가

1. 다음 명령을 사용하여 `Newtonsoft.json` 패키지를 설치합니다.

.NET CLI

```
dotnet add package Newtonsoft.Json
```

2. 명령이 완료되면 Visual Studio에서 *Nuget.Quickstart.csproj* 파일을 열어 추가된 NuGet 패키지 참조를 확인합니다.

XML

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
</ItemGroup>
```

앱에서 Newtonsoft.Json API 사용

1. Visual Studio에서 *Program.cs* 파일을 열고 파일 맨 위에 다음 줄을 추가합니다.

cs

```
using Newtonsoft.Json;
```

2. 다음 코드를 추가하여 `Console.WriteLine("Hello, World!");` 문을 대체하세요.

cs

```
namespace Nuget.Quickstart
{
    public class Account
    {
        public string? Name { get; set; }
        public string? Email { get; set; }
        public DateTime DOB { get; set; }
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            Account account = new Account
            {
```

```
                Name = "John Doe",
                Email = "john@nuget.org",
                DOB = new DateTime(1980, 2, 20, 0, 0, 0,
DateTimeKind.Utc),
            };

            string json = JsonConvert.SerializeObject(account,
Formatting.Indented);
            Console.WriteLine(json);
        }
    }
}
```

3. 파일을 저장한 다음 명령을 사용하여 앱을 빌드하고 실행합니다 `dotnet run` . 출력은 코드에 있는 개체의 `Account` JSON 표현입니다.

출력

```
{
  "Name": "John Doe",
  "Email": "john@nuget.org",
  "DOB": "1980-02-20T00:00:00Z"
}
```

첫 번째 NuGet 패키지를 설치하고 사용하는 것을 축하합니다.

관련 비디오

https://learn.microsoft.com/shows/NuGet-101/Install-and-Use-a-NuGet-Package-with-the-NET-CLI-3-of-5/player

채널 9 및 YouTube 에서 더 많은 NuGet 비디오를 찾습니다.

다음 단계

dotnet CLI를 사용하여 NuGet 패키지를 설치하고 사용하는 방법에 대해 자세히 알아봅니다.

dotnet CLI를 사용하여 패키지 설치 및 사용

- 패키지 사용 개요 및 워크플로
- 패키지 찾기 및 선택
- 프로젝트 파일에서 패키지 참조

피드백

이 페이지가 도움이 되었나요? Yes No

제품 사용자 의견 제공

빠른 시작: dotnet CLI를 사용하여 패키지 만들기 및 게시

아티클 • 2025. 03. 03.

이 빠른 시작에서는 .NET 클래스 라이브러리에서 NuGet 패키지를 빠르게 만들고 .NET 명령줄 인터페이스 또는 dotnet CLI를 사용하여 nuget.org 게시하는 방법을 보여줍니다.

필수 조건

- dotnet 명령줄 도구를 제공하는 .NET SDK 입니다. Visual Studio 2017부터 dotnet CLI는 .NET 또는 .NET Core 관련 워크로드와 함께 자동으로 설치됩니다.
- nuget.org 무료 계정. 새 개별 계정 추가의 지침을 따릅니다.

클래스 라이브러리 프로젝트 만들기

패키지하려는 코드에 기존 .NET 클래스 라이브러리 프로젝트를 사용하거나 다음과 같이 간단한 프로젝트를 만들 수 있습니다.

1. AppLogger*라는* 폴더를 만듭니다.
2. 명령 프롬프트를 열고 AppLogger *폴더로* 전환합니다. 이 빠른 시작의 모든 dotnet CLI 명령은 기본적으로 현재 폴더에서 실행됩니다.
3. 현재 폴더 이름을 사용하여 프로젝트를 만드는 를 입력 `dotnet new classlib`합니다.

자세한 내용은 dotnet new를 참조합니다.

프로젝트 파일에 패키지 메타데이터 추가

모든 NuGet 패키지에는 패키지의 콘텐츠 및 종속성을 설명하는 매니페스트가 있습니다. 최종 패키지에서 매니페스트는 *프로젝트 파일에 포함하는 NuGet 메타데이터 속성을 사용하는 .nuspec* 파일입니다.

.csproj, *.fsproj* 또는 *.vbproj* 프로젝트 파일을 열고 기존 `<PropertyGroup>` 태그 내에 다음 속성을 추가합니다. 이름 및 회사에 고유한 값을 사용하고 패키지 식별자를 고유한 값으로 바꿉다.

```
XML
```

```
<PackageId>Contoso.08.28.22.001.Test</PackageId>
<Version>1.0.0</Version>
<Authors>your_name</Authors>
<Company>your_company</Company>
```

> **ⓘ 중요**
>
> 패키지 식별자는 nuget.org 및 기타 패키지 원본에서 고유해야 합니다. 게시하면 패키지가 공개적으로 표시되므로 예제 AppLogger 라이브러리 또는 다른 테스트 라이브러리를 사용하는 경우 포함 `Sample` 하거나 `Test` 포함하는 고유한 이름을 사용합니다.

NuGet 메타데이터 속성에 설명된 선택적 속성을 추가할 수 있습니다.

> **ⓘ 참고**
>
> 공용 사용을 위해 빌드하는 패키지의 경우 속성에 `PackageTags` 특히 주의해야 합니다. 태그는 다른 사용자가 패키지를 찾아서 수행하는 작업을 이해하는 데 도움이 됩니다.

pack 명령 실행

프로젝트에서 NuGet 패키지 또는 *.nupkg* 파일을 빌드하려면 dotnet pack 명령을 실행합니다. 이 명령은 프로젝트를 자동으로 빌드합니다.

.NET CLI

```
dotnet pack
```

출력은 .nupkg *파일의* 경로를 보여 줍니다.

출력

```
MSBuild version 17.3.0+92e077650 for .NET
  Determining projects to restore...
  Restored C:\Users\myname\source\repos\AppLogger\AppLogger.csproj (in 64
ms).
  AppLogger ->
C:\Users\myname\source\repos\AppLogger\bin\Debug\net6.0\AppLogger.dll
  Successfully created package
```

```
'C:\Users\myname\source\repos\AppLogger\bin\Debug\Contoso.08.28.22.001.Test.
1.0.0.nupkg'.
```

빌드 시 패키지를 자동으로 생성

실행할 `dotnet build`때마다 자동으로 실행 `dotnet pack` 하려면 다음 줄을 프로젝트 파일에 `<PropertyGroup>`추가합니다.

XML

```
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
```

패키지 게시

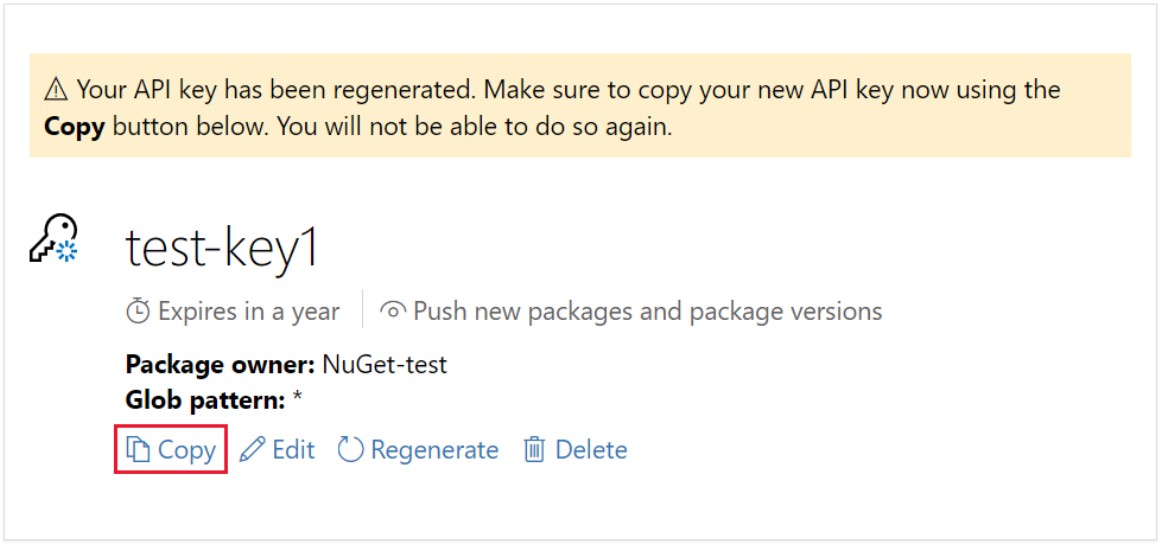
nuget.org 가져오는 API 키와 함께 dotnet nuget 푸시 명령을 사용하여 .nupkg 파일을 nuget.org 게시합니다.

> **참고**
>
> - Nuget.org 바이러스에 대해 업로드된 모든 패키지를 검사하고 바이러스가 발견되면 패키지를 거부합니다. 또한 Nuget.org 나열된 모든 기존 패키지를 주기적으로 검사합니다.
> - nuget.org 게시하는 패키지는 목록에 없는 한 다른 개발자에게 공개적으로 표시됩니다. 패키지를 비공개로 호스트하려면 사용자 고유의 NuGet 피드 호스트를 참조 **하세요**.

API key 가져오기

1. nuget.org 계정에 로그인하거나 계정이 없는 경우 계정을 만듭니다.
2. 오른쪽 위에서 사용자 이름을 선택한 다음, API 키를 **선택합니다**.
3. 만들기**를 선택하고** 키의 이름을 입력합니다.
4. 범위 선택에서 **푸시를 선택합니다.**
5. 패키지>**선택 Glob 패턴에서** *를 입력합니다.
6. **만들기**를 실행합니다.

7. 복사**를 선택하여** 새 키를 복사합니다.



> **ⓘ 중요**
>
> - 항상 API 키를 비밀로 유지합니다. API 키는 누구나 사용자를 대신하여 패키지를 관리할 수 있는 암호와 같습니다. 실수로 표시되는 경우 API 키를 삭제하거나 다시 생성합니다.
> - 나중에 키를 다시 복사할 수 없으므로 키를 안전한 위치에 저장합니다. API 키 페이지로 돌아갈 경우 키를 복사하려면 키를 다시 생성해야 합니다. 더 이상 패키지를 푸시하지 않으려는 경우 API 키를 제거할 수도 있습니다.

범위 지정을 사용하면 다양한 용도로 별도의 API 키를 만들 수 있습니다. 각 키에는 만료 기간이 있으며 키 범위를 특정 패키지 또는 glob 패턴으로 지정할 수 있습니다. 또한 각 키의 범위를 특정 작업(새 패키지 및 패키지 버전 푸시, 새 패키지 버전만 푸시 또는 목록 해제)으로 지정합니다.

범위 지정을 통해 조직의 패키지를 관리하는 다른 사용자에 대한 API 키를 만들어 필요한 권한만 가질 수 있습니다.

자세한 내용은 범위가 지정된 API 키를 참조하세요.

dotnet nuget push로 게시

.nupkg *파일이 포함된* 폴더에서 다음 명령을 실행합니다. *.nupkg* 파일 이름을 지정하고 키 값을 API 키로 바꿉다.

.NET CLI

```
dotnet nuget push Contoso.08.28.22.001.Test.1.0.0.nupkg --api-key
qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 --source
https://api.nuget.org/v3/index.json
```

출력은 게시 프로세스의 결과를 보여줍니다.

출력

```
Pushing Contoso.08.28.22.001.Test.1.0.0.nupkg to
'https://www.nuget.org/api/v2/package'...
  PUT https://www.nuget.org/api/v2/package/
warn : All published packages should have license information specified.
Learn more: https://aka.ms/nuget/authoring-best-practices#licensing.
  Created https://www.nuget.org/api/v2/package/ 1221ms
Your package was pushed.
```

자세한 내용은 dotnet nuget 푸시를 참조하세요.

> **참고**
>
> 테스트 패키지가 nuget.org 라이브되지 않도록 하려면 nuget.org 테스트 사이트 https://int.nugettest.org 로 푸시할 수 있습니다. int.nugettest.org 업로드된 패키지는 보존되지 않을 수 있습니다.

게시 오류

`push` 명령의 오류는 일반적으로 문제를 나타냅니다. 예를 들어 프로젝트에서 버전 번호를 업데이트하는 것을 잊어버렸을 수 있으므로 이미 존재하는 패키지를 게시하려고 합니다.

API 키가 잘못되었거나 만료되었거나 호스트에 이미 있는 식별자를 사용하여 패키지를 게시하려고 하면 오류가 표시됩니다. 예를 들어 식별자가 `AppLogger-test` 이미 nuget.org 있다고 가정합니다. 해당 식별자를 `push` 사용하여 패키지를 게시하려고 하면 다음 오류가 표시됩니다.

출력

```
Response status code does not indicate success: 403 (The specified API key
is invalid,
has expired, or does not have permission to access the specified package.).
```

이 오류가 발생하면 만료되지 않은 유효한 API 키를 사용하고 있음을 검사. 이 오류는 패키지 식별자가 호스트에 이미 있음을 나타냅니다. 오류를 해결하려면 패키지 식별자를 고

유하게 변경하고, 프로젝트를 다시 빌드하고, .nupkg 파일을 다시 *만들고, 명령을 다시 시도합니다* `push`.

게시된 패키지 관리

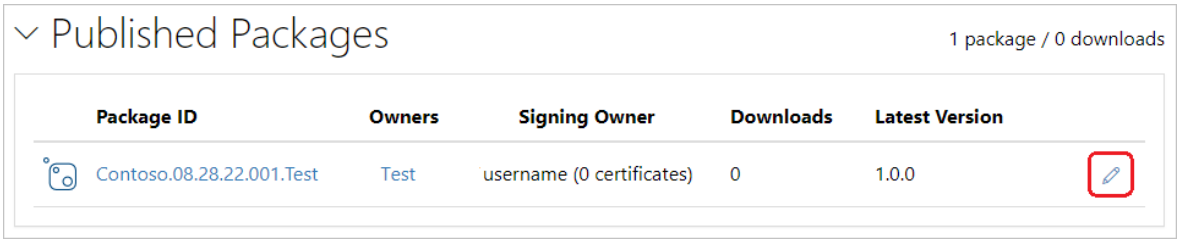
패키지가 성공적으로 게시되면 확인 이메일을 받게 됩니다. 방금 게시한 패키지를 보려면 nuget.org 오른쪽 위에서 사용자 이름을 선택한 다음 패키지 **관리를 선택합니다**.

> **참고**
>
> 패키지를 인덱싱하고 다른 사용자가 찾을 수 있는 검색 결과에 표시하는 데 시간이 걸릴 수 있습니다. 이 시간 동안 패키지가 목록에 없는 패키지 **아래에** 나타나고 패키지 페이지에는 다음 메시지가 표시됩니다.
>
> ⚠ **This package has not been published yet.** It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. Read more.

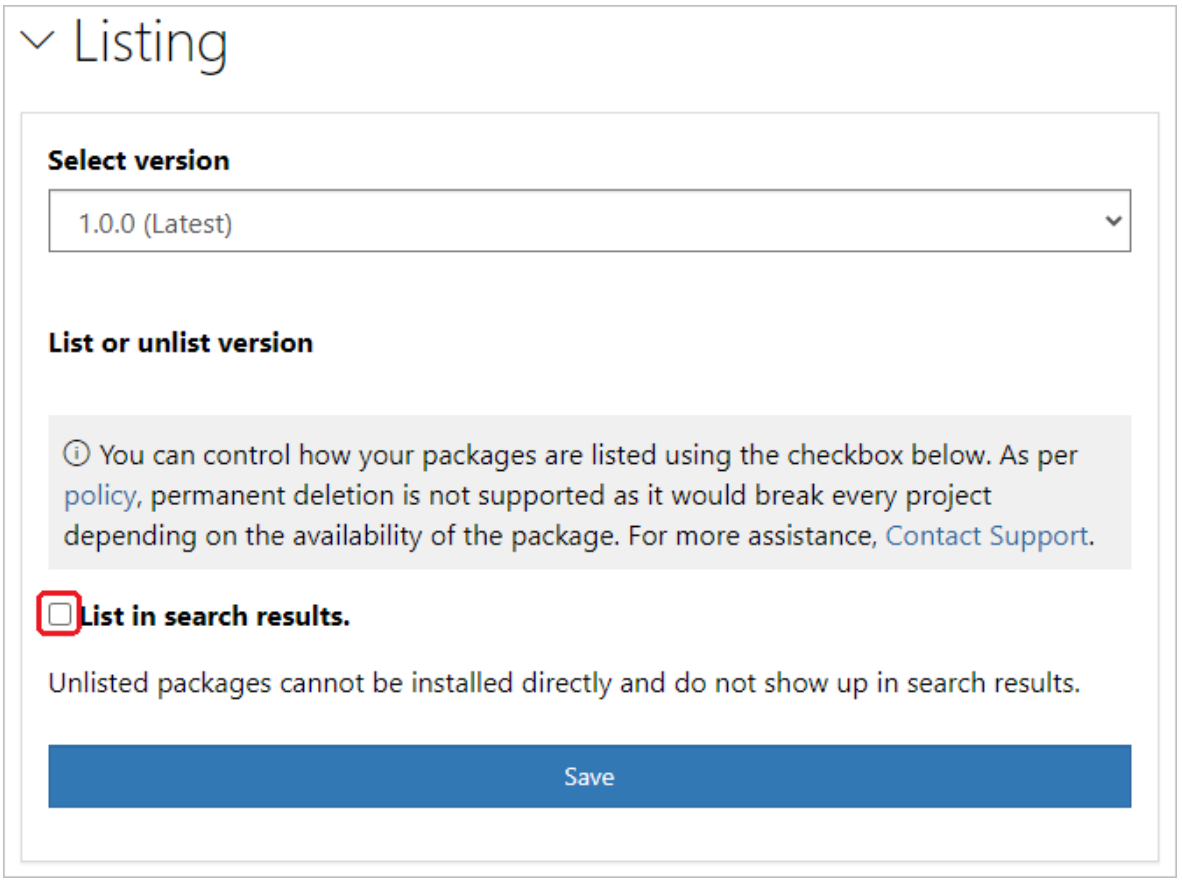
이제 다른 개발자가 프로젝트에서 사용할 수 있는 nuget.org NuGet 패키지를 게시했습니다.

유용하지 않은 패키지(예: 빈 클래스 라이브러리를 사용하여 만든 이 샘플 패키지)를 만들었거나 패키지를 표시하지 않기로 결정한 경우 패키지를 목록 *해제하여 검색 결과에서 숨길 수 있습니다*.

1. 패키지 관리 페이지의 게시된 패키지 **아래에 패키지가 표시되면 패키지** 목록 옆에 있는 연필 아이콘을 선택합니다.



2. 다음 페이지에서 목록을 선택하고 **검색 결과 검사 상자에서 목록을 선택 취소한 다음 저장을 선택합니다.**



이제 패키지가 패키지 관리의 목록에 없는 패키지 **아래에** 표시되고 더 이상 검색 결과에 표시되지 않습니다.

> **참고**
>
> 테스트 패키지가 nuget.org 라이브로 전환되지 않도록 하려면 nuget.org 테스트 사이트 https://int.nugettest.org 로 푸시할 수 있습니다. int.nugettest.org 업로드된 패키지는 보존되지 않을 수 있습니다.

첫 번째 NuGet 패키지를 만들고 게시해 주셔서 감사합니다.

관련 비디오

https://learn.microsoft.com/shows/NuGet-101/Create-and-Publish-a-NuGet-Package-with-the-NET-CLI-5-of-5/player

Channel 9 및 YouTube 에서 더 많은 NuGet 비디오를 확인하세요.

다음 단계

dotnet CLI를 사용하여 패키지를 만드는 방법에 대한 자세한 내용은 다음을 참조하세요.

dotnet CLI를 사용하여 NuGet 패키지 만들기

NuGet 패키지를 만들고 게시하는 방법에 대한 자세한 정보를 가져옵니다.

- 패키지 게시
- 시험판 패키지
- 여러 대상 프레임워크 지원
- 패키지 버전 관리
- 라이선스 식 또는 파일 추가
- 지역화된 패키지 만들기
- 기호 패키지 만들기
- 패키지 서명

피드백

이 페이지가 도움이 되었나요? Yes No

제품 사용자 의견 제공

C# 콘솔 앱 템플릿은 최상위 문을 생성합니다.

.NET 6부터 새 C# 콘솔 앱에 대한 프로젝트 템플릿은 *Program.cs* 파일에 다음 코드를 생성합니다.

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

.NET 5 이전 버전의 경우 콘솔 앱 템플릿은 다음 코드를 생성합니다.

C#

```
using System;

namespace MyApp
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

앞의 코드에서 실제 네임스페이스는 프로젝트 이름에 따라 달라집니다.

이러한 두 양식은 동일한 프로그램을 나타냅니다. 둘 다 C#에서 유효합니다. 최신 버전을 사용하는 경우 `Main` 메서드 본문을 작성하기만 하면 됩니다. 컴파일러는 진입점 메서드를 사용하여 `Program` 클래스를 생성하고 모든 최상위 문을 해당 메서드에 배치합니다. 생성된 메서드의 이름은 아닙니다 `Main`. 코드에서 직접 참조할 수 없는 구현 세부 정보입니다. 다른 프로그램 요소를 포함할 필요가 없습니다. 컴파일러에서 자동으로 생성합니다. 최상위 문을 사용할 때 컴파일러가 생성하는 코드에 대한 자세한 내용은 C# 가이드의 기본 사항 섹션에서 최상위 문을 참조하세요.

.NET 6개 이상의 템플릿을 사용하도록 업데이트되지 않은 자습서를 사용하는 경우 다음 두 가지 옵션 중 하나를 사용합니다.

- 새 프로그램 스타일을 사용하여 기능을 추가할 때 새 최상위 문을 추가합니다.
- 클래스와 `Program` 메서드를 사용하여 새 프로그램 스타일을 이전 스타일로 `Main` 변환합니다.

이전 템플릿을 사용하려면 이 문서의 뒷 부분에 있는 이전 프로그램 스타일 사용을 참조하세요.

새 프로그램 스타일 사용

새 프로그램을 더 간단하게 만드는 기능은 *최상위 문*, *전역* `using` *지시문* 및 *암시적* `using` *지시문입니다*.

*최상위 문*이라는 용어는 컴파일러가 주 프로그램에 대한 클래스 및 메서드 요소를 생성한다는 것을 의미합니다. 컴파일러는 전역 네임스페이스에서 생성된 클래스 및 진입점 메서드를 선언합니다. 새 애플리케이션의 코드를 살펴보고, 이전 템플릿에서 생성된 `Main` 메서드 내부의 문장들이 전역 네임스페이스에 존재한다고 상상해 보세요.

기존 스타일에서 메서드에 문을 추가하듯이 프로그램에도 문을 더 추가하세요 `Main` . 액세스 args (명령줄 인수)를 사용하고 await종료 코드를 설정합니다. 함수를 추가할 수도 있습니다. 컴파일러는 생성된 진입점 메서드 내에 중첩된 로컬 함수로 만듭니다. 로컬 함수는 액세스 한정자(예: `public` 또는 `protected`)를 포함할 수 없습니다.

최상위 문과 암시적 using 지시문 은 모두 애플리케이션을 구성하는 코드를 간소화합니다. 기존 자습서를 수행하려면 템플릿에서 생성된 *Program.cs* 파일에 새 문을 추가합니다. 작성하는 문이 자습서 지침의 메서드 안에 있는 열린 중괄호와 닫는 중괄호 `Main` 사이에 있다고 상상해 보십시오.

이전 형식을 원하는 경우 이 문서의 두 번째 예제에서 코드를 복사하고 이전과 같이 자습서를 계속 진행합니다.

최상위 문에 대한 자세한 내용은 최상위 문에 대한 자습서 탐색을 참조하세요.

암시적 `using` 지시문

암시적 `using` *지시문*이라는 용어는 컴파일러가 프로젝트 형식에 따라 지시문 집합using을 자동으로 추가한다는 것을 의미합니다. 콘솔 애플리케이션의 경우 다음 지시문은 애플리케이션에 암시적으로 포함됩니다.

- `using System;`
- `using System.IO;`
- `using System.Collections.Generic;`
- `using System.Linq;`
- `using System.Net.Http;`
- `using System.Threading;`
- `using System.Threading.Tasks;`

다른 애플리케이션 유형에는 해당 애플리케이션 유형에 공통적인 더 많은 네임스페이스가 포함됩니다.

암시적으로 포함되지 않은 지시문이 필요한 `using` 경우 최상위 문이 포함된 *.cs* 파일 또는 다른 *.cs* 파일에 추가합니다. 애플리케이션의 모든 `using` 파일에 필요한 지시문의 경우 전역 using 지시문을 사용합니다.

암시적 `using` 지시문 사용 안 함

이 동작을 제거하고 프로젝트의 모든 네임스페이스를 수동으로 제어하려면 다음 예제와 같이 프로젝트 파일의 `<PropertyGroup>` 요소에 <ImplicitUsings>disable</ImplicitUsings> 를 추가하십시오.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    ...
    <ImplicitUsings>disable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

전역 `using` 지시문

*전역 `using` 지시문*은 단일 파일 대신 전체 애플리케이션에 대한 네임스페이스를 가져옵니다. 프로젝트 파일에 <Using> 항목을 포함하거나 코드 파일에 `global using` 지시문을 추가하여 이러한 전역 지시문을 추가합니다.

특정 암시적 using 지시문을 제거하려면 특성이 있는 <Using> 항목을 `Remove` 프로젝트 파일에 추가합니다. 예를 들어 암시적 `using` 지시문 기능을 켠 <ImplicitUsings>enable</ImplicitUsings> 경우, 다음 `<Using>` 항목을 추가하면 암시적으로 가져오는 네임스페이스에서 제거됩니다.

XML

```xml
<ItemGroup>
  <Using Remove="System.Net.Http" />
</ItemGroup>
```

이전 프로그램 스타일 사용

.NET SDK 6.0.300부터 `console` 템플릿에는 `--use-program-main` 옵션이 있습니다. 이를 사용하여 최상위 문이 없고 `Main` 메서드가 있는 콘솔 프로젝트를 만드세요.

.NET CLI

```
dotnet new console --use-program-main
```

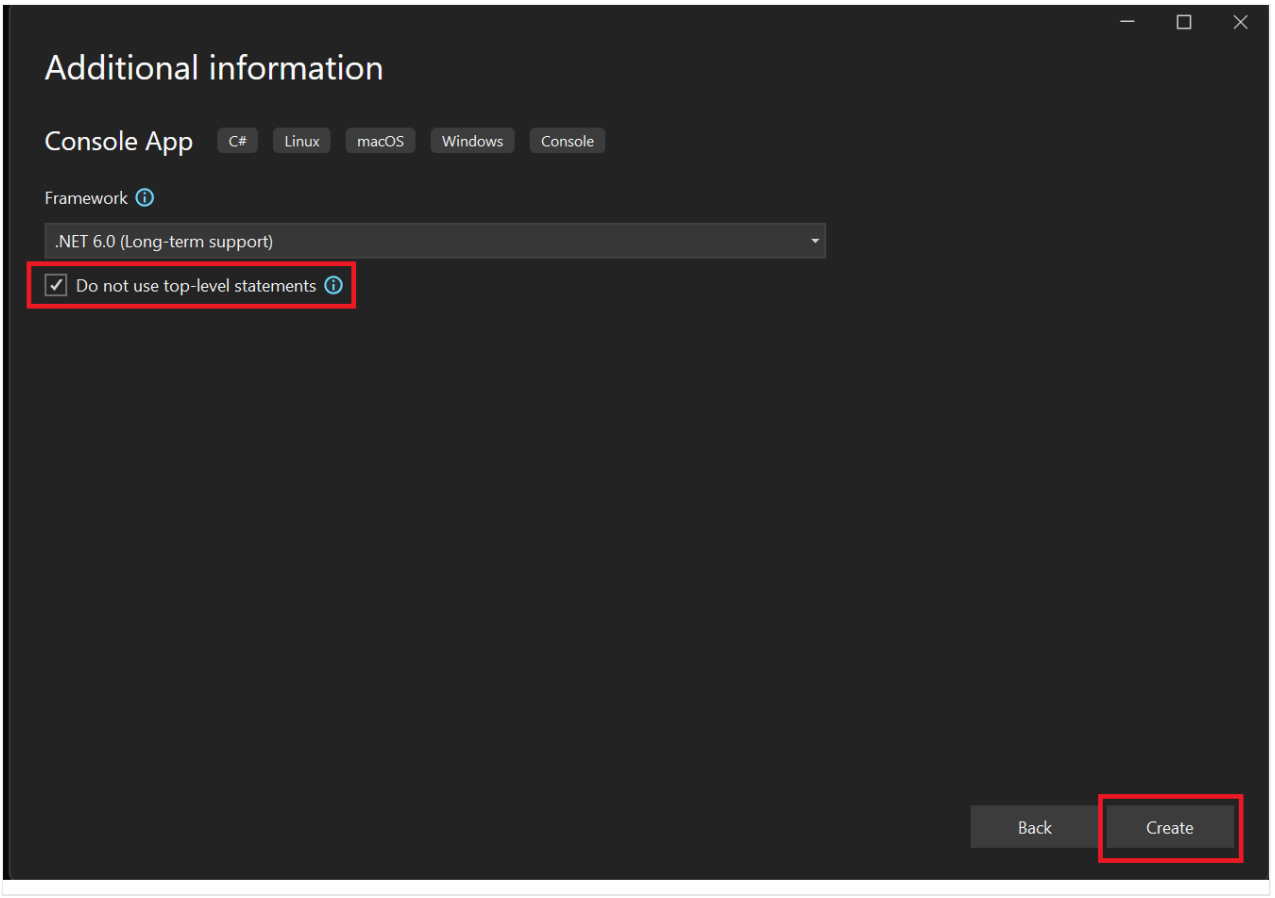
생성된 `Program.cs` 내용은 다음과 같습니다.

C#

```
namespace MyProject;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Visual Studio 이전 프로그램 스타일 사용

1. 새 프로젝트를 만들 때 설치 단계는 **추가 정보** 설정 페이지로 이동합니다. 이 페이지에서 **최상위 문 사용 안 함** 확인란을 선택합니다.



2. 프로젝트가 생성된 후 `Program.cs` 콘텐츠는 다음과 같습니다.

C#

```
namespace MyProject;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

> ⓘ **참고 항목**
>
> Visual Studio 다음에 동일한 템플릿을 기반으로 프로젝트를 만들 때 옵션 값을 유지합니다. **최상위 문 사용 안 함** 확인란이 선택된 콘솔 앱 프로젝트를 마지막으로 만든 경우 다음 콘솔 앱 프로젝트를 만들 때 해당 옵션이 선택됩니다. `Program.cs` 파일의 콘텐츠는 전역 Visual Studio 텍스트 편집기 설정 또는 `EditorConfig` 파일에 정의된 코드 스타일과 일치하도록 다를 수 있습니다.
>
> 자세한 내용은 EditorConfig 및 옵션, 텍스트 편집기, **C#, 고급**을 **사용하여 이식 가능한 사용자 지정 편집기 설정 만들기**를 참조하세요.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

이러한 자습서를 탐색하여 .NET 및 .NET SDK 도구 알아보기

다음 자습서에서는 .NET Core, .NET 5 이상 버전용 콘솔 앱 및 라이브러리를 개발하는 방법을 보여 줍니다. 다른 유형의 애플리케이션에 대해서는 .NET 시작하기 튜토리얼을 참조하세요.

앱 및 라이브러리 만들기

이러한 자습서에서는 Visual Studio, Visual Studio Code 및 GitHub Codespace를 지원합니다. 각 문서의 맨 위에 있는 영역 피벗 선택기를 사용하여 선호하는 개발 환경을 선택합니다.

- 콘솔 앱 만들기
- 앱 디버그
- 앱 게시
- 클래스 라이브러리 만들기
- 클래스 라이브러리 단위 테스트
- 패키지 설치 및 사용
- 패키지 만들기 및 게시

F# 자습서

- Visual Studio
- Visual Studio Code

고급 문서

- 라이브러리를 만드는 방법
- xUnit을 사용하여 앱을 단위 테스트하다
- NUnit/xUnit/MSTest에서 C#/VB/F#을 사용한 단위 테스트
- Visual Studio
- CLI용 템플릿 만들기
- CLI용 도구 만들기 및 사용
- 플러그 인을 사용하여 앱 만들기

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

.NET 11의 새로운 기능

이 문서에서는 .NET 11의 새로운 기능에 대해 설명합니다. 미리 보기 2에 대해 마지막으로 업데이트되었습니다.

.NET 11은 현재 미리 보기로 제공됩니다. 최종 릴리스는 2026년 11월에 출시될 예정입니다. 여기에서 .NET 11을 다운로드 할 수 있습니다.

여러분의 피드백은 중요하고 감사하게 생각합니다. 질문이나 의견이 있는 경우 GitHub 대한 토론을 사용하세요.

.NET 런타임

.NET 11 런타임에는 다음이 포함됩니다.

- x86/x64 및 Arm64 아키텍처에 대한 최소 하드웨어 요구 사항이 업데이트되어 성능을 향상시키고 유지 관리 복잡성을 줄이기 위해 최신 명령 집합이 필요했습니다.
- 더 깨끗한 스택 추적과 낮은 오버헤드를 생성하는 런타임 네이티브 비동기(런타임 비동기) 입니다.
- JIT 개선 사항으로는 경계 검사 제거, 중복 검사 컨텍스트 제거, 그리고 새로운 Arm SVE2 내장 함수를 포함합니다.

자세한 내용은 .NET 11 런타임의 새로운 기능입니다.

.NET 라이브러리

.NET 11 라이브러리에는 다음을 위한 새 API가 포함됩니다.

- 문자열 및 문자 조작, String의 Rune 기반 작업 및 BitConverter의 BFloat16 지원을 포함하여.
- 향상된 Base64 API 및 ZIP 보관 항목에 대한 새로운 메서드를 포함한 압축
- System.Text.Json에서 제네릭 형식 정보 검색.
- Tar 보관 형식 선택.
- 성능 향상을 포함한 Matrix4x4 숫자입니다.

자세한 내용은 .NET 11 라이브러리의 새로운 기능을 참조하세요.

.NET SDK

.NET 11 SDK에는 다음이 포함됩니다.

- 어셈블리 중복 제거를 통해 크기가 더 작은 SDK 설치 프로그램을 제공하는 Linux 및 macOS
- 노이즈가 줄어들고 진단 메시지가 더 명확해지는 CA1873 코드 분석기가 개선되었습니다.
- CA1515, CA1034 및 CA1859에 대한 분석기 버그 수정
- PackAsTool과 함께 사용되는 사용자 지정 `.nuspec` 파일에 대한 새 NETSDK1235 경고입니다.

자세한 내용은 .NET 11용 SDK의 새로운 기능입니다.

ASP.NET Core

ASP.NET Core의 새로운 기능과 관련하여 자세한 내용은 .NET 11용 ASP.NET Core의 새로운 기능입니다.

C# 15

C# 15에는 다음 기능이 포함됩니다.

- 컬렉션 식 인수

새로운 C# 기능에 대한 자세한 내용은 C# 15의 새로운 기능을 참조하세요.

파괴적 변경

.NET 11의 호환성이 손상되는 변경에 대한 자세한 내용은 .NET 11의 호환성이 손상되는 변경 내용을 참조하세요.

참고하십시오

- .NET 10의 새로운 기능

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

.NET 11 런타임의 새로운 기능

이 문서에서는 .NET 11용 .NET 런타임의 새로운 기능에 대해 설명합니다. 미리 보기 2에 대해 마지막으로 업데이트되었습니다.

업데이트된 최소 하드웨어 요구 사항

x86/x64 및 Arm64 아키텍처 모두에서 최신 명령 집합이 필요하도록 .NET 11에 대한 최소 하드웨어 요구 사항이 업데이트되었습니다. 또한 새로운 하드웨어 기능을 활용하도록 R2R(ReadyToRun) 컴파일 대상이 업데이트되었습니다.

Arm64 요구 사항

Apple의 경우 최소 하드웨어 또는 ReadyToRun 대상은 변경되지 않습니다. Apple M1 칩은 대략적으로 `armv8.5-a`에 동등하며, 적어도 `AdvSimd` (NEON), `CRC`, `DOTPROD`, `LSE`, `RCPC`, `RCPC2` 및 `RDMA` 명령 집합에 대한 지원을 제공합니다.

Linux의 경우 최소 하드웨어는 변경되지 않습니다. .NET은 명령 집합에 대한 지원만 제공할 수 있는 Raspberry Pi와 같은 디바이스를 `AdvSimd` 계속 지원합니다. ReadyToRun 대상이 명령 집합을 `LSE` 포함하도록 업데이트되어 애플리케이션을 시작할 때 추가적인 지팅 오버헤드가 발생할 수 있습니다.

Windows의 경우 `LSE` 지침 집합이 필요하도록 기준이 업데이트됩니다. 이는 Windows 11 및 Windows 10 에서 공식적으로 지원되는 모든 Arm64 CPU에 필요합니다. 또한 Arm SBSA(서버 기본 시스템 아키텍처) 요구 사항과 인라인으로 제공됩니다. ReadyToRun 대상이 `armv8.2-a + RCPC`으로 업데이트되었습니다. 이로 인해 이 대상은 최소한 `AdvSimd`, `CRC`, `LSE`, `RCPC` 및 `RDMA`을 지원하며, 공식적으로 지원되는 대부분의 하드웨어를 포괄합니다.

테이블 확장

OS	이전 JIT/AOT 최소값	새 JIT/AOT 최소값	이전 R2R 목표	새 R2R 대상
사과	Apple M1	(변경 내용 없음)	Apple M1	(변경 내용 없음)
리눅스	armv8.0-a	(변경 내용 없음)	armv8.0-a	armv8.0-a + LSE
윈도우즈	armv8.0-a	armv8.0-a + LSE	armv8.0-a	armv8.2-a + RCPC

x86/x64 요구 사항

세 운영 체제(Apple, Linux, Windows) 모두에서, 기준이 `x86-64-v1`에서 `x86-64-v2`으로 업데이트됩니다. 해당 변경은 하드웨어가 `CMOV`, `CX8`, `SSE`, `SSE2`을(를) 보장하는 것에서 `CX16`, `POPCNT`,

`SSE3`, `SSSE3`, `SSE4.1`, 및 `SSE4.2`도 보장하도록 변경합니다. 이 보장은 Windows 11 및 Windows 10에서 공식적으로 지원되는 모든 x86/x64 CPU에 필요합니다. 인텔과 AMD에 의해 여전히 공식적으로 지원되는 모든 칩이 포함되며, 마지막으로 오래된 칩들은 대략 2013년에 지원이 종료되었습니다.

ReadyToRun 대상은 Windows 및 Linux용으로 `x86-64-v3` 업데이트되었으며, 여기에는 , `AVX`, `AVX2`, `BMI1` `BMI2`, `F16C` `FMA` 및 `LZCNT` 명령 집합이 추가로 포함됩니다`MOVBE`. Apple의 ReadyToRun 대상은 변경되지 않습니다.

테이블 확장

OS	이전 JIT/AOT 최소값	새 JIT/AOT 최소값	이전 R2R 목표	새 R2R 대상
사과	x86-64-v1	x86-64-v2	x86-64-v2	(변경 내용 없음)
리눅스	x86-64-v1	x86-64-v2	x86-64-v2	x86-64-v3
윈도우즈	x86-64-v1	x86-64-v2	x86-64-v2	x86-64-v3

Impact

.NET 11부터 .NET은 이전 하드웨어에서 실행되지 않으며 다음과 유사한 메시지를 인쇄할 수 있습니다.

> 현재 CPU에 기준 명령 집합 중 하나 이상이 없습니다.

ReadyToRun 지원 어셈블리의 경우 지원되는 일부 하드웨어에서 일반적인 디바이스에 대한 예상 지원을 충족하지 않는 추가 시작 오버헤드가 있을 수 있습니다.

변경 이유

.NET은 기본 운영 체제에서 제공하는 최소 하드웨어 요구 사항을 초과하여 광범위한 하드웨어를 지원합니다. 이 지원은 코드베이스에 상당한 복잡성을 더합니다. 특히 여전히 사용 중일 가능성이 거의 없는 훨씬 오래된 하드웨어의 경우 특히 더 복잡합니다. 또한 AOT 대상이 기본값으로 지정해야 하는 "가장 낮은 공통 분모"를 정의하며, 일부 시나리오에서는 성능이 저하될 수 있습니다.

최소 기준에 대한 업데이트는 코드베이스의 유지 관리 복잡성을 줄이고 기본 운영 체제의 문서화된(종종 적용되는) 하드웨어 요구 사항에 더 잘 부합하도록 만들어졌습니다.

자세한 내용은 업데이트된 최소 하드웨어 요구 사항을 참조하세요.

런타임 비동기

.NET 11에는 컴파일러에서 생성된 비동기 상태 컴퓨터를 런타임 관리형 일시 중단 및 재개로 바꾸기 위한 중요한 단계인 런타임 네이티브 비동기(런타임 비동기 V2)가 도입되었습니다. 컴파일러가 상태-컴퓨터 클래스를 내보내는 대신 런타임 자체는 비동기 실행을 추적하여 더 깨끗한 스택 추적, 더 나은 디버깅 기능 및 낮은 오버헤드를 생성합니다.

런타임 비동기는 미리 보기 기능입니다. 옵트인하려면 프로젝트 파일에 다음 속성을 추가합니다.

XML

```xml
<PropertyGroup>
  <Features>runtime-async=on</Features>
  <EnablePreviewFeatures>true</EnablePreviewFeatures>
</PropertyGroup>
```

보다 간결한 실시간 스택 추적

가장 눈에 띄는 개선 사항은 실행 중 프로파일러, 디버거 및 `new StackTrace()`이 어떻게 보이는가에 있습니다. 컴파일러에서 생성된 비동기를 사용하면 각 비동기 메서드는 상태-컴퓨터 인프라에서 여러 프레임을 생성합니다. 런타임 비동기를 사용하면 실제 메서드가 호출 스택에 직접 표시됩니다.

C#

```csharp
// To enable runtime async, add the following to your .csproj:
//   <Features>runtime-async=on</Features>
//   <EnablePreviewFeatures>true</EnablePreviewFeatures>

await OuterAsync();

static async Task OuterAsync()
{
    await Task.CompletedTask;
    await MiddleAsync();
}

static async Task MiddleAsync()
{
    await Task.CompletedTask;
    await InnerAsync();
}

static async Task InnerAsync()
{
    await Task.CompletedTask;
```

```
    Console.WriteLine(new StackTrace(fNeedFileInfo: true));
}
```

-13프레임이 없으면 상태 머신 인프라가 표시됩니다.

text

```
   at Program.<<Main>$>g__InnerAsync|0_2() in Program.cs:line 24
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.Start[TStateMachine]
(...)
   at Program.<<Main>$>g__InnerAsync|0_2()
   at Program.<<Main>$>g__MiddleAsync|0_1() in Program.cs:line 14
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.Start[TStateMachine]
(...)
   at Program.<<Main>$>g__MiddleAsync|0_1()
   at Program.<<Main>$>g__OuterAsync|0_0() in Program.cs:line 8
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.Start[TStateMachine]
(...)
   at Program.<<Main>$>g__OuterAsync|0_0()
   at Program.<Main>$(String[] args) in Program.cs:line 3
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.Start[TStateMachine]
(...)
   at Program.<Main>$(String[] args)
   at Program.<Main>(String[] args)
```

`runtime-async` -5프레임의 경우 실제 호출 체인은 다음과 같습니다.

text

```
   at Program.<<Main>$>g__InnerAsync|0_2() in Program.cs:line 24
   at Program.<<Main>$>g__MiddleAsync|0_1() in Program.cs:line 14
   at Program.<<Main>$>g__OuterAsync|0_0() in Program.cs:line 8
   at Program.<Main>$(String[] args) in Program.cs:line 3
   at Program.<Main>(String[] args)
```

> ⓘ **참고 항목**
>
> 컴파일러에서 생성된 코드의 기존 `catch (Exception ex)` 정리가 해당 사례를 처리하므로 예외 스택 추적(원본 `ExceptionDispatchInfo`)은 런타임 비동기 사용 여부에 관계없이 이미 동일하게 표시됩니다. 라이브 실행 *중에* 볼 수 있는 내용이 개선되었습니다.

이 향상된 기능은 프로파일링 도구, 진단 로깅 및 디버거 호출 스택 창을 포함하여 라이브 실행 스택을 검사하는 모든 이점을 제공합니다.

디버깅 개선 사항

이를 통해 이제 중단점은 런타임 비동기 메서드 내에서 올바르게 바인딩되며, 디버거는 컴파일러 생성 인프라로 이동하지 않고도 `await` 경계를 따라서 단계별로 진행할 수 있습니다.

JIT 개선 사항

- **경계 확인 제거:** JIT(Just-In-Time) 컴파일러는 이제 인덱스와 상수가 같은 길이 `i + cns < len`와 비교되는 공통 패턴에 대한 경계 검사를 제거합니다. 이렇게 하면 타이트 루프에서 중복 검사가 감소하고 배열 및 범위 작업에 대한 처리량이 향상됩니다.
- **중복 확인된 컨텍스트 제거:** 이제 JIT는 중복된 확인된 산술 컨텍스트(예: 값이 이미 범위에 있는 것으로 알려진 경우)를 증명하고 제거할 수 있습니다. 이 최적화는 생성된 코드에서 불필요한 오버플로 검사를 제거합니다.
- **ReadyToRun 이미지의 비정상화:** 이제 R2R(ReadyToRun) 이미지는 공유되지 않는 제네릭 가상 메서드 호출을 역가상화하여 제네릭 시나리오에 대해 미리 컴파일된 코드의 성능을 향상시킬 수 있습니다.
- **SVE2 내장 함수:** 새 Arm SVE2(확장 가능한 벡터 확장 2) 내장 함수를 사용할 수 있습니다 `ShiftRightLogicalNarrowingSaturate(Even|Odd)`. 그러면 SVE2를 지원하는 Arm 하드웨어에서 사용할 수 있는 벡터화된 작업 집합이 확장됩니다.

VM 개선 사항

- **비 JIT 플랫폼에서 캐시된 인터페이스 디스패치:** iOS와 같은 JIT 지원이 부족한 플랫폼에서 인터페이스 디스패치는 비용이 많이 드는 일반 수정 경로로 대체되었습니다. 캐시된 디스패치는 이러한 대상에서 인터페이스 집중 코드의 성능을 최대 200배까지 향상시킵니다.
- **`Guid.NewGuid()` Linux:**Guid.NewGuid() 이제 Linux에서는 `getrandom()` syscall과 일괄 처리 캐싱을 사용하여 `/dev/urandom`에서 읽는 대신 GUID 생성 시 약 12%의 처리량 향상을 제공합니다.

참고하십시오

- .NET 11용 .NET 라이브러리의 새로운 기능
- .NET 11용 SDK의 새로운 기능
- .NET 11의 주요 변경 내용

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

.NET 11용 .NET 라이브러리의 새로운 기능

이 문서에서는 .NET 11용 .NET 라이브러리의 새로운 기능에 대해 설명합니다. 미리 보기 2에 대해 마지막으로 업데이트되었습니다.

문자열 및 문자 기능 향상

.NET 11에서는 문자열 및 문자 조작 API가 크게 향상되어 유니코드 문자와 룬을 더 쉽게 사용할 수 있습니다.

String 메서드의 Rune 지원

이제 클래스에는 String 매개 변수를 허용하는 Rune 메서드가 포함되어 유니코드 스칼라 값을 사용하여 문자열을 직접 검색, 바꾸기 및 조작할 수 있습니다. 이러한 새로운 메서드는 다음과 같습니다.

- `Contains` - 문자열에 특정 rune이 포함되어 있는지 확인합니다 String.Contains(Rune)String.Contains(Rune, StringComparison).
- `StartsWith` 및 `EndsWith` - 문자열이 시작되거나 특정 rune로 끝나는지 확인합니다 String.StartsWith(Rune)String.StartsWith(Rune, StringComparison)String.EndsWith(Rune)String.EndsWith(Rune, StringComparison).
- `IndexOf` 및 `LastIndexOf` - 문자열String.IndexOf(Rune)String.IndexOf(Rune, StringComparison)String.LastIndexOf(Rune)String.LastIndexOf(Rune, StringComparison)에서 rune의 위치를 찾습니다.
- `Replace` - 하나의 룬을 다른 룬으로 교체합니다: String.Replace(Rune, Rune).
- `Split` - 룬을 구분 기호로 사용하여 문자열을 분할합니다 String.Split(Rune, StringSplitOptions)String.Split(Rune, Int32, StringSplitOptions).
- `Trim`, `TrimStart` 및 `TrimEnd` - 문자열에서 룬 제거하기: String.Trim(Rune), String.TrimStart(Rune) 및 String.TrimEnd(Rune).

이러한 메서드의 대부분은 문화권 인식 비교를 위한 매개 변수를 StringComparison 허용하는 오버로드를 포함합니다.

StringComparison과 Char.Equals

이제 Char 구조체에 Char.Equals(Char, StringComparison) 매개 변수를 허용하는 StringComparison 메서드가 포함되어 있어, 문화권 인식 또는 서수 비교를 통해 문자를 비교할 수 있습니다.

TextInfo의 Rune 지원

이제 TextInfo 클래스는 TextInfo.ToLower(Rune) 및 TextInfo.ToUpper(Rune) 메서드를 제공하며, 이는 Rune 매개 변수를 수용하여 개별 유니코드 스칼라 값에 대한 대/소문자 변환을 수행할 수 있게 합니다.

Base64 인코딩 개선 사항

.NET 11은 기존 형식에 새 API 및 오버로드를 Base64 추가하여 Base64 인코딩 및 디코딩을 포괄적으로 지원합니다. 이러한 추가 기능은 기존 메서드에 비해 향상된 성능과 유연성을 제공합니다.

새 Base64 API

새 API는 다양한 입력 및 출력 형식으로 인코딩 및 디코딩 작업을 지원합니다.

- **문자로 인코딩**: EncodeToChars 및 EncodeToString
- **UTF-8로 인코딩**: EncodeToUtf8
- **문자에서 디코딩**: DecodeFromChars
- **UTF-8에서 디코딩**: DecodeFromUtf8

이러한 메서드는 높은 수준의 편리한 메서드(배열 또는 문자열을 할당 및 반환)와 하위 수준 범위 기반 메서드(할당 0 시나리오의 경우)를 모두 제공합니다.

압축 성능 향상

.NET 11에는 압축 API에 대한 몇 가지 개선 사항이 포함되어 있습니다.

ZIP 보관 항목 액세스 모드

이제 클래스는 ZipArchiveEntry 새 오버로드를 통해 특정 파일 액세스 모드로 항목 열기를 지원합니다. ZipArchiveEntry.Open(FileAccess)ZipArchiveEntry.OpenAsync(FileAccess, CancellationToken) 이러한 오버로드는 FileAccess 매개 변수를 받아들이며, 읽기 전용, 쓰기 전용 또는 읽기/쓰기가 가능한 형태로 ZIP 항목을 열 수 있습니다.

또한 새 CompressionMethod 속성을 사용하면 ZipCompressionMethod 열거형을 통해 사용되는 압축 메서드를 제공합니다. 여기에는 `Stored`, `Deflate`, `Deflate64`의 값이 포함됩니다.

DeflateStream 및 GZipStream 동작 변경

.NET 11부터 DeflateStream 및 GZipStream는 데이터가 기록되지 않더라도 항상 형식 헤더와 바닥글을 출력 스트림에 씁니다. 이렇게 하면 Deflate 및 GZip 사양에 따라 출력이 유효한 압축 스트림이 됩니다.

이전에는 데이터가 기록되지 않은 경우 이러한 스트림이 출력을 생성하지 않아 빈 출력 스트림이 생성되었습니다. 이렇게 변경하면 적절한 형식의 압축 스트림이 필요한 도구와의 호환성이 보장됩니다.

자세한 내용은 DeflateStream 및 GZipStream이 비어 있는 페이로드에 대해 헤더 및 바닥글을 쓰는 방법을 참고하세요.

BitConverter의 BFloat16 지원

이제 클래스에는 BitConverter 값과 바이트 배열 또는 비트 표현 간에 BFloat16 변환하는 메서드가 포함됩니다. 이러한 새로운 메서드는 다음과 같습니다.

- BitConverter.GetBytes(BFloat16) - BFloat16 값을 바이트 배열로 변환합니다.
- BitConverter.ToBFloat16(Byte[], Int32) and BitConverter.ToBFloat16(ReadOnlySpan<Byte>) - 바이트 배열을 BFloat16 값으로 변환합니다.
- BitConverter.BFloat16ToInt16Bits(BFloat16), BitConverter.BFloat16ToUInt16Bits(BFloat16), BitConverter.Int16BitsToBFloat16(Int16) 및 BitConverter.UInt16BitsToBFloat16(UInt16) - BFloat16과 해당 비트 표현으로서의 `short` 또는 `ushort`로 변환하는 방법들.

BFloat16(브레인 부동 소수점)은 기계 학습 및 과학 컴퓨팅에 일반적으로 사용되는 16비트 부동 소수점 형식입니다.

컬렉션 개선 사항

BitArray.PopCount

이제 클래스에는 BitArray 배열에 BitArray.PopCount() 설정된 비트 수를 반환하는 `true` 메서드가 포함됩니다. 이렇게 하면 배열을 수동으로 반복하지 않고 집합 비트를 계산하는 효율적인 방법을 제공합니다.

JSON serialization에서 IReadOnlySet 지원

이제 JsonMetadataServices 클래스에는 JsonMetadataServices.CreateIReadOnlySetInfo 컬렉션에 대한 JSON serialization 지원을 가능하게 하는 IReadOnlySet<T> 메서드가 포함됩니다.

URI 데이터 구성표 상수

URI 구성표를 나타내는 새 Uri.UriSchemeData 상수가 `data:` 추가되었습니다. 이 상수는 데이터 URI를 참조하는 표준화된 방법을 제공합니다.

StringSyntax 특성 향상

이제 클래스에는 StringSyntaxAttribute 일반적인 프로그래밍 언어에 대한 상수가 포함됩니다.

- `CSharp` - C# 구문을 나타냅니다.
- `FSharp` - F# 구문을 나타냅니다.
- `VisualBasic` - Visual Basic 구문을 나타냅니다.

이러한 상수는 특성과 함께 `StringSyntax` 사용하여 이러한 언어의 코드를 포함하는 문자열 리터럴에 대한 더 나은 도구 지원을 제공할 수 있습니다.

System.Text.Json 개선 사항

제네릭 형식 정보 검색

형식 메타데이터를 `System.Text.Json` 사용할 때 일반적인 패턴은 JsonSerializerOptions에서 JsonTypeInfo<T>를 검색하는 것입니다. 이전에는 제네릭 GetTypeInfo(Type) 이 아닌 메서드에서 수동으로 다운캐스트해야 했습니다. 새로운 제네릭 `System.Text.Json.JsonSerializerOptions.GetTypeInfo<T>` 및 `TryGetTypeInfo<T>()` 메서드는 강력하게 형식화된 메타데이터를 직접 반환하므로 캐스트할 필요가 없어집니다.

C#

```
JsonSerializerOptions options = new(JsonSerializerDefaults.Web);
options.MakeReadOnly();

// Before: manual downcast required
JsonTypeInfo<MyRecord> info1 =
(JsonTypeInfo<MyRecord>)options.GetTypeInfo(typeof(MyRecord));

// After: generic method returns the right type directly
JsonTypeInfo<MyRecord> info2 = options.GetTypeInfo<MyRecord>();

// TryGetTypeInfo variant for cases where the type may not be registered
if (options.TryGetTypeInfo<MyRecord>(out JsonTypeInfo<MyRecord>? typeInfo))
{
    // Use typeInfo
    _ = typeInfo;
}
```

형식 메타데이터 액세스가 흔한 소스 생성, NativeAOT 및 다형적 직렬화 시나리오에서 사용할 때 특히 유용합니다.

Tar 보관 형식 선택

새 오버로드는 CreateFromDirectory 및 CreateFromDirectoryAsync에서 TarEntryFormat 매개 변수를 받아 보관 형식을 직접 제어할 수 있게 합니다. `CreateFromDirectory` 이전에는 항상 Pax 아카이브를 생성했습니다. 새 오버로드는 특정 도구 및 환경과의 호환성을 위해 Pax, Ustar, GNU 및 V7의 네 가지 tar 형식을 모두 지원합니다.

C#

```
// Create a GNU format tar archive for Linux compatibility
TarFile.CreateFromDirectory("/source/dir", "/dest/archive.tar",
    includeBaseDirectory: true, format: TarEntryFormat.Gnu);

// Create a Ustar format archive for broader compatibility
using Stream outputStream = File.OpenWrite("/dest/ustar.tar");
TarFile.CreateFromDirectory("/source/dir", outputStream,
    includeBaseDirectory: false, format: TarEntryFormat.Ustar);

// Async version
CancellationToken cancellationToken = CancellationToken.None;
await TarFile.CreateFromDirectoryAsync("/source/dir", "/dest/archive.tar",
    includeBaseDirectory: true, format: TarEntryFormat.Pax,
    cancellationToken: cancellationToken);
```

숫자 개선 사항

Matrix4x4 성능

Matrix4x4.GetDeterminant() 이제 SSE 벡터화된 구현을 사용하여 성능을 약 15%향상합니다.

참고하십시오

- .NET 11 런타임의 새로운 기능
- .NET 11용 SDK의 새로운 기능
- .NET 11의 주요 변경 내용

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

.NET 11용 SDK 및 도구의 새로운 기능

이 문서에서는 .NET 11용 .NET SDK의 새로운 기능과 향상된 기능을 설명합니다. 미리 보기 2에 대해 마지막으로 업데이트되었습니다.

Linux 및 macOS의 소규모 SDK 설치 프로그램

심볼릭 링크를 사용하여 어셈블리를 중복 제거함으로써 Linux 및 macOS에서 .NET SDK 설치 관리자 크기가 감소되었습니다. 중복 `.dll` 및 `.exe` 파일은 콘텐츠 해시로 식별되고 단일 복사본을 가리키는 기호 링크로 대체됩니다. 이는 tarball, `.pkg`, `.deb` 및 `.rpm` 설치 관리자에 영향을 줍니다.

분석에 따르면 SDK 디렉터리의% 35개는 중복 파일로 구성됩니다. Linux x64에서는 총 816개의 파일이 디스크에서 140MB(압축된 53MB)입니다. 중복 항목을 기호 링크로 대체하면 Linux x64 보관 파일의 크기가 크게 줄어듭니다.

테이블 확장

플랫폼	SDK 아티팩트	.NET 10 크기(MB)	.NET 11 미리 보기 2 크기(MB)	축소
linux-x64	타르볼	230	189	17.8%
linux-x64	뎁	164	122	25.6%
linux-x64	Rpm	165	122	26.0%
linux-x64	컨테이너	다릅니다	다릅니다	8-17%

Windows 중복 제거 기능은 향후 예정된 공개 프리뷰에서 선보일 예정입니다.

코드 분석기 개선 사항

CA1873: 노이즈 감소 및 메시지 개선

CA1873의 두 가지 개선 사항(잠재적으로 비용이 많이 드는 로깅 방지)

오탐 감소: 속성 액세스, `GetType()`, `GetHashCode()` 및 `GetTimestamp()` 호출이 더 이상 플래그되지 않습니다. 경고, 오류 및 중요 코드 경로는 거의 핫 경로가 없기 때문에 진단은 이제 정보 수준 로깅 및 아래의 기본값에만 적용됩니다.

진단 메시지의 특정 이유: 이제 진단 메시지에 인수 플래그가 지정된 이유가 포함되어 다음을 해결할 경고의 우선 순위를 지정하는 데 도움이 됩니다.

text

```
// Before
warning CA1873: Evaluation of this argument may be expensive and unnecessary if
logging is disabled

// After
warning CA1873: Evaluation of this argument may be expensive and unnecessary if
logging is disabled (method invocation)
```

9가지 구체적인 이유는 다음과 같습니다.

- 메서드 호출
- 개체 만들기
- 배열 생성
- Boxing 변환
- 문자열 보간
- 컬렉션 식
- 익명 개체 만들기
- Await 식
- 표현과 함께

분석기 버그 수정

테이블 확장

Analyzer	수정
CA1515	C# 확장 멤버가 있을 때 오탐 문제 수정
CA1034	C# 확장 멤버가 있는 경우 오탐 수정됨
CA1859	기본 인터페이스 구현의 부적절한 처리가 수정되었습니다.

.NET 11에 대해 AnalysisLevel 수정됨

`AnalysisLevel=latest` 예상된 .NET 11 규칙 대신 .NET 9 분석기 규칙을 잘못 사용하는 프로젝트가 있었습니다. 이제 이 문제는 해결되었습니다.

새 SDK 경고

NETSDK1235: PackAsTool을 사용한 사용자 지정 .nuspec

프로젝트가 사용자 지정 `PackAsTool=true` 속성을 설정하고 `NuspecFile` 지정하면 새 경고가 내보내집니다. 도구 패키지에는 사용자 지정 `.nuspec` 파일이 일반적으로 위반하는 특정 레이아웃 및 식별자 규칙이 필요합니다.

```text
warning NETSDK1235: .NET Tools do not support using a custom .nuspec file, but the nuspec file 'custom.nuspec' was provided. Remove the NuspecFile property from this project to enable packing it as a .NET Tool.
```

팩 작업은 기존 프로젝트를 중단하지 않도록 경고와 함께 계속 진행됩니다.

파괴적 변경

.NET 11에는 SDK에서 다음과 같은 호환성이 손상되는 변경이 포함됩니다.

Mono 시작 대상이 더 이상 자동으로 설정되지 않음

.NET 11부터 .NET SDK는 더 이상 Linux에서 .NET Framework 앱의 시작 대상으로 자동으로 설정 `mono` 되지 않습니다. 실행을 위해 Mono를 사용하는 경우 시작 구성을 업데이트하여 `mono`을(를) 명시적으로 지정합니다.

자세한 내용은 Mono 시작 대상이 더 이상 자동으로 설정되지 않음을 참조하세요.

참고하십시오

- .NET 11 런타임의 새로운 기능
- .NET 11용 .NET 라이브러리의 새로운 기능
- .NET 11의 주요 변경 내용

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 12.

.NET 11의 주요 변경 내용

앱을 .NET 11로 마이그레이션하는 경우 여기에 나열된 주요 변경 내용이 영향을 줄 수 있습니다. 변경 내용은 ASP.NET Core 또는 Windows Forms 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경을 *이진 파일 비호환*, *원본 비호환* 또는 *동작 변경*으로 분류합니다.

- **이진 파일 비호환** - 새 런타임이나 구성 요소에 대해 실행할 때 기존 이진 파일의 동작이 크게 변경될 수 있습니다(예: 로드 또는 실행 실패). 그런 경우 다시 컴파일이 필요합니다.
- **원본 비호환** - 새 SDK 또는 구성 요소를 사용하여 다시 컴파일하거나 새 런타임을 대상으로 하는 경우 기존 소스 코드를 성공적으로 컴파일하려면 원본을 변경해야 할 수도 있습니다.
- **동작 변경** - 기존 코드 및 이진 파일은 런타임에 다르게 동작할 수 있습니다. 새 동작이 바람직하지 않은 경우 기존 코드를 업데이트하고 다시 컴파일해야 합니다.

> **참고 항목**
>
> 이 문서는 진행 중인 작업입니다. .NET 11의 주요 변경 내용의 전체 목록은 아닙니다.

핵심 .NET 라이브러리

테이블 확장

제목	변경 유형
DateOnly 및 TimeOnly TryParse 메서드가 잘못된 입력을 위해 throw됨	동작 변경
DeflateStream 및 GZipStream은 빈 페이로드에 대한 머리글 및 바닥글을 작성합니다.	동작 변경
Environment.TickCount는 Windows 시간 제한 동작과 일치합니다	동작 변경
MemoryStream 최대 용량 업데이트 및 예외 동작 변경	동작 변경
기본 설정이 아닌 진단 ID를 포함한 API 폐기(.NET 11)	원본이 호환되지 않음
TAR 읽기 API는 읽을 때 헤더 체크섬을 확인합니다.	동작 변경

제목	변경 유형
ZipArchive.CreateAsync에서 ZIP 보관 항목을 열심히 로드합니다.	동작 변경

Cryptography

테이블 확장

제목	변경 유형
macOS에서 DSA가 제거됨	동작 변경

Extensions

테이블 확장

제목	변경 유형
BackgroundService가 실패할 때 IHost.RunAsync 및 IHost.StopAsync throw	동작 변경

Globalization

테이블 확장

제목	변경 유형
일본어 일정 최소 지원 날짜 수정	동작 변경

Interop

테이블 확장

제목	변경 유형
NativeAOT는 Unix의 네이티브 라이브러리 출력에 lib 접두사를 사용합니다.	동작 변경

JIT 컴파일러

테이블 확장

제목	변경 유형
업데이트된 최소 하드웨어 요구 사항	동작 변경

SDK 및 MSBuild

테이블 확장

제목	변경 유형
모노 시작 대상은 .NET Framework 앱에 대해 설정되지 않았습니다	동작 변경

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 04. 08.

.NET 10의 새로운 기능

.NET 10의 새로운 기능에 대해 알아보고 추가 설명서에 대한 링크를 찾습니다.

.NET 9 후속 버전인 .NET 10은 LTS(장기 지원) 릴리스로 3년 동안 지원됩니다. .NET 10을 다운로드할 수 있습니다.

여러분의 피드백은 중요하고 감사하게 생각합니다. 질문이나 의견이 있는 경우 GitHub 대한 토론을 사용하세요.

.NET 런타임

.NET 10 런타임에서는 JIT 인라인 처리, 메서드 비정상화 및 스택 할당이 개선되었습니다. 또한 AVX10.2 지원, NativeAOT 향상된 기능, 구조체 인수에 대한 향상된 코드 생성 및 향상된 최적화를 위한 향상된 루프 반전이 포함됩니다.

자세한 내용은 .NET 10 런타임 새로운 기능참조하세요.

.NET 라이브러리

.NET 10 라이브러리는 암호화, 세계화, 숫자, serialization, 컬렉션 및 진단 및 ZIP 파일 작업 시 새로운 API를 도입합니다. 새로운 JSON 직렬화 옵션에는 중복 속성 허용 불가, 엄격한 serialization 설정 및 향상된 효율성 지원 등이 `PipeReader` 있습니다. Windows Cryptography API: 차세대(CNG) 지원, 간소화된 API 및 HashML-DSA 지원을 통해 향상된 ML-DSA 및 복합 ML-DSA를 사용하여 양자 후 암호화 지원이 확장되었습니다. 추가 암호화 기능 향상에는 패딩 지원이 포함된 AES KeyWrap이 포함됩니다. 새로운 네트워킹 기능에는 간소화된 `WebSocketStream` 사용량 및 macOS 클라이언트에 대한 TLS 1.3 지원이 포함 `WebSocket` 됩니다. 프로세스 관리를 통해 더 나은 신호 격리를 위한 Windows 프로세스 그룹 지원이 향상됩니다.

자세한 내용은 .NET 10 라이브러리 새로운 기능을 참조하세요.

.NET SDK

.NET 10 SDK는 Microsoft.Testing.Platform 에 `dotnet test`대한 지원을 포함하고 CLI 명령 순서를 표준화하며 인기 있는 셸에 대한 네이티브 탭 완성 스크립트를 생성하도록 CLI를 업데이트합니다. 컨테이너의 경우 콘솔 앱은 기본적으로 컨테이너 이미지를 만들 수 있으며, 새 속성을 사용하면 컨테이너 이미지의 형식을 명시적으로 설정할 수 있습니다. 또한 SDK는 RuntimeIdentifier를 통한 `any` 향상된 호환성, 단발 도구 실행, 새 `dotnet tool exec` 도구 실행 `dnx`스크립트, CLI를 사용한 검색 `--cli-schema`및 게시 지원 및 네이티브 AOT를 사용하는 향상된 파일 기반 앱을 통해 플랫폼별 .NET 도구를 지원합니다.

자세한 내용은 .NET 10용 SDK의 새로운 기능참조하세요.

갈망

Aspire의 새로운 기능에 대한 정보는 Aspire — what's new?를 참조하세요.

ASP.NET Core

ASP.NET Core 10.0 릴리스에는 Blazor 개선 사항, OpenAPI 향상된 기능 및 최소 API 업데이트를 비롯한 몇 가지 새로운 기능과 향상된 기능이 도입되었습니다. 기능에는 Blazor WebAssembly 미리 로드, 자동 메모리 풀 제거, 향상된 양식 유효성 검사, 향상된 진단 및 ID에 대한 암호 지원이 포함됩니다.

자세한 내용은 .NET 10용 ASP.NET Core의 새로운 기능입니다.

C# 14

C# 14에는 개발자 생산성 및 코드 품질을 개선하기 위한 몇 가지 새로운 기능과 향상된 기능이 도입되었습니다. 주요 업데이트는 다음과 같습니다.

- 필드 지원 속성은 자동 구현 속성에서 사용자 지정 `get` 및 `set` 접근자 작성에 이르기까지 더 원활한 경로를 제공합니다. 상황에 맞는 키워드를 사용하여 `field` 컴파일러에서 생성된 백업 필드에 액세스할 수 있습니다.
- 이제 식은 `nameof` 형식 인수 없이 형식의 이름을 반환하는 것과 같은 `List<>`바인딩되지 않은 제네릭 형식을 지원합니다.
- 암시적 변환 `Span<T>` 및 `ReadOnlySpan<T>`에 대한 일급 지원.
- 람다 식에서 매개 변수 형식을 지정하지 않고도 `ref`, `in`, `out`와 같은 매개 변수 한정자를 사용할 수 있습니다.
- 부분 인스턴스 생성자 및 부분 이벤트를 지원하여 C# 13에 도입된 부분 메서드 및 속성을 보완합니다.
- 새 `extension` 블록은 정적 확장 메서드와 정적 및 인스턴스 확장 속성에 대한 지원을 추가합니다.
- Null 조건부 할당은 `?.` 연산자를 사용합니다.
- 사용자 정의 복합 할당 연산자(예: `+=` 및 `-=`.
- 사용자 정의 증가(`++`) 및 감소(`--`) 연산자입니다.

C# 14의 새로운 기능 에 대한 자세한 내용은을 참조하세요.

F#

.NET 10의 F# 업데이트에는 언어, 표준 라이브러리 및 컴파일러 서비스에 대한 몇 가지 새로운 기능과 향상된 기능이 포함되어 있습니다. 주요 업데이트는 다음과 같습니다.

- **F# 언어**:

  새 언어 기능을 사용하려면 `<LangVersion>preview</LangVersion>` 파일에서 `.fsproj` 프로젝트 속성을 사용하도록 설정해야 합니다. 이러한 기능은 .NET 10 릴리스의 기본값이 됩니다.

- **FSharp.Core 표준 라이브러리**:

  `FSharp.Core` 표준 라이브러리에 대한 변경 내용은 하위 `FSharp.Core` 버전이 명시적으로 고정되지 않는 한 새 SDK로 컴파일된 프로젝트에 자동으로 적용됩니다.

- **FSharp.Compiler.Service**:

  컴파일러 구현의 일반적인 개선 사항 및 버그 수정.

자세한 내용은 F# 10의 새로운 기능 또는 F# 릴리스 정보를 참조하세요.

Visual Basic (비주얼 베이직 언어)

.NET 10의 Visual Basic 업데이트에는 컴파일러에 대한 다음과 같은 향상된 기능이 포함되어 있습니다.

- 이제 컴파일러는 제네릭 제약 조건을 해석하고 적용 `unmanaged` 하므로 런타임 API와의 호환성이 향상됩니다.
- 컴파일러는 OverloadResolutionPriorityAttribute을/를 준수합니다. 이 개선은 범위 기반 오버로드가 더 빠르게 선호되고 오버로드 모호성을 해결하는 데 도움이 됩니다.

이러한 업데이트를 통해 Visual Basic은 C# 및 런타임에서 업데이트된 기능을 사용할 수 있습니다. 추가 정보는 Visual Basic의 새로운 기능을 참조하세요.

.NET 마우이

.NET 10의 .NET MAUI 업데이트에는 .NET MAUI, Android용 .NET 및 iOS, Mac Catalyst, macOS 및 tvOS용 .NET에 대한 몇 가지 새로운 기능 및 품질 향상이 포함됩니다. 기능에는 여러 파일 및 이미지 압축을 선택하기 위한 MediaPicker 개선 사항, WebView 요청 가로채기 및 Android API 수준 35 및 36에 대한 지원이 포함됩니다.

자세한 내용은 .NET 10의 .NET MAUI의 새로운 기능입니다.

EF Core

EF Core 10 릴리스에는 LINQ 개선 사항, 성능 최적화, Azure Cosmos DB에 대한 향상된 지원 및 선택적 비활성화를 사용하여 엔터티 유형당 여러 필터를 허용하는 명명된 쿼리 필터를 비롯한 몇 가지 새로운 기능과 개선 사항이 도입되었습니다.

자세한 내용은 .NET 10용 EF Core의 새로운 기능입니다.

윈도우 폼즈 (Windows Forms)

.NET 10용 Windows Forms의 변경 내용에는 .NET Framework에서 이식된 `UITypeEditors` 클립보드 관련 업데이트 및 품질 향상이 포함됩니다.

자세한 내용은 .NET 10용 Windows Forms의 새로운 기능입니다.

WPF(Windows Presentation Foundation)

.NET 10의 WPF 업데이트에는 몇 가지 성능 향상, Fluent 스타일 변경, 버그 수정 등이 포함됩니다.

자세한 내용은 .NET 10의 WPF의 새로운 기능입니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 11. 07.

.NET 10 런타임의 새로운 기능

이 문서에서는 .NET 10용 .NET 런타임의 새로운 기능 및 성능 향상에 대해 설명합니다.

JIT 컴파일러 개선 사항

.NET 10의 JIT 컴파일러에는 더 나은 코드 생성 및 최적화 전략을 통해 성능을 향상시키는 향상된 기능이 포함되어 있습니다.

구조체 인수에 대한 코드 생성 개선

. NET의 JIT 컴파일러는 구조체의 멤버가 스택이 아닌 레지스터에 배치되어 메모리 액세스가 제거되는 물리적 승격이라는 최적화를 수행할 수 있습니다. 이 최적화는 메서드에 구조체를 전달할 때 특히 유용하며 호출 규칙에 따라 구조체 멤버를 레지스터에 전달해야 합니다.

.NET 10은 레지스터를 공유하는 값을 처리하도록 JIT 컴파일러의 내부 표현을 향상시킵니다. 이전에는 구조체 멤버를 단일 레지스터로 압축해야 하는 경우 JIT는 먼저 메모리에 값을 저장한 다음 레지스터에 로드합니다. 이제 JIT 컴파일러는 구조체 인수의 승격된 멤버를 공유 레지스터에 직접 배치하여 불필요한 메모리 작업을 제거할 수 있습니다.

다음 예제를 고려하세요.

C#

```
struct Point
{
    public long X;
    public long Y;

    public Point(long x, long y)
    {
        X = x;
        Y = y;
    }
}

[MethodImpl(MethodImplOptions.NoInlining)]
private static void Consume(Point p)
{
    Console.WriteLine(p.X + p.Y);
}

private static void Main()
{
    Point p = new Point(10, 20);
    Consume(p);
}
```

x64에서 멤버 `Point` 는 별도의 레지스터에 `Consume` 전달되며 로컬에 대한 `p`물리적 승격이 시작되었으므로 먼저 스택에 할당된 항목이 없습니다.

asm

```
Program:Main() (FullOpts):
       mov      edi, 10
       mov      esi, 20
       tail.jmp [Program:Consume(Program+Point)]
```

이제 멤버 `Point` 의 형식이 대신 .로 변경되었다고 가정합니다 `int` `long`. `int` 너비는 4바이트이고 레지스터는 x64의 너비가 8바이트이므로 호출 규칙을 사용하려면 멤버 `Point` 를 하나의 레지스터로 전달해야 합니다. 이전에는 JIT 컴파일러가 먼저 값을 메모리에 저장한 다음 8 바이트 청크를 레지스터에 로드했습니다. .NET 10 개선 사항으로 JIT 컴파일러는 이제 구조체 인수의 승격된 멤버를 공유 레지스터에 직접 배치할 수 있습니다.

asm

```
Program:Main() (FullOpts):
       mov      rdi, 0x140000000A
       tail.jmp [Program:Consume(Program+Point)]
```

이렇게 하면 중간 메모리 스토리지가 필요하지 않고 어셈블리 코드가 더 효율적입니다.

향상된 루프 반전

JIT 컴파일러는 루프의 `while` 조건을 끌어와 루프 본문을 루프로 `do-while` 변환하여 최종 셰이프를 생성할 수 있습니다.

C#

```
if (loopCondition)
{
    do
    {
        // loop body
    } while (loopCondition);
}
```

이 변환을 루프 반전이라고 합니다. JIT는 조건을 루프의 맨 아래로 이동하여 조건을 테스트하기 위해 루프의 맨 위로 분기할 필요가 없도록 하여 코드 레이아웃을 개선합니다. 루프 복제, 루프 언롤링 및 유도 변수 최적화와 같은 수많은 최적화도 루프 반전에 의존하여 이 모양을 생성하여 분석을 지원합니다.

.NET 10은 어휘 분석 구현에서 그래프 기반 루프 인식 구현으로 전환하여 루프 반전을 향상시킵니다. 이 변경은 모든 자연 루프(단일 진입점이 있는 루프)를 고려하고 이전에 고려했던 가양성 값을 무시하여 정밀도를 향상합니다. 이렇게 하면 `for` 및 `while` 명령문을 사용하여 .NET 프로그램의 최적화 잠재력을 높일 수 있습니다.

배열 인터페이스 메서드 비가상화

.NET 10에 포커스 영역 중 하나는 인기 있는 언어 기능의 추상화 오버헤드를 줄이는 것입니다. 이 목표를 달성하기 위해 배열 인터페이스 메서드를 포함하도록 JIT의 메서드 호출을 비가상화하는 기능이 확장되었습니다.

배열을 순회하는 일반적으로 활용되는 방법을 고려합니다.

C#

```
static int Sum(int[] array)
{
    int sum = 0;
    for (int i = 0; i < array.Length; i++)
    {
        sum += array[i];
    }
    return sum;
}
```

이 코드 셰이프는 주로 추론할 가상 호출이 없기 때문에 JIT를 쉽게 최적화할 수 있습니다. 대신 JIT는 배열 액세스에 대한 경계 검사를 제거하고 .NET 9 추가된루프 최적화를 적용하는 데 집중할 수 있습니다. 다음 예제에서는 몇 가지 가상 호출을 추가합니다.

C#

```
static int Sum(int[] array)
{
    int sum = 0;
    IEnumerable<int> temp = array;

    foreach (var num in temp)
    {
        sum += num;
    }
    return sum;
}
```

기본 컬렉션의 형식은 명확하며 JIT는 이 코드 조각을 첫 번째 컬렉션으로 변환할 수 있어야 합니다. 그러나 배열 인터페이스는 "일반" 인터페이스와 다르게 구현되기 때문에 JIT는 이를 비상화하는 방법을 모릅니다. 즉, `foreach` 루프의 열거자 호출은 가상으로 유지되어 인라인 및 스택 할당과 같은 여러 최적화를 차단합니다.

.NET 10부터 JIT는 배열 인터페이스 메서드를 비가상화하고 인라인화할 수 있습니다. .NET 10 추상화 해제 계획 자세히 설명한 대로 구현 간의 성능 패리티를 달성하기 위한 여러 단계 중 첫 번째 단계입니다.

배열 열거형 추상화 해제

열거자를 통해 배열 반복의 추상화 오버헤드를 줄이려는 노력으로 JIT의 인라인 처리, 스택 할당 및 루프 복제 기능이 향상되었습니다. 예를 들어 `IEnumerable` 통해 배열을 열거하는 오버헤드가 줄어들고 조건부 이스케이프 분석을 통해 특정 시나리오에서 열거자를 스택 할당할 수 있습니다.

향상된 코드 레이아웃

.NET 10의 JIT 컴파일러는 런타임 성능을 향상하기 위해 메서드 코드를 기본 블록으로 구성하는 새로운 접근 방식을 도입했습니다. 이전에는 JIT에서 프로그램 흐름 그래프의 역방향 포스트오더 순회를 초기 레이아웃으로 사용한 후 반복 변환을 수행했습니다. 유효하지만 이 방법은 분기를 줄이고 핫 코드 밀도를 높이는 것 사이의 장차를 모델링하는 데 제한이 있었습니다.

.NET 10에서는 JIT가 블록 재배치 문제를 비대칭 여행 세일즈맨 문제의 축소로 모형화하고, 3-opt 휴리스틱을 구현하여 거의 최적에 가까운 순회를 찾아냅니다. 이 최적화는 핫 경로 밀도를 향상시키고 분기 거리를 줄여 런타임 성능을 향상시킵니다.

인라인 최적화

.NET 10에서는 다양한 인라인 개선이 이루어졌습니다.

이제 JIT는 이전 인라인으로 인해 비가상화가 가능한 메서드를 인라인할 수 있습니다. 이러한 향상된 기능을 통해 JIT는 추가 인라인화 및 비가상화와 같은 더 많은 최적화 기회를 발견할 수 있습니다.

예외 처리 의미 체계가 있는 일부 메서드, 특히 블록이 있는 `try-finally` 메서드도 인라인 처리할 수 있습니다.

JIT의 일부 배열을 스택 할당하는 기능을 더 잘 활용하기 위해 인라이너의 추론이 조정되어 작은 고정 크기 배열을 반환할 수 있는 후보의 수익성을 높일 수 있습니다.

반환 형식

인라인 처리 중에 JIT는 이제 반환 값을 보유하는 임시 변수의 형식을 업데이트합니다. 호출 수신자의 모든 반환 사이트가 동일한 타입을 생성하는 경우, 이 정확한 타입 정보는 후속 호출을 가상 제거하는 데 사용됩니다. 이 향상된 기능은 후반의 역가상화 및 배열 열거형 추상화 해제의 향상된 기능을 보완합니다.

프로필 데이터

.NET 10은 프로필 데이터를 더 잘 활용하도록 JIT의 인라인 정책을 향상시킵니다. JIT의 인라이너는 특정 크기보다 큰 메서드를 고려하지 않음으로써, 호출자 메서드가 비대해지는 것을 방지합니다. 호출자에게 인라인 후보가 자주 실행됨을 시사하는 프로필 데이터가 있는 경우, 인라이너는 후보에 대한 크기 허용 범위를 증가시킵니다.

JIT가 프로필 데이터가 없는 특정 호출 수신자 `Callee`를 프로필 데이터가 있는 특정 호출자 `Caller`에 인라인으로 삽입한다고 가정합니다. 호출 수신자가 너무 작아서 계측하는 것이 가치가 없거나, 혹은 너무 자주 인라인되어 충분한 호출 횟수를 가질 수 없을 경우 이러한 불일치가 발생할 수 있습니다. 자체 인라인 후보가 있는 경우 `Callee` JIT는 이전에 프로필 데이터가 부족하여 `Callee` 기본 크기 제한으로 고려하지 않았습니다. 이제 JIT는 프로필 데이터가 있다는 것을 깨닫고 `Caller` 크기 제한을 완화합니다(하지만 프로필 데이터가 있는 경우 `Callee` 와 같은 정도가 아니라 정밀도 손실을 고려하여).

마찬가지로 JIT에서 호출 사이트가 인라인 처리에 수익성이 없다고 판단하면 향후 인라인 처리 시도를 고려하지 못하도록 하는 방법을 `NoInlining` 표시합니다. 그러나 많은 인라인 추론은 프로필 데이터에 민감합니다. 예를 들어 JIT는 프로필 데이터가 없는 경우 메서드가 너무 커서 인라인 처리할 가치가 없다고 결정할 수 있습니다. 그러나 호출자가 충분히 뜨거울 때 JIT는 크기 제한을 완화하고 호출을 인라인으로 처리할 수 있습니다. .NET 10에서 JIT는 프로필 데이터를 사용한 통화 사이트 최적화를 방지하기 위해 수익성이 없는 인라인을 `NoInlining`로 플래그 지정하지 않습니다.

AVX10.2 지원

.NET 10에는 x64 기반 프로세서에 대한 AVX(Advanced Vector Extensions) 10.2 지원이 도입되었습니다. System.Runtime.Intrinsics.X86.Avx10v2 클래스에서 사용할 수 있는 새 내장 함수는 지원되는 하드웨어를 사용할 수 있게 되면 테스트할 수 있습니다.

AVX10.2 사용 하드웨어는 아직 사용할 수 없으므로 AVX10.2에 대한 JIT의 지원은 현재 기본적으로 사용하지 않도록 설정되어 있습니다.

스택 할당

스택 할당은 GC가 추적해야 하는 개체의 수를 줄이고 다른 최적화의 잠금을 해제합니다. 예를 들어 개체가 스택 할당된 후 JIT는 개체를 스칼라 값으로 완전히 바꾸는 것을 고려할 수 있습니다. 따라서 스택 할당은 참조 형식의 추상화 페널티를 줄이는 데 중요합니다. .NET 10은 값 형식의 작은 배열과 참조 형식의작은 배열에 대한 스택 할당을 추가합니다. 로컬 구조체 필드와 대리자를 위한 스케이프 분석도 포함됩니다. (이스케이프할 수 없는 개체는 스택에 할당할 수 있습니다.)

값 형식의 작은 배열

이제 JIT는 부모 메서드보다 오래 가지 않도록 보장할 수 있는 경우 GC 포인터를 포함하지 않는 작은 고정 크기 배열의 값 형식을 스택 할당합니다. 다음 예제에서 JIT는 컴파일 시간에 `numbers` 이 호출 `Sum`보다 오래 존재하지 않는 세 개의 정수로 구성된 배열임을 알기 때문에, 이를 스택에 할당합니다.

C#

```
static void Sum()
{
    int[] numbers = {1, 2, 3};
    int sum = 0;

    for (int i = 0; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }

    Console.WriteLine(sum);
}
```

참조 형식의 작은 배열

.NET 10은 .NET 9 스택 할당 개선 사항을 참조 형식의 작은 배열로 확장합니다. 이전에 참조 형식 배열은 수명이 단일 메서드 내로 제한되더라도 항상 힙에 할당되었습니다. 이제 JIT는 생성 컨텍스트를 초과하여 지속되지 않는다고 판단할 때 이러한 배열을 스택 할당할 수 있습니다. 다음 예제에서는 이제 배열 `words` 이 스택에 할당됩니다.

C#

```
static void Print()
{
    string[] words = {"Hello", "World!"};
    foreach (var str in words)
    {
        Console.WriteLine(str);
    }
}
```

이스케이프 분석

이스케이프 분석은 개체가 부모 메서드보다 오래 지속될 수 있는지 여부를 결정합니다. 개체는 "escape"되는 것으로, 지역 변수가 아닌 변수에 할당되거나 JIT에서 인라인되지 않는 함수에 전

달될 때 발생합니다. 개체가 이스케이프할 수 없다면 스택에 할당할 수 있습니다. .NET 10에는 다음에 대한 이스케이프 분석이 포함됩니다.

- 로컬 구조체 필드
- 대표자

로컬 구조체 필드

.NET 10부터 JIT는 더 많은 스택 할당을 가능하게 하고 힙 오버헤드를 줄이는 *구조체 필드에서* 참조하는 개체를 고려합니다. 다음 예제를 고려하세요.

C#

```
public class Program
{
    struct GCStruct
    {
        public int[] arr;
    }

    public static int Main()
    {
        int[] x = new int[10];
        GCStruct y = new GCStruct() { arr = x };
        return y.arr[0];
    }
}
```

일반적으로 JIT 스택은 다음과 같이 `x`이스케이프되지 않는 작은 고정 크기 배열을 할당합니다. `y.arr`에 대한 할당은 `x`이 이스케이프되지 않기 때문에 `y` 또한 이스케이프되지 않습니다. 그러나 JIT의 이전 이스케이프 분석 구현은 구조체 필드 참조를 모델링하지 않았습니다. .NET 9에서 생성된 x64 어셈블리에는 `Main` 호출을 통해 힙에 `CORINFO_HELP_NEWARR_1_VC`를 할당하기 위한 `x` 호출이 포함되어 있으며, 이는 이스케이프로 표시되었음을 나타냅니다.

asm

```
Program:Main():int (FullOpts):
       push     rax
       mov      rdi, 0x719E28028A98      ; int[]
       mov      esi, 10
       call     CORINFO_HELP_NEWARR_1_VC
       mov      eax, dword ptr [rax+0x10]
       add      rsp, 8
       ret
```

.NET 10에서 JIT는 문제의 구조체가 이스케이프되지 않는 한 더 이상 로컬 구조체 필드에서 참조하는 개체를 이스케이프로 표시하지 않습니다. 이제 어셈블리가 다음과 같이 표시됩니다(힙 할

당 도우미 호출이 사라졌습니다.)

asm

```
Program:Main():int (FullOpts):
       sub      rsp, 56
       vxorps   xmm8, xmm8, xmm8
       vmovdqu  ymmword ptr [rsp], ymm8
       vmovdqa  xmmword ptr [rsp+0x20], xmm8
       xor      eax, eax
       mov      qword ptr [rsp+0x30], rax
       mov      rax, 0x7F9FC16F8CC8      ; int[]
       mov      qword ptr [rsp], rax
       lea      rax, [rsp]
       mov      dword ptr [rax+0x08], 10
       lea      rax, [rsp]
       mov      eax, dword ptr [rax+0x10]
       add      rsp, 56
       ret
```

.NET 10의 추상화 해제 개선에 대한 자세한 내용은 dotnet/runtime#108913 참조하세요.

대표자

소스 코드가 IL로 컴파일되면 각 대리자는 대리자의 정의에 해당하는 메서드와 캡처된 변수와 일치하는 필드를 사용하여 클로저 클래스로 변환됩니다. 런타임에 캡처된 변수들을 인스턴스화하고 대리자를 호출하기 위해 클로저 개체와 `Func` 개체가 함께 생성됩니다. 이스케이프 분석에서 `Func` 개체가 현재의 범위를 벗어나지 않는 것으로 확인되면, JIT는 그 개체를 스택에 할당합니다.

다음 `Main` 방법을 고려합니다.

C#

```
 public static int Main()
{
    int local = 1;
    int[] arr = new int[100];
    var func = (int x) => x + local;
    int sum = 0;

    foreach (int num in arr)
    {
        sum += func(num);
    }

    return sum;
}
```

이전에 JIT는 다음과 같은 축약된 x64 어셈블리를 생성했습니다 `Main`. 루프에 들어가기 전에 `arr` 호출에 의해 표시된 대로 `func`, `func`, 그리고 `Program+<>c__DisplayClass0_0`라는 이름의 의 클로저 클래스가 모두 `CORINFO_HELP_NEW*`에 할당됩니다.

asm

```
       ; prolog omitted for brevity
       mov      rdi, 0x7DD0AE362E28      ; Program+<>c__DisplayClass0_0
       call     CORINFO_HELP_NEWSFAST
       mov      rbx, rax
       mov      dword ptr [rbx+0x08], 1
       mov      rdi, 0x7DD0AE268A98      ; int[]
       mov      esi, 100
       call     CORINFO_HELP_NEWARR_1_VC
       mov      r15, rax
       mov      rdi, 0x7DD0AE4A9C58      ; System.Func`2[int,int]
       call     CORINFO_HELP_NEWSFAST
       mov      r14, rax
       lea      rdi, bword ptr [r14+0x08]
       mov      rsi, rbx
       call     CORINFO_HELP_ASSIGN_REF
       mov      rsi, 0x7DD0AE461140      ; code for Program+<>c__DisplayClass0_0:
<Main>b__0(int):int:this
       mov      qword ptr [r14+0x18], rsi
       xor      ebx, ebx
       add      r15, 16
       mov      r13d, 100
G_M24375_IG03:  ;; offset=0x0075
       mov      esi, dword ptr [r15]
       mov      rdi, gword ptr [r14+0x08]
       call     [r14+0x18]System.Func`2[int,int]:Invoke(int):int:this
       add      ebx, eax
       add      r15, 4
       dec      r13d
       jne      SHORT G_M24375_IG03
       ; epilog omitted for brevity
```

`func` 이제는 범위 `Main`외부에서 참조되지 않으므로 스*택*에도 할당됩니다.

asm

```
       ; prolog omitted for brevity
       mov      rdi, 0x7B52F7837958      ; Program+<>c__DisplayClass0_0
       call     CORINFO_HELP_NEWSFAST
       mov      rbx, rax
       mov      dword ptr [rbx+0x08], 1
       mov      rsi, 0x7B52F7718CC8      ; int[]
       mov      qword ptr [rbp-0x1C0], rsi
       lea      rsi, [rbp-0x1C0]
       mov      dword ptr [rsi+0x08], 100
       lea      r15, [rbp-0x1C0]
       xor      r14d, r14d
```

```
        add      r15, 16
        mov      r13d, 100
G_M24375_IG03:  ;; offset=0x0099
        mov      esi, dword ptr [r15]
        mov      rdi, rbx
        mov      rax, 0x7B52F7901638      ; address of definition for "func"
        call     rax
        add      r14d, eax
        add      r15, 4
        dec      r13d
        jne      SHORT G_M24375_IG03
        ; epilog omitted for brevity
```

남은 `CORINFO_HELP_NEW*` 호출 중 하나가 종료를 위한 힙 할당이라는 점에 주목하십시오. 런타임 팀은 이후 릴리스에서 종료 스택 할당을 지원하기 위해 이스케이프 분석을 확장할 계획입니다.

NativeAOT 타입 사전 초기화기 개선

NativeAOT의 형식 프리니티알라이저는 이제 `conv.*` 및 `neg` opcode의 모든 변형을 지원합니다. 이러한 향상된 기능을 통해 캐스팅 또는 부정 작업을 포함하는 메서드를 미리 초기화하여 런타임 성능을 더욱 최적화할 수 있습니다.

Arm64 쓰기 장벽 개선

. NET의 GC(가비지 수집기)는 세대별이므로 수집 성능을 향상시키기 위해 라이브 개체를 연령별로 구분합니다. GC는 수명이 긴 개체가 특정 시점에 참조가 끊기거나 "죽은" 상태가 될 가능성이 적다는 가정 하에 젊은 세대를 더 자주 수집합니다. 그러나 이전 개체가 젊은 개체를 참조하기 시작한다고 가정합니다. GC는 젊은 개체를 수집할 수 없다는 것을 알아야 합니다. 그러나 젊은 개체를 수집하기 위해 이전 개체를 스캔해야 하는 경우 세대 GC의 성능 향상이 저하됩니다.

이 문제를 해결하기 위해 JIT는 GC 정보를 유지하기 위해 개체 참조 업데이트 전에 쓰기 장벽을 삽입합니다. x64에서 런타임은 GC의 구성에 따라 쓰기 속도와 컬렉션 효율성의 균형을 맞추기 위해 쓰기 장벽 구현 간에 동적으로 전환할 수 있습니다. .NET 10에서 이 기능은 Arm64에서도 사용할 수 있습니다. 특히 Arm64의 새로운 기본 쓰기 장벽 구현은 GC 지역을 보다 정확하게 처리하므로 쓰기 장벽 처리량에 약간의 비용으로 컬렉션 성능이 향상됩니다. 벤치마크는 새 GC 기본값을 사용하여 GC 일시 중지가 8%에서 20% 이상으로 개선된 것을 보여줍니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 11. 07.

.NET 10용 .NET 라이브러리의 새로운 기능

이 문서에서는 .NET 10용 .NET 라이브러리의 새로운 기능에 대해 설명합니다.

암호화

- SHA-1 이외의 지문으로 인증서 찾기
- ASCII/UTF-8에서 PEM으로 인코딩된 데이터 찾기
- PKCS#12/PFX 내보내기 암호화 알고리즘
- PQC(사후 양자 암호화)

SHA-1 이외의 지문으로 인증서 찾기

지문으로 고유하게 인증서를 찾는 것은 매우 일반적인 작업이지만 X509Certificate2Collection.Find(X509FindType, Object, Boolean) 메서드(모드의 경우 FindByThumbprint)는 SHA-1 지문 값만 검색합니다.

이러한 해시 알고리즘의 길이가 동일하기 때문에 SHA-2-256("SHA256") 및 SHA-3-256 지문을 찾기 위해 `Find` 메서드를 사용할 위험이 있습니다.

대신 .NET 10에서는 일치에 사용할 해시 알고리즘의 이름을 허용하는 새 메서드 가 도입되었습니다.

C#

```
X509Certificate2Collection coll =
store.Certificates.FindByThumbprint(HashAlgorithmName.SHA256, thumbprint);
Debug.Assert(coll.Count < 2, "Collection has too many matches, has SHA-2 been
broken?");
return coll.SingleOrDefault();
```

ASCII/UTF-8에서 PEM으로 인코딩된 데이터 찾기

PEM 인코딩(원래 *개인 정보 보호 강화 메일*, 하지만 이제는 전자 메일 외부에서 널리 사용됨)은 "텍스트"에 대해 정의됩니다. 즉, PemEncoding 클래스는 String 및 `ReadOnlySpan<char>`에서 실행되도록 설계되었습니다. 그러나 일반적으로(특히 Linux에서) ASCII(문자열) 인코딩을 사용하는 파일에 작성된 인증서와 같은 항목이 있습니다. 과거에는 파일을 열고 바이트를 문자(또는 문자열)로 변환해야 `PemEncoding`를 사용할 수 있었습니다.

새 PemEncoding.FindUtf8(ReadOnlySpan<Byte>) 메서드는 PEM이 7비트 ASCII 문자에 대해서만 정의되고 7비트 ASCII가 싱글 바이트 UTF-8 값과 완벽하게 겹칩니다. 이 새 메서드를 호출하면 UTF-8/ASCII-char 변환을 건너뛰고 파일을 직접 읽을 수 있습니다.

diff

```
byte[] fileContents = File.ReadAllBytes(path);
-char[] text = Encoding.ASCII.GetString(fileContents);
-PemFields pemFields = PemEncoding.Find(text);
+PemFields pemFields = PemEncoding.FindUtf8(fileContents);

-byte[] contents = Base64.DecodeFromChars(text.AsSpan()[pemFields.Base64Data]);
+byte[] contents = Base64.DecodeFromUtf8(fileContents.AsSpan()
[pemFields.Base64Data]);
```

PKCS#12/PFX 내보내기 암호화 알고리즘

새로운 ExportPkcs12 메서드를 통해 호출자는 암호화 및 다이제스트 알고리즘을 선택하여 X509Certificate에서 출력을 생성할 수 있습니다.

- Pkcs12ExportPbeParameters.Pkcs12TripleDesSha1 는 Windows XP-era 사실상 표준을 나타냅니다. 이전 암호화 알고리즘을 선택하여 PKCS#12/PFX 읽기를 지원하는 거의 모든 라이브러리 및 플랫폼에서 지원하는 출력을 생성합니다.
- Pkcs12ExportPbeParameters.Pbes2Aes256Sha256 는 3DES(SHA-1 대신 SHA-2-256) 대신 AES를 사용해야 하지만 모든 판독기(예: Windows XP)가 출력을 이해하지 못할 수 있음을 나타냅니다.

더 많은 제어를 원하는 경우 을 허용하는 오버로드 를 PbeParameters사용할 수 있습니다.

PQC(사후 양자 암호화)

.NET 10에는 ML-KEM(FIPS 203), ML-DSA(FIPS 204) 및 SLH-DSA(FIPS 205)의 세 가지 새로운 비대칭 알고리즘에 대한 지원이 포함되어 있습니다. 새 형식은 다음과 같습니다.

- System.Security.Cryptography.MLKem
- System.Security.Cryptography.MLDsa
- System.Security.Cryptography.SlhDsa

혜택을 거의 추가하지 않으므로 이러한 새 형식은 .에서 AsymmetricAlgorithm파생되지 않습니다. 새 형식은 `AsymmetricAlgorithm` 개체를 만든 다음 키를 가져오거나 새 키를 생성하는 방식 대신 정적 메서드를 사용하여 키를 생성하거나 가져옵니다.

C#

```
using System;
using System.IO;
using System.Security.Cryptography;

private static bool ValidateMLDsaSignature(ReadOnlySpan<byte> data,
ReadOnlySpan<byte> signature, string publicKeyPath)
```

```
{
    string publicKeyPem = File.ReadAllText(publicKeyPath);

    using (MLDsa key = MLDsa.ImportFromPem(publicKeyPem))
    {
        return key.VerifyData(data, signature);
    }
}
```

또한 개체 속성을 설정하고 키를 구체화하는 대신 이러한 새 형식의 키 생성은 필요한 모든 옵션을 사용합니다.

C#

```
using (MLKem key = MLKem.GenerateKey(MLKemAlgorithm.MLKem768))
{
    string publicKeyPem = key.ExportSubjectPublicKeyInfoPem();
    ...
}
```

이러한 알고리즘은 모두 현재 시스템에서 알고리즘이 지원되는지 여부를 나타내는 정적 `IsSupported` 속성을 갖는 패턴을 계속합니다.

.NET 10에는 PQC(Post-Quantum Cryptography)에 대한 Windows Cryptography API: 차세대(CNG) 지원이 포함되어 있으며, PQC를 지원하는 Windows 시스템에서 이러한 알고리즘을 사용할 수 있습니다. 다음은 그 예입니다.

C#

```
using System;
using System.IO;
using System.Security.Cryptography;

private static bool ValidateMLDsaSignature(ReadOnlySpan<byte> data,
ReadOnlySpan<byte> signature, string publicKeyPath)
{
    string publicKeyPem = File.ReadAllText(publicKeyPath);

    using MLDsa key = MLDsa.ImportFromPem(publicKeyPem);
    return key.VerifyData(data, signature);
}
```

PQC 알고리즘은 시스템 암호화 라이브러리가 OpenSSL 3.5 이상 또는 PQC 지원을 사용하는 Windows CNG인 시스템에서 사용할 수 있습니다. 형식은 MLKem로 `[Experimental]`로 표시되지 않지만, 일부 메서드는 표시되어 있으며, 기본 표준이 완료될 때까지 계속 그렇게 유지될 것입니다. 개발이 완료될 때까지 MLDsa, SlhDsa, 및 CompositeMLDsa 클래스는 진단 `[Experimental]` 상태로 표시됩니다`SYSLIB5006`.

ML-DSA

클래스에는 MLDsa 일반적인 코드 패턴을 간소화하는 사용 편의성 기능이 포함되어 있습니다.

diff

```
private static byte[] SignData(string privateKeyPath, ReadOnlySpan<byte> data)
{
    using (MLDsa signingKey =
MLDsa.ImportFromPem(File.ReadAllBytes(privateKeyPath)))
    {
-       byte[] signature = new byte[signingKey.Algorithm.SignatureSizeInBytes];
-       signingKey.SignData(data, signature);
+       return signingKey.SignData(data);
-       return signature;
    }
}
```

또한 .NET 10은 "순수" ML-DSA와 구별할 수 있도록 "PreHash"라고 하는 HashML-DSA에 대한 지원을 추가합니다. 기본 사양이 OID(개체 식별자) 값과 상호 작용할 때 이 `[Experimental]` 형식의 SignPreHash 및 VerifyPreHash 메서드는 점선 소수점 OID를 문자열로 사용합니다. 이는 HashML-DSA 사용하는 더 많은 시나리오가 잘 정의됨에 따라 진화할 수 있습니다.

C#

```
private static byte[] SignPreHashSha3_256(MLDsa signingKey, ReadOnlySpan<byte> data)
{
    const string Sha3_256Oid = "2.16.840.1.101.3.4.2.8";
    return signingKey.SignPreHash(SHA3_256.HashData(data), Sha3_256Oid);
}
```

RC 1부터 ML-DSA 고급 암호화 시나리오에 대한 추가 유연성을 제공하는 "외부" mu 값에서 생성되고 확인된 서명도 지원합니다.

C#

```
private static byte[] SignWithExternalMu(MLDsa signingKey, ReadOnlySpan<byte>
externalMu)
{
    return signingKey.SignMu(externalMu);
}

private static bool VerifyWithExternalMu(MLDsa verifyingKey, ReadOnlySpan<byte>
externalMu, ReadOnlySpan<byte> signature)
{
    return verifyingKey.VerifyMu(externalMu, signature);
}
```

복합 ML-DSA

.NET 10은 .NET 10 GA의 초안 8에서 ietf-lamps-pq-composite-sigs 를 지원하기 위해 새로운 형식을 도입했으며, 이에는 CompositeMLDsa 와 CompositeMLDsaAlgorithm 형식이 포함되고, RSA 변형을 위한 기본 메서드 구현이 포함됩니다.

C#

```
var algorithm = CompositeMLDsaAlgorithm.MLDsa65WithRSA4096Pss;
using var privateKey = CompositeMLDsa.GenerateKey(algorithm);

byte[] data = [42];
byte[] signature = privateKey.SignData(data);

using var publicKey = CompositeMLDsa.ImportCompositeMLDsaPublicKey(algorithm,
privateKey.ExportCompositeMLDsaPublicKey());
Console.WriteLine(publicKey.VerifyData(data, signature)); // True

signature[0] ^= 1; // Tamper with signature
Console.WriteLine(publicKey.VerifyData(data, signature)); // False
```

AES KeyWrap with Padding(IETF RFC 5649)

AES-KWP CMS(암호화 메시지 구문) EnvelopedData와 같은 생성에서 가끔 사용되는 알고리즘입니다. 여기서 콘텐츠는 한 번 암호화되지만 암호 해독 키는 각각 고유한 비밀 형식의 여러 당사자에게 배포되어야 합니다.

이제 .NET은 클래스의 인스턴스 메서드를 통해 AES-KWP 알고리즘을 Aes 지원합니다.

C#

```
private static byte[] DecryptContent(ReadOnlySpan<byte> kek, ReadOnlySpan<byte>
encryptedKey, ReadOnlySpan<byte> ciphertext)
{
    using (Aes aes = Aes.Create())
    {
        aes.SetKey(kek);

        Span<byte> dek = stackalloc byte[256 / 8];
        int length = aes.DecryptKeyWrapPadded(encryptedKey, dek);

        aes.SetKey(dek.Slice(0, length));
        return aes.DecryptCbc(ciphertext);
    }
}
```

세계화 및 날짜/시간

- DateOnly 형식에 대한 ISOWeek의 새 메서드 오버로드
- 문자열 비교를 위한 숫자 순서 지정
- 단일 매개 변수를 사용하는 새 TimeSpan.FromMilliseconds 오버로드

DateOnly 형식에 대한 새로운 메서드 오버로드가 ISOWeek에 추가되었습니다.

ISOWeek 클래스는 원래 DateTime 형식이 존재하기 전에 도입되었기 때문에 DateOnly단독으로 작동하도록 설계되었습니다. 이제 `DateOnly` 사용할 수 있으므로 `ISOWeek` 지원하는 것이 좋습니다. 다음 새로 추가된 오버로드는 다음과 같습니다.

- GetWeekOfYear(DateOnly)
- GetYear(DateOnly)
- ToDateOnly(Int32, Int32, DayOfWeek)

문자열 비교를 위한 숫자 순서 지정

숫자 문자열 비교는 사전적이 아닌 숫자로 문자열을 비교하기 위해 요청이 많은 기능입니다. 예를 들어 `2`은 `10`보다 작으므로, 숫자 순서대로 정렬할 때 `"2"`가 `"10"` 앞에 나와야 합니다. 마찬가지로 `"2"` 및 `"02"` 같은 숫자입니다. 이제 새 NumericOrdering 옵션을 사용하여 다음과 같은 유형의 비교를 수행할 수 있습니다.

C#

```
StringComparer numericStringComparer =
StringComparer.Create(CultureInfo.CurrentCulture, CompareOptions.NumericOrdering);

Console.WriteLine(numericStringComparer.Equals("02", "2"));
// Output: True

foreach (string os in new[] { "Windows 8", "Windows 10", "Windows 11"
}.Order(numericStringComparer))
{
    Console.WriteLine(os);
}

// Output:
// Windows 8
// Windows 10
// Windows 11

HashSet<string> set = new HashSet<string>(numericStringComparer) { "007" };
Console.WriteLine(set.Contains("7"));
// Output: True
```

`IndexOf`, `LastIndexOf`, `StartsWith`, `EndsWith`, `IsPrefix` 및 `IsSuffix` 인덱스 기반 문자열 작업에는 이 옵션이 유효하지 않습니다.

단일 매개 변수를 사용하여 새 `TimeSpan.FromMilliseconds` 오버로드

TimeSpan.FromMilliseconds(Int64, Int64) 메서드는 이전에 단일 매개 변수를 사용하는 오버로드를 추가하지 않고 도입되었습니다.

두 번째 매개 변수는 선택 사항이기 때문에 작동하지만 다음과 같이 LINQ 식에서 사용할 때 컴파일 오류가 발생합니다.

C#

```
Expression<Action> a = () => TimeSpan.FromMilliseconds(1000);
```

LINQ 식에서 선택적 매개 변수를 처리할 수 없기 때문에 문제가 발생합니다. 이 문제를 해결하기 위해 .NET 10에서는 단일 매개 변수 를 사용하는 새 오버로드 가 도입되었습니다. 또한 기존 메서드를 수정하여 두 번째 매개 변수를 필수로 만듭니다.

현악기들

- 문자열 범위 내에서 작업을 위한 문자열 정규화 API
- 16진수 문자열 변환에 대한 UTF-8 지원

문자 범위를 다루는 문자열 정규화 API

유니코드 문자열 정규화는 오랫동안 지원되었지만 기존 API는 문자열 형식에서만 작동했습니다. 즉, 문자 배열 또는 범위와 같이 다른 형식으로 저장된 데이터를 가진 호출자는 이러한 API를 사용하기 위해 새 문자열을 할당해야 합니다. 또한 정규화된 문자열을 반환하는 API는 항상 정규화된 출력을 나타내는 새 문자열을 할당합니다.

.NET 10에는 문자열 형식을 넘어 정규화를 확장하고 불필요한 할당을 방지하는 데 도움이 되는 문자 범위에서 작동하는 새로운 API가 도입되었습니다.

- StringNormalizationExtensions.GetNormalizedLength(ReadOnlySpan<Char>, NormalizationForm)
- StringNormalizationExtensions.IsNormalized(ReadOnlySpan<Char>, NormalizationForm)
- StringNormalizationExtensions.TryNormalize(ReadOnlySpan<Char>, Span<Char>, Int32, NormalizationForm)

16진수 문자열 변환에 대한 UTF-8 지원

.NET 10은 클래스에서 Convert 16진수 문자열 변환 작업에 대한 UTF-8 지원을 추가합니다. 이러한 새로운 메서드는 중간 문자열 할당 없이 UTF-8 바이트 시퀀스와 16진수 표현 간에 변환하는 효율적인 방법을 제공합니다.

- Convert.FromHexString(ReadOnlySpan<Byte>)
- Convert.FromHexString(ReadOnlySpan<Byte>, Span<Byte>, Int32, Int32)
- Convert.TryToHexString(ReadOnlySpan<Byte>, Span<Byte>, Int32)
- Convert.TryToHexStringLower(ReadOnlySpan<Byte>, Span<Byte>, Int32)

기존의 `string` 및 `ReadOnlySpan<char>`과 함께 작동하는 오버로드를 반영하되, 이미 UTF-8 데이터를 사용하는 경우 성능을 향상시키기 위해 UTF-8로 인코딩된 바이트에서 직접 작동하는 메서드입니다.

수집품

- 에 대한 추가 TryAdd 및 TryGetValue 오버로드 OrderedDictionary<TKey, TValue>

`TryAdd`에 대한 추가적인 `TryGetValue` 및 `OrderedDictionary<TKey, TValue>` 오버로드

OrderedDictionary<TKey,TValue> 다른 `TryAdd` 구현처럼 추가 및 검색을 위한 `TryGetValue` 및 `IDictionary<TKey, TValue>` 제공합니다. 그러나 더 많은 작업을 수행할 수 있는 시나리오가 있으므로 항목에 인덱스를 반환하는 새 오버로드가 추가됩니다.

- TryAdd(TKey, TValue, Int32)
- TryGetValue(TKey, TValue, Int32)

그런 다음 이 인덱스를 GetAt 및 SetAt과 함께 사용하여 항목에 빠르게 액세스할 수 있습니다. 새 `TryAdd` 오버로드를 사용하는 예제는 정렬된 사전에서 키-값 쌍을 추가하거나 업데이트하는 것입니다.

C#

```
// Try to add a new key with value 1.
if (!orderedDictionary.TryAdd(key, 1, out int index))
{
    // Key was present, so increment the existing value instead.
    int value = orderedDictionary.GetAt(index).Value;
    orderedDictionary.SetAt(index, value + 1);
}
```

이 새로운 API는 이미 JsonObject에 사용되고 있으며, 속성을 업데이트하는 성능이 10~20%%향상됩니다.

직렬화

- 에서 ReferenceHandler 지정 허용 JsonSourceGenerationOptions
- 중복 JSON 속성을 허용하지 않는 옵션
- 엄격한 JSON serialization 옵션
- JSON serializer에 대한 PipeReader 지원

`JsonSourceGenerationOptions`에서 ReferenceHandler를 지정할 수 있습니다.

JSON serialization에 소스 생성기를 사용하면, 주기가 직렬화되거나 역직렬화될 때 생성된 컨텍스트가 예외를 발생시킵니다. 이제 ReferenceHandler에서 JsonSourceGenerationOptionsAttribute을 지정하여 이 동작을 사용자 지정할 수 있습니다. 다음은 `JsonKnownReferenceHandler.Preserve`사용하는 예제입니다.

C#

```
public static void MakeSelfRef()
{
    SelfReference selfRef = new SelfReference();
    selfRef.Me = selfRef;

    Console.WriteLine(JsonSerializer.Serialize(selfRef,
ContextWithPreserveReference.Default.SelfReference));
    // Output: {"$id":"1","Me":{"$ref":"1"}}
}

[JsonSourceGenerationOptions(ReferenceHandler = JsonKnownReferenceHandler.Preserve)]
[JsonSerializable(typeof(SelfReference))]
internal partial class ContextWithPreserveReference : JsonSerializerContext
{
}

internal class SelfReference
{
    public SelfReference Me { get; set; } = null!;
}
```

중복 JSON 속성을 허용하지 않는 옵션

JSON 사양은 JSON 페이로드를 역직렬화할 때 중복 속성을 처리하는 방법을 지정하지 않습니다. 이로 인해 예기치 않은 결과와 보안 취약성이 발생할 수 있습니다. .NET 10에는 중복 JSON

속성을 허용하지 않는 옵션이 도입되었습니다 JsonSerializerOptions.AllowDuplicateProperties .

C#

```
string json = """{ "Value": 1, "Value": -1 }""";
Console.WriteLine(JsonSerializer.Deserialize<MyRecord>(json).Value); // -1

JsonSerializerOptions options = new() { AllowDuplicateProperties = false };
JsonSerializer.Deserialize<MyRecord>(json, options);                // throws
JsonException
JsonSerializer.Deserialize<JsonObject>(json, options);              // throws
JsonException
JsonSerializer.Deserialize<Dictionary<string, int>>(json, options); // throws
JsonException

JsonDocumentOptions docOptions = new() { AllowDuplicateProperties = false };
JsonDocument.Parse(json, docOptions);   // throws JsonException

record MyRecord(int Value);
```

역직렬화 중에 값이 여러 번 할당되는지 확인하여 중복 항목이 검색되므로, 대소문자 구분 및 명명 정책과 같은 다른 옵션에서도 기대한 대로 기능이 작동합니다.

엄격한 JSON serialization 옵션

JSON 직렬 변환기는 직렬화 및 역직렬화를 사용자 지정하는 많은 옵션을 허용하지만 일부 애플리케이션에서는 기본값이 너무 완화될 수 있습니다. .NET 10은 다음 옵션을 포함하여 모범 사례를 따르는 새 JsonSerializerOptions.Strict 사전 설정을 추가합니다.

- 정책을 적용합니다. JsonUnmappedMemberHandling.Disallow
- JsonSerializerOptions.AllowDuplicateProperties을 비활성화합니다.
- 대/소문자 구분 속성 바인딩을 유지합니다.
- JsonSerializerOptions.RespectNullableAnnotations 및 JsonSerializerOptions.RespectRequiredConstructorParameters 설정을 모두 활성화합니다.

이러한 옵션은 JsonSerializerOptions.Default와 읽기 호환됩니다. JsonSerializerOptions.Default로 직렬화된 개체는 JsonSerializerOptions.Strict로 역직렬화할 수 있습니다.

JSON serialization에 대한 자세한 내용은 System.Text.Json 개요를 참조하세요.

JSON serializer에 대한 PipeReader 지원

JsonSerializer.Deserialize 는 기존 지원을 보완하는 을 지원 PipeReader합니다 PipeWriter . 이전에는 필수 변환에서 `PipeReader` a로 역 Stream직렬화했지만 새 오버로드는 직렬 변환기에 직접 통합하여 `PipeReader` 해당 단계를 제거합니다. 보너스로, 이미 보유하고있는 것에서 변환 할 필요가 없는 것은 몇 가지 효율성 이점을 얻을 수 있습니다.

기본 사용법은 다음과 같습니다.

C#

```
using System;
using System.IO.Pipelines;
using System.Text.Json;
using System.Threading.Tasks;

var pipe = new Pipe();

// Serialize to writer
await JsonSerializer.SerializeAsync(pipe.Writer, new Person("Alice"));
await pipe.Writer.CompleteAsync();

// Deserialize from reader
var result = await JsonSerializer.DeserializeAsync<Person>(pipe.Reader);
await pipe.Reader.CompleteAsync();

Console.WriteLine($"Your name is {result.Name}.");
// Output: Your name is Alice.

record Person(string Name);
```

다음은 청크로 토큰을 생성하는 생산자와 토큰을 수신하고 표시하는 소비자의 예입니다.

C#

```
using System;
using System.Collections.Generic;
using System.IO.Pipelines;
using System.Text.Json;
using System.Threading.Tasks;

var pipe = new Pipe();

// Producer writes to the pipe in chunks.
var producerTask = Task.Run(async () =>
{
    async static IAsyncEnumerable<Chunk> GenerateResponse()
    {
        yield return new Chunk("The quick brown fox", DateTime.Now);
        await Task.Delay(500);
        yield return new Chunk(" jumps over", DateTime.Now);
        await Task.Delay(500);
        yield return new Chunk(" the lazy dog.", DateTime.Now);
    }

    await JsonSerializer.SerializeAsync<IAsyncEnumerable<Chunk>>(pipe.Writer,
GenerateResponse());
    await pipe.Writer.CompleteAsync();
});
```

```
// Consumer reads from the pipe and outputs to console.
var consumerTask = Task.Run(async () =>
{
    var thinkingString = "...";
    var clearThinkingString = new string("\b\b\b");
    var lastTimestamp = DateTime.MinValue;

    // Read response to end.
    Console.Write(thinkingString);
    await foreach (var chunk in JsonSerializer.DeserializeAsyncEnumerable<Chunk>
(pipe.Reader))
    {
        Console.Write(clearThinkingString);
        Console.Write(chunk.Message);
        Console.Write(thinkingString);
        lastTimestamp = DateTime.Now;
    }

    Console.Write(clearThinkingString);
    Console.WriteLine($" Last message sent at {lastTimestamp}.");

    await pipe.Reader.CompleteAsync();
});

await producerTask;
await consumerTask;

record Chunk(string Message, DateTime Timestamp);
```

이 모든 항목은 가독성을 위해 여기에 서식이 지정된 JSON Pipe 으로 직렬화됩니다.

JSON

```
[
    {
        "Message": "The quick brown fox",
        "Timestamp": "2025-08-01T18:37:27.2930151-07:00"
    },
    {
        "Message": " jumps over",
        "Timestamp": "2025-08-01T18:37:27.8594502-07:00"
    },
    {
        "Message": " the lazy dog.",
        "Timestamp": "2025-08-01T18:37:28.3753669-07:00"
    }
]
```

System.Numerics

- 더 많은 왼손 매트릭스 변환 메서드

- 텐서 개선

더 많은 왼손 매트릭스 변환 메서드

.NET 10은 광고판 및 제한된 광고판 행렬에 대한 왼손 변환 매트릭스를 만들기 위한 나머지 API를 추가합니다. 예를 들어 CreateBillboard(Vector3, Vector3, Vector3, Vector3)왼손 좌표계를 대신 사용하는 경우 기존 오른손잡이와 같은 메서드를 사용할 수 있습니다.

- Matrix4x4.CreateBillboardLeftHanded(Vector3, Vector3, Vector3, Vector3)
- Matrix4x4.CreateConstrainedBillboardLeftHanded(Vector3, Vector3, Vector3, Vector3, Vector3)

텐서 향상

네임스페이스 System.Numerics.Tensors는 이제 Lengths 및 Strides에 접근하는 작업과 같은 작업에 대해 비제너릭 인터페이스 IReadOnlyTensor를 포함합니다. 조각 작업은 더 이상 데이터를 복사하지 않아 성능이 향상됩니다. 또한 성능이 중요하지 않은 경우, `object`로 값을 감싸 비일반적으로 데이터에 액세스할 수 있습니다.

이제 텐서 API가 안정적이며 더 이상 실험적 API로 표시되지 않습니다. API는 여전히 System.Numerics.Tensors NuGet 패키지를 참조해야 하지만 .NET 10 릴리스에 대해 철저히 검토 및 완료되었습니다. 이 형식은 기본 형식 `T` 이 작업을 지원할 때 C# 14 확장 연산자를 활용하여 산술 연산을 제공합니다. `T`가 관련 제네릭 수학 인터페이스를 구현하는 경우, 예를 들어 `IAdditionOperators<TSelf, TOther, TResult>` 나 `INumber<TSelf>`를 구현하면 연산이 지원됩니다. 예를 들어, `tensor + tensor`은 `Tensor<int>`로 사용할 수 있지만, `Tensor<bool>`에는 사용할 수 없습니다.

옵션 유효성 검사

- ValidationContext를 위한 AOT 안전한 새 생성자

`ValidationContext`용 새로운 AOT 안전 생성자

옵션 유효성 검사 중에 사용되는 클래스에는 ValidationContext 매개 변수를 명시적으로 수락하는 새 생성자 오버로드가 `displayName` 포함됩니다.

ValidationContext(Object, String, IServiceProvider, IDictionary<Object,Object>)

표시 이름은 AOT 안전을 보장하고 경고 없이 네이티브 빌드에서 사용할 수 있도록 합니다.

진단

- ActivitySource 및 Meter의 원격 분석 스키마 URL 지원
- 작업 이벤트 및 링크에 대한 프로세스 외부 추적 지원
- 비율 제한 추적 샘플링 지원

`ActivitySource` 및 `Meter` 에서 원격 분석 스키마 URL 지원

ActivitySource 및 Meter 은 이제 생성 시 OpenTelemetry 사양에 맞는 원격 분석 스키마 URL을 지정할 수 있도록 지원합니다. 원격 분석 스키마는 추적 및 메트릭 데이터에 대한 일관성과 호환성을 보장합니다. 또한, .NET 10에서는 ActivitySourceOptions, 여러 구성 옵션(예: ActivitySource)을 사용하여 인스턴스를 간편하게 생성할 수 있게 합니다.

새 API는 다음과 같습니다.

- ActivitySource(ActivitySourceOptions)
- ActivitySource.TelemetrySchemaUrl
- Meter.TelemetrySchemaUrl
- ActivitySourceOptions

활동 이벤트 및 링크에 대한 프로세스 외부 추적 지원

이 클래스를 Activity 사용하면 서비스 또는 구성 요소 간에 작업의 흐름을 추적하여 분산 추적을 수행할 수 있습니다. .NET은 이벤트 원본 공급자를 통해 이 추적 데이터를 프로세스 외부에서 `Microsoft-Diagnostics-DiagnosticSource` 직렬화할 수 있습니다. `Activity`는 ActivityLink 및 ActivityEvent 같은 추가 메타데이터를 포함할 수 있습니다. .NET 10은 이러한 링크 및 이벤트를 직렬화하기 위한 지원을 추가하므로 이제 프로시던트 외 추적 데이터에 해당 정보가 포함됩니다. 다음은 그 예입니다.

```txt
Events->"[(TestEvent1,2025-03-27T23:34:10.6225721+00:00,[E11:EV1,E12:EV2]),
(TestEvent2,2025-03-27T23:34:11.6276895+00:00,[E21:EV21,E22:EV22])]"
Links->"[(19b6e8ea216cb2ba36dd5d957e126d9f,98f7abcb3418f217,Recorded,null,false,
[alk1:alv1,alk2:alv2]),(2d409549aadfdbdf5d1892584a5f2ab2,4f3526086a350f50,None,null,
false)]"
```

속도 제한 추적 샘플링 기능 지원

분산 추적 데이터가 이벤트 원본 공급자를 통해 프로세스 외부에서 직렬화되면, 기록된 모든 활동을 내보내거나 비율에 따른 샘플링을 적용할 수 있습니다.

속도 제한 샘플링이라는 새 샘플링 옵션은 초당 직렬화된 *루트 활동* 수를 제한합니다. 이렇게 하면 데이터 볼륨을 보다 정확하게 제어할 수 있습니다.

out-of-proc 추적 데이터 집계기는 FilterAndPayloadSpecs의 옵션을 지정하여 샘플링을 사용하도록 설정하고 구성할 수 있습니다. 예를 들어 다음 설정은 모든 `ActivitySource` 인스턴스에서 직렬화를 초당 100개의 루트 활동으로 제한합니다.

```txt
[AS]*/-ParentRateLimitingSampler(100)
```

ZIP 파일

- ZipArchive 성능 및 메모리 향상
- 새 비동기 ZIP API
- 연결된 스트림에 대한 GZipStream의 성능 향상

ZipArchive 성능 및 메모리 향상

.NET 10은 ZipArchive성능 및 메모리 사용량을 향상시킵니다.

첫째, `ZipArchive` 모드에서 항목이 `Update`에 기록되는 방식이 최적화되었습니다. 이전에는 모든 ZipArchiveEntry 인스턴스가 메모리에 로드되고 다시 작성되어 높은 메모리 사용량 및 성능 병목 현상이 발생할 수 있었습니다. 최적화는 메모리 사용량을 줄이고 메모리에 모든 항목을 로드할 필요가 없도록 하여 성능을 향상시킵니다.

둘째, ZipArchive 항목의 추출은 이제 병렬 처리되고 내부 데이터 구조는 메모리 사용을 향상하기 위해 최적화됩니다. 이러한 향상된 기능은 성능 병목 상태 및 높은 메모리 사용량과 관련된 문제를 해결하므로 특히 대규모 보관을 처리할 때 `ZipArchive` 더 효율적이고 더 빠릅니다.

새 비동기 ZIP API

.NET 10에는 ZIP 파일을 읽거나 쓸 때 비차단 작업을 더 쉽게 수행할 수 있는 새로운 비동기 API가 도입되었습니다. 이 기능은 커뮤니티에서 매우 요청되었습니다.

ZIP 보관 파일을 추출, 생성 및 업데이트하는 데 새 `async` 메서드를 사용할 수 있습니다. 이러한 방법을 통해 개발자는 특히 I/O 바인딩된 작업과 관련된 시나리오에서 큰 파일을 효율적으로 처리하고 애플리케이션 응답성을 향상시킬 수 있습니다. 이러한 메서드는 다음과 같습니다.

- ZipArchive.CreateAsync(Stream, ZipArchiveMode, Boolean, Encoding, CancellationToken)
- ZipArchiveEntry.OpenAsync(CancellationToken)
- ZipFile.CreateFromDirectoryAsync

- ZipFile.ExtractToDirectoryAsync
- ZipFile.OpenAsync
- ZipFile.OpenReadAsync(String, CancellationToken)
- ZipFileExtensions.CreateEntryFromFileAsync
- ZipFileExtensions.ExtractToDirectoryAsync
- ZipFileExtensions.ExtractToFileAsync

이러한 API를 사용하는 예제는 미리 보기 4 블로그 게시물을 참조하세요.

연결된 스트림에 대한 GZipStream의 성능 향상

커뮤니티 기여는 연결된 GZip 데이터 스트림을 처리할 때의 GZipStream 성능을 향상시켰습니다. 이전에는 각 새 스트림 세그먼트가 내부 `ZLibStreamHandle`를 해제하고 다시 할당하여 추가 메모리 할당 및 초기화 오버헤드가 발생했습니다. 이 변경으로 이제 핸들이 다시 설정되고 다시 사용되어 관리되는 메모리 할당과 관리되지 않는 메모리 할당을 모두 줄이고 실행 시간을 개선합니다. 많은 수의 작은 데이터 스트림을 처리할 때 가장 큰 성능 개선(~35% 더 빠름)을 경험할 수 있습니다. 이 변경 사항:

- 연결된 스트림당 최대 64-80바이트의 메모리를 반복적으로 할당하지 않고 관리되지 않는 메모리를 추가로 절약합니다.
- 연결된 스트림당 실행 시간을 약 400 ns 줄입니다.

Windows 프로세스 관리

새 프로세스 그룹에서 Windows 프로세스 시작

Windows의 경우 이제 별도의 프로세스 그룹에서 프로세스를 시작하는 데 사용할 ProcessStartInfo.CreateNewProcessGroup 수 있습니다. 이렇게 하면 적절한 처리 없이 부모를 쓰러뜨릴 수 있는 격리된 신호를 자식 프로세스에 보낼 수 있습니다. 신호를 보내는 것은 강제 종료를 방지하는 데 편리합니다.

C#

```
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.InteropServices;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        bool isChildProcess = args.Length > 0 && args[0] == "child";
```

```
        if (!isChildProcess)
        {
            var psi = new ProcessStartInfo
            {
                FileName = Environment.ProcessPath,
                Arguments = "child",
                CreateNewProcessGroup = true,
            };

            using Process process = Process.Start(psi)!;
            Thread.Sleep(5_000);

            GenerateConsoleCtrlEvent(CTRL_C_EVENT, (uint)process.Id);
            process.WaitForExit();

            Console.WriteLine("Child process terminated gracefully, continue with
the parent process logic if needed.");
        }
        else
        {
            // If you need to send a CTRL+C, the child process needs to re-enable
CTRL+C handling, if you own the code, you can call SetConsoleCtrlHandler(NULL,
FALSE).
            // see
https://learn.microsoft.com/windows/win32/api/processthreadsapi/nf-
processthreadsapi-createprocessw#remarks
            SetConsoleCtrlHandler((IntPtr)null, false);

            Console.WriteLine("Greetings from the child process!  I need to be
gracefully terminated, send me a signal!");

            bool stop = false;

            var registration = PosixSignalRegistration.Create(PosixSignal.SIGINT,
ctx =>
            {
                stop = true;
                ctx.Cancel = true;
                Console.WriteLine("Received CTRL+C, stopping...");
            });

            StreamWriter sw = File.AppendText("log.txt");
            int i = 0;
            while (!stop)
            {
                Thread.Sleep(1000);
                sw.WriteLine($"{++i}");
                Console.WriteLine($"Logging {i}...");
            }

            // Clean up
            sw.Dispose();
            registration.Dispose();

            Console.WriteLine("Thanks for not killing me!");
```

```
        }
    }

    private const int CTRL_C_EVENT = 0;
    private const int CTRL_BREAK_EVENT = 1;

    [DllImport("kernel32.dll", SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool SetConsoleCtrlHandler(IntPtr handler,
[MarshalAs(UnmanagedType.Bool)] bool Add);

    [DllImport("kernel32.dll", SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool GenerateConsoleCtrlEvent(uint dwCtrlEvent, uint
dwProcessGroupId);
}
```

WebSocket 향상된 기능

WebSocketStream

.NET 10에서는 WebSocketStream.NET에서 가장 일반적인 시나리오와 이전에 번거로운 WebSocket 시나리오를 간소화하도록 설계된 새로운 API가 도입되었습니다.

기존 `WebSocket` API는 하위 수준이며 버퍼링 및 프레이밍 처리, 메시지 재구성, 인코딩/디코딩 관리, 스트림, 채널 또는 기타 전송 추상화와 통합하기 위한 사용자 지정 래퍼 작성 등 상당한 상용구가 필요합니다. 이러한 복잡성으로 인해 특히 스트리밍 또는 텍스트 기반 프로토콜이 있는 앱 또는 이벤트 기반 처리기의 경우 WebSocket을 전송으로 사용하기가 어렵습니다.

`WebSocketStream` 에서는 WebSocket을 Stream통해 -based 추상화가 제공되어 이러한 문제가 해결됩니다. 이렇게 하면 이진 또는 텍스트에 관계없이 데이터를 읽고, 쓰고, 구문 분석하기 위해 기존 API와 원활하게 통합할 수 있으며 수동 배관의 필요성을 줄일 수 있습니다.

`WebSocketStream` 에서는 일반적인 WebSocket 사용 및 프로덕션 패턴에 대해 친숙한 고급 API를 사용할 수 있습니다. 이러한 API는 마찰을 줄이고 고급 시나리오를 보다 쉽게 구현할 수 있도록 합니다.

일반적인 사용 패턴

다음은 일반적인 `WebSocketStream` 워크플로를 간소화하는 방법에 `WebSocket` 대한 몇 가지 예입니다.

스트리밍 텍스트 프로토콜(예: STOMP)

C#

```
using System.IO;
using System.Net.WebSockets;
using System.Threading;
using System.Threading.Tasks;

// Streaming text protocol (for example, STOMP).
using Stream transportStream = WebSocketStream.Create(
    connectedWebSocket,
    WebSocketMessageType.Text,
    ownsWebSocket: true);
// Integration with Stream-based APIs.
// Don't close the stream, as it's also used for writing.
using var transportReader = new StreamReader(transportStream, leaveOpen: true);
var line = await transportReader.ReadLineAsync(cancellationToken); // Automatic UTF-8 and new line handling.
transportStream.Dispose(); // Automatic closing handshake handling on `Dispose`.
```

스트리밍 이진 프로토콜(예: AMQP)

C#

```
using System;
using System.IO;
using System.Net.WebSockets;
using System.Threading;
using System.Threading.Tasks;

// Streaming binary protocol (for example, AMQP).
Stream transportStream = WebSocketStream.Create(
    connectedWebSocket,
    WebSocketMessageType.Binary,
    closeTimeout: TimeSpan.FromSeconds(10));
await message.SerializeToStreamAsync(transportStream, cancellationToken);
var receivePayload = new byte[payloadLength];
await transportStream.ReadExactlyAsync(receivePayload, cancellationToken);
transportStream.Dispose();
// `Dispose` automatically handles closing handshake.
```

단일 메시지를 스트림으로 읽습니다(예: JSON 역직렬화).

C#

```
using System.IO;
using System.Net.WebSockets;
using System.Text.Json;

// Reading a single message as a stream (for example, JSON deserialization).
using Stream messageStream =
```

```
WebSocketStream.CreateReadableMessageStream(connectedWebSocket,
WebSocketMessageType.Text);
// JsonSerializer.DeserializeAsync reads until the end of stream.
var appMessage = await JsonSerializer.DeserializeAsync<AppMessage>(messageStream);
```

단일 메시지를 스트림으로 작성(예: 이진 serialization)

C#

```
using System;
using System.IO;
using System.Net.WebSockets;
using System.Threading;
using System.Threading.Tasks;

// Writing a single message as a stream (for example, binary serialization).
public async Task SendMessageAsync(AppMessage message, CancellationToken
cancellationToken)
{
    using Stream messageStream =
WebSocketStream.CreateWritableMessageStream(_connectedWebSocket,
WebSocketMessageType.Binary);
    foreach (ReadOnlyMemory<byte> chunk in message.SplitToChunks())
    {
        await messageStream.WriteAsync(chunk, cancellationToken);
    }
} // EOM sent on messageStream.Dispose().
```

TLS 향상된 기능

macOS용 TLS 1.3(클라이언트)

.NET 10은 Apple의 Network.framework SslStream 를 통합하여 macOS에서 클라이언트 쪽 TLS 1.3 지원을 추가합니다 HttpClient. 지금까지 macOS는 TLS 1.3을 지원하지 않는 보안 전송을 사용했습니다. Network.framework를 옵트인하면 TLS 1.3을 사용할 수 있습니다.

범위 및 동작

- macOS만 이 릴리스의 클라이언트 쪽입니다.
- 옵트인. 기존 앱은 사용하도록 설정하지 않는 한 현재 스택을 계속 사용합니다.
- 사용하도록 설정하면 Network.framework를 통해 이전 TLS 버전(TLS 1.0 및 1.1)을 더 이상 사용할 수 없습니다.

사용 방법

코드에서 AppContext 스위치를 사용합니다.

C#

```
// Opt in to Network.framework-backed TLS on Apple platforms.
AppContext.SetSwitch("System.Net.Security.UseNetworkFramework", true);

using var client = new HttpClient();
var html = await client.GetStringAsync("https://example.com");
```

또는 환경 변수를 사용합니다.

Bash

```
# Opt-in via environment variable (set for the process or machine as appropriate)
DOTNET_SYSTEM_NET_SECURITY_USENETWORKFRAMEWORK=1
# or
DOTNET_SYSTEM_NET_SECURITY_USENETWORKFRAMEWORK=true
```

비고

- TLS 1.3은 SslStream TLS 1.3에 적용되며 이를 기반으로 빌드된 API(예: HttpClient/HttpMessageHandler)입니다.
- 암호 그룹은 Network.framework를 통해 macOS에서 제어됩니다.
- Network.framework를 사용하는 경우(예: 버퍼링, 읽기/쓰기 완료, 취소 의미 체계) 기본 스트림 동작이 다를 수 있습니다.
- 의미 체계는 0 바이트 읽기에 대해 다를 수 있습니다. 데이터 가용성을 검색하기 위해 길이가 0인 읽기를 사용하지 마세요.
- 특정 IDN(국제화된 도메인 이름) 호스트 이름은 Network.framework에서 거부될 수 있습니다. ASCII/Punycode(A-label) 호스트 이름을 선호하거나 macOS/Network.framework 제약 조건에 대해 이름의 유효성을 검사합니다.
- 앱이 특정 SslStream 에지 사례 동작을 사용하는 경우 Network.framework에서 유효성을 검사합니다.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 11. 07.

.NET 10용 SDK 및 도구의 새로운 기능

이 문서에서는 .NET 10용 .NET SDK의 새로운 기능 및 향상된 기능을 설명합니다.

.NET 도구의 향상된 기능

플랫폼별 .NET 도구

이제 단일 패키지에서 여러 RID(RuntimeIdentifiers)를 지원하여 .NET 도구를 게시할 수 있습니다. 도구 작성자는 지원되는 모든 플랫폼에 대해 이진 파일을 번들로 묶을 수 있으며. .NET CLI는 설치 또는 런타임에 올바른 이진 파일을 선택합니다. 이렇게 하면 플랫폼 간 도구 작성 및 배포가 훨씬 쉬워집니다.

이러한 향상된 도구는 다양한 패키징 변형을 지원합니다.

- **프레임워크에 종속된 플랫폼에 구애받지 않습니다** (클래식 모드는 .NET 10이 설치된 모든 위치에서 실행됨)
- **프레임워크 종속, 플랫폼별** (더 작고 각 플랫폼에 최적화됨)
- **자체 포함된 플랫폼별** (런타임 포함, .NET 설치 필요 없음)
- **최적화된 플랫폼별** (더 작은 크기, 사용되지 않는 코드 제거)
- **AOT 컴파일, 플랫폼별** (최대 성능 및 가장 작은 배포)

이러한 새 도구는 일반 게시된 애플리케이션과 매우 유사하게 작동하므로 애플리케이션에서 사용할 수 있는 모든 게시 옵션(예: 자체 포함, 트리밍 또는 AOT)도 도구에 적용할 수 있습니다.

일회성 도구 실행

이제 명령을 사용하여 `dotnet tool exec` 전역 또는 로컬로 설치하지 않고 .NET 도구를 실행할 수 있습니다. 이는 CI/CD 또는 임시 사용에 특히 유용합니다.

Bash

```
dotnet tool exec --source ./artifacts/package/ dotnetsay  "Hello, World!"
Tool package dotnetsay@1.0.0 will be downloaded from source <source>.
Proceed? [y/n] (y): y
  _   _          _   _                  __        __                          _       _   _
 | | | |   ___  | | | |   ___           \ \      / /   ___    _ __   | |   __| | | |
 | |_| |  / _ \ | | | |  / _ \           \ \ /\ / /   / _ \  | '__| | |  / _` | | |
 |  _  | |  __/ | | | | | (_) |  _        \ V  V /   | (_) | | |    | | | (_| | |_|
 |_| |_|  \___| |_| |_|  \___/  ( )        \_/\_/     \___/  |_|    |_|  \__,_| (_)
                               |/
```

이렇게 하면 지정된 도구 패키지가 한 명령으로 다운로드되고 실행됩니다. 기본적으로 도구가 로컬에 아직 없는 경우 다운로드를 확인하라는 메시지가 사용자에게 표시됩니다. 명시적 버전을 지정하지 않는 한 선택한 도구 패키지의 최신 버전이 사용됩니다(예: `dotnetsay@0.1.0`).

원샷 도구 실행은 로컬 도구 매니페스트에서 원활하게 작동합니다. 근처에 있는 `.config/dotnet-tools.json` 위치에서 도구를 실행하는 경우 해당 구성의 도구 버전이 사용 가능한 최신 버전 대신 사용됩니다.

새 `dnx` 도구 실행 스크립트

스크립트는 `dnx` 도구를 실행하는 간소화된 방법을 제공합니다. 처리를 위해 모든 인수를 `dotnet` CLI에 전달하여 도구를 최대한 간단하게 사용합니다.

Bash

```
dnx dotnetsay "Hello, World!"
```

명령의 `dnx` 실제 구현은 CLI 자체에 `dotnet` 있으므로 시간이 지남에 따라 동작이 진화할 수 있습니다.

.NET 도구 관리에 대한 자세한 내용은 .NET 도구 관리를 참조하세요.

`any` 플랫폼별 .NET 도구와 함께 RuntimeIdentifier 사용

플랫폼별 .NET 도구 기능은 도구를 미리 대상으로 하는 특정 플랫폼에 최적화되어 있는지 확인하는 데 적합합니다. 그러나 대상으로 지정하려는 플랫폼을 모두 알지 못하거나 .NET 자체에서 새 플랫폼을 지원하는 방법을 학습하고 도구를 실행할 수도 있습니다.

도구가 이러한 방식으로 작동하도록 하려면 프로젝트 파일에 런타임 식별자를 추가 `any` 합니다.

XML

```
<PropertyGroup>
  <RuntimeIdentifiers>
        linux-x64;
        linux-arm64;
        macos-arm64;
        win-x64;
        win-arm64;
        any
  </RuntimeIdentifiers>
</PropertyGroup>
```

이 RuntimeIdentifier는 플랫폼 호환성 검사의 '루트'에 있으며 , *모든* 플랫폼에 대한 지원을 선언하기 때문에 패키지되는 도구는 호환되는 종류의 도구인 프레임워크 종속, 플랫폼에 구애받지

않는 .NET DLL이며 호환되는 .NET 런타임을 실행해야 합니다. `dotnet pack` 도구를 만들 때 다른 플랫폼별 패키지 및 최상위 매니페스트 패키지와 함께 RuntimeIdentifier에 대한 `any` 새 패키지가 표시됩니다.

CLI 탐색 `--cli-schema` 사용

모든 CLI 명령에서 새 `--cli-schema` 옵션을 사용할 수 있습니다. 사용하는 경우 호출된 명령 또는 하위 명령에 대한 CLI 명령 트리의 JSON 표현을 출력합니다. 도구 작성자, 셸 통합 및 고급 스크립팅에 유용합니다.

Bash

```
dotnet clean --cli-schema
```

출력은 명령의 인수, 옵션 및 하위 명령에 대해 기계에서 읽을 수 있는 구조화된 설명을 제공합니다.

JSON

```
{
  "name": "clean",
  "version": "10.0.100-dev",
  "description": ".NET Clean Command",
  "arguments": {
    "PROJECT | SOLUTION": {
      "description": "The project or solution file to operate on. If a file is not
specified, the command will search the current directory for one.",
      "arity": { "minimum": 0, "maximum": null }
    }
  },
  "options": {
    "--artifacts-path": {
      "description": "The artifacts path. All output from the project, including
build, publish, and pack output, will go in subfolders under the specified path.",
      "helpName": "ARTIFACTS_DIR"
    }
  },
  "subcommands": {}
}
```

.NET Framework MSBuild에서 .NET MSBuild 작업 사용

MSBuild는 .NET의 기본 빌드 시스템으로, 프로젝트 빌드(예: 명령과 같이 `dotnet build`)를 구동하고 `dotnet pack`프로젝트에 대한 일반적인 정보 공급자 역할을 수행합니다(예: `dotnet list`

`package`명령에서 볼 수 있듯이 프로젝트를 실행하는 방법을 검색하는 것과 같은 `dotnet run` 명령에서 암시적으로 사용됨).

CLI 명령을 실행할 `dotnet` 때 사용되는 MSBuild 버전은 .NET SDK와 함께 제공되는 버전입니다. 그러나 Visual Studio를 사용하거나 MSBuild를 직접 호출하는 경우 사용되는 MSBuild 버전은 Visual Studio와 함께 설치된 버전입니다. 이러한 환경 차이에는 몇 가지 중요한 결과가 있습니다. 가장 중요한 것은 Visual Studio (또는 `msbuild.exe`에서 실행되는 경우)에서 실행되는 MSBuild가 .NET Framework 애플리케이션인 반면 CLI에서 `dotnet` 실행되는 MSBuild는 .NET 애플리케이션입니다. 즉, .NET에서 실행하도록 작성된 MSBuild 작업은 Visual Studio에서 빌드하거나 사용할 `msbuild.exe`때 사용할 수 없습니다.

.NET 10 `msbuild.exe` 부터 Visual Studio 2026은 .NET용으로 빌드된 MSBuild 작업을 실행할 수 있습니다. 즉, 이제 Visual Studio에서 빌드할 때와 마찬가지로 `msbuild.exe` CLI에서 빌드할 때도 동일한 MSBuild 작업을 `dotnet` 사용할 수 있습니다. 대부분의 .NET 사용자에게는 아무 것도 변경되지 않을 것입니다. 그러나 사용자 지정 MSBuild 작업의 작성자의 경우 이제 .NET을 대상으로 하는 작업을 작성하고 어디서나 작업하도록 할 수 있습니다. 이 변경의 목표는 MSBuild 작업을 더 쉽게 작성하고 공유할 수 있도록 하고 태스크 작성자가 .NET의 최신 기능을 활용할 수 있도록 하는 것입니다. 또한 이 변경으로 인해 .NET Framework와 .NET을 모두 지원하고 MSBuild .NET Framework 실행 공간에서 암시적으로 사용할 수 있는 .NET Framework 종속성 버전을 처리하는 다중 대상 지정 작업과 관련된 어려움이 줄어듭니다.

.NET 작업 구성

작업 작성자의 경우 이 새로운 동작을 쉽게 옵트인할 수 있습니다. MSBuild에 작업에 대해 알리도록 선언을 변경 `UsingTask` 하기만 하면 됩니다.

XML

```xml
<UsingTask TaskName="MyTask"
    AssemblyFile="path\to\MyTask.dll"
    Runtime="NET"
    TaskFactory="TaskHostFactory"
/>
```

`Runtime="NET"` 및 `TaskFactory="TaskHostFactory"` 특성은 MSBuild 엔진에 태스크를 실행하는 방법을 알려줍니다.

- `Runtime="NET"` 는 작업이 .NET Framework가 아닌 .NET용으로 빌드되었음을 MSBuild에 알릴 수 있습니다.
- `TaskFactory="TaskHostFactory"` 는 MSBuild에 `TaskHostFactory` 을 사용하여 태스크를 실행하도록 지시합니다. 이는 MSBuild의 기존 기능으로, 태스크가 out-of-process로 실행될 수 있습니다.

주의 사항 및 성능 튜닝

앞의 예제는 MSBuild에서 .NET 작업을 사용하는 가장 간단한 방법이지만 몇 가지 제한 사항이 있습니다. `TaskHostFactory` 항상 태스크가 out-of-process로 실행되므로 새 .NET 작업은 항상 MSBuild와 별도의 프로세스에서 실행됩니다. 즉, MSBuild 엔진과 태스크가 In-Process 통신 대신 IPC(프로세스 간 통신)를 통해 통신하기 때문에 작업 실행에 약간의 오버헤드가 있습니다. 대부분의 작업에서 이 오버헤드는 무시할 수 있지만 빌드에서 여러 번 실행되거나 많은 로깅을 수행하는 작업의 경우 이 오버헤드가 더 중요할 수 있습니다.

작업을 조금 더 수행하면, `dotnet`를 통해 실행될 때에도 작업이 여전히 프로세스 내에서 실행되도록 구성할 수 있습니다.

XML

```
<UsingTask TaskName="MyTask"
    AssemblyFile="path\to\MyTask.dll"
    Runtime="NET"
    TaskFactory="TaskHostFactory"
    Condition="$(MSBuildRuntimeType) == 'Full'"
/>
<UsingTask TaskName="MyTask"
    AssemblyFile="path\to\MyTask.dll"
    Runtime="NET"
    Condition="$(MSBuildRuntimeType) == 'Core'"
/>
```

MSBuild의 기능 덕분에 `Condition` MSBuild가 .NET Framework(Visual Studio 또는) 또는 `msbuild.exe`.NET `dotnet` (CLI)에서 실행되는지 여부에 따라 작업을 다르게 로드할 수 있습니다. 이 예제에서는 Visual Studio나 `msbuild.exe`에서 실행 시 태스크가 독립적으로 실행되지만, `dotnet` CLI에서 실행 시 프로세스 내에서 실행됩니다. 이렇게 하면 CLI에서 `dotnet` 실행할 때 최상의 성능을 제공하지만 Visual Studio 및 `msbuild.exe`에서 작업을 사용할 수 있습니다.

MSBuild에서 .NET 작업을 사용할 때 알아야 할 작은 기술적 제한 사항도 있습니다. 그 중 가장 주목할 만한 점은 `Host Object` MSBuild 작업의 기능이 아직 프로세스 부족 .NET 작업에 지원되지 않는다는 점입니다. 즉, 작업이 호스트 개체를 사용하는 경우 Visual Studio 또는 `msbuild.exe`에서 실행할 때 작동하지 않습니다. 호스트 개체에 대한 추가 지원은 향후 릴리스에서 계획되어 있습니다.

파일 기반 앱의 향상된 기능

.NET 10은 게시 지원 및 네이티브 AOT 기능을 포함하여 파일 기반 앱 환경에 중요한 업데이트를 제공합니다. 파일 기반 앱에 대한 소개는 파일 기반 앱 및 C# 프로그램 빌드 및 실행을 참조하세요.

게시 지원 및 네이티브 AOT를 사용하는 향상된 파일 기반 앱

파일 기반 앱은 이제 명령을 통해 `dotnet publish app.cs` 네이티브 실행 파일에 게시되도록 지원하므로 네이티브 실행 파일로 재배포할 수 있는 간단한 앱을 더 쉽게 만들 수 있습니다. 이제 모든 파일 기반 앱은 기본적으로 네이티브 AOT를 대상으로 합니다. 네이티브 AOT와 호환되지 않는 패키지 또는 기능을 사용해야 하는 경우 .cs 파일의 `#:property PublishAot=false` 지시문을 사용하여 사용하지 않도록 설정할 수 있습니다.

파일 기반 앱에는 향상된 기능도 포함됩니다.

- **프로젝트 참조**: 지시문을 통해 프로젝트를 참조할 수 있습니다 `#:project` .
- **런타임 경로 액세스**: 앱 파일 및 디렉터리 경로는 런타임에 을 통해 `System.AppContext.GetData` 사용할 수 있습니다.
- **향상된 shebang 지원**: 확장명 없는 파일에 대한 지원을 포함하여 향상된 shebang 처리로 직접 셸 실행

프로젝트 참조 예제

C#

```
#:project ../ClassLib/ClassLib.csproj

var greeter = new ClassLib.Greeter();
var greeting = greeter.Greet(args.Length > 0 ? args[0] : "World");
Console.WriteLine(greeting);
```

향상된 shebang 지원 예제

이제 셸에서 직접 실행되는 실행 파일 C# 파일을 만들 수 있습니다.

C#

```
#!/usr/bin/env dotnet

Console.WriteLine("Hello shebang!");
```

확장 없는 파일의 경우:

Bash

```
# 1. Create a single-file C# app with a shebang
cat << 'EOF' > hello.cs
#!/usr/bin/env dotnet
Console.WriteLine("Hello!");
EOF
```

```
# 2. Copy it (extensionless) into ~/utils/hello (~/utils is on my PATH)
mkdir -p ~/utils
cp hello.cs ~/utils/hello

# 3. Mark it executable
chmod +x ~/utils/hello

# 4. Run it directly from anywhere
cd ~
hello
```

이러한 향상된 기능은 빠른 스크립팅 및 프로토타입 시나리오에 대한 단순성을 유지하면서 파일 기반 앱을 더욱 강력하게 만듭니다.

네이티브 AOT에 대한 자세한 내용은 .NET 네이티브 AOT를 참조하세요.

프레임워크 제공 패키지 참조 삭제

.NET 10부터 NuGet 감사 기능은 프로젝트에서 사용하지 않는 프레임워크 제공 패키지 참조를 정리 할 수 있습니다. 이 기능은 최신 SDK에서 = .NET 10.0을 대상으로 >하는 프로젝트의 모든 프레임워크에 대해 기본적으로 사용하도록 설정됩니다. 이러한 변경은 빌드 프로세스 중에 복원 및 분석되는 패키지 수를 줄이는 데 도움이 되며, 이로 인해 빌드 시간이 빨라지고 디스크 공간 사용량이 감소할 수 있습니다. 또한 NuGet 감사 및 기타 종속성 검사 메커니즘에서 거짓 양성을 줄일 수 있습니다.

이 기능을 사용하도록 설정하면 애플리케이션에서 생성된 *.deps.json* 파일의 내용이 감소할 수 있습니다. .NET 런타임에서 제공하는 모든 패키지 참조는 생성된 종속성 파일에서 자동으로 제거됩니다. 직접 패키지 참조가 정리 범위 내에 있을 때, PrivateAssets="all"과 IncludeAssets="none"이 적용됩니다.

이 기능은 나열된 TFM에 대해 기본적으로 사용하도록 설정되어 있지만 프로젝트 파일 또는 `RestoreEnablePackagePruning` 파일에서 `false` 속성을 설정하여 사용하지 않도록 설정할 수 있습니다.

보다 일관된 명령 순서

.NET 10부터 `dotnet` CLI 도구에는 보다 쉽게 기억하고 입력할 수 있도록 공통 명령에 대한 새로운 별칭이 포함되어 있습니다. 새로운 명령어는 다음 표에 표시됩니다.

테이블 확장

새 명사 우선 양식	별칭에 대한
dotnet package add	`dotnet add package`
dotnet package list	`dotnet list package`
dotnet package remove	`dotnet remove package`
dotnet reference add	`dotnet add reference`
dotnet reference list	`dotnet list reference`
dotnet reference remove	`dotnet remove reference`

새 명사 우선 양식은 일반 CLI 표준과 일치하므로 `dotnet` CLI가 다른 도구와 더 일치합니다. 동사 우선 형태가 여전히 작동하긴 하지만, 스크립트와 문서에서 가독성과 일관성을 높이기 위해 명사 우선 형태를 사용하는 것이 좋습니다.

CLI 명령은 기본적으로 대화형 터미널에서 대화형 모드로 설정됩니다.

`--interactive` 이제 대화형 터미널의 CLI 명령에 대해 플래그가 기본적으로 사용하도록 설정됩니다. 이 변경을 통해 명령은 플래그를 명시적으로 설정하지 않고도 동적으로 자격 증명을 검색하거나 다른 대화형 동작을 수행할 수 있습니다. 비대화형 시나리오의 경우 를 지정 `--interactive false`하여 대화형 작업을 사용하지 않도록 설정할 수 있습니다.

네이티브 셸 탭 완성 스크립트

이제 `dotnet` CLI는 `dotnet completions generate [SHELL]` 명령을 통해 인기 있는 셸에 대한 네이티브 탭 완성 스크립트를 생성하는 기능을 지원합니다. 지원되는 셸에는 `bash`, `fish`, `nushell`, `powershell`, 및 `zsh`가 포함됩니다. 이러한 스크립트는 더 빠르고 통합된 탭 완성 기능을 제공하여 유용성을 향상시킵니다. 예를 들어, PowerShell에서 다음 코드를 추가하여 자동 완성을 활성화할 수 있습니다 `$PROFILE`.

PowerShell

```
dotnet completions script pwsh | out-String | Invoke-Expression -ErrorAction
SilentlyContinue
```

콘솔 앱은 기본적으로 컨테이너 이미지를 만들 수 있습니다.

이제 콘솔 앱은 프로젝트 파일의 `dotnet publish /t:PublishContainer` 속성을 요구하지 않고 컨테이너 이미지를 `<EnableSdkContainerSupport>` 만들 수 있습니다. 그러면 콘솔 앱이 ASP.NET Core 및 Worker SDK 앱의 동작과 일치합니다.

컨테이너의 이미지 형식을 명시적으로 제어

새 `<ContainerImageFormat>` 속성을 사용하면 컨테이너 이미지의 형식을 명시적으로 둘 중 하나 `Docker` 또는 `OCI`로 설정할 수 있습니다. 이 속성은 기본 이미지 형식 및 컨테이너가 다중 아키텍처인지 여부에 따라 달라지는 기본 동작을 재정의합니다.

`dotnet test`에서 Microsoft 테스트 플랫폼 지원

.NET 10 `dotnet test` 부터 기본적으로 Microsoft.Testing.Platform을 지원합니다. 이 기능을 사용하도록 설정하려면 *global.json* 파일에 다음 구성을 추가합니다.

JSON

```
{
    "test": {
        "runner": "Microsoft.Testing.Platform"
    }
}
```

자세한 내용은 테스트를 dotnet test참조하세요.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 11. 07.

.NET 10의 주요 변경 내용

앱을 .NET 10으로 마이그레이션하는 경우 여기에 나열된 주요 변경 내용이 영향을 줄 수 있습니다. 변경 내용은 ASP.NET Core 또는 Windows Forms와 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경을 *이진 파일 비호환*, *원본 비호환* 또는 *동작 변경*으로 분류합니다.

- **이진 파일 비호환** - 새 런타임이나 구성 요소에 대해 실행할 때 기존 이진 파일의 동작이 크게 변경될 수 있습니다(예: 로드 또는 실행 실패). 그런 경우 다시 컴파일이 필요합니다.
- **원본 비호환** - 새 SDK 또는 구성 요소를 사용하여 다시 컴파일하거나 새 런타임을 대상으로 하는 경우 기존 소스 코드를 성공적으로 컴파일하려면 원본을 변경해야 할 수도 있습니다.
- **동작 변경** - 기존 코드 및 이진 파일은 런타임에 다르게 동작할 수 있습니다. 새 동작이 바람직하지 않은 경우 기존 코드를 업데이트하고 다시 컴파일해야 합니다.

> **참고 항목**
>
> 이 문서는 진행 중인 작업입니다. .NET 10의 주요 변경 내용의 전체 목록은 아닙니다.

ASP.NET Core

ASP.NET Core 10의 주요 변경 사항을 참조하시오.

Containers

테이블 확장

제목	변경 유형
기본 .NET 이미지는 Ubuntu 사용합니다.	동작 변경

핵심 .NET 라이브러리

테이블 확장

제목	변경 유형
API 사용 중지	원본이 호환되지 않음

제목	변경 유형
ActivitySource.CreateActivity 및 ActivitySource.StartActivity 동작 변경	동작 변경
Arm64 SVE 비장애 로드에는 마스크가 필요합니다.	이진/소스 호환되지 않음
BufferedStream.WriteByte는 더 이상 암시적 플러시를 수행하지 않습니다.	동작 변경
스팬 매개 변수 을 사용한 C# 14 오버로드 해석	동작 변경
제네릭 수학의 일관된 시프트 동작	동작 변경
W3C 표준으로 업데이트된 기본 추적 컨텍스트 전파자	동작 변경
DriveInfo.DriveFormat은 Linux 파일 시스템 형식을 반환합니다.	동작 변경
DefaultValueAttribute ctor에서 DynamicallyAccessedMembers 주석 삭제됨	이진/소스 호환되지 않음
InlineArray에서 명시적 구조체 크기가 허용되지 않음	바이너리 비호환
FilePatternMatch.Stem이 nullable이 아닌 것으로 변경됨	소스 호환 불가/동작 변화
GnuTarEntry 및 PaxTarEntry는 기본적으로 atime 및 ctime을 더 이상 포함하지 않습니다.	동작 변경
LDAP DirectoryControl 구문 분석이 이제 더 엄격해졌습니다	동작 변경
MacCatalyst 버전 정규화	동작 변경
.NET 런타임은 더 이상 기본 종료 신호 처리기를 제공하지 않습니다.	동작 변경
System.Linq.AsyncEnumerable는 핵심 라이브러리에 포함되었습니다	원본이 호환되지 않음
Type.MakeGenericSignatureType 인수 유효성 검사	동작 변경

Cryptography

테이블 확장

제목	변경 유형
CompositeMLDsa가 draft-08로 업데이트됨	동작 변경
CoseSigner.Key는 null일 수 있습니다.	동작/소스 호환되지 않는 변경
MLDsa 및 SlhDsa 'SecretKey' 멤버 이름이 바뀌었습니다.	원본이 호환되지 않음

제목	변경 유형
OpenSSL 암호화 기본 형식은 macOS에서 지원되지 않습니다.	동작 변경
Unix에 OpenSSL 1.1.1 이상이 필요합니다.	동작 변경
X500DistinguishedName의 유효성 검사가 더 엄격해졌습니다	동작 변경
X509Certificate 및 PublicKey 키 매개 변수는 null일 수	동작/소스 호환되지 않는 변경
환경 변수 이름이 DOTNET_OPENSSL_VERSION_OVERRIDE로 변경되었습니다	동작 변경

Entity Framework Core (엔티티 프레임워크 코어)

EF Core 10의 주요 변경 사항을 참조하십시오.

Extensions

테이블 확장

제목	변경 유형
BackgroundService는 모든 ExecuteAsync를 작업으로 실행합니다.	동작 변경
AnyKey를 사용하여 GetKeyedService() 및 GetKeyedServices()의 문제 해결	동작 변경
구성에서 유지되는 Null 값	동작 변경
콘솔 로그 출력에서 메시지가 더 이상 중복되지 않음	동작 변경
ProviderAliasAttribute가 Microsoft.Extensions.Logging.Abstractions 어셈블리로 이동됨	원본이 호환되지 않음
Microsoft.Extensions.Configuration의 트리밍 안전하지 않은 코드에서 DynamicallyAccessedMembers 특성이 제거됨	바이너리 비호환

Globalization

테이블 확장

제목	변경 유형
환경 변수 이름이 DOTNET_ICU_VERSION_OVERRIDE로 변경되었습니다	동작 변경

설치 도구

테이블 확장

제목	변경 유형
VS Code용 dotnet.acquire API가 더 이상 항상 최신 다운로드되지 않음	동작 변경

Interop

테이블 확장

제목	변경 유형
IDispatchEx COM 개체를 IReflect로 캐스팅 실패	동작 변경
단일 파일 앱은 더 이상 실행 파일 디렉터리에서 네이티브 라이브러리를 찾지 않습니다.	동작 변경
DllImportSearchPath.AssemblyDirectory를 지정하면 어셈블리 디렉터리만 검색됩니다.	동작 변경

Networking

테이블 확장

제목	변경 유형
PublishTrimmed에서 기본적으로 사용하지 않도록 설정된 HTTP/3 지원	원본이 호환되지 않음
MailAddress는 연속된 점에 대한 유효성 검사를 적용합니다.	동작 변경
브라우저 HTTP 클라이언트에서 기본적으로 사용하도록 설정된 스트리밍 HTTP 응답	동작 변경
Uri 길이 제한 제거됨	동작 변경

Reflection

테이블 확장

제목	변경 유형	소개된 버전
InvokeMember/FindMembers/DeclaredMembers에서 더 제한된 주석	동작/소스 호환되지 않음	

SDK 및 MSBuild

테이블 확장

제목	변경 유형
.NET CLI --interactive 는 사용자 시나리오에서 기본값으로 true 설정됩니다.	동작 변경
dotnet CLI 명령은 명령과 관련이 없는 데이터를 stderr에 기록합니다.	동작 변경
.NET 도구 패키징은 RuntimeIdentifier 관련 도구 패키지를 만듭니다.	동작 변경
기본 워크로드 구성을 '느슨한 매니페스트'에서 '워크로드 세트' 모드로	동작 변경
코드 검사 EnableDynamicNativeInstrumentation 기본값은 false입니다.	동작 변경
dnx.ps1 파일은 더 이상 .NET SDK에 포함되지 않습니다.	원본이 호환되지 않음
dotnet new sln 기본값은 SLNX 파일 형식입니다.	동작 변경
dotnet package list 복원 수행	동작 변경
dotnet restore 전이적 패키지 감사	동작 변경
dotnet tool install --local 는 기본적으로 매니페스트를 만듭니다.	동작 변경
`dotnet watch`	동작 변경
project.json 지원되지 않음 dotnet restore	원본이 호환되지 않음
SHA-1 지문 지원이 중단됨 dotnet nuget sign	동작 변경
MSBUILDCUSTOMBUILDEVENTWARNING 우회 방법 삭제됨	동작 변경
MSBuild 사용자 지정 문화권 리소스 처리	동작 변경
NU1510은 NuGet에 의해 제거된 직접 참조에 대해 발생합니다.	원본이 호환되지 않음
런타임 자산이 없는 NuGet 패키지는 deps.json	원본이 호환되지 않음
버전이 없는 PackageReference에서 오류가 발생함	동작 변경
PrunePackageReference는 직접 정리 가능한 참조를 민영화합니다.	동작 변경

제목	변경 유형
HTTP 경고가 dotnet package list 및 dotnet package search에서 오류로 승격되었습니다	동작/소스 호환되지 않는 변경
NUGET_ENABLE_ENHANCED_HTTP_RETRY 환경 변수 제거됨	동작 변경
NuGet에서 잘못된 패키지 ID에 대한 오류를 기록합니다.	동작 변경
ToolCommandName 도구가 아닌 패키지에 대해 설정되지 않음	원본이 호환되지 않음

직렬화

테이블 확장

제목	변경 유형
System.Text.Json이 속성 이름 충돌을 확인합니다.	동작 변경
XmlSerializer는 더 이상 ObsoleteAttribute로 표시된 속성을 무시하지 않습니다.	동작 변경

Windows Forms

테이블 확장

제목	변경 유형
API 사용 중지	원본이 호환되지 않음
WPF와 WinForms를 모두 참조하는 애플리케이션은 MenuItem과 ContextMenu 형식을 명확하게 구분해야 합니다.	원본이 호환되지 않음
HtmlElement.InsertAdjacentElement의 매개변수 이름 변경	원본이 호환되지 않음
TreeView 확인란 이미지 잘림	동작 변경
StatusStrip은 기본적으로 System RenderMode를	동작 변경
System.Drawing OutOfMemoryException이 ExternalException으로 변경됨	동작 변경

WPF(Windows Presentation Foundation)

테이블 확장

제목	변경 유형
빈 ColumnDefinitions 및 RowDefinitions는 허용되지 않습니다.	원본이 호환되지 않음
DynamicResource를 잘못 사용하면 애플리케이션이 충돌합니다.	소스 호환 불가/동작 변화

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 29.

.NET 9의 새로운 기능

2025. 08. 26.

.NET 9의 새로운 기능에 대해 알아보고 추가 설명서에 대한 링크를 찾습니다.

.NET 8 후속인 .NET 9는 클라우드 네이티브 앱 및 성능에 특별히 중점을 두고 있습니다. STS(표준 기간 지원) 릴리스로 2년 동안 지원 됩니다. .NET 9를 다운로드할 수 있습니다.

.NET 9의 새로운 기능에 대해 엔지니어링 팀은 GitHub 토론에서 .NET 9 미리 보기 업데이트를 게시합니다. 이 곳에서 질문을 하고 릴리스에 대한 피드백을 제공할 수 있습니다.

.NET 런타임

.NET 9 런타임에는 트리밍 지원이 있는 기능 스위치에 대한 새 특성 모델이 포함되어 있습니다. 새 특성을 사용하면 라이브러리에서 기능 영역을 토글하는 데 사용할 수 있는 기능 스위치를 정의할 수 있습니다.

가비지 수집에는 기본적으로 서버 GC 대신 사용되는 *동적 적응 기능이 포함되어 있으며, 이는 애플리케이션 크기*에 맞추어 조정됩니다.

또한 런타임에는 루프 최적화, 인라인 처리, Arm64 벡터화 및 코드 생성을 비롯한 다양한 성능 향상이 포함됩니다.

자세한 내용은 .NET 9 런타임의 새로운 기능에 대해 알아보세요.

.NET 라이브러리

System.Text.Json는 nullable 참조 형식 주석의 지원을 추가하고, 형식에서 JSON 스키마를 내보내는 기능을 제공합니다. 기록된 JSON의 들여쓰기를 사용자 지정하고 단일 스트림에서 여러 루트 수준 JSON 값을 읽을 수 있는 새로운 옵션을 추가합니다.

LINQ에서 새 메서드는 CountByAggregateByGroupBy통해 중간 그룹을 할당할 필요 없이 키별로 상태를 집계할 수 있도록 합니다.

컬렉션 형식의 경우 System.Collections.Generic.PriorityQueue<TElement,TPriority> 형식에는 큐에 있는 항목의 우선 순위를 Remove(TElement, TElement, TPriority, IEqualityComparer<TElement>) 업데이트하는 데 사용할 수 있는 새 메서드가 포함됩니다.

암호화의 경우 .NET 9는 CryptographicOperations 형식에 새로운 원샷 해시 메서드를 추가합니다. 또한 KMAC 알고리즘을 사용하는 새 클래스를 추가합니다.

새 PersistedAssemblyBuilder 형식을 사용하면 리플렉션으로 내보낸 어셈블리를 저장할 수 있습니다. 이 새 클래스에는 PDB 지원도 포함되어 있습니다. 즉, 기호 정보를 내보내고 이를 사용하

여 생성된 어셈블리를 디버그할 수 있습니다.

TimeSpan 클래스에는 `From*`로부터 `TimeSpan` 객체를 생성할 수 있는 새로운 `int` 메서드가 포함되어 있으며, 이는 `double`대신 사용됩니다. 이러한 메서드는 부동 소수점 계산에서 내재된 부정확성으로 인한 오류를 방지하는 데 도움이 됩니다.

자세한 내용은 .NET 9 라이브러리새로운 기능을 참조하세요.

.NET SDK

.NET 9 SDK는 모든 워크로드가 명시적으로 업데이트될 때까지 단일 특정 버전으로 유지되는 *워크로드 집합*도입합니다. 도구의 경우 dotnet tool install 대한 새 옵션을 사용하면 도구 작성자가 아닌 사용자가 도구가 대상으로 하는 버전보다 최신 .NET 런타임 버전에서 도구를 실행할 수 있는지 여부를 결정할 수 있습니다. 또한:

- 단위 테스트에는 병렬로 테스트를 실행할 수 있는 더 나은 MSBuild 통합이 있습니다.
- 터미널 로거는 기본적으로 사용하도록 설정되며 유용성도 향상되었습니다. 예를 들어 이제 총 실패 및 경고 수가 빌드 끝에 요약됩니다.
- 새 MSBuild 스크립트 분석기("빌드 검사")를 사용할 수 있습니다.
- SDK는 .NET SDK와 MSBuild 간의 버전 불일치를 감지하고 조정할 수 있습니다.
- `dotnet workload history` 명령은 현재 .NET SDK 설치에 대한 워크로드 설치 및 수정의 기록을 보여 줍니다.

자세한 내용은 .NET 9용 SDK의 새로운 기능참조하세요.

AI 구성 요소

.NET 9는 Microsoft.Extensions.AI 및 Microsoft.Extensions.VectorData 패키지를 통해 C# 추상화의 통합 계층을 도입합니다. 이러한 추상화는 소형 및 대형 언어 모델(SLLM 및 LLM), 포함, 벡터 저장소 및 미들웨어를 포함하여 AI 서비스와의 상호 작용을 용이하게 합니다.

.NET 9에는 AI 기능을 확장하는 새로운 텐서 유형도 포함되어 있습니다. TensorPrimitives 및 새로운 Tensor<T> 형식은 다차원 데이터의 효율적인 인코딩, 조작 및 계산을 사용하도록 설정하여 AI 기능을 확장합니다. 이러한 형식은 System.Numerics.Tensors 패키지 최신 릴리스에서 찾을 수 있습니다.

TensorPrimitives

- 확장된 메서드 범위: 오버로드가 40개에서 거의 200개로 증가했으며, 이제 값 범위에 대한 숫자 연산을 포함합니다. 이는 `Math`, `MathF` 및 `INumber<T>`와 유사한 작업을 포함합니다.
- 성능 향상: 이제 성능 향상을 위해 많은 작업이 SIMD 최적화되었습니다.

- 제네릭 오버로드: 특정 인터페이스를 구현하는 모든 형식 `T`를 지원하며 .NET에서의 float 값의 범위를 넘어 확장합니다.

Tensor<T>

- 효율적인 수학 연산을 위해 `TensorPrimitives` 기반으로 합니다.
- 가능한 경우 0개의 복사본을 사용하여 AI 라이브러리(ML.NET, TorchSharp, ONNX 런타임)와의 효율적인 상호 운용성을 제공합니다.
- 인덱싱 및 조각화 작업을 통해 쉽고 효율적인 데이터 조작을 가능하게 합니다.
- .NET 9에서 는 실험적입니다.

ML.NET

ML.NET 사용자 지정 기계 학습 모델을 .NET 애플리케이션에 통합할 수 있는 오픈 소스 플랫폼 간 프레임워크입니다.

ML.NET 4.0은 다음과 같은 개선 사항을 제공합니다.

- 프로그래밍 방식으로 `MLContext` 옵션을 구성하는 새로운 방법입니다.
- ONNX 모델을 `Stream`에 로드합니다.
- 데이터 프레임이 개선되었습니다.
- 토큰 변환기에 대한 새로운 기능입니다.
- (실험적) 라마 및 피 모델 제품군의 TorchSharp 포트입니다.
- (실험적) CausalLM 파이프라인 API.

자세한 내용은 ML.NET의 새로운 기능을 참조하세요.

토큰화기

Microsoft.ML.Tokenizers 라이브러리는 .NET 개발자에게 토큰에 텍스트를 인코딩하고 디코딩하는 기능을 제공합니다. AI 시나리오의 경우 로컬 모델을 사용할 때 컨텍스트를 관리하고, 비용을 계산하고, 텍스트를 전처리하는 것이 중요합니다.

최신 릴리스에서는 토큰라이저에 대한 중요한 새로운 기능이 도입되었습니다.

- GPT (3, 3.5, 4, 4o, o1) 및 Llam3 모델을 위한 Tiktoken
- 라마 및 미스트랄 모델에 대한 라마(SentencePiece 기반)
- codegen-350M-mono와 같은 코드 생성 모델용 CodeGen
- Microsoft Phi2 모델용 Phi2(CodeGen 기반)
- WordPiece
- Bert(WordPiece 기반) - Bert 지원 모델(예: 최적--all-MiniLM-L6-v2)

.NET Aspire

.NET Aspire는 관찰 가능한 프로덕션 준비 앱을 빌드하기 위한 강력한 도구, 템플릿 및 패키지 집합입니다. .NET Aspire의 최신 릴리스에는 대시보드 및 리소스 수명 주기 관리가 개선되었습니다. 또한 개발 중에 더 많은 유연성을 위해 새로운 통합 및 API를 추가합니다. .NET Aspire 9는 .NET 9 및 .NET 8 앱 모두에서 작동합니다. 자세한 내용은 .NET Aspire 9에서 새로운 기능을 참조하세요.

ASP.NET Core

.NET 9로 빌드된 ASP.NET Core 앱은 기본적으로 안전하며, 미리 컴파일에 대한 지원을 확장했으며, 모니터링 및 추적이 향상되었습니다. 성능이 향상되면 더 높은 처리량과 더 빠른 시작 시간이 표시되며 메모리 사용량이 줄어듭니다. .NET 9의 ASP.NET Core에는 다음이 포함됩니다.

- 자동 지문 버전 관리로 빌드 및 게시 시간에 JavaScript 및 CSS와 같은 정적 파일의 최적화된 처리
- Blazor: 새로운 하이브리드 및 웹앱 템플릿, 구성 요소 렌더링 모드 검색, 서버 렌더링을 사용한 새로운 다시 연결 환경.
- API: `Microsoft.AspNetCore.OpenAPI`의 사용을 통해 OpenAPI 문서 생성에 대한 기본 지원과 향상된 네이티브 AOT 지원을 제공합니다.
- 인증 및 권한 부여를 위한 새 API를 사용하여 보안이 향상되었습니다.
- Linux에서 신뢰할 수 있는 개발 인증서를 더 쉽게 설정하여 개발 중에 HTTPS를 사용하도록 설정합니다.

다음은 .NET 9의 기능 및 향상된 기능 중 일부에 불과합니다. 자세한 내용은 ASP.NET Core 9.0 새로운 기능참조하세요.

.NET MAUI

.NET 9에서 .NET 다중 플랫폼 앱 UI(.NET MAUI)의 초점은 향상된 성능과 안정성, 데스크톱 및 모바일 애플리케이션에 대한 심층 통합입니다. .NET MAUI에는 iOS 및 Mac Catalyst를 위한 새로운 성능 향상된 CollectionView 및 CarouselView의 구현, 기존 컨트롤 업데이트, 새로운 앱 수명 주기 이벤트, 그리고 앱 크기 및 시작 시간을 개선하기 위한 네이티브 AOT 및 트리밍 기능 향상이 포함되어 있습니다. 또한:

- Windows에서 새로운 TitleBar 데스크톱 컨트롤을 사용할 수 있습니다.
- 새 HybridWebView 컨트롤을 사용하면 ReactJS, Vue.js및 Angular와 같은 프레임워크에서 JavaScript 사용 콘텐츠를 더 쉽게 포함할 수 있습니다.
- 이제 Entry 추가 키보드 모드를 지원합니다.
- 컨트롤 처리기는 가능할 경우 컨트롤과의 연결을 자동으로 해제합니다.

- MainPage Application.CreateWindow(IActivationState) 클래스를 재정의하여 앱의 기본 페이지를 설정하기 위해 더 이상 사용되지 않습니다.

이러한 새로운 기능에 대한 자세한 내용은 .NET 9.NET MAUI의 새로운 기능을 참조하세요.

EF Core

Entity Framework Core에는 Azure Cosmos DB for NoSQL용 데이터베이스 공급자에 대한 중요한 업데이트가 포함되어 있습니다. 또한 AOT 컴파일 및 미리 컴파일된 쿼리에 대한 단계와 함께, 기타 개선 사항도 포함되어 있습니다. 자세한 내용은 EF Core 9의 새로운 기능참조하세요.

C# 13

C# 13은 .NET 9 SDK와 함께 제공되며 다음과 같은 새로운 기능을 포함합니다.

- `params` 컬렉션
- 새 `lock` 형식 및 의미 체계
- 새 이스케이프 시퀀스 - `\e`
- 메서드 그룹 자연 형식 개선
- 개체 이니셜라이저의 암시적 인덱서 액세스
- 반복기 및 비동기 메서드에서 `ref` 로컬 및 `unsafe` 컨텍스트 사용
- `ref struct` 형식이 인터페이스를 구현할 수 있도록 허용
- 제네릭의 형식 매개 변수에 대한 인수로 ref 구조체 형식을 허용합니다.
- 이제 부분 속성 및 인덱서가 `partial` 형식에서 허용됩니다.
- 오버로드 확인 우선 순위를 사용하면 라이브러리 작성자가 하나의 오버로드를 다른 오버로드보다 더 나은 것으로 지정할 수 있습니다.

또한 C# 13에는 '`field` 지원 속성'이라는 미리 보기 기능이 추가됩니다.

자세한 내용은 C# 13의 새로운 기능참조하십시오.

F# 9

F# 9는 .NET 9 SDK와 함께 제공되며 다음과 같은 새로운 기능을 포함합니다.

- Nullable 참조 형식
- 차별 된 연합 . Is* 속성
- 부분 활성 패턴은 단위 옵션 대신 bool을 반환할 수 있습니다.
- 인수가 제공될 때 기본 속성에 확장 메서드를 사용하는 것이 좋습니다.
- 빈 본문 계산 표현식
- 해시 지시문은 문자열이 아닌 인수를 취할 수 있습니다.

- 확장된 #help 지시문으로 읽기-평가-출력 루프(REPL)에서 문서를 표시하도록 fsi를 조정합니다.
- #nowarn 오류 코드에서 FS 접두사를 지원하여 경고를 사용하지 않도록 설정하도록 허용
- 비재귀적 함수 또는 let 바인딩 값에 대한 TailCall 특성에 대한 경고
- 특성 대상 적용
- 컬렉션에 대한 임의 함수
- F# 목록 및 집합에 대한 C# 컬렉션 식 지원
- 다양한 개발자 생산성, 성능 및 도구 개선 사항

자세한 내용은 F# 9의 새로운 기능을 참조하세요.

Windows Presentation Foundation

.NET 9의 WPF는 다음과 같은 몇 가지 테마 개선 사항과 함께 최신 앱을 빌드하기 위한 향상된 지원을 제공합니다.

- Windows Fluent 테마에 대한 지원.
- Windows의 밝은 모드 및 어두운 모드에 대한 테마 지원이 추가되었습니다.
- 테마는 이제 Windows 강조 색을 지원합니다.
- 하이픈 기반 합자를 지원하도록 글꼴 렌더링이 향상되었습니다.
- `BinaryFormatter` 더 이상 지원되지 않습니다.

자세한 내용은 .NET 9용 WPF의 새로운 기능을 참조하세요.

Windows Forms

.NET 9의 WinForms는 새로운 테마, 비동기 개발을 위한 향상된 기능 등에 대한 지원을 제공합니다.

- `Form` 및 `TaskDialog`이 이제 `ShowDialogAsync`를 지원합니다. (실험적 기능)
- `BinaryFormatter` 더 이상 지원되지 않습니다.
- Windows에서 지원하는 어둡게 모드로 앱을 렌더링하기 위한 실험적 지원.
- `FolderBrowserDialog` 및 `ToolStrip` 몇 가지 사소한 개선이 있었습니다.
- **System.Drawing** 라이브러리에는 GDI+ 효과 래핑, `ReadOnlySpan`지원 및 더 나은 interop 코드 생성을 비롯한 많은 개선 사항이 있습니다.

자세한 내용은 .NET 9용 Windows Forms의 새로운 기능참조하세요.

참조

- .NET 9에 대한 비전 블로그 게시물

- ASP.NET Core 9.0의 새로운 기능
- .NET MAUI의 새로운 기능
- EF Core 새로운 기능

.NET 9 런타임의 새로운 기능

2025. 08. 22.

이 문서에서는 .NET 9용 .NET 런타임의 새로운 기능 및 성능 향상에 대해 설명합니다.

트리밍 지원이 있는 기능 스위치에 대한 특성 모델

두 가지 새로운 특성을 사용하면 .NET 라이브러리(및 사용자)가 기능 영역을 토글하는 데 사용할 수 있는 기능 스위치 를 정의할 수 있습니다. 기능이 지원되지 않는 경우 네이티브 AOT를 사용하여 트리밍하거나 컴파일할 때 지원되지 않는(따라서 사용되지 않는) 기능이 제거되어 앱 크기가 더 작게 유지됩니다.

- FeatureSwitchDefinitionAttribute 는 트리밍 시 기능 스위치 속성을 상수로 처리하는 데 사용되며 스위치로 보호되는 데드 코드는 제거할 수 있습니다.

  C#

  ```
  if (Feature.IsSupported)
      Feature.Implementation();

  public class Feature
  {
      [FeatureSwitchDefinition("Feature.IsSupported")]
      internal static bool IsSupported =>
  AppContext.TryGetSwitch("Feature.IsSupported", out bool isEnabled) ?
  isEnabled : true;

      internal static void Implementation() => ...;
  }
  ```

  앱이 프로젝트 파일 `Feature.IsSupported` 에서 다음 기능 설정으로 트리밍되면 로 처리 `false`되고 `Feature.Implementation` 코드가 제거됩니다.

  XML

  ```
  <ItemGroup>
    <RuntimeHostConfigurationOption Include="Feature.IsSupported" Value="false"
  Trim="true" />
  </ItemGroup>
  ```

- FeatureGuardAttribute는 기능 스위치 속성을 주석으로 주석이 추가된 RequiresUnreferencedCodeAttributeRequiresAssemblyFilesAttribute코드 또는 RequiresDynamicCodeAttribute. 다음은 그 예입니다.

C#

```
if (Feature.IsSupported)
    Feature.Implementation();

public class Feature
{
    [FeatureGuard(typeof(RequiresDynamicCodeAttribute))]
    internal static bool IsSupported =>
RuntimeFeature.IsDynamicCodeSupported;

    [RequiresDynamicCode("Feature requires dynamic code support.")]
    internal static void Implementation() => ...; // Uses dynamic code
}
```

빌드 `<PublishAot>true</PublishAot>`된 경우 호출 `Feature.Implementation()` 은 분석기 경고 IL3050을 생성하지 않으며 `Feature.Implementation` 게시할 때 코드가 제거됩니다.

UnsafeAccessorAttribute는 제네릭 매개 변수를 지원합니다.

이 UnsafeAccessorAttribute 기능을 사용하면 호출자에 액세스할 수 없는 형식 멤버에 안전하지 않은 액세스를 허용합니다. 이 기능은 .NET 8에서 설계되었지만 제네릭 매개 변수를 지원하지 않고 구현되었습니다. .NET 9는 CoreCLR 및 네이티브 AOT 시나리오에 대한 제네릭 매개 변수에 대한 지원을 추가합니다. 다음 코드에서는 사용 예제를 보여 줍니다.

C#

```
using System.Runtime.CompilerServices;

public class Class<T>
{
    private T? _field;
    private void M<U>(T t, U u) { }
}

class Accessors<V>
{
    [UnsafeAccessor(UnsafeAccessorKind.Field, Name = "_field")]
    public extern static ref V GetSetPrivateField(Class<V> c);

    [UnsafeAccessor(UnsafeAccessorKind.Method, Name = "M")]
    public extern static void CallM<W>(Class<V> c, V v, W w);
}

internal class UnsafeAccessorExample
{
    public void AccessGenericType(Class<int> c)
```

```
    {
        ref int f = ref Accessors<int>.GetSetPrivateField(c);

        Accessors<int>.CallM<string>(c, 1, string.Empty);
    }
}
```

가비지 수집

이제 데이터(애플리케이션 크기)에 대한 동적 적응이 기본적으로 사용하도록 설정됩니다. 애플리케이션 메모리 요구 사항에 맞게 조정하는 것을 목표로 합니다. 즉, 애플리케이션 힙 크기는 수명이 긴 데이터 크기에 거의 비례해야 합니다. DATAS는 .NET 8에서 옵트인 기능으로 도입되었으며 .NET 9에서 크게 업데이트되고 개선되었습니다.

자세한 내용은 애플리케이션 크기에 대한 동적 적응(DATAS)을 참조하세요.

제어 흐름 적용 기술

CET(제어 흐름 적용 기술) 는 Windows의 앱에 대해 기본적으로 사용하도록 설정 됩니다. ROP(반환 지향 프로그래밍) 악용에 대해 하드웨어 적용 스택 보호를 추가하여 보안을 크게 향상시킵니다. 최신 .NET 런타임 보안 완화입니다↗.

CET는 CET 사용 프로세스에 몇 가지 제한 사항을 적용하며 성능이 저하될 수 있습니다. CET를 옵트아웃하는 다양한 컨트롤이 있습니다.

.NET 설치 검색 동작

.NET 앱은 이제 .NET 런타임을 검색하는 방법에 대해 구성할 수 있습니다. 이 기능은 프라이빗 런타임 설치와 함께 사용하거나 실행 환경을 보다 강력하게 제어할 수 있습니다.

성능 향상

.NET 9에 대해 다음과 같은 성능이 향상되었습니다.

- 루프 최적화
- 향상된 인라인 기능
- PGO 개선 사항: 형식 검사 및 캐스트
- .NET 라이브러리의 Arm64 벡터화
- Arm64 코드 생성
- 빠른 예외
- 코드 레이아웃

- 주소 노출 감소
- AVX10v1 지원
- 하드웨어 내장 코드 생성
- 부동 소수점 및 SIMD 작업에 대한 상수 접기
- Arm64 SVE 지원
- 상자에 대한 개체 스택 할당

루프 최적화

루프에 대한 코드 생성을 개선하는 것이 .NET 9의 우선 순위입니다. 이제 다음과 같은 개선 사항을 사용할 수 있습니다.

- 유도 변수 확대
- 인덱싱 후 주소 지정
- 강도 감소
- 루프 카운터 변수 방향

> ⓘ **참고**
>
> 유도 변수 확대 및 인덱싱 후 주소 지정은 유사합니다. 둘 다 루프 인덱스 변수를 사용하여 메모리 액세스를 최적화합니다. 그러나 Arm64는 CPU 기능을 제공하고 x64는 그렇지 않으므로 다른 접근 방식을 사용합니다. CPU/ISA 기능 및 요구 사항의 차이로 인해 x64에 대해 유도 변수 확대가 구현되었습니다.

유도 변수 확대

64비트 컴파일러는 *IV(유도 변수) 확대*라는 새로운 최적화 기능을 제공합니다.

IV는 포함하는 루프가 반복될 때 값이 변경되는 변수입니다. 다음 `for` 루프 `i` 에서는 IV `for (int i = 0; i < 10; i++)` 입니다. 컴파일러가 루프 반복을 통해 IV 값이 어떻게 진화하는지 분석할 수 있는 경우 관련 식에 대해 더 성능이 뛰어난 코드를 생성할 수 있습니다.

배열을 반복하는 다음 예제를 살펴보겠습니다.

C#

```
static int Sum(int[] nums)
{
    int sum = 0;
    for (int i = 0; i < nums.Length; i++)
    {
        sum += nums[i];
    }
```

```
    return sum;
}
```

인덱스 변수 `i`는 크기가 4바이트입니다. 어셈블리 수준에서 64비트 레지스터는 일반적으로 x64에 배열 인덱스를 보유하는 데 사용되며, 이전 .NET 버전에서는 컴파일러가 배열 액세스를 위해 8바이트까지 0으로 확장 `i` 되지만 다른 곳에서는 4바이트 정수로 계속 처리 `i` 되는 코드를 생성했습니다. 그러나 8바이트까지 확장 `i` 하려면 x64에 대한 추가 지침이 필요합니다. IV 확대를 사용하면 이제 64비트 JIT 컴파일러가 루프 전체에서 8바이트로 확장 `i` 되어 확장이 0이 생략됩니다. 배열을 반복하는 것은 매우 일반적이며 이 명령 제거의 이점은 빠르게 추가됩니다.

Arm64에서 인덱싱된 후 주소 지정

인덱스 변수는 메모리의 순차 영역을 읽는 데 자주 사용됩니다. idiomatic `for` 루프를 고려합니다.

C#

```
static int Sum(int[] nums)
{
    int sum = 0;
    for (int i = 0; i < nums.Length; i++)
    {
        sum += nums[i];
    }

    return sum;
}
```

루프의 각 반복에 대해 인덱스 변수 `i` 를 사용하여 정수(`nums` in)를 읽은 다음 `i` 증가합니다. Arm64 어셈블리에서 이러한 두 작업은 다음과 같습니다.

AL

```
ldr w0, [x1]
add x1, x1, #4
```

`ldr w0, [x1]` 에 메모리 주소 `x1` `w0` 의 정수가 로드됩니다. 이는 소스 코드의 `nums[i]` 액세스에 해당합니다. `add x1, x1, #4` 그런 다음 주소를 `x1` 4바이트(정수 크기)로 늘려 다음 정 `nums` 수로 이동합니다. 이 명령은 각 반복이 `i++` 끝날 때 실행되는 작업에 해당합니다.

Arm64는 인덱싱된 후 주소 지정을 지원하며, 주소가 사용된 후 "인덱스" 레지스터가 자동으로 증가합니다. 즉, 두 명령이 하나로 결합되어 루프의 효율성을 높일 수 있습니다. CPU는 두 명령 대신 하나의 명령만 디코딩하면 되며 루프의 코드는 이제 캐시 친화적입니다.

업데이트된 어셈블리의 모양은 다음과 같습니다.

```
AL

ldr w0, [x1], #0x04
```

`#0x04` 끝에 있는 주소 `x1` 는 정`w0`수로 로드하는 데 사용된 후 4바이트로 증가한다는 것을 의미합니다. 이제 64비트 컴파일러는 Arm64 코드를 생성할 때 인덱싱 후 주소 지정을 사용합니다.

강도 감소

강도 감소는 작업이 더 빠르고 논리적으로 동등한 작업으로 대체되는 컴파일러 최적화입니다. 이 기술은 루프를 최적화하는 데 특히 유용합니다. idiomatic `for` 루프를 고려합니다.

```
C#

static int Sum(int[] nums)
{
    int sum = 0;
    for (int i = 0; i < nums.Length; i++)
    {
        sum += nums[i];
    }

    return sum;
}
```

다음 x64 어셈블리 코드는 루프 본문에 대해 생성된 코드 조각을 보여 줍니다.

```
AL

add ecx, dword ptr [rax+4*rdx+0x10]
inc edx
```

이러한 명령은 식과 `i++`각각에 해당합니다`sum += nums[i]`. `rcx` (`ecx` 이 레지스터의 하위 32비트 보유)에는 값 `sum`이 포함되고, `rax` 기본 주소 `nums`가 포함되며 `rdx` , 값 `i`이 포함됩니다. 주소를 `nums[i]`계산하기 위해 인덱 `rdx` 스의 4(정수 크기)를 **곱** 합니다. 그러면 이 오프셋이 기본 주소에 `rax`추가되고 일부 안쪽 여백이 **추가**됩니다. (읽은 정 `nums[i]` 수가 읽은 후에는 인 `rcx` 덱 `rdx` 스가 증분됩니다.) 즉, 각 배열 액세스에는 곱하기 *및* 추가 작업이 필요합니다.

곱셈은 추가보다 더 비싸며 전자를 후자로 교체하는 것은 강도 감소를위한 고전적인 동기 부여입니다. 각 메모리 액세스에서 요소 주소 계산을 방지하려면 인덱스 변수가 아닌 포인터를 사용하여 정수에 `nums` 액세스하도록 예제를 다시 작성할 수 있습니다.

C#

```
static int Sum2(Span<int> nums)
{
    int sum = 0;
    ref int p = ref MemoryMarshal.GetReference(nums);
    ref int end = ref Unsafe.Add(ref p, nums.Length);
    while (Unsafe.IsAddressLessThan(ref p, ref end))
    {
        sum += p;
        p = ref Unsafe.Add(ref p, 1);
    }

    return sum;
}
```

소스 코드는 더 복잡하지만 초기 구현과 논리적으로 동일합니다. 또한 어셈블리가 더 잘 보입니다.

AL

```
add ecx, dword ptr [rdx]
add rdx, 4
```

`rcx` (`ecx` 이 레지스터의 하위 32비트 보유)는 여전히 값을 `sum`보유하지만 `rdx` 이제는 가리키는 `p`주소를 보유하므로 요소에 `nums` 액세스하려면 역참조 `rdx`해야 합니다. 첫 번째 예제의 모든 곱하기 및 추가는 포인터를 앞으로 이동하는 단일 `add` 명령으로 대체되었습니다.

.NET 9에서 JIT *컴파일러는 코드를* 다시 작성할 필요 없이 첫 번째 인덱싱 패턴을 자동으로 두 번째 인덱싱 패턴으로 변환합니다.

루프 카운터 변수 방향

이제 64비트 컴파일러는 루프의 카운터 변수가 반복 횟수를 제어하는 데만 사용될 때를 인식하고 루프를 최대값 대신 카운트다운하도록 변환합니다.

idiomatic `for (int i = ...)` 패턴에서 카운터 변수는 일반적으로 증가합니다. 다음 예시를 참조하세요.

C#

```
for (int i = 0; i < 100; i++)
{
    DoSomething();
}
```

그러나 많은 아키텍처에서 다음과 같이 루프의 카운터를 감소하는 것이 더 성능이 높습니다.

```C#
for (int i = 100; i > 0; i--)
{
    DoSomething();
}
```

첫 번째 예제에서는 컴파일러가 증가 `i`에 대한 명령을 내보낸 다음 비교를 수행하는 `i < 100` 명령과 조건이 여전히 `true`있는 경우 루프를 계속하기 위한 조건부 점프를 수행해야 합니다. 이는 총 세 가지 지침입니다. 그러나 카운터의 방향이 대칭 이동되면 한 가지 명령이 덜 필요합니다. 예를 들어 x64에서 컴파일러는 명령을 사용하여 `dec` 감소`i`할 수 있습니다. 0 `dec` 에 도달하면 `i` 명령이 바로 다음 점프 명령의 조건으로 사용할 수 있는 CPU 플래그를 `dec`설정합니다.

코드 크기 감소는 작지만 반복 수가 아닌 반복에 대해 루프가 실행되는 경우 성능이 크게 향상될 수 있습니다.

인라인 최적화

중 하나입니다. JIT 컴파일러의 인라인 처리에 대한 NET의 목표는 메서드가 가능한 한 많이 인라인되지 않도록 차단하는 제한을 제거하는 것입니다. .NET 9를 사용하면 다음을 인라인 처리할 수 있습니다.

- 런타임 조회가 필요한 공유 제네릭입니다.

  예를 들어 다음 메서드를 고려합니다.

  ```C#
  static bool Test<T>() => Callee<T>();
  static bool Callee<T>() => typeof(T) == typeof(int);
  ```

  참조 형식과 같은 `string`경우 `T` 런타임은 특수 인스턴스화 `Test` 이며 `Callee` 모든 ref 형식 형식 `T` 에서 공유하는 *공유 제네릭*을 만듭니다. 이 작업을 위해 런타임은 제네릭 형식을 내부 형식에 매핑하는 사전을 빌드합니다. 이러한 사전은 제네릭 형식(또는 제네릭 메서드 당)에 따라 특수화되며 런타임에 액세스하여 의존하는 `T`형식에 대한 `T` 정보를 가져옵니다. 지금까지 Just-In-Time으로 컴파일된 코드는 루트 메서드의 사전에 대해 이러한 런타임 조회만 수행할 수 있었습니다. 즉, 두 메서드가 동일한 형식으로 `Test`인스턴스화되었음에도 불구하고 인라인 `Callee` 코드에서 `Callee` 적절한 사전에 액세스할 수 있는 방법이 없었습니다.

.NET 9는 호출자에서 런타임 형식 조회를 자유롭게 사용하도록 설정하여 이 제한을 해제했습니다. 즉, JIT 컴파일러에서 다음과 같은 `Callee` 인라인 메서드를 `Test` 사용할 수 있습니다.

다른 메서드에서 호출 `Test<string>` 한다고 가정합니다. 의사 코드에서 인라인은 다음과 같습니다.

```
C#
```

```
static bool Test<string>() => typeof(string) == typeof(int);
```

컴파일 중에 해당 형식 검사를 계산할 수 있으므로 최종 코드는 다음과 같습니다.

```
C#
```

```
static bool Test<string>() => false;
```

JIT 컴파일러의 인라인 기능을 개선하면 다른 인라인 결정에 복합적인 영향을 미칠 수 있으므로 성능이 크게 향상됩니다. 예를 들어 인라인 `Callee` 으로 결정하면 호출 `Test<string>` 도 인라인 처리할 수 있습니다. 이로 인해 수백 개의 벤치마크 개선이 발생했으며, 최소 80개의 벤치마크가 10개 이상의% 개선되었습니다.

- Windows x64, Linux x64 및 Linux Arm64에서 *스레드 로컬 정적* 에 액세스합니다.

  클래스 멤버의 경우 `static` 멤버를 "공유"하는 클래스의 모든 인스턴스에 정확히 하나의 멤버 인스턴스가 있습니다. 멤버의 `static` 값이 각 스레드에 고유한 경우 해당 값을 스레드 로컬로 만들면 포함하는 스레드에서 멤버에 안전하게 액세스 `static` 하기 위한 동시성 기본 형식이 필요하지 않으므로 성능이 향상될 수 있습니다.

  이전에는 네이티브 AOT 컴파일 프로그램에서 스레드 로컬 정적에 액세스하려면 컴파일러가 스레드 로컬 스토리지의 기본 주소를 가져오기 위해 런타임에 호출을 내보내야 했습니다. 이제 컴파일러가 이러한 호출을 인라인 처리하여 이 데이터에 액세스하는 지침이 훨씬 줄어듭니다.

PGO 개선 사항: 형식 검사 및 캐스트

.NET 8은 기본적으로 동적 프로필 기반 최적화(PGO) 를 사용하도록 설정했습니다. NET 9는 JIT 컴파일러의 PGO 구현을 확장하여 더 많은 코드 패턴을 프로파일러합니다. 계층화된 컴파일을 사용하도록 설정하면 JIT 컴파일러가 이미 프로그램에 계측을 삽입하여 동작을 프로파일러합니다. 최적화를 사용하여 다시 컴파일하는 경우 컴파일러는 런타임에 빌드한 프로필을 활용하여 프로그램의 현재 실행과 관련된 결정을 내립니다. .NET 9에서 JIT 컴파일러는 PGO 데이터를 사용하여 *형식 검사*의 성능을 향상시킵니다.

개체의 형식을 확인하려면 런타임에 대한 호출이 필요하며 성능 저하가 발생합니다. 개체의 형식을 확인해야 하는 경우 JIT 컴파일러는 정확성을 위해 이 호출을 내보낸다(컴파일러는 일반적으로 불가능해 보이는 경우에도 가능성을 배제할 수 없음). 그러나 PGO 데이터에 개체가 특정 형식일 가능성이 있다고 제안하면 JIT 컴파일러는 이제 해당 형식을 저렴하게 확인하는 *빠른 경로*를 내보내고 필요한 경우에만 런타임으로 호출하는 느린 경로로 대체됩니다.

.NET 라이브러리의 Arm64 벡터화

새 `EncodeToUtf8` 구현은 ARM64에서 다중 레지스터 부하/저장 지침을 내보내는 JIT 컴파일러의 기능을 활용합니다. 이 동작을 사용하면 프로그램에서 더 적은 명령으로 더 큰 데이터 청크를 처리할 수 있습니다. 다양한 도메인의 .NET 앱은 이러한 기능을 지원하는 Arm64 하드웨어의 처리량 향상을 확인해야 합니다. 일부 벤치마크는 실행 시간을 절반 이상 줄였습니다.

Arm64 코드 생성

JIT 컴파일러에는 Arm64의 명령(값 로드용)을 사용하기 `ldp` 위해 연속 부하의 표현을 변환하는 기능이 이미 있습니다. .NET 9는 이 기능을 확장하여 작업을 *저장* 합니다.

명령은 `str` 단일 레지스터에서 메모리로 데이터를 저장하는 반면 `stp` , 명령은 한 *쌍* 의 레지스터에서 데이터를 저장합니다. 대신 사용하는 `stp` 것은 동일한 작업을 더 적은 수의 `str` 저장소 작업으로 수행할 수 있으므로 실행 시간이 향상됩니다. 한 명령의 면도는 약간 개선된 것처럼 보일 수 있지만, 코드가 반복 횟수가 아닌 루프에서 실행되는 경우 성능 향상이 빠르게 추가될 수 있습니다.

예를 들어 다음 코드 조각을 고려합니다.

C#

```
class Body { public double x, y, z, vx, vy, vz, mass; }

static void Advance(double dt, Body[] bodies)
{
    foreach (Body b in bodies)
    {
        b.x += dt * b.vx;
        b.y += dt * b.vy;
        b.z += dt * b.vz;
    }
}
```

의 `b.y` 값 `b.x` 이며 `b.z` 루프 본문에서 업데이트됩니다. 어셈블리 수준에서 각 멤버는 명령으로 `str` 저장할 수 있습니다. 또는 사용 `stp` 시 두 개의 저장소(`b.x` 및 `b.y` `b.y` `b.z` 메모리에서 이러한 쌍이 연속적이기 때문에)를 하나의 명령으로 처리할 수 있습니다. 명령을 사용하여 `stp` 동시에

저장 `b.x` `b.y` 하려면 컴파일러에서도 계산 `b.x + (dt * b.vx)` `b.y + (dt * b.vy)` 이 서로 독립적이며 저장 `b.x` `b.y`하기 전에 수행할 수 있는지 확인해야 합니다.

빠른 예외

CoreCLR 런타임은 예외 처리의 성능을 향상시키는 새로운 예외 처리 방법을 채택했습니다. 새 구현은 NativeAOT 런타임의 예외 처리 모델을 기반으로 합니다. 이 변경으로 인해 Unix에서 Windows SEH(구조적 예외 처리) 및 해당 에뮬레이션에 대한 지원이 제거됩니다. 새 방법은 Windows x86(32비트)을 제외한 모든 환경에서 지원됩니다.

새로운 예외 처리 구현은 마이크로 벤치마크를 처리하는 일부 예외에 따라 2-4배 더 빠릅니다. 성능 향상은 성능 랩에서 측정되었습니다.

- Windows x64: https://github.com/dotnet/perf-autofiling-issues/issues/32280
- Windows Arm64: https://github.com/dotnet/perf-autofiling-issues/issues/32016
- Linux x64: https://github.com/dotnet/perf-autofiling-issues/issues/31367
- Linux Arm64: https://github.com/dotnet/perf-autofiling-issues/issues/31631

새 구현은 기본적으로 사용하도록 설정됩니다. 그러나 레거시 예외 처리 동작으로 다시 전환해야 하는 경우 다음 방법 중 하나를 수행하면 됩니다.

- `true` 파일로 runtimeconfig.json설정합니다`System.Runtime.LegacyExceptionHandling`.
- 환경 변수를 `DOTNET_LegacyExceptionHandling` .로 `1`설정합니다.

코드 레이아웃

컴파일러는 일반적으로 각 블록이 첫 번째 명령에서만 입력되고 마지막 명령을 통해 종료될 수 있는 코드 청크인 기본 *블록을* 사용하는 프로그램의 제어 흐름을 추론합니다. 기본 블록의 순서가 중요합니다. 블록이 분기 명령으로 끝나는 경우 제어 흐름은 다른 블록으로 전송됩니다. 블록 다시 정렬의 한 가지 목표는 *대체* 동작을 최대화하여 생성된 코드의 분기 명령 수를 줄이는 것입니다. 각 기본 블록 다음에 가장 가능성이 큰 후속 블록이 뒤따를 경우 점프 없이 후속 블록에 "빠질" 수 있습니다.

최근까지 JIT 컴파일러의 블록 재주문은 흐름 그래프 구현에 의해 제한되었습니다. .NET 9에서는 JIT 컴파일러의 블록 다시 정렬 알고리즘이 더 간단하고 전역적인 접근 방식으로 대체되었습니다. 흐름 그래프 데이터 구조가 다음과 같이 리팩터링되었습니다.

- 블록 순서 지정과 관련된 몇 가지 제한을 제거합니다.
- 블록 간의 모든 제어 흐름 변경으로 실행 가능성이 높아집니다.

또한 프로필 데이터는 메서드의 흐름 그래프가 변환될 때 전파되고 유지 관리됩니다.

주소 노출 감소

.NET 9에서 JIT 컴파일러는 지역 변수 주소의 사용량을 더 잘 추적하고 불필요한 *주소 노출*을 방지할 수 있습니다.

지역 변수의 주소를 사용하는 경우 JIT 컴파일러는 메서드를 최적화할 때 추가적인 예방 조치를 취해야 합니다. 예를 들어 컴파일러가 다른 메서드에 대한 호출에서 지역 변수의 주소를 전달하는 메서드를 최적화한다고 가정합니다. 호출 수신자는 주소를 사용하여 지역 변수에 액세스할 수 있으므로 정확성을 유지하기 위해 컴파일러는 변수 변환을 방지합니다. 주소가 노출된 로컬은 컴파일러의 최적화 가능성을 크게 억제할 수 있습니다.

AVX10v1 지원

INTEL의 새로운 SIMD 명령 집합인 AVX10 에 대한 새 API가 추가되었습니다. 새 Avx10v1 API를 사용하여 벡터화된 작업으로 AVX10 지원 하드웨어에서 .NET 애플리케이션을 가속화할 수 있습니다.

하드웨어 내장 코드 생성

많은 하드웨어 내장 API는 사용자가 특정 매개 변수에 대한 상수 값을 전달할 것으로 예상합니다. 이러한 상수는 레지스터에 로드되거나 메모리에서 액세스되는 대신 내장 명령으로 직접 인코딩됩니다. 상수가 제공되지 않으면 내장 함수가 기능적으로 동일하지만 속도가 느린 대체(fallback) 구현에 대한 호출로 바뀝니다.

다음 예시를 참조하세요.

C#

```
static byte Test1()
{
    Vector128<byte> v = Vector128<byte>.Zero;
    const byte size = 1;
    v = Sse2.ShiftRightLogical128BitLane(v, size);
    return Sse41.Extract(v, 0);
}
```

호출 `Sse2.ShiftRightLogical128BitLane` 에서의 사용 `size` 은 상수 1로 대체될 수 있으며 정상적인 상황에서 JIT 컴파일러는 이미 이 대체 최적화를 수행할 수 있습니다. 그러나 컴파일러는 가속 코드 또는 대체 코드를 `Sse2.ShiftRightLogical128BitLane`생성할지 여부를 결정할 때 변수가 상수 대신 전달되고 있음을 감지하고 호출을 "내장"하도록 조기에 결정합니다. .NET 9부터 컴파일러는 이와 같은 더 많은 사례를 인식하고 변수 인수를 상수 값으로 대체하여 가속 코드를 생성합니다.

부동 소수점 및 SIMD 작업에 대한 상수 접기

*상수 접*기는 JIT 컴파일러의 기존 최적화입니다. *상수 접기는* 컴파일 시간에 계산할 수 있는 식을 계산할 수 있는 상수로 대체하여 런타임에 계산을 제거하는 것을 의미합니다. .NET 9는 다음과 같은 새로운 상수 접기 기능을 추가합니다.

- 부동 소수점 이진 작업의 경우 피연산자 중 하나가 상수입니다.
  - `x + NaN`가 이제 접힌 경우 `NaN`
  - `x * 1.0`가 이제 접힌 경우 `x`
  - `x + -0`가 이제 접힌 경우 `x`
- 하드웨어 내장 함수의 경우 예를 들어 다음을 가정 `x` 합니다.`Vector<T>`
  - `x + Vector<T>.Zero`가 이제 접힌 경우 `x`
  - `x & Vector<T>.Zero`가 이제 접힌 경우 `Vector<T>.Zero`
  - `x & Vector<T>.AllBitsSet`가 이제 접힌 경우 `x`

Arm64 SVE 지원

.NET 9에서는 ARM64 CPU에 대한 SIMD 명령 집합인 SVE(확장 가능 벡터 확장)에 대한 실험적 지원을 도입했습니다. .NET은 이미 NEON 명령 집합을 지원하므로 NEON 지원 하드웨어에서 애플리케이션은 128비트 벡터 레지스터를 활용할 수 있습니다. SVE는 최대 2048비트까지 유연한 벡터 길이를 지원하여 명령당 더 많은 데이터 처리의 잠금을 해제합니다. .NET 9 Vector<T>에서는 SVE를 대상으로 할 때 너비가 128비트이며, 향후 작업을 통해 대상 컴퓨터의 벡터 레지스터 크기와 일치하도록 너비를 확장할 수 있습니다. 새 System.Runtime.Intrinsics.Arm.Sve API를 사용하여 SVE 지원 하드웨어에서 .NET 애플리케이션을 가속화할 수 있습니다.

> **참고**
>
> .NET 9의 SVE 지원은 실험적입니다. 아래 `System.Runtime.Intrinsics.Arm.Sve` 의 API는 다음 ExperimentalAttribute릴리스에서 변경될 수 있음을 의미합니다. 또한 SVE 생성 코드를 통한 디버거 단계별 실행 및 중단점이 제대로 작동하지 않아 애플리케이션이 충돌하거나 데이터가 손상될 수 있습니다.

상자에 대한 개체 스택 할당

값 형식(예: `int` 및 `struct`)은 일반적으로 힙 대신 스택에 할당됩니다. 그러나 다양한 코드 패턴을 사용하도록 설정하기 위해 개체에 자주 "boxed"됩니다.

다음 코드 조각을 고려합니다.

```
C#
```

```
static bool Compare(object? x, object? y)
{
    if ((x == null) || (y == null))
    {
        return x == y;
    }

    return x.Equals(y);
}

public static int RunIt()
{
    bool result = Compare(3, 4);
    return result ? 0 : 100;
}
```

`Compare` 는 문자열이나 `double` 값과 같은 다른 형식을 비교하려는 경우 동일한 구현을 다시 사용할 수 있도록 편리하게 작성됩니다. 그러나 이 예제에서는 전달된 모든 값 형식을 *boxed*로 요구하는 성능 단점도 있습니다.

생성된 `RunIt` x64 어셈블리 코드는 다음과 같습니다.

AL

```
push     rbx
sub      rsp, 32
mov      rcx, 0x7FFB9F8074D0      ; System.Int32
call     CORINFO_HELP_NEWSFAST
mov      rbx, rax
mov      dword ptr [rbx+0x08], 3
mov      rcx, 0x7FFB9F8074D0      ; System.Int32
call     CORINFO_HELP_NEWSFAST
mov      dword ptr [rax+0x08], 4
add      rbx, 8
mov      ecx, dword ptr [rbx]
cmp      ecx, dword ptr [rax+0x08]
sete     al
movzx    rax, al
xor      ecx, ecx
mov      edx, 100
test     eax, eax
mov      eax, edx
cmovne   eax, ecx
add      rsp, 32
pop      rbx
ret
```

호출 `CORINFO_HELP_NEWSFAST` 은 boxed 정수 인수에 대한 힙 할당입니다. 또한 호출 `Compare`이 없습니다. 컴파일러가 인라인으로 `RunIt`하기로 결정했습니다. 이 인라인은 상자가 결코 "이스케

이프"되지 않는 것을 의미합니다. 즉, 실행 `Compare` 내내 정수는 알고 `x` `y` 실제로 정수이며 비교 논리에 영향을 주지 않고 안전하게 언박싱할 수 있습니다.

.NET 9부터 64비트 컴파일러는 스택에 이스케이프되지 않은 상자를 할당하여 다른 여러 최적화의 잠금을 해제합니다. 이 예제에서 컴파일러는 이제 힙 할당을 생략하지만 3과 `y` 4를 알고 `x` 있기 때문에 본문을 `Compare` 생략할 수도 있습니다 `x.Equals(y)`. 컴파일러는 컴파일 타임 `RunIt` 에 false이므로 항상 100을 반환해야 합니다. 업데이트된 어셈블리는 다음과 같습니다.

```
AL
```

```
mov      eax, 100
ret
```

What's new in .NET libraries for .NET 9

아티클 • 2024. 11. 11.

This article describes new features in the .NET libraries for .NET 9.

Base64Url

Base64 is an encoding scheme that translates arbitrary bytes into text composed of a specific set of 64 characters. It's a common approach for transferring data and has long been supported via a variety of methods, such as with Convert.ToBase64String or Base64.DecodeFromUtf8(ReadOnlySpan<Byte>, Span<Byte>, Int32, Int32, Boolean). However, some of the characters it uses makes it less than ideal for use in some circumstances you might otherwise want to use it, such as in query strings. In particular, the 64 characters that comprise the Base64 table include '+' and '/', both of which have their own meaning in URLs. This led to the creation of the Base64Url scheme, which is similar to Base64 but uses a slightly different set of characters that makes it appropriate for use in URLs contexts. .NET 9 includes the new Base64Url class, which provides many helpful and optimized methods for encoding and decoding with `Base64Url` to and from a variety of data types.

The following example demonstrates using the new class.

C#

```
ReadOnlySpan<byte> bytes = ...;
string encoded = Base64Url.EncodeToString(bytes);
```

BinaryFormatter

.NET 9 removes BinaryFormatter from the .NET runtime. The APIs are still present, but their implementations always throw an exception, regardless of project type. For more information about the removal and your options if you're affected, see BinaryFormatter migration guide.

Collections

The collection types in .NET gain the following updates for .NET 9:

- Collection lookups with spans
- OrderedDictionary<TKey, TValue>
- PriorityQueue.Remove() method lets you update the priority of an item in the queue.
- ReadOnlySet<T>

Collection lookups with spans

In high-performance code, spans are often used to avoid allocating strings unnecessarily, and lookup tables with types like Dictionary<TKey,TValue> and HashSet<T> are frequently used as caches. However, there has been no safe, built-in mechanism for doing lookups on these collection types with spans. With the new `allows ref struct` feature in C# 13 and new features on these collection types in .NET 9, it's now possible to perform these kinds of lookups.

The following example demonstrates using Dictionary<TKey,TValue>.GetAlternateLookup.

C#

```
private static Dictionary<string, int> CountWords(ReadOnlySpan<char> input)
{
    Dictionary<string, int> wordCounts = new(StringComparer.OrdinalIgnoreCase);
    Dictionary<string, int>.AlternateLookup<ReadOnlySpan<char>> spanLookup =
        wordCounts.GetAlternateLookup<ReadOnlySpan<char>>();

    foreach (Range wordRange in Regex.EnumerateSplits(input, @"\b\w+\b"))
    {
        ReadOnlySpan<char> word = input[wordRange];
        spanLookup[word] = spanLookup.TryGetValue(word, out int count) ? count + 1
: 1;
    }

    return wordCounts;
}
```

`OrderedDictionary<TKey, TValue>`

In many scenarios, you might want to store key-value pairs in a way where order can be maintained (a list of key-value pairs) but where fast lookup by key is also supported (a dictionary of key-value pairs). Since the early days of .NET, the OrderedDictionary type has supported this scenario, but only in a non-generic manner, with keys and values typed as `object`. .NET 9 introduces the long-requested OrderedDictionary<TKey,TValue> collection, which provides an efficient, generic type to support these scenarios.

The following code uses the new class.

C#

```
OrderedDictionary<string, int> d = new()
{
    ["a"] = 1,
    ["b"] = 2,
    ["c"] = 3,
};
```

```
d.Add("d", 4);
d.RemoveAt(0);
d.RemoveAt(2);
d.Insert(0, "e", 5);

foreach (KeyValuePair<string, int> entry in d)
{
    Console.WriteLine(entry);
}

// Output:
// [e, 5]
// [b, 2]
// [c, 3]
```

PriorityQueue.Remove() method

.NET 6 introduced the PriorityQueue<TElement,TPriority> collection, which provides a simple and fast array-heap implementation. One issue with array heaps in general is that they don't support priority updates, which makes them prohibitive for use in algorithms such as variations of Dijkstra's algorithm.

While it's not possible to implement efficient $O(\log n)$ priority updates in the existing collection, the new PriorityQueue<TElement,TPriority>.Remove(TElement, TElement, TPriority, IEqualityComparer<TElement>) method makes it possible to emulate priority updates (albeit at $O(n)$ time):

C#

```
public static void UpdatePriority<TElement, TPriority>(
    this PriorityQueue<TElement, TPriority> queue,
    TElement element,
    TPriority priority
    )
{
    // Scan the heap for entries matching the current element.
    queue.Remove(element, out _, out _);
    // Re-insert the entry with the new priority.
    queue.Enqueue(element, priority);
}
```

This method unblocks users who want to implement graph algorithms in contexts where asymptotic performance isn't a blocker. (Such contexts include education and prototyping.) For example, here's a toy implementation of Dijkstra's algorithm that uses the new API.

`ReadOnlySet<T>`

It's often desirable to give out read-only views of collections. ReadOnlyCollection<T> lets you create a read-only wrapper around an arbitrary mutable IList<T>, and ReadOnlyDictionary<TKey,TValue> lets you create a read-only wrapper around an arbitrary mutable IDictionary<TKey,TValue>. However, past versions of .NET had no built-in support for doing the same with ISet<T>. .NET 9 introduces ReadOnlySet<T> to address this.

The new class enables the following usage pattern.

C#

```
private readonly HashSet<int> _set = [];
private ReadOnlySet<int>? _setWrapper;

public ReadOnlySet<int> Set => _setWrapper ??= new(_set);
```

Component model - `TypeDescriptor` trimming support

System.ComponentModel includes new opt-in trimmer-compatible APIs for describing components. Any application, especially self-contained trimmed applications, can use these new APIs to help support trimming scenarios.

The primary API is the TypeDescriptor.RegisterType method on the `TypeDescriptor` class. This method has the DynamicallyAccessedMembersAttribute attribute so that the trimmer preserves members for that type. You should call this method once per type, and typically early on.

The secondary APIs have a `FromRegisteredType` suffix, such as TypeDescriptor.GetPropertiesFromRegisteredType(Type). Unlike their counterparts that don't have the `FromRegisteredType` suffix, these APIs don't have `[RequiresUnreferencedCode]` or `[DynamicallyAccessedMembers]` trimmer attributes. The lack of trimmer attributes helps consumers by no longer having to either:

- Suppress trimming warnings, which can be risky.
- Propagate a strongly typed `Type` parameter to other methods, which can be cumbersome or infeasible.

C#

```
public static void RunIt()
{
    // The Type from typeof() is passed to a different method.
    // The trimmer doesn't know about ExampleClass anymore
    // and thus there will be warnings when trimming.
    Test(typeof(ExampleClass));
```

```
    Console.ReadLine();
}

private static void Test(Type type)
{
    // When publishing self-contained + trimmed,
    // this line produces warnings IL2026 and IL2067.
    PropertyDescriptorCollection properties = TypeDescriptor.GetProperties(type);

    // When publishing self-contained + trimmed,
    // the property count is 0 here instead of 2.
    Console.WriteLine($"Property count: {properties.Count}");

    // To avoid the warning and ensure reflection
    // can see the properties, register the type:
    TypeDescriptor.RegisterType<ExampleClass>();
    // Get properties from the registered type.
    properties = TypeDescriptor.GetPropertiesFromRegisteredType(type);

    Console.WriteLine($"Property count: {properties.Count}");
}

public class ExampleClass
{
    public string? Property1 { get; set; }
    public int Property2 { get; set; }
}
```

For more information, see the API proposal.

Cryptography

- CryptographicOperations.HashData() method
- KMAC algorithm
- AES-GCM and ChaChaPoly1305 algorithms enabled for iOS/tvOS/MacCatalyst
- X.509 certificate loading
- OpenSSL providers support
- Windows CNG virtualization-based security

CryptographicOperations.HashData() method

.NET includes several static "one-shot" implementations of hash functions and related functions. These APIs include SHA256.HashData and HMACSHA256.HashData. One-shot APIs are preferable to use because they can provide the best possible performance and reduce or eliminate allocations.

If a developer wants to provide an API that supports hashing where the caller defines which hash algorithm to use, it's typically done by accepting a HashAlgorithmName argument. However, using that pattern with one-shot APIs would require switching over every possible HashAlgorithmName and then using the appropriate method. To solve that problem, .NET 9 introduces the CryptographicOperations.HashData API. This API lets you produce a hash or HMAC over an input as a one-shot where the algorithm used is determined by a HashAlgorithmName.

C#

```
static void HashAndProcessData(HashAlgorithmName hashAlgorithmName, byte[] data)
{
    byte[] hash = CryptographicOperations.HashData(hashAlgorithmName, data);
    ProcessHash(hash);
}
```

KMAC algorithm

.NET 9 provides the KMAC algorithm as specified by NIST SP-800-185. KECCAK Message Authentication Code (KMAC) is a pseudorandom function and keyed hash function based on KECCAK.

The following new classes use the KMAC algorithm. Use instances to accumulate data to produce a MAC, or use the static `HashData` method for a one-shot over a single input.

- Kmac128
- Kmac256
- KmacXof128
- KmacXof256

KMAC is available on Linux with OpenSSL 3.0 or later, and on Windows 11 Build 26016 or later. You can use the static `IsSupported` property to determine if the platform supports the desired algorithm.

C#

```
if (Kmac128.IsSupported)
{
    byte[] key = GetKmacKey();
    byte[] input = GetInputToMac();
    byte[] mac = Kmac128.HashData(key, input, outputLength: 32);
}
else
{
```

```
    // Handle scenario where KMAC isn't available.
}
```

AES-GCM and ChaChaPoly1305 algorithms enabled for iOS/tvOS/MacCatalyst

IsSupported and `ChaChaPoly1305.IsSupported` now return true when running on iOS 13+, tvOS 13+, and Mac Catalyst.

AesGcm only supports 16-byte (128-bit) tag values on Apple operating systems.

X.509 certificate loading

Since .NET Framework 2.0, the way to load a certificate has been `new X509Certificate2(bytes)`. There have also been other patterns, such as `new X509Certificate2(bytes, password, flags)`, `new X509Certificate2(path)`, `new X509Certificate2(path, password, flags)`, and `X509Certificate2Collection.Import(bytes, password, flags)` (and its overloads).

Those methods all used content-sniffing to figure out if the input was something it could handle, and then loaded it if it could. For some callers, this strategy was very convenient. But it also has some problems:

- Not every file format works on every OS.
- It's a protocol deviation.
- It's a source of security issues.

.NET 9 introduces a new X509CertificateLoader class, which has a "one method, one purpose" design. In its initial version, it only supports two of the five formats that the X509Certificate2 constructor supported. Those are the two formats that worked on all operation systems.

OpenSSL providers support

.NET 8 introduced the OpenSSL-specific APIs OpenPrivateKeyFromEngine(String, String) and OpenPublicKeyFromEngine(String, String). They enable interacting with OpenSSL ENGINE components and use hardware security modules (HSM), for example.

.NET 9 introduces SafeEvpPKeyHandle.OpenKeyFromProvider(String, String), which enables using OpenSSL providers and interacting with providers such as `tpm2` or `pkcs11`.

Some distros have removed ENGINE support since it is now deprecated.

The following snippet shows basic usage:

C#

```
byte[] data = [ /* example data */ ];

// Refer to your provider documentation, for example, https://github.com/tpm2-
software/tpm2-openssl/tree/master.
using (SafeEvpPKeyHandle priKeyHandle =
SafeEvpPKeyHandle.OpenKeyFromProvider("tpm2", "handle:0x81000007"))
using (ECDsa ecdsaPri = new ECDsaOpenSsl(priKeyHandle))
{
    byte[] signature = ecdsaPri.SignData(data, HashAlgorithmName.SHA256);
    // Do stuff with signature created by TPM.
}
```

There are some performance improvements during the TLS handshake as well as improvements to interactions with RSA private keys that use `ENGINE` components.

Windows CNG virtualization-based security

Windows 11 has added new APIs to help secure Windows keys with virtualization-based security (VBS). With this new capability, keys can be protected from admin-level key theft attacks with negligible effect on performance, reliability, or scale.

.NET 9 has added matching CngKeyCreationOptions flags. The following three flags were added:

- `CngKeyCreationOptions.PreferVbs` matching `NCRYPT_PREFER_VBS_FLAG`
- `CngKeyCreationOptions.RequireVbs` matching `NCRYPT_REQUIRE_VBS_FLAG`
- `CngKeyCreationOptions.UsePerBootKey` matching `NCRYPT_USE_PER_BOOT_KEY_FLAG`

The following snippet demonstrates how to use one of the flags:

C#

```
using System.Security.Cryptography;

CngKeyCreationParameters cngCreationParams = new()
{
    Provider = CngProvider.MicrosoftSoftwareKeyStorageProvider,
    KeyCreationOptions = CngKeyCreationOptions.RequireVbs |
CngKeyCreationOptions.OverwriteExistingKey,
};

using (CngKey key = CngKey.Create(CngAlgorithm.ECDsaP256, "myKey",
cngCreationParams))
using (ECDsaCng ecdsa = new ECDsaCng(key))
{
```

```
    // Do stuff with the key.
}
```

Date and time - new TimeSpan.From* overloads

The TimeSpan class offers several `From*` methods that let you create a `TimeSpan` object using a `double`. However, since `double` is a binary-based floating-point format, inherent imprecision can lead to errors. For instance, `TimeSpan.FromSeconds(101.832)` might not precisely represent `101 seconds, 832 milliseconds`, but rather approximately `101 seconds, 831.999999999936335370875895023345947265625 milliseconds`. This discrepancy has caused frequent confusion, and it's also not the most efficient way to represent such data. To address this, .NET 9 adds new overloads that let you create `TimeSpan` objects from integers. There are new overloads from `FromDays`, `FromHours`, `FromMinutes`, `FromSeconds`, `FromMilliseconds`, and `FromMicroseconds`.

The following code shows an example of calling the `double` and one of the new integer overloads.

C#

```
TimeSpan timeSpan1 = TimeSpan.FromSeconds(value: 101.832);
Console.WriteLine($"timeSpan1 = {timeSpan1}");
// timeSpan1 = 00:01:41.8319999

TimeSpan timeSpan2 = TimeSpan.FromSeconds(seconds: 101, milliseconds: 832);
Console.WriteLine($"timeSpan2 = {timeSpan2}");
// timeSpan2 = 00:01:41.8320000
```

Dependency injection - `ActivatorUtilities.CreateInstance` constructor

The constructor resolution for ActivatorUtilities.CreateInstance has changed in .NET 9. Previously, a constructor that was explicitly marked using the ActivatorUtilitiesConstructorAttribute attribute might not be called, depending on the ordering of constructors and the number of constructor parameters. The logic has changed in .NET 9 such that a constructor that has the attribute is always called.

Diagnostics

- Debug.Assert reports assert condition by default

- New Activity.AddLink method
- Metrics.Gauge instrument
- Out-of-proc Meter wildcard listening

Debug.Assert reports assert condition by default

Debug.Assert is commonly used to help validate conditions that are expected to always be true. Failure typically indicates a bug in the code. There are many overloads of Debug.Assert, the simplest of which just accepts a condition:

C#

```
Debug.Assert(a > 0 && b > 0);
```

The assert fails if the condition is false. Historically, however, such asserts were void of any information about what condition failed. Starting in .NET 9, if no message is explicitly provided by the user, the assert will contain the textual representation of the condition. For example, for the previous assert example, rather than getting a message like:

콘솔

```
Process terminated. Assertion failed.
   at Program.SomeMethod(Int32 a, Int32 b)
```

The message would now be:

콘솔

```
Process terminated. Assertion failed.
a > 0 && b > 0
   at Program.SomeMethod(Int32 a, Int32 b)
```

New Activity.AddLink method

Previously, you could only link a tracing Activity to other tracing contexts when you created the Activity. New in .NET 9, the AddLink(ActivityLink) API lets you link an `Activity` object to other tracing contexts after it's created. This change aligns with the OpenTelemetry specifications as well.

C#

```
ActivityContext activityContext = new(ActivityTraceId.CreateRandom(),
ActivitySpanId.CreateRandom(), ActivityTraceFlags.None);
```

```
ActivityLink activityLink = new(activityContext);

Activity activity = new("LinkTest");
activity.AddLink(activityLink);
```

Metrics.Gauge instrument

System.Diagnostics.Metrics now provides the Gauge<T> instrument according to the OpenTelemetry specification. The `Gauge` instrument is designed to record non-additive values when changes occur. For example, it can measure the background noise level, where summing the values from multiple rooms would be nonsensical. The `Gauge` instrument is a generic type that can record any value type, such as `int`, `double`, or `decimal`.

The following example demonstrates using the the `Gauge` instrument.

C#

```
Meter soundMeter = new("MeasurementLibrary.Sound");
Gauge<int> gauge = soundMeter.CreateGauge<int>(
    name: "NoiseLevel",
    unit: "dB", // Decibels.
    description: "Background Noise Level"
    );
gauge.Record(10, new TagList() { { "Room1", "dB" } });
```

Out-of-proc Meter wildcard listening

It's already possible to listen to meters out-of-process using the System.Diagnostics.Metrics event source provider, but prior to .NET 9, you had to specify the full meter name. In .NET 9, you can listen to all meters by using the wildcard character `*`, which allows you to capture metrics from every meter in a process. Additionally, it adds support for listening by meter prefix, so you can listen to all meters whose names start with a specified prefix. For example, specifying `MyMeter*` enables listening to all meters with names that begin with `MyMeter`.

C#

```
// The complete meter name is "MyCompany.MyMeter".
var meter = new Meter("MyCompany.MyMeter");
// Create a counter and allow publishing values.
meter.CreateObservableCounter("MyCounter", () => 1);

// Create the listener to use the wildcard character
// to listen to all meters using prefix names.
MyEventListener listener = new MyEventListener();
```

The `MyEventListener` class is defined as follows.

C#

```
internal class MyEventListener : EventListener
{
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        Console.WriteLine(eventSource.Name);
        if (eventSource.Name == "System.Diagnostics.Metrics")
        {
            // Listen to all meters with names starting with "MyCompany".
            // If using "*", allow listening to all meters.
            EnableEvents(
                eventSource,
                EventLevel.Informational,
                (EventKeywords)0x3,
                new Dictionary<string, string?>() { { "Metrics", "MyCompany*" } }
                );
        }
    }

    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // Ignore other events.
        if (eventData.EventSource.Name != "System.Diagnostics.Metrics" ||
            eventData.EventName == "CollectionStart" ||
            eventData.EventName == "CollectionStop" ||
            eventData.EventName == "InstrumentPublished"
            )
            return;

        Console.WriteLine(eventData.EventName);

        if (eventData.Payload is not null)
        {
            for (int i = 0; i < eventData.Payload.Count; i++)
                Console.WriteLine($"\t{eventData.PayloadNames![i]}:
{eventData.Payload[i]}");
        }
    }
}
```

When you execute the code, the output is as follows:

txt

```
CounterRateValuePublished
        sessionId: 7cd94a65-0d0d-460e-9141-016bf390d522
        meterName: MyCompany.MyMeter
        meterVersion:
        instrumentName: MyCounter
```

```
        unit:
        tags:
        rate: 0
        value: 1
        instrumentId: 1
CounterRateValuePublished
        sessionId: 7cd94a65-0d0d-460e-9141-016bf390d522
        meterName: MyCompany.MyMeter
        meterVersion:
        instrumentName: MyCounter
        unit:
        tags:
        rate: 0
        value: 1
        instrumentId: 1
```

You can also use the wildcard character to listen to metrics with monitoring tools like dotnet-counters.

LINQ

New methods CountBy and AggregateBy have been introduced. These methods make it possible to aggregate state by key without needing to allocate intermediate groupings via GroupBy.

CountBy lets you quickly calculate the frequency of each key. The following example finds the word that occurs most frequently in a text string.

C#

```
string sourceText = """
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Sed non risus. Suspendisse lectus tortor, dignissim sit amet,
    adipiscing nec, ultricies sed, dolor. Cras elementum ultrices amet diam.
""";

// Find the most frequent word in the text.
KeyValuePair<string, int> mostFrequentWord = sourceText
    .Split(new char[] { ' ', '.', ',' }, StringSplitOptions.RemoveEmptyEntries)
    .Select(word => word.ToLowerInvariant())
    .CountBy(word => word)
    .MaxBy(pair => pair.Value);

Console.WriteLine(mostFrequentWord.Key); // amet
```

AggregateBy lets you implement more general-purpose workflows. The following example shows how you can calculate scores that are associated with a given key.

C#

```
(string id, int score)[] data =
    [
        ("0", 42),
        ("1", 5),
        ("2", 4),
        ("1", 10),
        ("0", 25),
    ];

var aggregatedData =
    data.AggregateBy(
        keySelector: entry => entry.id,
        seed: 0,
        (totalScore, curr) => totalScore + curr.score
        );

foreach (var item in aggregatedData)
{
    Console.WriteLine(item);
}
//(0, 67)
//(1, 15)
//(2, 4)
```

Index<TSource>(IEnumerable<TSource>) makes it possible to quickly extract the implicit index of an enumerable. You can now write code such as the following snippet to automatically index items in a collection.

C#

```
IEnumerable<string> lines2 = File.ReadAllLines("output.txt");
foreach ((int index, string line) in lines2.Index())
{
    Console.WriteLine($"Line number: {index + 1}, Line: {line}");
}
```

Logging source generator

C# 12 introduced primary constructors, which allow you to define a constructor directly on the class declaration. The logging source generator now supports logging using classes that have a primary constructor.

C#

```
public partial class ClassWithPrimaryConstructor(ILogger logger)
{
```

```
    [LoggerMessage(0, LogLevel.Debug, "Test.")]
    public partial void Test();
}
```

Miscellaneous

In this section, find information about:

- allows ref struct used in libraries
- SearchValues expansion

`allows ref struct` used in libraries

C# 13 introduces the ability to constrain a generic parameter with `allows ref struct`, which tells the compiler and runtime that a `ref struct` can be used for that generic parameter. Many APIs that are compatible with this have now been annotated. For example, the String.Create method has an overload that lets you create a `string` by writing directly into its memory, represented as a span. This method has a `TState` argument that's passed from the caller into the delegate that does the actual writing.

That `TState` type parameter on `String.Create` is now annotated with `allows ref struct`:

C#

```
public static string Create<TState>(int length, TState state, SpanAction<char,
TState> action)
    where TState : allows ref struct;
```

This annotation enables you to pass a span (or any other `ref struct`) as input to this method.

The following example shows a new String.ToLowerInvariant() overload that uses this capability.

C#

```
public static string ToLowerInvariant(ReadOnlySpan<char> input) =>
    string.Create(span.Length, input, static (stringBuffer, input) =>
span.ToLowerInvariant(stringBuffer));
```

`SearchValues` expansion

.NET 8 introduced the SearchValues<T> type, which provides an optimized solution for searching for specific sets of characters or bytes within spans. In .NET 9, `SearchValues` has been

extended to support searching for substrings within a larger string.

The following example searches for multiple animal names within a string value, and returns an index to the first one found.

```
C#
```

```
private static readonly SearchValues<string> s_animals =
    SearchValues.Create(["cat", "mouse", "dog", "dolphin"],
StringComparison.OrdinalIgnoreCase);

public static int IndexOfAnimal(string text) =>
    text.AsSpan().IndexOfAny(s_animals);
```

This new capability has an optimized implementation that takes advantage of the SIMD support in the underlying platform. It also enables higher-level types to be optimized. For example, Regex now utilizes this functionality as part of its implementation.

Networking

- SocketsHttpHandler is default in HttpClientFactory
- System.Net.ServerSentEvents
- TLS resume with client certificates on Linux
- WebSocket keep-alive ping and timeout
- HttpClientFactory no longer logs header values by default

SocketsHttpHandler is default in HttpClientFactory

`HttpClientFactory` creates HttpClient objects backed by HttpClientHandler, by default. `HttpClientHandler` is itself backed by SocketsHttpHandler, which is much more configurable, including around connection lifetime management. `HttpClientFactory` now uses `SocketsHttpHandler` by default and configures it to set limits on its connection lifetimes to match that of the rotation lifetime specified in the factory.

System.Net.ServerSentEvents

Server-sent events (SSE) is a simple and popular protocol for streaming data from a server to a client. It's used, for example, by OpenAI as part of streaming generated text from its AI services. To simplify the consumption of SSE, the new System.Net.ServerSentEvents library provides a parser for easily ingesting server-sent events.

The following code demonstrates using the new class.

C#

```
Stream responseStream = new MemoryStream();
await foreach (SseItem<string> e in
SseParser.Create(responseStream).EnumerateAsync())
{
    Console.WriteLine(e.Data);
}
```

TLS resume with client certificates on Linux

TLS resume is a feature of the TLS protocol that allows resuming previously established sessions to a server. Doing so avoids a few roundtrips and saves computational resources during TLS handshake.

TLS resume has already been supported on Linux for SslStream connections without client certificates. .NET 9 adds support for TLS resume of mutually authenticated TLS connections, which are common in server-to-server scenarios. The feature is enabled automatically.

WebSocket keep-alive ping and timeout

New APIs on ClientWebSocketOptions and WebSocketCreationOptions let you opt in to sending WebSocket pings and aborting the connection if the peer doesn't respond in time.

Until now, you could specify a KeepAliveInterval to keep the connection from staying idle, but there was no built-in mechanism to enforce that the peer is responding.

The following example pings the server every 5 seconds and aborts the connection if it doesn't respond within a second.

C#

```
using var cws = new ClientWebSocket();
cws.Options.HttpVersionPolicy = HttpVersionPolicy.RequestVersionOrHigher;
cws.Options.KeepAliveInterval = TimeSpan.FromSeconds(5);
cws.Options.KeepAliveTimeout = TimeSpan.FromSeconds(1);

await cws.ConnectAsync(uri, httpClient, cancellationToken);
```

HttpClientFactory no longer logs header values by default

LogLevel.Trace events logged by `HttpClientFactory` no longer include header values by default. You can opt in to logging values for specific headers via the RedactLoggedHeaders helper method.

The following example redacts all headers, except for the user agent.

C#

```
services.AddHttpClient("myClient")
    .RedactLoggedHeaders(name => name != "User-Agent");
```

For more information, see HttpClientFactory logging redacts header values by default.

Reflection

- Persisted assemblies
- Type-name parsing

Persisted assemblies

In .NET Core versions and .NET 5-8, support for building an assembly and emitting reflection metadata for dynamically created types was limited to a runnable AssemblyBuilder. The lack of support for *saving* an assembly was often a blocker for customers migrating from .NET Framework to .NET. .NET 9 adds a new type, PersistedAssemblyBuilder, that you can use to save an emitted assembly.

To create a `PersistedAssemblyBuilder` instance, call its constructor and pass the assembly name, the core assembly, `System.Private.CoreLib`, to reference base runtime types, and optional custom attributes. After you emit all members to the assembly, call the PersistedAssemblyBuilder.Save(String) method to create an assembly with default settings. If you want to set the entry point or other options, you can call PersistedAssemblyBuilder.GenerateMetadata and use the metadata it returns to save the assembly. The following code shows an example of creating a persisted assembly and setting the entry point.

C#

```
public void CreateAndSaveAssembly(string assemblyPath)
{
    PersistedAssemblyBuilder ab = new PersistedAssemblyBuilder(
        new AssemblyName("MyAssembly"),
        typeof(object).Assembly
        );
    TypeBuilder tb = ab.DefineDynamicModule("MyModule")
        .DefineType("MyType", TypeAttributes.Public | TypeAttributes.Class);

    MethodBuilder entryPoint = tb.DefineMethod(
        "Main",
        MethodAttributes.HideBySig | MethodAttributes.Public |
```

```
MethodAttributes.Static
        );
    ILGenerator il = entryPoint.GetILGenerator();
    // ...
    il.Emit(OpCodes.Ret);

    tb.CreateType();

    MetadataBuilder metadataBuilder = ab.GenerateMetadata(
        out BlobBuilder ilStream,
        out BlobBuilder fieldData
        );
    PEHeaderBuilder peHeaderBuilder = new PEHeaderBuilder(
                    imageCharacteristics: Characteristics.ExecutableImage);

    ManagedPEBuilder peBuilder = new ManagedPEBuilder(
                    header: peHeaderBuilder,
                    metadataRootBuilder: new MetadataRootBuilder(metadataBuilder),
                    ilStream: ilStream,
                    mappedFieldData: fieldData,
                    entryPoint:
MetadataTokens.MethodDefinitionHandle(entryPoint.MetadataToken)
                    );

    BlobBuilder peBlob = new BlobBuilder();
    peBuilder.Serialize(peBlob);

    using var fileStream = new FileStream("MyAssembly.exe", FileMode.Create,
FileAccess.Write);
    peBlob.WriteContentTo(fileStream);
}

public static void UseAssembly(string assemblyPath)
{
    Assembly assembly = Assembly.LoadFrom(assemblyPath);
    Type? type = assembly.GetType("MyType");
    MethodInfo? method = type?.GetMethod("SumMethod");
    Console.WriteLine(method?.Invoke(null, [5, 10]));
}
```

The new PersistedAssemblyBuilder class includes PDB support. You can emit symbol info and use it to debug a generated assembly. The API has a similar shape to the .NET Framework implementation. For more information, see Emit symbols and generate PDB.

Type-name parsing

TypeName is a parser for ECMA-335 type names that provides much the same functionality as System.Type but is decoupled from the runtime environment. Components like serializers and compilers need to parse and process type names. For example, the Native AOT compiler has switched to using TypeName.

The new `TypeName` class provides:

- Static `Parse` and `TryParse` methods for parsing input represented as `ReadOnlySpan<char>`. Both methods accept an instance of `TypeNameParseOptions` class (an option bag) that lets you customize the parsing.
- `Name`, `FullName`, and `AssemblyQualifiedName` properties that work exactly like their counterparts in System.Type.
- Multiple properties and methods that provide additional information about the name itself:
  - `IsArray`, `IsSZArray` (`SZ` stands for single-dimension, zero-indexed array), `IsVariableBoundArrayType`, and `GetArrayRank` for working with arrays.
  - `IsConstructedGenericType`, `GetGenericTypeDefinition`, and `GetGenericArguments` for working with generic type names.
  - `IsByRef` and `IsPointer` for working with pointers and managed references.
  - `GetElementType()` for working with pointers, references, and arrays.
  - `IsNested` and `DeclaringType` for working with nested types.
  - `AssemblyName`, which exposes the assembly name information via the new AssemblyNameInfo class. In contrast to `AssemblyName`, the new type is *immutable*, and parsing culture names doesn't create instances of `CultureInfo`.

Both `TypeName` and `AssemblyNameInfo` types are immutable and don't provide a way to check for equality (they don't implement `IEquatable`). Comparing assembly names is simple, but different scenarios need to compare only a subset of exposed information (`Name`, `Version`, `CultureName`, and `PublicKeyOrToken`).

The following code snippet shows some example usage.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection.Metadata;

internal class RestrictedSerializationBinder
{
    Dictionary<string, Type> AllowList { get; set; }

    RestrictedSerializationBinder(Type[] allowedTypes)
        => AllowList = allowedTypes.ToDictionary(type => type.FullName!);

    Type? GetType(ReadOnlySpan<char> untrustedInput)
    {
```

```
        if (!TypeName.TryParse(untrustedInput, out TypeName? parsed))
        {
            throw new InvalidOperationException($"Invalid type name:
'{untrustedInput.ToString()}'");
        }

        if (AllowList.TryGetValue(parsed.FullName, out Type? type))
        {
            return type;
        }
        else if (parsed.IsSimple // It's not generic, pointer, reference, or an
array.
            && parsed.AssemblyName is not null
            && parsed.AssemblyName.Name == "MyTrustedAssembly"
            )
        {
            return Type.GetType(parsed.AssemblyQualifiedName, throwOnError: true);
        }

        throw new InvalidOperationException($"Not allowed:
'{untrustedInput.ToString()}'");
    }
}
```

The new APIs are available from the System.Reflection.Metadata NuGet package, which can be used with down-level .NET versions.

Regular expressions

- [GeneratedRegex] on properties
- Regex.EnumerateSplits

`[GeneratedRegex]` on properties

.NET 7 introduced the `Regex` source generator and corresponding GeneratedRegexAttribute attribute.

The following partial method will be source generated with all the code necessary to implement this `Regex`.

C#

```
[GeneratedRegex(@"\b\w{5}\b")]
private static partial Regex FiveCharWord();
```

C# 13 supports partial *properties* in addition to partial methods, so starting in .NET 9 you can also use `[GeneratedRegex(...)]` on a property.

The following partial property is the property equivalent of the previous example.

C#

```
[GeneratedRegex(@"\b\w{5}\b")]
private static partial Regex FiveCharWordProperty { get; }
```

`Regex.EnumerateSplits`

The Regex class provides a Split method, similar in concept to the String.Split method. With `String.Split`, you supply one or more `char` or `string` separators, and the implementation splits the input text on those separators. With `Regex.Split`, instead of specifying the separator as a `char` or `string`, it's specified as a regular expression pattern.

The following example demonstrates `Regex.Split`.

C#

```
foreach (string s in Regex.Split("Hello, world! How are you?", "[aeiou]"))
{
    Console.WriteLine($"Split: \"{s}\"");
}

// Output, split by all English vowels:
// Split: "H"
// Split: "ll"
// Split: ", w"
// Split: "rld! H"
// Split: "w "
// Split: "r"
// Split: " y"
// Split: ""
// Split: "?"
```

However, `Regex.Split` only accepts a `string` as input and doesn't support input being provided as a `ReadOnlySpan<char>`. Also, it outputs the full set of splits as a `string[]`, which requires allocating both the `string` array to hold the results and a `string` for each split. In .NET 9, the new EnumerateSplits method enables performing the same operation, but with a span-based input and without incurring any allocation for the results. It accepts a `ReadOnlySpan<char>` and returns an enumerable of Range objects that represent the results.

The following example demonstrates `Regex.EnumerateSplits`, taking a `ReadOnlySpan<char>` as input.

C#

```
ReadOnlySpan<char> input = "Hello, world! How are you?";
foreach (Range r in Regex.EnumerateSplits(input, "[aeiou]"))
{
    Console.WriteLine($"Split: \"{input[r]}\"");
}
```

Serialization (System.Text.Json)

- Indentation options
- Default web options singleton
- JsonSchemaExporter
- Respect nullable annotations
- Require non-optional constructor parameters
- Order JsonObject properties
- Customize enum member names
- Stream multiple JSON documents

Indentation options

JsonSerializerOptions includes new properties that let you customize the indentation character and indentation size of written JSON.

C#

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
    IndentCharacter = '\t',
    IndentSize = 2,
};

string json = JsonSerializer.Serialize(
    new { Value = 1 },
    options
    );
Console.WriteLine(json);
//{
//                "Value": 1
//}
```

Default web options singleton

If you want to serialize with the default options that ASP.NET Core uses for web apps, use the new JsonSerializerOptions.Web singleton.

C#

```
string webJson = JsonSerializer.Serialize(
    new { SomeValue = 42 },
    JsonSerializerOptions.Web // Defaults to camelCase naming policy.
    );
Console.WriteLine(webJson);
// {"someValue":42}
```

JsonSchemaExporter

JSON is frequently used to represent types in method signatures as part of remote procedure–calling schemes. It's used, for example, as part of OpenAPI specifications, or as part of tool calling with AI services like those from OpenAI. Developers can serialize and deserialize .NET types as JSON using System.Text.Json. But they also need to be able to get a JSON schema that describes the shape of the .NET type (that is, describes the shape of what would be serialized and what can be deserialized). System.Text.Json now provides the JsonSchemaExporter type, which supports generating a JSON schema that represents a .NET type.

For more information, see JSON schema exporter.

Respect nullable annotations

System.Text.Json now recognizes nullability annotations of properties and can be configured to enforce those during serialization and deserialization using the RespectNullableAnnotations flag.

The following code shows how to set the option:

C#

```
public static void RunIt()
{
    JsonSerializerOptions options = new() { RespectNullableAnnotations = true };

    // Throws exception: System.Text.Json.JsonException: The property or field
    // 'Title' on type 'Serialization+Book' doesn't allow getting null values.
    // Consider updating its nullability annotation.
    JsonSerializer.Serialize(new Book { Title = null! }, options);

    // Throws exception: System.Text.Json.JsonException: The property or field
```

```
    // 'Title' on type 'Serialization+Book' doesn't allow setting null values.
    // Consider updating its nullability annotation.
    JsonSerializer.Deserialize<Book>("""{ "Title" : null }""", options);
}

public class Book
{
    public required string Title { get; set; }
    public string? Author { get; set; }
    public int PublishYear { get; set; }
}
```

For more information, see Respect nullable annotations.

Require non-optional constructor parameters

Historically, System.Text.Json has treated non-optional constructor parameters as optional when using constructor-based deserialization. You can change that behavior using the new RespectRequiredConstructorParameters flag.

The following code shows how to set the option:

C#

```
JsonSerializerOptions options = new() { RespectRequiredConstructorParameters =
true };

// Throws exception: System.Text.Json.JsonException: JSON deserialization
// for type 'Serialization+MyPoco' was missing required properties including:
'Value'.
JsonSerializer.Deserialize<MyPoco>("""{}""", options);
```

The `MyPoco` type is defined as follows:

C#

```
record MyPoco(string Value);
```

For more information, see Non-optional constructor parameters.

Order JsonObject properties

The JsonObject type now exposes ordered dictionary–like APIs that enable explicit property order manipulation.

C#

```
JsonObject jObj = new()
{
    ["key1"] = true,
    ["key3"] = 3
};

Console.WriteLine(jObj is IList<KeyValuePair<string, JsonNode?>>); // True.

// Insert a new key-value pair at the correct position.
int key3Pos = jObj.IndexOf("key3") is int i and >= 0 ? i : 0;
jObj.Insert(key3Pos, "key2", "two");

foreach (KeyValuePair<string, JsonNode?> item in jObj)
{
    Console.WriteLine($"{item.Key}: {item.Value}");
}

// Output:
// key1: true
// key2: two
// key3: 3
```

For more information, see Manipulate property order.

Customize enum member names

The new System.Text.Json.Serialization.JsonStringEnumMemberNameAttribute attribute can be used to customize the names of individual enum members for types that are serialized as strings:

C#

```
JsonSerializer.Serialize(MyEnum.Value1 | MyEnum.Value2); // "Value1, Custom enum
value"

[Flags, JsonConverter(typeof(JsonStringEnumConverter))]
enum MyEnum
{
    Value1 = 1,
    [JsonStringEnumMemberName("Custom enum value")]
    Value2 = 2,
}
```

For more information, see Custom enum member names.

Stream multiple JSON documents

System.Text.Json.Utf8JsonReader now supports reading multiple, whitespace-separated JSON documents from a single buffer or stream. By default, the reader throws an exception if it detects any non-whitespace characters that are trailing the first top-level document. You can change this behavior using the AllowMultipleValues flag.

For more information, see Read multiple JSON documents.

Spans

In high-performance code, spans are often used to avoid allocating strings unnecessarily. Span<T> and ReadOnlySpan<T> continue to revolutionize how code is written in .NET, and every release more and more methods are added that operate on spans. .NET 9 includes the following span-related updates:

- File helpers
- params ReadOnlySpan<T> overloads
- Enumerate over ReadOnlySpan<char>.Split() segments

File helpers

The File class now has new helpers to easily and directly write `ReadOnlySpan<char>`/`ReadOnlySpan<byte>` and `ReadOnlyMemory<char>`/`ReadOnlyMemory<byte>` to files.

The following code efficiently writes a `ReadOnlySpan<char>` to a file.

C#

```
ReadOnlySpan<char> text = ...;
File.WriteAllText(filePath, text);
```

New StartsWith<T>(ReadOnlySpan<T>, T) and EndsWith<T>(ReadOnlySpan<T>, T) extension methods have also been added for spans, making it easy to test whether a ReadOnlySpan<T> starts or ends with a specific `T` value.

The following code uses these new convenience APIs.

C#

```
ReadOnlySpan<char> text = "some arbitrary text";
return text.StartsWith('"') && text.EndsWith('"'); // false
```

`params ReadOnlySpan<T>` overloads

C# has always supported marking array parameters as params. This keyword enables a simplified calling syntax. For example, the String.Join(String, String[]) method's second parameter is marked with `params`. You can call this overload with an array or by passing the values individually:

C#

```
string result = string.Join(", ", new string[3] { "a", "b", "c" });
string result = string.Join(", ", "a", "b", "c");
```

Prior to .NET 9, when you pass the values individually, the C# compiler emits code identical to the first call by producing an implicit array around the three arguments.

Starting in C# 13, you can use `params` with any argument that can be constructed via a collection expression, including spans (Span<T> and ReadOnlySpan<T>). That's beneficial for usability and performance. The C# compiler can store the arguments on the stack, wrap a span around them, and pass that off to the method, which avoids the implicit array allocation that would have otherwise resulted.

.NET 9 includes over 60 methods with a `params ReadOnlySpan<T>` parameter. Some are brand new overloads, and some are existing methods that already took a `ReadOnlySpan<T>` but now have that parameter marked with `params`. The net effect is if you upgrade to .NET 9 and recompile your code, you'll see performance improvements without making any code changes. That's because the compiler prefers to bind to span-based overloads than to the array-based overloads.

For example, `String.Join` now includes the following overload, which implements the new pattern: String.Join(String, ReadOnlySpan<String>)

Now, a call like `string.Join(", ", "a", "b", "c")` is made without allocating an array to pass in the `"a"`, `"b"`, and `"c"` arguments.

Enumerate over ReadOnlySpan<char>.Split() segments

`string.Split` is a convenient method for quickly partitioning a string with one or more supplied separators. For code focused on performance, however, the allocation profile of `string.Split` can be prohibitive, because it allocates a string for each parsed component and a `string[]` to store them all. It also doesn't work with spans, so if you have a `ReadOnlySpan<char>`, you're forced to allocate yet another string when you convert it to a string to be able to call `string.Split` on it.

In .NET 8, a set of `Split` and `SplitAny` methods were introduced for `ReadOnlySpan<char>`. Rather than returning a new `string[]`, these methods accept a destination `Span<Range>` into which the bounding indices for each component are written. This makes the operation fully allocation-free. These methods are appropriate to use when the number of ranges is both known and small.

In .NET 9, new overloads of `Split` and `SplitAny` have been added to allow incrementally parsing a `ReadOnlySpan<T>` with an *a priori* unknown number of segments. The new methods enable enumerating through each segment, which is similarly represented as a `Range` that can be used to slice into the original span.

C#

```
public static bool ListContainsItem(ReadOnlySpan<char> span, string item)
{
    foreach (Range segment in span.Split(','))
    {
        if (span[segment].SequenceEquals(item))
        {
            return true;
        }
    }

    return false;
}
```

System.Formats

The position or offset of the data in the enclosing stream for a TarEntry object is now a public property. TarEntry.DataOffset returns the position in the entry's archive stream where the entry's first data byte is located. The entry's data is encapsulated in a substream that you can access via TarEntry.DataStream, which hides the real position of the data relative to the archive stream. That's enough for most users, but if you need more flexibility and want to know the real starting position of the data in the archive stream, the new TarEntry.DataOffset API makes it easy to support features like concurrent access with very large TAR files.

C#

```
// Create stream for tar ball data in Azure Blob Storage.
BlobClient blobClient = new(connectionString, blobContainerName, blobName);
Stream blobClientStream = await blobClient.OpenReadAsync(options,
cancellationToken);

// Create TarReader for the stream and get a TarEntry.
TarReader tarReader = new(blobClientStream);
System.Formats.Tar.TarEntry? tarEntry = await tarReader.GetNextEntryAsync();
```

```
if (tarEntry is null)
    return;

// Get position of TarEntry data in blob stream.
long entryOffsetInBlobStream = tarEntry.DataOffset;
long entryLength = tarEntry.Length;

// Create a separate stream.
Stream newBlobClientStream = await blobClient.OpenReadAsync(options,
cancellationToken);
newBlobClientStream.Seek(entryOffsetInBlobStream, SeekOrigin.Begin);

// Read tar ball content from separate BlobClient stream.
byte[] bytes = new byte[entryLength];
await newBlobClientStream.ReadExactlyAsync(bytes, 0, (int)entryLength);
```

System.Guid

NewGuid() creates a `Guid` filled mostly with cryptographically secure random data, following the UUID Version 4 specification in RFC 9562. That same RFC also defines other versions, including Version 7, which "features a time-ordered value field derived from the widely implemented and well-known Unix Epoch timestamp source". In other words, much of the data is still random, but some of it is reserved for data based on a timestamp, which enables these values to have a natural sort order. In .NET 9, you can create a `Guid` according to Version 7 via the new Guid.CreateVersion7() and Guid.CreateVersion7(DateTimeOffset) methods. You can also use the new Version property to retrieve a `Guid` object's version field.

System.IO

- Compression with zlib-ng
- ZLib and Brotli compression options
- XPS documents from XPS virtual printer

Compression with zlib-ng

System.IO.Compression features like ZipArchive, DeflateStream, GZipStream, and ZLibStream are all based primarily on the zlib library. Starting in .NET 9, these features instead all use zlib-ng, a library that yields more consistent and efficient processing across a wider array of operating systems and hardware.

ZLib and Brotli compression options

ZLibCompressionOptions and BrotliCompressionOptions are new types for setting algorithm-specific compression level and strategy (`Default`, `Filtered`, `HuffmanOnly`, `RunLengthEncoding`, or `Fixed`). These types are aimed at users who want more fine-tuned settings than the only existing option, <System.IO.Compression.CompressionLevel>.

The new compression option types might be expanded in the future.

The following code snippet shows some example usage:

C#

```
private MemoryStream CompressStream(Stream uncompressedStream)
{
    MemoryStream compressorOutput = new();
    using ZLibStream compressionStream = new(
        compressorOutput,
        new ZLibCompressionOptions()
        {
            CompressionLevel = 6,
            CompressionStrategy = ZLibCompressionStrategy.HuffmanOnly
        }
        );
    uncompressedStream.CopyTo(compressionStream);
    compressionStream.Flush();

    return compressorOutput;
}
```

XPS documents from XPS virtual printer

XPS documents coming from a V4 XPS virtual printer previously couldn't be opened using the System.IO.Packaging library, due to lack of support for handling *.piece* files. This gap has been addressed in .NET 9.

System.Numerics

- BigInteger upper limit
- BigMul APIs
- Vector conversion APIs
- Vector create APIs
- Additional acceleration

BigInteger upper limit

BigInteger supports representing integer values of essentially arbitrary length. However, in practice, the length is constrained by limits of the underlying computer, such as available memory or how long it would take to compute a given expression. Additionally, there exist some APIs that fail given inputs that result in a value that's too large. Because of these limits, .NET 9 enforces a maximum length of `BigInteger`, which is that it can contain no more than `(2^31) - 1` (approximately 2.14 billion) bits. Such a number represents an almost 256 MB allocation and contains approximately 646.5 million digits. This new limit ensures that all APIs exposed are well behaved and consistent while still allowing numbers that are far beyond most usage scenarios.

`BigMul` APIs

`BigMul` is an operation that produces the full product of two numbers. .NET 9 adds dedicated `BigMul` APIs on `int`, `long`, `uint`, and `ulong` whose return type is the next larger integer type than the parameter types.

The new APIs are:

- BigMul(Int32, Int32) (returns `long`)
- BigMul(Int64, Int64) (returns `Int128`)
- BigMul(UInt32, UInt32) (returns `ulong`)
- BigMul(UInt64, UInt64) (returns `UInt128`)

Vector conversion APIs

.NET 9 adds dedicated extension APIs for converting between Vector2, Vector3, Vector4, Quaternion, and Plane.

The new APIs are as follows:

- AsPlane(Vector4)
- AsQuaternion(Vector4)
- AsVector2(Vector4)
- AsVector3(Vector4)
- AsVector4(Plane)
- AsVector4(Quaternion)
- AsVector4(Vector2)
- AsVector4(Vector3)
- AsVector4Unsafe(Vector2)
- AsVector4Unsafe(Vector3)

For same-sized conversions, such as between `Vector4`, `Quaternion`, and `Plane`, these conversions are zero cost. The same can be said for narrowing conversions, such as from `Vector4` to `Vector2` or `Vector3`. For widening conversions, such as from `Vector2` or `Vector3` to `Vector4`, there is the normal API, which initializes new elements to 0, and an `Unsafe` suffixed API that leaves these new elements undefined and therefore can be zero cost.

Vector create APIs

There are new `Create` APIs exposed for Vector, Vector2, Vector3, and Vector4 that parity the equivalent APIs exposed for the hardware vector types exposed in the System.Runtime.Intrinsics namespace.

For more information about the new APIs, see:

- Vector.Create
- Vector2.Create
- Vector3.Create
- Vector4.Create

These APIs are primarily for convenience and overall consistency across .NET's SIMD-accelerated types.

Additional acceleration

Additional performance improvements have been made to many types in the System.Numerics namespace, including to BigInteger, Vector2, Vector3, Vector4, Quaternion, and Plane.

In some cases, this has resulted in a 2-5x speedup to core APIs including `Matrix4x4` multiplication, creation of Plane from a series of vertices, Quaternion concatenation, and computing the cross product of a Vector3.

There's also constant folding support for the `SinCos` API, which computes both `Sin(x)` and `Cos(x)` in a single call, making it more efficient.

Tensors for AI

Tensors are the cornerstone data structure of artificial intelligence (AI). They can often be thought of as multidimensional arrays.

Tensors are used to:

- Represent and encode data such as text sequences (tokens), images, video, and audio.

- Efficiently manipulate higher-dimensional data.
- Efficiently apply computations on higher-dimensional data.
- Store weight information and intermediate computations (in neural networks).

To use the .NET tensor APIs, install the System.Numerics.Tensors NuGet package.

New Tensor<T> type

The new Tensor<T> type expands the AI capabilities of the .NET libraries and runtime. This type:

- Provides efficient interop with AI libraries like ML.NET, TorchSharp, and ONNX Runtime using zero copies where possible.
- Builds on top of TensorPrimitives for efficient math operations.
- Enables easy and efficient data manipulation by providing indexing and slicing operations.
- Is not a replacement for existing AI and machine learning libraries. Instead, it's intended to provide a common set of APIs to reduce code duplication and dependencies, and to achieve better performance by using the latest runtime features.

The following codes shows some of the APIs included with the new `Tensor<T>` type.

C#

```
// Create a tensor (1 x 3).
Tensor<int> t0 = Tensor.Create([1, 2, 3], [1, 3]); // [[1, 2, 3]]

// Reshape tensor (3 x 1).
Tensor<int> t1 = t0.Reshape(3, 1); // [[1], [2], [3]]

// Slice tensor (2 x 1).
Tensor<int> t2 = t1.Slice(1.., ..); // [[2], [3]]

// Broadcast tensor (3 x 1) -> (3 x 3).
// [
//  [ 1, 1, 1],
//  [ 2, 2, 2],
//  [ 3, 3, 3]
// ]
var t3 = Tensor.Broadcast<int>(t1, [3, 3]);

// Math operations.
var t4 = Tensor.Add(t0, 1); // [[2, 3, 4]]
var t5 = Tensor.Add(t0.AsReadOnlyTensorSpan(), t0); // [[2, 4, 6]]
var t6 = Tensor.Subtract(t0, 1); // [[0, 1, 2]]
var t7 = Tensor.Subtract(t0.AsReadOnlyTensorSpan(), t0); // [[0, 0, 0]]
var t8 = Tensor.Multiply(t0, 2); // [[2, 4, 6]]
var t9 = Tensor.Multiply(t0.AsReadOnlyTensorSpan(), t0); // [[1, 4, 9]]
```

```
var t10 = Tensor.Divide(t0, 2); // [[0.5, 1, 1.5]]
var t11 = Tensor.Divide(t0.AsReadOnlyTensorSpan(), t0); // [[1, 1, 1]]
```

> ⓘ **참고**
>
> This API is marked as experimental for .NET 9.

TensorPrimitives

The `System.Numerics.Tensors` library includes the TensorPrimitives class, which provides static methods for performing numerical operations on spans of values. In .NET 9, the scope of methods exposed by TensorPrimitives has been significantly expanded, growing from 40 (in .NET 8) to almost 200 overloads. The surface area encompasses familiar numerical operations from types like Math and MathF. It also includes the generic math interfaces like INumber<TSelf>, except instead of processing an individual value, they process a span of values. Many operations have also been accelerated via SIMD-optimized implementations for .NET 9.

TensorPrimitives now exposes generic overloads for any type `T` that implements a certain interface. (The .NET 8 version only included overloads for manipulating spans of `float` values.) For example, the new CosineSimilarity<T>(ReadOnlySpan<T>, ReadOnlySpan<T>) overload performs cosine similarity on two vectors of `float`, `double`, or `Half` values, or values of any other type that implements IRootFunctions<TSelf>.

Compare the precision of the cosine similarity operation on two vectors of type `float` versus `double`:

C#

```
ReadOnlySpan<float> vector1 = [1, 2, 3];
ReadOnlySpan<float> vector2 = [4, 5, 6];
Console.WriteLine(TensorPrimitives.CosineSimilarity(vector1, vector2));
// Prints 0.9746318

ReadOnlySpan<double> vector3 = [1, 2, 3];
ReadOnlySpan<double> vector4 = [4, 5, 6];
Console.WriteLine(TensorPrimitives.CosineSimilarity(vector3, vector4));
// Prints 0.9746318461970762
```

Threading

The threading APIs include improvements for iterating through tasks, for prioritized channels, which can order their elements instead of being first-in-first-out (FIFO), and `Interlocked.CompareExchange` for more types.

`Task.WhenEach`

A variety of helpful new APIs have been added for working with Task<TResult> objects. The new Task.WhenEach method lets you iterate through tasks as they complete using an `await foreach` statement. You no longer need to do things like repeatedly call Task.WaitAny on a set of tasks to pick off the next one that completes.

The following code makes multiple `HttpClient` calls and operates on their results as they complete.

C#

```
using HttpClient http = new();

Task<string> dotnet = http.GetStringAsync("http://dot.net");
Task<string> bing = http.GetStringAsync("http://www.bing.com");
Task<string> ms = http.GetStringAsync("http://microsoft.com");

await foreach (Task<string> t in Task.WhenEach(bing, dotnet, ms))
{
    Console.WriteLine(t.Result);
}
```

Prioritized unbounded channel

The System.Threading.Channels namespace lets you create first-in-first-out (FIFO) channels using the CreateBounded and CreateUnbounded methods. With FIFO channels, elements are read from the channel in the order they were written to it. In .NET 9, the new CreateUnboundedPrioritized method has been added, which orders the elements such that the next element read from the channel is the one deemed to be most important, according to either Comparer<T>.Default or a custom IComparer<T>.

The following example uses the new method to create a channel that outputs the numbers 1 through 5 in order, even though they're written to the channel in a different order.

C#

```
Channel<int> c = Channel.CreateUnboundedPrioritized<int>();

await c.Writer.WriteAsync(1);
await c.Writer.WriteAsync(5);
```

```
await c.Writer.WriteAsync(2);
await c.Writer.WriteAsync(4);
await c.Writer.WriteAsync(3);
c.Writer.Complete();

while (await c.Reader.WaitToReadAsync())
{
    while (c.Reader.TryRead(out int item))
    {
        Console.Write($"{item} ");
    }
}

// Output: 1 2 3 4 5
```

Interlocked.CompareExchange for more types

In previous versions of .NET, Interlocked.Exchange and Interlocked.CompareExchange had overloads for working with `int`, `uint`, `long`, `ulong`, `nint`, `nuint`, `float`, `double`, and `object`, as well as a generic overload for working with any reference type `T`. In .NET 9, there are new overloads for atomically working with `byte`, `sbyte`, `short`, and `ushort`. Also, the generic constraint on the generic `Interlocked.Exchange<T>` and `Interlocked.CompareExchange<T>` overloads has been removed, so those methods are no longer constrained to only work with reference types. They can now work with any primitive type, which includes all of the aforementioned types as well as `bool` and `char`, as well as any `enum` type.

.NET 9용 SDK 및 도구의 새로운 기능

아티클 • 2025. 03. 27.

이 문서에서는 .NET SDK 및 .NET 9용 도구의 새로운 기능에 대해 설명합니다.

단위 테스트

이 섹션에서는 .NET 9의 단위 테스트 업데이트, 즉 병렬로 테스트 실행 및 터미널 로거 테스트 출력에 대해 설명합니다.

병렬로 테스트 실행

.NET 9에서는 `dotnet test` MSBuild와 완전히 통합됩니다. MSBuild는 병렬 빌드를 지원하므로 여러 대상 프레임워크에서 동일한 프로젝트에 대한 테스트를 병렬로 실행할 수 있습니다. 기본적으로 MSBuild는 병렬 프로세스 수를 컴퓨터의 프로세서 수로 제한합니다. -maxcpucount 스위치를 사용하여 사용자 고유의 제한을 설정할 수도 있습니다. 병렬 처리를 옵트아웃하려면 `TestTfmsInParallel` MSBuild 속성을 `false` 설정합니다.

터미널 로거 테스트 표시

이제 MSBuild 터미널 로거에서 dotnet test 대한 테스트 결과 보고가 직접 지원됩니다. *테스트가 실행되는 동안(* 보고하고 실행 중인 테스트 이름을 표시) 및 *테스트가 완료된 후(* 모든 테스트 오류가 더 나은 방식으로 렌더링됨) 더 완전한 기능의 테스트 보고서를 받을 수 있습니다.

터미널 로거에 대한 자세한 내용은 dotnet 빌드 옵션을 참조하세요.

.NET 도구 전진

.NET 도구 전역 또는 로컬로 설치한 다음 .NET SDK 및 설치된 .NET 런타임을 사용하여 실행할 수 있는 프레임워크 종속 앱입니다. 모든 .NET 앱과 같은 이러한 도구는 특정 주 버전의 .NET을 대상으로 합니다. 기본적으로 앱은 *최신* 버전의 .NET에서 실행되지 않습니다. 도구 작성자는 `RollForward` MSBuild 속성을 설정하여 최신 버전의 .NET 런타임에서 도구를 실행하도록 옵트인할 수 있었습니다. 그러나 모든 도구가 그렇게 하는 것은 아닙니다.

dotnet tool install에 대한 새로운 옵션을 통해 *사용자가* .NET 도구를 실행하는 방법을 결정할 수 있습니다. `dotnet tool install`통해 도구를 설치하거나 dotnet tool run <toolname>통해 도구를 실행할 때 `--allow-roll-forward`이라는 새 플래그를 지정할 수

있습니다. 이 옵션은 도구를 롤 포워드 모드 `Major`로 구성합니다. 이 모드를 사용하면 일치하는 .NET 버전을 사용할 수 없는 경우 최신 주 버전의 .NET에서 도구를 실행할 수 있습니다. 이 기능을 사용하면 얼리어답터에서 도구 작성자가 코드를 변경하지 않고도 .NET 도구를 사용할 수 있습니다.

터미널 로거

이제 터미널 로거는 기본적으로 사용하도록유용성향상되었습니다.

기본적으로 사용

.NET 9부터 MSBuild를 사용하는 모든 .NET CLI 명령에 대한 기본 환경은 .NET 8에서 릴리스된 향상된 로깅 환경인 터미널 로거입니다. 이 새 출력은 최신 터미널의 기능을 사용하여 다음과 같은 기능을 제공합니다.

- 클릭 가능한 링크
- MSBuild 작업에 대한 기간 타이머
- 경고 및 오류 메시지의 색 코딩

출력은 기존 MSBuild 콘솔 로거보다 더 압축되고 사용할 수 있습니다.

새 로거는 사용할 수 있는지 자동 검색을 시도하지만 터미널 로거가 사용되는지 여부를 수동으로 제어할 수도 있습니다. 특정 명령에 대해 터미널 로거를 사용하지 않도록 설정하려면 `--tl:off` 명령줄 옵션을 지정합니다. 또는 터미널 로거를 더 광범위하게 비활성화하려면 `MSBUILDTERMINALLOGGER` 환경 변수를 `off`로 설정하십시오.

기본적으로 터미널 로거를 사용하는 명령 집합은 다음과 같습니다.

- `build`
- `clean`
- `msbuild`
- `pack`
- `publish`
- `restore`
- `test`

사용성

이제 터미널 로거는 빌드 종료 시 총 오류 및 경고 수를 요약합니다. 행바꿈이 포함된 오류도 표시합니다. 터미널 로거에 대한 자세한 내용은 'dotnet build' 옵션, 특히 `--tl` 옵션을 참조하세요.

프로젝트를 빌드할 때 경고를 내보내는 다음 프로젝트 파일을 고려합니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <Target Name="Error" BeforeTargets="Build">
    <Warning Code="ECLIPSE001" Text="Black Hole Sun, won't you come
  And wash away the rain
    Black Hole Sun, won't you come
      won't you come" />
  </Target>
</Project>
```

.NET 8 SDK에서 `dotnet build -tl` 실행하는 경우 출력은 이 단락 뒤에 나와 있습니다. 여러 줄 경고의 각 줄은 읽기 어려운 출력에 전체 오류 메시지 접두사를 포함하는 별도의 줄입니다. 또한 최종 빌드 요약에 따르면 *경고가* 있지만 *수는* 않습니다. 누락된 정보는 특정 빌드가 이전 빌드보다 더 나은지 또는 더 나쁜지 확인하기 어렵게 만들 수 있습니다.

terminal

```
$ dotnet build -tl
MSBuild version 17.8.5+b5265ef37 for .NET
Restore complete (0.5s)
  multiline-error-example succeeded with warnings (0.2s) →
bin\Debug\net8.0\multiline-error-example.dll
    E:\Code\multiline-error-example.csproj(11,5): warning ECLIPSE001: Black
Hole Sun, won't you come
E:\Code\multiline-error-example.csproj(11,5): warning ECLIPSE001:   And wash
away the rain
E:\Code\multiline-error-example.csproj(11,5): warning ECLIPSE001:     Black
Hole Sun, won't you come
E:\Code\multiline-error-example.csproj(11,5): warning ECLIPSE001:
won't you come
Build succeeded with warnings in 0.9s
```

.NET 9 SDK를 사용하여 동일한 프로젝트를 빌드하는 경우 출력은 다음과 같습니다.

terminal

```
> dotnet build -tl
Restore complete (0.4s)
You are using a preview version of .NET. See: https://aka.ms/dotnet-support-
policy
  multiline-error-example succeeded with 3 warning(s) (0.2s) →
bin\Debug\net8.0\multiline-error-example.dll
```

```
    E:\Code\multiline-error-example.csproj(11,5): warning ECLIPSE001:
      Black Hole Sun, won't you come
        And wash away the rain
          Black Hole Sun, won't you come
            won't you come
Build succeeded with 3 warning(s) in 0.8s
```

경고의 메시지 줄에는 더 이상 디스플레이를 어지럽히는 반복된 프로젝트 및 위치 정보가 없습니다. 또한 빌드 요약에서는 빌드 중에 생성된 경고 수(및 오류가 있는 경우)를 보여 줍니다.

터미널 로거에 대한 피드백이 있는 경우 MSBuild 리포지토리 제공할 수 있습니다.

대규모 리포지토리의 NuGet 종속성 처리 속도 향상

NuGet 종속성 확인자는 모든 `<PackageReference>` 프로젝트의 성능과 확장성을 개선하기 위해 철저히 검사되었습니다. 기본적으로 사용하도록 설정된 새 알고리즘은 핵심 종속성 확인 규칙을 엄격하게 준수하면서 기능을 손상시키지 않고 복원 작업을 가속화합니다.

복원 실패 또는 예기치 않은 패키지 버전과 같은 문제가 발생하는 경우 레거시 해결 프로그램 되돌릴 수 있습니다.

MSBuild 스크립트 분석기("BuildChecks")

.NET 9에는 빌드 스크립트의 결함 및 회귀를 방지할 수 있는 기능이 도입되었습니다. 빌드 검사를 실행하려면 MSBuild를 호출하는 명령에 `/check` 플래그를 추가합니다. 예를 들어 `dotnet build myapp.sln /check myapp` 솔루션을 빌드하고 구성된 모든 빌드 검사를 실행합니다.

.NET 9 SDK에는 몇 가지 초기 검사로 예를 들어, BC0101 및 BC0102이 포함되어 있습니다. 전체 목록은 BuildCheck 코드 참조하세요.

문제가 감지되면 문제가 포함된 프로젝트의 빌드 출력에서 진단이 생성됩니다.

자세한 내용은 디자인 설명서 참조하세요.

분석기 불일치

많은 사용자가 .NET SDK 및 Visual Studio를 다양한 주기로 설치합니다. 이러한 유연성은 바람직하지만 두 환경 간에 상호 운용해야 하는 도구에 문제가 발생할 수 있습니다. 이러한 종류의 도구의 한 가지 예는 Roslyn 분석기입니다. 분석기 작성자는 Roslyn의 특정 버

전에 대해 코딩해야 하지만 사용 가능한 버전과 지정된 빌드에서 사용되는 버전은 때때로 불분명합니다.

.NET SDK와 MSBuild 간의 이러한 종류의 버전 불일치는 *분리된 SDK*라고 합니다. 이 상태인 경우 다음과 같은 오류가 표시될 수 있습니다.

> CSC: 경고 CS9057: 분석기 어셈블리 '..\dotnet\sdk\8.0.200\Sdks\Microsoft.NET.Sdk.Razor\source-generators\Microsoft.CodeAnalysis.Razor.Compiler.SourceGenerators.dll'는 현재 실행 중인 버전 '4.8.0.0'보다 최신 컴파일러의 버전 '4.9.0.0'을 참조합니다.

.NET 9는 이 문제 시나리오를 감지하고 자동으로 조정할 수 있습니다. SDK의 MSBuild 논리에는 제공된 MSBuild 버전이 포함되며, 해당 정보를 사용하여 SDK가 다른 버전의 환경에서 실행되는 시기를 감지할 수 있습니다. 이 경우 SDK는 일관된 분석기 환경을 보장하는 Microsoft.Net.Sdk.Compilers.Toolset이라는 지원 패키지의 암시적 다운로드를 삽입합니다.

워크로드 세트

워크로드 집합 설치하는 워크로드 및 해당 워크로드의 변경 주기를 사용자에게 더 자세히 제어할 수 있는 SDK 기능입니다. 이전 버전에서는 구성된 NuGet 피드에 새 버전의 개별 워크로드가 릴리스됨에 따라 워크로드가 주기적으로 업데이트되었습니다. 이제 *명시적 업데이트 제스처를 취할 때까지 모든* 워크로드가 특정 단일 버전에 머뭅니다.

`dotnet workload --info`실행하여 SDK 설치의 모드를 확인할 수 있습니다.

.NET CLI

```
> dotnet workload --info
Workload version: 9.0.100-manifests.400dd185
Configured to use loose manifests when installing new manifests.
 [aspire]
   Installation Source: VS 17.10.35027.167, VS 17.11.35111.106
   Manifest Version:    8.0.2/8.0.100
   Manifest Path:       C:\Program Files\dotnet\sdk-
manifests\8.0.100\microsoft.net.sdk.aspire\8.0.2\WorkloadManifest.json
   Install Type:              Msi
```

이 예제에서 SDK 설치는 '매니페스트' 모드로, 업데이트가 사용 가능한 대로 설치됩니다. 새 모드를 옵트인하려면 `dotnet workload install` 또는 `dotnet workload update` 명령에 `--version` 옵션을 추가합니다. 새 `dotnet workload config` 명령을 사용하여 작업 모드를 명시적으로 제어할 수도 있습니다.

.NET CLI

```
> dotnet workload config --update-mode workload-set
Successfully updated workload install mode to use workload-set.
```

어떤 이유로든 다시 변경해야 하는 경우 `workload-set` 대신 `manifests` 사용하여 동일한 명령을 실행할 수 있습니다. `dotnet workload config --update-mode` 사용하여 현재 작업 모드를 확인할 수도 있습니다.

자세한 내용은 .NET SDK 워크로드 집합 참조하세요.

워크로드 기록

.NET SDK 워크로드는 .NET MAUI 및 Blazor WebAssembly의 필수적인 부분입니다. 기본 구성에서는 .NET 도구 작성자가 새 버전을 릴리스할 때 워크로드를 독립적으로 업데이트할 수 있습니다. 또한 Visual Studio를 통해 수행된 .NET SDK 설치는 병렬 버전 집합을 설치합니다. 주의하지 않고 지정된 .NET SDK 설치의 워크로드 설치 상태는 시간이 지남에 따라 드리프트될 수 있지만 이 드리프트를 시각화하는 방법은 없습니다.

이 문제를 해결하기 위해 .NET 9는 .NET SDK에 새 `dotnet workload history` 명령을 추가합니다. `dotnet workload history` 현재 .NET SDK 설치에 대한 워크로드 설치 기록 및 수정 내용의 표를 출력합니다. 이 표에서는 설치 또는 수정 날짜, 실행된 명령, 설치 또는 수정된 워크로드 및 명령에 대한 관련 버전을 보여 줍니다. 이 출력은 시간이 지남에 따라 워크로드 설치의 드리프트를 이해하는 데 도움이 되며 설치를 설정할 워크로드 버전에 대해 정보에 입각한 결정을 내리는 데 도움이 됩니다. `git reflog`를 워크로드의 기준으로 생각할 수 있습니다.

.NET CLI

```
> dotnet workload history

Id  Date                           Command       Workloads
Global.json Version  Workload Version
--------------------------------------------------------------------------------
--------------------------------------------------------------------
1   1/1/0001 12:00:00 AM +00:00  InitialState  android, ios, maccatalyst,
maui-windows                                  9.0.100-manifests.6d3c8f5d
2   9/4/2024 8:15:33 PM -05:00   install       android, aspire, ios,
maccatalyst, maui-windows                      9.0.100-rc.1.24453.3
```

이 예제에서 SDK는 처음에 `android`, `ios`, `maccatalyst` 및 `maui-windows` 워크로드와 함께 설치되었습니다. 그런 다음 `dotnet workload install aspire --version 9.0.100-rc.1.24453.3` 명령을 사용하여 `aspire` 워크로드를 설치한 후, 워크로드 집합 모드를 로

전환하고모드로 설정했습니다. 이전 상태로 돌아가려면 기록 테이블의 첫 번째 열에 있는 ID(예: `dotnet workload update --from-history 1`)를 사용할 수 있습니다.

컨테이너

- 안전하지 않은 레지스트리 대한 게시 지원
- 환경 변수 명명

안전하지 않은 레지스트리에 대한 게시 지원

SDK의 기본 제공 컨테이너 게시 지원은 컨테이너 레지스트리에 이미지를 게시할 수 있습니다. .NET 9까지 해당 레지스트리를 보호해야 했습니다. .NET SDK가 작동하려면 HTTPS 지원 및 유효한 인증서가 필요했습니다. 컨테이너 엔진은 일반적으로 안전하지 않은 레지스트리, 즉 TLS가 구성되지 않았거나 TLS가 잘못된 인증서로 구성된 레지스트리에서도 작동하도록 구성할 수 있습니다. 이는 유효한 사용 사례이지만 도구에서 이 통신 모드를 지원하지 않았습니다.

.NET 9부터 SDK는 안전하지 않은 레지스트리와 통신할 수 있습니다.

요구 사항(사용자 환경에 따라 다름):

- 레지스트리를 안전하지 않은 것으로 표시하도록 Docker CLI를 구성합니다.
- 레지스트리를 안전하지 않은 것으로 표시하도록 Podman을 구성합니다.
- `DOTNET_CONTAINER_INSECURE_REGISTRIES` 환경 변수를 사용하여 안전하지 않은 것으로 처리할 레지스트리 도메인의 세미콜론으로 구분된 목록을 전달합니다.

환경 변수 이름 지정

컨테이너 게시 도구에서 레지스트리 통신 및 보안의 몇 가지 세부적인 측면을 제어하는 데 사용하는 환경 변수는 이제 `SDK`대신 접두사 `DOTNET` 시작합니다. `SDK` 접두사는 단기적으로 계속 지원됩니다.

코드 분석

.NET 9에는 .NET 라이브러리 API를 정확하고 효율적으로 사용하고 있는지 확인하는 데 도움이 되는 몇 가지 새로운 코드 분석기 및 수정기가 포함되어 있습니다. 다음 표에서는 새 분석기를 요약합니다.

테이블 확장

규칙 ID	범주	묘사
CA1514: 중복 길이 인수를 피하십시오	유지 관리	명시적으로 계산된 길이 인수는 오류가 발생하기 쉬울 수 있으며 문자열 또는 버퍼의 끝으로 조각화할 때는 필요하지 않습니다.
CA1515: 공용 형식을 내부적 형식으로 만드는 것을 고려하십시오.	유지 관리	실행 파일 어셈블리 내의 형식은 `internal`선언해야 합니다.
CA1871: null을 허용하는 구조체를 'ArgumentNullException.ThrowIfNull'에 전달하지 마십시오.	성능	성능을 향상시키려면 null 허용 구조체를 `ArgumentNullException.ThrowIfNull`전달하는 것보다 `HasValue` 속성을 확인하고 수동으로 예외를 throw하는 것이 좋습니다.
CA1872: 'BitConverter.ToString' 기반의 호출 체인보다 'Convert.ToHexString' 및 'Convert.ToHexStringLower'를 선호합니다.	공연	바이트를 16진수 문자열 표현으로 인코딩할 때 Convert.ToHexString 또는 Convert.ToHexStringLower 사용합니다.
CA2022: Stream.Read로 부정확한 읽기 피하기	신뢰도	`Stream.Read` 호출하면 요청된 바이트보다 적은 바이트를 반환할 수 있으므로 반환 값이 확인되지 않으면 신뢰할 수 없는 코드가 생성됩니다.
CA2262: 'MaxResponseHeadersLength'를 올바르게 설정하십시오	사용법	HttpClientHandler.MaxResponseHeadersLength 속성은 바이트가 아닌 킬로바이트 단위로 측정됩니다.
CA2263: 형식이 알려진 경우 일반 오버로드를 선호합니다.	사용법	제네릭 오버로드는 컴파일 시간에 형식이 알려지면 형식 System.Type 인수를 허용하는 오버로드보다 좋습니다.
CA2264: null을 허용하지 않는 값을 'ArgumentNullException.ThrowIfNull' 전달하지 마세요.	사용법	null을 허용하지 않는 구조체(Nullable<T>제외), 'nameof()' 식 및 'new' 식과 같은 특정 구문은 null이 아닌 것으로 알려져 있으므로 `ArgumentNullException.ThrowIfNull` throw되지 않습니다.
CA2265: Span<T> null 또는 기본 비교하지 마세요.	사용법	범위를 `null` 또는 `default`과 비교해도 원하는 대로 작동하지 않을 수 있습니다. `default` 및 `null` 리터럴은 암시적으로 Span<T>.Empty로 변환됩니다. 중복 비교를 제거하거나 `IsEmpty`사용하여 코드를 보다 명시적으로 만듭니다.

.NET 9의 호환성을 깨뜨리는 변경 사항

앱을 .NET 9로 마이그레이션하는 경우, 여기에 나열된 호환성을 깨뜨리는 변경 사항이 영향을 미칠 수 있습니다. 변경 내용은 ASP.NET Core 또는 Windows Forms와 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경을 *이진 파일 비호환*, *원본 비호환* 또는 *동작 변경*으로 분류합니다.

- **이진 파일 비호환** - 새 런타임이나 구성 요소에 대해 실행할 때 기존 이진 파일의 동작이 크게 변경될 수 있습니다(예: 로드 또는 실행 실패). 그런 경우 다시 컴파일이 필요합니다.
- **원본 비호환** - 새 SDK 또는 구성 요소를 사용하여 다시 컴파일하거나 새 런타임을 대상으로 하는 경우 기존 소스 코드를 성공적으로 컴파일하려면 원본을 변경해야 할 수도 있습니다.
- **동작 변경** - 기존 코드 및 이진 파일은 런타임에 다르게 동작할 수 있습니다. 새로운 동작이 바람직하지 않은 경우 기존 코드를 업데이트하고 다시 컴파일해야 합니다.

ASP.NET Core

ASP.NET Core 9의 주요 변경 사항을 참조하세요.

컨테이너

테이블 확장

타이틀	변경 형식	소개된 버전
컨테이너 이미지는 더 이상 zlib을 설치하지 않습니다.	동작 변경	미리 보기 7
.NET Monitor 이미지가 버전 전용 태그로 간소화됨	동작 변경	미리 보기 5

핵심 .NET 라이브러리

테이블 확장

타이틀	변경 형식	소개된 버전
CompressionLevel을 사용하여 ZipArchiveEntry를 추가하면 ZIP 중앙 디렉터리 헤더 범용 비트 플래그가 설정됨	동작 변경	미리 보기 5

타이틀	변경 형식	소개된 버전
비개방형 제네릭에 대한 UnsafeAccessor 지원이 변경됨	동작 변경	미리 보기 6
사용자 지정 진단 ID를 사용한 API 폐기	원본이 호환되지 않음	(다중)
StringValues 암시적 연산자에 영향을 미치는 모호한 오버로드 해석	원본이 호환되지 않음	미국 조지아주
BigInteger 최대 길이	동작 변경	미리 보기 6
BinaryReader.ReadString()은 형식이 잘못된 시퀀스에서 "\uFFFD"를 반환합니다.	동작 변경	미리 보기 7
C# 오버로드 결정은 params 범위 형식 오버로드를 선호합니다	원본이 호환되지 않음	
System.Void 배열 형식을 만들 수 없음	동작 변경	미리 보기 1
Equals()로 표시된 형식의 기본 GetHashCode() 및 InlineArrayAttribute throw	동작 변경	미리 보기 6
EnumConverter는 등록된 형식이 열거형이 되도록 유효성을 검사합니다.	동작 변경	미리 보기 7
FromKeyedServicesAttribute는 키가 지정되지 않은 매개 변수를 더 이상 삽입하지 않음	동작 변경	RC 1
IncrementingPollingCounter 초기 콜백은 비동기적임	동작 변경	RC 1
인라인 배열 구조체 크기 제한이 적용됨	동작 변경	미리 보기 1
InMemoryDirectoryInfo는 파일 앞에 rootDir을 추가	동작 변경	미리 보기 1
정수를 사용하는 새로운 TimeSpan.From*() 오버로드	원본이 호환되지 않음	미리 보기 3
일부 OOB 패키지의 새 버전	원본이 호환되지 않음	미리 보기 5
RuntimeHelpers.GetSubArray가 다른 형식을 반환함	동작 변경	미리 보기 1
String.Trim(params ReadOnlySpan<char>) 오버로드가 제거됨	원본/이진 파일 비호환	미국 조지아주

타이틀	변경 형식	소개된 버전
빈 환경 변수에 대한 지원	동작 변경	미리 보기 6
ZipArchiveEntry 이름 및 주석은 UTF8 플래그를 따름	동작 변경	RC 1

암호화

테이블 확장

타이틀	변경 형식	소개된 버전
API가 System.Security.Cryptography.Pkcs netstandard2.0에서 제거되었습니다	원본이 호환되지 않음	미국 조지아주
SafeEvpPKeyHandle.DuplicateHandle은 핸들의 참조를 증가시킴	동작 변경	미리 보기 7
일부 X509Certificate2 및 X509Certificate 생성자는 사용되지 않음	원본이 호환되지 않음	미리 보기 7
Windows 프라이빗 키 수명 단순화	동작 변경	미리 보기 7

배포

테이블 확장

타이틀	변경 형식	소개된 버전
사용 중단된 데스크톱 Windows/macOS/Linux MonoVM 런타임 패키지	원본이 호환되지 않음	미리 보기 7
환경 변수가 앱 런타임 구성 설정에서 우선합니다.	동작 변경	미국 조지아주

Entity Framework Core

EF Core 9의 주요 변경 내용을 참조하세요.

Interop

테이블 확장

타이틀	변경 형식	소개된 버전
기본적으로 지원되는 CET	바이너리 비호환	미리 보기 6

JIT 컴파일러

테이블 확장

타이틀	변경 형식	소개된 버전
부동 소수점에서 정수로의 변환이 포화 상태입니다	동작 변경	미리 보기 4
일부 SVE API가 제거됨	원본이 호환되지 않음	RC 2

네트워킹

테이블 확장

타이틀	변경 형식	소개된 버전
HttpClient 메트릭은 무조건 보고 server.port 합니다.	동작 변경	미리 보기 7
HttpClientFactory 로깅은 기본적으로 헤더 값을 수정함	동작 변경	RC 1
HttpClientFactory는 SocketsHttpHandler를 기본 처리기로 사용합니다.	동작 변경	미리 보기 6
HttpListenerRequest.UserAgent가 null 허용임	원본이 호환되지 않음	미리 보기 1
HttpClient EventSource 이벤트의 URI 쿼리 수정	동작 변경	미리 보기 7
IHttpClientFactory 로그에서 URI 쿼리 수정	동작 변경	미리 보기 7

SDK 및 MSBuild

테이블 확장

타이틀	변경 형식	소개된 버전
dotnet sln add 잘못된 파일 이름 허용하지 않습니다.	동작 변경	9.0.2xx

타이틀	변경 형식	소개된 버전
dotnet watch이전 프레임워크에 대한 핫 다시 로드 호환되지 않음	동작 변경	RC 1
dotnet workload 명령 출력 변경	동작 변경	미리 보기 1
installer 리포지토리 버전이 더 이상 문서화되지 않음	동작 변경	미리 보기 5
MSBuild 사용자 지정 문화권 리소스 처리	동작 변경	9.0.200/9.0.300
새 기본 RID는 .NET Framework 대상으로 지정할 때 사용됩니다.	원본이 호환되지 않음	미국 조지아주
터미널 로거는 기본값입니다.	동작 변경	미리 보기 1
.NET 9 SDK에 대한 버전 요구 사항	원본이 호환되지 않음	미국 조지아주
.NET Standard 1.x 대상에 대해 내보낸 경고	원본이 호환되지 않음	미리 보기 6
.NET 7 대상에 대해 내보낸 경고	원본이 호환되지 않음	미국 조지아주

직렬화

테이블 확장

타이틀	변경 형식	소개된 버전
BinaryFormatter는 항상 throw함	동작 변경	미리 보기 6
Nullable JsonDocument 속성은 JsonValueKind.Null로 역직렬화됩니다.	동작 변경	미리 보기 1
System.Text.Json 메타데이터 판독기가 이제 메타데이터 속성 이름의 이스케이프를 해제합니다	동작 변경	미국 조지아주

윈도우 폼즈 (Windows Forms)

테이블 확장

타이틀	변경 형식	소개된 버전
BindingSource.SortDescriptions가 null을 반환하지 않음	동작 변경	미리 보기 1
null 가능성 어노테이션 변경	원본이 호환되지 않음	미리 보기 1
ComponentDesigner.Initialize가 ArgumentNullException을 throw 함	동작 변경	미리 보기 1
DataGridViewRowAccessibleObject.Name 시작 행 인덱스	동작 변경	미리 보기 1
IMsoComponent 지원은 선택 사항입니다	동작 변경	미리 보기 2
새 보안 분석기	원본이 호환되지 않음	RC 1
DataGridView가 null인 경우 예외 없음	동작 변경	미리 보기 1
PictureBox에서 HttpClient 예외 발생	동작 변경	미리 보기 6
StatusStrip은 다른 기본 렌더러 사용합니다.	동작 변경	미국 조지아주

WPF (Windows 프레젠테이션 파운데이션)

테이블 확장

타이틀	변경 형식	소개된 버전
GetXmlNamespaceMaps 형식 변경	동작 변경/원본 비호환	미리 보기 3

참조

- .NET 9의 새로운 기능
- C# 13 호환성을 깨는 변경

Last updated on 2026. 01. 29.

.NET 8의 새로운 기능

2025. 10. 11.

.NET 8은 .NET 7의 후속 작업입니다. 3 년 동안 LTS(장기 지원) 릴리스로 지원됩니다. 여기에서 .NET 8을 다운로드 할 수 있습니다.

.NET 런타임

.NET 8 런타임에는 성능, 가비지 수집 및 핵심 및 확장 라이브러리의 향상된 기능이 포함되어 있습니다. 또한 모바일 앱에 대한 새로운 세계화 모드와 COM interop 및 구성 바인딩을 위한 새 원본 생성기가 포함됩니다. 자세한 내용은 .NET 8 런타임의 새로운 기능입니다.

.NET SDK

.NET SDK의 새로운 기능, 코드 분석 및 진단에 대한 자세한 내용은 SDK의 새로운 기능 및 .NET 8용 도구를 참조하세요.

C# 12

.NET 8 SDK와 함께 제공되는 C# 12. 자세한 내용은 C# 12의 새로운 기능입니다.

갈망

Aspire는 관찰 가능한 프로덕션 준비 분산 애플리케이션을 빌드하기 위한 의견 있는 클라우드 준비 스택입니다. Aspire는 특정 클라우드 네이티브 문제를 처리하는 NuGet 패키지 컬렉션을 통해 제공되며 .NET 8용 미리 보기로 제공됩니다. 자세한 내용은 Aspire를 참조하세요.

ASP.NET Core

ASP.NET Core에는 Blazor, SignalR, 최소 API, 네이티브 AOT, Kestrel 및 HTTP.sys 서버, 인증 및 권한 부여가 개선되었습니다. 자세한 내용은 ASP.NET Core 8.0의 새로운 기능입니다.

.NET MAUI

.NET MAUI에는 컨트롤, 제스처 인식기, Windows 앱, 탐색 및 플랫폼 통합을 위한 새로운 기능이 포함되어 있습니다. 또한 몇 가지 동작 변경 내용과 많은 성능 향상이 포함됩니다. 자세한 내용은 .NET MAUI for .NET 8의 새로운 기능입니다.

EF Core

Entity Framework Core에는 복잡한 형식 개체, 기본 형식 컬렉션, JSON 열 매핑, 원시 SQL 쿼리, 지연 로드, 추적된 엔터티 액세스, 모델 빌드, 수학 번역 및 기타 기능이 포함되어 있습니다. 새 형식도 포함됩니다 `HierarchyId` . 자세한 내용은 EF Core 8의 새로운 기능

Windows Forms

Windows Forms에는 데이터 바인딩, Visual Studio DPI 및 높은 DPI에 대한 향상된 기능이 포함되어 있습니다. 단추 명령도 이제 완전히 사용하도록 설정됩니다. 자세한 내용은 .NET 8의 새로운 기능(Windows Forms)을 참조하세요.

윈도우즈 프레젠테이션 파운데이션 (Windows Presentation Foundation)

WPF(Windows Presentation Foundation)는 하드웨어 가속 및 새 OpenFolderDialog 컨트롤을 사용하는 기능을 추가합니다. 자세한 내용은 .NET 8용 WPF의 새로운 기능입니다.

참고하십시오

- .NET 8의 주요 변경 내용

.NET 미리 보기 알림

- .NET 8 발표
- .NET 8 RC 2 발표
- .NET 8 RC 1 발표
- .NET 8 미리 보기 7 발표
- .NET 8 미리 보기 6 발표
- .NET 8 미리 보기 5 발표
- .NET 8 미리 보기 4 발표
- .NET 8 미리 보기 3 발표
- .NET 8 미리 보기 2 발표
- .NET 8 미리 보기 1 발표

ASP.NET Core 미리 보기 공지

- .NET 8의 ASP.NET Core
- .NET 8 RC 2의 ASP.NET Core 업데이트

- .NET 8 RC 1의 ASP.NET Core 업데이트
- .NET 8 Preview 7의 ASP.NET Core 업데이트
- .NET 8 Preview 6의 ASP.NET Core 업데이트
- .NET 8 Preview 5의 ASP.NET Core 업데이트
- .NET 8 Preview 4의 ASP.NET Core 업데이트
- .NET 8 Preview 3의 ASP.NET Core 업데이트
- .NET 8 Preview 2의 ASP.NET Core 업데이트
- .NET 8 Preview 1의 ASP.NET Core 업데이트

.NET 8 런타임의 새로운 기능

아티클 • 2025. 01. 31.

이 문서에서는 .NET 8용 .NET 런타임의 새로운 기능에 대해 설명합니다.

성능 향상

.NET 8에는 코드 생성 및 JIT(Just-In-Time) 컴파일 기능이 향상되었습니다.

- Arm64 성능 향상
- SIMD 개선 사항
- AVX-512 ISA 확장 지원(Vector512 및 AVX-512참조)
- 클라우드 네이티브 개선 사항
- JIT 처리량 향상
- 루프 및 일반 최적화
- ThreadStaticAttribute 표시된 필드에 대한 액세스 최적화
- 연속 레지스터 할당 Arm64에는 테이블 벡터 조회를 위한 두 개의 명령어가 있으며, 이는 모든 항목이 튜플 피연산자에서 연속된 레지스터에 있어야 함을 필요로 합니다.
- 이제 JIT/NativeAOT는 컴파일 시간에 크기를 확인할 수 있는 경우, SIMD를 사용하여 비교, 복사 및 초기화를 포함한 일부 메모리 작업의 루프를 펼치고 자동 벡터화할 수 있습니다.

또한 동적 PGO(프로필 기반 최적화)가 개선되었으며 이제 기본적으로 사용하도록 설정됩니다. 더 이상 런타임 구성 옵션 사용하여 사용하도록 설정할 필요가 없습니다. 동적 PGO는 계층화된 컴파일과 함께 작동하여 계층 0 중에 배치되는 추가 계측에 따라 코드를 더욱 최적화합니다.

평균적으로 동적 PGO는 성능을 약 15%향상시킵니다. 약 4,600개의 테스트로 구성된 벤치마크 제품군에서, 23개의% 테스트에서 20% 이상의 성능 향상이 관찰되었습니다.

Codegen 구조체 승격

.NET 8에는 구조체 변수를 승격하는 JIT의 기능을 일반화하는 codegen에 대한 새로운 물리적 승격 최적화 패스가 포함되어 있습니다. 이 최적화(집계 스칼라 대체라고도 함)는 구조체 변수의 필드를 JIT가 추론하고 보다 정확하게 최적화할 수 있는 기본 변수로 대체합니다.

JIT는 이미 이 최적화를 지원했지만 다음을 비롯한 몇 가지 큰 제한 사항이 있습니다.

- 4개 이하의 필드가 있는 구조체에 대해서만 지원되었습니다.

- 각 필드가 기본 형식이거나 기본 형식을 래핑하는 간단한 구조체인 경우에만 지원되었습니다.

물리적 승격은 이러한 제한을 제거하여 여러 가지 오랜 JIT 문제를 해결합니다.

쓰레기 수거

.NET 8은 즉시 메모리 제한을 조정하는 기능을 추가합니다. 이는 수요가 오고 가는 클라우드 서비스 시나리오에서 유용합니다. 비용 효율성을 위해 서비스는 수요가 변동함에 따라 리소스 소비를 확장 및 축소해야 합니다. 서비스가 수요 감소를 감지하면 메모리 제한을 줄여 리소스 사용량을 축소할 수 있습니다. 이전에는 GC(가비지 수집기)가 변경 내용을 인식하지 못하고 새 제한보다 더 많은 메모리를 할당할 수 있으므로 실패했습니다. 이 변경으로 RefreshMemoryLimit() API를 호출하여 GC를 새 메모리 제한으로 업데이트할 수 있습니다.

주의해야 할 몇 가지 제한 사항은 다음과 같습니다.

- 32비트 플랫폼(예: Windows x86 및 Linux ARM)에서 .NET은 아직 없는 경우 새 힙 하드 제한을 설정할 수 없습니다.
- API는 새로 고침 실패를 나타내는 0이 아닌 상태 코드를 반환할 수 있습니다. 규모 축소가 너무 공격적이고 GC가 기동할 여지가 없는 경우에 발생할 수 있습니다. 이 경우 `GC.Collect(2, GCCollectionMode.Aggressive)` 호출하여 현재 메모리 사용량을 축소한 다음 다시 시도하는 것이 좋습니다.
- GC가 시작 중에 프로세스가 처리할 수 있다고 생각하는 크기를 초과하여 메모리 제한을 확장하면 `RefreshMemoryLimit` 호출이 성공하지만 제한으로 인식되는 것보다 더 많은 메모리를 사용할 수는 없습니다.

다음 코드 조각은 API를 호출하는 방법을 보여줍니다.

C#

```
GC.RefreshMemoryLimit();
```

메모리 제한과 관련된 일부 GC 구성 설정을 새로 고칠 수도 있습니다. 다음 코드 조각은 힙 하드 제한을 100 mebibytes(MiB)로 설정합니다.

C#

```
AppContext.SetData("GCHeapHardLimit", (ulong)100 * 1_024 * 1_024);
GC.RefreshMemoryLimit();
```

API는 하드 제한이 유효하지 않을 때, 예를 들어 힙 하드 제한 백분율이 음수이거나 하드 제한이 너무 낮으면, InvalidOperationException 에러를 발생시킬 수 있습니다. 이는 새

AppData 설정으로 인해 또는 컨테이너 메모리 제한 변경으로 인해 새로 고침이 설정되는 힙 하드 제한이 이미 커밋된 것보다 낮은 경우에 발생할 수 있습니다.

모바일 앱의 세계화

iOS, tvOS 및 MacCatalyst의 모바일 앱은 더 가벼운 ICU 번들을 사용하는 새로운 *하이브리드* 세계화 모드를 선택할 수 있습니다. 하이브리드 모드에서 세계화 데이터는 ICU 번들에서 부분적으로 가져오고 부분적으로 네이티브 API 호출에서 가져옵니다. 하이브리드 모드는 모바일 이 지원하는 모든 로캘을 제공합니다.

하이브리드 모드는 고정 세계화 모드에서 작동할 수 없고 모바일의 ICU 데이터에서 트리밍된 문화권을 사용하는 앱에 가장 적합합니다. 더 작은 ICU 데이터 파일을 로드하려는 경우에도 사용할 수 있습니다. (*icudt_hybrid.dat* 파일은 기본 ICU 데이터 파일 *icudt.dat*34.5 % 작습니다.)

하이브리드 세계화 모드를 사용하려면 `HybridGlobalization` MSBuild 속성을 true로 설정합니다.

XML

```
<PropertyGroup>
  <HybridGlobalization>true</HybridGlobalization>
</PropertyGroup>
```

주의해야 할 몇 가지 제한 사항은 다음과 같습니다.

- 네이티브 API의 제한 사항으로 인해 하이브리드 모드에서 모든 세계화 API가 지원되지는 않습니다.
- 지원되는 API 중 일부는 다른 동작을 갖습니다.

애플리케이션이 영향을 받는지 확인하려면 동작 차이점 참조하세요.

원본에서 생성된 COM interop

.NET 8에는 COM 인터페이스와의 상호 운용을 지원하는 새 원본 생성기가 포함되어 있습니다. GeneratedComInterfaceAttribute 사용하여 인터페이스를 원본 생성기의 COM 인터페이스로 표시할 수 있습니다. 소스 생성기는 C# 코드에서 관리되지 않는 코드로 호출할 수 있도록 하는 코드를 생성합니다. 또한 비관리 코드에서 C#으로 호출할 수 있도록 하는 코드를 생성합니다. 이 소스 생성기는 LibraryImportAttribute과 통합되며, GeneratedComInterfaceAttribute이 있는 형식을 매개 변수 및 반환 형식으로 `LibraryImport`특성이 지정된 메서드에서 사용할 수 있습니다.

C#

```
using System.Runtime.InteropServices;
using System.Runtime.InteropServices.Marshalling;

[GeneratedComInterface]
[Guid("5401c312-ab23-4dd3-aa40-3cb4b3a4683e")]
partial interface IComInterface
{
    void DoWork();
}

internal partial class MyNativeLib
{
    [LibraryImport(nameof(MyNativeLib))]
    public static partial void GetComInterface(out IComInterface
comInterface);
}
```

또한 소스 생성기는 GeneratedComInterfaceAttribute 특성을 사용하여 인터페이스를 구현하는 형식을 관리되지 않는 코드에 전달할 수 있도록 새 GeneratedComClassAttribute 특성을 지원합니다. 소스 생성기는 인터페이스를 구현하고 관리되는 구현에 호출을 전달하는 COM 개체를 노출하는 데 필요한 코드를 생성합니다.

GeneratedComInterfaceAttribute 특성이 있는 인터페이스의 메서드는 `LibraryImportAttribute`동일한 형식을 모두 지원하며, `LibraryImportAttribute` 이제 `GeneratedComInterface`-attributed 형식 및 `GeneratedComClass`-attributed 형식을 지원합니다.

C# 코드는 `GeneratedComInterface`특성 인터페이스만 사용하여 관리되지 않는 코드에서 COM 개체를 래핑하거나 C#에서 관리되는 개체를 래핑하여 관리되지 않는 코드에 노출하는 경우 Options 속성의 옵션을 사용하여 생성될 코드를 사용자 지정할 수 있습니다. 이러한 옵션은 사용되지 않을 것으로 알고 있는 시나리오에 대해 마샬러를 작성할 필요가 없음을 의미합니다.

소스 생성기는 새 StrategyBasedComWrappers 형식을 사용하여, COM 객체 래퍼 및 관리 개체 래퍼를 생성하고 관리합니다. 이 새로운 형식은 고급 사용자에 대한 사용자 지정 지점을 제공하면서 COM interop에 대한 예상 .NET 사용자 환경을 제공하는 것을 처리합니다. 애플리케이션에 COM에서 형식을 정의하는 고유한 메커니즘이 있거나 원본에서 생성된 COM이 현재 지원하지 않는 시나리오를 지원해야 하는 경우 새 StrategyBasedComWrappers 형식을 사용하여 시나리오에 대한 누락된 기능을 추가하고 COM 형식에 대해 동일한 .NET 사용자 환경을 가져오는 것이 좋습니다.

Visual Studio를 사용하는 경우 새 분석기 및 코드 수정을 통해 소스 생성 interop을 사용하도록 기존 COM interop 코드를 쉽게 변환할 수 있습니다. ComImportAttribute이 있는 각 인터페이스 옆의 전구 아이콘은 소스 생성 인터럽으로 변환할 수 있는 옵션을 제공합

니다. 수정은 GeneratedComInterfaceAttribute 특성을 사용하도록 인터페이스를 변경합니다. 또한 `GeneratedComInterfaceAttribute`사용하여 인터페이스를 구현하는 모든 클래스 옆에 전구는 GeneratedComClassAttribute 특성을 형식에 추가하는 옵션을 제공합니다. 형식이 변환되면 `DllImport` 메서드를 이동하여 `LibraryImportAttribute`사용할 수 있습니다.

제한

COM 소스 생성기는 아파트 친화성, `new` 키워드를 사용한 COM CoClass의 활성화 및 다음 API를 지원하지 않습니다.

- IDispatch기반 인터페이스입니다.
- IInspectable기반 인터페이스.
- COM 속성 및 이벤트입니다.

구성 바인딩 소스 생성기

.NET 8은 ASP.NET Core에서 AOT 및 트리밍 친화적인 구성 제공하는 원본 생성기를 도입했습니다. 생성기는 기존 리플렉션 기반 구현의 대안입니다.

원본 생성기는 Configure(TOptions), Bind및 Get 호출을 검색하여 형식 정보를 검색합니다. 프로젝트에서 생성기를 사용하도록 설정하면 컴파일러는 기존 리플렉션 기반 프레임워크 구현을 통해 생성된 메서드를 암시적으로 선택합니다.

생성기를 사용하기 위해 소스 코드 변경이 필요하지 않습니다. AOT로 컴파일된 웹 애플리케이션에서는 기본적으로 활성화되며, PublishTrimmed이 `true`로 설정된 경우 (.NET 8+ 앱)에도 활성화됩니다. 다른 프로젝트 형식의 경우 원본 생성기는 기본적으로 꺼져 있지만 프로젝트 파일에서 `true` `EnableConfigurationBindingGenerator` 속성을 설정하여 옵트인할 수 있습니다.

XML

```xml
<PropertyGroup>

<EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerator>
</PropertyGroup>
```

다음 코드는 바인더를 호출하는 예제를 보여줍니다.

C#

```
public class ConfigBindingSG
{
    static void RunIt(params string[] args)
    {
        WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
        IConfigurationSection section =
builder.Configuration.GetSection("MyOptions");

        // !! Configure call - to be replaced with source-gen'd
implementation
        builder.Services.Configure<MyOptions>(section);

        // !! Get call - to be replaced with source-gen'd implementation
        MyOptions? options0 = section.Get<MyOptions>();

        // !! Bind call - to be replaced with source-gen'd implementation
        MyOptions options1 = new();
        section.Bind(options1);

        WebApplication app = builder.Build();
        app.MapGet("/", () => "Hello World!");
        app.Run();
    }

    public class MyOptions
    {
        public int A { get; set; }
        public string S { get; set; }
        public byte[] Data { get; set; }
        public Dictionary<string, string> Values { get; set; }
        public List<MyClass> Values2 { get; set; }
    }

    public class MyClass
    {
        public int SomethingElse { get; set; }
    }
}
```

핵심 .NET 라이브러리

이 섹션에는 다음 하위 항목이 포함되어 있습니다.

- 리플렉션
- 직렬화
- 시간 추상화
- UTF8 개선 사항
- 임의성을 다루는 방법
- 성능 중심 유형

- System.Numerics 및 System.Runtime.Intrinsics
- 데이터 유효성 검사
- 지표
- 암호화
- 네트워킹
- 스트림 기반 ZipFile 메서드

반사

함수 포인터는 .NET 5에서 도입되었지만, 리플렉션에 대한 해당 지원은 당시에 추가되지 않았습니다. 함수 포인터에 `typeof` 또는 리플렉션을 사용하는 경우(예: `typeof(delegate* <void>())` 또는 `FieldInfo.FieldType` 각각) IntPtr 반환되었습니다. .NET 8부터 System.Type 개체가 대신 반환됩니다. 이 형식은 호출 규칙, 반환 형식 및 매개 변수를 포함하여 함수 포인터 메타데이터에 대한 액세스를 제공합니다.

> **참고**
>
> 함수에 대한 물리적 주소인 함수 포인터 인스턴스는 계속해서 IntPtr표시됩니다. 리플렉션 형식만 변경되었습니다.

새 기능은 현재 CoreCLR 런타임 및 MetadataLoadContext에서만 구현되고 있습니다.

System.Type및 System.Reflection.PropertyInfo, System.Reflection.FieldInfo, System.Reflection.ParameterInfo에 IsFunctionPointer과 같은 새로운 API가 추가되었습니다. 다음 코드에서는 리플렉션에 새 API 중 일부를 사용하는 방법을 보여 줍니다.

C#

```
using System;
using System.Reflection;

// Sample class that contains a function pointer field.
public unsafe class UClass
{
    public delegate* unmanaged[Cdecl, SuppressGCTransition]<in int, void>
_fp;
}

internal class FunctionPointerReflection
{
    public static void RunIt()
    {
        FieldInfo? fieldInfo = typeof(UClass).GetField(nameof(UClass._fp));

        // Obtain the function pointer type from a field.
        Type? fpType = fieldInfo?.FieldType;
```

```
        // New methods to determine if a type is a function pointer.
        Console.WriteLine(
        $"IsFunctionPointer: {fpType?.IsFunctionPointer}");
        Console.WriteLine(
            $"IsUnmanagedFunctionPointer:
{fpType?.IsUnmanagedFunctionPointer}");

        // New methods to obtain the return and parameter types.
        Console.WriteLine($"Return type:
{fpType?.GetFunctionPointerReturnType()}");

        if (fpType is not null)
        {
            foreach (Type parameterType in
fpType.GetFunctionPointerParameterTypes())
            {
                Console.WriteLine($"Parameter type: {parameterType}");
            }
        }

        // Access to custom modifiers and calling conventions requires a
"modified type".
        Type? modifiedType = fieldInfo?.GetModifiedFieldType();

        // A modified type forwards most members to its underlying type.
        Type? normalType = modifiedType?.UnderlyingSystemType;

        if (modifiedType is not null)
        {
            // New method to obtain the calling conventions.
            foreach (Type callConv in
modifiedType.GetFunctionPointerCallingConventions())
            {
                Console.WriteLine($"Calling convention: {callConv}");
            }
        }

        // New method to obtain the custom modifiers.
        Type[]? modifiers =
            modifiedType?.GetFunctionPointerParameterTypes()
[0].GetRequiredCustomModifiers();

        if (modifiers is not null)
        {
            foreach (Type modreq in modifiers)
            {
                Console.WriteLine($"Required modifier for first parameter:
{modreq}");
            }
        }
    }
}
```

이전 예제에서는 다음 출력을 생성합니다.

Output

```
IsFunctionPointer: True
IsUnmanagedFunctionPointer: True
Return type: System.Void
Parameter type: System.Int32&
Calling convention:
System.Runtime.CompilerServices.CallConvSuppressGCTransition
Calling convention: System.Runtime.CompilerServices.CallConvCdecl
Required modifier for first parameter:
System.Runtime.InteropServices.InAttribute
```

직렬화

.NET 8에서 System.Text.Json 직렬화 및 역직렬화 기능의 많은 개선이 이루어졌습니다. 예를 들어 POCO 에 없는 JSON 속성의 처리를 사용자 지정할수 있습니다.

다음 섹션에서는 다른 직렬화 개선 사항에 대해 설명합니다.

- 추가 형식에 대한 기본 지원
- 원본 생성기
- 인터페이스 계층 구조
- 명명 정책
- 읽기 전용 속성
- 리플렉션 기반 기본 사용 안 함
- 새 JsonNode API 메서드
- 비공용 멤버
- 스트리밍 역직렬화 API
- WithAddedModifier 확장 메서드
- 새 JsonContent. 오버로드 만들기
- JsonSerializerOptions 인스턴스를 불변으로 만들기

일반적으로 JSON 직렬화에 대한 자세한 내용은 .NET JSON 직렬화 및 역직렬화참조하세요.

추가 형식에 대한 내장 지원

serializer는 다음과 같은 추가 형식을 기본적으로 지원합니다.

- Half, Int128및 UInt128 숫자 형식입니다.

  C#

```
Console.WriteLine(JsonSerializer.Serialize(
    [ Half.MaxValue, Int128.MaxValue, UInt128.MaxValue ]
));
//
[65500,170141183460469231731687303715884105727,340282366920938463463374
607431768211455]
```

- Memory<T> 및 ReadOnlyMemory<T> 값입니다. `byte` 값은 Base64 문자열로 직렬화되고 다른 형식은 JSON 배열로 직렬화됩니다.

```
C#
```

```
JsonSerializer.Serialize<ReadOnlyMemory<byte>>(new byte[] { 1, 2, 3 });
// "AQID"
JsonSerializer.Serialize<Memory<int>>(new int[] { 1, 2, 3 }); //
[1,2,3]
```

원본 생성기

.NET 8에는 리플렉션 기반 직렬 변환기동등하게 Native AOT 환경을 만들기 위한 System.Text.Json 원본 생성기 향상된 기능이 포함되어 있습니다. 예를 들어:

- 원본 생성기는 이제 required 및 init 속성을 사용하여 형식을 직렬화하도록 지원합니다. 모두 리플렉션 기반 serialization에서 이미 지원되었습니다.
- 소스 생성 코드의 서식이 향상되었습니다.
- JsonSourceGenerationOptionsAttribute와 JsonSerializerOptions의 기능 동등성. 자세한 내용은 옵션 지정(원본 생성)참조하세요.
- 추가 진단(예: SYSLIB1034 및 SYSLIB1039).
- 무시되거나 액세스할 수 없는 속성 형식은 포함하지 마세요.
- 임의 형식 종류 내에서 `JsonSerializerContext` 선언 중첩을 지원합니다.
- 약한 형식의 소스 생성 시나리오에서 컴파일러 생성 형식 또는 표현할 수 없는 형식(,)을 지원합니다. 컴파일러가 생성하는 형식을 소스 생성기에서 명시적으로 지정할 수 없기 때문에, 이제 System.Text.Json는 런타임 시 가장 가까운 상위 유형을 확인하는 기능을 수행합니다. 이 결의안은 값을 직렬화할 가장 적절한 상위 형식을 결정합니다.
- 새 변환기 유형 `JsonStringEnumConverter<TEnum>`. 기존 JsonStringEnumConverter 클래스는 Native AOT에서 지원되지 않습니다. 다음과 같이 열거형 형식에 주석을 달 수 있습니다.

```
C#
```

```
[JsonConverter(typeof(JsonStringEnumConverter<MyEnum>))]
public enum MyEnum { Value1, Value2, Value3 }

[JsonSerializable(typeof(MyEnum))]
public partial class MyContext : JsonSerializerContext { }
```

자세한 내용은 열거형 필드를 문자열로 직렬화하기 를 참조하세요.

- 새 `JsonConverter.Type` 속성을 사용하면 제네릭이 아닌 `JsonConverter` 인스턴스의 형식을 조회할 수 있습니다.

```
C#
```

```
Dictionary<Type, JsonConverter>
CreateDictionary(IEnumerable<JsonConverter> converters)
    => converters.Where(converter => converter.Type != null)
                 .ToDictionary(converter => converter.Type!);
```

이 속성은 `JsonConverterFactory` 인스턴스에 대해 `null`을 반환하고 `JsonConverter<T>` 인스턴스에 대해 `typeof(T)`를 반환하기 때문에 null 값을 허용합니다.

연쇄 소스 생성기

JsonSerializerOptions 클래스에는 기존 TypeInfoResolver 속성을 보완하는 새 TypeInfoResolverChain 속성이 포함되어 있습니다. 이러한 속성은 소스 생성기를 연결하기 위한 계약 사용자 지정에 사용됩니다. 새 속성을 추가하면 하나의 호출 사이트에서 모든 연결된 구성 요소를 지정할 필요가 없어지며, 나중에 추가할 수 있습니다. 또한 TypeInfoResolverChain 체인을 분석하거나 해당 체인에서 구성 요소를 제거할 수 있습니다. 자세한 내용은 소스 생성기결합을 참조하세요.

또한 JsonSerializerOptions.AddContext<TContext>() 이제 사용되지 않습니다. TypeInfoResolver 및 TypeInfoResolverChain 속성으로 대체되었습니다. 자세한 내용은 SYSLIB0049참조하세요.

인터페이스 계층 구조

.NET 8은 인터페이스 계층에서 속성을 직렬화하는 지원을 추가합니다.

다음 코드는 즉시 구현된 인터페이스와 해당 기본 인터페이스의 속성이 직렬화되는 예제를 보여줍니다.

C#

```
public static void InterfaceHierarchies()
{
    IDerived value = new DerivedImplement { Base = 0, Derived = 1 };
    string json = JsonSerializer.Serialize(value);
    Console.WriteLine(json); // {"Derived":1,"Base":0}
}

public interface IBase
{
    public int Base { get; set; }
}

public interface IDerived : IBase
{
    public int Derived { get; set; }
}

public class DerivedImplement : IDerived
{
    public int Base { get; set; }
    public int Derived { get; set; }
}
```

명명 정책

JsonNamingPolicy `snake_case`(밑줄 포함) 및 `kebab-case`(하이픈 포함) 속성 이름 변환에 대한 새 명명 정책을 포함합니다. 기존 JsonNamingPolicy.CamelCase 정책과 유사하게 다음 정책을 사용합니다.

C#

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.SnakeCaseLower
};
JsonSerializer.Serialize(new { PropertyName = "value" }, options);
// { "property_name" : "value" }
```

자세한 내용은 기본 제공 명명 정책을 참조하세요.

읽기 전용 속성

이제 읽기 전용 필드 또는 속성(즉, `set` 접근자가 없는 필드)으로 역직렬화할 수 있습니다.

이 지원을 전역적으로 받으려면 새 옵션 PreferredObjectCreationHandling을(를) JsonObjectCreationHandling.Populate으로 설정해야 합니다. 호환성이 중요한 경우 속성을 채울 특정 형식 또는 개별 속성에 `[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]` 특성을 배치하여 기능을 보다 세분화할 수도 있습니다.

예를 들어 두 개의 읽기 전용 속성이 있는 `CustomerInfo` 형식으로 역직렬화하는 다음 코드를 고려해 보세요.

C#

```
public static void ReadOnlyProperties()
{
    CustomerInfo customer = JsonSerializer.Deserialize<CustomerInfo>("""
        { "Names":["John Doe"], "Company":{"Name":"Contoso"} }
        """)!;

    Console.WriteLine(JsonSerializer.Serialize(customer));
}

class CompanyInfo
{
    public required string Name { get; set; }
    public string? PhoneNumber { get; set; }
}

[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
class CustomerInfo
{
    // Both of these properties are read-only.
    public List<string> Names { get; } = new();
    public CompanyInfo Company { get; } = new()
    {
        Name = "N/A",
        PhoneNumber = "N/A"
    };
}
```

.NET 8 이전에는 입력 값이 무시되었고 `Names` 및 `Company` 속성이 기본값을 유지했습니다.

출력

```
{"Names":[],"Company":{"Name":"N/A","PhoneNumber":"N/A"}}
```

이제 입력 값은 역직렬화하는 동안 읽기 전용 속성을 채우는 데 사용됩니다.

출력

```
{"Names":["John Doe"],"Company":{"Name":"Contoso","PhoneNumber":"N/A"}}
```

채우기 역직렬화 동작에 대한 자세한 내용은 초기화된 속성 채우기를 참조하세요.

리플렉션 기반 기본값 사용 안 함

이제 기본적으로 리플렉션 기반 직렬 변환기를 사용하지 않도록 설정할 수 있습니다. 이 비활성화는 특히 트리밍 및 네이티브 AOT 앱에서 사용되지 않는 리플렉션 구성 요소의 우발적인 루팅을 방지하는 데 유용합니다. JsonSerializerOptions 인수를 JsonSerializer serialization 및 deserialization 메서드에 전달하도록 요구하여 기본 리플렉션 기반 serialization을 사용하지 않도록 설정하려면 프로젝트 파일에서 `JsonSerializerIsReflectionEnabledByDefault` MSBuild 속성을 `false` 설정합니다.

새 IsReflectionEnabledByDefault API를 사용하여 기능 스위치의 값을 확인합니다. System.Text.Json을 기반으로 라이브러리를 구축하는 작성자는 리플렉션 구성 요소를 실수로 고정하지 않고 기본 설정을 구성하는 속성을 활용할 수 있습니다.

자세한 내용은 리플렉션 기본값 사용 안 함참조하세요.

새 JsonNode API 메서드

JsonNode 및 System.Text.Json.Nodes.JsonArray 형식에는 다음과 같은 새 메서드가 포함됩니다.

C#

```
public partial class JsonNode
{
    // Creates a deep clone of the current node and all its descendants.
    public JsonNode DeepClone();

    // Returns true if the two nodes are equivalent JSON representations.
    public static bool DeepEquals(JsonNode? node1, JsonNode? node2);

    // Determines the JsonValueKind of the current node.
    public JsonValueKind GetValueKind(JsonSerializerOptions options = null);

    // If node is the value of a property in the parent
    // object, returns its name.
    // Throws InvalidOperationException otherwise.
    public string GetPropertyName();

    // If node is the element of a parent JsonArray,
    // returns its index.
    // Throws InvalidOperationException otherwise.
    public int GetElementIndex();
```

```
    // Replaces this instance with a new value,
    // updating the parent object/array accordingly.
    public void ReplaceWith<T>(T value);

    // Asynchronously parses a stream as UTF-8 encoded data
    // representing a single JSON value into a JsonNode.
    public static Task<JsonNode?> ParseAsync(
        Stream utf8Json,
        JsonNodeOptions? nodeOptions = null,
        JsonDocumentOptions documentOptions = default,
        CancellationToken cancellationToken = default);
}

public partial class JsonArray
{
    // Returns an IEnumerable<T> view of the current array.
    public IEnumerable<T> GetValues<T>();
}
```

비공개 멤버

JsonIncludeAttribute 및 JsonConstructorAttribute 특성 주석을 사용하여 지정된 형식에 대한 serialization 계약에 public이 아닌 멤버를 옵트인할 수 있습니다.

C#

```
public static void NonPublicMembers()
{
    string json = JsonSerializer.Serialize(new MyPoco(42));
    Console.WriteLine(json);
    // {"X":42}

    JsonSerializer.Deserialize<MyPoco>(json);
}

public class MyPoco
{
    [JsonConstructor]
    internal MyPoco(int x) => X = x;

    [JsonInclude]
    internal int X { get; }
}
```

자세한 내용은 변경할 수 없는 형식 및 public이 아닌 멤버 및 접근자사용을 참조하세요.

스트리밍 역직렬화 API

.NET 8에는 GetFromJsonAsAsyncEnumerable같은 새로운 IAsyncEnumerable<T> 스트리밍 역직렬화 확장 메서드가 포함되어 있습니다. Task<TResult>를 반환하는 유사한 메서드들이 있으며, 예를 들어 HttpClientJsonExtensions.GetFromJsonAsync이 있습니다. 새 확장 메서드는 스트리밍 API를 호출하고 IAsyncEnumerable<T>반환합니다.

다음 코드에서는 새 확장 메서드를 사용하는 방법을 보여 줍니다.

C#

```
public async static void StreamingDeserialization()
{
    const string RequestUri = "https://api.contoso.com/books";
    using var client = new HttpClient();
    IAsyncEnumerable<Book?> books =
client.GetFromJsonAsAsyncEnumerable<Book>(RequestUri);

    await foreach (Book? book in books)
    {
        Console.WriteLine($"Read book '{book?.title}'");
    }
}

public record Book(int id, string title, string author, int publishedYear);
```

WithAddedModifier 확장 메서드

새 WithAddedModifier(IJsonTypeInfoResolver, Action<JsonTypeInfo>) 확장 메서드를 사용하면 임의 `IJsonTypeInfoResolver` 인스턴스의 serialization 계약을 쉽게 수정할 수 있습니다.

C#

```
var options = new JsonSerializerOptions
{
    TypeInfoResolver = MyContext.Default
        .WithAddedModifier(static typeInfo =>
        {
            foreach (JsonPropertyInfo prop in typeInfo.Properties)
            {
                prop.Name = prop.Name.ToUpperInvariant();
            }
        })
};
```

새 JsonContent.오버로드 만들기

이제 트리밍 안전 계약 또는 소스 생성 계약을 사용해 JsonContent 인스턴스를 만들 수 있습니다. 새 메서드는 다음과 같습니다.

- JsonContent.Create(Object, JsonTypeInfo, MediaTypeHeaderValue)
- JsonContent.Create<T>(T, JsonTypeInfo<T>, MediaTypeHeaderValue)

```
C#
```

```
var book = new Book(id: 42, "Title", "Author", publishedYear: 2023);
HttpContent content = JsonContent.Create(book, MyContext.Default.Book);

public record Book(int id, string title, string author, int publishedYear);

[JsonSerializable(typeof(Book))]
public partial class MyContext : JsonSerializerContext
{
}
```

JsonSerializerOptions 인스턴스를 변경 불가능하게 설정

다음 새 메서드를 사용하면 JsonSerializerOptions 인스턴스가 고정되는 시기를 제어할 수 있습니다.

- JsonSerializerOptions.MakeReadOnly()

  이 오버로드는 트리밍에 안전하도록 설계되었으므로, 옵션 인스턴스가 리졸버로 구성되지 않은 경우 예외를 발생시킵니다.

- JsonSerializerOptions.MakeReadOnly(Boolean)

  이 오버로드에 `true` 전달하면 옵션 인스턴스가 누락된 경우 기본 리플렉션 확인자를 사용하여 해당 인스턴스를 채웁니다. 이 메서드는 `RequiresUnreferenceCode`/`RequiresDynamicCode` 표시되므로 네이티브 AOT 애플리케이션에 적합하지 않습니다.

새 IsReadOnly 속성을 사용하면 옵션 인스턴스가 고정되었는지 확인할 수 있습니다.

시간 추상화

새 TimeProvider 클래스 및 ITimer 인터페이스는 테스트 시나리오에서 시간을 모의할 수 있는 *시간 추상화* 기능을 추가합니다. 또한 시간 추상화 기능을 사용하여 Task.Delay 및 Task.WaitAsync사용하여 시간 진행에 의존하는 Task 작업을 모의할 수 있습니다. 시간 추상화는 다음과 같은 필수 시간 작업을 지원합니다.

- 로컬 및 UTC 시간 검색

- 성능 측정을 위한 타임스탬프 가져오기
- 타이머 만들기

다음 코드 조각은 몇 가지 사용 예제를 보여 줍니다.

C#

```
// Get system time.
DateTimeOffset utcNow = TimeProvider.System.GetUtcNow();
DateTimeOffset localNow = TimeProvider.System.GetLocalNow();

TimerCallback callback = s => ((State)s!).Signal();

// Create a timer using the time provider.
ITimer timer = _timeProvider.CreateTimer(
    callback, null, TimeSpan.Zero, Timeout.InfiniteTimeSpan);

// Measure a period using the system time provider.
long providerTimestamp1 = TimeProvider.System.GetTimestamp();
long providerTimestamp2 = TimeProvider.System.GetTimestamp();

TimeSpan period = _timeProvider.GetElapsedTime(providerTimestamp1,
providerTimestamp2);
```

C#

```
// Create a time provider that works with a
// time zone that's different than the local time zone.
private class ZonedTimeProvider(TimeZoneInfo zoneInfo) : TimeProvider()
{
    private readonly TimeZoneInfo _zoneInfo = zoneInfo ??
TimeZoneInfo.Local;

    public override TimeZoneInfo LocalTimeZone => _zoneInfo;

    public static TimeProvider FromLocalTimeZone(TimeZoneInfo zoneInfo) =>
        new ZonedTimeProvider(zoneInfo);
}
```

UTF8 개선 사항

형식의 문자열과 유사한 표현을 대상 범위에 쓸 수 있도록 하려면 형식에 새 IUtf8SpanFormattable 인터페이스를 구현합니다. 이 새 인터페이스는 ISpanFormattable 밀접하게 관련되어 있지만 UTF16 및 `Span<char>` 대신 UTF8 및 `Span<byte>` 대상으로 합니다.

모든 기본 형식(및 기타 형식)에서 IUtf8SpanFormattable이 구현되었으며, `string`, `Span<char>`, 또는 `Span<byte>`을 대상으로 할 때도 동일한 공유 논리를 사용합니다. 모든

형식(새 "B" 이진 지정자포함) 및 모든 문화권에 대한 모든 지원을 제공합니다. 즉, 이제 `Byte`, `Complex`, `Char`, `DateOnly`, `DateTime`, `DateTimeOffset`, `Decimal`, `Double`, `Guid`, `Half`, `IPAddress`, `IPNetwork`, `Int16`, `Int32`, `Int64`, `Int128`, `IntPtr`, `NFloat`, `SByte`, `Single`, `Rune`, `TimeOnly`, `TimeSpan`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `UIntPtr`, `Version`부터 UTF8로 직접 서식을 지정할 수 있습니다.

새로운 Utf8.TryWrite 메서드는 UTF16 기반의 기존 MemoryExtensions.TryWrite 메서드에 UTF8 기반 메서드를 제공합니다. 보간된 문자열 구문을 사용하여 복합 식의 형식을 UTF8 바이트 범위로 직접 지정할 수 있습니다. 예를 들면 다음과 같습니다.

C#

```
static bool FormatHexVersion(
    short major,
    short minor,
    short build,
    short revision,
    Span<byte> utf8Bytes,
    out int bytesWritten) =>
    Utf8.TryWrite(
        utf8Bytes,
        CultureInfo.InvariantCulture,
        $"{major:X4}.{minor:X4}.{build:X4}.{revision:X4}",
        out bytesWritten);
```

구현은 형식 값의 IUtf8SpanFormattable을 인식하고, 인식한 구현을 사용하여 UTF8 표현을 대상 스팬에 직접 씁니다.

또한 구현은 새 Encoding.TryGetBytes(ReadOnlySpan<Char>, Span<Byte>, Int32) 메서드를 활용하며, 이 메서드는 Encoding.TryGetChars(ReadOnlySpan<Byte>, Span<Char>, Int32) 해당 메서드와 함께 대상 범위로 인코딩 및 디코딩을 지원합니다. 범위가 결과 상태를 보유하기에 충분하지 않은 경우 메서드는 예외를 throw하는 대신 `false` 반환합니다.

임의성 작업 방법

System.Random 및 System.Security.Cryptography.RandomNumberGenerator 형식은 임의성 작업을 위한 두 가지 새로운 메서드를 도입합니다.

항목 가져오기<T>()

새 System.Random.GetItems 및 System.Security.Cryptography.RandomNumberGenerator.GetItems 메서드를 사용하면 입력 집합에서 지정된 수의 항목을 임의로 선택할 수 있습니다. 다음 예제에서는

`System.Random.GetItems<T>()`(Random.Shared 속성에서 제공하는 인스턴스에서)를 사용하여 배열에 31개 항목을 임의로 삽입하는 방법을 보여 줍니다. 이 예제는 플레이어가 색이 지정된 단추 시퀀스를 기억해야 하는 "Simon" 게임에서 사용할 수 있습니다.

C#

```
private static ReadOnlySpan<Button> s_allButtons = new[]
{
    Button.Red,
    Button.Green,
    Button.Blue,
    Button.Yellow,
};

// ...

Button[] thisRound = Random.Shared.GetItems(s_allButtons, 31);
// Rest of game goes here ...
```

순서 섞기<T>()

새 Random.Shuffle 및 RandomNumberGenerator.Shuffle<T>(Span<T>) 메서드를 사용하면 범위의 순서를 임의로 지정할 수 있습니다. 이러한 방법은 기계 학습에서 학습 편향을 줄이는 데 유용합니다(따라서 첫 번째 방법은 항상 학습이 아니며 마지막 방법은 항상 테스트하는 것이 아닙니다).

C#

```
YourType[] trainingData = LoadTrainingData();
Random.Shared.Shuffle(trainingData);

IDataView sourceData = mlContext.Data.LoadFromEnumerable(trainingData);

DataOperationsCatalog.TrainTestData split =
mlContext.Data.TrainTestSplit(sourceData);
model = chain.Fit(split.TrainSet);

IDataView predictions = model.Transform(split.TestSet);
// ...
```

성능 중심 유형

.NET 8에는 앱 성능 향상을 위한 몇 가지 새로운 형식이 도입되었습니다.

- 새 System.Collections.Frozen 네임스페이스에는 FrozenDictionary<TKey,TValue> 및 FrozenSet<T>컬렉션 형식이 포함됩니다. 이러한 형식은 컬렉션이 만들어지면 키와

값을 변경할 수 없습니다. 이 요구 사항을 통해 더 빠른 읽기 작업(예: `TryGetValue()`)을 수행할 수 있습니다. 이러한 형식은 처음 사용할 때 채워진 다음 수명이 긴 서비스 기간 동안 유지되는 컬렉션에 특히 유용합니다. 예를 들면 다음과 같습니다.

```
C#

private static readonly FrozenDictionary<string, bool>
s_configurationData =
    LoadConfigurationData().ToFrozenDictionary();

// ...
if (s_configurationData.TryGetValue(key, out bool setting) && setting)
{
    Process();
}
```

- MemoryExtensions.IndexOfAny 같은 메서드는 전달된 컬렉션 *에서*값이 처음으로 나타나는 위치를 찾습니다. 새 System.Buffers.SearchValues<T> 형식은 이러한 메서드에 전달되도록 설계되었습니다. 이에 따라 .NET 8은 새 형식의 인스턴스를 허용하는 MemoryExtensions.IndexOfAny 같은 메서드의 새 오버로드를 추가합니다. SearchValues<T>인스턴스를 만들 때 후속 검색을 최적화하는 데 필요한 모든 데이터는 파생됩니다. 즉, 작업이 미리 수행됩니다.

- 새 System.Text.CompositeFormat 형식은 컴파일 시간에 알려지지 않은 형식 문자열을 최적화하는 데 유용합니다(예: 리소스 파일에서 형식 문자열이 로드되는 경우). 문자열 구문 분석과 같은 작업을 수행하기 위해 약간의 추가 시간이 소요되지만 각 사용 시 작업이 수행되지 않도록 합니다.

```
C#

private static readonly CompositeFormat s_rangeMessage =
    CompositeFormat.Parse(LoadRangeMessageResource());

// ...
static string GetMessage(int min, int max) =>
    string.Format(CultureInfo.InvariantCulture, s_rangeMessage, min,
max);
```

- 새로운 System.IO.Hashing.XxHash3 및 System.IO.Hashing.XxHash128 형식은 빠른 XXH3 및 XXH128 해시 알고리즘의 구현을 제공합니다.

System.Numerics 및 System.Runtime.Intrinsics

이 섹션에서는 System.Numerics 및 System.Runtime.Intrinsics 네임스페이스의 향상된 기능에 대해 설명합니다.

- Vector256<T>, Matrix3x2및 Matrix4x4 .NET 8에서 하드웨어 가속이 향상되었습니다. 예를 들어 Vector256<T> 가능한 경우 내부적으로 `2x Vector128<T>` 작업으로 다시 구현되었습니다. 이렇게 하면 arm64와 같이 `Vector128.IsHardwareAccelerated == true` 있지만 `Vector256.IsHardwareAccelerated == false`경우 일부 함수의 부분 가속이 가능합니다.
- 하드웨어 내장 함수에는 이제 `ConstExpected` 속성이 주석 처리됩니다. 이렇게 하면 기본 하드웨어에 상수가 필요할 때와 상수가 아닌 값이 예기치 않게 성능에 저하될 수 있는 경우를 사용자가 인식할 수 있습니다.
- Lerp(TSelf, TSelf, TSelf) `Lerp` API가 IFloatingPointIeee754<TSelf> 추가되었으므로 `float`(Single), `double`(Double) 및 Half. 이 API를 사용하면 두 값 간의 선형 보간을 효율적이고 올바르게 수행할 수 있습니다.

Vector512 및 AVX-512

.NET Core 3.0은 x86/x64용 플랫폼별 하드웨어 내장 API를 포함하도록 SIMD 지원을 확장했습니다. .NET 5는 Arm64 및 .NET 7에 대한 지원을 추가하여 플랫폼 간 하드웨어 내장 함수를 추가했습니다. .NET 8은 Intel Advanced Vector Extensions 512(AVX-512) 지침에 대한 Vector512<T> 및 지원을 도입하여 SIMD 지원을 강화합니다.

특히 .NET 8에는 AVX-512의 다음 주요 기능에 대한 지원이 포함되어 있습니다.

- 512비트 벡터 작업
- 추가 16개의 SIMD 레지스터
- 128비트, 256비트 및 512비트 벡터에 사용할 수 있는 추가 지침

이제 기능을 지원하는 하드웨어가 있는 경우, Vector512.IsHardwareAccelerated이(가) `true`을(를) 보고합니다.

.NET 8은 또한 System.Runtime.Intrinsics.X86 네임스페이스 아래에 여러 플랫폼별 클래스를 추가합니다.

- Avx512F(기본)
- Avx512BW(바이트 및 단어)
- Avx512CD(충돌 탐지)
- Avx512DQ(더블워드 및 쿼드워드)
- Avx512Vbmi(벡터 바이트 조작 명령)

이러한 클래스는 64비트 프로세스에서만 사용할 수 있는 명령에 대해 IsSupported 속성 및 중첩된 Avx512F.X64 클래스를 노출한다는 측면에서 ISA(명령 집합 아키텍처)와 동일

한 일반 셰이프를 따릅니다. 또한 각 클래스에는 해당 명령 집합에 대한 `Avx512VL`(벡터 길이) 확장을 노출하는 중첩된 Avx512F.VL 클래스가 있습니다.

코드에서 `Vector512`특정 또는 `Avx512F`특정 지침을 명시적으로 사용하지 않더라도 새 AVX-512 지원을 계속 활용할 수 있습니다. JIT는 Vector128<T> 또는 Vector256<T>사용할 때 암시적으로 추가 레지스터 및 지침을 활용할 수 있습니다. 기본 클래스 라이브러리는 Span<T> 및 ReadOnlySpan<T> 의해 노출되는 대부분의 작업과 기본 형식에 대해 노출된 많은 수학 API에서 내부적으로 이러한 하드웨어 내장 함수를 사용합니다.

데이터 유효성 검사

System.ComponentModel.DataAnnotations 네임스페이스에는 클라우드 네이티브 서비스의 유효성 검사 시나리오를 위한 새 데이터 유효성 검사 특성이 포함되어 있습니다. 기존 `DataAnnotations` 유효성 검사기는 양식의 필드와 같은 일반적인 UI 데이터 항목 유효성 검사에 맞춰지지만 새 특성은 구성 옵션과 같은 비 사용자 항목 데이터의 유효성을 검사하도록 설계되었습니다. 새 특성 외에도 새 속성이 RangeAttribute 형식에 추가되었습니다.

테이블 확장

새 API	묘사
RangeAttribute.MinimumIsExclusive RangeAttribute.MaximumIsExclusive	경계가 허용 범위에 포함되는지 여부를 지정합니다.
System.ComponentModel.DataAnnotations.LengthAttribute	문자열 또는 컬렉션의 하한과 상한을 모두 지정합니다. 예를 들어 `[Length(10, 20)]` 컬렉션에 10개 이상의 요소와 최대 20개의 요소가 필요합니다.
System.ComponentModel.DataAnnotations.Base64StringAttribute	문자열이 유효한 Base64 표현인지 확인합니다.
System.ComponentModel.DataAnnotations.AllowedValuesAttribute System.ComponentModel.DataAnnotations.DeniedValuesAttribute	허용 목록과 거부 목록을 각각 지정합니다. 예를 들어 `[AllowedValues("apple", "banana", "mango")]`.

지표

새 API를 사용하면 개체를 만들 때 Meter 및 Instrument 개체에 키-값 쌍 태그를 연결할 수 있습니다. 게시된 메트릭 측정값의 집계자는 태그를 사용하여 집계된 값을 구분할 수

있습니다.

```
C#
```

```
var options = new MeterOptions("name")
{
    Version = "version",
    // Attach these tags to the created meter.
    Tags = new TagList()
    {
        { "MeterKey1", "MeterValue1" },
        { "MeterKey2", "MeterValue2" }
    }
};

Meter meter = meterFactory!.Create(options);

Counter<int> counterInstrument = meter.CreateCounter<int>(
    "counter", null, null, new TagList() { { "counterKey1", "counterValue1"
} }
);
counterInstrument.Add(1);
```

새 API에는 다음이 포함됩니다.

- MeterOptions
- Meter(MeterOptions)
- CreateCounter<T>(String, String, String, IEnumerable<KeyValuePair<String,Object>>)

암호화

.NET 8은 SHA-3 해시 기본 형식에 대한 지원을 추가합니다. (SHA-3은 현재 OpenSSL 1.1.1 이상 및 Windows 11 빌드 25324 이상을 사용하는 Linux에서 지원됩니다.) SHA-2를 사용할 수 있는 API는 이제 SHA-3 칭찬을 제공합니다. 여기에는 해시에 대한 `SHA3_256`, `SHA3_384` 및 `SHA3_512` 포함됩니다. HMAC에 대한 `HMACSHA3_256`, `HMACSHA3_384` 및 `HMACSHA3_512`. 알고리즘을 구성할 수 있는 해시에 대한 `HashAlgorithmName.SHA3_256`, `HashAlgorithmName.SHA3_384` 및 `HashAlgorithmName.SHA3_512`. RSA OAEP 암호화에 대한 `RSAEncryptionPadding.OaepSHA3_256`, `RSAEncryptionPadding.OaepSHA3_384` 및 `RSAEncryptionPadding.OaepSHA3_512`.

다음 예제에서는 플랫폼이 SHA-3을 지원하는지 여부를 확인하기 위해 `SHA3_256.IsSupported` 속성을 포함하여 API를 사용하는 방법을 보여 줍니다.

```
C#
```

```
// Hashing example
if (SHA3_256.IsSupported)
{
    byte[] hash = SHA3_256.HashData(dataToHash);
}
else
{
    // ...
}

// Signing example
if (SHA3_256.IsSupported)
{
     using ECDsa ec = ECDsa.Create(ECCurve.NamedCurves.nistP256);
     byte[] signature = ec.SignData(dataToBeSigned,
HashAlgorithmName.SHA3_256);
}
else
{
    // ...
}
```

SHA-3 지원은 현재 암호화 기본 형식을 지원하기 위한 것입니다. 상위 수준 생성 및 프로토콜은 처음에 SHA-3을 완전히 지원하지 않을 것으로 예상됩니다. 이러한 프로토콜에는 X.509 인증서, SignedXml및 COSE가 포함됩니다.

네트워킹

HTTPS 프록시 지원

지금까지 HttpClient 지원되는 프록시 형식은 모두 "man-in-the-middle"을 허용하여 HTTPS URI에 대해서도 클라이언트가 연결 중인 사이트를 확인할 수 있도록 했습니다. 이제 HttpClient는 클라이언트와 프록시 간에 암호화된 채널을 생성하여 모든 요청을 완전한 개인정보 보호 상태에서 처리할 수 있는 *HTTPS 프록시*를 지원합니다.

HTTPS 프록시를 사용하도록 설정하려면 `all_proxy` 환경 변수를 설정하거나 WebProxy 클래스를 사용하여 프로그래밍 방식으로 프록시를 제어합니다.

Unix: `export all_proxy=https://x.x.x.x:3218` Windows: `set all_proxy=https://x.x.x.x:3218`

WebProxy 클래스를 사용하여 프로그래밍 방식으로 프록시를 제어할 수도 있습니다.

스트림 기반 ZipFile 메서드

.NET 8에는 디렉터리에 포함된 모든 파일을 수집하고 압축한 다음 결과 zip 파일을 제공된 스트림에 저장할 수 있는 새로운 ZipFile.CreateFromDirectory 오버로드가 포함되어 있습니다. 마찬가지로 새 ZipFile.ExtractToDirectory 오버로드를 사용하면 압축된 파일이 포함된 스트림을 제공하고 파일 시스템에 해당 콘텐츠를 추출할 수 있습니다. 새 오버로드는 다음과 같습니다.

C#

```
namespace System.IO.Compression;

public static partial class ZipFile
{
    public static void CreateFromDirectory(
        string sourceDirectoryName, Stream destination);

    public static void CreateFromDirectory(
        string sourceDirectoryName,
        Stream destination,
        CompressionLevel compressionLevel,
        bool includeBaseDirectory);

    public static void CreateFromDirectory(
        string sourceDirectoryName,
        Stream destination,
        CompressionLevel compressionLevel,
        bool includeBaseDirectory,
    Encoding? entryNameEncoding);

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName) { }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, bool overwriteFiles)
{ }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, Encoding?
entryNameEncoding) { }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, Encoding?
entryNameEncoding, bool overwriteFiles) { }
}
```

이러한 새 API는 디스크 공간을 제한할 때 디스크를 중간 단계로 사용할 필요가 없으므로 유용할 수 있습니다.

확장 라이브러리

이 섹션에는 다음 하위 항목이 포함되어 있습니다.

- 옵션 유효성 검사
- LoggerMessageAttribute 생성자
- 확장 메트릭
- 호스팅된 수명 주기 서비스
- 키드 DI 서비스
- System.Numerics.Tensors.TensorPrimitives

키 지정된 DI 서비스

DI(키 종속성 주입) 서비스는 키를 사용하여 DI 서비스를 등록하고 검색하는 방법을 제공합니다. 키를 사용하여 서비스를 등록하고 사용하는 방법의 범위를 지정할 수 있습니다. 다음은 몇 가지 새로운 API입니다.

- IKeyedServiceProvider 인터페이스입니다.
- 생성자의 등록/확인에 사용된 키를 삽입하는 데 사용할 수 있는 ServiceKeyAttribute 특성입니다.
- 서비스 생성자 매개 변수에서 사용할 키 지정 서비스를 지정하는 데 사용할 수 있는 FromKeyedServicesAttribute 특성입니다.
- 키 지정된 서비스를 지원하기 위한 IServiceCollection 위한 다양한 새 확장 메서드(예: ServiceCollectionServiceExtensions.AddKeyedScoped.)
- IKeyedServiceProvider의 ServiceProvider 구현입니다.

다음 예제에서는 키 지정된 DI 서비스를 사용하는 방법을 보여 줍니다.

C#

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<BigCacheConsumer>();
builder.Services.AddSingleton<SmallCacheConsumer>();
builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
WebApplication app = builder.Build();
app.MapGet("/big", (BigCacheConsumer data) => data.GetData());
app.MapGet("/small", (SmallCacheConsumer data) => data.GetData());
app.MapGet("/big-cache", ([FromKeyedServices("big")] ICache cache) =>
cache.Get("data"));
app.MapGet("/small-cache", (HttpContext httpContext) =>
httpContext.RequestServices.GetRequiredKeyedService<ICache>
("small").Get("data"));
app.Run();

class BigCacheConsumer([FromKeyedServices("big")] ICache cache)
{
    public object? GetData() => cache.Get("data");
}
```

```
class SmallCacheConsumer(IServiceProvider serviceProvider)
{
    public object? GetData() =>
serviceProvider.GetRequiredKeyedService<ICache>("small").Get("data");
}

public interface ICache
{
    object Get(string key);
}

public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

자세한 내용은 dotnet/runtime#64427 참조하세요.

호스트된 수명 주기 서비스

이제 호스트된 서비스에는 애플리케이션 수명 주기 동안 더 많은 실행 옵션이 있습니다. IHostedService `StartAsync` 및 `StopAsync`제공되었으며, 이제 IHostedLifecycleService 다음과 같은 추가 메서드를 제공합니다.

- StartingAsync(CancellationToken)
- StartedAsync(CancellationToken)
- StoppingAsync(CancellationToken)
- StoppedAsync(CancellationToken)

이러한 메서드는 각각 기존 지점 전후에 실행됩니다.

다음 예제에서는 새 API를 사용하는 방법을 보여 줍니다.

C#

```
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

internal class HostedLifecycleServices
{
    public async static void RunIt()
```

```
    {
        IHostBuilder hostBuilder = new HostBuilder();
        hostBuilder.ConfigureServices(services =>
        {
            services.AddHostedService<MyService>();
        });

        using (IHost host = hostBuilder.Build())
        {
            await host.StartAsync();
        }
    }

    public class MyService : IHostedLifecycleService
    {
        public Task StartingAsync(CancellationToken cancellationToken) => /*
add logic here */ Task.CompletedTask;
        public Task StartAsync(CancellationToken cancellationToken) => /*
add logic here */ Task.CompletedTask;
        public Task StartedAsync(CancellationToken cancellationToken) => /*
add logic here */ Task.CompletedTask;
        public Task StopAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
        public Task StoppedAsync(CancellationToken cancellationToken) => /*
add logic here */ Task.CompletedTask;
        public Task StoppingAsync(CancellationToken cancellationToken) => /*
add logic here */ Task.CompletedTask;
    }
}
```

자세한 내용은 dotnet/runtime#86511 참조하세요.

옵션 유효성 검사

원본 생성기

시작 오버헤드를 줄이고 유효성 검사 기능 집합을 개선하기 위해 유효성 검사 논리를 구현하는 소스 코드 생성기를 도입했습니다. 다음 코드에서는 예제 모델 및 유효성 검사기 클래스를 보여 줍니다.

C#

```
public class FirstModelNoNamespace
{
    [Required]
    [MinLength(5)]
    public string P1 { get; set; } = string.Empty;

    [Microsoft.Extensions.Options.ValidateObjectMembers(
        typeof(SecondValidatorNoNamespace))]
```

```
    public SecondModelNoNamespace? P2 { get; set; }
}

public class SecondModelNoNamespace
{
    [Required]
    [MinLength(5)]
    public string P4 { get; set; } = string.Empty;
}

[OptionsValidator]
public partial class FirstValidatorNoNamespace
    : IValidateOptions<FirstModelNoNamespace>
{
}

[OptionsValidator]
public partial class SecondValidatorNoNamespace
    : IValidateOptions<SecondModelNoNamespace>
{
}
```

앱에서 종속성 주입을 사용하는 경우 다음 예제 코드와 같이 유효성 검사를 삽입할 수 있습니다.

C#

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
builder.Services.Configure<FirstModelNoNamespace>(
    builder.Configuration.GetSection("some string"));

builder.Services.AddSingleton<
    IValidateOptions<FirstModelNoNamespace>, FirstValidatorNoNamespace>();
builder.Services.AddSingleton<
    IValidateOptions<SecondModelNoNamespace>, SecondValidatorNoNamespace>();
```

ValidateOptionsResultBuilder 형식

.NET 8에서는 ValidateOptionsResult 개체를 쉽게 만들 수 있도록 ValidateOptionsResultBuilder 형식을 도입했습니다. 중요한 것은 이 빌더가 여러 오류의 누적을 가능하게 한다는 점입니다. 이전에는 IValidateOptions<TOptions>.Validate(String, TOptions) 구현하는 데 필요한 ValidateOptionsResult 개체를 만드는 것이 어려웠으며 때로는 계층화된 유효성 검사 오류가 발생했습니다. 여러 오류가 있는 경우 첫 번째 오류에서 유효성 검사 프로세스가 중지되는 경우가 많습니다.

다음 코드 조각은 ValidateOptionsResultBuilder사용하는 예제를 보여 줍니다.

C#

```
ValidateOptionsResultBuilder builder = new();
builder.AddError("Error: invalid operation code");
builder.AddResult(ValidateOptionsResult.Fail("Invalid request parameters"));
builder.AddError("Malformed link", "Url");

// Build ValidateOptionsResult object has accumulating multiple errors.
ValidateOptionsResult result = builder.Build();

// Reset the builder to allow using it in new validation operation.
builder.Clear();
```

LoggerMessageAttribute 생성자

이제 LoggerMessageAttribute 추가 생성자 오버로드를 제공합니다. 이전에는 매개 변수가 없는 생성자 또는 모든 매개 변수(이벤트 ID, 로그 수준 및 메시지)가 필요한 생성자를 선택해야 했습니다. 새 오버로드는 코드가 감소된 필수 매개 변수를 지정하는 데 더 큰 유연성을 제공합니다. 이벤트 ID를 제공하지 않으면 시스템에서 자동으로 생성됩니다.

C#

```
public LoggerMessageAttribute(LogLevel level, string message);
public LoggerMessageAttribute(LogLevel level);
public LoggerMessageAttribute(string message);
```

확장 지표

IMeterFactory 인터페이스

DI(종속성 주입) 컨테이너에 새 IMeterFactory 인터페이스를 등록하고 이를 사용하여 격리된 방식으로 Meter 개체를 만들 수 있습니다.

기본 미터 팩터리 구현을 사용하여 DI 컨테이너에 IMeterFactory 등록합니다.

C#

```
// 'services' is the DI IServiceCollection.
services.AddMetrics();
```

그런 다음 소비자는 미터 팩터리를 가져와서 새 Meter 개체를 만드는 데 사용할 수 있습니다.

C#

```
IMeterFactory meterFactory =
serviceProvider.GetRequiredService<IMeterFactory>();

MeterOptions options = new MeterOptions("MeterName")
{
    Version = "version",
};

Meter meter = meterFactory.Create(options);
```

MetricCollector<T> 클래스

새 MetricCollector<T> 클래스를 사용하면 타임스탬프와 함께 메트릭 측정값을 기록할 수 있습니다. 또한 클래스는 정확한 타임스탬프 생성을 위해 선택한 시간 공급자를 유연하게 사용할 수 있습니다.

C#

```
const string CounterName = "MyCounter";
DateTimeOffset now = DateTimeOffset.Now;

var timeProvider = new FakeTimeProvider(now);
using var meter = new Meter(Guid.NewGuid().ToString());
Counter<long> counter = meter.CreateCounter<long>(CounterName);
using var collector = new MetricCollector<long>(counter, timeProvider);

Assert.IsNull(collector.LastMeasurement);

counter.Add(3);

// Verify the update was recorded.
Assert.AreEqual(counter, collector.Instrument);
Assert.IsNotNull(collector.LastMeasurement);

Assert.AreSame(collector.GetMeasurementSnapshot().Last(),
collector.LastMeasurement);
Assert.AreEqual(3, collector.LastMeasurement.Value);
Assert.AreEqual(now, collector.LastMeasurement.Timestamp);
```

System.Numerics.Tensors.TensorPrimitives

업데이트된 System.Numerics.Tensors NuGet 패키지에는 텐서 작업에 대한 지원을 추가하는 새 System.Numerics.Tensors.TensorPrimitives 형식의 API가 포함되어 있습니다. 텐서 기본 형식은 AI 및 기계 학습과 같은 데이터 집약적 워크로드를 최적화합니다.

의미 체계 검색 및 RAG(검색 보강 세대)와 같은 AI 워크로드는 관련 데이터로 프롬프트를 보강하여 ChatGPT와 같은 대규모 언어 모델의 자연어 기능을 확장합니다. 이러한 워크로

드의 경우 질문에 대답할 가장 관련성이 큰 데이터를 찾기 위한 *코사인 유사성* 같은 벡터에 대한 작업이 매우 중요합니다. TensorPrimitives 형식은 벡터 작업에 대한 API를 제공합니다.

자세한 내용은 .NET 8 RC 2 출시를 알리는 블로그 게시물을 참조하세요.

기본 AOT 지원

네이티브 AOT으로 게시하는 옵션은 .NET 7에서 처음 도입되었습니다. Native AOT를 사용하여 앱을 게시하면 런타임이 필요하지 않은 완전히 자체 포함된 버전의 앱이 만들어집니다. 모든 것이 단일 파일에 포함됩니다. .NET 8은 네이티브 AOT 게시에 다음과 같은 개선 사항을 제공합니다.

- *macOS*x64 및 Arm64 아키텍처에 대한 지원을 추가합니다.
- Linux에서 네이티브 AOT 앱의 크기를 최대 50%줄입니다. 다음 표에서는 .NET 7과 .NET 8의 전체 .NET 런타임을 포함하는 네이티브 AOT를 사용하여 게시된 "Hello World" 앱의 크기를 보여 니다.

  테이블 확장

운영 체제	.NET 7	.NET 8
Linux x64(`-p:StripSymbols=true` 포함)	3.76MB	1.84MB
Windows x64	2.85MB	1.77MB

- 최적화 기본 설정(크기 또는 속도)을 지정할 수 있습니다. 기본적으로 컴파일러는 애플리케이션의 크기를 염두에 두고 빠른 코드를 생성하도록 선택합니다. 그러나 `<OptimizationPreference>` MSBuild 속성을 사용하여 하나 또는 다른 속성에 대해 특별히 최적화할 수 있습니다. 자세한 내용은 AOT 배포 최적화를 참조하세요.

네이티브 AOT를 사용하여 iOS와 유사한 플랫폼 대상 지정

.NET 8은 iOS와 유사한 플랫폼에 대해 네이티브 AOT 지원을 사용하도록 설정하는 작업을 시작합니다. 이제 다음 플랫폼에서 Native AOT를 사용하여 .NET iOS 및 .NET MAUI 애플리케이션을 빌드하고 실행할 수 있습니다.

- `ios`
- `iossimulator`
- `maccatalyst`
- `tvos`

- `tvossimulator`

예비 테스트에 따르면 모노 대신 네이티브 AOT를 사용하는 .NET iOS 앱의 경우 디스크의 앱 크기가 약 35% 감소합니다. .NET MAUI iOS 앱용 디스크의 앱 크기는 최대 50%감소합니다. 또한 시작 시간도 더 빠릅니다. .NET iOS 앱은 시작 시간이 약%% 더 빠르고, .NET MAUI iOS 앱은 Mono에 비해 시작 성능이 약%% 더 우수합니다. .NET 8 지원은 실험적이며 기능 전체의 첫 번째 단계일 뿐입니다. 보다 자세한 내용은 .NET MAUI 블로그 게시물의 .NET 8 성능 향상을 참조하세요.

네이티브 AOT 지원은 앱 배포를 위한 옵트인 기능으로 사용할 수 있습니다. Mono는 여전히 앱 개발 및 배포의 기본 런타임입니다. iOS 디바이스에서 Native AOT를 사용하여 .NET MAUI 애플리케이션을 빌드하고 실행하려면 `dotnet workload install maui` 사용하여 .NET MAUI 워크로드를 설치하고 `dotnet new maui -n HelloMaui` 앱을 만듭니다. 그런 다음 프로젝트 파일에서 MSBuild 속성 `PublishAot`을 `true`으로 설정합니다.

XML

```xml
<PropertyGroup>
  <PublishAot>true</PublishAot>
</PropertyGroup>
```

다음 예제와 같이 필수 속성을 설정하고 `dotnet publish` 실행하면 네이티브 AOT를 사용하여 앱이 배포됩니다.

.NET CLI

```
dotnet publish -f net8.0-ios -c Release -r ios-arm64  /t:Run
```

제한

모든 iOS 기능이 네이티브 AOT와 호환되는 것은 아닙니다. 마찬가지로 iOS에서 일반적으로 사용되는 모든 라이브러리가 NativeAOT와 호환되는 것은 아닙니다. 또한 네이티브 AOT 배포 기존제한 사항 외에도 다음 목록에서는 iOS와 유사한 플랫폼을 대상으로 지정할 때 몇 가지 다른 제한 사항을 보여 줍니다.

- 네이티브 AOT 사용은 앱 배포(`dotnet publish`) 중에만 사용하도록 설정됩니다.
- 관리 코드 디버깅은 Mono에서만 지원됩니다.
- .NET MAUI 프레임워크와의 호환성이 제한됩니다.

Android 앱용 AOT 컴파일

앱 크기를 줄이기 위해 Android를 대상으로 하는 .NET 및 .NET MAUI 앱은 릴리스 모드에서 빌드될 때 *프로파일된* AOT(미리) 컴파일 모드를 사용합니다. 프로파일된 AOT 컴파일은 일반 AOT 컴파일보다 더 적은 수의 메서드에 영향을 줍니다. .NET 8에는 Android 앱에 대한 AOT 컴파일을 추가로 옵트인하여 앱 크기를 훨씬 더 줄일 수 있는 `<AndroidStripILAfterAOT>` 속성이 도입되었습니다.

XML

```
<PropertyGroup>
  <AndroidStripILAfterAOT>true</AndroidStripILAfterAOT>
</PropertyGroup>
```

기본적으로 `AndroidStripILAfterAOT`에서 `true`로 설정하면 기본 `AndroidEnableProfiledAot` 설정이 재정의되어 AOT 컴파일된 거의 모든 메서드를 트리밍할 수 있습니다. 두 속성을 모두 `true`로 명시적으로 설정하여 프로파일된 AOT와 IL 스트리핑을 함께 사용할 수도 있습니다.

XML

```
<PropertyGroup>
  <AndroidStripILAfterAOT>true</AndroidStripILAfterAOT>
  <AndroidEnableProfiledAot>true</AndroidEnableProfiledAot>
</PropertyGroup>
```

교차 빌드된 Windows 앱

Windows가 아닌 플랫폼에서 Windows를 대상으로 하는 앱을 빌드할 때 결과 실행 파일은 이제 지정된 Win32 리소스(예: 애플리케이션 아이콘, 매니페스트, 버전 정보)로 업데이트됩니다.

이전에는 이러한 리소스를 갖기 위해 Windows에서 애플리케이션을 빌드해야 했습니다. 인프라 복잡성과 리소스 사용량 모두에 영향을 미치는 중요한 문제이기 때문에 크로스 빌딩 지원에서 이러한 격차를 해결하는 것이 인기 있는 요청이었습니다.

참고 항목

- .NET 8의 새로운 기능

.NET 8용 SDK 및 도구의 새로운 기능

2025. 09. 30.

이 문서에서는 .NET SDK 및 .NET 8용 도구의 새로운 기능에 대해 설명합니다.

SDK

이 섹션에는 다음 하위 항목이 포함되어 있습니다.

- CLI 기반 프로젝트 평가
- 터미널 빌드 출력
- 간소화된 출력 경로
- 'dotnet workload clean' 명령
- 'dotnet publish' 및 'dotnet pack' 자산
- dotnet restore 보안 감사
- 템플릿 엔진
- 원본 링크
- 원본 빌드 SDK

CLI 기반 프로젝트 평가

MSBuild에는 MSBuild의 데이터를 스크립트 또는 도구에 쉽게 통합할 수 있는 새로운 기능이 포함되어 있습니다. dotnet publish와 같은 CLI 명령에 다음 새 플래그를 사용하여 CI 파이프라인 및 다른 곳에서 사용할 데이터를 가져올 수 있습니다.

테이블 확장

Flag	Description
`--getProperty:<PROPERTYNAME>`	지정된 이름의 MSBuild 속성을 검색합니다.
`--getItem:<ITEMTYPE>`	지정된 형식의 MSBuild 항목을 검색합니다.
`--getTargetResult:<TARGETNAME>`	지정된 대상을 실행하여 출력을 검색합니다.

값은 표준 출력에 기록됩니다. 다음 예제와 같이 여러 값 또는 복합 값이 JSON으로 출력됩니다.

.NET CLI

```
>dotnet publish --getProperty:OutputPath
bin\Release\net8.0\
```

.NET CLI

```
>dotnet publish -p PublishProfile=DefaultContainer --getProperty:GeneratedContainerDigest --
getProperty:GeneratedContainerConfiguration
{
  "Properties": {
    "GeneratedContainerDigest":
"sha256:ef880a503bbabcb84bbb6a1aa9b41b36dc1ba08352e7cd91c0993646675174c4",
```

```
    "GeneratedContainerConfiguration": "{\u0022config\u0022:{\u0022ExposedPorts\u0022:
{\u00228080/tcp\u0022:{}},\u0022Labels\u0022...}}"
  }
}
```

.NET CLI

```
>dotnet publish -p PublishProfile=DefaultContainer --getItem:ContainerImageTags
{
  "Items": {
    "ContainerImageTags": [
      {
        "Identity": "latest",
        ...
    ]
  }
}
```

터미널 빌드 출력

`dotnet build` 에는 보다 현대화된 빌드 출력을 생성하는 새로운 옵션이 있습니다. 이 *터미널 로거* 출력은 오류가 생성된 프로젝트와 함께 그룹화되고, 다중 대상 프로젝트에 대한 다양한 대상 프레임워크를 더 잘 구분하며, 빌드가 수행하는 작업에 대한 실시간 정보를 제공합니다. 새 출력을 옵트인하려면 이 `--tl` 옵션을 사용합니다. 이 옵션에 대한 자세한 내용은 dotnet 빌드 옵션을 참조하세요.

간소화된 출력 경로

.NET 8에는 빌드 출력에 대한 출력 경로 및 폴더 구조를 간소화하는 옵션이 도입되었습니다. 이전에는 .NET 앱이 다양한 빌드 아티팩트를 위한 깊고 복잡한 출력 경로 집합을 생성했습니다. 간소화된 새로운 출력 경로 구조는 모든 빌드 출력을 공통 위치로 수집하므로 도구를 쉽게 예측할 수 있습니다.

자세한 내용은 아티팩트 출력 레이아웃을 참조하세요.

`dotnet workload clean` 명령

.NET 8에는 여러 .NET SDK 또는 Visual Studio 업데이트를 통해 남아 있을 수 있는 워크로드 팩을 정리하는 새 명령이 도입되었습니다. `workload clean`를 사용하여 워크로드를 관리할 때 문제가 발생하는 경우, 다시 시도하기 전에 알려진 상태로 안전하게 복원하는 것이 좋습니다. 명령에는 두 가지 모드가 있습니다.

- `dotnet workload clean`

  파일 기반 또는 MSI 기반 워크로드의 고아 팩을 정리하는 워크로드 가비지 수집을 실행합니다. 고아 팩은 제거된 .NET SDK 버전 또는 더 이상 설치 기록이 존재하지 않는 팩을 의미합니다.

  Visual Studio가 설치된 경우 명령은 Visual Studio를 사용하여 수동으로 정리해야 하는 워크로드도 나열합니다.

- `dotnet workload clean --all`

  이 모드는 보다 강력하여 현재 SDK 워크로드 설치 유형에 해당하는 머신의 모든 팩을 정리합니다(Visual Studio에서 설치된 것이 아닌 경우). 또한 실행 중인 .NET SDK 기능 밴드 및 아래에 대한 모든 워크로드 설

치 레코드를 제거합니다.

`dotnet publish` 및 `dotnet pack` 자산

dotnet publish 및 dotnet pack 명령은 생산 자산을 생성하기 위한 것이므로 이제 기본적으로 `Release` 자산을 생성합니다.

다음 출력은 `dotnet build`와 `dotnet publish`의 서로 다른 동작을 보여 주고, `Debug` 자산을 게시 상태로 되돌리기 위해 `PublishRelease` 속성을 `false`로 설정하는 방법을 설명합니다.

콘솔

```
/app# dotnet new console
/app# dotnet build
  app -> /app/bin/Debug/net8.0/app.dll
/app# dotnet publish
  app -> /app/bin/Release/net8.0/app.dll
  app -> /app/bin/Release/net8.0/publish/
/app# dotnet publish -p:PublishRelease=false
  app -> /app/bin/Debug/net8.0/app.dll
  app -> /app/bin/Debug/net8.0/publish/
```

자세한 내용은 'dotnet pack'이 릴리스 구성을 사용하고'dotnet publish'가 릴리스 구성을 사용하는 경우를 참조하세요.

`dotnet restore` 보안 감사

.NET 8부터 종속성 패키지가 복원될 때 알려진 취약성에 대한 보안 검사를 옵트인할 수 있습니다. 이 감사에서는 영향을 받은 패키지 이름, 취약성의 심각도 및 자세한 내용을 볼 수 있는 권고 링크가 포함된 보안 취약성 보고서를 생성합니다. `dotnet add` 또는 `dotnet restore`을 실행하면 발견된 취약성에 대해 경고 NU1901-NU1904가 표시됩니다. 자세한 내용은 보안 취약성에 대한 감사를 참조하세요.

템플릿 엔진

템플릿 엔진 은 NuGet의 보안 관련 기능 중 일부를 통합하여 .NET 8에서 보다 안전한 환경을 제공합니다. 향상된 기능은 다음과 같습니다.

- 기본적으로 피드에서 `http://` 패키지 다운로드를 방지합니다. 예를 들어 다음 명령은 원본 URL이 HTTPS를 사용하지 않으므로 템플릿 패키지를 설치하지 못합니다.

  `dotnet new install console --add-source "http://pkgs.dev.azure.com/dnceng/public/_packaging/dotnet-public/nuget/v3/index.json"`

  `--force` 플래그를 사용하여 이 제한을 재정의할 수 있습니다.

- `dotnet new`, `dotnet new install`, `dotnet new update` 각각의 템플릿 패키지에서 알려진 취약성을 확인합니다. 취약성이 발견되고 계속 진행하려는 경우 플래그를 `--force` 사용해야 합니다.

- 의 경우 `dotnet new`템플릿 패키지 소유자에 대한 정보를 제공합니다. 소유권은 NuGet 포털에서 확인되며 신뢰할 수 있는 특성으로 간주될 수 있습니다.

- `dotnet search` 및 `dotnet uninstall`에 대해 "신뢰할 수 있는" 패키지, 즉 예약된 접두사를 사용하는 패키지에서 템플릿이 설치되어 있는지 여부를 나타냅니다.

원본 링크

소스 링크 는 이제 .NET SDK에 포함됩니다. 목표는 원본 링크를 SDK에 번들로 묶어 패키지에 대해 별도로 `<PackageReference>` 요구하는 대신, 더 많은 패키지에 기본적으로 이 정보가 포함된다는 것입니다. 이 정보는 개발자를 위한 IDE 환경을 향상시킵니다.

> ⓘ **참고**
>
> 이러한 변경의 부작용으로, .NET 7 또는 이전 버전을 대상으로 하는 라이브러리 및 애플리케이션의 `InformationalVersion` 값에 커밋 정보가 포함됩니다. 자세한 내용은 **.NET SDK에 포함된 원본 링크를 참조하세요**.

소스 코드 빌드 SDK

Linux 배포 빌드(원본 빌드) SDK에는 이제 원본 빌드 런타임 패키지를 사용하여 자체 포함 애플리케이션을 빌드하는 기능이 있습니다. 배포별 런타임 패키지는 원본 빌드 SDK와 함께 번들로 제공됩니다. 독립적 배포 과정에서 이 번들된 런타임 패키지가 참조되어, 사용자에게 기능을 활성화할 수 있습니다.

네이티브 AOT 콘솔 앱 템플릿

기본 콘솔 앱 템플릿에는 이제 AOT 지원이 별도의 설정 없이 기본적으로 포함됩니다. AOT 컴파일용으로 구성된 프로젝트를 만들려면 실행 `dotnet new console --aot`하기만 하면됩니다. 추가된 `--aot` 프로젝트 구성에는 다음 세 가지 효과가 있습니다.

- 프로젝트를 게시할 때, 예를 들어 Visual Studio 또는 `dotnet publish`를 사용할 때, Native AOT로 네이티브 자체 포함 실행 파일을 생성합니다.
- 트리밍, AOT 및 단일 파일에 대한 호환성 분석기를 사용하도록 설정합니다. 이러한 분석기는 프로젝트의 문제가 있을 수 있는 부분에 대해 경고합니다(있는 경우).
- AOT 컴파일 없이 프로젝트를 디버그할 때 AOT와 비슷한 환경을 얻을 수 있도록 AOT의 디버그 시간 에뮬레이션을 사용하도록 설정합니다. 예를 들어 AOT에 주석이 추가되지 않았으므로 호환성 분석기에서 누락된 NuGet 패키지에서 사용하는 System.Reflection.Emit 경우 에뮬레이션은 AOT를 사용하여 프로젝트를 게시하려고 할 때 놀랄 일이 없다는 것을 의미합니다.

Linux의 .NET

Linux에 대한 최소 지원 기준

Linux에 대한 최소 지원 기준이 .NET 8에 대해 업데이트되었습니다. .NET은 모든 아키텍처에 대해 Ubuntu 16.04를 대상으로 빌드됩니다. 이는 .NET 8의 최소 `glibc` 버전을 정의하는 데 주로 중요합니다. .NET 8은 Ubuntu 14.04 또는 Red Hat Enterprise Linux 7과 같은 이전 glibc를 포함하는 배포판 버전에서 시작하지 못합니다.

자세한 내용은 Red Hat Enterprise Linux 제품군 지원을 참조하세요.

Linux에서 고유한 .NET 빌드

이전 .NET 버전에서는 원본에서 .NET을 빌드할 수 있었지만 릴리스에 해당하는 dotnet/installer 리포지토리 커밋에서 "원본 tarball"을 만들어야 했습니다. .NET 8에서는 더 이상 필요하지 않으며 dotnet/dotnet 리포지토리에서 직접 Linux에서 .NET을 빌드할 수 있습니다. 해당 리포지토리는 dotnet/source-build 를 사용하여 .NET 런타임, 도구 및 SDK를 빌드합니다. Red Hat 및 Canonical에서 .NET을 빌드하는 데 사용하는 것과 동일한 빌드입니다.

컨테이너 이미지에는 필요한 모든 종속성이 포함되어 있으므로 컨테이너에서 `dotnet-buildtools/prereqs` 빌드하는 것이 대부분의 사람들에게 가장 쉬운 방법입니다. 자세한 내용은 빌드 지침을 참조하세요.

NuGet 서명 확인

.NET 8부터 NuGet은 기본적으로 Linux에서 서명된 패키지를 확인합니다. NuGet은 Windows에서도 서명된 패키지를 계속 확인합니다.

대부분의 사용자는 확인을 알 수 없습니다. 그러나 */etc/pki/ca-trust/extracted/pem/objsign-ca-bundle.pem*에 있는 기존 루트 인증서 번들이 있는 경우 NU3042 경고와 함께 트러스트 오류가 표시될 수 있습니다.

환경 변수 `DOTNET_NUGET_SIGNATURE_VERIFICATION`를 `false`로 설정하여 검증을 비활성화할 수 있습니다.

코드 분석

.NET 8에는 .NET 라이브러리 API를 정확하고 효율적으로 사용하고 있는지 확인하는 데 도움이 되는 몇 가지 새로운 코드 분석기 및 수정기가 포함되어 있습니다. 다음 표에서는 새 분석기를 요약합니다.

테이블 확장

규칙 ID	카테고리	Description
CA1856	Performance	매개변수에 ConstantExpectedAttribute 속성이 올바르게 적용되지 않을 때 발생합니다.
CA1857	Performance	ConstantExpectedAttribute로 매개 변수에 주석이 추가되었지만 제공된 인수가 상수가 아닌 경우에 발생합니다.
CA1858	Performance	문자열이 지정된 접두사로 시작하는지 여부를 확인하려면 호출한 다음 결과를 0과 비교하는 것보다 호출 String.StartsWithString.IndexOf 하는 것이 좋습니다.
CA1859	Performance	이 규칙은 가능하면 특정 지역 변수, 필드, 속성, 메서드 매개 변수 및 메서드 반환 형식의 형식을 인터페이스 또는 추상 형식에서 구체적인 형식으로 업그레이드하는 것이 좋습니다. 구체적인 형식을 사용하면 더 높은 품질의 생성된 코드가 생성됩니다.
CA1860	Performance	컬렉션 형식에 요소가 있는지 확인할 때 `Length`, `Count`, 또는 `IsEmpty`를 사용하는 것이 Enumerable.Any를 호출하는 것보다 더 좋습니다.
CA1861	Performance	인수로 전달된 상수 배열은 반복적으로 호출될 때 다시 사용되지 않으며, 이는 매번 새 배열이 생성됨을 의미합니다. 성능을 향상시키려면 배열을 정적 읽기 전용 필드로 추출하는 것이 좋습니다.
CA1865-CA1867	Performance	char 오버로드는 단일 문자가 있는 문자열에 대해 성능이 뛰어난 오버로드입니다.
CA2021	Reliability	Enumerable.Cast<TResult>(IEnumerable)와 Enumerable.OfType<TResult>(IEnumerable)는 올바르게 작동하려면 호환되는 형식을 필요로 합니다. 확대 및 사용자 정의 변환은 제네릭 형식에서 지원되지

규칙 ID	카테고리	Description
		않습니다.
CA1510-CA1513	유지 관리	Throw 도우미는 새 예외 인스턴스를 `if` 생성하는 블록보다 더 간단하고 효율적입니다. 이러한 4개의 분석기는 다음과 같은 예외에 대해 생성되었습니다. ArgumentNullExceptionArgumentExceptionArgumentOutOfRangeExceptionObjectDisposedException

Diagnostics

C# 핫 다시 로드는 제네릭 수정을 지원합니다.

.NET 8부터 C# 핫 다시 로드 는 제네릭 형식 및 제네릭 메서드 수정을 지원합니다. Visual Studio를 사용하여 콘솔, 데스크톱, 모바일 또는 WebAssembly 애플리케이션을 디버그하는 경우 C# 코드 또는 Razor 페이지에서 제네릭 클래스 및 제네릭 메서드에 변경 내용을 적용할 수 있습니다. 자세한 내용은 Roslyn에서 지원하는 편집의 전체 목록을 참조하세요.

참고하십시오

- .NET 8의 새로운 기능

.NET 8용 컨테이너의 새로운 기능

아티클 • 2025. 01. 08.

이 문서에서는 .NET 8용 컨테이너의 새로운 기능에 대해 설명합니다.

컨테이너 이미지

.NET 8용 .NET 컨테이너 이미지는 다음과 같이 변경되었습니다.

- 루트가 아닌 사용자
- Debian 12
- 다듬어진 Ubuntu 이미지
- 다중 플랫폼 컨테이너 이미지 빌드
- ASP.NET 복합 이미지

루트가 아닌 사용자

이미지에는 `non-root` 사용자가 포함되었습니다. 이 사용자는 이미지를 `non-root`에 맞도록 설정합니다. `non-root`으로 실행하려면 Dockerfile의 끝에 다음 줄을 추가하세요(또는 Kubernetes 매니페스트에서 비슷한 명령어를 추가하세요).

Dockerfile

```
USER app
```

.NET 8은 1654인 `non-root` 사용자에 대한 UID에 대한 환경 변수를 추가합니다. 이 환경 변수는 Kubernetes `runAsNonRoot` 테스트에 유용하며, 컨테이너 사용자는 이름이 아닌 UID를 통해 설정해야 합니다. 이 dockerfile 사용 예제를 보여 줍니다.

기본 포트가 또한 포트 `80`에서 `8080`로 변경되었습니다. 이 변경을 지원하기 위해 포트를 보다 쉽게 변경할 수 있도록 새 환경 변수 `ASPNETCORE_HTTP_PORTS` 사용할 수 있습니다. 변수는 `ASPNETCORE_URLS`필요한 형식보다 간단한 포트 목록을 허용합니다. 이러한 변수 중 하나를 사용하여 포트를 다시 `80`로 변경하는 경우 `non-root`로 실행할 수 없습니다.

자세한 내용은 기본 ASP.NET Core 포트가 80에서 8080으로 변경되고 Linux 이미지루트가 아닌 새 '앱' 사용자를 참조하세요.

Debian 12

이제 컨테이너 이미지는 Debian 12(Bookworm) 사용합니다. Debian은 .NET 컨테이너 이미지의 기본 Linux 배포판입니다.

자세한 내용을 보려면 Debian 12로 업그레이드된 Debian 컨테이너 이미지를 참조하세요.

잘 다듬어진 Ubuntu 이미지

Chiseled Ubuntu 이미지는 .NET 8에서 사용할 수 있습니다. 치즐 이미지는 매우 작고 패키지 관리자나 셸이 없으며 `non-root`으로 인해 공격 가능 표면이 줄어듭니다. 이 유형의 이미지는 어플라이언스 스타일 컴퓨팅의 이점을 원하는 개발자를 위한 것입니다.

깎은 이미지는 기본적으로 세계화를 지원하지 않습니다. extra 이미지는 `icu` 및 `tzdata` 패키지를 포함하여 제공됩니다.

세계화 및 컨테이너에 대한 자세한 내용은 세계화 테스트 앱 참조하세요.

다중 플랫폼 컨테이너 이미지 빌드

Docker는 여러 환경에서 작동하는 다중 플랫폼 이미지 사용 및 빌드를 지원합니다. .NET 8에는 빌드한 .NET 이미지와 아키텍처를 혼합하고 일치시킬 수 있는 새로운 패턴이 도입되었습니다. 예를 들어 macOS를 사용하고 Azure에서 x64 클라우드 서비스를 대상으로 지정하려는 경우 다음과 같이 `--platform` 스위치를 사용하여 이미지를 빌드할 수 있습니다.

```
docker build --pull -t app --platform linux/amd64
```

이제 .NET SDK는 복원 시 `$TARGETARCH` 값과 `-a` 인수를 지원합니다. 다음 코드 조각은 예제를 보여줍니다.

Dockerfile

```
RUN dotnet restore -a $TARGETARCH

# Copy everything else and build app.
COPY aspnetapp/. .
RUN dotnet publish -a $TARGETARCH --self-contained false --no-restore -o
/app
```

자세한 내용은 멀티플랫폼 컨테이너 지원 개선 블로그 게시물을 참조하세요.

ASP.NET 복합 이미지

컨테이너화 성능을 개선하기 위한 노력의 일환으로, 런타임의 복합 버전이 있는 새로운 ASP.NET Docker 이미지를 사용할 수 있습니다. 이 컴포지트는 여러 CIL 어셈블리를 R2R(즉시 실행) 출력 바이너리로 컴파일하여 빌드됩니다. 이러한 어셈블리는 단일 이미지에 포함되므로 지팅 시간이 줄어들고 앱의 시작 성능이 향상됩니다. 일반 ASP.NET 이미지에 비해 복합 이미지의 다른 큰 장점은 복합 이미지의 크기가 디스크에서 더 작다는 것입니다.

주의해야 할 사항이 있습니다. 복합체에는 여러 어셈블리가 하나로 내장되어 있어 버전 간 결합이 더 엄격해집니다. 앱은 사용자 지정 버전의 프레임워크 또는 ASP.NET 이진 파일을 사용할 수 없습니다.

복합 이미지 는 `mcr.microsoft.com/dotnet/aspnet` 리포지토리에서 Alpine Linux, Ubuntu("자미") 치즐 및 마리너 배포판 플랫폼용으로 제공됩니다. 태그는 `-composite` 접미사가 붙은 형태로 ASP.NET Docker 페이지 에 나열됩니다.

컨테이너 게시

- 생성된 이미지의 기본값은
- 성능 및 호환성
- 인증
- tar.gz 아카이브에 게시

생성 이미지 기본값

`dotnet publish` 컨테이너 이미지를 생성할 수 있습니다. 기본적으로 이미지를 생성하므로 앱이 기본적으로 안전하게 유지됩니다. `ContainerUser` 속성(예: `root`)을 설정하여 언제든지 이 기본값을 변경합니다.

이제 기본 출력 컨테이너 태그가 `latest`. 이 기본값은 컨테이너 공간의 다른 도구와 일치하며 내부 개발 루프에서 컨테이너를 더 쉽게 사용할 수 있도록 합니다.

성능 및 호환성

.NET 8은 컨테이너를 원격 레지스트리, 특히 Azure 레지스트리에 푸시하기 위한 성능이 향상되었습니다. 속도 향상은 계층을 한 번의 작업으로 푸시하는 것에서 비롯되며, 원자성 업로드를 지원하지 않는 레지스트리의 경우 보다 안정적인 청크 처리 메커니즘을 제공합니다.

또한 이러한 향상된 기능은 더 많은 레지스트리(Harbor, Artifactory, Quay.io 및 Podman)가 지원됨을 의미합니다.

인증

.NET 8은 레지스트리에 컨테이너를 푸시할 때 OAuth 토큰 교환 인증(Azure 관리 ID)에 대한 지원을 추가합니다. 이 지원은 이제 인증 오류 없이 Azure Container Registry와 같은 레지스트리로 푸시할 수 있음을 의미합니다. 다음 명령은 예제 게시 흐름을 보여 줍니다.

콘솔

```
> az acr login -n <your registry name>
> dotnet publish -r linux-x64 -p PublishProfile=DefaultContainer
```

.NET 앱을 dotnet publish로 컨테이너화하는 방법에 대한 자세한 내용은 .NET 앱을 컨테이너화하기을 참조하세요.

tar.gz 아카이브에 게시하기

.NET 8부터 컨테이너를 *tar.gz* 보관 파일로 직접 만들 수 있습니다. 이 기능은 워크플로가 간단하지 않고 이미지를 푸시하기 전에 이미지에 대해 검사 도구를 실행해야 하는 경우에 유용합니다. 보관이 만들어지면 이동하거나 스캔하거나 로컬 Docker 도구체인에 로드할 수 있습니다.

보관 파일에 게시하려면 `dotnet publish` 명령에 `ContainerArchiveOutputPath` 속성을 추가합니다. 예를 들면 다음과 같습니다.

.NET CLI

```
dotnet publish \
  -p PublishProfile=DefaultContainer \
  -p ContainerArchiveOutputPath=./images/sdk-container-demo.tar.gz
```

폴더 이름 또는 특정 파일 이름을 가진 경로를 지정할 수 있습니다.

참조하세요

- .NET 8의 새로운 기능

.NET 8의 호환성이 손상되는 변경

앱을 .NET 8로 마이그레이션하는 경우 여기에 나열된 호환성이 손상되는 변경이 영향을 줄 수 있습니다. 변경 내용은 ASP.NET Core 또는 Windows Forms와 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경을 *이진 파일 비호환*, *원본 비호환* 또는 *동작 변경*으로 분류합니다.

- **이진 파일 비호환** - 새 런타임이나 구성 요소에 대해 실행할 때 기존 이진 파일의 동작이 크게 변경될 수 있습니다(예: 로드 또는 실행 실패). 그런 경우 다시 컴파일이 필요합니다.
- **원본 비호환** - 새 SDK 또는 구성 요소를 사용하여 다시 컴파일하거나 새 런타임을 대상으로 하는 경우 기존 소스 코드를 성공적으로 컴파일하려면 원본을 변경해야 할 수도 있습니다.
- **동작 변경** - 기존 코드 및 이진 파일은 런타임에 다르게 동작할 수 있습니다. 새로운 동작이 바람직하지 않은 경우 기존 코드를 업데이트하고 다시 컴파일해야 합니다.

ASP.NET Core

ASP.NET Core 8의 주요 변경 내용을 참조하세요.

컨테이너

테이블 확장

타이틀	변경 형식
Alpine 이미지에서 제거된 'ca-certificates' 패키지	이진 파일 비호환
Debian 12로 업그레이드된 Debian 컨테이너 이미지	이진 파일 비호환/동작 변경
기본 ASP.NET Core 포트가 8080으로 변경됨	동작 변경
Alpine 및 Debian 이미지에서 제거된 Kerberos 패키지	이진 파일 비호환
Alpine 이미지에서 'libintl' 패키지가 제거됨	동작 변경
다중 플랫폼 컨테이너 태그는 Linux 전용임	동작 변경
Linux 이미지의 새로운 '앱' 사용자	동작 변경

핵심 .NET 라이브러리

테이블 확장

타이틀	변경 형식
null인 경우 활동 작업 이름	동작 변경
AnonymousPipeServerStream.Dispose 동작	동작 변경
사용자 지정 진단 ID를 사용한 API 폐기	원본 비호환
Unix 파일 경로의 백슬래시 매핑	동작 변경
Base64.DecodeFromUtf8 메서드는 공백을 무시함	동작 변경
부울 지원 열거형 형식 지원이 제거됨	동작 변경
Complex.ToString 형식이 <a; b>(으)로 변경되었습니다	동작 변경
드라이브의 현재 디렉터리 경로 열거형	동작 변경
Enumerable.Sum이 일부 입력에 대해 새로운 OverflowException을 throw함	동작 변경
FileStream은 파이프가 닫힐 때 씀	동작 변경
FindSystemTimeZoneById가 새 개체를 반환하지 않음	동작 변경
GC.GetGeneration이 Int32.MaxValue를 반환할 수 있음	동작 변경
Unix에서의 GetFolderPath 동작	동작 변경
GetSystemVersion이 더 이상 ImageRuntimeVersion을 반환하지 않음	동작 변경
ITypeDescriptorContext null 허용 주석	원본 비호환
.NET Standard/.NET Framework에서 LDAP API를 사용할 수 없음	이진 파일 비호환
레거시 Console.ReadKey가 제거됨	동작 변경
메서드 작성기는 HasDefaultValue를 false로 설정한 매개 변수를 생성함	동작 변경
이제 System.IO.Packaging에서 패키지 파트 URI를 대/소문자를 구분하지 않고 비교합니다.	동작 변경
UseShellExecute가 false인 경우 ProcessStartInfo.WindowStyle이 적용됨	동작 변경
RuntimeIdentifier는 런타임이 빌드된 플랫폼을 반환함	동작 변경
Type.GetType은 모든 유효하지 않은 요소 형식에 대해 예외를 throw함	동작 변경

암호화

테이블 확장

타이틀	변경 형식	도입
macOS의 AesGcm 인증 태그 크기	동작 변경	미리 보기 1
RSA.EncryptValue 및 RSA.DecryptValue는 더 이상 사용되지 않음	원본 비호환	미리 보기 1

배포

테이블 확장

타이틀	변경 형식
호스트가 RID 관련 자산을 결정함	이진 파일 비호환/동작 변경
.NET 모니터에는 distroless 이미지만 포함됨	동작 변경
openSUSE 및 SLES용 .NET 패키지는 OpenSSL 3.x에 따라 달라집니다.	동작 변경
StripSymbols의 기본값은 true임	동작 변경

Entity Framework Core (엔티티 프레임워크 코어)

EF Core 8의 호환성에 영향을 미치는 변경을 참조하세요.

확장

테이블 확장

타이틀	변경 형식
ActivatorUtilities.CreateInstance는 일관되게 동작합니다.	동작 변경
ActivatorUtilities.CreateInstance에 null이 아닌 공급자 필요	동작 변경
일치하지 않는 값으로 인해 ConfigurationBinder을 throw함	동작 변경
ConfigurationManager 패키지는 더 이상 System.Security.Permissions를 참조하지 않음	원본 비호환
DirectoryServices 패키지가 더 이상 System.Security.Permissions를 참조하지 않음	원본 비호환
구성 바인더에 의해 사전에 빈 키가 추가됨	동작 변경
FromKeyedServicesAttribute.Key는 null일 수 있습니다.	원본 비호환

타이틀	변경 형식
HostApplicationBuilder ctor에서 적용하는 HostApplicationBuilderSettings.Args	동작 변경
ManagementDateTimeConverter.ToDateTime이 현지 시간을 반환함	동작 변경
System.Formats.Cbor DateTimeOffset 서식 변경	동작 변경

전역화

테이블 확장

타이틀	변경 형식
날짜 및 시간 변환기가 문화권 인수 적용	동작 변경
TwoDigitYearMax 기본값은 2049임	동작 변경

Interop

테이블 확장

타이틀	변경 형식
CreateObjectFlags.Unwrap은 대상 인스턴스에서만 래핑 해제됨	동작 변경
사용자 지정 마샬러에 추가 멤버 필요	원본 비호환
IDispatchImplAttribute API가 제거되었습니다.	이진 파일 비호환
JSFunctionBinding 암시적 공용 기본 생성자가 제거됨	이진 파일 비호환
SafeHandle 형식에는 공용 생성자가 있어야 함	원본 비호환
Linux 네이티브 라이브러리 해상도는 더 이상 사용하지 않습니다. netcoredeps	동작 변경

네트워킹

테이블 확장

타이틀	변경 형식
SendFile이 연결 없는 소켓에 대해 NotSupportedException을 throw함	동작 변경
mailto: URI의 사용자 정보를 비교합니다	동작 변경

반영

테이블 확장

타이틀	변경 형식
함수 포인터 형식에 더 이상 IntPtr이 사용되지 않음	동작 변경

SDK (소프트웨어 개발 키트)

테이블 확장

타이틀	변경 형식
CLI 콘솔 출력에서는 UTF-8을 사용함	동작 변경/원본 및 이진 파일 비호환
완료 후 콘솔 인코딩이 UTF-8이 아님	동작 변경/이진 파일 비호환
컨테이너는 기본적으로 '최신' 태그를 사용함	동작 변경
'dotnet pack'은 릴리스 구성을 사용합니다.	동작 변경/원본 비호환
'dotnet publish'는 릴리스 구성을 사용합니다.	동작 변경/원본 비호환
-getItem, -getProperty 및 -getTargetResult에 대한 중복 출력	동작 변경
System.Net.Http에 대한 암시적 using이 더 이상 추가되지 않음	동작 변경/원본 비호환
MSBuild 사용자 지정 파생 빌드 이벤트가 더 이상 사용되지 않음	동작 변경
MSBuild는 DOTNET_CLI_UI_LANGUAGE를 준수함	동작 변경
자체 포함되지 않은 런타임 관련 앱	원본/이진 파일 비호환
--arch 옵션이 독립형을 의미하지 않음	동작 변경
'dotnet restore'는 보안 취약성 경고를 생성함	동작 변경
원본이 취약성 데이터를 제공하지 않는 경우 'dotnet list package'가 경고합니다.	동작 변경
SDK는 더 작은 RID 그래프를 사용함	동작 변경/원본 비호환
DebugSymbols를 false로 설정하면 PDB 생성이 사용하지 않도록 설정됨	동작 변경
.NET SDK에 포함된 소스 링크	원본 비호환
.NET Standard 또는 .NET Framework에서는 트리밍을 사용할 수 없음	동작 변경

타이틀	변경 형식
.NET 도구에 대해 목록에 없는 패키지는 기본적으로 설치되지 않음	동작 변경
외부 빌드에서 가져온 .user 파일	동작 변경
.NET 8 SDK의 버전 요구 사항	원본 비호환

직렬화

테이블 확장

타이틀	변경 형식
대부분의 프로젝트에서 BinaryFormatter가 사용하지 않도록 설정됨	동작 변경
PublishedTrimmed 프로젝트가 반사 기반 직렬화에 실패함	동작 변경
반사 기반 역직렬 변환기는 메타데이터를 적극적으로 확인함	동작 변경

윈도우 폼즈 (Windows Forms)

테이블 확장

타이틀	변경 형식
PictureBox에서 원격 이미지를 로드하기 전에 인증서 확인	동작 변경
DateTimePicker.Text가 빈 문자열임	동작 변경
일부 특성에서 DefaultValueAttribute가 제거됨	동작 변경
ExceptionCollection ctor이 ArgumentException을 throw함	동작 변경
AutoScaleMode에 따라 양식 크기 조정	동작 변경
ImageList.ColorDepth 기본값은 Depth32Bit임	동작 변경
System.Windows.Extensions는 System.Drawing.Common을 참조하지 않음	원본 비호환
TableLayoutStyleCollection이 ArgumentException을 throw함	동작 변경
최상위 양식이 최소 및 최대 크기를 DPI로 조정함	동작 변경
WFDEV002 폐기는 이제 오류임	원본 비호환

참고 항목

- C# 12/.NET 8의 C# 컴파일러 호환성이 손상되는 변경
- .NET 8의 새로운 기능

Last updated on 2026. 01. 29.

.NET 7의 새로운 기능

아티클 • 2025. 01. 08.

.NET 7은 .NET 6 후속작으로, 통합되고, 현대적이고, 단순하며, 빠른중 하나입니다. .NET 7은 STS(표준 기간 지원) 릴리스(이전의 현재 릴리스라고 함)로 18개월 동안 지원됩니다.

이 문서에서는 .NET 7의 새로운 기능을 나열하고 각각에 대한 자세한 정보에 대한 링크를 제공합니다.

성능

성능은 .NET 7의 핵심 초점이며 모든 기능은 성능을 염두에 두고 설계되었습니다. 또한 .NET 7에는 순전히 성능을 목표로 하는 다음과 같은 향상된 기능이 포함되어 있습니다.

- OSR(온-스택 대체)은 계층화된 컴파일을 보완합니다. 런타임은 실행 중간에 현재 실행 중인 메서드에 의해 실행되는 코드를 변경할 수 있습니다(즉, "스택에 있는"동안). 장기 실행 메서드는 실행 중 최적화된 버전으로 전환할 수 있습니다.
- PGO(프로필 기반 최적화)는 이제 OSR에서 작동하며 프로젝트 파일에 `<TieredPGO>true</TieredPGO>` 추가하여 더 쉽게 사용하도록 설정할 수 있습니다. PGO는 대리자와 같은 추가 항목을 계측하고 최적화할 수도 있습니다.
- Arm64에 대한 코드 생성이 향상되었습니다.
- Native AOT 외부 종속성이 없는 대상 플랫폼의 파일 형식으로 독립 실행형 실행 파일을 생성합니다. IL이나 JIT 없이 시스템에 맞게 완전히 최적화되어 빠르게 시작할 수 있고 외부 의존성이 없는 소형 배포를 제공합니다. .NET 7에서 Native AOT는 콘솔 앱에 초점을 맞추고, 앱을 최적화해야 합니다.
- Blazor WebAssembly, Android 및 iOS 앱을 지원하는 Mono 런타임의 성능이 향상되었습니다.

.NET 7을 매우 빠르게 만드는 많은 성능 중심 기능에 대한 자세한 내용은 .NET 7 블로그 게시물의 성능 향상을 참조하세요.

System.Text.Json 직렬화

.NET 7에는 다음 영역에서 System.Text.Json 직렬화가 개선되었습니다.

- **계약 사용자 지정** 통해 형식이 직렬화 및 역직렬화되는 방식을 보다 세부적으로 제어할 수 있습니다. 자세한 내용은 JSON 계약사용자 지정을 참조하세요.
- **사용자 정의 형식 계층 구조에 대한 다형성 직렬화**. 자세한 내용은 파생 클래스의 속성 직렬화을 참조하세요.
- 역직렬화가 성공하려면 JSON 페이로드에 반드시 있어야 하는 속성인 필수 멤버을 에서 지원합니다. 자세한 내용은 필수 속성참조하세요.

이러한 업데이트 및 기타 업데이트에 대한 자세한 내용은 .NET 7의 System.Text.Json의 새로운 기능 블로그 게시물을 참조하십시오.

제네릭 수학

.NET 7 및 C# 11에는 수학 연산을 일반적으로 수행할 수 있는 혁신이 포함되어 있습니다. 즉, 작업 중인 정확한 형식을 알 필요가 없습니다. 예를 들어 두 개의 숫자를 추가하는 메서드를 작성하려는 경우 이전에는 각 형식에 대한 메서드의 오버로드를 추가해야 했습니다. 이제 형식 매개 변수가 숫자와 유사한 형식으로 제한되는 단일 제네릭 메서드를 작성할 수 있습니다. 자세한 내용은 제네릭 수학 문서 및 제네릭 수학 블로그 게시물을 참조하세요.

정규식

. NET의 정규식 라이브러리는 .NET 7에서 상당한 기능 및 성능 향상을 보였습니다.

- 새 옵션 RegexOptions.NonBacktracking은 역추적을 피하고 입력 길이에 비례하는 선형 시간 처리를 보장하는 방법으로 일치를 가능하게 합니다. 역추적되지 않는 엔진은 오른쪽에서 왼쪽으로 검색하는 데 사용할 수 없으며 몇 가지 다른 제한 사항이 있지만 모든 정규식 및 입력에 대해 빠릅니다. 자세한 내용은 비백트래킹 모드참조하세요.

- 정규식 원본 생성기는 새로운 기능입니다. 원본 생성기는 컴파일 시점에 *및* 패턴에 최적화된 엔진을 빌드하여 처리량 성능의 이점을 제공합니다. 내보낸 원본은 프로젝트의 일부이므로 보고 디버그할 수 있습니다. 또한 새 원본 생성기 진단 `SYSLIB1045`는 Regex을 사용하는 위치가 원본 생성기로 변환될 수 있음을 경고합니다. 자세한 내용은 .NET 정규식 원본 생성기참조하세요.

- 대/소문자를 구분하지 않는 검색의 경우 .NET 7에는 큰 성능 향상이 포함됩니다. RegexOptions.IgnoreCase이 지정되면, 패턴의 각 문자와 입력의 각 문자에서 더 이상 ToLower을 호출하지 않기 때문에 성능이 향상됩니다. 대신 Regex 생성될 때 모든 대/소문자 관련 작업이 수행됩니다.

- 이제 Regex 일부 API에 대한 범위를 지원합니다. 이 지원의 일부로 다음과 같은 새 메서드가 추가되었습니다.
  - Regex.EnumerateMatches
  - Regex.Count
  - Regex.IsMatch(ReadOnlySpan<Char>)(및 몇 가지 다른 오버로드)

이러한 기능 및 기타 개선 사항에 대한 자세한 내용은 .NET 7 블로그 게시물의 정규식 향상을 참조하세요.

.NET 라이브러리

.NET 라이브러리 API가 많이 개선되었습니다. 일부는 이 문서의 다른 전용 섹션에서 설명합니다. 일부 다른 항목은 다음 표에 요약되어 있습니다.

테이블 확장

묘사	API들	추가 정보
TimeSpan, TimeOnly, DateTime및 DateTimeOffset 형식의 마이크로초 및 나노초 지원	- DateTime.Microsecond - DateTime.Nanosecond - DateTime.AddMicroseconds(Double) - 새 DateTime 생성자 오버로드 - DateTimeOffset.Microsecond - DateTimeOffset.Nanosecond - DateTimeOffset.AddMicroseconds(Double) - 새 DateTimeOffset 생성자 오버로드 - TimeOnly.Microsecond - TimeOnly.Nanosecond - TimeSpan.Microseconds - TimeSpan.Nanoseconds - TimeSpan.FromMicroseconds(Double) - 그리고 다른 사람 ...	이러한 API는 더 이상 "틱" 값에 대한 계산을 수행하여 마이크로초 및 나노초 값을 결정할 필요가 없음을 의미합니다. 자세한 내용은 .NET 7 Preview 4 블로그 게시물을 참조하세요.
Tar 보관 파일 읽기, 쓰기, 보관 및 추출을 위한 API	System.Formats.Tar	자세한 내용은 .NET 7 Preview 4 및 .NET 7 Preview 6 블로그 게시물을 참조하세요.
트래픽을 안전한 수준으로 유지하여 리소스를 보호하기 위한 속도 제한 API	RateLimiter 및 System.Threading.RateLimiting NuGet 패키지의 다른	자세한 내용은 .NET의 HTTP 처리기에 대한 속도 제한 및 .NET의 속도 제한 발표 를 참조하세요.
모든 데이터를 Stream 읽는 API	- Stream.ReadExactly - Stream.ReadAtLeast	Stream.Read 스트림에서 사용할 수 있는 것보다 적은 데이터를 반환할 수 있습니다. 새 메서드는 요청된 바이트 수를 정확히 읽고, 새 메서드는 요청된 바이트 수를 적어도 읽습니다. 자세한 내용은 .NET 7 Preview 5 블로그 게시물을 참조하세요.
`DateOnly`, `TimeOnly`, `Int128`, `UInt128`및 `Half` 대한 새 형식 변환기	System.ComponentModel 네임스페이스에서 다음을 수행합니다. - DateOnlyConverter - TimeOnlyConverter - Int128Converter	형식 변환기는 값 형식을 문자열로 변환하는 데 자주 사용됩니다. 이러한 새 API는 최근에 추가된 형식에 대한 형식 변환기를 추가합니다.

묘사	API들	추가 정보
	- UInt128Converter - HalfConverter	
IMemoryCache 대한 메트릭 지원	- MemoryCacheStatistics - MemoryCache.GetCurrentStatistics()	GetCurrentStatistics() 이벤트 카운터 또는 메트릭 API를 사용하여 하나 이상의 메모리 캐시에 대한 통계를 추적할 수 있습니다. 자세한 내용은 .NET 7 Preview 4 블로그 게시물을 참조하세요.
Unix 파일 권한을 가져와서 설정하는 API	- System.IO.UnixFileMode 열거형 - File.GetUnixFileMode - File.SetUnixFileMode - FileSystemInfo.UnixFileMode - Directory.CreateDirectory(String, UnixFileMode) - FileStreamOptions.UnixCreateMode	자세한 내용은 .NET 7 Preview 7 블로그 게시물을 참조하세요.
문자열에 필요한 구문 종류를 나타내는 특성	StringSyntaxAttribute	예를 들어, 매개 변수에 `[StringSyntax(StringSyntaxAttribute.Regex)]` 특성을 부여하면 `string` 매개 변수가 정규식을 기대하도록 지정할 수 있습니다.
브라우저 또는 다른 WebAssembly 아키텍처에서 실행할 때 JavaScript와 상호 운용하는 API	System.Runtime.InteropServices.JavaScript	JavaScript 앱은 .NET 7에서 확장된 WebAssembly 지원을 사용하여 JavaScript에서 .NET 라이브러리를 다시 사용할 수 있습니다. 자세한 내용은 .NET 7에서 JavaScript 앱에서 .NET을 사용하는 방법 을 참조하세요. .

관찰 가능성

.NET 7은 *모니터링 기능*개선을 진행합니다. 관찰성은 앱의 크기가 조정되고 기술 복잡성이 증가함에 따라 앱의 상태를 이해하는 데 도움이 됩니다. . NET의 관찰 가능성 구현은 주로 OpenTelemetry 중심으로 구축됩니다. 향상된 기능은 다음과 같습니다.

- 관리되는 스레드의 범위 컨텍스트가 변경되는 시기를 감지하는 데 사용할 수 있는 새 Activity.CurrentChanged 이벤트입니다.
- EnumerateTagObjects(), EnumerateLinks()및 EnumerateEvents()등 Activity 속성에 대한 새로운 성능 열거자 메서드입니다.

자세한 내용은 .NET 7 Preview 4 블로그 게시물을 참조하세요.

.NET SDK

.NET 7 SDK CLI 템플릿 환경을 개선합니다. 또한 컨테이너에 게시하고 NuGet을 사용하여 중앙 패키지 관리를 수행할 수 있습니다.

템플릿

`dotnet new` 명령 및 템플릿 작성에 대한 몇 가지 환영 개선 사항이 적용되었습니다.

dotnet new

템플릿을 기반으로 새 프로젝트, 구성 파일 또는 솔루션을 만드는 dotnet new CLI 명령은 이제 탐색을 위한 탭 완성 지원합니다.

- 사용 가능한 템플릿 이름
- 템플릿 옵션
- 허용되는 옵션 값

또한 더 나은 적합성을 위해 `install`, `uninstall`, `search`, `list` 및 `update` 하위 명령의 `--` 접두사는 더 이상 없습니다.

저작

새로운 .NET 7의 개념인 템플릿 *제약 조건*은 템플릿이 허용되는 컨텍스트를 정의할 수 있게 해줍니다. 제약 조건은 템플릿 엔진이 `dotnet new list`같은 명령에 표시해야 하는 템플릿을 결정하는 데 도움이 됩니다. 템플릿을 운영 체제, 템플릿 엔진 호스트(예: Visual Studio의 .NET CLI 또는 새 프로젝트 대화 상자) 및 설치된 워크로드로 제한할 수 있습니다. 템플릿의 구성 파일에서 제약 조건을 정의합니다.

또한 템플릿 구성 파일에서 이제 템플릿 매개 변수에 여러 값을 허용하는 것으로 주석을 달 수 있습니다. 예를 들어 web 템플릿 여러 형태의 인증을 허용합니다.

자세한 내용은 .NET 7 Preview 6 블로그 게시물을 참조하세요.

컨테이너에 게시

컨테이너는 클라우드에서 다양한 애플리케이션 및 서비스를 배포하고 실행하는 가장 쉬운 방법 중 하나입니다. 컨테이너 이미지는 이제 .NET SDK의 지원되는 출력 형식이며, dotnet publish사용하여 컨테이너화된 버전의 애플리케이션을 만들 수 있습니다. 이 기능에 대한 자세한 내용은 .NET SDK의 기본 제공 컨테이너 지원 발표 를 참조하세요. 자습서를 참조하세요: .NET 앱을 컨테이너화하는 방법을 알고 싶다면, dotnet publish을 사용하세요 .

중앙 패키지 관리

이제 NuGet의 CPM(중앙 패키지 관리) 기능을 사용하여 한 위치에서 프로젝트의 일반적인 종속성을 관리할 수 있습니다. 사용하도록 설정하려면 *Directory.Packages.props* 파일을 리포지토리의 루트에 추가합니다. 이 파일에서 MSBuild 속성 `ManagePackageVersionsCentrally` `true` 설정하고 `PackageVersion` 항목을 사용하여 일반적인 패키지 종속성에 대한 버전을 추가합니다. 그런 다음 개별 프로젝트 파일에서 중앙에서 관리되는 패키지를 참조하는 PackageReference 항목에서 `Version` 특성을 생략할 수 있습니다.

자세한 내용은 Central 패키지 관리참조하세요.

P/Invoke 원본 생성

.NET 7에서는 C#에서 플랫폼 호출(P/Invokes)에 대한 원본 생성기를 소개합니다. 소스 생성기는 마샬링 코드의 컴파일 시간 소스 생성을 트리거하는 `static` `partial` 메서드에서 LibraryImportAttribute 찾습니다. 컴파일 시간에 마샬링 코드를 생성하면 DllImportAttribute 사용할 때와 마찬가지로 런타임에 IL 스텁을 생성할 필요가 없습니다. 원본 생성기는 애플리케이션 성능을 향상시키고 앱을 AOT(미리) 컴파일할 수도 있습니다. 자세한 내용은 플랫폼 호출에 대한 원본 생성을 참조하시고, 원본 생성 P/Invokes에서 사용자 지정 마샬러 사용을 참조하십시오.

관련 릴리스

이 섹션에는 .NET 7 릴리스와 일치하는 릴리스가 있는 관련 제품에 대한 정보가 포함되어 있습니다.

Visual Studio 2022 버전 17.4

자세한 내용은 Visual Studio 2022의 새로운 기능을 참조하세요.

C# 11

C# 11에는 제네릭 수학, 원시 문자열 리터럴, 파일 범위 형식 및 기타 새로운 기능에 대한 지원이 포함됩니다. 자세한 내용은 C# 11의 새로운 기능을 참조하세요.

F# 7

F# 7은 언어를 더 간단하게 만들고 새로운 C# 기능과의 성능 및 상호 운용성을 향상시키기 위한 여정을 계속합니다. 자세한 내용은 F# 7 발표 공지를 참조하세요.

.NET MAUI

.NET 다중 플랫폼 앱 UI(.NET MAUI)는 C# 및 XAML을 사용하여 네이티브 모바일 및 데스크톱 앱을 만들기 위한 플랫폼 간 프레임워크입니다. Android, iOS, macOS 및 Windows API를 단일 API로 통합합니다. 최신 업데이트에 대한 정보를 보려면 .NET MAUI의 새로운 기능 .NET 7을 참조하세요.

ASP.NET Core

ASP.NET Core 7.0에는 속도 제한 미들웨어, 최소 API 개선, gRPC JSON 코드 변환이 포함됩니다. 모든 업데이트에 대한 정보는 ASP.NET Core 7의 새로운 기능을 참조하십시오.

EF Core

Entity Framework Core 7.0에는 JSON 열에 대한 공급자 중립적 지원, 변경 내용 저장을 위한 향상된 성능 및 사용자 지정 리버스 엔지니어링 템플릿이 포함됩니다. 모든 업데이트에 대한 자세한 내용은 EF Core 7.0의 새로운 기능을 참조하십시오.

Windows Forms

.NET 7을 위한 Windows Forms에 많은 작업이 들어갔습니다. 다음 영역에서 개선이 이루어졌습니다.

- 접근성
- 높은 DPI 및 크기 조정
- 데이터 바인딩

자세한 내용은 .NET 7의 Windows Forms의 새로운 기능을 참조하세요.

WPF

.NET 7의 WPF에는 성능 및 접근성 향상뿐만 아니라 다양한 버그 수정이 포함되어 있습니다. 자세한 내용은 .NET 7 블로그 게시물에서 WPF의 새로운 기능 참조하세요.

Orleans

Orleans 강력하고 확장 가능한 분산 애플리케이션을 빌드하기 위한 플랫폼 간 프레임워크입니다. 대한 최신 업데이트에 대한 자세한 내용은 3.x에서 7.0마이그레이션을 참조하세요.

.NET 업그레이드 도우미 및 CoreWCF

.NET 업그레이드 도우미는 이제 서버 쪽 WCF 앱을 WCF에서 .NET(Core)으로 커뮤니티에서 만든 포트인 CoreWCF업그레이드하도록 지원합니다. 자세한 내용은 CoreWCF사용하도록 WCF 서버 쪽 프로젝트 업그레이드 참조하세요.

ML.NET

이제 ML.NET 최신 딥 러닝 기술을 사용하여 사용자 지정 텍스트 분류 모델을 쉽게 학습시킬 수 있는 텍스트 분류 API가 포함되어 있습니다. 자세한 내용은 AutoML 및 도구의 새로운 기능과 ML.NET 텍스트 분류 API를 소개하는 및 블로그 게시물을 참조하세요.

참조

- .NET 7 릴리스 정보

.NET 7의 호환성이 손상되는 변경

앱을 .NET 7로 마이그레이션하는 경우 여기에 나열된 호환성이 손상되는 변경이 영향을 줄 수 있습니다. 변경 내용은 ASP.NET Core 또는 Windows Forms와 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경을 *이진 파일 비호환*, *원본 비호환* 또는 *동작 변경*으로 분류합니다.

- **이진 파일 비호환** - 새 런타임이나 구성 요소에 대해 실행할 때 기존 이진 파일의 동작이 크게 변경될 수 있습니다(예: 로드 또는 실행 실패). 그런 경우 다시 컴파일이 필요합니다.
- **원본 비호환** - 새 SDK 또는 구성 요소를 사용하여 다시 컴파일하거나 새 런타임을 대상으로 하는 경우 기존 소스 코드를 성공적으로 컴파일하려면 원본을 변경해야 할 수도 있습니다.
- **동작 변경** - 기존 코드 및 이진 파일은 런타임에 다르게 동작할 수 있습니다. 새 동작이 바람직하지 않은 경우 기존 코드를 업데이트하고 다시 컴파일해야 합니다.

ASP.NET Core

ASP.NET Core 7의 주요 변경 내용을 참조하세요.

핵심 .NET 라이브러리

테이블 확장

타이틀	변경 유형
기본 진단 ID를 사용하는 API 사용되지 않음	원본이 호환되지 않음
기본이 아닌 진단 ID를 사용하는 API 사용되지 않음	원본이 호환되지 않음
별표는 어셈블리 이름 특성 대해 더 이상 허용되지 않습니다.	동작 변경
BinaryFormatter serialization API에서 컴파일러 오류를 생성	원본이 호환되지 않음
BrotliStream은 정의되지 않은 CompressionLevel 값을 더 이상 허용하지 않음	바이너리 비호환
Visual Studio의 C++/CLI 프로젝트	원본이 호환되지 않음
리플렉션 호출 API 예외에 대한 변경 내용	바이너리 비호환
수집 불가능한 AssemblyLoadContext의 수집 가능한 어셈블리	바이너리 비호환
DateTime 추가 메서드 정밀도 변경	동작 변경

타이틀	변경 유형
NaN에 대한 메서드 동작 변경과 동일	바이너리 비호환
EventSource 콜백 동작	동작 변경
PatternContext<T>에 대한 제네릭 형식 제약 조건	이진/소스 호환되지 않음
레거시 FileStream 전략이 제거됨	바이너리 비호환
이전 프레임워크에 대한 라이브러리 지원	이진/소스 호환되지 않음
숫자 형식 문자열의 최대 전체 자릿수	바이너리 비호환
범위가 수정된 정규식 패턴	동작 변경
SerializationFormat.Binary는 사용되지 않습니다	이진/소스 호환되지 않음
System. Drawing.Common 구성 스위치가 제거됨	동작 변경
System.Runtime.CompilerServices.Unsafe NuGet 패키지	동작 변경
기호 링크의 시간 필드	바이너리 비호환
연결된 캐시 항목 추적	바이너리 비호환
BrotliStream에 대한 CompressionLevel 유효성 검사	바이너리 비호환

구성

테이블 확장

타이틀	변경 유형
app.config의 System.diagnostics 항목	바이너리 비호환

암호화

테이블 확장

타이틀	변경 유형
EnvelopedCms 암호 해독은 래핑을 두 번 해제하지 않음	바이너리 비호환
동적 X509ChainPolicy 검증 시간	바이너리 비호환
식별 이름의 X500DistinguishedName 구문 분석	바이너리 비호환

배포

테이블 확장

타이틀	변경 유형
기본적으로 모든 어셈블리가 잘림	원본이 호환되지 않음
다단계 조회가 사용하지 않도록 설정됨	바이너리 비호환
64비트 Windows의 x86 호스트 경로	동작 변경
TrimmerDefaultAction 더 이상 사용되지 않음	원본이 호환되지 않음

Entity Framework Core (엔티티 프레임워크 코어)

EF Core 7의 주요 변경 사항을 참조하세요.

확장

테이블 확장

타이틀	변경 유형
구성을 사전에 바인딩하면 값이 확장됨	동작 변경
Windows Shell에서 시작되는 앱에 대한 ContentRootPath	바이너리 비호환
환경 변수 접두사	바이너리 비호환

전역화

테이블 확장

타이틀	변경 유형
Windows Server에서 세계화 API가 ICU 라이브러리 사용	바이너리 비호환

Interop

테이블 확장

타이틀	변경 유형
에뮬레이션의 동안의 RuntimeInformation.OSArchitecture	바이너리 비호환

.NET 마우이

테이블 확장

타이틀	변경 유형
생성자가 구체적 형식 대신 기본 인터페이스 허용	바이너리 비호환
흐름 방향 도우미 메서드가 제거됨	이진/소스 호환되지 않음
새 UpdateBackground 매개 변수	바이너리 비호환
ScrollToRequest 속성 이름이 변경됨	이진/소스 호환되지 않음
일부 Windows API가 제거됨	이진/소스 호환되지 않음

네트워킹

테이블 확장

타이틀	변경 유형
AllowRenegotiation 기본값은 false임	이진/소스 호환되지 않음
Linux의 사용자 지정 ping 페이로드	바이너리 비호환
Socket.End 메서드가 ObjectDisposedException을 throw하지 않음	바이너리 비호환

SDK 및 MSBuild

테이블 확장

타이틀	변경 유형
게시 전용 자동 RuntimeIdentifier	이진/소스 호환되지 않음
CLI 콘솔 출력에서는 UTF-8을 사용함	이진/소스 호환되지 않음
완료 후 콘솔 인코딩이 UTF-8이 아님	바이너리 비호환
.NET 7에서사용자 지정 형식의 MSBuild serialization	이진/소스 호환되지 않음

타이틀	변경 유형
병렬 SDK 설치	이진/소스 호환되지 않음
루트 폴더의 도구 매니페스트	동작 변경
.NET 7 SDK의 버전 요구 사항	동작 변경
dotnet test: -a를 --arch 대신 별칭 --test-adapter-path로 전환	이진/소스 호환되지 않음
dotnet test: -r를 --runtime 대신 별칭 --results-dir로 전환	이진/소스 호환되지 않음
--output 옵션이 더 이상 솔루션 수준 명령에 유효하지 않음	이진/소스 호환되지 않음
SDK는 더 이상 ResolvePackageDependencies를 호출하지 않음	원본이 호환되지 않음

직렬화

테이블 확장

타이틀	변경 유형
DataContractSerializer는 -0을 역직렬화할 때 기호를 유지	바이너리 비호환
선행 또는 후행 공백을 가진 버전 형식 역직렬화	바이너리 비호환
JsonSerializerOptions 복사 생성자에 JsonSerializerContext가 포함됨	바이너리 비호환
개체 형식에 대한 다형 serialization	바이너리 비호환
System.Text.Json 원본 생성기 대체	바이너리 비호환

윈도우 폼즈 (Windows Forms)

테이블 확장

타이틀	변경 유형
폐기 및 경고	원본이 호환되지 않음
일부 API가 ArgumentNullException을 throw함	바이너리 비호환

WPF (Windows Presentation Foundation, 윈도우 프레젠테이션 파운데이션)

테이블 확장

타이틀	변경 유형
텍스트 편집기에서 복원된 끌어서 놓기 작업 동작	동작 변경

XML 및 XSLT

테이블 확장

타이틀	변경 유형
XmlSecureResolver는 사용되지 않음	이진/소스 호환되지 않음

참고 항목

- C# 11/.NET 7의 C# 컴파일러 호환성이 손상되는 변경
- .NET 7의 새로운 기능

Last updated on 2026. 01. 29.

.NET 6의 새로운 기능

아티클 • 2024. 12. 18.

.NET 6은 .NET 5시작된 .NET 통합 계획의 마지막 부분을 제공합니다. .NET 6은 모바일, 데스크톱, IoT 및 클라우드 앱에서 SDK, 기본 라이브러리 및 런타임을 통합합니다. 이 통합 외에도 .NET 6 에코시스템은 다음을 제공합니다.

- **간소화된 개발**: 시작하기가 쉽습니다. C# 10의 새로운 언어 기능은 작성해야 하는 코드의 양을 줄일 있습니다. 또한 웹 스택 및 최소 API를 투자하면 더 작고 빠른 마이크로 서비스를 쉽게 작성할 수 있습니다.
- **성능 향상**: .NET 6은 클라우드에서 실행하는 경우 컴퓨팅 비용을 낮추는 가장 빠른 전체 스택 웹 프레임워크입니다.
- **Ultimate 생산성**: .NET 6 및 Visual Studio 2022는 핫 다시 로드, 새로운 git 도구, 지능형 코드 편집, 강력한 진단 및 테스트 도구 및 더 나은 팀 공동 작업을 제공할 있습니다.

.NET 6은 LTS(장기 지원) 릴리스로 3년 동안 지원됩니다.

미리 보기 기능은 기본적으로 사용하지 않도록 설정됩니다. 또한 프로덕션에서 사용할 수 없으며 이후 버전에서 제거될 수 있습니다. 새 RequiresPreviewFeaturesAttribute 미리 보기 API에 주석을 추가하는 데 사용되며, 이러한 미리 보기 API를 사용하는 경우 해당 분석기가 경고합니다.

.NET 6은 Visual Studio 2022 이상 버전에서 지원됩니다.

이 문서에서는 .NET 6의 새로운 기능을 모두 다루지 않습니다. 새로운 기능을 모두 보려면, 또한 이 문서에 나열된 기능에 대한 더 많은 정보를 얻으려면 .NET 6 블로그 게시물을 참조하세요.

성능

.NET 6에는 다양한 성능 향상이 포함되어 있습니다. 이 섹션에서는 몇 가지 개선 사항을 나열합니다— FileStream, 프로필 기반 최적화, 및 AOT 컴파일. 자세한 내용은 .NET 6 블로그 게시물의 성능 향상을 참조하세요.

파일 스트림 (FileStream)

Windows에서 더 나은 성능과 안정성을 제공하기 위해 .NET 6에 대해 System.IO.FileStream 형식을 다시 작성했습니다. 이제 FileStream Windows에서 비동기

I/O를 만들 때 차단되지 않습니다. 자세한 내용은 .NET 6 블로그 게시물의 파일 IO 개선 사항을 참조하세요.

프로필 기반 최적화

PGO(프로필 기반 최적화)는 JIT 컴파일러가 가장 자주 사용되는 형식 및 코드 경로 측면에서 최적화된 코드를 생성하는 위치입니다. .NET 6에는 *동적* PGO가 도입되었습니다. 동적 PGO는 계층형 컴파일과 함께 작동하여 계층 0 단계에서 배치되는 추가 계측에 따라 코드를 더욱 최적화합니다. 동적 PGO는 기본적으로 사용하지 않도록 설정되지만 `DOTNET_TieredPGO` 환경 변수사용하여 사용하도록 설정할 수 있습니다. 자세한 내용은 JIT 성능 향상 참조하세요.

Crossgen2

.NET 6에는 크로스겐의 후속작인 Crossgen2가 도입되어 제거되었습니다. Crossgen 및 Crossgen2는 앱의 시작 시간을 개선하기 위해 AOT(미리 실행) 컴파일을 제공하는 도구입니다. Crossgen2는 C++가 아닌 C#으로 작성되었으며 이전 버전에서는 불가능했던 분석 및 최적화를 수행할 수 있습니다. 자세한 내용은 Crossgen2대한 대화를 참조하세요.

Arm64 지원

.NET 6 릴리스에는 네이티브 Arm64 실행 및 x64 에뮬레이션 모두에 대한 macOS Arm64(또는 "Apple Silicon") 및 Windows Arm64 운영 체제에 대한 지원이 포함되어 있습니다. 또한 x64 및 Arm64 .NET 설치 관리자는 이제 나란히 설치합니다. 자세한 내용은 macOS 11 및 Windows 11에 대한 .NET 지원 (Arm64 및 x64)을 참조하십시오.

즉시 다시 로드

*핫 리로드*는 앱의 소스 코드를 수정하고 그 변경 사항을 실행 중인 앱에 즉시 적용할 수 있는 기능입니다. 이 기능의 목적은 편집 사이에 앱이 다시 시작되지 않도록 하여 생산성을 높이는 것입니다. 핫 다시 로드는 Visual Studio 2022 및 `dotnet watch` 명령줄 도구에서 사용할 수 있습니다. 핫 다시 로드는 대부분의 .NET 앱 유형과 C#, Visual Basic 및 C++ 소스 코드에서 작동합니다. 자세한 내용은 핫 리로드 블로그 게시물 확인하세요.

.NET MAUI

.NET 다중 플랫폼 앱 UI(.NET MAUI)는 2022년 1분기 릴리스 후보와 2022년 2분기 GA(일반 공급)로 *미리 보기*있습니다. .NET MAUI를 사용하면 단일 코드베이스를 사용하여 데스

크톱 및 모바일 운영 체제용 네이티브 클라이언트 앱을 빌드할 수 있습니다. 자세한 내용은 .NET 다중 플랫폼 앱 UI 블로그 게시물의 업데이트를 참조하세요.

C# 10 및 템플릿

C# 10에는 `global using` 지시문, 파일 범위 네임스페이스 선언 및 레코드 구조체와 같은 혁신이 포함되어 있습니다. 자세한 내용은 C# 언어 버전 기록참조하세요.

이 작업과 함께 C#용 .NET SDK 프로젝트 템플릿은 새로운 언어 기능 중 일부를 사용하도록 현대화되었습니다.

- `async Main` 메서드
- 최상위 수준의 문장들
- 대상 형식의 새 식
- 암시적 global using 지시문
- 파일 범위의 네임스페이스
- Nullable 참조 형식

이러한 새 언어 기능을 프로젝트 템플릿에 추가하면 새 코드가 활성화된 기능으로 시작됩니다. 그러나 .NET 6으로 업그레이드할 때 기존 코드는 영향을 받지 않습니다. 이러한 템플릿 변경에 대한 자세한 내용은 .NET SDK: C# 프로젝트 템플릿 현대화 블로그 게시물을 참조하세요.

F# 및 Visual Basic

F# 6은 F# 언어 및 F# Interactive에 몇 가지 개선 사항을 추가합니다. 자세한 내용은 F# 6의 새로운 기능을 참조하세요.

Visual Basic에는 Visual Studio 환경 및 Windows Forms 프로젝트 시작이 개선되었습니다.

SDK 워크로드

.NET SDK의 크기를 더 작게 유지하기 위해 일부 구성 요소는새로운 선택적 SDK 워크로드에 배치되었습니다. 이러한 구성 요소에는 .NET MAUI 및 Blazor WebAssembly AOT가 포함됩니다. Visual Studio를 사용하는 경우 필요한 모든 SDK 워크로드 설치를 처리합니다. .NET CLI사용하는 경우 새 `dotnet workload` 명령을 사용하여 워크로드를 관리할 수 있습니다.

명령	묘사
dotnet 워크로드 검색	사용 가능한 워크로드를 검색합니다.
dotnet 워크로드 설치	지정된 워크로드를 설치합니다.
dotnet 워크로드 제거	지정된 워크로드를 제거합니다.
dotnet 워크로드 업데이트	설치된 워크로드를 업데이트합니다.
dotnet 워크로드 복구	설치된 모든 워크로드를 다시 설치하여 끊어진 설치를 복구합니다.
dotnet 워크로드 목록	설치된 워크로드를 나열합니다.

자세한 내용은 선택적 SDK 워크로드참조하세요.

System.Text.Json API들

.NET 6의 System.Text.Json에서 많은 개선이 이루어져, 이제는 "산업용" 직렬화 기술로 자리 잡았습니다.

원본 생성기

.NET 6은 System.Text.Json에 대한 새로운 원본 생성기을 추가합니다. 소스 생성은 JsonSerializer와 함께 작동하며 여러 가지 방법으로 구성할 수 있습니다. 성능을 향상시키고 메모리 사용량을 줄이며 어셈블리 트리밍을 용이하게 할 수 있습니다. 자세한 내용은 System.Text.Json 리플렉션 또는 원본 생성을 선택하는 방법 및 System.Text.Json원본 생성을 사용하는 방법을 참조하세요.

쓰기 가능한 DOM

기존 읽기 전용 DOM을 보완하는 새 DOM(문서 개체 모델)이 추가되었습니다. 새 API는 일반적인 CLR 개체(POCO) 형식을 사용할 수 없는 경우에 대한 경량화된 serialization 대안을 제공합니다. 또한 큰 JSON 트리의 하위 섹션으로 효율적으로 이동하여 배열을 읽거나 해당 하위 섹션에서 POCO를 역직렬화할 수 있습니다. 쓰기 가능한 DOM을 지원하기 위해 다음과 같은 새 형식이 추가되었습니다.

- JsonNode
- JsonArray
- JsonObject
- JsonValue

자세한 내용은 JSON DOM 옵션을 참조하십시오.

IAsyncEnumerable 직렬화

이제 System.Text.JsonIAsyncEnumerable<T> 인스턴스를 사용한 직렬화 및 역직렬화를 지원합니다. 비동기 직렬화 메서드는 개체 그래프의 모든 IAsyncEnumerable<T> 인스턴스를 나열한 다음 JSON 배열로 직렬화합니다. 역직렬화를 위해 새 메서드 JsonSerializer.DeserializeAsyncEnumerable<TValue>(Stream, JsonSerializerOptions, CancellationToken) 추가되었습니다. 자세한 내용은 IAsyncEnumerable serialization참조하세요.

기타 새 API

유효성 검사와 기본값 설정을 위한 새로운 serialization 인터페이스:

- IJsonOnDeserialized
- IJsonOnDeserializing
- IJsonOnSerialized
- IJsonOnSerializing

자세한 내용은 콜백참조하세요.

새 속성 순서 지정 특성:

- JsonPropertyOrderAttribute

  자세한 내용은 직렬화된 속성순서 구성을 참조하세요.

"원시" JSON을 작성하는 새 메서드:

- Utf8JsonWriter.WriteRawValue

  자세한 내용은 원시 JSON 쓰기을 참조하세요.

스트림에 대한 동기 직렬화 및 역직렬화:

- JsonSerializer.Deserialize(Stream, Type, JsonSerializerOptions)
- JsonSerializer.Deserialize(Stream, Type, JsonSerializerContext)
- JsonSerializer.Deserialize<TValue>(Stream, JsonSerializerOptions)
- JsonSerializer.Deserialize<TValue>(Stream, JsonTypeInfo<TValue>)
- JsonSerializer.Serialize(Stream, Object, Type, JsonSerializerOptions)
- JsonSerializer.Serialize(Stream, Object, Type, JsonSerializerContext)
- JsonSerializer.Serialize<TValue>(Stream, TValue, JsonSerializerOptions)
- JsonSerializer.Serialize<TValue>(Stream, TValue, JsonTypeInfo<TValue>)

serialization 중에 참조 주기가 검색될 때 개체를 무시하는 새 옵션:

- ReferenceHandler.IgnoreCycles

  자세한 내용은 순환 참조 무시을 참고하십시오.

.NET에서 JSON 직렬화 및 역직렬화 을 참조하여 의 직렬화 및 역직렬화에 대한 자세한 정보를 확인하세요.

HTTP/3

.NET 6에는 새 버전의 HTTP인 HTTP/3에 대한 미리 보기 지원이 포함되어 있습니다. HTTP/3은 QUIC라는 새로운 기본 연결 프로토콜을 사용하여 기존 기능 및 성능 문제를 해결합니다. QUIC는 연결을 더 빠르게 설정하고, 연결은 IP 주소와 독립적이므로 모바일 클라이언트가 Wi-Fi와 셀룰러 네트워크 간에 로밍할 수 있습니다. 자세한 내용은 에서 HttpClient로 HTTP/3을 사용하는 방법을 참조하세요.을 확인해 보세요.

ASP.NET Core

ASP.NET Core에는 최소 API, Blazor WebAssembly 앱용 AOT(Ahead-Of-Time) 컴파일 및 단일 페이지 앱의 향상된 기능이 포함되어 있습니다. 또한 이제 Blazor 구성 요소를 JavaScript에서 렌더링하고 기존 JavaScript 기반 앱과 통합할 수 있습니다. 자세한 내용은 ASP.NET Core 6의 새로운 기능을 참조하세요.

OpenTelemetry

.NET 6은 소프트웨어의 성능 및 동작을 분석하는 데 도움이 되는 도구, API 및 SDK 컬렉션인 OpenTelemetry 대한 향상된 지원을 제공합니다. System.Diagnostics.Metrics 네임스페이스의 API는 OpenTelemetry 메트릭 API 사양 를 구현합니다. 예를 들어 다양한 메트릭 시나리오를 지원하는 네 가지 계측 클래스가 있습니다. 계측 클래스는 다음과 같습니다.

- Counter<T>
- Histogram<T>
- ObservableCounter<T>
- ObservableGauge<T>

안전

.NET 6은 CET(제어 흐름 적용 기술) 및 "쓰기 전용 실행"(W^X)의 두 가지 주요 보안 완화에 대한 미리 보기 지원을 추가합니다.

CET는 일부 최신 Intel 및 AMD 프로세서에서 사용할 수 있는 인텔 기술입니다. 일부 제어 흐름 하이재킹 공격으로부터 보호하는 기능을 하드웨어에 추가합니다. .NET 6은 Windows x64 앱용 CET를 지원하며 명시적으로 사용하도록 설정해야 합니다. 자세한 내용은 Intel CET 섀도 스택과의 호환성에 대한 .NET 6을 참조하세요.

W^X는 .NET 6이 있는 모든 운영 체제에서 사용할 수 있지만 Apple Silicon에서 기본적으로만 사용하도록 설정됩니다. W^X는 메모리 페이지를 쓰기 가능하고 실행 가능하도록 동시에 허용하지 않음으로써 가장 간단한 공격 경로를 차단합니다.

IL 트리밍

독립형 배포의 트리밍이 향상되었습니다. .NET 5에서는 사용되지 않는 어셈블리만 잘립니다. .NET 6은 사용되지 않는 형식 및 멤버의 트리밍도 추가합니다. 또한 런타임에 사용되는 코드를 제거할 수 있는 경우를 경고하는 트리밍 경고는 이제 기본적으로 설정되어 있습니다. 자세한 내용은 자체 포함 배포 및 실행 파일 트리밍을 참조하세요.

코드 분석

.NET 6 SDK에는 API 호환성, 플랫폼 호환성, 트리밍 안전성, 문자열 연결 및 분할에서 범위 사용, 더 빠른 문자열 API 및 더 빠른 컬렉션 API와 관련된 몇 가지 새로운 코드 분석기가 포함되어 있습니다. 새(및 제거된) 분석기의 전체 목록은 Analyzer 릴리스 - .NET 6 참조하세요.

사용자 지정 플랫폼 가드

플랫폼 호환성 분석기OperatingSystem 클래스의 `Is<Platform>` 메서드(예: OperatingSystem.IsWindows())를 플랫폼 가드로 인식합니다. 사용자 지정 플랫폼 가드를 허용하기 위해 .NET 6에는 지원되거나 지원되지 않는 플랫폼 이름으로 필드, 속성 또는 메서드에 주석을 추가하는 데 사용할 수 있는 두 가지 새로운 특성이 도입되었습니다.

- SupportedOSPlatformGuardAttribute
- UnsupportedOSPlatformGuardAttribute

Windows Forms

Application.SetDefaultFont(Font) 애플리케이션 전체에서 기본 글꼴을 설정하는 .NET 6의 새로운 메서드입니다.

C# Windows Forms 앱용 템플릿은 `global using` 지시문, 파일 범위 네임스페이스 및 nullable 참조 형식을 지원하도록 업데이트되었습니다. 또한 상용구 코드를 줄이고

Windows Forms 디자이너가 기본 설정 글꼴로 디자인 화면을 렌더링할 수 있도록 하는 애플리케이션 부트스트랩 코드를 포함합니다. 부트스트랩 코드는 Application.EnableVisualStyles()같은 다른 구성 메서드에 대한 호출을 내보내는 소스 생성 메서드인 `ApplicationConfiguration.Initialize()`대한 호출입니다. 또한 ApplicationDefaultFont MSBuild 속성을 통해 기본이 아닌 글꼴을 설정하는 경우 `ApplicationConfiguration.Initialize()`가 SetDefaultFont(Font)호출을 생성한다.

자세한 내용은 Windows Forms의 새로운 기능 블로그 게시물을 참조하세요.

원본 빌드

.NET SDK의 모든 소스를 포함한 *소스 tarball*은 이제 .NET SDK 빌드의 결과물입니다. Red Hat과 같은 다른 조직에서는 이 원본 tarball을 사용하여 자체 버전의 SDK를 빌드할 수 있습니다.

대상 프레임워크 이름

.NET 6에 추가 OS 관련 TFM(대상 프레임워크 모니커)이 추가되었습니다(예: `net6.0-android`, `net6.0-ios` 및 `net6.0-macos`). 자세한 내용은 .NET 5+ OS별 TFM참조하세요.

제네릭 수학

*미리 보기*는 .NET 6에서 제네릭 형식에 연산자를 사용할 수 있는 기능입니다. .NET 6에는 C# 10의 새로운 미리 보기 기능인 `static abstract` 인터페이스 멤버를 활용하는 다양한 인터페이스가 도입되었습니다. 이러한 인터페이스는 다른 연산자에 해당합니다. 예를 들어 `IAdditionOperators` `+` 연산자를 나타냅니다. 인터페이스는 System.Runtime.Experimental NuGet 패키지에서 사용할 수 있습니다. 자세한 내용은 일반 수학 블로그 게시물을 참조하세요.

NuGet 패키지 유효성 검사

NuGet 라이브러리 개발자인 경우 새로운 패키지 유효성 검사 도구 사용하면 패키지가 일관되고 올바른 형식인지 확인할 수 있습니다. 다음을 확인하여 여부를 결정할 수 있습니다.

- 패키지 버전 간에 호환성을 깨뜨리는 변경 사항이 있습니다.
- 패키지에는 모든 런타임별 구현에 대해 동일한 공용 API 집합이 있습니다.
- 대상 프레임워크나 런타임 적용 가능성에 어떠한 간격이 있습니까?

자세한 내용은 패키지 유효성 검사 블로그 게시물을 참조하세요.

리플렉션 API들

.NET 6에는 코드를 검사하고 Null 허용 여부 정보를 제공하는 다음과 같은 새로운 API가 도입되었습니다.

- System.Reflection.NullabilityInfo
- System.Reflection.NullabilityInfoContext
- System.Reflection.NullabilityState

이러한 API는 리플렉션 기반 도구 및 serializer에 유용합니다.

Microsoft.Extensions API들

다음 표와 같이 여러 확장 네임스페이스가 .NET 6에서 개선되었습니다.

테이블 확장

Namespace	개선
Microsoft.Extensions.DependencyInjection	CreateAsyncScope은 IAsyncDisposable 서비스를 등록하는 서비스 공급자에 대해 `using` 문을 안전하게 사용할 수 있도록 합니다.
Microsoft.Extensions.Hosting	새로운 ConfigureHostOptions 메서드는 애플리케이션 설정을 간소화합니다.
Microsoft.Extensions.Logging	Microsoft.Extensions.Logging에는 성능이 우수한 로깅 API를 위한 새로운 소스 생성기가 있습니다. `partial` 로깅 메서드에 새 LoggerMessageAttribute를 추가하면 소스 생성기가 트리거됩니다. 컴파일 시 생성기는 기존 로깅 솔루션보다 런타임에 일반적으로 더 빠른 `partial` 메서드의 구현을 생성합니다. 자세한 내용은 컴파일 타임 로깅 소스 생성을 참조하세요.

새로운 LINQ API들

.NET 6에는 수많은 LINQ 메서드가 추가되었습니다. 다음 표에 나열된 대부분의 새 메서드에는 System.Linq.Queryable 형식에 동일한 메서드가 있습니다.

테이블 확장

메서드	묘사
Enumerable.TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	열거형을 강제 적용하지 않고 시퀀스의 요소 수를 확인하려고 시도합니다.
Enumerable.Chunk<TSource>(IEnumerable<TSource>, Int32)	시퀀스의 요소를 지정된 크기의 청크로 분할합니다.
Enumerable.MaxBy 및 Enumerable.MinBy	키 선택기를 사용하여 최대 또는 최소 요소를 찾습니다.
Enumerable.DistinctBy, Enumerable.ExceptBy, Enumerable.IntersectBy및 Enumerable.UnionBy	집합 기반 작업을 수행하는 이러한 새로운 메서드 변형을 사용하면 키 선택기 함수를 사용하여 같음을 지정할 수 있습니다.
Enumerable.ElementAt<TSource>(IEnumerable<TSource>, Index) 및 Enumerable.ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	시퀀스의 시작 또는 끝에서 계산되는 인덱스를 허용합니다(예: `Enumerable.Range(1, 10).ElementAt(^2)` `9`반환합니다.
Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, TSource) 및 Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource) Enumerable.LastOrDefault<TSource>(IEnumerable<TSource>, TSource) 및 Enumerable.LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource) Enumerable.SingleOrDefault<TSource>(IEnumerable<TSource>, TSource) 및 Enumerable.SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	새 오버로드를 사용하면 시퀀스가 비어 있는 경우 사용할 기본값을 지정할 수 있습니다.
Enumerable.Max<TSource>(IEnumerable<TSource>, IComparer<TSource>) 및 Enumerable.Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	새 오버로드를 사용하면 비교자를 지정할 수 있습니다.
Enumerable.Take<TSource>(IEnumerable<TSource>, Range)	Range 인수를 수락하여 시퀀스 조각을 간단하게 만들 수 있습니다. 예를 들어 `source.Take(7).Skip(2)`대신 `source.Take(2..7)` 사용할 수 있습니다.
Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	*세 개의 지정된* 시퀀스에서 요소를 사용하여 튜플 시퀀스를 생성합니다.

날짜, 시간 및 표준 시간대 개선 사항

.NET 6에는 System.DateOnly 및 System.TimeOnly두 구조체가 추가되었습니다. 각각 DateTime의 날짜 부분과 시간 부분을 나타냅니다. DateOnly 생일 및 기념일에 유용하며, TimeOnly 매일 경보 및 주간 업무 시간에 유용합니다.

이제 표준 시간대 데이터가 설치된 운영 체제에서 IANA(Internet Assigned Numbers Authority) 또는 Windows 표준 시간대 ID를 사용할 수 있습니다. 시스템에서 요청된 표준 시간대를 찾을 수 없는 경우 windows 표준 시간대에서 IANA 표준 시간대로 입력을 자동으로 변환하도록 TimeZoneInfo.FindSystemTimeZoneById(String) 메서드가 업데이트되었습니다. 또한 한 표준 시간대 형식에서 다른 표준 시간대 형식으로 수동으로 변환해야 하는 경우 시나리오에 대해 새 메서드 TryConvertIanaIdToWindowsId(String, String) 및 TryConvertWindowsIdToIanaId 추가되었습니다.

몇 가지 다른 표준 시간대 개선 사항도 있습니다. 자세한 내용은 .NET 6날짜, 시간 및 표준 시간대 개선 사항을 참조하세요.

PriorityQueue 클래스

새 PriorityQueue<TElement,TPriority> 클래스는 값과 우선 순위가 모두 있는 항목의 컬렉션을 나타냅니다. 항목은 우선 순위가 증가하는 순서로 큐에서 삭제됩니다. 즉, 우선 순위 값이 가장 낮은 항목이 먼저 큐에서 삭제됩니다. 이 클래스는 최소 힙 데이터 구조를 구현합니다.

참고하세요

- F# 6 새로운 기능
- EF Core 6의 새로운 기능
- ASP.NET Core 6의 새로운 기능은 무엇인가?
- .NET 6 릴리스 정보
- Visual Studio 2022 릴리스 정보
- 블로그: .NET 6 발표
- 블로그: 새 System.Text.Json 원본 생성기 사용해 보세요.

.NET 6의 호환성이 손상되는 변경

앱을 .NET 6으로 마이그레이션하는 경우 여기에 나열된 호환성이 손상되는 변경이 영향을 줄 수 있습니다. 변경 내용은 ASP.NET Core 또는 Windows Forms와 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경이 *이진 파일 호환*인지 또는 *원본 호환*인지 여부를 나타냅니다.

- **이진 호환** - 기존 이진 파일이 다시 컴파일하지 않고 성공적으로 로드되고 실행되며 런타임 동작은 변경되지 않습니다.
- **원본 호환** - 새 런타임을 대상으로 하거나 새 SDK 또는 구성 요소를 사용할 때 소스 코드는 변경 없이 성공적으로 컴파일됩니다.

ASP.NET Core

ASP.NET Core 6의 주요 변경 내용을 참조하세요.

컨테이너

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
컨테이너 이미지의 기본 콘솔 로거 서식 지정	✔	✘

.NET 6 컨테이너의 기타 호환성이 손상되는 변경에 대한 자세한 내용은 .NET 6 컨테이너 릴리스 정보를 참조하세요.

핵심 .NET 라이브러리

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
기본이 아닌 진단 ID를 사용하는 API 사용되지 않음	✔	✘
Nullable 참조 형식 주석의 변경 내용	✔	✘
디버그 메서드의 조건부 문자열 평가	✔	✘
Windows의 Environment.ProcessorCount 동작	✔	✘

타이틀	이진 호환 가능	소스 호환 가능
EventSource 콜백 동작	✔	✔
Unix의 File.Replace는 Windows와 일치하도록 예외를 throw함	✔	✘
파일 스트림이 Unix에서 공유 잠금을 사용하여 파일을 잠금	✘	✔
FileStream이 더 이상 파일 오프셋을 OS와 동기화하지 않음	✘	✘
ReadAsync 또는 WriteAsync 완료 후 FileStream.Position 업데이트	✘	✘
사용 중지 API에 대한 새 진단 ID	✔	✘
새 System.Linq.Queryable 메서드 오버로드	✔	✘
패키지에서 삭제된 이전 프레임워크 버전	✘	✔
매개 변수 이름이 변경됨	✔	✘
스트림 파생 형식의 매개 변수 이름	✔	✘
DeflateStream, GZipStream 및 CryptoStream의 부분 및 0바이트 읽기	✔	✘
Windows 읽기 전용 파일에 타임스탬프 설정	✘	✔
표준 숫자 형식 구문 분석 정밀도	✔	✘
인터페이스의 정적 추상 멤버	✘	✔
StringBuilder.Append 오버로드 및 평가 순서	✘	✔
강력한 이름 API가 PlatformNotSupportedException을 throw함	✘	✔
System.Drawing.Common, Windows에서만 지원	✘	✘
System.Security.SecurityContext가 사용되지 않는 것으로 표시됨	✔	✘
Task.FromResult가 싱글톤을 반환할 수 있음	✘	✔
BackgroundService에서 처리되지 않은 예외	✔	✘

암호화

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
CreateEncryptor 메서드가 잘못된 피드백 크기에 대해 예외를 throw함	✘	✔

배포

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
64비트 Windows의 x86 호스트 경로	✔	✔

Entity Framework Core (엔티티 프레임워크 코어)

EF Core 6의 주요 변경 사항을 참조하세요.

확장

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
AddProvider가 null이 아닌 공급자를 확인함	✔	✘
FileConfigurationProvider.Load가 InvalidDataException을 throw함	✔	✘
반복되는 XML 요소에 인덱스가 포함됨	✘	✔
삭제한 ServiceProvider 분석을 통해 예외를 throw함	✔	✘

전역화

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
세계화 고정 모드의 문화권 만들기 및 대/소문자 매핑		

Interop

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
인터페이스의 정적 추상 멤버	✘	✔

JIT 컴파일러

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
ECMA-335에 따라 호출 인수 강제 변환	✔	✔

네트워킹

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
Kerberos 및 Negotiate를 위해 SPN에서 포트가 제거됨	✘	✔
WebRequest, WebClient, ServicePoint는 사용되지 않음	✔	✘

SDK (소프트웨어 개발 키트)

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
-p에 대한 dotnet run 옵션은 사용되지 않음	✔	✘
이전 버전에서 지원되지 않는 템플릿의 C# 코드	✔	✔
EditorConfig 파일이 암시적으로 포함됨	✔	✘
macOS용 apphost 생성	✔	✘
게시 출력의 중복 파일에 대한 오류 생성	✘	✔
GetTargetFrameworkProperties 및 GetNearestTargetFramework가 ProjectReference 프로토콜에서 제거됨	✘	✔
Arm64에 에뮬레이트된 x64의 설치 위치	✔	✘
MSBuild, GetType() 호출 지원 중지		
.NET을 사용자 지정 위치에 설치할 수 없음	✔	✔
OutputType이 WinExe로 자동으로 설정되지 않음	✔	✘

타이틀	이진 호환 가능	소스 호환 가능
--no-restore를 사용하여 ReadyToRun을 게시하려면 변경이 필요함	✔	✘
runtimeconfig.dev.json 파일이 생성되지 않음	✘	✔
자체 포함이 지정되지 않은 경우 RuntimeIdentifier 경고	✔	✘
루트 폴더의 도구 매니페스트	✔	✔
.NET 6 SDK의 버전 요구 사항	✔	✔
.version 파일에 빌드 버전이 포함되어 있음	✔	✔
IntermediateOutputPath에 참조 어셈블리 쓰기	✘	✔

직렬화

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
DataContractSerializer는 -0을 역직렬화할 때 기호를 유지	✘	✔
TimeSpan에 대한 기본 serialization 형식	✘	✔
IAsyncEnumerable 직렬화	✔	✘
JSON 소스 생성 API 리팩터링	✘	✔
컬렉션 속성의 JsonNumberHandlingAttribute	✘	✔
새 JsonSerializer 소스 생성기 오버로드	✘	✔

윈도우 폼즈 (Windows Forms)

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
C# 템플릿이 애플리케이션 부트스트랩을 사용함	✔	✘
선택한 TableLayoutSettings 속성이 InvalidEnumArgumentException을 throw함	✘	✔

타이틀	이진 호환 가능	소스 호환 가능
DataGridView 관련 API가 이제 InvalidOperationException을 throw함	✗	✓
ListViewGroupCollection 메서드가 새 InvalidOperationException을 throw함	✗	✓
NotifyIcon.Text 최대 텍스트 길이가 늘어남	✗	✓
필요할 때만 호출되는 ScaleControl	✓	✗
일부 API가 ArgumentNullException을 throw함	✗	✓
노드가 다른 곳에 할당될 경우 TreeNodeCollection.Item이 예외를 throw함	✗	✓

XML 및 XSLT

테이블 확장

타이틀	이진 호환 가능	소스 호환 가능
잘못된 인덱스에 대한 XNodeReader.GetAttribute 동작	✓	✗

참고 항목

- C# 9 이후의 C# 컴파일러 호환성이 손상되는 변경
- C# 10의 C# 컴파일러 호환성이 손상되는 변경
- .NET 6의 새로운 기능

Last updated on 2026. 01. 29.

.NET 5의 새로운 기능

아티클 • 2025. 01. 29.

.NET 5는 3.1에 이어 .NET Core의 다음 주요 릴리스입니다. 이 릴리스는 다음 두 가지 이유로 .NET Core 4 대신 .NET 5로 명명되었습니다.

- .NET Framework 4.x와 혼동하지 않도록 버전 번호 4.x를 건너뛰었다.
- "Core"는 앞으로 .NET의 주요 구현임을 강조하기 위해 이름에서 삭제되었습니다. .NET 5는 .NET Core 또는 .NET Framework보다 더 많은 유형의 앱과 더 많은 플랫폼을 지원합니다.

ASP.NET Core 5.0은 .NET 5를 기반으로 하지만 이름이 "Core"인 것을 유지하여 ASP.NET MVC 5와 혼동하지 않도록 합니다. 마찬가지로 Entity Framework Core 5.0은 Entity Framework 5 및 6과 혼동하지 않도록 "Core"라는 이름을 유지합니다.

.NET 5에는 .NET Core 3.1에 비해 다음과 같은 향상된 기능과 새로운 기능이 포함되어 있습니다.

- C# 업데이트
- F# 업데이트
- Visual Basic 업데이트
- System.Text.Json의 새로운 기능
- 단일 파일 앱
- 앱 최적화
- Windows Arm64 및 Arm64 내장 함수
- 덤프 디버깅에 대한 도구 지원
- 런타임 라이브러리는 널 허용 참조 형식을 위한 80개의% 주석이
- 성능 향상:
  - 가비지 수집(GC)
  - System.Text.Json
  - System.Text.RegularExpressions
  - 비동기 ValueTask 풀링
  - 컨테이너 크기 최적화
  - 더 많은 영역

.NET 5는 .NET Framework를 대체하지 않습니다.

.NET 5 이상 버전은 앞으로 .NET의 기본 구현이지만 .NET Framework 4.x는 여전히 지원됩니다. .NET Framework에서 .NET 5로 다음 기술을 이식할 계획은 없지만 .NET에는 대안이 있습니다.

테이블 확장

기술	권장되는 대안
웹 양식	ASP.NET Core Blazor 또는 Razor Pages
Windows 워크플로(WF)	Elsa-Workflows

Windows Communication Foundation

WCF(Windows Communication Foundation) 원래 구현은 Windows에서만 지원되었습니다. 그러나 .NET Foundation에서 사용할 수 있는 클라이언트 포트가 있습니다. 완전히 오픈 소스 이며, 여러 플랫폼을 지원하며 Microsoft가 지원합니다. 핵심 NuGet 패키지는 다음과 같습니다.

- system.ServiceModel.Duplex
- System.ServiceModel.Federation
- System.ServiceModel.Http
- System.ServiceModel.NetTcp
- System.ServiceModel.Primitives
- System.ServiceModel.Security

앞에서 언급한 클라이언트 라이브러리를 보완하는 서버 구성 요소는 CoreWCF 통해 사용할 수 있습니다. 2022년 4월부터 CoreWCF는 Microsoft에서 공식적으로 지원됩니다. 그러나 WCF의 대안으로 gRPC를 고려해 보세요.

.NET 5는 .NET Standard를 대체하지 않습니다.

새 애플리케이션 개발에서는 클래스 라이브러리를 비롯한 모든 프로젝트 형식에 대해 `net5.0` TFM(Target Framework Moniker)을 지정할 수 있습니다. .NET 5 워크로드 간의 코드 공유가 간소화되었습니다. `net5.0` TFM만 있으면 됩니다.

.NET 5 앱 및 라이브러리의 경우 `net5.0` TFM은 `netcoreapp` 및 `netstandard` TFM을 결합하고 대체합니다. 그러나 .NET Framework, .NET Core 및 .NET 5 워크로드 간에 코드를 공유하려는 경우 `netstandard2.0` TFM으로 지정하면 됩니다. 자세한 내용은 .NET Standard 참조하세요.

C# 업데이트

.NET 5 앱을 작성하는 개발자는 최신 C# 버전 및 기능에 액세스할 수 있습니다. .NET 5는 C# 9와 페어링되어 언어에 많은 새로운 기능을 제공합니다. 다음은 몇 가지 주요 사항입니다.

- **레코드**: 새로운 `with` 식이 지원하는 값 기반 동일성 의미 체계 및 비파괴적 변형을 가진 참조 형식입니다.
- **관계형 패턴 일치**: 논리 패턴을 포함하여 비교 평가 및 식에 대한 관계형 연산자(새 키워드 `and`, `or` 및 `not`)로 패턴 일치 기능을 확장합니다.
- **최상위 문**: C#의 채택 및 학습을 가속화하기 위한 수단으로 `Main` 메서드를 생략할 수 있으며 다음 예제와 같이 간단한 애플리케이션이 유효합니다.

  C#

  ```
  System.Console.Write("Hello world!");
  ```

- **함수 포인터**: 중간 언어(IL) 명령어 코드를 노출하는 언어 구문: `ldftn` 및 `calli`.

사용 가능한 C# 9 기능에 대한 자세한 내용은 C# 9새로운 기능을 참조하세요.

원본 생성기

강조 표시된 새로운 C# 기능 중 일부 외에도 원본 생성기가 개발자 프로젝트에 진출하고 있습니다. 소스 생성기를 사용하면 컴파일 중에 실행되는 코드가 프로그램을 검사하고 나머지 코드와 함께 컴파일되는 추가 파일을 생성할 수 있습니다.

원본 생성기에 대한 자세한 내용은 C# 원본 생성기 소개 및 C# 원본 생성기 샘플 참조하세요.

F# 업데이트

F#은 .NET 기능 프로그래밍 언어이며, .NET 5를 사용하면 개발자가 F# 5에 액세스할 수 있습니다. 새로운 기능 중 하나는 보간된 문자열로, C#의 보간된 문자열과 유사할 뿐만 아니라 JavaScript에서도 유사한 기능이 있습니다.

F#

```
let name = "David"
let age = 36
let message = $"{name} is {age} years old."
```

기본 문자열 보간 외에도 형식화된 보간이 있습니다. 형식화된 보간을 사용하면 지정된 형식이 형식 지정자와 일치해야 합니다.

F#

```
let name = "David"
let age = 36
let message = $"%s{name} is %d{age} years old."
```

이 형식은 형식이 안전한 입력에 따라 문자열의 형식을 지정하는 sprintf 함수와 유사합니다.

자세한 내용은 F# 5의 새로운 기능을 참조하세요.

Visual Basic 업데이트

.NET 5에는 Visual Basic에 대한 새로운 언어 기능이 없습니다. 그러나 .NET 5를 사용하면 Visual Basic 지원이 다음으로 확장됩니다.

테이블 확장

묘사	`dotnet new` 매개 변수
콘솔 애플리케이션	`console`
클래스 라이브러리	`classlib`
WPF 애플리케이션	`wpf`
WPF 클래스 라이브러리	`wpflib`
WPF 사용자 지정 컨트롤 라이브러리	`wpfcustomcontrollib`
WPF 사용자 제어 라이브러리	`wpfusercontrollib`
Windows Forms(WinForms) 애플리케이션	`winforms`
Windows Forms(WinForms) 클래스 라이브러리	`winformslib`
단위 테스트 프로젝트	`mstest`
NUnit 3 테스트 프로젝트	`nunit`
NUnit 3 테스트 항목	`nunit-test`
xUnit 테스트 프로젝트	`xunit`

.NET CLI의 프로젝트 템플릿에 대한 자세한 내용은 dotnet new참조하세요.

System.Text.Json의 새로운 기능

및의 System.Text.Json에 새로운 기능이 있습니다.

- 참조를 유지하고 순환 참조를 처리
- HttpClient serialization 확장 메서드
- 따옴표 안에 숫자를 허용하거나 기재하시오.
- 변경할 수 없는 형식 및 C# 9 레코드 지원
- 비공용 속성 접근자 지원
- 지원 필드
- 조건부로 속성 무시
- 비 문자열 키 사전 지원
- 사용자 지정 변환기가 null 처리하도록 허용
- JsonSerializerOptions 복사
- 웹 기본값 사용하여 JsonSerializerOptions 만들기

참고 항목

- 하나의 .NET 대한 여정
- .NET 5에 있어서 성능 향상
- .NET SDK 다운로드

.NET 5의 주요 변경 내용

앱을 .NET 5로 마이그레이션하는 경우 여기에 나열된 주요 변경 내용이 영향을 줄 수 있습니다. 변경 내용은 ASP.NET Core 또는 암호화와 같은 기술 영역별로 그룹화됩니다.

이 문서에서는 각 호환성이 손상되는 변경 내용이 *이진 호환* 되는지 *또는 원본과 호환*되는지를 나타냅니다.

- **이진 호환** - 기존 이진 파일이 다시 컴파일하지 않고 성공적으로 로드되고 실행되며 런타임 동작은 변경되지 않습니다.
- **원본 호환** - 소스 코드는 새 런타임을 대상으로 하거나 새 SDK 또는 구성 요소를 사용할 때 변경 없이 성공적으로 컴파일됩니다.

ASP.NET Core

ASP.NET Core 5의 주요 변경 내용을 참조하세요.

코드 분석

테이블 확장

제목	이진 호환	원본 호환
CA1416 경고	✔	✘
CA1417 경고	✔	✘
CA1831 경고	✔	✘
CA2013 경고	✔	✘
CA2014 경고	✔	✘
CA2015 경고	✔	✘
CA2200 경고	✔	✘
CA2247 경고	✔	✘

핵심 .NET 라이브러리

테이블 확장

제목	이진 호환	원본 호환
단일 파일 게시에 대한 어셈블리 관련 API 변경 내용	✗	✓
BinaryFormatter serialization 메서드는 사용되지 않습니다.	✓	✗
코드 액세스 보안 API는 더 이상 사용되지 않습니다.	✓	✗
CreateCounterSetInstance가 InvalidOperationException을 throw합니다.	✓	✗
기본 ActivityIdFormat은 W3C입니다.	✗	✓
Environment.OSVersion은 올바른 버전을 반환합니다.	✗	✓
FrameworkDescription의 값은 .NET Core가 아닌 .NET입니다.	✓	✗
GAC API는 사용되지 않습니다.	✓	✗
하드웨어 내장 IsSupported 검사	✗	✓
IntPtr 및 UIntPtr에서 IFormattable 구현	✓	✗
LastIndexOf가 빈 검색 문자열을 처리합니다.	✗	✓
Unix에서 ASCII가 아닌 문자가 있는 URI 경로	✗	✓
기본이 아닌 진단 ID()을 사용한 API 사용 중단 ()	✓	✗
ConsoleLoggerOptions의 사용되지 않는 속성	✓	✗
LINQ OrderBy.First의 복잡성	✗	✓
OSPlatform 특성의 이름이 변경되거나 제거됨	✓	✗
Microsoft.DotNet.PlatformAbstractions 패키지가 제거됨	✗	✓
PrincipalPermissionAttribute가 사용되지 않음	✓	✗
미리 보기 버전에서 매개 변수 이름 변경	✓	✗
참조 어셈블리의 매개 변수 이름 변경	✓	✗
원격 API는 사용되지 않습니다.	✗	✓
Activity.Tags 목록의 순서가 반대로 바뀝니다.	✓	✗
SSE 및 SSE2 비교 메서드는 NaN을 처리합니다.	✓	✗
Thread.Abort가 더 이상 사용되지 않음	✓	✗
Unix에서 UNC 경로의 URI 인식	✗	✓
UTF-7 코드 경로는 더 이상 사용되지 않음	✓	✗

제목	이진 호환	원본 호환
Vector2.Lerp 및 Vector4.Lerp의 동작 변경	✔	✘
벡터<T> 가 NotSupportedException을 throw합니다.	✘	✔

암호화

테이블 확장

제목	이진 호환	원본 호환
브라우저에서 지원되지 않는 암호화 API	✘	✔
Cryptography.Oid는 init 전용입니다.	✔	✘
Linux의 기본 TLS 암호 그룹	✘	✔
암호화 추상화에 대한 Create() 오버로드는 사용되지 않습니다.	✔	✘
기본 FeedbackSize 값이 변경됨	✔	✘

Entity Framework Core (엔티티 프레임워크 코어)

EF Core 5.0의 주요 변경 사항을 참조하세요.

세계화

테이블 확장

제목	이진 호환	원본 호환
Windows에서 ICU 라이브러리 사용	✘	✔
StringInfo 및 TextElementEnumerator는 UAX29 규격입니다.	✘	✔
라틴 문자 1자로 변경된 유니코드 범주	✔	✘
TextInfo.ListSeparator 값이 변경됨	✔	✘

Interop

테이블 확장

제목	이진 호환	원본 호환
WinRT에 대한 지원이 제거됨	✗	✓
InterfaceIsIInspectable에 RCW를 캐스팅하는 경우 예외가 throw됩니다.	✗	✓
비 Windows 플랫폼에서 A/W 접미사 검색 없음	✗	✓

네트워킹

테이블 확장

제목	이진 호환	원본 호환
쿠키 경로 처리는 RFC 6265를 준수합니다.	✓	✗
SendToAsync를 호출한 후 LocalEndPoint가 업데이트됨	✓	✗
MulticastOption.Group이 null을 허용하지 않음	✓	✗
스트림은 연속 시작 작업을 허용합니다.	✗	✓
.NET 런타임에서 제거된 WinHttpHandler	✗	✓

SDK (소프트웨어 개발 키트)

테이블 확장

제목	이진 호환	원본 호환
기본적으로 가져온 Directory.Packages.props 파일	✗	✓
실행 파일이 일치하지 않는 실행 파일을 참조할 때 발생하는 오류		✓
FrameworkReference가 Windows SDK용 WindowsSdkPackageVersion으로 대체됨	✓	✗
NETCOREAPP3_1 전처리기 기호가 정의되지 않음	✓	✗
OutputType이 WinExe로 설정	✗	✓
PublishDepsFilePath 동작 변경	✗	✓
Netcoreapp에서 net으로 TargetFramework 변경	✗	✓
WinForms 및 WPF 앱은 Microsoft.NET.Sdk를 사용합니다.	✗	✓

안전

테이블 확장

제목	이진 호환	원본 호환
코드 액세스 보안 API는 더 이상 사용되지 않습니다.	✔	✘
PrincipalPermissionAttribute가 사용되지 않음	✔	✘
UTF-7 코드 경로는 더 이상 사용되지 않음	✔	✘

직렬화

테이블 확장

제목	이진 호환	원본 호환
BinaryFormatter.Deserialize 예외를 다시 포장함	✔	✘
JsonSerializer.Deserialize에는 단일 문자 문자열이 필요합니다.	✔	✘
ASP.NET Core 앱은 따옴표 붙은 숫자를 역직렬화합니다.	✔	✘
JsonSerializer.Serialize가 ArgumentNullException을 throw합니다.	✔	✘
역직렬화에 사용되지 않는 public이 아닌 매개 변수 없는 생성자	✔	✘
키-값 쌍을 직렬화할 때 옵션이 적용됩니다.	✔	✘

윈도우 폼즈 (Windows Forms)

테이블 확장

제목	이진 호환	원본 호환
네이티브 코드는 Windows Forms 개체에 액세스할 수 없습니다.	✔	✘
OutputType이 WinExe로 설정	✘	✔
DataGridView가 사용자 지정 글꼴을 다시 설정하지 않음	✔	✘
메서드 throw ArgumentException	✔	✘
메서드 throw ArgumentNullException	✔	✘

제목	이진 호환	원본 호환
속성이 ArgumentOutOfRangeException을 던집니다	✔	✘
TextFormatFlags.ModifyString이 사용되지 않음	✔	✘
DataGridView API가 InvalidOperationException을 throw합니다.	✔	✘
WinForms 앱은 Microsoft.NET.Sdk를 사용합니다.	✘	✔
상태 표시줄 컨트롤이 제거됨	✔	✘

WPF (Windows Presentation Foundation, 윈도우 프레젠테이션 파운데이션)

테이블 확장

제목	이진 호환	원본 호환
OutputType이 WinExe로 설정	✘	✔
WPF 앱은 Microsoft.NET.Sdk를 사용합니다.	✘	✔

참고하십시오

- C# 9/.NET 5의 C# 컴파일러 호환성을 깨뜨리는 변경 사항
- .NET 5의 새로운 기능

Last updated on 2026. 01. 29.

.NET Core 3.1의 새로운 기능

2025. 06. 17.

이 문서에서는 .NET Core 3.1의 새로운 기능을 설명합니다. 이 릴리스에는 작지만 중요한 수정 내용에 중점을 둔 .NET Core 3.0의 소소한 개선 사항이 포함되어 있습니다. .NET Core 3.1의 가장 중요한 기능은 LTS(장기 지원) 릴리스라는 것입니다.

Visual Studio 2019를 사용하는 경우 .NET Core 3.1 프로젝트와 함께 작동하려면 Visual Studio 2019 버전 16.4 이상으로 업데이트해야 합니다. Visual Studio 버전 16.4의 새로운 기능에 대한 내용은 Visual Studio 2019 버전 16.4의 새로운 기능을 참조하세요.

릴리스에 대한 자세한 내용은 .NET Core 3.1 공지를 참조하세요.

- Windows, macOS 또는 Linux에서 .NET Core 3.1을 다운로드하여 시작하세요.

장기 지원

.NET Core 3.1은 릴리스 후 3년 동안 Microsoft의 지원을 받는 LTS 릴리스입니다. 앱을 최신 LTS 릴리스로 이동하는 것이 좋습니다. 지원되는 릴리스 목록은 .NET 및 .NET Core 지원 정책 페이지를 참조하세요.

테이블 확장

해제	수명 주기 끝
.NET Core 3.1	2022년 12월 13일에 수명이 종료됩니다.
.NET Core 3.0	2020년 3월 3일에 수명이 종료됩니다.
.NET Core 2.2	2019년 12월 23일에 수명이 종료됩니다.
.NET Core 2.1	2021년 8월 21일에 수명이 종료됩니다.

자세한 내용은 .NET 및 .NET Core 지원 정책을 참조하세요.

macOS appHost 및 공증

'macOS만 해당'

공증된 macOS용 .NET Core SDK 3.1부터, appHost 설정이 기본적으로 사용하지 않도록 설정됩니다. 자세한 내용은 macOS Catalina 공증과 이것이 .NET Core 다운로드 및 프로젝트에 미치는 영향을 참조하세요.

appHost 설정이 사용하도록 설정된 경우, 빌드 또는 게시할 때 .NET Core가 네이티브 Mach-O 실행 파일을 생성합니다. 앱을 `dotnet run` 명령을 사용하여 소스 코드에서 실행하거나 Mach-O 실행 파일을 직접 시작하면 앱이 appHost 컨텍스트에서 실행됩니다.

사용자가 appHost 없이 프레임워크 종속 앱을 시작할 수 있는 유일한 방법은 `dotnet <filename.dll>` 명령을 사용하는 것입니다. appHost는 앱을 자체 포함 방식으로 게시하면 항상 만들어집니다.

appHost는 다음과 같이 프로젝트 수준에서 구성하거나 `dotnet` 매개 변수를 사용하여 특정 `-p:UseAppHost` 명령에 대해 켜거나 끌 수 있습니다.

- 프로젝트 파일

  XML

  ```xml
  <PropertyGroup>
    <UseAppHost>true</UseAppHost>
  </PropertyGroup>
  ```

- 명령줄 매개 변수

  .NET CLI

  ```
  dotnet run -p:UseAppHost=true
  ```

`UseAppHost` 설정에 대한 자세한 내용은 MSBuild properties for Microsoft.NET.Sdk(Microsoft.NET.Sdk의 MSBuild 속성)를 참조하세요.

윈도우 폼즈 (Windows Forms)

Windows만 해당

> **⚠ 경고**
>
> Windows Forms 관련 호환성이 손상되는 변경이 있습니다.

레거시 컨트롤은 Visual Studio Designer Toolbox에서 잠시 사용할 수 없었던 Windows Forms에 포함되었습니다. 이러한 항목은 .NET Framework 2.0에서 새 컨트롤로 바뀌었으며 .NET Core 3.1용 데스크톱 SDK에서 제거되었습니다.

테이블 확장

제거된 컨트롤	권장된 대체	제거된 연결 API
DataGrid	DataGridView	DataGridCell DataGridRow DataGridTableCollection DataGridColumnCollection 데이터 그리드 테이블 스타일 (DataGridTableStyle) DataGridColumnStyle 데이터 그리드 라인 스타일 데이터 그리드 부모 행 레이블 데이터그리드부모행레이블스타일 데이터그리드불트컬럼 데이터 그리드 텍스트 상자 GridColumnStylesCollection GridTableStylesCollection 히트 테스트 타입
도구 모음	ToolStrip	툴바 외형
도구 모음 버튼	ToolStripButton	ToolBarButtonClickEventArgs 툴바버튼클릭이벤트핸들러 도구 모음 버튼 스타일 툴바텍스트정렬
컨텍스트 메뉴	ContextMenuStrip	
Menu	ToolStripDropDown ToolStripDropDownMenu	메뉴 항목 컬렉션
주메뉴	MenuStrip	
메뉴 항목	ToolStripMenuItem	

애플리케이션을 .NET Core 3.1로 업데이트하고 대체 컨트롤로 이동하는 것이 좋습니다. 컨트롤을 바꾸는 것은 기본적으로 유형을 "찾고 대체"하는 간단한 프로세스입니다.

C++/CLI

Windows만 해당

C++/CLI("관리되는 C++"라고도 함) 프로젝트를 만들기 위한 지원이 추가되었습니다. 이러한 프로젝트에서 생성된 이진 파일은 .NET Core 3.0 이상 버전과 호환됩니다.

Visual Studio 2019 버전 16.4에서 C++/CLI에 대한 지원을 추가하려면 C++ 워크로드로 데스크톱 개발을 설치합니다. 이 워크로드는 Visual Studio에 다음의 두 가지 템플릿을 추가합니다.

- CLR 클래스 라이브러리(.NET Core)
- CLR 빈 프로젝트(.NET Core)

다음 단계

- .NET Core 3.0 및 3.1 간의 호환성이 손상되는 변경을 검토합니다.
- Windows Forms 앱용 .NET Core 3.1의 주요 변경 사항을 검토합니다.

.NET Core 3.1의 호환성을 깨뜨리는 변경사항

2025. 06. 17.

.NET Core 또는 ASP.NET Core 버전 3.1로 마이그레이션하는 경우 이 문서에 나열된 주요 변경 내용이 앱에 영향을 줄 수 있습니다.

ASP.NET Core

HTTP: 브라우저 SameSite 변경 내용이 인증에 영향

Chrome 및 Firefox와 같은 일부 브라우저에서는 쿠키에 대한 `SameSite`의 구현에 호환성이 손상되는 변경이 수행되었습니다. 이 변경 사항은 `SameSite=None`을 전송하여 옵트아웃해야 하는 OpenID Connect 및 WS-Federation과 같은 원격 인증 시나리오에 영향을 줍니다. 그러나 `SameSite=None`이(가) iOS 12 및 일부 이전 버전의 기타 브라우저에서 중단됩니다. 이 앱은 이러한 버전을 감지하고 `SameSite`를 생략해야 합니다.

이 문제에 대한 자세한 내용은 dotnet/aspnetcore#14996 을 참조하세요.

도입된 버전

3.1 미리 보기 1

기존 동작

`SameSite`은(는) HTTP 쿠키의 2016 초안 표준 확장입니다. CSRF(교차 사이트 요청 위조)를 완화하기 위한 것입니다. 원래는 새 매개 변수를 추가하여 서버에서 옵트인하는 기능으로 설계되었습니다. ASP.NET Core 2.0에서 `SameSite`에 대한 초기 지원을 추가했습니다.

새 동작

Google은 이전 버전과 호환되지 않는 새 초안 표준을 제안했습니다. 이 표준은 기본 모드를 `Lax`으로 변경하고 옵트아웃할 새 항목 `None`을 추가합니다. 대부분의 앱 쿠키의 경우 `Lax`가 충분하지만, OpenID Connect 및 WS-Federation 로그인과 같은 교차 사이트 시나리오는 방해합니다. 대부분의 OAuth 로그인은 요청 진행 방법의 차이로 인해 영향을 받지 않습니다. 새 `None` 매개 변수로 인해 이전 초안 표준(예: iOS 12)을 구현한 클라이언트에 호환성 문제가 발생합니다. Chrome 80에는 변경 내용이 포함됩니다. Chrome 제품 출시 타임라인을 보려면 SameSite 업데이트 를 참조하세요.

ASP.NET Core 3.1이 새 `SameSite` 동작을 구현하도록 업데이트되었습니다. 업데이트는 `SameSiteMode.None`의 동작을 재정의하여 `SameSite=None`을 내보내고 새 값 `SameSiteMode.Unspecified`를 추가하여 `SameSite` 특성을 생략합니다. 일부 구성 요소에서 OpenID Connect 상관 관계 및 nonce 쿠키와 같은 특정 시나리오와 관련된 값을 설정하지만 이제 모든 쿠키 API는 `Unspecified`를 기본값으로 설정됩니다.

이 영역의 다른 최근 변경 내용에 관해서는 HTTP: 일부 쿠키 SameSite 기본값이 없음으로 변경됨을 참조하세요. ASP.NET Core 3.0에서 대부분의 기본값이 SameSiteMode.Lax에서 SameSiteMode.None(으)로 변경되었지만, 이전 표준은 여전히 사용됩니다.

변경 이유

이전 텍스트에서 설명한 대로 브라우저 및 사양이 변경됩니다.

권장 작업

타사 로그인 등 원격 사이트와 상호 작용하는 앱은 다음 작업을 수행해야 합니다.

- 이 시나리오를 여러 브라우저에서 테스트합니다.
- 이전 브라우저 지원에서 설명된 대로 쿠키 정책 브라우저 검색 완화를 적용합니다.

테스트 및 브라우저 검색 지침은 다음 섹션을 참조하세요.

사용자가 영향을 받는지 확인합니다.

새 동작을 옵트인할 수 있는 클라이언트 버전을 사용하여 웹앱을 테스트합니다. Chrome, Firefox 및 Microsoft Edge Chromium에는 테스트에 사용할 수 있는 새 옵트인 기능 플래그가 있습니다. 패치, 특히 Safari를 적용한 후 앱이 이전 클라이언트 버전과 호환되는지 확인합니다. 자세한 내용은 이전 브라우저 지원을 참조하세요.

크롬

Chrome 78 이상에서는 잘못된 테스트 결과를 생성합니다. 이러한 버전에는 임시 해결 방법이 있으며 2분 이내 쿠키를 허용합니다. 적절한 테스트 플래그를 사용하도록 설정하면 Chrome 76 및 77에서 더 정확한 결과를 생성합니다. 새 동작을 테스트하려면 `chrome://flags/#same-site-by-default-cookies`를 활성화된 상태로 전환하십시오. Chrome 75 이전 버전이 새 `None` 설정에서 실패한다고 보고됩니다. 자세한 내용은 이전 브라우저 지원을 참조하세요.

Google은 이전 Chrome 버전을 사용하도록 설정할 수 없습니다. 그러나 테스트에 충분한 이전 버전의 Chromium을를 다운로드할 수 있습니다. Chromium 다운로드에 있는 지침을 따릅니다.

- Chromium 76 Win64
- Chromium 74 Win64

사파리

Safari 12는 이전 초안을 엄격하게 구현했으며 쿠키에 새 `None` 값이 표시되는 경우 실패합니다. 이전 브라우저 지원에 표시된 브라우저 검색 코드를 통해 이를 방지해야 합니다. Microsoft Authentication Library(MSAL), Active Directory 인증 라이브러리(ADAL) 또는 사용 중인 라이브러리를 사용하여 Safari 12 및 13과 WebKit 기반 OS 스타일 로그인도 테스트해야 합니다. 문제는 기본 OS 버전에 따라 달라집니다. OSX Mojave 10.14 및 iOS 12는 새 동작에 호환성 문제가 있는 것으로 알려져 있습니다. OSX Catalina 10.15 또는 iOS 13으로 업그레이드하면 문제가 해결됩니다. Safari에는 현재 새 사양 동작을 테스트하기 위한 옵트인 플래그가 없습니다.

Firefox

새 표준에 대한 Firefox 지원은 기능 플래그 `about:config`를 사용하여 `network.cookie.sameSite.laxByDefault` 페이지에서 옵트인하여 버전 68 이상에서 테스트할 수 있습니다. 이전 버전의 Firefox에서는 호환성 문제가 보고되지 않았습니다.

Microsoft Edge

Microsoft Edge는 이전 `SameSite` 표준을 지원하지만, 버전 44부터 새 표준에 호환성 문제가 없습니다.

Microsoft Edge Chromium

기능 플래그는 `edge://flags/#same-site-by-default-cookies`입니다. Microsoft Edge Chromium 78을 사용하여 테스트할 때 호환성 문제가 관찰되지 않았습니다.

전자

Electron 버전에는 이전 버전의 Chromium이 포함되어 있습니다. 예를 들어, Microsoft Teams에서 사용하는 Electron의 버전은 Chromium 66이며, 이는 이전 동작을 보입니다. 사용자의 제품에서 사용하는 Electron 버전으로 자체 호환성 테스트를 수행합니다. 자세한 내용은 이전 브라우저 지원을 참조하세요.

이전 브라우저 지원

2016 `SameSite` 표준에 따라 알 수 없는 값을 `SameSite=Strict` 값으로 처리합니다. 따라서 원래 표준을 지원하는 모든 이전 브라우저는 값이 `SameSite`인 `None` 속성이 표시될 때 중단될 수 있습

니다. 이러한 이전 브라우저를 지원하려는 경우 웹앱은 브라우저 검색을 구현해야 합니다. `User-Agent` 요청 헤더 값이 매우 불안정하고 매주 변경되기 때문에 ASP.NET Core에서 브라우저 검색을 구현하지 않습니다. 대신, 쿠키 정책의 확장 지점을 사용하여 `User-Agent`별 논리를 추가할 수 있습니다.

*Startup.cs*에서 다음 코드를 추가합니다.

C#

```
private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        // TODO: Use your User Agent library of choice here.
        if (/* UserAgent doesn't support new behavior */)
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });
}

public void Configure(IApplicationBuilder app)
{
    // Before UseAuthentication or anything else that writes cookies.
    app.UseCookiePolicy();

    app.UseAuthentication();
    // code omitted for brevity
}
```

옵트아웃 스위치

`Microsoft.AspNetCore.SuppressSameSiteNone` 호환성 스위치를 사용하면 새 ASP.NET Core 쿠키 동작을 임시로 옵트아웃할 수 있습니다. 프로젝트에서 *runtimeconfig.template.json* 파일에 다음 JSON을 추가합니다.

JSON

```
{
  "configProperties": {
    "Microsoft.AspNetCore.SuppressSameSiteNone": "true"
  }
}
```

다른 버전

관련 `SameSite` 패치가 곧 출시될 예정입니다.

- ASP.NET Core 2.1, 2.2 및 3.0
- `Microsoft.Owin` 4.1
- `System.Web`(.NET Framework 4.7.2 이상)

카테고리

ASP.NET

영향을 받는 API

- Microsoft.AspNetCore.Builder.CookiePolicyOptions.MinimumSameSitePolicy
- Microsoft.AspNetCore.Http.CookieBuilder.SameSite
- Microsoft.AspNetCore.Http.CookieOptions.SameSite
- Microsoft.AspNetCore.Http.SameSiteMode
- Microsoft.Net.Http.Headers.SameSiteMode
- Microsoft.Net.Http.Headers.SetCookieHeaderValue.SameSite

배치

64비트 Windows의 x86 호스트 경로

MSBuild

디자인 타임 빌드는 최상위 패키지 참조만 반환합니다.

.NET Core SDK 3.1.400부터는 `RunResolvePackageDependencies` 대상에 최상위 패키지 참조만 반환됩니다.

도입된 버전

.NET Core SDK 3.1.400

변경 내용 설명

이전 버전의 .NET Core SDK에서 `RunResolvePackageDependencies` 대상은 NuGet 자산 파일의 정보를 포함하는 다음의 MSBuild 항목을 생성했습니다.

- `PackageDefinitions`
- `PackageDependencies`
- `TargetDefinitions`
- `FileDefinitions`
- `FileDependencies`

이 데이터는 Visual Studio에서 솔루션 탐색기에서 종속성 노드를 채우는 데 사용됩니다. 그러나 많은 양의 데이터가 될 수 있으며 종속성 노드를 확장하지 않는 한 데이터가 필요하지 않습니다.

.NET Core SDK 버전 3.1.400부터 이러한 항목의 대부분은 기본적으로 생성되지 않습니다. 형식 `Package` 의 항목만 반환됩니다. Visual Studio에서 종속성 노드를 채우는 데 항목이 필요한 경우 자산 파일에서 직접 정보를 읽습니다.

변경 이유

이 변경 내용은 Visual Studio 내에서 솔루션 로드 성능을 개선하기 위해 도입되었습니다. 이전에는 대부분의 사용자가 절대 보지 않을 많은 참조를 로드해야 했던 모든 패키지 참조가 로드되었습니다.

권장 작업

이러한 항목의 생성에 의존하는 MSBuild 논리가 있는 경우, 프로젝트 파일에서 `EmitLegacyAssetsFileItems` 속성을 `true`로 설정합니다. 이 설정을 사용하면 모든 항목이 만들어지는 이전 동작을 사용할 수 있습니다.

카테고리

MSBuild

영향을 받는 API

해당 없음(N/A)

SDK (소프트웨어 개발 키트)

루트 폴더의 도구 매니페스트

윈도우 폼즈 (Windows Forms)

제거된 컨트롤

.NET Core 3.1부터 일부 Windows Forms 컨트롤을 더 이상 사용할 수 없습니다.

변경 내용 설명

.NET Core 3.1부터 다양한 Windows Forms 컨트롤을 더 이상 사용할 수 없습니다. 더 나은 디자인과 지원을 제공하는 대체 컨트롤이 .NET Framework 2.0에 도입되었습니다. 사용되지 않는 컨트롤은 이전에 디자이너 도구 상자에서 제거되었지만 여전히 사용할 수 있었습니다.

다음 형식은 더 이상 사용할 수 없습니다.

- ContextMenu
- DataGrid
- DataGrid.HitTestType
- DataGrid.HitTestInfo
- DataGridBoolColumn
- DataGridCell
- DataGridColumnStyle
- DataGridColumnStyle.DataGridColumnHeaderAccessibleObject
- DataGridColumnStyle.CompModSwitches
- DataGridLineStyle
- DataGridParentRowsLabelStyle
- DataGridPreferredColumnWidthTypeConverter
- DataGridTableStyle
- DataGridTextBox
- DataGridTextBoxColumn
- GridColumnStylesCollection
- GridTablesFactory
- GridTableStylesCollection
- IDataGridEditingService
- IMenuEditorService
- MainMenu

- Menu
- Menu.MenuItemCollection
- MenuItem
- ToolBar
- ToolBarAppearance
- ToolBarButton
- ToolBar.ToolBarButtonCollection
- ToolBarButtonClickEventArgs
- ToolBarButtonStyle
- ToolBarTextAlign

도입된 버전

3.1

권장 작업

제거된 각 컨트롤에는 권장되는 대체 컨트롤이 있습니다. 다음 표를 참조하세요.

테이블 확장

제거된 제어 (API)	권장 교체	제거된 관련된 API
컨텍스트 메뉴	컨텍스트 메뉴 스트립	
DataGrid	데이터 그리드 보기 (DataGridView)	DataGridCell, DataGridRow, DataGridTableCollection, DataGridColumnCollection, DataGridTableStyle, DataGridColumnStyle, DataGridLineStyle, DataGridParentRowsLabel, DataGridParentRowsLabelStyle, DataGridBoolColumn, DataGridTextBox, GridColumnStylesCollection, GridTableStylesCollection, HitTestType
주메뉴	메뉴 스트립	
메뉴	ToolStripDropDown(도구 막대 드롭다운), ToolStripDropDownMenu(도구 막대 드롭다운 메뉴)	메뉴 항목 컬렉션
메뉴 항목	도구막대 메뉴 항목	

제거된 제어 (API)	권장 교체	제거된 관련된 API
도구 모음	툴스트립	툴바 외형
도구 모음 버튼	도구 모음 버튼	툴바 버튼 클릭 이벤트 인자 (ToolBarButtonClickEventArgs), 툴바 버튼 클릭 이벤트 핸들러 (ToolBarButtonClickEventHandler), 툴바 버튼 스타일 (ToolBarButtonStyle), 툴바 텍스트 정렬 (ToolBarTextAlign)

카테고리

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- System.Windows.Forms.ContextMenu
- System.Windows.Forms.GridColumnStylesCollection
- System.Windows.Forms.GridTablesFactory
- System.Windows.Forms.GridTableStylesCollection
- System.Windows.Forms.IDataGridEditingService
- System.Windows.Forms.MainMenu
- System.Windows.Forms.Menu
- System.Windows.Forms.Menu.MenuItemCollection
- System.Windows.Forms.MenuItem
- System.Windows.Forms.ToolBar
- System.Windows.Forms.ToolBar.ToolBarButtonCollection
- System.Windows.Forms.ToolBarAppearance
- System.Windows.Forms.ToolBarButton
- System.Windows.Forms.ToolBarButtonClickEventArgs
- System.Windows.Forms.ToolBarButtonStyle
- System.Windows.Forms.ToolBarTextAlign
- System.Windows.Forms.DataGrid
- System.Windows.Forms.DataGrid.HitTestType
- System.Windows.Forms.DataGridBoolColumn
- System.Windows.Forms.DataGridCell
- System.Windows.Forms.DataGridColumnStyle
- System.Windows.Forms.DataGridLineStyle
- System.Windows.Forms.DataGridParentRowsLabelStyle
- System.Windows.Forms.DataGridPreferredColumnWidthTypeConverter

- System.Windows.Forms.DataGridTableStyle
- System.Windows.Forms.DataGridTextBox
- System.Windows.Forms.DataGridTextBoxColumn
- System.Windows.Forms.Design.IMenuEditorService

툴팁이 표시될 경우 CellFormatting 이벤트가 발생하지 않습니다.

이제 DataGridView는 마우스로 셀 위에 가리키거나 키보드로 선택할 때 셀의 텍스트와 오류 툴팁을 표시합니다. 툴팁이 표시되면 DataGridView.CellFormatting 이벤트가 발생하지 않습니다.

변경 내용 설명

.NET Core 3.1 이전에는 DataGridView 속성이 ShowCellToolTips로 설정된 `true` 셀이 마우스를 가리키면 셀의 텍스트와 오류에 대한 툴팁이 표시되었습니다. 키보드를 통해 셀을 선택할 때 도구 설명이 표시되지 않았습니다(예: Tab 키, 바로 가기 키 또는 화살표 탐색 사용). 사용자가 셀을 편집했을 때, DataGridView가 여전히 편집 모드인 동안 ToolTipText 속성이 설정되지 않은 셀 위에 마우스를 올리면, 셀에 표시할 텍스트를 서식화하기 위한 CellFormatting 이벤트가 발생합니다.

.NET Core 3.1부터 접근성 표준을 충족하기 위해, DataGridView 속성이 ShowCellToolTips로 설정된 `true`에는 셀을 마우스로 가리키거나 키보드로 선택할 때 셀의 텍스트와 오류에 대한 도구 설명이 표시됩니다. 이 변경의 결과로, 가 편집 모드에 있는 동안 속성 집합이 없는 셀을 마우스로 가리키면 이벤트는 발생하지 않습니다. . 마우스를 올린 셀의 내용이 셀에 표시되는 대신 툴팁으로 표시되기 때문에 이벤트가 발생하지 않습니다.

도입된 버전

3.1

권장 작업

CellFormatting 편집 모드에 있는 동안 DataGridView 이벤트에 종속된 코드를 리팩터링합니다.

카테고리

윈도우 폼즈 (Windows Forms)

영향을 받는 API

없음

참고하십시오

- .NET Core 3.1의 새로운 기능

.NET Core 3.0의 새로운 기능

아티클 • 2023. 05. 10.

이 문서에서는 .NET Core 3.0의 새로운 기능을 설명합니다. 가장 중요한 개선 사항 중 하나는 Windows 데스크톱 애플리케이션에 대한 지원(Windows만 해당)입니다. .NET Core 3.0 SDK 구성 요소 Windows 데스크톱을 사용하여 Windows Forms 및 Windows Presentation Foundation(WPF) 애플리케이션을 포트할 수 있습니다. 분명히 말하지만, Windows 데스크톱 구성 요소는 Windows에서만 지원되고 포함됩니다. 자세한 내용은 이 문서 후반부의 Windows 데스크톱 섹션을 참조하세요.

.NET Core 3.0에서는 C# 8.0에 대한 지원이 추가되었습니다. Visual Studio 2019 버전 16.3 이상, Mac용 Visual Studio 8.3 이상 또는 Visual Studio Code 를 최신 **C# 확장**과 함께 사용하는 것이 좋습니다.

Windows, macOS 또는 Linux에서 지금 바로 .NET Core 3.0을 다운로드하여 시작 하세요.

릴리스에 대한 자세한 내용은 .NET Core 3.0 알림 을 참조하세요.

.NET Core 3.0 RC 1은 Microsoft에서 프로덕션용으로 간주되었으며 완전히 지원되었습니다. 미리 보기 릴리스를 사용하는 경우, 계속 지원을 받으려면 RTM 버전으로 이동해야 합니다.

언어 향상 C# 8.0

null 허용 참조 형식 기능, 비동기 스트림, 추가 패턴 등을 포함하는 C# 8.0도 이 릴리스의 일부입니다. C# 8.0 기능에 대한 자세한 내용은 C# 8.0의 새로운 기능을 참조하세요.

C#8.0 언어 기능과 관련된 자습서:

- 자습서: nullable 참조 형식 및 nullable이 아닌 참조 형식을 사용하여 디자인 의도를 보다 명확하게 표현
- 자습서: C# 8.0 및 .NET Core 3.0을 사용하여 비동기 스트림 생성 및 사용
- 자습서: 패턴 일치를 사용하여 형식 기반 및 데이터 기반 알고리즘 빌드

아래에 설명된 다음 API 기능을 지원하기 위해 언어 향상 기능이 추가되었습니다.

- 범위 및 인덱스
- 비동기 스트림

.NET Standard 2.1

.NET Core 3.0에서는 **.NET Standard 2.1**을 구현합니다. 하지만 기본 `dotnet new classlib` 템플릿은 여전히 **.NET Standard 2.0**을 대상으로 하는 프로젝트를 생성합니다. **.NET 표준 2.1**을 대상으로 지정하려면 프로젝트 파일을 편집하고 `TargetFramework` 속성을 `netstandard2.1`로 변경하세요.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Visual Studio를 사용하고 있는 경우 Visual Studio 2017은 **.NET Standard 2.1** 또는 **.NET Core 3.0**을 지원하지 않으므로 Visual Studio 2019가 필요합니다.

컴파일/배포

기본 실행 파일

이제 .NET Core에서 기본적으로 프레임워크 종속 실행 파일을 빌드합니다. 이는 전역적으로 설치된 .NET Core 버전을 사용하는 애플리케이션을 위한 새로운 동작입니다. 지금까지는 자체 포함 배포만 실행 파일을 생성했습니다.

`dotnet build` 또는 `dotnet publish` 중에 사용 중인 SDK의 환경 및 플랫폼과 일치하는 실행 파일(**appHost**)이 생성됩니다. 다른 네이티브 실행 파일과 마찬가지로 이러한 실행 파일에서도 다음과 같은 동일한 기능을 예상할 수 있습니다.

- 실행 파일을 두 번 클릭할 수 있습니다.
- Windows의 `myapp.exe`, Linux 및 macOS의 `./myapp`과 같은 애플리케이션을 명령 프롬프트에서 직접 시작할 수 있습니다.

macOS appHost 및 공증

'macOS만 해당'

공증된 macOS용 .NET Core SDK 3.0부터, 기본 실행 파일(appHost)을 생성하는 설정이 기본적으로 사용하지 않도록 설정됩니다. 자세한 내용은 macOS Catalina 공증과 이것이 .NET Core 다운로드 및 프로젝트에 미치는 영향을 참조하세요.

appHost 설정이 사용하도록 설정된 경우, 빌드 또는 게시할 때 .NET Core가 네이티브 Mach-O 실행 파일을 생성합니다. 앱을 `dotnet run` 명령을 사용하여 소스 코드에서 실행하거나 Mach-O 실행 파일을 직접 시작하면 앱이 appHost 컨텍스트에서 실행됩니다.

사용자가 appHost 없이 프레임워크 종속 앱을 시작할 수 있는 유일한 방법은 `dotnet <filename.dll>` 명령을 사용하는 것입니다. appHost는 앱을 자체 포함 방식으로 게시하면 항상 만들어집니다.

appHost는 다음과 같이 프로젝트 수준에서 구성하거나 `-p:UseAppHost` 매개 변수를 사용하여 특정 `dotnet` 명령에 대해 켜거나 끌 수 있습니다.

- 프로젝트 파일

  XML

  ```xml
  <PropertyGroup>
    <UseAppHost>true</UseAppHost>
  </PropertyGroup>
  ```

- 명령줄 매개 변수

  .NET CLI

  ```
  dotnet run -p:UseAppHost=true
  ```

`UseAppHost` 설정에 대한 자세한 내용은 MSBuild properties for Microsoft.NET.Sdk(Microsoft.NET.Sdk의 MSBuild 속성)를 참조하세요.

단일 실행 파일

`dotnet publish` 명령은 플랫폼별 단일 실행 파일로 앱 패키징을 지원합니다. 실행 파일은 자동 압축 풀기 파일이며 앱을 실행하는 데 필요한 모든 종속 항목(네이티브 포함)을 포함하고 있습니다. 앱을 처음 실행하면 애플리케이션은 앱 이름과 빌드 식별자에 기반하여 디렉터리로 압축이 풀립니다. 해당 애플리케이션을 다시 실행하면 시작이 빨라집니다. 새 버전을 사용한 경우가 아니라면 두 번째에는 애플리케이션의 압축을 풀 필요가 없습니다.

단일 실행 파일을 게시하려면 `dotnet publish` 명령을 사용하여 프로젝트 또는 명령줄에 `PublishSingleFile`을 설정합니다.

XML

```xml
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
```

```
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

또는

.NET CLI

```
dotnet publish -r win10-x64 -p:PublishSingleFile=true
```

단일 파일 게시에 대한 자세한 내용은 단일 파일 번들러 설계 문서를 참조하세요.

어셈블리 트리밍

.NET Core SDK 3.0에는 IL을 분석하고 사용되지 않는 어셈블리를 잘라내어 앱의 크기를 줄일 수 있는 도구가 포함되어 있습니다.

자체 포함 앱에는 호스트 컴퓨터에 .NET을 설치하지 않고도 코드를 실행하는 데 필요한 모든 요소가 포함됩니다. 그러나 앱을 실행하는 데 프레임워크의 작은 하위 집합만 필요한 경우가 많으므로 사용되지 않는 다른 라이브러리를 제거할 수 있습니다.

이제 .NET Core에 IL 트리머 도구를 사용하여 앱의 IL을 검사하는 설정이 포함되어 있습니다. 이 도구는 필요한 코드를 검색한 다음, 사용되지 않는 라이브러리를 자릅니다. 이 도구를 통해 일부 앱의 배포 크기를 훨씬 줄일 수 있습니다.

이 도구를 사용하려면 프로젝트에서 `<PublishTrimmed>` 설정을 추가하고 자체 포함 앱을 게시합니다.

XML

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

.NET CLI

```
dotnet publish -r <rid> -c Release
```

예를 들어 포함된 기본 "hello world" 새 콘솔 프로젝트 템플릿은 게시될 때 크기가 약 70MB입니다. `<PublishTrimmed>`를 사용하면 크기가 약 30MB로 줄어듭니다.

잘라낼 때 리플렉션이나 관련된 동적 기능을 사용하는 애플리케이션 또는 프레임워크(ASP.NET Core, WPF 등)가 중단되는 경우가 많다는 것을 고려해야 합니다. 이러한 중단은 트리머에서 이 동적 동작을 알지 못하고 리플렉션에 필요한 프레임워크 유형을 확인할

수 없기 때문에 발생합니다. 이 시나리오를 인식하도록 IL 트리머 도구를 구성할 수 있습니다.

무엇보다도, 잘라낸 후 앱을 테스트해야 합니다.

IL 트리머 도구에 관한 자세한 내용은 설명서를 참조하거나 mono/linker 리포지토리를 방문하세요.

계층화된 컴파일

계층화된 컴파일(TC)은 .NET Core 3.0에서 기본적으로 켜져 있습니다. 런타임 시 JIT(Just-In-Time) 컴파일러를 더욱 유연하게 사용하여 성능을 개선할 수 있도록 하는 기능입니다.

계층화된 컴파일의 주요 혜택은 품질은 낮지만 빠른 계층의 (재)JIT 메서드 또는 품질은 높지만 느린 계층의 (재)JIT 메서드의 두 가지 방법을 제공하는 것입니다. 품질은 메서드가 얼마나 제대로 최적화되었는지를 나타냅니다. TC는 시작에서 정적인 상태까지 다양한 실행 단계를 거치므로 애플리케이션의 성능을 개선하는 데 도움이 됩니다. 계층화된 컴파일을 사용하지 않도록 설정하는 경우 모든 메서드는 시작 성능보다 정적인 상태 성능에 편향된 단일 방식으로 컴파일됩니다.

TC를 사용하도록 설정하면 앱이 시작될 때 메서드 컴파일에 다음과 같은 동작이 적용됩니다.

- 메서드에 Ahead Of Time 컴파일 코드(ReadyToRun)가 있는 경우 사전 생성된 코드가 사용됩니다.
- 그렇지 않으면 메서드가 JIT 컴파일됩니다. 일반적으로 이 메서드는 값 형식의 제네릭입니다.
  - *빠른 JIT*은 품질이 더 낮거나 덜 최적화된 코드를 더욱 빠르게 생성합니다. .NET Core 3.0에서 빠른 JIT는 루프를 포함하지 않은 메서드에 기본적으로 사용하도록 설정되며, 시작하는 동안 사용하는 것이 좋습니다.
  - 완전히 최적화된 JIT는 품질이 더 높거나 더 최적화된 코드를 더욱 느리게 생성합니다. 빠른 JIT를 사용하지 않는 메서드의 경우(예: 메서드가 MethodImplOptions.AggressiveOptimization으로 특성이 지정된 경우) 완전히 최적화된 JIT가 사용됩니다.

자주 호출되는 메서드의 경우 Just-In-Time 컴파일러는 결과적으로 백그라운드에서 완전히 최적화된 코드를 만듭니다. 그런 다음, 최적화된 코드는 해당 메서드에 대해 미리 컴파일된 코드를 대체합니다.

빠른 JIT에 의해 생성된 코드는 실행 속도가 저하되거나, 더 많은 메모리를 할당하거나, 더 많은 스택 공간을 사용할 수 있습니다. 이슈가 있는 경우 프로젝트 파일에서 이 MSBuild 속성을 사용하여 빠른 JIT를 사용하지 않도록 설정할 수 있습니다.

XML

```xml
<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>
```

TC를 완전히 사용하지 않도록 설정하려면 프로젝트 파일에서 이 MSBuild 속성을 사용합니다.

XML

```xml
<PropertyGroup>
  <TieredCompilation>false</TieredCompilation>
</PropertyGroup>
```

팁

프로젝트 파일에서 이러한 설정을 변경하는 경우 새 설정이 반영되도록 클린 빌드(`obj` 및 `bin` 디렉터리를 삭제한 후 다시 빌드)를 수행해야 할 수 있습니다.

런타임에 컴파일을 구성하는 방법에 대한 자세한 내용은 컴파일을 위한 런타임 구성 옵션을 참조하세요.

ReadyToRun 이미지

R2R(ReadyToRun) 형식으로 애플리케이션 어셈블리를 컴파일하면 .NET Core 애플리케이의 시작 시간을 향상할 수 있습니다. R2R은 AOT(Ahead-Of-Time) 컴파일 양식입니다.

R2R 이진 파일은 애플리케이션이 로드될 때 JIT(Just-In-Time) 컴파일러에서 수행해야 하는 작업량을 줄여 시작 성능을 향상합니다. 이진 파일에는 JIT에서 생성되는 코드와 비슷한 네이티브 코드가 포함되어 있습니다. 그러나 R2R 이진 파일은 일부 시나리오에서 필요한 IL(중간 언어) 코드와 동일한 코드의 네이티브 버전을 모두 포함하므로 크기가 더 큽니다. R2R은 Linux x64 또는 Windows x64와 같은 특정 런타임 환경(RID)을 대상으로 하는 자체 포함 앱을 게시하는 경우에만 사용할 수 있습니다.

프로젝트를 ReadyToRun으로 컴파일하려면 다음을 수행합니다.

1. 프로젝트에 `<PublishReadyToRun>` 설정 추가:

   XML

   ```xml
   <PropertyGroup>
     <PublishReadyToRun>true</PublishReadyToRun>
   ```

```
</PropertyGroup>
```

2. 자체 포함 앱을 게시합니다. 예를 들어 이 명령은 Windows 64비트 버전용 자체 포함 앱을 만듭니다.

```
.NET CLI

dotnet publish -c Release -r win-x64 --self-contained
```

교차 플랫폼/아키텍처 제한 사항

현재 ReadyToRun 컴파일러는 교차 대상 지정을 지원하지 않습니다. 지정된 대상에서 컴파일해야 합니다. 예를 들어 Windows x64용 R2R 이미지를 만들려는 경우 해당 환경에서 게시 명령을 실행해야 합니다.

교차 대상 지정 예외:

- Windows x64를 사용하여 Windows Arm32, Arm64 및 x86 이미지를 컴파일할 수 있습니다.
- Windows x86을 사용하여 Windows Arm32 이미지를 컴파일할 수 있습니다.
- Linux x64를 사용하여 Linux Arm32 및 Arm64 이미지를 컴파일할 수 있습니다.

자세한 내용은 실행 준비를 참조하세요.

런타임/SDK

주 버전 런타임 롤포워드

.NET Core 3.0은 애플리케이션을 .NET Core의 최신 주 버전으로 롤포워드할 수 있는 옵트인(opt-in) 기능을 소개합니다. 또한, 롤포워드가 애플리케이션에 적용되는 방식을 제어하기 위해 새 설정이 추가되었습니다. 이 설정은 다음과 같은 방법으로 구성할 수 있습니다.

- 프로젝트 파일 속성: `RollForward`
- 런타임 구성 파일 속성: `rollForward`
- 환경 변수: `DOTNET_ROLL_FORWARD`
- 명령줄 인수: `--roll-forward`

다음 값 하나를 지정해야 합니다. 설정을 생략하면 **Minor**가 기본값입니다.

- **LatestPatch**
  가장 높은 패치 버전으로 롤포워드합니다. 부 버전 롤포워드를 사용하지 않도록 설정합니다.

- **Minor**
  요청된 부 버전이 없을 경우 가장 낮은 높은 부 버전으로 롤포워드합니다. 요청된 부 버전이 있다면 **LatestPatch** 정책이 사용됩니다.
- **Major**
  요청된 주 버전이 없을 경우 가장 낮은 높은 주 버전으로 롤포워드합니다. 요청된 주 버전이 있다면 **Minor** 정책이 사용됩니다.
- **LatestMinor**
  요청된 부 버전이 있는 경우에도 가장 높은 부 버전으로 롤포워드합니다. 구성 요소 호스팅 시나리오를 위한 것입니다.
- **LatestMajor**
  요청된 주 버전이 있는 경우에도 가장 높은 주, 가장 높은 부 버전으로 롤포워드합니다. 구성 요소 호스팅 시나리오를 위한 것입니다.
- 사용 안 함
  롤포워드하지 않습니다. 지정된 버전에만 바인딩합니다. 이 정책은 최신 패치를 롤포워드할 수 있는 기능을 사용하지 않도록 설정하므로 일반 용도에는 좋지 않습니다. 이 값은 테스트용으로만 사용하는 것이 좋습니다.

Disable 설정 이외에도 모든 설정에서 사용 가능한 가장 높은 패치 버전을 사용합니다.

기본적으로 요청된 버전(애플리케이션에 대한 `.runtimeconfig.json`에 지정됨)이 릴리스 버전인 경우에만 릴리스 버전이 롤포워드를 고려합니다. 시험판 버전은 무시됩니다. 일치하는 릴리스 버전이 없는 경우 시험판 버전이 고려됩니다. 이 동작은 `DOTNET_ROLL_FORWARD_TO_PRERELEASE=1`을 설정하여 변경할 수 있으며, 이 경우 모든 버전이 항상 고려됩니다.

빌드 복사본 종속성

`dotnet build` 명령은 이제 애플리케이션에 대한 NuGet 종속성을 NuGet 캐시에서 빌드 출력 폴더로 복사합니다. 이전에는 종속성이 `dotnet publish` 중에만 복사되었습니다.

트리밍, Razor 페이지 게시 등 여전히 게시해야 하는 일부 작업이 있습니다.

로컬 도구

.NET core 3.0에서는 로컬 도구를 소개합니다. 로컬 도구는 전역 도구와 유사하지만 디스크의 특정 위치와 연결됩니다. 로컬 도구는 전역적으로 사용할 수 없으며 NuGet 패키지로 배포됩니다.

로컬 도구는 현재 디렉터리에 있는 매니페스트 파일 이름 `dotnet-tools.json`을 사용합니다. 이 매니페스트 파일은 해당 폴더 및 그 아래에서 사용할 수 있는 도구를 정의합니다.

코드를 사용하는 누구나 동일한 도구를 복구하고 사용할 수 있도록 코드로 매니페스트 파일을 배포할 수 있습니다.

전역 도구와 로컬 도구 둘 다, 호환되는 버전의 런타임이 필요합니다. 현재 NuGet.org의 많은 도구는 .NET Core 런타임 2.1을 대상으로 합니다. 이러한 도구를 전역 또는 로컬로 설치하려면 여전히 NET Core 2.1 런타임 을 설치해야 합니다.

새 global.json 옵션

global. json 파일에는 사용되는 .NET Core SDK 버전을 정의하려고 할 때 더 많은 유연성을 제공하는 새로운 옵션이 있습니다. 새 옵션은 다음과 같습니다.

- `allowPrerelease`: 사용할 SDK 버전을 선택할 때 SDK 확인자가 시험판 버전을 고려해야 하는지 여부를 나타냅니다.
- `rollForward`: SDK 버전을 선택할 때, 특정 SDK 버전이 누락된 경우 대체하거나 상위 버전을 사용하기 위한 지시문으로 사용할 롤포워드 정책을 나타냅니다.

기본값, 지원되는 값, 새 일치 규칙을 비롯한 변경 내용에 대한 자세한 내용은 global.json 개요를 참조하세요.

더 작은 가비지 수집 힙 크기

가비지 수집기의 기본 힙 크기가 감소되어 .NET Core에서 사용되는 메모리가 줄어듭니다. 이러한 변화는 최신 프로세서 캐시 크기의 생성 0 할당 예산과 부합됩니다.

가비지 수집 Large Page 지원

Large Page(또는 Linux Huge Page)는 운영 체제가 이러한 큰 페이지를 요청하는 애플리케이션의 성능을 개선하기 위해 네이티브 페이지 크기(보통 4K)보다 더 큰 메모리 영역을 구축할 수 있는 기능입니다.

이제 가비지 수집기는 Windows에서 큰 페이지를 할당하기 위해 선택할 수 있는 옵트인(opt-in) 기능으로 **GCLargePages** 설정을 사용하여 구성할 수 있습니다.

Windows 데스크톱 & COM

.NET Core SDK Windows Installer

Windows용 MSI 설치 관리자는 .NET Core 3.0부터 변경되었습니다. 이제 SDK 설치 관리자는 준비된 SDK 기반 밴드 릴리스를 하려고 합니다. 기능 밴드는 번호 버전의 *패치* 섹션에서 *수백 개*의 그룹에 정의되어 있습니다. 예를 들어, **3.0.*101*** 및 **3.0.*201***은 두 가지 기능

밴드의 버전이며, **3.0.*101*** 및 **3.0.*199***는 동일한 기능 밴드에 있습니다. 그리고 .NET Core SDK **3.0.*101***이 설치되어 있는 경우 .NET Core SDK**3.0.*100***은 머신에서 제거됩니다. .NET Core SDK **3.0.*200***이 동일한 머신에 설치되어 있는 경우 .NET Core SDK **3.0.*101***은 제거되지 않습니다.

버전 관리에 대한 자세한 내용은 .NET Core의 버전 관리 방법을 참조하세요.

Windows 바탕 화면

.NET Core 3.0은 Windows Presentation Foundation(WPF) 및 Windows Forms를 사용하여 Windows 데스크톱 애플리케이션을 지원합니다. 이러한 프레임워크는 XAML 아일랜드를 통해 Windows UI XAML 라이브러리(WinUI)의 최신 컨트롤 및 Fluent 스타일을 사용하는 것도 지원합니다.

Windows 데스크톱 구성 요소는 Windows .NET Core 3.0 SDK의 일부입니다.

다음 `dotnet` 명령을 사용하여 새 WPF 또는 Windows Forms 앱을 만들 수 있습니다.

.NET CLI

```
dotnet new wpf
dotnet new winforms
```

Visual Studio 2019에는 .NET Core 3.0 Windows Forms 및 WPF용 **새 프로젝트** 템플릿이 추가됩니다.

기존 .NET Framework 애플리케이션을 포트하는 방법에 대한 자세한 내용은 WPF 프로젝트 포트 및 Windows Forms 프로젝트 포트를 참조하세요.

WinForms의 높은 DPI

.NET Core Windows Forms 애플리케이션은 Application.SetHighDpiMode(HighDpiMode)을(를) 사용하여 높은 DPI 모드를 설정할 수 있습니다. `SetHighDpiMode` 메서드는 해당 설정이 `Application.Run` 전에 `App.Manifest` 또는 P/Invoke와 같은 다른 수단으로 설정된 경우가 아니라면 해당하는 높은 DPI 모드를 설정합니다.

System.Windows.Forms.HighDpiMode 열거형으로 표현되는 가능한 `highDpiMode` 값은 다음과 같습니다.

- `DpiUnaware`
- `SystemAware`
- `PerMonitor`
- `PerMonitorV2`

- `DpiUnawareGdiScaled`

높은 DPI 모드에 대한 자세한 내용은 Windows에서 높은 DPI 데스크톱 애플리케이션 개발을 참조하세요.

COM 구성 요소 생성

Windows에서 이제는 COM 호출 가능 관리형 구성 요소를 생성할 수 있습니다. 이 기능은 COM 추가 기능 모델을 통해 .NET Core를 사용할 뿐만 아니라 .NET Framework에 패리티를 제공하는 데 중요합니다.

*mscoree.dll*이 COM 서버처럼 사용되는 .NET Framework와는 달리, .NET Core는 COM 구성 요소 빌드 시 네이티브 시작 관리자 dll을 *bin* 디렉터리에 추가합니다.

COM 구성 요소를 만들고 사용하는 방법에 대한 예는 COM 데모 를 참조하세요.

Windows 네이티브 Interop

Windows는 플랫 C API, COM 및 WinRT의 형태로 다양한 네이티브 API를 제공합니다. .NET Core에서는 **P/Invoke**를 지원하지만, .NET Core 3.0은 **CoCreate COM API** 및 **Activate WinRT API**에 대한 기능을 추가합니다. 코드 예제는 Excel 데모 를 참조하세요.

MSIX 배포

MSIX는 새로운 Windows 애플리케이션 패키지 형식입니다. Windows 10에 .NET Core 3.0 데스크톱 애플리케이션을 배포하는 데 사용할 수 있습니다.

Visual Studio 2019에 제공되는 Windows 애플리케이션 패키징 프로젝트를 사용하면 자체 포함 .NET Core 애플리케이션을 사용하여 MSIX 패키지를 만들 수 있습니다.

NET Core 프로젝트 파일은 `<RuntimeIdentifiers>` 속성에 지원되는 런타임을 지정해야 합니다.

XML

```xml
<RuntimeIdentifiers>win-x86;win-x64</RuntimeIdentifiers>
```

Linux 기능 향상

Linux용 SerialPort

.NET Core 3.0은 Linux에서 System.IO.Ports.SerialPort에 대한 기본 지원을 제공합니다.

이전에는 .NET Core가 Windows에서만 `SerialPort` 사용을 지원했습니다.

Linux의 제한적 직렬 포트 지원에 대한 자세한 내용은 GitHub 이슈 #33146 을 참조하세요.

Docker 및 cgroup 메모리 한도

Docker를 사용하여 Linux에서 .NET Core 3.0을 실행하면 cgroup 메모리 한도에 훨씬 더 효과적입니다. 메모리 한도가 있는 Docker 컨테이너를 실행하면(`docker run -m` 사용) .NET Core 동작 방식이 바뀝니다.

- 기본 가비지 수집기(GC) 힙 크기: 최대 20mb 또는 컨테이너에서 75%의 메모리 한도
- 명시적 크기는 절대수 또는 cgroup 한도의 비율로 설정할 수 있습니다.
- GC 힙당 최소 예약된 세그먼트 크기는 16mb입니다. 이 크기는 머신에서 생성된 힙 수를 줄여 줍니다.

Raspberry Pi에 대한 GPIO 지원

GPIO 프로그래밍에 사용할 수 있는 두 개의 패키지가 NuGet에 릴리스되었습니다.

- System.Device.Gpio
- Iot.Device.Bindings

GPIO 패키지에는 *GPIO*, *SPI*, *I2C* 및 *PWM*디바이스용 API가 포함됩니다. IoT 바인딩 패키지에 디바이스 바인딩이 포함되어 있습니다. 자세한 내용은 디바이스 GitHub 리포지토리 를 참조하세요.

Arm64 Linux 지원

.NET Core 3.0은 Linux용 Arm64에 대한 지원을 추가합니다. Arm64의 기본 사용 사례는 현재 IoT 시나리오에 있습니다. 자세한 내용은 .NET Core Arm64 상태를 참조하세요.

Arm64의 .NET Core용 Docker 이미지는 Alpine, Debian 및 Ubuntu에서 사용할 수 있습니다.

> **참고**
>
> macOS Arm64(또는 "Apple Silicon") 및 Windows Arm64 운영 체제에 대한 지원은 나중에 .NET 6에 추가되었습니다.

보안

Linux의 TLS 1.3 & OpenSSL 1.1.1

이제 .NET Core는 지정된 환경에서 사용 가능한 경우 OpenSSL 1.1.1의 TLS 1.3 지원을 이용합니다. TLS 1.3:

- 클라이언트와 서버 간에 필요한 왕복 횟수가 감소하여 연결 시간이 향상됩니다.
- 더 이상 사용하지 않고 안전하지 않은 암호화 알고리즘을 제거하므로 보안이 향상됩니다.

사용 가능한 경우 .NET Core 3.0은 Linux 시스템에서 **OpenSSL 1.1.1**, **OpenSSL 1.1.0** 또는 **OpenSSL 1.0.2**를 사용합니다. **OpenSSL 1.1.1**이 사용 가능한 경우 System.Net.Security.SslStream 및 System.Net.Http.HttpClient 형식은 **TLS 1.3**을 사용합니다(클라이언트와 서버 둘 다 **TLS 1.3**을 지원한다고 가정).

> **ⓘ 중요**
>
> Windows 및 macOS에서는 아직 **TLS 1.3**을 지원하지 않습니다.

다음 C# 8.0 예제는 https://www.cloudflare.com에 연결하는 Ubuntu 18.10의 .NET Core 3.0을 보여 줍니다.

C#

```
using System;
using System.Net.Security;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace whats_new
{
    public static class TLS
    {
        public static async Task ConnectCloudFlare()
        {
            var targetHost = "www.cloudflare.com";

            using TcpClient tcpClient = new TcpClient();

            await tcpClient.ConnectAsync(targetHost, 443);

            using SslStream sslStream = new
SslStream(tcpClient.GetStream());

            await sslStream.AuthenticateAsClientAsync(targetHost);
            await Console.Out.WriteLineAsync($"Connected to {targetHost}
```

```
with {sslStream.SslProtocol}");
        }
    }
}
```

암호화 암호

.NET Core 3.0에서는 각각 System.Security.Cryptography.AesGcm 및 System.Security.Cryptography.AesCcm으로 구현된 **AES-GCM** 및 **AES-CCM** 암호에 대한 지원이 추가되었습니다. 이러한 알고리즘은 둘 다 AEAD(Authenticated Encryption with Association Data) 알고리즘 입니다.

다음 코드는 `AesGcm` 암호를 사용하여 임의 데이터를 암호화하고 암호를 해독하는 방법을 보여 줍니다.

C#

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace whats_new
{
    public static class Cipher
    {
        public static void Run()
        {
            // key should be: pre-known, derived, or transported via another
channel, such as RSA encryption
            byte[] key = new byte[16];
            RandomNumberGenerator.Fill(key);

            byte[] nonce = new byte[12];
            RandomNumberGenerator.Fill(nonce);

            // normally this would be your data
            byte[] dataToEncrypt = new byte[1234];
            byte[] associatedData = new byte[333];
            RandomNumberGenerator.Fill(dataToEncrypt);
            RandomNumberGenerator.Fill(associatedData);

            // these will be filled during the encryption
            byte[] tag = new byte[16];
            byte[] ciphertext = new byte[dataToEncrypt.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Encrypt(nonce, dataToEncrypt, ciphertext, tag,
associatedData);
            }
```

```
            // tag, nonce, ciphertext, associatedData should be sent to the
other part

            byte[] decryptedData = new byte[ciphertext.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Decrypt(nonce, ciphertext, tag, decryptedData,
associatedData);
            }

            // do something with the data
            // this should always print that data is the same
            Console.WriteLine($"AES-GCM: Decrypted data is
{(dataToEncrypt.SequenceEqual(decryptedData) ? "the same as" : "different
than")} original data.");
        }
    }
}
```

암호화 키 가져오기/내보내기

.NET Core 3.0은 표준 형식에서 비대칭 퍼블릭 키와 프라이빗 키를 가져오고 내보낼 수 있도록 지원합니다. X.509 인증서를 사용할 필요가 없습니다.

모든 키 형식(*RSA*, *DSA*, *ECDsa* 및 *ECDiffieHellman*)에서 다음 형식을 지원합니다.

- **공개 키**
  - X.509 SubjectPublicKeyInfo
- **프라이빗 키**
  - PKCS#8 PrivateKeyInfo
  - PKCS#8 EncryptedPrivateKeyInfo

RSA 키는 다음도 지원합니다.

- **공개 키**
  - PKCS#1 RSAPublicKey
- **프라이빗 키**
  - PKCS#1 RSAPrivateKey

내보내기 메서드는 DER로 인코드된 이진 데이터를 생성하고, 가져오기 메서드도 동일한 동작을 예상합니다. 키가 텍스트에 편리한 PEM 형식으로 저장된 경우 호출자는 가져오기 메서드를 호출하기 전에 콘텐츠를 base64로 디코드해야 합니다.

```
C#
```

```
using System;
using System.Security.Cryptography;

namespace whats_new
{
    public static class RSATest
    {
        public static void Run(string keyFile)
        {
            using var rsa = RSA.Create();

            byte[] keyBytes = System.IO.File.ReadAllBytes(keyFile);
            rsa.ImportRSAPrivateKey(keyBytes, out int bytesRead);

            Console.WriteLine($"Read {bytesRead} bytes, {keyBytes.Length -
bytesRead} extra byte(s) in file.");
            RSAParameters rsaParameters = rsa.ExportParameters(true);
            Console.WriteLine(BitConverter.ToString(rsaParameters.D));
        }
    }
}
```

PKCS#8 파일은 System.Security.Cryptography.Pkcs.Pkcs8PrivateKeyInfo로 검사하고 **PFX/PKCS#12** 파일은 System.Security.Cryptography.Pkcs.Pkcs12Info로 검사할 수 있습니다. **PFX/PKCS#12** 파일은 System.Security.Cryptography.Pkcs.Pkcs12Builder로 조작할 수 있습니다.

.NET Core 3.0 API 변경 내용

범위 및 인덱스

새 System.Index 형식을 인덱싱에 사용할 수 있습니다. 시작부터 계산되는 `int`의 인덱스를 만들거나, 접두사 `^` 연산자(C#)를 사용하여 끝부터 계산되는 인덱스를 만들 수 있습니다.

C#

```
Index i1 = 3;  // number 3 from beginning
Index i2 = ^4; // number 4 from end
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine($"{a[i1]}, {a[i2]}"); // "3, 6"
```

시작 인덱스와 끝 인덱스의 두 `Index` 값으로 구성되고 `x..y` 범위 식(C#)으로 작성할 수 있는 System.Range 유형도 있습니다. 그런 다음, 조각을 생성하는 `Range`로 인덱싱할 수 있습니다.

C#

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

자세한 내용은 범위 및 인덱스 자습서를 참조하세요.

비동기 스트림

IAsyncEnumerable<T> 형식은 IEnumerable<T>의 새로운 비동기 버전입니다. 이 언어를 사용하면 `IAsyncEnumerable<T>`을 통해 `await foreach`를 수행하여 요소를 사용하고 `yield return`을 사용하여 요소를 생성할 수 있습니다.

다음 예제에서는 비동기 스트림의 생성 및 사용을 둘 다 보여 줍니다. `foreach` 문은 비동기이며, `yield return`을 사용하여 호출자에 대한 비동기 스트림을 생성합니다. 이 패턴(`yield return` 사용)은 비동기 스트림 생성을 위한 권장 모델입니다.

C#

```
async IAsyncEnumerable<int> GetBigResultsAsync()
{
    await foreach (var result in GetResultsAsync())
    {
        if (result > 20) yield return result;
    }
}
```

`await foreach`를 수행할 수 있을 뿐 아니라, 비동기 반복기(예: `await` 및 `yield`가 둘 다 가능한 `IAsyncEnumerable/IAsyncEnumerator`를 반환하는 반복기)를 만들 수도 있습니다. 삭제해야 하는 개체의 경우 `Stream` 및 `Timer`와 같은 다양한 BCL 유형이 구현하는 `IAsyncDisposable`을 사용할 수 있습니다.

자세한 내용은 비동기 스트림 자습서를 참조하세요.

IEEE 부동 소수점

부동 소수점 API는 IEEE 754-2008 개정판을 준수하도록 업데이트되고 있습니다. 이러한 변경의 목표는 **필요한** 모든 작업을 노출하고 이 작업이 IEEE 사양을 준수하는지 확인하는 것입니다. 부동 소수점 개선 사항에 대한 자세한 내용은 .NET Core 3.0에서 부동 소수점 구문 분석 및 서식 지정 개선 사항을 참조하세요.

구문 분석 및 서식 지정 개선 사항:

- 모든 길이의 입력을 올바르게 구문 분석하고 반올림합니다.

- 음수 0을 올바르게 구문 분석하고 형식을 지정합니다.
- 대/소문자를 구분하지 않는 검사를 수행하여 `Infinity`와 `NaN`을 올바르게 구문 분석하고, 적용 가능한 경우 선행 `+` 옵션을 허용합니다.

새 System.Math API의 구성 내용:

- BitIncrement(Double) 및 BitDecrement(Double)
  `nextUp` 및 `nextDown` IEEE 연산에 해당합니다. 입력보다 크거나 작은 값을 각각 비교하는 최소 부동 소수점 숫자를 반환합니다. 예를 들어 `Math.BitIncrement(0.0)`는 `double.Epsilon`을 반환합니다.

- MaxMagnitude(Double, Double) 및 MinMagnitude(Double, Double)
  `maxNumMag` 및 `minNumMag` IEEE 연산에 해당하며 두 입력의 규모 중 더 크거나 작은 값을 반환합니다. 예를 들어 `Math.MaxMagnitude(2.0, -3.0)`는 `-3.0`을 반환합니다.

- ILogB(Double)
  `logB` IEEE 연산에 해당하며 정수값을 반환하고, 입력 매개 변수의 정수 이진 로그를 반환합니다. 이 메서드는 `floor(log2(x))`와 사실상 동일하지만 반올림 오류를 최소화하면서 수행됩니다.

- ScaleB(Double, Int32)
  정수 값을 취하는 `scaleB` IEEE 연산에 해당하며 사실상 `x * pow(2, n)`을 반환하지만 반올림 오류를 최소화하면서 수행됩니다.

- Log2(Double)
  `log2` IEEE 연산에 해당하며, 기본-2 로그를 반환합니다. 반올림 오류를 최소화합니다.

- FusedMultiplyAdd(Double, Double, Double)
  `fma` IEEE 연산에 해당하며, 단일 곱셈 누산기(fused multiply add) 계산을 수행합니다. 다시 말해, `(x * y) + z`을(를) 단일 연산으로 수행하기 때문에 반올림 오류가 최소화됩니다. 한 예로 `FusedMultiplyAdd(1e308, 2.0, -1e308)`는 `1e308`을 반환합니다. 일반 `(1e308 * 2.0) - 1e308`은 `double.PositiveInfinity`를 반환합니다.

- CopySign(Double, Double)
  `copySign` IEEE 연산에 해당하며, `x`의 값을 반환하지만 `y`의 부호를 반환합니다.

.NET 플랫폼 종속 내장 함수

SIMD 또는 **비트 조작 명령어** 세트와 같은 특정 perf-oriented CPU 명령어에 대한 액세스를 허용하는 API가 추가되었습니다. 이러한 명령어를 사용하면 특정 시나리오(효율적인 데이터 병렬 처리)에서 성능을 크게 향상시킬 수 있습니다.

적절한 경우 .NET 라이브러리는 성능을 개선하기 위해 이러한 명령을 사용하기 시작했습니다.

자세한 내용은 .NET 플랫폼 종속 내장 함수를 참조하세요.

향상된 .NET Core Version API

.NET Core 3.0부터 .NET Core에 제공된 버전 API가 이제 사용자가 예상하는 정보를 반환합니다. 예를 들면 다음과 같습니다.

C#

```
System.Console.WriteLine($"Environment.Version:
{System.Environment.Version}");

// Old result
//   Environment.Version: 4.0.30319.42000
//
// New result
//   Environment.Version: 3.0.0
```

C#

```
System.Console.WriteLine($"RuntimeInformation.FrameworkDescription:
{System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");

// Old result
//   RuntimeInformation.FrameworkDescription: .NET Core 4.6.27415.71
//
// New result (notice the value includes any preview release information)
//   RuntimeInformation.FrameworkDescription: .NET Core 3.0.0-preview4-
27615-11
```

> ⚠ **경고**
>
> 주요 변경 내용. 버전 관리 체계 변경되었기 때문에 엄밀히 따지면 주요 변경 내용입니다.

빠른 기본 제공 JSON 지원

.NET 사용자는 Newtonsoft.Json 및 여전히 유용한 기타 인기 있는 JSON 라이브러리를 많이 사용해 왔습니다. `Newtonsoft.Json`은 내부적으로 UTF-16인 .NET 문자열을 기본 데이터 형식으로 사용합니다.

새로운 기본 제공 JSON 지원은 고성능, 낮은 할당이며 UTF-8로 인코딩된 JSON 텍스트와 함께 작동합니다. System.Text.Json 네임스페이스 및 형식에 대한 자세한 내용은 다음 문서를 참조하세요.

- .NET의 JSON serialization - 개요
- .NET에서 JSON을 직렬화 및 역직렬화하는 방법
- Newtonsoft.json에서 System.Text.Json으로 마이그레이션하는 방법

HTTP/2 지원

System.Net.Http.HttpClient 형식은 HTTP/2 프로토콜을 지원합니다. HTTP/2를 사용할 수 있는 경우 TLS/ALPN을 통해 HTTP 프로토콜 버전이 협상되며, 서버에서 사용하기로 선택하면 HTTP/2가 사용됩니다.

기본 프로토콜은 여전히 HTTP/1.1이지만, 두 가지 방법으로 HTTP/2를 사용할 수 있습니다. 첫째, HTTP/2를 사용하도록 HTTP 요청 메시지를 설정할 수 있습니다.

C#

```
var client = new HttpClient() { BaseAddress = new
Uri("https://localhost:5001") };

// HTTP/1.1 request
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);

// HTTP/2 request
using (var request = new HttpRequestMessage(HttpMethod.Get, "/") { Version =
new Version(2, 0) })
using (var response = await client.SendAsync(request))
    Console.WriteLine(response.Content);
```

둘째, 기본적으로 HTTP/2를 사용하도록 HttpClient를 변경할 수 있습니다.

C#

```
var client = new HttpClient()
{
    BaseAddress = new Uri("https://localhost:5001"),
    DefaultRequestVersion = new Version(2, 0)
};

// HTTP/2 is default
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);
```

애플리케이션 개발 중에 암호화되지 않은 연결을 사용하려는 경우가 많습니다. 대상 엔드포인트에서 HTTP/2를 사용할 것을 알고 있으면, HTTP/2에 대해 암호화되지 않은 연결을 켤 수 있습니다. `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2UNENCRYPTEDSUPPORT` 환경 변수를 `1` 로 설정하거나 앱 컨텍스트에서 사용하도록 설정하면 됩니다.

C#

```
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
```

다음 단계

- .NET Core 2.2 및 3.0 간의 호환성이 손상되는 변경 검토
- Windows Forms 앱용 .NET Core 3.0의 주요 변경 사항을 검토합니다.

.NET Core 3.0의 호환성이 손상되는 변경

2025. 07. 02.

.NET Core, ASP.NET Core 또는 EF Core 버전 3.0으로 마이그레이션하는 경우 이 문서에 나열된 주요 변경 내용이 앱에 영향을 줄 수 있습니다.

ASP.NET Core

- 사용되지 않는 위조 방지, CORS, 진단, MVC 및 라우팅 API가 제거됨
- "Pubternal" API가 제거됨
- 인증: Google+ 사용 중단
- 인증: HttpContext.Authentication 속성이 제거됨
- 인증: Newtonsoft.Json 형식이 바뀜
- 인증: OAuthHandler ExchangeCodeAsync 서명이 변경됨
- 권한 부여: AddAuthorization 오버로드가 다른 어셈블리로 이동됨
- 권한 부여: AuthorizationFilterContext.Filters에서 IAllowAnonymous가 제거됨
- 권한 부여: IAuthorizationPolicyProvider 구현에는 새로운 메서드 필요
- 캐싱: CompactOnMemoryPressure 속성이 제거됨
- 캐싱: Microsoft.Extensions.Caching.SqlServer가 새로운 SqlClient 패키지를 사용함
- 캐싱: ResponseCaching "pubternal" 형식이 내부로 변경됨
- 데이터 보호: DataProtection.Blobs가 새로운 Azure Storage API를 사용함.
- 호스팅: Windows 호스팅 번들에서 AspNetCoreModule V1이 제거됨
- 호스팅: 제네릭 호스트가 시작 생성자 주입을 제한함.
- 호스팅: IIS 독립 프로세스 앱에 대해 HTTPS 리디렉션이 사용하도록 설정됨
- 호스팅: IHostingEnvironment 및 IApplicationLifetime 형식이 바뀜
- 호스팅: WebHostBuilder 종속성에서 ObjectPoolProvider가 제거됨
- HTTP: DefaultHttpContext 확장성이 제거됨
- HTTP: HeaderNames 필드가 정적 읽기 전용으로 변경됨
- HTTP: 응답 본문 인프라 변경
- HTTP: 일부 쿠키 SameSite 기본값이 변경됨
- HTTP: 동기식 IO는 기본적으로 사용하지 않도록 설정됨
- ID: AddDefaultUI 메서드 오버로드가 제거됨
- ID: UI 부트스트랩 버전 변경
- ID: 인증되지 않은 ID에 대해 SignInAsync에서 예외 발생
- ID: SignInManager 생성자가 새 매개 변수를 허용함
- ID: UI가 정적 웹 자산 기능을 사용함
- Kestrel: 연결 어댑터가 제거됨
- Kestrel: 빈 HTTPS 어셈블리가 제거됨
- Kestrel: 요청 후행부 헤더가 새 컬렉션으로 이동됨
- Kestrel: 전송 추상화 계층 변경

- 지역화: API가 사용되지 않음으로 표시됨
- 로깅: DebugLogger 클래스가 내부적으로 만들어짐
- MVC: 컨트롤러 작업 비동기 접미사가 제거됨
- MVC: JsonResult를 Microsoft.AspNetCore.Mvc.Core로 이동
- MVC: 사전 컴파일 도구가 사용되지 않음
- MVC: 형식이 내부로 변경됨
- MVC: 웹 API 호환성 shim이 제거됨
- Razor: RazorTemplateEngine API가 제거됨
- Razor: 런타임 컴파일이 패키지로 이동됨
- 세션 상태: 사용되지 않는 API가 제거됨
- 공유 프레임워크: Microsoft.AspNetCore.App에서 어셈블리 제거
- 공유 프레임워크: 제거된 Microsoft.AspNetCore.All이 제거됨
- SignalR: HandshakeProtocol.SuccessHandshakeData가 바뀜
- SignalR: HubConnection 메서드가 제거됨
- SignalR: HubConnectionContext 생성자가 변경됨
- SignalR: JavaScript 클라이언트 패키지 이름 변경
- SignalR: 사용되지 않는 API
- SPA: SpaServices 및 NodeServices가 사용되지 않음으로 표시됨
- SPAs: SpaServices 및 NodeServices 콘솔 로거 대체 기본 변경
- 대상 프레임워크: 지원되지 않는 .NET Framework

사용되지 않는 위조 방지, CORS, 진단, MVC 및 라우팅 API가 제거됨

ASP.NET Core 2.2에서 사용되지 않는 멤버 및 호환성 스위치가 제거되었습니다.

도입된 버전

3.0

변경 이유

시간에 따른 API 표면 개선.

권장 작업

.NET Core 2.2를 대상으로 하는 동안 사용되지 않는 빌드 메시지의 지침에 따라 새 API를 대신 채택합니다.

범주

영향을 받는 API

다음 형식 및 멤버는 ASP.NET Core 2.1 및 2.2에서 사용되지 않는 것으로 표시되었습니다.

유형

- `Microsoft.AspNetCore.Diagnostics.Views.WelcomePage`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.AttributeValue`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.BaseView`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.HelperResult`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.ProblemDetails21Wrapper`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.ValidationProblemDetails21Wrapper`
- `Microsoft.AspNetCore.Mvc.Razor.Compilation.ViewsFeatureProvider`
- `Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageArgumentBinder`
- `Microsoft.AspNetCore.Routing.IRouteValuesAddressMetadata`
- `Microsoft.AspNetCore.Routing.RouteValuesAddressMetadata`

생성자

- `Microsoft.AspNetCore.Cors.Infrastructure.CorsService(IOptions{CorsOptions})`
- `Microsoft.AspNetCore.Routing.Tree.TreeRouteBuilder(ILoggerFactory,UrlEncoder,ObjectPool{UriBuildingContext},IInlineConstraintResolver)`
- `Microsoft.AspNetCore.Mvc.Formatters.OutputFormatterCanWriteContext`
- `Microsoft.AspNetCore.Mvc.ApiExplorer.DefaultApiDescriptionProvider(IOptions{MvcOptions},IInlineConstraintResolver,IModelMetadataProvider)`
- `Microsoft.AspNetCore.Mvc.ApiExplorer.DefaultApiDescriptionProvider(IOptions{MvcOptions},IInlineConstraintResolver,IModelMetadataProvider,IActionResultTypeMapper)`
- `Microsoft.AspNetCore.Mvc.Formatters.FormatFilter(IOptions{MvcOptions})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ArrayModelBinder`1(IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ByteArrayModelBinder`
- Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CollectionModelBinder'1(IModelBinder)
- Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ComplexTypeModelBinder(IDictionary'2)
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DictionaryModelBinder`2(IModelBinder,IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DoubleModelBinder(System.Globalization.NumberStyles)`

- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FloatModelBinder(System.Globalization.NumberStyles)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormCollectionModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormFileModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.HeaderModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.KeyValuePairModelBinder`2(IModelBinder,IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.SimpleTypeModelBinder(System.Type)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ModelAttributes(IEnumerable{System.Object})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ModelAttributes(IEnumerable{System.Object},IEnumerable{System.Object})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ModelBinderFactory(IModelMetadataProvider,IOptions{MvcOptions})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder(IModelMetadataProvider,IModelBinderFactory,IObjectModelValidator)`
- Microsoft.AspNetCore.Mvc.Routing.KnownRouteValueConstraint()
- `Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter(System.Boolean)`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter(MvcOptions)`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter(System.Boolean)`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter(MvcOptions)`
- Microsoft.AspNetCore.Mvc.TagHelpers.ImageTagHelper(IHostingEnvironment,IMemoryCache,HtmlEncoder,IUrlHelperFactory)
- `Microsoft.AspNetCore.Mvc.TagHelpers.LinkTagHelper(IHostingEnvironment,IMemoryCache,HtmlEncoder,JavaScriptEncoder,IUrlHelperFactory)`
- `Microsoft.AspNetCore.Mvc.TagHelpers.ScriptTagHelper(IHostingEnvironment,IMemoryCache,HtmlEncoder,JavaScriptEncoder,IUrlHelperFactory)`
- `Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.RazorPageAdapter(RazorPageBase)`

속성

- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieDomain`
- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieName`
- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookiePath`
- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.RequireSsl`

- `Microsoft.AspNetCore.Mvc.ApiBehaviorOptions.AllowInferringBindingSourceForCollectionTypesAsFromQuery`
- `Microsoft.AspNetCore.Mvc.ApiBehaviorOptions.SuppressUseValidationProblemDetailsForInvalidModelStateResponses`
- `Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.CookieName`
- `Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.Domain`
- `Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.Path`
- `Microsoft.AspNetCore.Mvc.DataAnnotations.MvcDataAnnotationsLocalizationOptions.AllowDataAnnotationsLocalizationForEnumDisplayAttributes`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.MvcXmlOptions.AllowRfc7807CompliantProblemDetailsFormat`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowBindingHeaderValuesToNonStringModelTypes`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowCombiningAuthorizeFilters`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowShortCircuitingValidationWhenNoValidatorsArePresent`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowValidatingTopLevelNodes`
- `Microsoft.AspNetCore.Mvc.MvcOptions.InputFormatterExceptionPolicy`
- `Microsoft.AspNetCore.Mvc.MvcOptions.SuppressBindingUndefinedValueToEnumType`
- `Microsoft.AspNetCore.Mvc.MvcViewOptions.AllowRenderingMaxLengthAttribute`
- `Microsoft.AspNetCore.Mvc.MvcViewOptions.SuppressTempDataAttributePrefix`
- `Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowAreas`
- `Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowDefaultHandlingForOptionsRequests`
- `Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowMappingHeadRequestsToGetHandler`

메서드

- `Microsoft.AspNetCore.Mvc.LocalRedirectResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectToActionResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectToPageResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectToRouteResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder.BindModelAsync(ActionContext,IValueProvider,ParameterDescriptor)`
- Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder.BindModelAsync(ActionContext,IValueProvider,ParameterDescriptor,Object)

“Pubternal” API가 제거됨

ASP.NET Core의 공용 API 노출 영역을 보다 효율적으로 유지 관리하기 위해 `*.Internal` 네임스페이스("pubternal" API라고 함)에 있는 대부분의 형식이 진정한 내부용이 되었습니다. 이러한 네임스페이스의 멤버는 공용 API로 지원되지 않았습니다. 이러한 API는 마이너 릴리스에서 중단 가능성이 있었고 실제로 자주 중단되었습니다. ASP.NET Core 3.0으로 업데이트할 때 이러한 API에 종속된 코드가 중단됩니다.

자세한 내용은 dotnet/aspnetcore#4932 및 dotnet/aspnetcore#11312 를 참조하세요.

도입된 버전

3.0

이전 동작

영향을 받는 API가 `public` 액세스 한정자로 표시되고 `*.Internal` 네임스페이스에 존재합니다.

새 동작

영향을 받는 API가 internal 액세스 한정자로 표시되며, 해당 어셈블리 외부의 코드에서 이 API를 더 이상 사용할 수 없습니다.

변경 이유

이러한 "pubternal" API에 대한 지침은 다음과 같은 특성이 있었습니다.

- 예고 없이 변경될 수 있었습니다.
- 호환성이 손상되는 변경을 방지하기 위해 .NET 정책에 적용되지 않았습니다.

API를 `public`으로 유지하여(`*.Internal` 네임스페이스에서도) 고객이 혼동할 수 있었습니다.

권장 작업

이러한 "pubternal" API 사용을 중지합니다. 대체 API에 대한 질문이 있는 경우 dotnet/aspnetcore 리포지토리에서 문제를 여세요.

예를 들어 ASP.NET Core 2.2 프로젝트에서 다음 HTTP 요청 버퍼링 코드가 있습니다. `EnableRewind` 확장 메서드는 `Microsoft.AspNetCore.Http.Internal` 네임스페이스에 있습니다.

```
C#
```

```
HttpContext.Request.EnableRewind();
```

ASP.NET Core 3.0 프로젝트에서 `EnableRewind` 호출을 `EnableBuffering` 확장 메서드에 대한 호출로 바꿉니다. 요청 버퍼링 기능은 이전과 동일하게 작동합니다. `EnableBuffering`은 이제 `internal` API를 호출합니다.

C#

```
HttpContext.Request.EnableBuffering();
```

범주

ASP.NET Core

영향을 받는 API

네임 스페이스 이름에 `Microsoft.AspNetCore.*` 세그먼트가 있는 `Microsoft.Extensions.*` 및 `Internal` 네임스페이스의 모든 API. 예시:

- `Microsoft.AspNetCore.Authentication.Internal`
- `Microsoft.AspNetCore.Builder.Internal`
- `Microsoft.AspNetCore.DataProtection.Cng.Internal`
- `Microsoft.AspNetCore.DataProtection.Internal`
- `Microsoft.AspNetCore.Hosting.Internal`
- `Microsoft.AspNetCore.Http.Internal`
- `Microsoft.AspNetCore.Mvc.Core.Infrastructure`
- `Microsoft.AspNetCore.Mvc.Core.Internal`
- `Microsoft.AspNetCore.Mvc.Cors.Internal`
- `Microsoft.AspNetCore.Mvc.DataAnnotations.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Json.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.Internal`
- `Microsoft.AspNetCore.Mvc.Internal`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Internal`
- `Microsoft.AspNetCore.Mvc.Razor.Internal`
- `Microsoft.AspNetCore.Mvc.RazorPages.Internal`
- `Microsoft.AspNetCore.Mvc.TagHelpers.Internal`
- `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal`

- `Microsoft.AspNetCore.Rewrite.Internal`
- `Microsoft.AspNetCore.Routing.Internal`
- `Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal`
- `Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http`
- `Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Infrastructure`
- `Microsoft.AspNetCore.Server.Kestrel.Https.Internal`

인증: Google+는 사용되지 않고 바뀜

Google은 2019년 1월 28일부터 앱에 대한 Google+ 로그인을 종료 하기 시작했습니다.

변경 내용 설명

ASP.NET 4.x 및 ASP.NET Core는 Google+ 로그인 API를 사용하여 웹앱에서 Google 계정 사용자를 인증하고 있습니다. 영향을 받는 NuGet 패키지는 Microsoft.AspNetCore.Authentication.Google 이며 ASP.NET Web Forms 및 MVC가 있는 의 경우 `Microsoft.Owin`입니다.

Google의 대체 API는 다른 데이터 원본 및 형식을 사용합니다. 아래에 제공된 완화 및 솔루션은 구조적 변경을 제공합니다. 앱은 데이터 자체가 여전히 요구 사항을 충족하는지 확인해야 합니다. 예를 들어 이름, 이메일 주소, 프로필 링크 및 프로필 사진은 이전과는 약간 다른 값을 제공할 수 있습니다.

도입된 버전

모든 버전. 이 변경 내용은 ASP.NET Core 외부에 있습니다.

권장 작업

ASP.NET Web Forms 및 MVC를 사용한 Owin

`Microsoft.Owin` 3.1.0 이상에서는 Google+ 종료 영향에 임시 완화가 설명되어 있습니다. 앱은 데이터 형식의 변경 내용을 확인하기 위해 완화를 사용하여 테스트를 완료해야 합니다. 수정 내용이 있는 `Microsoft.Owin` 4.0.1을 릴리스할 계획이 있습니다. 이전 버전을 사용하는 앱은 버전 4.0.1로 업데이트해야 합니다.

ASP.NET Core 1.x

ASP.NET Web Forms 및 MVC를 사용한 Owin의 완화를 ASP.NET Core 1.x에 맞게 조정할 수 있습니다. 1.x가 수명 종료 상태에 도달했기 때문에 NuGet 패키지 패치가 계획되지 않았습니다.

ASP.NET Core 2.x

`Microsoft.AspNetCore.Authentication.Google` 버전 2.x의 경우 `AddGoogle`의 `Startup.ConfigureServices`에 대한 기존 호출을 다음 코드로 바꿉니다.

C#

```
.AddGoogle(o =>
{
    o.ClientId = Configuration["Authentication:Google:ClientId"];
    o.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    o.UserInformationEndpoint = "https://www.googleapis.com/oauth2/v2/userinfo";
    o.ClaimActions.Clear();
    o.ClaimActions.MapJsonKey(ClaimTypes.NameIdentifier, "id");
    o.ClaimActions.MapJsonKey(ClaimTypes.Name, "name");
    o.ClaimActions.MapJsonKey(ClaimTypes.GivenName, "given_name");
    o.ClaimActions.MapJsonKey(ClaimTypes.Surname, "family_name");
    o.ClaimActions.MapJsonKey("urn:google:profile", "link");
    o.ClaimActions.MapJsonKey(ClaimTypes.Email, "email");
});
```

2월 2.1 및 2.2 패치는 이전의 재구성을 새 기본값으로 통합했습니다. 수명 종료에 도달했기 때문에 ASP.NET Core 2.0의 패치가 계획되지 않았습니다.

ASP.NET Core 3.0

ASP.NET Core 2.x에 제공된 완화는 ASP.NET Core 3.0에도 사용할 수 있습니다. 향후 3.0 미리 보기에서 `Microsoft.AspNetCore.Authentication.Google` 패키지가 제거될 수 있습니다. 사용자는 대신 `Microsoft.AspNetCore.Authentication.OpenIdConnect`로 이동해야 합니다. 다음 코드는 `AddGoogle`에서 `AddOpenIdConnect`을 `Startup.ConfigureServices`로 바꾸는 방법을 보여줍니다. 이 대체 항목은 ASP.NET Core 2.0 이상에서 사용할 수 있으며, 필요에 따라 ASP.NET Core 1.x에 맞게 조정할 수 있습니다.

C#

```
.AddOpenIdConnect("Google", o =>
{
    o.ClientId = Configuration["Authentication:Google:ClientId"];
    o.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    o.Authority = "https://accounts.google.com";
    o.ResponseType = OpenIdConnectResponseType.Code;
    o.CallbackPath = "/signin-google"; // Or register the default "/signin-oidc"
    o.Scope.Add("email");
```

```
});
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
```

범주

ASP.NET Core

영향을 받는 API

Microsoft.AspNetCore.Authentication.Google

인증: HttpContext.Authentication 속성이 제거됨

`Authentication`에서 사용되지 않는 `HttpContext` 속성이 제거되었습니다.

변경 내용 설명

dotnet/aspnetcore#6504 의 일부로, `Authentication`에서 사용되지 않는 `HttpContext` 속성이 제거되었습니다. `Authentication` 속성은 2.0 이후 더 이상 사용되지 않습니다. 사용되지 않는 이 속성을 사용하여 코드를 새 대체 API로 마이그레이션하기 위해 마이그레이션 가이드가 게시되었습니다. 이전 ASP.NET Core 1.x 인증 스택과 관련된 나머지 사용되지 않는 클래스/API는 커밋 dotnet/aspnetcore@d7a7c65 에서 제거되었습니다.

토론은 dotnet/aspnetcore#6533 을 참조하세요.

도입된 버전

3.0

변경 이유

ASP.NET Core 1.0 API는 Microsoft.AspNetCore.Authentication.AuthenticationHttpContextExtensions의 확장 메서드로 대체되었습니다.

권장 작업

마이그레이션 가이드를 참조하세요.

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Http.Authentication.AuthenticateInfo
- Microsoft.AspNetCore.Http.Authentication.AuthenticationManager
- Microsoft.AspNetCore.Http.Authentication.AuthenticationProperties
- Microsoft.AspNetCore.Http.Features.Authentication.AuthenticateContext
- Microsoft.AspNetCore.Http.Features.Authentication.ChallengeBehavior
- Microsoft.AspNetCore.Http.Features.Authentication.ChallengeContext
- Microsoft.AspNetCore.Http.Features.Authentication.DescribeSchemesContext
- Microsoft.AspNetCore.Http.Features.Authentication.IAuthenticationHandler
- Microsoft.AspNetCore.Http.Features.Authentication.IHttpAuthenticationFeature.Handler
- Microsoft.AspNetCore.Http.Features.Authentication.SignInContext
- Microsoft.AspNetCore.Http.Features.Authentication.SignOutContext
- Microsoft.AspNetCore.Http.HttpContext.Authentication

인증: Newtonsoft.Json 형식이 바뀜

ASP.NET Core 3.0에서는 인증 API에 사용된 `Newtonsoft.Json` 형식이 `System.Text.Json` 형식으로 대체되었습니다. 다음 경우를 제외하고 인증 패키지의 기본 사용량은 영향을 받지 않습니다.

- OAuth 공급자에서 파생된 클래스(예: aspnet-contribaspnet 의 클래스)입니다.
- 고급 클레임 조작 구현.

자세한 내용은 dotnet/aspnetcore#7105 를 참조하세요. 토론은 dotnet/aspnetcore#7289 를 참조하세요.

도입된 버전

3.0

권장 작업

파생 OAuth 구현의 경우, 재정의에서 `JObject.Parse`를 `JsonDocument.Parse`로 바꾸어야 하는 것이 가장 일반적인 변경 사항입니다. 이는 `CreateTicketAsync`에 나와 있습니다. `JsonDocument`는 `IDisposable`를 구현합니다.

다음 목록에서는 알려진 변경 내용을 간략하게 설명합니다.

- ClaimAction.Run(JObject, ClaimsIdentity, String)은 `ClaimAction.Run(JsonElement userData, ClaimsIdentity identity, string issuer)`가 됩니다. `ClaimAction`의 모든 파생 구현은 이와 유사하게 영향을 받습니다.
- ClaimActionCollectionMapExtensions.MapCustomJson(ClaimActionCollection, String, Func<JObject,String>)은 `MapCustomJson(this ClaimActionCollection collection, string claimType, Func<JsonElement, string> resolver)`이 됨
- ClaimActionCollectionMapExtensions.MapCustomJson(ClaimActionCollection, String, String, Func<JObject,String>)은 `MapCustomJson(this ClaimActionCollection collection, string claimType, string valueType, Func<JsonElement, string> resolver)`이 됨
- OAuthCreatingTicketContext에는 하나의 이전 생성자를 제거했으며 다른 생성자는 `JObject`를 `JsonElement`로 대체했습니다. `User` 속성과 `RunClaimActions` 메서드가 일치하도록 업데이트되었습니다.
- 이제 Success(JObject)는 `JsonDocument` 대신 `JObject` 형식의 매개 변수를 허용합니다. `Response` 속성이 일치하도록 업데이트되었습니다. 이제 `OAuthTokenResponse`는 삭제 가능하며 `OAuthHandler`에 의해 삭제됩니다. `ExchangeCodeAsync`를 재정의하는 파생 OAuth 구현은 `JsonDocument` 또는 `OAuthTokenResponse`를 삭제할 필요가 없습니다.
- UserInformationReceivedContext.User이 `JObject`에서 `JsonDocument`로 변경되었습니다.
- TwitterCreatingTicketContext.User이 `JObject`에서 `JsonElement`로 변경되었습니다.
- TwitterHandler.CreateTicketAsync(ClaimsIdentity,AuthenticationProperties,AccessToken,JObject)의 마지막 매개 변수가 `JObject`에서 `JsonElement`로 변경되었습니다. 대체 메서드는 TwitterHandler.CreateTicketAsync(ClaimsIdentity, AuthenticationProperties, AccessToken, JsonElement)입니다.

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Authentication.Facebook
- Microsoft.AspNetCore.Authentication.Google
- Microsoft.AspNetCore.Authentication.MicrosoftAccount
- Microsoft.AspNetCore.Authentication.OAuth
- Microsoft.AspNetCore.Authentication.OpenIdConnect
- Microsoft.AspNetCore.Authentication.Twitter

인증: OAuthHandler ExchangeCodeAsync 서명이 변경됨

ASP.NET Core 3.0에서 `OAuthHandler.ExchangeCodeAsync`의 서명이 다음에서 변경되었습니다.

C#

```
protected virtual
System.Threading.Tasks.Task<Microsoft.AspNetCore.Authentication.OAuth.OAuthTokenRe
sponse> ExchangeCodeAsync(string code, string redirectUri) { throw null; }
```

다음으로 변경:

C#

```
protected virtual
System.Threading.Tasks.Task<Microsoft.AspNetCore.Authentication.OAuth.OAuthTokenRe
sponse>
ExchangeCodeAsync(Microsoft.AspNetCore.Authentication.OAuth.OAuthCodeExchangeConte
xt context) { throw null; }
```

도입된 버전

3.0

이전 동작

`code` 및 `redirectUri` 문자열은 별도의 인수로 전달되었습니다.

새 동작

`Code` 및 `RedirectUri`는 `OAuthCodeExchangeContext` 생성자를 통해 설정할 수 있는 `OAuthCodeExchangeContext`의 속성입니다. 새 `OAuthCodeExchangeContext` 형식은 `OAuthHandler.ExchangeCodeAsync`에 전달된 유일한 인수입니다.

변경 이유

이러한 변경으로 인해 추가 매개 변수가 중단되지 않은 방식으로 제공될 수 있습니다. 새 `ExchangeCodeAsync` 오버로드를 만들 필요가 없습니다.

권장 작업

적절한 `OAuthCodeExchangeContext` 및 `code` 값을 사용하여 `redirectUri`를 구성합니다. AuthenticationProperties 인스턴스를 제공해야 합니다. 이 단일 `OAuthCodeExchangeContext` 인스턴스는 여러 인수 대신 `OAuthHandler.ExchangeCodeAsync`에 전달할 수 있습니다.

범주

ASP.NET Core

영향을 받는 API

OAuthHandler<TOptions>.ExchangeCodeAsync(String, String)

권한 부여: AddAuthorization 오버로드가 다른 어셈블리로 이동됨

`AddAuthorization`에 상주하는 데 사용되는 핵심 `Microsoft.AspNetCore.Authorization` 메서드가 `AddAuthorizationCore`로 이름이 변경되었습니다. 이전 `AddAuthorization` 메서드는 여전히 존재하지만, 대신 `Microsoft.AspNetCore.Authorization.Policy` 어셈블리에 있습니다. 두 방법을 모두 사용하는 앱은 영향을 받지 않아야 합니다. `Microsoft.AspNetCore.Authorization.Policy`는 이제 공유 프레임워크: Microsoft.AspNetCore.App에서 제거된 어셈블리에 설명된 대로 독립 실행형 패키지가 아닌 공유 프레임워크로 제공됩니다.

도입된 버전

3.0

이전 동작

`AddAuthorization` 메서드가 `Microsoft.AspNetCore.Authorization`에 있었습니다.

새 동작

`AddAuthorization` 메서드가 `Microsoft.AspNetCore.Authorization.Policy`에 있습니다. `AddAuthorizationCore`는 이전 메서드의 새 이름입니다.

변경 이유

`AddAuthorization`는 권한 부여에 필요한 모든 일반 서비스를 추가하는 데 더 나은 메서드 이름입니다.

권장 작업

`Microsoft.AspNetCore.Authorization.Policy`에 대한 참조를 추가하거나 대신 `AddAuthorizationCore`를 사용합니다.

범주

ASP.NET Core

영향을 받는 API

Microsoft.Extensions.DependencyInjection.AuthorizationServiceCollectionExtensions.AddAuthorization(IServiceCollection, Action<AuthorizationOptions>)

권한 부여: AuthorizationFilterContext.Filters에서 IAllowAnonymous가 제거됨

ASP.NET Core 3.0을 기준으로 MVC에서 컨트롤러 및 작업 메서드에서 검색된 [AllowAnonymous] 특성에 대해 AllowAnonymousFilters를 추가하지 않습니다. 이 변경 사항은 AuthorizeAttribute의 파생에 대해 로컬로 처리하지만 IAsyncAuthorizationFilter 및 IAuthorizationFilter 구현에 대해 호환성이 손상됩니다. [TypeFilter] 특성에 래핑된 이러한 구현은 구성 및 종속성 주입이 모두 필요한 경우 강력한 형식의 특성 기반 권한 부여를 달성하는 일반적이고 지원되는 방식입니다.

도입된 버전

3.0

이전 동작

IAllowAnonymous가 AuthorizationFilterContext.Filters 컬렉션에 표시되었습니다. 인터페이스의 현재 상태에 대한 테스트는 개별 컨트롤러 메서드에서 필터를 재정의하거나 사용하지 않도록 설정하는 유효한 방법이었습니다.

새 동작

`IAllowAnonymous` 컬렉션에 `AuthorizationFilterContext.Filters`가 더 이상 표시되지 않습니다. 이전 동작에 의존하는 `IAsyncAuthorizationFilter` 구현은 일반적으로 일시적인 HTTP 401 Unauthorized 또는 HTTP 403 Forbidden 응답의 원인이 됩니다.

변경 이유

새 엔드포인트 라우팅 전략은 ASP.NET Core 3.0에서 도입되었습니다.

권장 작업

`IAllowAnonymous`에 대한 엔드포인트 메타데이터를 검색합니다. 예시:

C#

```
var endpoint = context.HttpContext.GetEndpoint();
if (endpoint?.Metadata?.GetMetadata<IAllowAnonymous>() != null)
{
}
```

이 기법에 대한 예는 HasAllowAnonymous 메서드에 표시됩니다.

범주

ASP.NET Core

영향을 받는 API

없음

권한 부여: IAuthorizationPolicyProvider 구현에는 새로운 메서드가 필요함

ASP.NET Core 3.0에서는 새 `GetFallbackPolicyAsync` 메서드가 `IAuthorizationPolicyProvider`에 추가되었습니다. 이 대체 정책은 정책이 지정되지 않은 경우 권한 부여 미들웨어에서 사용됩니다.

자세한 내용은 dotnet/aspnetcore#9759를 참조하세요.

도입된 버전

3.0

이전 동작

`IAuthorizationPolicyProvider`의 구현에는 `GetFallbackPolicyAsync` 메서드가 필요하지 않습니다.

새 동작

`IAuthorizationPolicyProvider`의 구현에는 `GetFallbackPolicyAsync` 메서드가 필요합니다.

변경 이유

정책이 지정되지 않은 경우 새 `AuthorizationMiddleware`를 사용할 수 있는 새로운 메서드가 필요했습니다.

권장 작업

`GetFallbackPolicyAsync`의 구현에 `IAuthorizationPolicyProvider` 메서드를 추가합니다.

범주

ASP.NET Core

영향을 받는 API

Microsoft.AspNetCore.Authorization.IAuthorizationPolicyProvider

---

캐싱: CompactOnMemoryPressure 속성이 제거됨

ASP.NET Core 3.0 릴리스에서는 사용되지 않는 MemoryCacheOptions API를 제거했습니다.

변경 내용 설명

이 변경 내용은 aspnet/Caching#221에 대한 후속 작업입니다. 자세한 내용은 dotnet/extensions#1062를 참조하세요.

도입된 버전

3.0

이전 동작

`MemoryCacheOptions.CompactOnMemoryPressure` 속성을 사용할 수 있었습니다.

새 동작

`MemoryCacheOptions.CompactOnMemoryPressure` 속성이 제거되었습니다.

변경 이유

자동으로 캐시를 압축하면 문제가 발생합니다. 예기치 않은 동작을 방지하려면 필요한 경우에만 캐시를 압축해야 합니다.

권장 작업

캐시를 압축하려면 `MemoryCache`를 다운캐스트하고 필요한 경우 `Compact`를 호출합니다.

범주

ASP.NET Core

영향을 받는 API

MemoryCacheOptions.CompactOnMemoryPressure

캐싱: Microsoft.Extensions.Caching.SqlServer가 새로운 SqlClient 패키지를 사용함

`Microsoft.Extensions.Caching.SqlServer` 패키지는 `Microsoft.Data.SqlClient` 패키지 대신 새 `System.Data.SqlClient` 패키지를 사용합니다. 이 변경으로 인해 약간의 동작이 크게 변경될 수 있습니다. 자세한 내용은 새 Microsoft.Data.SqlClient 소개를 참조하세요.

도입된 버전

3.0

이전 동작

`Microsoft.Extensions.Caching.SqlServer` 패키지는 `System.Data.SqlClient` 패키지를 사용했습니다.

새 동작

`Microsoft.Extensions.Caching.SqlServer`가 현재 `Microsoft.Data.SqlClient` 패키지를 사용하고 있습니다.

변경 이유

`Microsoft.Data.SqlClient`는 `System.Data.SqlClient`에서 빌드된 새 패키지입니다. 이제부터 모든 새 기능 작업이 수행됩니다.

권장 작업

고객은 `Microsoft.Extensions.Caching.SqlServer` 패키지에서 반환된 형식을 사용하고 `System.Data.SqlClient` 형식으로 캐스팅하지 않는 한 이러한 호환성이 손상되는 변경에 대해 걱정할 필요가 없습니다. 예를 들어 다른 사용자가 `DbConnection`을 이전 SqlConnection 형식으로 캐스팅하는 경우 캐스팅을 새 `Microsoft.Data.SqlClient.SqlConnection` 형식으로 변경해야 합니다.

범주

ASP.NET Core

영향을 받는 API

없음

캐싱: ResponseCaching "pubternal" 형식이 내부로 변경됨

ASP.NET Core 3.0에서는 `ResponseCaching`의 "pubternal" 유형이 `internal`로 변경되었습니다.

또한 `IResponseCachingPolicyProvider` 및 `IResponseCachingKeyProvider`의 기본 구현은 더 이상 `AddResponseCaching` 메서드의 일부로 서비스에 추가되지 않습니다.

변경 내용 설명

ASP.NET Core에서 "pubternal" 유형은 `public`으로 선언되지만 `.Internal`로 접미사가 지정된 네임스페이스에 있습니다. 이러한 형식은 공용이지만 지원 정책이 없으며 호환성이 손상되는 변경이 적용됩니다. 아쉽게도 이러한 형식의 우발적 사용이 일반적으로 발생하므로 이러한 프로젝트에 대한 호환성이 손상되는 변경이 발생하고 프레임워크를 유지 관리하는 기능이 제한됩니다.

도입된 버전

3.0

이전 동작

이러한 형식은 공개적으로 표시되지만 지원되지는 않습니다.

새 동작

이러한 유형은 이제는 `internal`입니다.

변경 이유

`internal` 범위는 지원되지 않는 정책을 더 잘 반영합니다.

권장 작업

앱 또는 라이브러리에서 사용하는 형식을 복사합니다.

범주

ASP.NET Core

영향을 받는 API

- `Microsoft.AspNetCore.ResponseCaching.Internal.CachedResponse`
- `Microsoft.AspNetCore.ResponseCaching.Internal.CachedVaryByRules`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCache`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCacheEntry`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCachingKeyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCachingPolicyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.MemoryResponseCache`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingContext`

- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingKeyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingPolicyProvider`
- Microsoft.AspNetCore.ResponseCaching.ResponseCachingMiddleware.ResponseCachingMiddleware(RequestDelegate, IOptions<ResponseCachingOptions>, ILoggerFactory, IResponseCachingPolicyProvider, IResponseCache, IResponseCachingKeyProvider)

데이터 보호: DataProtection.Blobs가 새로운 Azure Storage API를 사용함

`Azure.Extensions.AspNetCore.DataProtection.Blobs`은 Azure Storage 라이브러리에 따라 달라집니다. 이러한 라이브러리는 해당 어셈블리, 패키지 및 네임 스페이스의 이름을 바꿨습니다. ASP.NET Core 3.0부터 `Azure.Extensions.AspNetCore.DataProtection.Blobs`는 새 `Azure.Storage.` 접두사가 지정된 API 및 패키지를 사용합니다.

Azure Storage API에 대한 질문은 https://github.com/Azure/azure-storage-net을 사용하세요. 이 문제에 대한 자세한 내용은 dotnet/aspnetcore#19570을 참조하세요.

도입된 버전

3.0

이전 동작

패키지는 `WindowsAzure.Storage` NuGet 패키지를 참조했습니다. 패키지는 `Microsoft.Azure.Storage.Blob` NuGet 패키지를 참조합니다.

새 동작

패키지는 `Azure.Storage.Blob` NuGet 패키지를 참조합니다.

변경 이유

이 변경으로 인해 `Azure.Extensions.AspNetCore.DataProtection.Blobs`는 권장 Azure Storage 패키지로 마이그레이션할 수 있습니다.

권장 작업

ASP.NET Core 3.0과 함께 이전 Azure Storage API를 계속 사용해야 하는 경우 WindowsAzure.Storage 또는 Microsoft.Azure.Storage 패키지에 직접 종속성을 추가합니다.

이 패키지는 새 `Azure.Storage` API와 함께 설치할 수 있습니다.

대부분의 경우 업그레이드는 새 네임스페이스를 사용하도록 `using` 문만 변경하면 됩니다.

diff

```
- using Microsoft.WindowsAzure.Storage;
- using Microsoft.WindowsAzure.Storage.Blob;
- using Microsoft.Azure.Storage;
- using Microsoft.Azure.Storage.Blob;
+ using Azure.Storage;
+ using Azure.Storage.Blobs;
```

범주

ASP.NET Core

영향을 받는 API

없음

호스팅: AspNetCoreModule V1이 Windows 호스팅 번들에서 제거됨

ASP.NET Core 3.0부터 Windows Hosting Bundle에는 ANCM(AspNetCoreModule) V1이 포함되어 있지 않습니다.

ANCM V2는 ANCM OutOfProcess와 역호환되며 ASP.NET Core 3.0 앱에서 사용하는 것이 좋습니다.

토론은 dotnet/aspnetcore#7095를 참조하세요.

도입된 버전

3.0

이전 동작

ANCM V1은 Windows Hosting Bundle에 포함되어 있습니다.

새 동작

ANCM V1은 Windows Hosting Bundle에 포함되어 있지 않습니다.

변경 이유

ANCM V2는 ANCM OutOfProcess와 역호환되며 ASP.NET Core 3.0 앱에서 사용하는 것이 좋습니다.

권장 작업

ASP.NET Core 3.0 앱에서 ANCM V2를 사용합니다.

ANCM V1이 필요한 경우 ASP.NET Core 2.1 또는 2.2 Windows Hosting Bundle을 사용하여 설치할 수 있습니다.

이 변경으로 인해 다음과 같은 ASP.NET Core 3.0 앱이 중단됩니다.

- `<AspNetCoreModuleName>AspNetCoreModule</AspNetCoreModuleName>`에서 ANCM V1을 사용하도록 명시적으로 선택했습니다.
- 를 사용하여 사용자 지정 `<add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />` 파일을 만듭니다.

범주

ASP.NET Core

영향을 받는 API

없음

호스팅: 제네릭 호스트가 시작 생성자 주입을 제한함

제네릭 호스트가 `Startup` 클래스 생성자 주입을 지원하는 형식은 `IHostEnvironment`, `IWebHostEnvironment` 및 `IConfiguration`뿐입니다. `WebHost`를 사용하는 앱은 영향을 받지 않습니다.

변경 내용 설명

ASP.NET Core 3.0 이전에는 `Startup` 클래스의 생성자에 있는 임의 형식에 생성자 주입을 사용할 수 있습니다. ASP.NET Core 3.0에서는 웹 스택이 제네릭 호스트 라이브러리로 다시 플랫폼화되었습니다. 템플릿의 *Program.cs* 파일에서 변경 내용을 확인할 수 있습니다.

ASP.NET Core 2.x:

https://github.com/dotnet/aspnetcore/blob/5cb615fcbe8559e49042e93394008077e30454c0/src/Templating/src/Microsoft.DotNet.Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L20-L22

ASP.NET Core 3.0:

https://github.com/dotnet/aspnetcore/blob/b1ca2c1155da3920f0df5108b9fedbe82efaa11c/src/ProjectTemplates/Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L19-L24

`Host`는 하나의 DI(종속성 주입) 컨테이너를 사용하여 앱을 빌드합니다. `WebHost`는 두 개의 컨테이너(호스트용과 앱용)를 사용합니다. 따라서 `Startup` 생성자는 더 이상 사용자 지정 서비스 주입을 지원하지 않습니다. `IHostEnvironment`, `IWebHostEnvironment` 및 `IConfiguration`만 주입할 수 있습니다. 이 변경으로 인해 싱글톤 서비스의 중복 만들기와 같은 DI 문제가 방지됩니다.

도입된 버전

3.0

변경 이유

이 변경 내용은 웹 스택을 제네릭 호스트 라이브러리로 다시 플랫폼화한 결과입니다.

권장 작업

`Startup.Configure` 메서드 서명에 서비스를 주입합니다. 예시:

```
C#
public void Configure(IApplicationBuilder app, IOptions<MyOptions> options)
```

범주

ASP.NET Core

영향을 받는 API

없음

호스팅: IIS Out-of-Process 앱에 대해 HTTPS 리디렉션이 사용하도록 설정됨

IIS Out of Process를 통해 호스팅하기 위한 ASP.NET Core 모듈(ANCM)의 버전 13.0.19218.0은 ASP.NET Core 3.0 및 2.2 앱에 대한 기존 HTTPS 리디렉션 기능을 사용하도록 설정합니다.

토론은 dotnet/AspNetCore#15243 을 참조하세요.

도입된 버전

3.0

이전 동작

ASP.NET Core 2.1 프로젝트 템플릿은 먼저 UseHttpsRedirection 및 UseHsts와 같은 HTTPS 미들웨어 메서드에 대한 지원을 도입했습니다. 개발 중인 앱은 기본 포트 443을 사용하지 않으므로 HTTPS 리디렉션을 사용하도록 설정하려면 구성을 추가해야 합니다. HSTS(HTTP Strict Transport Security) 는 요청이 HTTPS를 이미 사용하고 있을 경우에만 활성화됩니다. localhost는 기본적으로 생략됩니다.

새 동작

ASP.NET Core 3.0에서 IIS HTTPS 시나리오는 향상 되었습니다. 향상된 기능을 사용하면 앱이 서버의 HTTPS 포트를 검색하고 기본적으로 `UseHttpsRedirection` 작업을 수행할 수 있습니다. In Process 구성 요소는 `IServerAddresses` 기능을 사용하여 포트 검색을 완료했습니다. In Process 라이브러리가 프레임워크를 통해 버전이 지정되었기 때문에 이 기능은 ASP.NET Core 3.0 앱에만 영향을 줍니다. `ASPNETCORE_HTTPS_PORT` 환경 변수를 자동으로 추가하도록 Out of Process 구성 요소가 변경되었습니다. Out of Process 구성 요소가 전역적으로 공유되기 때문에 이 변경은 ASP.NET Core 2.2 및 3.0 앱 모두에 영향을 미쳤습니다. ASP.NET Core 2.1 앱은 기본적으로 이전 버전의 버전을 ANCM을 사용하므로 영향을 받지 않습니다.

ASP.NET Core 3.0.1과 3.1.0 Preview 3에서 이전 동작을 수정하여 ASP.NET Core 2.x의 동작 변경을 되돌립니다. 해당 변경 내용은 IIS Out of Process 앱에만 영향을 줍니다.

위에서 설명한 대로 ASP.NET Core 3.0.0을 설치하면 ASP.NET Core 2.x 앱에서 `UseHttpsRedirection` 미들웨어도 활성화되는 부작용이 있었습니다. ASP.NET Core 3.0.1 및 3.1.0 Preview 3을 설치해도 ASP.NET Core 2.x 앱에 더 이상 영향을 주지 않도록 변경되었습니다. ASP.NET Corea 3.0.0에서 ANCM이 채운 `ASPNETCORE_HTTPS_PORT` 환경 변수가 ASP.NET Core 3.0.1 및 3.1.0 Preview 3의 `ASPNETCORE_ANCM_HTTPS_PORT`로 변경되었습니다. `UseHttpsRedirection`는 새로운 변수와 이전 변수를 모두 이해하기 위해 이 릴리스에서도 업데이트되었습니다.

ASP.NET Core 2.x는 업데이트되지 않습니다. 그 결과, 기본적으로 사용하지 않도록 설정된 이전 동작으로 돌아갑니다.

변경 이유

ASP.NET Core 3.0 기능이 향상되었습니다.

권장 작업

모든 클라이언트가 HTTPS를 사용하도록 하려면 작업이 필요 없습니다. 일부 클라이언트가 HTTPS를 사용하도록 허용하려면 다음 단계 중 하나를 수행합니다.

- 프로젝트의 `UseHttpsRedirection` 메서드에서 `UseHsts` 및 `Startup.Configure`에 대한 호출을 제거하고 앱을 다시 배포하세요.
- *web.config* 파일에서 `ASPNETCORE_HTTPS_PORT` 환경 변수를 빈 문자열로 설정하세요. 변경은 앱을 다시 배포하지 않고 서버에서 직접 발생할 수 있습니다. 예시:

  XML

  ```
  <aspNetCore processPath="dotnet" arguments=".\WebApplication3.dll"
  stdoutLogEnabled="false" stdoutLogFile="\\?\%home%\LogFiles\stdout" >
      <environmentVariables>
      <environmentVariable name="ASPNETCORE_HTTPS_PORT" value="" />
      </environmentVariables>
  </aspNetCore>
  ```

`UseHttpsRedirection`은 여전히 다음이 될 수 있습니다.

- `ASPNETCORE_HTTPS_PORT` 환경 변수를 적절한 포트 번호(대부분의 프로덕션 시나리오에서 443)로 설정하여 ASP.NET Core 2.x에서 수동으로 활성화했습니다.
- 빈 문자열 값으로 `ASPNETCORE_ANCM_HTTPS_PORT`를 정의하여 ASP.NET Core 3.x에서 비활성화했습니다. 이 값은 앞의 `ASPNETCORE_HTTPS_PORT` 예제와 동일한 방식으로 설정됩니다.

ASP.NET Core 3.0.0 앱을 실행하는 컴퓨터는 ASP.NET Core 3.1.0 Preview 3 ANCM을 설치하기 전에 ASP.NET Core 3.0.1 런타임을 설치해야 합니다. 이렇게 하면 `UseHttpsRedirection`이 ASP.NET Core 3.0 앱에 대해 예상대로 작동합니다.

Azure App Service에서는 ANCM이 전역 특성 때문에 런타임과 별도의 일정으로 배포됩니다. ANCM은 ASP.NET Core 3.0.1 및 3.1.0가 배포된 후 해당 변경과 함께 Azure에 배포되었습니다.

범주

영향을 받는 API

HttpsPolicyBuilderExtensions.UseHttpsRedirection(IApplicationBuilder)

호스팅: IHostingEnvironment 및 IApplicationLifetime 형식이 사용되지 않음으로 표시되고 바뀜

기존 `IHostingEnvironment` 및 `IApplicationLifetime` 형식을 대체하기 위해 새로운 형식이 도입되었습니다.

도입된 버전

3.0

이전 동작

`IHostingEnvironment` 및 `IApplicationLifetime`과는 다른 두 가지 (`Microsoft.Extensions.Hosting` 및 `Microsoft.AspNetCore.Hosting`) 형식이 있습니다.

새 동작

이전 형식은 사용되지 않음으로 표시되었으며 새 형식으로 대체되었습니다.

변경 이유

ASP.NET Core 2.1에 `Microsoft.Extensions.Hosting`이 도입되었을 때 `IHostingEnvironment` 및 `IApplicationLifetime`과 같은 일부 형식이 `Microsoft.AspNetCore.Hosting`에서 복사되었습니다. 일부 ASP.NET Core 3.0 변경으로 인해 앱에 `Microsoft.Extensions.Hosting` 및 `Microsoft.AspNetCore.Hosting` 네임스페이스가 모두 포함됩니다. 이러한 중복 형식을 사용하면 네임스페이스가 모두 참조될 때 "모호한 참조" 컴파일러 오류가 발생합니다.

권장 작업

이전 형식의 사용을 다음과 같이 새로 도입된 형식으로 대체했습니다.

사용되지 않는 형식(경고):

- Microsoft.Extensions.Hosting.IHostingEnvironment
- Microsoft.AspNetCore.Hosting.IHostingEnvironment
- Microsoft.Extensions.Hosting.IApplicationLifetime
- Microsoft.AspNetCore.Hosting.IApplicationLifetime
- Microsoft.Extensions.Hosting.EnvironmentName
- Microsoft.AspNetCore.Hosting.EnvironmentName

새 형식:

- Microsoft.Extensions.Hosting.IHostEnvironment
- `Microsoft.AspNetCore.Hosting.IWebHostEnvironment : IHostEnvironment`
- Microsoft.Extensions.Hosting.IHostApplicationLifetime
- Microsoft.Extensions.Hosting.Environments

새 `IHostEnvironment` `IsDevelopment` 및 `IsProduction` 확장 메서드는 `Microsoft.Extensions.Hosting` 네임스페이스에 있습니다. 해당 네임스페이스를 프로젝트에 추가해야 할 수도 있습니다.

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Hosting.EnvironmentName
- Microsoft.AspNetCore.Hosting.IApplicationLifetime
- Microsoft.AspNetCore.Hosting.IHostingEnvironment
- Microsoft.Extensions.Hosting.EnvironmentName
- Microsoft.Extensions.Hosting.IApplicationLifetime
- Microsoft.Extensions.Hosting.IHostingEnvironment

호스팅: WebHostBuilder 종속성에서 ObjectPoolProvider가 제거됨

ASP.NET Core가 재생에 대해 추가 비용을 지불하도록 하는 과정에서 `ObjectPoolProvider`가 기본 종속성 세트에서 제거되었습니다. `ObjectPoolProvider`에 의존하는 특정 구성 요소가 이제 해당 구성 요소를 추가합니다.

토론은 dotnet/aspnetcore#5944를 참조하세요.

도입된 버전

3.0

이전 동작

`WebHostBuilder`에서는 기본적으로 DI 컨테이너에 `ObjectPoolProvider`를 제공합니다.

새 동작

`WebHostBuilder`에서는 더 이상 기본적으로 DI 컨테이너에 `ObjectPoolProvider`를 제공하지 않습니다.

변경 이유

이러한 변경으로 인해 ASP.NET Core가 더 많은 비용을 지불하게 되었습니다.

권장 작업

구성 요소에 `ObjectPoolProvider`가 필요한 경우 `IServiceCollection`을 통해 종속성에 추가해야 합니다.

범주

ASP.NET Core

영향을 받는 API

없음

HTTP: DefaultHttpContext 확장성이 제거됨

ASP.NET Core 3.0 성능 향상의 일환으로 `DefaultHttpContext`의 확장성이 제거되었습니다. 클래스는 이제 `sealed`입니다. 자세한 내용은 dotnet/aspnetcore#6504를 참조하세요.

단위 테스트에 `Mock<DefaultHttpContext>`가 사용되는 경우에는 대신 `Mock<HttpContext>` 또는 `new DefaultHttpContext()`를 사용합니다.

토론은 dotnet/aspnetcore#6534를 참조하세요.

도입된 버전

3.0

이전 동작

클래스는 `DefaultHttpContext`에서 파생될 수 있습니다.

새 동작

클래스는 `DefaultHttpContext`에서 파생될 수 없습니다.

변경 이유

확장성은 처음에는 `HttpContext`의 풀링을 허용하기 위해 제공되었지만 불필요 한 복잡성을 야기하고 기타 최적화를 방해했습니다.

권장 작업

단위 테스트에서 `Mock<DefaultHttpContext>`를 사용하는 경우, 대신 `Mock<HttpContext>`를 사용합니다.

범주

ASP.NET Core

영향을 받는 API

Microsoft.AspNetCore.Http.DefaultHttpContext

HTTP: HeaderNames 상수가 정적 읽기 전용으로 변경됨

ASP.NET Core 3.0 Preview 5부터 Microsoft.Net.Http.Headers.HeaderNames의 필드가 `const`에서 `static readonly`로 변경되었습니다.

토론은 dotnet/aspnetcore#9514를 참조하세요.

도입된 버전

3.0

이전 동작

이러한 필드는 `const`가 되는 데 사용됩니다.

새 동작

이러한 필드는 이제 `static readonly`입니다.

변경 이유

변경 내용:

- 값이 어셈블리 경계를 넘어 포함되지 않도록 하여 필요에 따라 값을 수정할 수 있도록 합니다.
- 참조 같음 검사를 더 빠르게 수행할 수 있습니다.

권장 작업

3.0에 대해 다시 컴파일합니다. 다음 방법으로 이러한 필드를 사용하는 소스 코드는 더 이상 이렇게 할 수 없습니다.

- 특성 인수로 사용
- `case` 문의 `switch`로 사용
- 다른 `const`를 정의하는 경우

호환성이 손상되는 변경을 해결하려면 자체 정의된 헤더 이름 상수 또는 문자열 리터럴을 사용하도록 전환합니다.

범주

ASP.NET Core

영향을 받는 API

Microsoft.Net.Http.Headers.HeaderNames

HTTP: 응답 본문 인프라 변경

HTTP 응답 본문을 지원하는 인프라가 변경되었습니다. `HttpResponse`를 직접 사용하는 경우 코드를 변경할 필요가 없습니다. `HttpResponse.Body`를 래핑 또는 대체하거나 `HttpContext.Features`에 액세스하는 경우 자세히 읽어보세요.

도입된 버전

3.0

이전 동작

HTTP 응답 본문과 관련된 세 가지 API가 있습니다.

- `IHttpResponseFeature.Body`
- `IHttpSendFileFeature.SendFileAsync`
- `IHttpBufferingFeature.DisableResponseBuffering`

새 동작

`HttpResponse.Body`를 바꾸면 `IHttpResponseBodyFeature`를 사용하여 전체 `StreamResponseBodyFeature`를 지정된 스트림 주변의 래퍼로 대체하여 모든 예상 API에 대한 기본 구현을 제공합니다. 원래 스트림으로 다시 설정하면 이 변경 내용이 되돌려집니다.

변경 이유

응답 본문 API를 새로운 단일 기능 인터페이스로 결합하는 것이 좋습니다.

권장 작업

이전에 `IHttpResponseBodyFeature`, `IHttpResponseFeature.Body` 또는 `IHttpSendFileFeature`를 사용했던 `IHttpBufferingFeature`를 사용합니다.

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Http.Features.IHttpBufferingFeature
- Microsoft.AspNetCore.Http.Features.IHttpResponseFeature.Body
- Microsoft.AspNetCore.Http.Features.IHttpSendFileFeature

HTTP: 일부 쿠키 SameSite 기본값이 없음으로 변경됨

`SameSite`은(는) 일부 CSRF(교차 사이트 요청 위조) 공격을 완화하는 데 도움이 되는 쿠키의 옵션입니다. 처음에 이 옵션을 도입했을 때 다양한 ASP.NET Core API에서 일관되지 않은 기본값이 사용되었습니다. 이와 같은 불일치로 혼란스러운 결과가 발생했습니다. ASP.NET Core 3.0부터, 이 기본값은 효율적으로 정렬되었습니다. 구성 요소별로 이 기능을 설정해야 합니다.

도입된 버전

3.0

이전 동작

유사한 ASP.NET Core API에서 다른 기본 SameSiteMode 값을 사용했습니다. 불일치에 대한 예는 기본값인 `HttpResponse.Cookies.Append(String, String)` 및 `HttpResponse.Cookies.Append(String, String, CookieOptions)`(으)로 각각 설정된 `SameSiteMode.None` 및 `SameSiteMode.Lax`에서 볼 수 있습니다.

새 동작

영향을 받는 모든 API의 기본값은 `SameSiteMode.None`입니다.

변경 이유

`SameSite`을(를) 옵트인 기능으로 설정하도록 기본값이 변경되었습니다.

권장 작업

쿠키를 내보내는 각 구성 요소는 `SameSite`이(가) 시나리오에 적합한지 여부를 결정해야 합니다. 영향을 받는 API의 사용을 검토하고 필요에 따라 `SameSite`을(를) 다시 구성합니다.

범주

ASP.NET Core

영향을 받는 API

- IResponseCookies.Append(String, String, CookieOptions)

- CookiePolicyOptions.MinimumSameSitePolicy

HTTP: 모든 서버에서 동기 IO가 사용하지 않도록 설정됨

ASP.NET Core 3.0부터 동기 서버 작업은 기본적으로 비활성화되어 있습니다.

변경 내용 설명

`AllowSynchronousIO`는 `HttpRequest.Body.Read`, `HttpResponse.Body.Write` 및 `Stream.Flush`와 같은 동기 IO API를 사용하거나 사용하지 않도록 설정하는 각 서버의 옵션입니다. 이러한 API는 오랫동안 스레드 고갈과 앱 중지의 원인이었습니다. ASP.NET Core 3.0 미리 보기 3부터 이러한 동기 작업은 기본적으로 비활성화되어 있습니다.

영향을 받는 서버:

- 황조롱이
- HttpSys
- IIS In-Process
- 테스트 서버

다음과 유사한 오류가 예상됩니다.

- `Synchronous operations are disallowed. Call ReadAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call WriteAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call FlushAsync or set AllowSynchronousIO to true instead.`

각 서버에는 이 동작을 제어하는 `AllowSynchronousIO` 옵션이 있으며 모든 항목에 대한 기본값은 이제 `false`입니다.

이 동작은 요청에 따라 임시 완화로 재정의할 수도 있습니다. 예시:

C#

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();
if (syncIOFeature != null)
{
    syncIOFeature.AllowSynchronousIO = true;
}
```

`TextWriter` 또는 `Dispose`에서 동기 API를 호출하는 다른 스트림에 문제가 있는 경우 대신 새 `DisposeAsync` API를 호출합니다.

토론은 dotnet/aspnetcore#7644를 참조하세요.

도입된 버전

3.0

이전 동작

기본적으로 `HttpRequest.Body.Read`, `HttpResponse.Body.Write` 및 `Stream.Flush`가 허용됩니다.

새 동작

이러한 동기 API는 기본적으로 허용되지 않습니다.

다음과 유사한 오류가 예상됩니다.

- `Synchronous operations are disallowed. Call ReadAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call WriteAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call FlushAsync or set AllowSynchronousIO to true instead.`

변경 이유

이러한 동기 API는 오랫동안 스레드 고갈과 앱 중지의 원인이었습니다. ASP.NET Core 3.0 미리 보기 3부터 동기 작업은 기본적으로 비활성화되어 있습니다.

권장 작업

비동기 버전의 메서드를 사용합니다. 이 동작은 요청에 따라 임시 완화로 재정의할 수도 있습니다.

C#

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();
if (syncIOFeature != null)
{
```

```
    syncIOFeature.AllowSynchronousIO = true;
}
```

범주

ASP.NET Core

영향을 받는 API

- Stream.Flush
- Stream.Read
- Stream.Write

ID: AddDefaultUI 메서드 오버로드가 제거됨

ASP.NET Core 3.0부터는 IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder,UIFramework) 메서드 오버로드가 더 이상 존재하지 않습니다.

도입된 버전

3.0

변경 이유

이 변경 내용은 정적 웹 자산 기능을 채택한 결과입니다.

권장 작업

두 개의 인수를 사용하는 오버로드 대신 IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder)를 호출합니다. 부트스트랩 3을 사용하는 경우 프로젝트 파일의 `<PropertyGroup>` 요소에 다음 줄을 추가합니다.

XML

```
<IdentityUIFrameworkVersion>Bootstrap3</IdentityUIFrameworkVersion>
```

범주

영향을 받는 API

IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder,UIFramework)

ID: UI의 기본 부트스트랩 버전이 변경됨

ASP.NET Core 3.0부터 ID UI는 기본적으로 부트스트랩 버전 4를 사용합니다.

도입된 버전

3.0

이전 동작

`services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();` 메서드 호출은 `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap3);`와 동일했습니다.

새 동작

`services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();` 메서드 호출은 `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap4);`와 동일합니다.

변경 이유

부트스트랩 4는 ASP.NET Core 3.0 기간 동안 릴리스되었습니다.

권장 작업

기본 ID UI를 사용하고 다음 예제와 같이 `Startup.ConfigureServices`에 추가한 경우 이 변경 내용의 영향을 받게 됩니다.

C#

```
services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();
```

다음 작업 중 하나를 수행합니다.

- 마이그레이션 가이드 를 사용하여 부트스트랩 4를 사용하도록 앱을 마이그레이션합니다.
- 부트스트랩 3의 사용을 적용하도록 `Startup.ConfigureServices`를 업데이트합니다. 예시:

```
C#

services.AddDefaultIdentity<IdentityUser>
().AddDefaultUI(UIFramework.Bootstrap3);
```

범주

ASP.NET Core

영향을 받는 API

없음

ID: 인증되지 않은 ID에 대해 SignInAsync에서 예외가 throw됨

기본적으로 `SignInAsync`는 `IsAuthenticated`가 `false`인 보안 주체/ID에 대해 예외를 throw합니다.

도입된 버전

3.0

이전 동작

`SignInAsync`는 `IsAuthenticated`가 `false`인 ID를 포함하여 모든 보안 주체/ID를 허용합니다.

새 동작

기본적으로 `SignInAsync`는 `IsAuthenticated`가 `false`인 보안 주체/ID에 대해 예외를 throw합니다. 이 동작을 표시하지 않는 새로운 플래그가 있지만 기본 동작이 변경되었습니다.

변경 이유

기본적으로 이러한 보안 주체는 `[Authorize]` / `RequireAuthenticatedUser()`에 의해 거부되었기 때문에 이전 동작은 문제가 있었습니다.

권장 작업

ASP.NET Core 3.0 Preview 6에서는 기본적으로 `RequireAuthenticatedSignIn`인 `AuthenticationOptions`에 `true` 플래그가 있습니다. 이전 동작을 복원하려면 이 플래그를 `false`로 설정합니다.

범주

ASP.NET Core

영향을 받는 API

없음

ID: SignInManager 생성자가 새 매개 변수를 허용함

ASP.NET Core 3.0부터 새 `IUserConfirmation<TUser>` 매개 변수가 `SignInManager` 생성자에 추가되었습니다. 자세한 내용은 dotnet/aspnetcore#8356 을 참조하세요.

도입된 버전

3.0

변경 이유

변경에 대한 동기는 ID에 새 이메일/확인 흐름에 대한 지원을 추가하는 것이었습니다.

권장 작업

`SignInManager`를 수동으로 구성하는 경우 `IUserConfirmation`의 구현을 제공하거나 종속성 주입에서 하나를 가져와 제공합니다.

범주

ASP.NET Core

영향을 받는 API

SignInManager<TUser>

ID: UI가 정적 웹 자산 기능을 사용함

ASP.NET Core 3.0에는 정적 웹 자산 기능이 도입되었으며 ID UI에서 이를 채택했습니다.

변경 내용 설명

ID UI가 정적 웹 자산 기능을 채택한 결과는 다음과 같습니다.

- 프레임워크 선택은 프로젝트 파일에서 `IdentityUIFrameworkVersion` 속성을 사용하여 수행됩니다.
- 부트스트랩 4는 ID UI의 기본 UI 프레임워크입니다. 부트스트랩 3은 수명이 다되었으므로 지원되는 버전으로 마이그레이션하는 것을 고려해야 합니다.

도입된 버전

3.0

이전 동작

ID UI의 기본 UI 프레임워크는 **부트스트랩 3**입니다. `AddDefaultUI`에서 `Startup.ConfigureServices` 메서드 호출에 대한 매개 변수를 사용하여 UI 프레임워크를 구성할 수 있습니다.

새 동작

ID UI의 기본 UI 프레임워크는 **부트스트랩 4**입니다. UI 프레임워크는 `AddDefaultUI` 메서드 호출 대신 프로젝트 파일에 구성되어야 합니다.

변경 이유

정적 웹 자산 기능을 채택하려면 UI 프레임워크 구성이 MSBuild로 이동해야 했습니다. 포함할 프레임워크를 결정하는 것은 런타임 결정이 아니라 빌드 시간 결정입니다.

권장 작업

사이트 UI를 검토하여 새 부트스트랩 4 구성 요소가 호환되는지 확인합니다. 필요한 경우 `IdentityUIFrameworkVersion` MSBuild 속성을 사용하여 부트스트랩 3으로 되돌립니다. 프로젝트 파일의 `<PropertyGroup>` 요소에 속성을 추가합니다.

```xml
<IdentityUIFrameworkVersion>Bootstrap3</IdentityUIFrameworkVersion>
```

범주

ASP.NET Core

영향을 받는 API

IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder, UIFramework)

Kestrel: 연결 어댑터가 제거됨

"pubternal" API를 `public`으로 이동하는 과정의 일부로, `IConnectionAdapter`의 개념이 Kestrel에서 제거되었습니다. 연결 어댑터는 연결 미들웨어(ASP.NET Core 파이프라인의 HTTP 미들웨어와 유사하지만 하위 수준 연결용)로 대체되고 있습니다. HTTPS 및 연결 로깅은 연결 어댑터에서 연결 미들웨어로 이동했습니다. 이러한 확장 메서드는 계속해서 원활하게 작동하지만 구현 세부 정보는 변경되었습니다.

자세한 내용은 dotnet/aspnetcore#11412 를 참조하세요. 토론은 dotnet/aspnetcore#11475 를 참조하세요.

도입된 버전

3.0

이전 동작

`IConnectionAdapter`를 사용하여 kestrel 확장성 구성 요소를 만들었습니다.

새 동작

kestrel 확장성 구성 요소는 미들웨어 로 만들었습니다.

변경 이유

이 변경은 보다 유연한 확장성 아키텍처를 제공하기 위한 것입니다.

권장 작업

`IConnectionAdapter`에 표시된 대로 새 미들웨어 패턴을 사용하도록 구현 을 변환합니다.

범주

ASP.NET Core

영향을 받는 API

`Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal.IConnectionAdapter`

Kestrel: 빈 HTTPS 어셈블리가 제거됨

Microsoft.AspNetCore.Server.Kestrel.Https 어셈블리가 제거되었습니다.

도입된 버전

3.0

변경 이유

ASP.NET Core 2.1에서 `Microsoft.AspNetCore.Server.Kestrel.Https`의 내용이 Microsoft.AspNetCore.Server.Kestrel.Core으로 이동했습니다. 이 변경은 `[TypeForwardedTo]` 특성을 사용하여 중단되지 않는 방식으로 수행되었습니다.

권장 작업

- `Microsoft.AspNetCore.Server.Kestrel.Https` 2.0를 참조하는 라이브러리는 모든 ASP.NET Core 종속성을 2.1 이상으로 업데이트해야 합니다. 그렇지 않으면 ASP.NET Core 3.0 앱에 로드될 때 중단될 수 있습니다.
- ASP.NET Core 2.1 이상을 대상으로 하는 앱 및 라이브러리는 `Microsoft.AspNetCore.Server.Kestrel.Https` NuGet 패키지에 대한 직접 참조를 모두 제거해야 합니다.

범주

ASP.NET Core

영향을 받는 API

없음

Kestrel: 요청 후행부 헤더가 새 컬렉션으로 이동됨

이전 버전에서 Kestrel은 요청 본문을 끝까지 읽을 때 HTTP/1.1 청크 처리된 트레일러 헤더를 요청 헤더 컬렉션에 추가했습니다. 이 동작은 헤더와 트레일러 사이의 모호성에 대한 문제를 발생시켰습니다. 트레일러를 새 컬렉션으로 이동하기로 결정했습니다.

HTTP/2 요청 트레일러는 ASP.NET Core 2.2에서 사용할 수 없었지만 이제 ASP.NET Core 3.0의 이 새 컬렉션에서도 사용할 수 있습니다.

이러한 트레일러에 액세스하기 위해 새 요청 확장 메서드가 추가되었습니다.

HTTP/1.1 트레일러는 전체 요청 본문을 읽은 후에 사용할 수 있습니다.

HTTP/2 트레일러는 클라이언트에서 수신한 후 사용할 수 있습니다. 클라이언트는 전체 요청 본문이 최소한 서버에 의해 버퍼링될 때까지 트레일러를 보내지 않습니다. 버퍼 공간을 확보하기 위해 요청 본문을 읽어야 할 수도 있습니다. 트레일러는 요청 본문을 끝까지 읽으면 항상 사용할 수 있습니다. 트레일러는 본문의 끝을 표시합니다.

도입된 버전

3.0

이전 동작

요청 트레일러 헤더가 `HttpRequest.Headers` 컬렉션에 추가됩니다.

새 동작

요청 트레일러 헤더가 컬렉션에 `HttpRequest.Headers`. `HttpRequest`에서 다음 확장 메서드를 사용하여 액세스합니다.

- `GetDeclaredTrailers()` - 본문 다음에 사용할 트레일러를 나열하는 요청 "트레일러" 헤더를 가져옵니다.

- `SupportsTrailers()` - 요청에서 트레일러 헤더 수신을 지원하는지 여부를 나타냅니다.
- `CheckTrailersAvailable()` - 요청에서 트레일러를 지원하는지 여부와 읽을 수 있는지 여부를 확인합니다.
- `GetTrailer(string trailerName)` - 응답에서 요청된 후행 헤더를 가져옵니다.

변경 이유

트레일러는 gRPC와 같은 시나리오의 주요 기능입니다. 트레일러를 병합하여 헤더를 요청하는 것은 사용자에게 혼동을 줍니다.

권장 작업

`HttpRequest`에서 트레일러 관련 확장 메서드를 사용하여 트레일러에 액세스합니다.

범주

ASP.NET Core

영향을 받는 API

HttpRequest.Headers

Kestrel: 전송 추상화 제거 및 공용

"pubternal" API에서 벗어나 이동하는 과정에서 Kestrel 전송 계층 API는 `Microsoft.AspNetCore.Connections.Abstractions` 라이브러리의 공용 인터페이스로 노출됩니다.

도입된 버전

3.0

이전 동작

- 전송 관련 추상화는 `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions` 라이브러리에서 사용할 수 있었습니다.
- `ListenOptions.NoDelay` 속성을 사용할 수 있었습니다.

새 동작

- `IConnectionListener` 인터페이스는 `Microsoft.AspNetCore.Connections.Abstractions` 라이브러리에서 가장 많이 사용되는 기능을 노출하기 위해 `...Transport.Abstractions` 라이브러리에 도입되었습니다.
- 이제 `NoDelay`는 전송 옵션(`LibuvTransportOptions` 및 `SocketTransportOptions`)에서 사용할 수 있습니다.
- `SchedulingMode`를 더 이상 사용할 수 없습니다.

변경 이유

ASP.NET Core 3.0은 "pubternal" API에서 벗어나 이동했습니다.

권장 작업

범주

ASP.NET Core

영향을 받는 API

없음

지역화: ResourceManagerWithCultureStringLocalizer 및 WithCulture가 사용되지 않음으로 표시됨

ResourceManagerWithCultureStringLocalizer 클래스 및 WithCulture 인터페이스 멤버는 특히 고유한 `IStringLocalizer` 구현을 만들 때 지역화 사용자에게 혼동의 원인이 됩니다. 이러한 항목은 사용자에게 `IStringLocalizer` 인스턴스가 "언어별, 리소스별"이라는 인상을 줍니다. 실제로 인스턴스는 "리소스별"이어야 합니다. 검색된 언어는 실행 시 `CultureInfo.CurrentUICulture`에 의해 결정됩니다. 혼동의 원인을 제거하기 위해 API는 ASP.NET Core 3.0에서 사용되지 않는 것으로 표시되었습니다. API는 이후 릴리스에서 제거됩니다.

컨텍스트는 dotnet/aspnetcore#3324를 참조하세요. 토론은 dotnet/aspnetcore#7756을 참조하세요.

도입된 버전

3.0

이전 동작

메서드가 `Obsolete`로 표시되지 않았습니다.

새 동작

메서드는 `Obsolete`로 표시됩니다.

변경 이유

API는 권장되지 않는 사용 사례를 나타냅니다. 지역화 디자인에 대한 혼동이 있었습니다.

권장 작업

대신 `ResourceManagerStringLocalizer`를 사용하는 것이 좋습니다. `CurrentCulture`에서 문화권을 설정하도록 합니다. 옵션이 아닌 경우 ResourceManagerWithCultureStringLocalizer 복사본을 만들고 사용합니다.

범주

ASP.NET Core

영향을 받는 API

- `Microsoft.Extensions.Localization.ResourceManagerWithCultureStringLocalizer`
- `Microsoft.Extensions.Localization.ResourceManagerStringLocalizer.WithCulture`

로깅: DebugLogger 클래스가 내부로 만들어짐

ASP.NET Core 3.0 이전에는 `DebugLogger`의 액세스 한정자가 `public`였습니다. ASP.NET Core 3.0에서 액세스 한정자가 `internal`로 변경되었습니다.

도입된 버전

3.0

변경 이유

변경 내용은 다음과 같습니다.

- `ConsoleLogger`와 같은 다른 로거 구현과 일관성을 유지합니다.
- API 표면을 줄입니다.

권장 작업

AddDebug `ILoggingBuilder` 확장 메서드를 사용하여 디버그 로깅을 사용하도록 설정합니다. 서비스를 수동으로 동록해야 하는 경우에도 DebugLoggerProvider는 여전히 `public`입니다.

범주

ASP.NET Core

영향을 받는 API

`Microsoft.Extensions.Logging.Debug.DebugLogger`

MVC: 컨트롤러 작업 이름에서 비동기 접미사가 잘림

dotnet/aspnetcore#4849 를 해결하는 과정의 일부로, ASP.NET Core MVC는 기본적으로 작업 이름에서 `Async` 접미사를 잘라냅니다. ASP.NET Core 3.0부터 이 변경 내용은 라우팅 및 링크 생성 모두에 영향을 줍니다.

도입된 버전

3.0

이전 동작

다음 ASP.NET Core MVC 컨트롤러를 고려하세요.

C#

```
public class ProductController : Controller
{
    public async IActionResult ListAsync()
    {
        var model = await DbContext.Products.ToListAsync();
        return View(model);
    }
}
```

작업은 `Product/ListAsync`를 통해 라우팅할 수 있습니다. 링크 생성을 위해서는 `Async` 접미사를 지정해야 합니다. 예시:

CSHTML

```
<a asp-controller="Product" asp-action="ListAsync">List</a>
```

새 동작

ASP.NET Core 3.0에서 작업은 `Product/List`를 통해 라우팅할 수 있습니다. 링크 생성 코드는 `Async` 접미사를 생략해야 합니다. 예시:

CSHTML

```
<a asp-controller="Product" asp-action="List">List</a>
```

이 변경은 `[ActionName]` 특성을 사용하여 지정된 이름에는 영향을 주지 않습니다. `MvcOptions.SuppressAsyncSuffixInActionNames`에서 `false`를 `Startup.ConfigureServices`로 설정하여 새 동작을 사용하지 않도록 설정할 수 있습니다.

C#

```
services.AddMvc(options =>
{
   options.SuppressAsyncSuffixInActionNames = false;
});
```

변경 이유

규칙에 따라 비동기 .NET 메서드는 `Async`로 접미사가 붙습니다. 그러나 메서드가 MVC 작업을 정의할 때 `Async` 접미사를 사용하는 것은 바람직하지 않습니다.

권장 작업

앱이 이름의 `Async` 접미사를 유지하는 MVC 작업에 의존하는 경우 다음 완화 방법 중 하나를 선택합니다.

- `[ActionName]` 특성을 사용하여 원래 이름을 유지합니다.
- `MvcOptions.SuppressAsyncSuffixInActionNames`에서 `false`를 `Startup.ConfigureServices`로 설정하여 전체 이름 바꾸기를 사용하지 않도록 설정합니다.

C#

```
services.AddMvc(options =>
{
   options.SuppressAsyncSuffixInActionNames = false;
});
```

범주

ASP.NET Core

영향을 받는 API

없음

MVC: JsonResult를 Microsoft.AspNetCore.Mvc.Core로 이동

`JsonResult`가 `Microsoft.AspNetCore.Mvc.Core` 어셈블리로 이동했습니다. 이 형식은 Microsoft.AspNetCore.Mvc.Formatters.Json 에서 정의되었습니다. 대부분 사용자의 문제를 해결하기 위해 어셈블리 수준 [TypeForwardedTo] 특성이 `Microsoft.AspNetCore.Mvc.Formatters.Json`에 추가되었습니다. 타사 라이브러리를 사용하는 앱에서 문제가 발생할 수 있습니다.

도입된 버전

3.0 미리 보기 6

이전 동작

2.2 기반 라이브러리를 사용하는 앱이 성공적으로 빌드됩니다.

새 동작

2.2 기반 라이브러리를 사용하는 앱이 컴파일에 실패합니다. 다음 텍스트의 변형을 포함하는 오류가 제공됩니다.

출력

```
The type 'JsonResult' exists in both 'Microsoft.AspNetCore.Mvc.Core,
Version=3.0.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60' and
```

```
'Microsoft.AspNetCore.Mvc.Formatters.Json, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=adb9793829ddae60'
```

이러한 문제의 예는 dotnet/aspnetcore#7220 을 참조하세요.

변경 이유

플랫폼 수준이 aspnet/Announcements#325 에서 설명하는 것과 같이 ASP.NET Core의 컴퍼지션으로 변경됩니다.

권장 작업

`Microsoft.AspNetCore.Mvc.Formatters.Json` 2.2 버전에 대해 컴파일된 라이브러리를 다시 컴파일하여 모든 소비자에 대한 문제를 해결해야 할 수 있습니다. 영향을 받는 경우 라이브러리 작성자에게 문의하세요. ASP.NET Core 3.0을 대상으로 하는 라이브러리의 재컴파일을 요청합니다.

범주

ASP.NET Core

영향을 받는 API

Microsoft.AspNetCore.Mvc.JsonResult

MVC: 사전 컴파일 도구가 사용되지 않음

ASP.NET Core 1.1에서는 Razor 파일(`Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` 파일)의 게시 시간 컴파일 지원을 추가하기 위해 (MVC 미리 컴파일 도구) 패키지가 도입되었습니다. ASP.NET Core 2.1에서는 Razor SDK가 미리 컴파일 도구의 기능을 확장하기 위해 도입되었습니다. Razor SDK는 Razor 파일의 빌드 및 게시 시간 컴파일에 대한 지원을 추가했습니다. SDK는 앱 시작 시간을 개선하면서 빌드 시 *.cshtml* 파일의 정확성을 확인합니다. Razor SDK는 기본적으로 켜져 있으며 사용을 시작하는 데 제스처는 필요하지 않습니다.

ASP.NET Core 3.0에서는 ASP.NET Core 1.1-era MVC 미리 컴파일 도구가 제거되었습니다. 이전 패키지 버전은 패치 릴리스에서 중요한 버그 및 보안 픽스를 계속 받습니다.

도입된 버전

3.0

이전 동작

`Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` 패키지는 MVC Razor 뷰를 미리 컴파일하는 데 사용되었습니다.

새 동작

Razor SDK는 기본적으로 이 기능을 지원합니다.
`Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` 패키지가 더 이상 업데이트되지 않습니다.

변경 이유

Razor SDK는 더 많은 기능을 제공하고 빌드 시 *.cshtml* 파일의 정확성을 확인합니다. 또한 SDK는 앱 시작 시간을 개선합니다.

권장 작업

ASP.NET Core 2.1 이상 사용자의 경우 Razor SDK에서 미리 컴파일에 대한 기본 지원을 사용하도록 업데이트합니다. 버그 또는 누락된 기능으로 인해 Razor SDK로 마이그레이션할 수 없는 경우 dotnet/aspnetcore 에서 문제를 엽니다.

범주

ASP.NET Core

영향을 받는 API

없음

MVC: "Pubternal" 형식이 내부로 변경됨

ASP.NET Core 3.0에서는 MVC의 모든 "pubternal" 형식이 지원되는 네임스페이스에서 `public` 또는 필요에 따라 `internal`로 업데이트되었습니다.

변경 내용 설명

ASP.NET Core에서 "pubternal" 형식은 `public`으로 선언되었지만 `.Internal` 접미사가 지정된 네임스페이스에 있습니다. 이러한 형식은 `public`이지만 지원 정책이 없으며 호환성이 손상되는 변경이 적용됩니다. 아쉽게도 이러한 형식의 우발적 사용이 일반적으로 발생하므로 이러한 프로

젝트에 대한 호환성이 손상되는 변경이 발생하고 프레임워크를 유지 관리하는 기능이 제한됩니다.

도입된 버전

3.0

이전 동작

MVC의 일부 형식은 `public`이지만 `.Internal` 네임스페이스에 있습니다. 이러한 형식에는 지원 정책이 없었으며 호환성이 손상되는 변경이 적용되었습니다.

새 동작

이러한 모든 형식은 지원되는 네임스페이스에서 `public`으로 업데이트되거나 `internal`로 표시됩니다.

변경 이유

"pubternal" 형식의 우발적 사용이 일반적으로 발생하므로 이러한 프로젝트에 대한 호환성이 손상되는 변경이 발생하고 프레임워크를 유지 관리하는 기능이 제한됩니다.

권장 작업

실제로 `public`이 되고 지원되는 새 네임스페이스로 이동한 형식을 사용하는 경우 새 네임스페이스와 일치하도록 참조를 업데이트합니다.

`internal`로 표시된 형식을 사용하는 경우에는 다른 방법을 찾아야 합니다. 이전에 "pubternal" 형식은 공용으로 지원되지 않았습니다. 이러한 네임스페이스의 앱에 중요한 특정 형식이 있는 경우 dotnet/aspnetcore에서 문제를 해결합니다. 요청된 형식 `public`을 만드는 것을 고려할 수 있습니다.

범주

ASP.NET Core

영향을 받는 API

이 변경 내용에는 다음 네임스페이스의 형식이 포함됩니다.

- `Microsoft.AspNetCore.Mvc.Cors.Internal`
- `Microsoft.AspNetCore.Mvc.DataAnnotations.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Json.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.Internal`
- `Microsoft.AspNetCore.Mvc.Internal`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Internal`
- `Microsoft.AspNetCore.Mvc.Razor.Internal`
- `Microsoft.AspNetCore.Mvc.RazorPages.Internal`
- `Microsoft.AspNetCore.Mvc.TagHelpers.Internal`
- `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal`

MVC: 웹 API 호환성 심이 제거됨

ASP.NET Core 3.0부터는 `Microsoft.AspNetCore.Mvc.WebApiCompatShim` 패키지를 더 이상 사용할 수 없습니다.

변경 내용 설명

`Microsoft.AspNetCore.Mvc.WebApiCompatShim`(WebApiCompatShim) 패키지는 ASP.NET Core에서 ASP.NET 4.x Web API 2와의 부분 호환성을 제공하여 기존 Web API 구현을 ASP.NET Core로 마이그레이션하는 것을 간소화합니다. 그러나 WebApiCompatShim을 사용하는 앱은 최신 ASP.NET Core 릴리스에서 제공되는 API 관련 기능의 이점을 활용하지 못합니다. 이러한 기능에는 향상된 Open API 사양 생성, 표준화된 오류 처리 및 클라이언트 코드 생성이 포함됩니다. 3.0에서 API 노력에 더 집중하기 위해 WebApiCompatShim이 제거되었습니다.
WebApiCompatShim을 사용하는 기존 앱은 최신 `[ApiController]` 모델로 마이그레이션해야 합니다.

도입된 버전

3.0

변경 이유

Web API 호환성 shim은 마이그레이션 도구였습니다. ASP.NET Core에 추가된 새로운 기능에 대한 사용자 액세스를 제한합니다.

권장 작업

이 shim의 사용을 제거하고 ASP.NET Core 자체의 유사한 기능으로 직접 마이그레이션합니다.

범주

ASP.NET Core

영향을 받는 API

Microsoft.AspNetCore.Mvc.WebApiCompatShim

Razor: RazorTemplateEngine API가 제거됨

`RazorTemplateEngine` API가 제거되고 `Microsoft.AspNetCore.Razor.Language.RazorProjectEngine`으로 바뀌었습니다.

자세한 내용은 GitHub 이슈 dotnet/aspnetcore#25215 를 참조하세요.

도입된 버전

3.0

이전 동작

템플릿 엔진을 만들고 Razor 파일의 코드를 구문 분석하고 생성하는 데 사용할 수 있습니다.

새 동작

`RazorProjectEngine`을 만들고 Razor 파일의 코드를 구문 분석하고 생성하기 위해 해당 엔진에 `RazorTemplateEngine`과 동일한 유형의 정보를 제공할 수 있습니다. `RazorProjectEngine`은 추가 수준의 구성도 제공합니다.

변경 이유

`RazorTemplateEngine`이 기존 구현과 너무 긴밀하게 결합되었습니다. 이러한 결합으로 인해 Razor 구문 분석/생성 파이프라인을 제대로 구성하려고 할 때 더 많은 질문이 발생했습니다.

권장 작업

`RazorProjectEngine` 대신 `RazorTemplateEngine`를 사용합니다. 다음 예를 살펴보세요.

RazorProjectEngine 만들기 및 구성

C#

```
RazorProjectEngine projectEngine =
    RazorProjectEngine.Create(RazorConfiguration.Default,
        RazorProjectFileSystem.Create(@"C:\source\repos\ConsoleApp4\ConsoleApp4"),
        builder =>
        {
            builder.ConfigureClass((document, classNode) =>
            {
                classNode.ClassName = "MyClassName";

                // Can also configure other aspects of the class here.
            });

            // More configuration can go here
        });
```

Razor 파일의 코드 생성

C#

```
RazorProjectItem item = projectEngine.FileSystem.GetItem(
    @"C:\source\repos\ConsoleApp4\ConsoleApp4\Example.cshtml",
    FileKinds.Legacy);
RazorCodeDocument output = projectEngine.Process(item);

// Things available
RazorSyntaxTree syntaxTree = output.GetSyntaxTree();
DocumentIntermediateNode intermediateDocument =
    output.GetDocumentIntermediateNode();
RazorCSharpDocument csharpDocument = output.GetCSharpDocument();
```

범주

ASP.NET Core

영향을 받는 API

- `RazorTemplateEngine`
- `RazorTemplateEngineOptions`

Razor: 런타임 컴파일이 패키지로 이동됨

Razor 뷰 및 Razor Pages의 런타임 컴파일을 지원하기 위해 별도의 패키지로 이동되었습니다.

도입된 버전

3.0

이전 동작

추가 패키지 없이 런타임 컴파일을 사용할 수 있습니다.

새 동작

이 기능은 Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation 패키지로 이동되었습니다.

런타임 컴파일을 지원하기 위해 이전에 `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions`에서 다음 API를 사용할 수 있었습니다. 이제 `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation.MvcRazorRuntimeCompilationOptions`를 통해 API를 사용할 수 있습니다.

- `RazorViewEngineOptions.FileProviders`는 이제 `MvcRazorRuntimeCompilationOptions.FileProviders`입니다.
- `RazorViewEngineOptions.AdditionalCompilationReferences`는 이제 `MvcRazorRuntimeCompilationOptions.AdditionalReferencePaths`입니다.

또한 `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions.AllowRecompilingViewsOnFileChange`가 제거되었습니다. 파일 변경 시 재컴파일은 `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` 패키지를 참조하여 기본적으로 사용하도록 설정됩니다.

변경 이유

이 변경은 Roslyn에서 ASP.NET Core 공유 프레임워크 종속성을 제거하기 위해 필요했습니다.

권장 작업

Razor 파일의 런타임 컴파일 또는 재컴파일이 필요한 앱은 다음 단계를 수행해야 합니다.

1. `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` 패키지에 대한 참조를 추가합니다.

2. `Startup.ConfigureServices`에 대한 호출을 포함하도록 프로젝트의 `AddRazorRuntimeCompilation` 메서드를 업데이트합니다. 예시:

```
C#

services.AddMvc()
    .AddRazorRuntimeCompilation();
```

범주

ASP.NET Core

영향을 받는 API

Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions

세션 상태: 사용되지 않는 API가 제거됨

세션 쿠키를 구성하는 데 사용되지 않는 API가 제거되었습니다. 자세한 내용은 aspnet/Announcements#257 을 참조하세요.

도입된 버전

3.0

변경 이유

이 변경 내용은 쿠키를 사용하는 기능을 구성하기 위해 API 간에 일관성을 유지합니다.

권장 작업

제거된 API의 사용을 새 대체 항목으로 마이그레이션합니다. `Startup.ConfigureServices`에서 다음 예제를 참조하세요.

```
C#

public void ConfigureServices(ServiceCollection services)
{
    services.AddSession(options =>
    {
        // Removed obsolete APIs
```

```
        options.CookieName = "SessionCookie";
        options.CookieDomain = "contoso.com";
        options.CookiePath = "/";
        options.CookieHttpOnly = true;
        options.CookieSecure = CookieSecurePolicy.Always;

        // new API
        options.Cookie.Name = "SessionCookie";
        options.Cookie.Domain = "contoso.com";
        options.Cookie.Path = "/";
        options.Cookie.HttpOnly = true;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    });
}
```

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Builder.SessionOptions.CookieDomain
- Microsoft.AspNetCore.Builder.SessionOptions.CookieHttpOnly
- Microsoft.AspNetCore.Builder.SessionOptions.CookieName
- Microsoft.AspNetCore.Builder.SessionOptions.CookiePath
- Microsoft.AspNetCore.Builder.SessionOptions.CookieSecure

공유 프레임워크: Microsoft.AspNetCore.App에서 제거되는 어셈블리

ASP.NET Core 3.0부터 ASP.NET Core 공유 프레임워크(`Microsoft.AspNetCore.App`)에는 Microsoft에서 완전히 개발하고, 지원하고, 서비스를 제공하는 자사 어셈블리만 포함되어 있습니다.

변경 내용 설명

이러한 변경은 ASP.NET Core "플랫폼"의 경계를 다시 정의하는 것으로 간주합니다. 공유 프레임워크는 GitHub를 통해 누구나 원본으로 빌드할 수 있으며, .NET Core 공유 프레임워크의 기존 이점을 앱에 계속 제공합니다. 몇 가지 이점으로 더 작은 배포 크기, 중앙 집중식 패치, 더 빠른 시작 시간 등이 있습니다.

변경의 일부로 몇 가지 주목할 만한 호환성이 손상되는 변경이 `Microsoft.AspNetCore.App`에 도입되었습니다.

도입된 버전

3.0

이전 동작

프로젝트는 프로젝트 파일의 `Microsoft.AspNetCore.App` 요소를 통해 `<PackageReference>`을 참조했습니다.

또한 `Microsoft.AspNetCore.App`에는 다음과 같은 하위 구성 요소가 포함되었습니다.

- Json.NET(`Newtonsoft.Json`)
- Entity Framework Core(접두사가 `Microsoft.EntityFrameworkCore.`인 어셈블리)
- Roslyn(`Microsoft.CodeAnalysis`)

새 동작

`Microsoft.AspNetCore.App`에 대한 참조에는 더 이상 프로젝트 파일의 `<PackageReference>` 요소가 필요하지 않습니다. .NET Core SDK는 `<FrameworkReference>`의 사용을 대체하는 `<PackageReference>`라는 새 요소를 지원합니다.

자세한 내용은 dotnet/aspnetcore#3612를 참조하세요.

Entity Framework Core는 NuGet 패키지로 제공됩니다. 이 변경은 배송 모델을 .NET의 다른 모든 데이터 액세스 라이브러리와 맞춥니다. 다양한 .NET 플랫폼을 지원하면서 계속 혁신할 수 있는 가장 간단한 경로인 Entity Framework Core를 제공합니다. Entity Framework Core를 공유 프레임워크 밖으로 이동하더라도 Microsoft에서 개발하고, 지원하고, 서비스를 제공하는 라이브러리의 상태에는 영향을 주지 않습니다. .NET Core 지원 정책에서는 이를 계속 다루고 있습니다.

Json.NET 및 Entity Framework Core는 ASP.NET Core를 계속 사용합니다. 그러나 공유 프레임워크에는 포함되지 않습니다.

자세한 내용은 .NET Core 3.0에서 JSON의 미래를 참조하세요. 또한 공유 프레임워크에서 제거된 이진 파일의 전체 목록도 참조하세요.

변경 이유

이 변경으로 인해 `Microsoft.AspNetCore.App`의 사용이 간소화되고 NuGet 패키지와 공유 프레임워크 간의 중복이 줄어듭니다.

이 변경의 동기에 대한 자세한 내용은 이 블로그 게시물을 참조하세요.

권장 작업

ASP.NET Core 3.0부터 더 이상 프로젝트가 `Microsoft.AspNetCore.App`에서 NuGet 패키지로 어셈블리를 사용할 필요가 없습니다. ASP.NET Core 공유 프레임워크의 대상 지정 및 사용을 간소화하기 위해 ASP.NET Core 1.0 이후에 제공된 많은 NuGet 패키지가 더 이상 생성되지 않습니다. 이러한 패키지에서 제공하는 API는 `<FrameworkReference>`를 `Microsoft.AspNetCore.App`에 사용하여 앱에서 계속 사용할 수 있습니다. 일반적인 API의 예로 Kestrel, MVC 및 Razor가 있습니다.

이 변경은 ASP.NET Core 2.x의 `Microsoft.AspNetCore.App`을 통해 참조되는 모든 이진 파일에 적용되지 않습니다. 주목할 만한 예외는 다음과 같습니다.

- 계속해서 .NET Standard를 대상으로 하는 `Microsoft.Extensions` 라이브러리는 NuGet 패키지로 사용할 수 있습니다(https://github.com/dotnet/extensions 참조).
- `Microsoft.AspNetCore.App`의 일부가 아닌 ASP.NET Core 팀에서 생성한 API입니다. 예를 들어 NuGet 패키지로 사용할 수 있는 구성 요소는 다음과 같습니다.
  - Entity Framework Core (엔티티 프레임워크 코어)
  - 타사 통합 기능을 제공하는 API
  - 실험적 기능
  - 공유 프레임워크에 있어야 하는 요구 사항을 충족시킬 수 없는 종속성이 있는 API
- Json.NET 지원을 유지하는 MVC 확장. API는 Json.NET 및 MVC 사용을 지원하기 위해 NuGet 패키지로 제공됩니다. 자세한 내용은 ASP.NET Core 마이그레이션 가이드를 참조하세요.
- SignalR .NET 클라이언트는 .NET Standard를 계속 지원하며 NuGet 패키지로 제공됩니다. 이는 Xamarin 및 UWP와 같은 많은 .NET 런타임에서 사용하기 위한 것입니다.

자세한 내용은 3.0에서 공유 프레임워크 어셈블리용 패키지 생성 중지를 참조하세요. 토론은 dotnet/aspnetcore#3757을 참조하세요.

범주

ASP.NET Core

영향을 받는 API

- Microsoft.CodeAnalysis
- Microsoft.EntityFrameworkCore

공유 프레임워크: Microsoft.AspNetCore.All이 제거됨

ASP.NET Core 3.0부터 `Microsoft.AspNetCore.All` 메타패키지 및 일치하는 `Microsoft.AspNetCore.All` 공유 프레임워크가 더 이상 생성되지 않습니다. 이 패키지는 ASP.NET Core 2.2에서 사용할 수 있으며 ASP.NET Core 2.1에서 서비스 업데이트를 계속 받을 수 있습니다.

도입된 버전

3.0

이전 동작

앱은 `Microsoft.AspNetCore.All` 메타패키지를 사용하여 .NET Core에서 `Microsoft.AspNetCore.All` 공유 프레임워크를 대상으로 할 수 있습니다.

새 동작

.NET Core 3.0에는 `Microsoft.AspNetCore.All` 공유 프레임워크가 포함되어 있지 않습니다.

변경 이유

`Microsoft.AspNetCore.All` 메타패키지에는 많은 수의 외부 종속성이 포함되어 있습니다.

권장 작업

`Microsoft.AspNetCore.App` 프레임워크를 사용하도록 프로젝트를 마이그레이션합니다. 이전에 `Microsoft.AspNetCore.All`에서 사용할 수 있었던 구성 요소는 NuGet에서 계속 사용할 수 있습니다. 이러한 구성 요소는 이제 공유 프레임워크에 포함되지 않고 앱과 함께 배포됩니다.

범주

ASP.NET Core

영향을 받는 API

없음

SignalR: HandshakeProtocol.SuccessHandshakeData가 바뀜

HandshakeProtocol.SuccessHandshakeData 필드가 제거되었으며 특정 `IHubProtocol`에 제공된 성공적인 핸드셰이크 응답을 생성하는 도우미 메서드로 대체되었습니다.

도입된 버전

3.0

이전 동작

`HandshakeProtocol.SuccessHandshakeData`는 `public static ReadOnlyMemory<byte>` 필드입니다.

새 동작

`HandshakeProtocol.SuccessHandshakeData`는 지정된 프로토콜을 기반으로 `static`를 반환하는 `GetSuccessfulHandshake(IHubProtocol protocol)` `ReadOnlyMemory<byte>` 메서드로 대체되었습니다.

변경 이유

비상수이며 선택된 프로토콜에 따라 변경되는 추가 필드가 핸드셰이크 *응답*에 추가되었습니다.

권장 작업

없음 이 형식은 사용자 코드에서 사용하도록 설계되지 않았습니다. `public`이기 때문에 SignalR 서버와 클라이언트 간에 공유할 수 있습니다. .NET으로 작성된 고객 SignalR 클라이언트에서도 사용할 수 있습니다. SignalR의 **사용자**는 이 변경 내용의 영향을 받지 않아야 합니다.

범주

ASP.NET Core

영향을 받는 API

HandshakeProtocol.SuccessHandshakeData

SignalR: HubConnection ResetSendPing 및 ResetTimeout 메서드가 제거됨

`ResetSendPing` 및 `ResetTimeout` 메서드가 SignalR `HubConnection` API에서 제거되었습니다. 이러한 메서드는 원래 내부용으로만 사용되었지만 ASP.NET Core 2.2에서 공개되었습니다. 이러한 메서드는 ASP.NET Core 3.0 Preview 4 릴리스부터 사용할 수 없습니다. 토론은 dotnet/aspnetcore#8543 을 참조하세요.

도입된 버전

3.0

이전 동작

API를 사용할 수 있었습니다.

새 동작

API가 제거되었습니다.

변경 이유

이러한 메서드는 원래 내부용으로만 사용되었지만 ASP.NET Core 2.2에서 공개되었습니다.

권장 작업

이러한 메서드를 사용하지 마세요.

범주

ASP.NET Core

영향을 받는 API

- HubConnection.ResetSendPing()
- HubConnection.ResetTimeout()

SignalR: HubConnectionContext 생성자가 변경됨

SignalR의 `HubConnectionContext` 생성자는 이후 증명 추가 옵션에 대해 여러 매개 변수 대신 옵션 유형을 허용하도록 변경되었습니다. 이 변경 내용은 두 개의 생성자를 옵션 유형을 허용하는 단일 생성자로 대체합니다.

도입된 버전

3.0

이전 동작

`HubConnectionContext`에는 두 개의 생성자가 있습니다.

C#

```
public HubConnectionContext(ConnectionContext connectionContext, TimeSpan
keepAliveInterval, ILoggerFactory loggerFactory);
public HubConnectionContext(ConnectionContext connectionContext, TimeSpan
keepAliveInterval, ILoggerFactory loggerFactory, TimeSpan clientTimeoutInterval);
```

새 동작

두 개의 생성자가 제거되고 하나의 생성자로 대체되었습니다.

C#

```
public HubConnectionContext(ConnectionContext connectionContext,
HubConnectionContextOptions contextOptions, ILoggerFactory loggerFactory)
```

변경 이유

새 생성자는 새 옵션 개체를 사용합니다. 따라서 나중에 추가 생성자 및 호환성이 손상되는 변경 없이 `HubConnectionContext`의 기능을 확장할 수 있습니다.

권장 작업

다음 생성자 사용하는 대신:

C#

```
HubConnectionContext connectionContext = new HubConnectionContext(
    connectionContext,
    keepAliveInterval: TimeSpan.FromSeconds(15),
    loggerFactory,
    clientTimeoutInterval: TimeSpan.FromSeconds(15));
```

다음 생성자를 사용합니다.

C#

```
HubConnectionContextOptions contextOptions = new HubConnectionContextOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(15),
    ClientTimeoutInterval = TimeSpan.FromSeconds(15)
};
HubConnectionContext connectionContext = new
HubConnectionContext(connectionContext, contextOptions, loggerFactory);
```

범주

ASP.NET Core

영향을 받는 API

- HubConnectionContext(ConnectionContext, TimeSpan, ILoggerFactory)
- HubConnectionContext(ConnectionContext, TimeSpan, ILoggerFactory, TimeSpan)

SignalR: JavaScript 클라이언트 패키지 이름이 변경됨

ASP.NET Core 3.0 Preview 7에서 SignalR JavaScript 클라이언트 패키지 이름이 `@aspnet/signalr`에서 `@microsoft/signalr`로 변경되었습니다. 이름 변경은 Azure SignalR Service 덕분에 SignalR은 ASP.NET Core 앱 이상에서만 유용하다는 사실을 반영합니다.

이 변경에 대응하려면 *package.json* 파일, `require` 문 및 ECMAScript `import` 문에서 참조를 변경합니다. 이 이름 바꾸기의 일부로 API가 변경되지 않습니다.

토론은 dotnet/aspnetcore#11637 을 참조하세요.

도입된 버전

3.0

이전 동작

클라이언트 패키지의 이름은 `@aspnet/signalr`이었습니다.

새 동작

클라이언트 패키지의 이름은 `@microsoft/signalr`입니다.

변경 이유

이름 변경은 Azure SignalR Service 덕분에 SignalR은 ASP.NET Core 앱 이외에도 유용합니다.

권장 작업

새 패키지 `@microsoft/signalr`로 전환합니다.

범주

ASP.NET Core

영향을 받는 API

없음

SignalR: UseSignalR 및 UseConnections 메서드가 사용되지 않음으로 표시됨

`UseConnections` 및 `UseSignalR` 메서드와 `ConnectionsRouteBuilder` 및 `HubRouteBuilder` 클래스는 ASP.NET Core 3.0에서 사용되지 않는 것으로 표시됩니다.

도입된 버전

3.0

이전 동작

`UseSignalR` 또는 `UseConnections`를 사용하여 SignalR 허브 라우팅이 구성되었습니다.

새 동작

라우팅을 구성하는 이전 방법은 사용되지 않으며 엔드포인트 라우팅으로 바뀌었습니다.

변경 이유

미들웨어가 새 엔드포인트 라우팅 시스템으로 이동되고 있습니다. 미들웨어를 추가하는 이전 방법은 사용되지 않습니다.

권장 작업

`UseSignalR`을 `UseEndpoints`로 바꿉니다.

이전 코드:

```
C#

app.UseSignalR(routes =>
{
    routes.MapHub<SomeHub>("/path");
});
```

새 코드:

```
C#

app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<SomeHub>("/path");
});
```

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Builder.ConnectionsAppBuilderExtensions.UseConnections(IApplicationBuilder, Action<ConnectionsRouteBuilder>)
- Microsoft.AspNetCore.Builder.SignalRAppBuilderExtensions.UseSignalR(IApplicationBuilder, Action<HubRouteBuilder>)
- Microsoft.AspNetCore.Http.Connections.ConnectionsRouteBuilder
- Microsoft.AspNetCore.SignalR.HubRouteBuilder

SPA: SpaServices 및 NodeServices가 사용되지 않음으로 표시됨

다음 NuGet 패키지의 내용은 ASP.NET Core 2.1 이후 모두 필요하지 않습니다. 따라서 다음 패키지는 사용되지 않는 것으로 표시됩니다.

- Microsoft.AspNetCore.SpaServices
- Microsoft.AspNetCore.NodeServices

동일한 이유로 다음 npm 모듈은 더 이상 사용되지 않는 것으로 표시됩니다.

- aspnet-angular
- aspnet-prerendering
- aspnet-webpack
- aspnet-webpack-react
- 도메인 작업

이전 패키지 및 npm 모듈은 나중에 .NET 5에서 제거됩니다.

도입된 버전

3.0

이전 동작

사용되지 않는 패키지 및 npm 모듈은 ASP.NET Core를 다양한 SPA(단일 페이지 앱) 프레임워크와 통합하기 위한 것입니다. 이러한 프레임워크에는 Angular, React 및 Redux를 사용한 React가 포함됩니다.

새 동작

Microsoft.AspNetCore.SpaServices.Extensions NuGet 패키지에 새 통합 메커니즘이 있습니다. 패키지는 ASP.NET Core 2.1 이후 Angular 및 React 프로젝트 템플릿의 기반으로 유지됩니다.

변경 이유

ASP.NET Core는 Angular, React 및 Redux를 React를 포함한 다양한 SPA(단일 페이지 앱) 프레임워크와의 통합을 지원합니다. 처음에 이러한 프레임워크와의 통합은 서버 쪽 렌더링 및 Webpack과의 통합과 같은 시나리오를 처리하는 ASP.NET Core 관련 구성 요소를 사용하여 수행되었습니다. 시간이 지남에 따라 업계 표준이 변경되었습니다. 각 SPA 프레임워크는 자체 표준 명령줄 인터페이스를 릴리스했습니다. 예를 들어 Angular CLI 및 create-react-app입니다.

2018년 5월에 ASP.NET Core 2.1이 릴리스되었을 때 팀은 표준 변경에 대응했습니다. SPA 프레임워크 자체 도구 체인과 통합하는 새롭고 간단한 방법이 제공되었습니다. 새로운 통합 메커니즘은 `Microsoft.AspNetCore.SpaServices.Extensions` 패키지에 존재하며 ASP.NET Core 2.1 이후 Angular 및 React 프로젝트 템플릿의 기반으로 유지됩니다.

이전 ASP.NET Core 관련 구성 요소는 관련이 없으며 권장되지 않는다는 것을 명확하게 하기 위해 다음과 같이 합니다.

- 2.1 이전 통합 메커니즘은 사용되지 않는 것으로 표시되었습니다.

- 지원되는 npm 패키지는 더 이상 사용되지 않는 것으로 표시됩니다.

권장 작업

이러한 패키지를 사용하는 경우 다음 기능을 사용하도록 앱을 업데이트합니다.

- `Microsoft.AspNetCore.SpaServices.Extensions` 패키지에 있습니다.
- 사용 중인 SPA 프레임워크에서 제공됨

서버 쪽 렌더링 및 핫 모듈 재로드와 같은 기능을 사용하려면 해당 SPA 프레임워크에 대한 설명서를 참조하세요. `Microsoft.AspNetCore.SpaServices.Extensions`의 기능은 사용되지 않는 것이 *아니며* 계속 지원됩니다.

범주

ASP.NET Core

영향을 받는 API

- Microsoft.AspNetCore.Builder.SpaRouteExtensions
- Microsoft.AspNetCore.Builder.WebpackDevMiddleware
- Microsoft.AspNetCore.NodeServices.EmbeddedResourceReader
- Microsoft.AspNetCore.NodeServices.INodeServices
- Microsoft.AspNetCore.NodeServices.NodeServicesFactory
- Microsoft.AspNetCore.NodeServices.NodeServicesOptions
- Microsoft.AspNetCore.NodeServices.StringAsTempFile
- Microsoft.AspNetCore.NodeServices.HostingModels.INodeInstance
- Microsoft.AspNetCore.NodeServices.HostingModels.NodeInvocationException
- Microsoft.AspNetCore.NodeServices.HostingModels.NodeInvocationInfo
- Microsoft.AspNetCore.NodeServices.HostingModels.NodeServicesOptionsExtensions
- Microsoft.AspNetCore.NodeServices.HostingModels.OutOfProcessNodeInstance
- Microsoft.AspNetCore.SpaServices.Prerendering.ISpaPrerenderer
- Microsoft.AspNetCore.SpaServices.Prerendering.ISpaPrerendererBuilder

- Microsoft.AspNetCore.SpaServices.Prerendering.JavaScriptModuleExport
- Microsoft.AspNetCore.SpaServices.Prerendering.Prerenderer
- Microsoft.AspNetCore.SpaServices.Prerendering.PrerenderTagHelper
- Microsoft.AspNetCore.SpaServices.Prerendering.RenderToStringResult
- Microsoft.AspNetCore.SpaServices.Webpack.WebpackDevMiddlewareOptions
- Microsoft.Extensions.DependencyInjection.NodeServicesServiceCollectionExtensions
- Microsoft.Extensions.DependencyInjection.PrerenderingServiceCollectionExtensions

---

SPA: SpaServices 및 NodeServices가 더 이상 콘솔 로거로 대체되지 않음

로깅을 구성하지 않으면 Microsoft.AspNetCore.SpaServices 및 Microsoft.AspNetCore.NodeServices에 콘솔 로그가 표시되지 않습니다.

도입된 버전

3.0

이전 동작

로깅이 구성되지 않은 경우 `Microsoft.AspNetCore.SpaServices` 및 `Microsoft.AspNetCore.NodeServices`는 콘솔 로거를 자동으로 만드는 데 사용됩니다.

새 동작

로깅을 구성하지 않으면 `Microsoft.AspNetCore.SpaServices` 및 `Microsoft.AspNetCore.NodeServices`에 콘솔 로그가 표시되지 않습니다.

변경 이유

다른 ASP.NET Core 패키지에서 로깅을 구현하는 방법에 맞춰 조정할 필요가 있습니다.

권장 작업

이전 동작이 필요한 경우 콘솔 로깅을 구성하려면 `services.AddLogging(builder => builder.AddConsole())` 메서드에 `Setup.ConfigureServices`을 추가합니다.

범주

ASP.NET Core

영향을 받는 API

없음

대상 프레임워크: .NET Framework 지원 삭제

ASP.NET Core 3.0부터 .NET Framework는 지원되는 않는 대상 프레임워크입니다.

변경 내용 설명

.NET Framework 4.8은 .NET Framework의 마지막 주 버전입니다. 새 ASP.NET Core 앱은 .NET Core를 기반으로 빌드해야 합니다. .NET Core 3.0 릴리스부터는 ASP.NET Core 3.0을 .NET Core의 일부로 간주할 수 있습니다.

.NET Framework와 함께 ASP.NET Core를 사용하는 고객은 2.1 LTS 릴리스를 사용하여 완전히 지원되는 방식으로 계속할 수 있습니다. 2.1에 대한 지원 및 서비스는 2021년 8월 21일까지 계속됩니다. 이 날짜는 .NET 지원 정책에 따라 LTS 릴리스가 선언된 후 3년입니다. **.NET Framework에서** ASP.NET Core 2.1 패키지에 대한 지원은 다른 패키지 기반 ASP.NET 프레임워크에 대한 서비스 정책과 유사하게 무기한으로 확장됩니다.

.NET Framework에서 .NET Core로 포팅하는 방법에 대한 자세한 내용은 .NET Core로 포팅을 참조하세요.

`Microsoft.Extensions` 패키지(예: 로깅, 종속성 주입 및 구성)와 Entity Framework Core는 영향을 받지 않습니다. .NET Standard를 계속 지원합니다.

이러한 변경의 동기에 대한 자세한 내용은 원래 블로그 게시물을 참조하세요.

도입된 버전

3.0

이전 동작

ASP.NET Core 앱은 .NET Core 또는 .NET Framework에서 실행될 수 있습니다.

새 동작

ASP.NET Core 앱은 .NET Core에서만 실행할 수 있습니다.

권장 작업

다음 작업 중 하나를 수행합니다.

- 앱을 ASP.NET Core 2.1에 보관합니다.
- 앱 및 종속성을 .NET Core로 마이그레이션합니다.

범주

ASP.NET Core

영향을 받는 API

없음

핵심 .NET 라이브러리

- 버전을 보고하는 API가 이제 파일 버전이 아닌 제품 버전을 보고함
- 사용자 지정 EncoderFallbackBuffer 인스턴스는 재귀적으로 대체될 수 없음
- 부동 소수점 서식 및 구문 분석 동작 변경
- 부동 소수점 구문 분석 작업은 더 이상 실패하거나 OverflowException을 throw하지 않음
- InvalidAsynchronousStateException이 다른 어셈블리로 이동됨
- 잘못된 형식의 UTF-8바이트 시퀀스 교체는 유니코드 지침을 따름
- TypeDescriptionProviderAttribute가 다른 어셈블리로 이동됨
- ZipArchiveEntry가 더 이상 일치하지 않는 항목 크기의 아카이브를 처리하지 않음
- FieldInfo.SetValue가 초기화 전용 정적 필드에 대해 예외를 throw
- IEnumerable<T>를 사용하는 확장 메서드에 GroupCollection을 전달하려면 명확성 필요

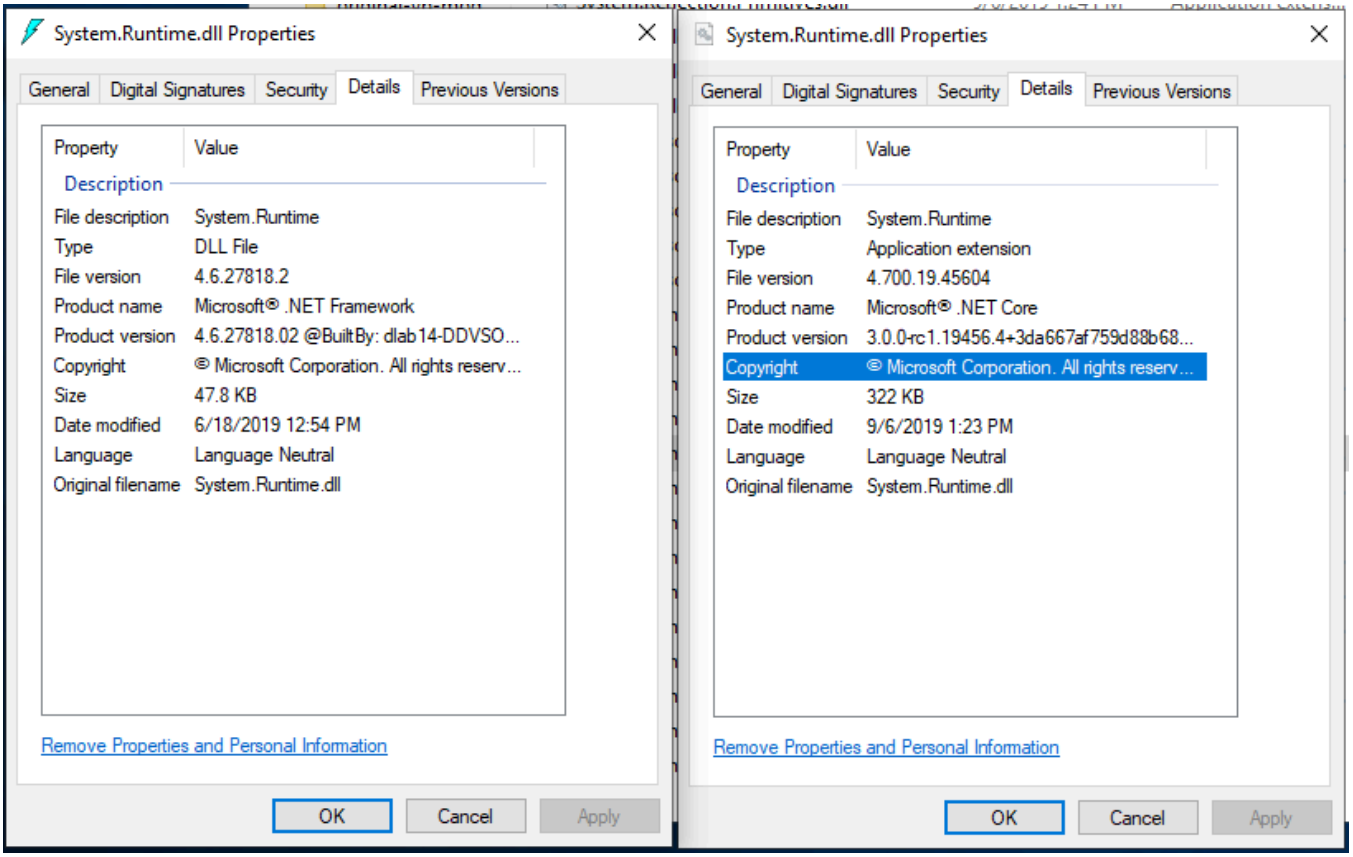
버전을 보고하는 API가 이제 파일 버전이 아닌 제품 버전을 보고함

.NET Core의 버전을 반환하는 대부분의 API가 이제 파일 버전이 아닌 제품 버전을 반환합니다.

변경 내용 설명

.NET Core 2.2 및 이전 버전에서는 Environment.Version, RuntimeInformation.FrameworkDescription 등의 메서드와 .NET Core 어셈블리의 파일 속성 대화 상자에 파일 버전이 반영됩니다. .NET Core 3.0부터는 제품 버전이 반영됩니다.

다음 그림은 *Windows 탐색기* 파일 속성 대화 상자에 표시되는 .NET Core 2.2(왼쪽) 및 .NET Core 3.0(오른쪽)에 대한 **System.Runtime.dll** 어셈블리의 버전 정보 차이를 보여 줍니다.



도입된 버전

3.0

권장 작업

없음 이 변경을 통해 직관적인 버전 검색이 가능합니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- Environment.Version

- RuntimeInformation.FrameworkDescription

사용자 지정 EncoderFallbackBuffer 인스턴스는 재귀적으로 대체될 수 없음

사용자 지정 EncoderFallbackBuffer 인스턴스는 재귀적으로 대체될 수 없습니다. EncoderFallbackBuffer.GetNextChar()를 구현하면 대상 인코딩으로 변환할 수 있는 문자 시퀀스가 생겨야 합니다. 그렇지 않으면 예외가 발생합니다.

변경 내용 설명

런타임은 문자-바이트 트랜스코딩 작업 도중 형식이 잘못되었거나 변환할 수 없는 UTF-16 시퀀스를 검색하여 해당 문자를 EncoderFallbackBuffer.Fallback 메서드에 제공합니다. `Fallback` 메서드는 변환할 수 없는 원래 데이터를 대체할 문자를 결정하며, 이러한 문자는 루프에서 EncoderFallbackBuffer.GetNextChar를 호출하여 드레이닝됩니다.

그런 다음 런타임은 이러한 대체 문자를 대상 인코딩으로 트랜스코딩하려고 합니다. 이 작업이 성공하면 런타임은 원래 입력 문자열에서 중단된 위치에서 트랜스코딩을 계속 실행합니다.

이전에는 EncoderFallbackBuffer.GetNextChar()에 대한 사용자 지정 구현은 대상 인코딩으로 변환할 수 없는 문자 시퀀스를 반환할 수 있습니다. 대체된 문자를 대상 인코딩으로 변환할 수 없는 경우 런타임은 대체 문자를 사용하여 EncoderFallbackBuffer.Fallback 메서드를 다시 한 번 호출하여 EncoderFallbackBuffer.GetNextChar() 메서드가 새 대체 시퀀스를 반환할 것을 예상합니다. 이 프로세스는 런타임이 올바른 형식의 변환 가능한 대체를 확인하거나 최대 재귀 횟수에 도달할 때까지 계속됩니다.

.Net Core 3.0부터 EncoderFallbackBuffer.GetNextChar()에 대한 사용자 지정 구현은 대상 인코딩으로 변환할 수 있는 문자 시퀀스를 반환해야 합니다. 대체된 문자를 대상 인코딩으로 트랜스 코딩할 수 없으면 ArgumentException이 throw됩니다. 런타임은 더 이상 EncoderFallbackBuffer 인스턴스에 대한 재귀 호출을 수행하지 않습니다.

이 동작은 다음 조건 중 세 가지를 모두 충족하는 경우에만 적용됩니다.

- 런타임은 잘못된 형식의 UTF-16 시퀀스 또는 대상 인코딩으로 변환할 수 없는 UTF-16 시퀀스를 검색합니다.
- 사용자 지정 EncoderFallback이 지정되었습니다.
- 사용자 지정 EncoderFallback은 잘못된 형식이거나 변환할 수 없는 새 UTF-16 시퀀스를 대체하려고 시도합니다.

도입된 버전

3.0

권장 작업

대부분의 개발자는 아무 작업도 수행하지 않아도 됩니다.

애플리케이션이 사용자 지정 EncoderFallback 및 EncoderFallbackBuffer 클래스를 사용하는 경우 런타임이 처음 EncoderFallbackBuffer.Fallback 메서드를 호출할 때 Fallback 구현이 대상 인코딩으로 직접 변환할 수 있는 올바른 형식의 UTF-16 데이터로 대체 버퍼를 채우도록 합니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- EncoderFallbackBuffer.Fallback
- EncoderFallbackBuffer.GetNextChar()

부동 소수점 서식 및 구문 분석 동작 변경됨

이제 부동 소수점 구문 분석 및 서식 동작(Double 및 Single 형식 사용)이 IEEE 규격으로 변경되었습니다. 따라서 .NET의 부동 소수점 형식 동작이 기타 IEEE 규격 언어의 동작과 일치합니다. 예를 들어 `double.Parse("SomeLiteral")`는 항상 C#에서 `double x = SomeLiteral`에 대해 생성하는 것과 일치해야 합니다.

변경 내용 설명

.NET Core 2.2 및 이전 버전에서는 Double.ToString 및 Single.ToString을 사용한 서식과 Double.Parse, Double.TryParse, Single.Parse, Single.TryParse을 사용한 구문 분석이 IEEE 규격이 아닙니다. 따라서 지원되는 표준 또는 사용자 지정 형식 문자열로 값이 왕복한다고 보장할 수 없습니다. 입력에 따라 서식이 지정된 값을 구문 분석하려는 시도가 실패하는 경우도 있고, 구문 분석된 값이 원래 값과 달라지는 경우도 있습니다.

.NET Core 3.0부터는 부동 소수점 구문 분석 및 서식 작업이 IEEE 754 규격입니다.

다음 표에서는 두 개의 코드 조각과 .NET Core 2.2와 .NET Core 3.1 사이에서 출력이 어떻게 변경되는지를 보여줍니다.

테이블 확장

코드 조각	.NET Core 2.2에서의 출력	.NET Core 3.1에서의 출력
`Console.WriteLine((-0.0).ToString());`	0	-0

`var value = -3.123456789123456789;`
`Console.WriteLine(value ==`
`double.Parse(value.ToString()));`
	`False`	`True`

자세한 내용은 Floating-point parsing and formatting improvements in .NET Core 3.0 (.NET Core 3.0의 부동 소수점 구문 분석 및 서식 개선 사항)에 대한 블로그 게시물을 참조하세요.

도입된 버전

3.0

권장 작업

.NET Core 3.0의 부동 소수점 구문 분석 및 서식 개선 사항 블로그 게시물의 기존 코드에 미치는 잠재적 영향 섹션에서는 이전 동작을 유지하고자 하는 경우 코드에 대해 수행할 수 있는 몇 가지 변경 사항을 제안합니다.

- 서식에서의 일부 차이점의 경우 다른 서식 문자열을 지정하여 이전 동작과 동일한 동작을 가져올 수 있습니다.
- 구문 분석에서의 차이점의 경우 이전 동작으로 되돌릴 메커니즘이 없습니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- Double.ToString
- Single.ToString
- Double.Parse
- Double.TryParse
- Single.Parse
- Single.TryParse

부동 소수점 구문 분석 작업은 더 이상 실패하거나 OverflowException을 throw하지 않음

부동 소수점 구문 분석 메서드는 더 이상 OverflowException을 throw하지 않거나, 숫자 값이 `false` 또는 Single 부동 소수점 형식의 범위를 벗어난 문자열을 구문 분석할 때 Double를 반환합니다.

변경 내용 설명

.NET Core 2.2 및 이전 버전에서 Double.Parse 및 Single.Parse 메서드는 해당 형식의 범위를 벗어난 값에 대해 OverflowException를 throw합니다. Double.TryParse 및 Single.TryParse 메서드는 범위를 벗어난 숫자 값의 문자열 표현에 대해 `false`를 반환합니다.

.NET Core 3.0부터 범위를 벗어난 숫자 문자열을 구문 분석할 때 Double.Parse, Double.TryParse, Single.Parse 및 Single.TryParse 메서드가 더 이상 실패하지 않습니다. 대신 Double 구문 분석 메서드는 Double.PositiveInfinity를 초과하는 값에 대해 Double.MaxValue를 반환하고 Double.NegativeInfinity보다 작은 값에 대해 Double.MinValue를 반환합니다. 마찬가지로 Single 구문 분석 메서드는 Single.PositiveInfinity를 초과하는 값에 대해 Single.MaxValue를 반환하고 Single.NegativeInfinity보다 작은 값에 대해 Single.MinValue를 반환합니다.

이러한 변경은 개정된 IEEE 754:2008 규정 준수를 위해 이루어졌습니다.

도입된 버전

3.0

권장 작업

이러한 변경 내용은 다음 두 가지 방법 중 하나로 코드에 영향을 줄 수 있습니다.

- 코드는 오버플로가 발생할 때 실행되는 OverflowException에 대한 처리기에 따라 다릅니다. 이 경우 `catch` 문을 제거하고 `If` 또는 Double.IsInfinity가 Single.IsInfinity인지 여부를 테스트하는 `true` 문에 필요한 코드를 넣어야 합니다.
- 코드에서 부동 소수점 값이 `Infinity`가 아니라고 가정합니다. 이 경우 `PositiveInfinity` 및 `NegativeInfinity`의 부동 소수점 값을 확인하는 데 필요한 코드를 추가해야 합니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- Double.Parse
- Double.TryParse
- Single.Parse
- Single.TryParse

InvalidAsynchronousStateException이 다른 어셈블리로 이동됨

InvalidAsynchronousStateException 클래스가 이동되었습니다.

변경 내용 설명

.NET Core 2.2 및 이전 버전에서는 InvalidAsynchronousStateException 클래스가 *System.ComponentModel.TypeConverter* 어셈블리에 있습니다.

.NET Core 3.0부터는 *System.ComponentModel.Primitives* 어셈블리에 있습니다.

도입된 버전

3.0

권장 작업

이 변경은 InvalidAsynchronousStateException등의 메서드나 형식이 특정 어셈블리에 있다고 가정하는 Assembly.GetType 오버로드를 호출하여 리플렉션을 통해 Activator.CreateInstance을 로드하는 애플리케이션에만 영향을 줍니다. 이 경우에는 메서드 호출에서 참조된 어셈블리를 업데이트하여 형식의 새 어셈블리 위치를 반영합니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

없음

잘못된 형식의 UTF-8바이트 시퀀스 교체는 유니코드 지침을 따름

바이트-문자 코드 변환 작업 중에 UTF8Encoding 클래스에서 형식이 잘못된 UTF-8바이트 시퀀스가 있는 경우 출력 문자열에서 이 시퀀스가 '&#xFFFD;' 문자(U+FFFD 대체 문자)로 바뀝니다. .NET Core 3.0은 이전 버전의 .NET Core 및 .NET Framework와 달리 트랜스코딩 작업 도중 이 대체를 수행하기 위해 유니코드 모범 사례를 따릅니다.

이는 새로운 System.Text.Unicode.Utf8 및 System.Text.Rune 형식을 포함하여 .NET 전체에서 UTF-8 처리를 향상하기 위해 기울인 보다 많은 노력의 일환입니다. UTF8Encoding 형식에는 새로 도입된 형식과 일치하는 출력을 생성하도록 향상된 오류 처리 메커니즘이 적용되었습니다.

변경 내용 설명

.Net Core 3.0부터 바이트를 문자로 트랜스코딩하는 경우 UTF8Encoding 클래스는 유니코드 모범 사례를 기반으로 문자 대체를 수행합니다. 사용되는 대체 메커니즘은 최대 하위 부분의 U+FFFD 대체 항목의 *유니코드 표준 버전 12.0, 섹션 3.9(PDF)*에서 설명합니다.

이 동작은 입력 바이트 시퀀스에 잘못된 형식의 UTF-8 데이터가 포함된 *경우에만* 적용됩니다. 또한 UTF8Encoding 인스턴스가 `throwOnInvalidBytes: true`를 사용하여 구성된 경우 `UTF8Encoding` 인스턴스는 U+FFFD 대체를 수행하는 대신 잘못된 입력에 대해 계속 throw합니다. `UTF8Encoding` 생성자에 대한 자세한 내용은 UTF8Encoding(Boolean, Boolean)를 참조하세요.

다음 표에서는 잘못된 3바이트 입력으로 인한 이 변경의 영향을 보여줍니다.

테이블 확장

잘못된 형식의 3바이트 입력	.NET Core 3.0 이하의 출력	.NET Core 3.0 이상의 출력
`[ ED A0 90 ]`	`[ FFFD FFFD ]`(2자 출력)	`[ FFFD FFFD FFFD ]`(3자 출력)

3자 출력은 위에 링크된 유니코드 표준 PDF의 *표 3-9*에 따라 선호되는 출력입니다.

도입된 버전

3.0

권장 작업

개발자는 아무 작업도 수행하지 않아도 됩니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- UTF8Encoding.GetCharCount
- UTF8Encoding.GetChars
- UTF8Encoding.GetString(Byte[], Int32, Int32)

TypeDescriptionProviderAttribute가 다른 어셈블리로 이동됨

TypeDescriptionProviderAttribute 클래스가 이동되었습니다.

변경 내용 설명

.NET Core 2.2 및 이전 버전에서는 TypeDescriptionProviderAttribute 클래스가 *System.ComponentModel.TypeConverter* 어셈블리에 있습니다.

.NET Core 3.0부터는 *System.ObjectModel* 어셈블리에 있습니다.

도입된 버전

3.0

권장 작업

이 변경은 TypeDescriptionProviderAttribute등의 메서드나 형식이 특정 어셈블리에 있다고 가정하는 Assembly.GetType 오버로드를 호출하여 리플렉션을 통해 Activator.CreateInstance 형식을 로드하는 애플리케이션에만 영향을 줍니다. 이 경우에는 메서드 호출에서 참조된 어셈블리를 형식의 새 어셈블리 위치에 맞게 업데이트해야 합니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

없음

ZipArchiveEntry가 더 이상 일치하지 않는 항목 크기의 아카이브를 처리하지 않음

Zip 보관 파일은 중앙 디렉터리 및 로컬 헤더에 압축 크기와 압축되지 않은 크기를 모두 나열합니다. 항목 데이터 자체도 크기를 표시합니다. .NET Core 2.2 버전 이하에서는 이러한 값에 대해 일관성을 검사하지 않았습니다. .NET Core 3.0부터는 검사됩니다.

변경 내용 설명

.Net Core 2.2 버전 이하에서는 로컬 헤더가 zip 파일의 중앙 헤더와 일치하지 않는 경우에도 ZipArchiveEntry.Open()이 성공합니다. 데이터는 압축된 스트림의 끝에 도달할 때까지 압축이 풀립니다. 이는 해당 길이가 중앙 디렉터리/로컬 헤더에 나열된 압축되지 않은 파일 크기를 초과하는 경우에도 마찬가지입니다.

.Net Core 3.0부터 ZipArchiveEntry.Open() 메서드는 로컬 헤더 및 중앙 헤더에서 항목의 압축된 크기와 압축되지 않은 크기가 일치하는지 확인합니다. 보관 파일의 로컬 헤더 및/또는 데이터 설명자가 zip 파일의 중앙 디렉터리와 일치하지 않는 크기를 나열하는 경우 메서드가 InvalidDataException을 throw합니다. 항목을 읽을 때 압축 해제된 데이터가 헤더에 나열된 압축되지 않은 파일 크기로 잘립니다.

이러한 변경은 ZipArchiveEntry가 해당 데이터의 크기를 정확하게 표시하고 해당 양의 데이터만 읽도록 하기 위함입니다.

도입된 버전

3.0

권장 작업

이러한 문제가 발생하는 zip 보관 파일을 다시 패키지합니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- ZipArchiveEntry.Open()
- ZipFileExtensions.ExtractToDirectory
- ZipFileExtensions.ExtractToFile
- ZipFile.ExtractToDirectory

FieldInfo.SetValue가 초기화 전용 정적 필드에 대해 예외를 throw

.NET Core 3.0부터 InitOnly를 호출하여 정적 System.Reflection.FieldInfo.SetValue 필드에 값을 설정하려고 하면 예외가 throw됩니다.

변경 내용 설명

.NET Framework 및 3.0 이전 버전의 .NET Core에서는 을 호출하여 초기화 후 상수인 정적 필드의 값을 설정할 수 있습니다(System.Reflection.FieldInfo.SetValue). 그러나 이러한 필드를 이 방식으로 설정하면 대상 프레임워크 및 최적화 설정에 따라 예기치 않은 동작이 발생합니다.

.NET Core 3.0 이상 버전에서 정적 SetValue 필드에 InitOnly를 호출하면 System.FieldAccessException 예외가 throw됩니다.

> **팁**
>
> InitOnly 필드는 선언되는 시점에 또는 포함하는 클래스의 생성자에서만 설정할 수 있는 필드입니다. 즉, 초기화 후에는 상수입니다.

도입된 버전

3.0

권장 작업

정적 생성자에서 정적 InitOnly 필드를 초기화합니다. 이는 동적 형식과 비동적 형식 모두에 적용됩니다.

또는 필드에서 FieldAttributes.InitOnly 특성을 제거한 다음 FieldInfo.SetValue를 호출할 수 있습니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- FieldInfo.SetValue(Object, Object)

- FieldInfo.SetValue(Object, Object, BindingFlags, Binder, CultureInfo)

IEnumerable<T>를 사용하는 확장 메서드에 GroupCollection을 전달하려면 명확성 필요

`IEnumerable<T>`에서 GroupCollection를 사용하는 확장 메서드를 호출하는 경우 캐스트를 사용하여 형식을 명확하게 해야 합니다.

변경 내용 설명

.NET Core 3.0부터 System.Text.RegularExpressions.GroupCollection은 구현하는 `IEnumerable<KeyValuePair<String,Group>>`을 비롯한 기타 형식 외에 `IEnumerable<Group>`도 구현합니다. 따라서 IEnumerable<T>를 사용하는 확장 메서드를 호출할 때 모호성이 발생합니다. GroupCollection 인스턴스에서 이러한 확장 메서드(예: Enumerable.Count)를 호출하는 경우 다음과 같은 컴파일러 오류가 표시됩니다.

CS1061: 'GroupCollection'에 'Count'에 대한 정의가 없고 'GroupCollection' 형식의 첫 번째 인수를 허용하는 액세스 가능한 확장 메서드 'Count'가 없습니다. using 지시문 또는 어셈블리 참조가 있는지 확인하세요.

이전 버전의 .NET에서는 모호성이 없어서 컴파일러 오류가 발생하지 않았습니다.

도입된 버전

3.0

변경 이유

이는 호환성이 손상되는 의도치 않은 변경 이었습니다. 이와 같은 상황은 한동안 지속되어 왔으므로 되돌릴 계획이 없습니다. 또한 되돌리는 변경 자체로도 호환성이 손상될 수 있습니다.

권장 작업

GroupCollection 인스턴스의 경우 `IEnumerable<T>`를 허용하는 확장 메서드의 호출을 캐스트를 사용하여 명확하게 합니다.

C#

```
// Without a cast - causes CS1061.
match.Groups.Count(_ => true)
```

```
// With a disambiguating cast.
((IEnumerable<Group>)m.Groups).Count(_ => true);
```

범주

핵심 .NET 라이브러리

영향을 받는 API

IEnumerable<T>를 허용하는 모든 확장 메서드가 영향을 받습니다. 예시:

- System.Collections.Immutable.ImmutableArray.ToImmutableArray<TSource>(IEnumerable<TSource>)
- System.Collections.Immutable.ImmutableDictionary.ToImmutableDictionary
- System.Collections.Immutable.ImmutableHashSet.ToImmutableHashSet
- System.Collections.Immutable.ImmutableList.ToImmutableList<TSource>(IEnumerable<TSource>)
- System.Collections.Immutable.ImmutableSortedDictionary.ToImmutableSortedDictionary
- System.Collections.Immutable.ImmutableSortedSet.ToImmutableSortedSet
- System.Data.DataTableExtensions.CopyToDataTable
- 대부분의 `System.Linq.Enumerable` 메서드. 예: System.Linq.Enumerable.Count
- System.Linq.ParallelEnumerable.AsParallel
- System.Linq.Queryable.AsQueryable

암호화

- Linux에서 더 이상 지원되지 않는 신뢰할 수 있는 인증서 구문 시작
- EnvelopedCms를 기본적으로 AES-256 암호화로 설정
- RSAOpenSsl 키 생성 최소 크기가 증가
- .NET Core 3.0은 OpenSSL 1.0.x보다 OpenSSL 1.1.x를 권장합니다.
- CryptoStream.Dispose는 쓰는 경우에만 최종 블록을 변환함

Linux에서 루트 인증서에 대해 더 이상 지원되지 않는 "BEGIN TRUSTED CERTIFICATE" 구문

Linux 및 기타 Unix 유사 시스템(macOS 제외)의 루트 인증서는 표준 `BEGIN CERTIFICATE` PEM 헤더와 OpenSSL별 `BEGIN TRUSTED CERTIFICATE` PEM 헤더의 두 가지 형태로 제공될 수 있습니다. 후자 구문은 .NET Core의 System.Security.Cryptography.X509Certificates.X509Chain 클래스와의

호환성 문제가 발생한 추가 구성을 허용합니다. .NET Core 3.0부터 체인 엔진이 `BEGIN TRUSTED CERTIFICATE` 루트 인증서 콘텐츠를 더 이상 로드하지 않습니다.

변경 내용 설명

이전에는 `BEGIN CERTIFICATE` 및 `BEGIN TRUSTED CERTIFICATE` 구문이 모두 루트 신뢰 목록을 채우는 데 사용되었습니다. `BEGIN TRUSTED CERTIFICATE` 구문이 사용되고 파일에 추가 옵션이 지정된 경우 X509Chain에서 체인 트러스트가 명시적으로 허용되지 않았음을 보고했을 수 있습니다(X509ChainStatusFlags.ExplicitDistrust). 그러나 인증서가 이전에 로드된 파일에서 `BEGIN CERTIFICATE` 구문으로 지정된 경우에는 체인 트러스트가 허용되었습니다.

.NET Core 3.0부터 `BEGIN TRUSTED CERTIFICATE` 콘텐츠를 더 이상 읽지 않습니다. 표준 `BEGIN CERTIFICATE` 구문을 통해서도 인증서가 지정되지 않은 경우 X509Chain은 루트를 신뢰할 수 없음으로 보고합니다(X509ChainStatusFlags.UntrustedRoot).

도입된 버전

3.0

권장 작업

대부분의 애플리케이션은 이러한 변경의 영향을 받지 않지만 권한 문제로 인해 루트 인증서 원본을 모두 볼 수 없는 애플리케이션은 업그레이드 후 예기치 않은 `UntrustedRoot` 오류가 발생할 수 있습니다.

많은 Linux 배포(또는 배포판)는 루트 인증서를 파일당 하나의 인증서 디렉터리 및 하나의 파일 연결의 두 위치에 씁니다. 일부 배포판에서는 파일당 하나의 인증서 디렉터리가 `BEGIN TRUSTED CERTIFICATE` 구문을 사용하는 반면 파일 연결은 표준 `BEGIN CERTIFICATE` 구문을 사용합니다. 사용자 지정 루트 인증서를 이러한 위치 중 하나 이상에 `BEGIN CERTIFICATE`로 추가하고 두 위치를 모두 애플리케이션에서 읽을 수 있는지 확인합니다.

일반적인 디렉터리는 */etc/ssl/certs/*이고 일반적인 연결 파일은 */etc/ssl/cert.pem*입니다. `openssl version -d` 명령을 사용하여 */etc/ssl/*과 다를 수 있는 플랫폼별 루트를 확인합니다. 예를 들어 Ubuntu 18.04에서 디렉터리는 */usr/lib/ssl/certs/*이고 파일은 */usr/lib/ssl/cert.pem*입니다. 그러나 */usr/lib/ssl/certs/*는 */etc/ssl/certs/*에 대한 symlink이며 */usr/lib/ssl/cert.pem*은 존재하지 않습니다.

Bash

```
$ openssl version -d
OPENSSLDIR: "/usr/lib/ssl"
$ ls -al /usr/lib/ssl
```

```
total 12
drwxr-xr-x  3 root root 4096 Dec 12 17:10 .
drwxr-xr-x 73 root root 4096 Feb 20 15:18 ..
lrwxrwxrwx  1 root root   14 Mar 27  2018 certs -> /etc/ssl/certs
drwxr-xr-x  2 root root 4096 Dec 12 17:10 misc
lrwxrwxrwx  1 root root   20 Nov 12 16:58 openssl.cnf -> /etc/ssl/openssl.cnf
lrwxrwxrwx  1 root root   16 Mar 27  2018 private -> /etc/ssl/private
```

범주

암호화

영향을 받는 API

- System.Security.Cryptography.X509Certificates.X509Chain

EnvelopedCms를 기본적으로 AES-256 암호화로 설정

`EnvelopedCms`에서 사용하는 기본 대칭 암호화 알고리즘이 TripleDES에서 AES-256으로 변경되었습니다.

변경 내용 설명

이전 버전에서는 생성자 오버로드를 통해 대칭 암호화 알고리즘을 지정하지 않고 EnvelopedCms가 데이터를 암호화하는 데 사용되는 경우 데이터가 TripleDES/3DES/3DEA/DES3-EDE 알고리즘을 사용하여 암호화됩니다.

.NET Core 3.0부터(System.Security.Cryptography.Pkcs NuGet 패키지의 버전 4.6.0을 통해) 기본 알고리즘이 현대화 알고리즘용 AES-256으로 변경되어 보안 기본 옵션을 개선합니다. 메시지 받는 사람 인증서에 (EC가 아닌) Diffie-Hellman 공개 키가 있는 경우 기본 플랫폼의 제한 사항으로 인해 CryptographicException에서 암호화 작업이 실패할 수 있습니다.

다음 샘플 코드에서는 .NET Core 2.2 이전 버전에서 실행되는 경우 데이터가 TripleDES로 암호화됩니다. .NET Core 3.0 이상에서 실행되는 경우 데이터가 AES-256으로 암호화됩니다.

C#

```
EnvelopedCms cms = new EnvelopedCms(content);
cms.Encrypt(recipient);
return cms.Encode();
```

도입된 버전

3.0

권장 작업

이 변경으로 부정적인 영향을 받는 경우 다음과 같이 EnvelopedCms 형식의 매개 변수를 포함하는 AlgorithmIdentifier 생성자에서 암호화 알고리즘 식별자를 명시적으로 지정하여 TripleDES 암호화를 복원할 수 있습니다.

C#

```
Oid tripleDesOid = new Oid("1.2.840.113549.3.7", null);
AlgorithmIdentifier tripleDesIdentifier = new AlgorithmIdentifier(tripleDesOid);
EnvelopedCms cms = new EnvelopedCms(content, tripleDesIdentifier);

cms.Encrypt(recipient);
return cms.Encode();
```

범주

암호화

영향을 받는 API

- EnvelopedCms()
- EnvelopedCms(ContentInfo)
- EnvelopedCms(SubjectIdentifierType, ContentInfo)

RSAOpenSsl 키 생성 최소 크기가 증가

Linux에서 새 RSA 키를 생성하기 위한 최소 크기가 384비트에서 512비트로 증가했습니다.

변경 내용 설명

.NET Core 3.0부터는 Linux의 `LegalKeySizes`, RSA.Create 및 RSAOpenSsl에서 RSA 인스턴스의 RSACryptoServiceProvider 속성이 보고하는 합법적인 최소 키 크기가 384에서 512로 증가했습니다.

따라서 .NET Core 2.2 버전 이하에서는 `RSA.Create(384)` 같은 메서드 호출이 성공합니다. .Net Core 3.0 버전 이상에서 메서드 호출 `RSA.Create(384)`는 크기가 너무 작음을 나타내는 예외를

throw합니다.

이 변경은 Linux에서 암호화 작업을 수행하는 OpenSSL이 버전 1.0.2와 1.1.0 간에 최소값을 상향했기 때문에 적용된 것입니다. .NET Core 3.0은 OpenSSL 1.0.x보다 1.1.x를 기본으로 적용하고, 이 높아진 새 종속성 제한을 반영하기 위해 보고된 최소 버전이 상향되었습니다.

도입된 버전

3.0

권장 작업

영향을 받는 API를 호출하는 경우 생성된 키의 크기가 공급자 최소값보다 작지 않은지 확인하세요.

> **참고**
>
> 384비트 RSA는 이미 안전하지 않은 것으로 간주됩니다(512비트 RSA도 마찬가지). NIST 특별 간행물 800-57 1부 수정 버전 4 와 같은 최신 권장 사항은 새로 생성된 키에 대한 최소 크기로 2048 비트를 제안합니다.

범주

암호화

영향을 받는 API

- AsymmetricAlgorithm.LegalKeySizes
- RSA.Create
- RSAOpenSsl
- RSACryptoServiceProvider

.NET Core 3.0은 OpenSSL 1.0.x보다 OpenSSL 1.1.x를 권장합니다.

여러 Linux 배포판에서 작동하는 Linux용 .NET Core는 OpenSSL 1.0.x와 OpenSSL 1.1.x를 모두 지원합니다. .NET Core 2.1 및 .NET Core 2.2는 먼저 1.0.x를 찾은 다음 1.1.x로 대체합니다. .NET Core 3.0은 1.1.x부터 찾습니다. 이 변경은 새로운 암호화 표준에 대한 지원을 추가하기 위해 실시된 것입니다.

이 변경이 .NET Core에서 OpenSSL 특정 상호 작용 형식으로 플랫폼 상호 작용을 수행하는 라이브러리나 애플리케이션에 영향을 줄 수 있습니다.

변경 내용 설명

.NET Core 2.2 버전 이하에서 런타임은 OpenSSL 1.0.x보다 1.1.x를 우선적으로 로드합니다. 이는 OpenSSL과 상호 작용을 위한 IntPtr 및 SafeHandle 형식이 기본적으로 libcrypto.so.1.0.0 / libcrypto.so.1.0 / libcrypto.so.10에 사용된다는 의미입니다.

.Net Core 3.0부터 런타임은 OpenSSL 1.0.x보다 OpenSSL 1.1.x를 우선적으로 로드하므로 OpenSSL과 상호 작용을 위한 IntPtr 및 SafeHandle 형식이 기본적으로 ibcrypto.so.1.1 / libcrypto.so.11 / libcrypto.so.1.1.0 / libcrypto.so.1.1.1에 사용됩니다. 따라서 OpenSSL과 직접 상호 운용되는 라이브러리 및 애플리케이션은 .NET Core 2.1 또는 .NET Core 2.2에서 업그레이드 할 경우 .NET Core에 노출된 값과 호환되지 않는 포인터를 가질 수 있습니다.

도입된 버전

3.0

권장 작업

OpenSSL을 사용하여 직접 작업을 수행하는 라이브러리 및 애플리케이션은 .NET Core 런타임과 동일한 버전의 OpenSSL을 사용하고 있는지 확인해야 합니다.

OpenSSL을 사용하여 직접 .NET Core 암호화 형식에서 IntPtr 또는 SafeHandle 값을 사용하는 모든 라이브러리 또는 애플리케이션은사용하는 라이브러리 버전을 새 SafeEvpPKeyHandle.OpenSslVersion 속성과 비교하여 포인터가 호환되는지 확인 해야 합니다.

범주

암호화

영향을 받는 API

- SafeEvpPKeyHandle
- RSAOpenSsl(IntPtr)
- RSAOpenSsl(SafeEvpPKeyHandle)
- RSAOpenSsl.DuplicateKeyHandle()
- DSAOpenSsl(IntPtr)
- DSAOpenSsl(SafeEvpPKeyHandle)
- DSAOpenSsl.DuplicateKeyHandle()

- ECDsaOpenSsl(IntPtr)
- ECDsaOpenSsl(SafeEvpPKeyHandle)
- ECDsaOpenSsl.DuplicateKeyHandle()
- ECDiffieHellmanOpenSsl(IntPtr)
- ECDiffieHellmanOpenSsl(SafeEvpPKeyHandle)
- ECDiffieHellmanOpenSsl.DuplicateKeyHandle()
- X509Certificate.Handle

CryptoStream.Dispose는 쓰는 경우에만 최종 블록을 변환함

CryptoStream.Dispose 연산을 완료하기 위해 사용되는 `CryptoStream` 메서드는 읽을 때 더 이상 최종 블록을 변환하려고 시도하지 않습니다.

변경 내용 설명

이전 .NET 버전에서는 사용자가 CryptoStream 모드에서 Read을 사용할 때 불완전한 읽기를 수행하는 경우 Dispose 메서드가 예외를 throw할 수 있습니다(예: 패딩이 포함된 AES를 사용할 경우). 최종 블록을 변환하려고 시도했지만 데이터가 불완전하기 때문에 예외가 throw되었습니다.

.NET Core 3.0 이상 버전에서 Dispose는 읽을 때 더 이상 최종 블록을 변환(불완전한 읽기가 가능함)하려고 시도하지 않습니다.

변경 이유

이렇게 변경하면 네트워크 작업이 취소되는 경우 예외를 catch하지 않고도 암호화 스트림에서 불완전한 읽기가 가능합니다.

도입된 버전

3.0

권장 작업

대부분의 앱은 이 변경의 영향을 받지 않습니다.

불완전한 읽기 발생 시 애플리케이션이 이전에 예외를 catch했다면 해당 `catch` 블록을 삭제할 수 있습니다. 해시 시나리오에서 앱이 최종 블록의 변환을 사용한 경우 삭제하기 전에 전체 스트림을 읽었는지 확인해야 할 수도 있습니다.

범주

암호화

영향을 받는 API

- System.Security.Cryptography.CryptoStream.Dispose

Entity Framework Core (엔티티 프레임워크 코어)

Entity Framework Core 호환성이 손상되는 변경 사항

전역화

- “C” 로캘이 고정 로캘에 매핑됩니다.

“C” 로캘이 고정 로캘에 매핑됩니다.

.NET Core 2.2 및 이전 버전은 "C" 로캘을 en_US_POSIX 로캘에 매핑하는 기본 ICU 동작에 의존합니다. en_US_POSIX 로캘은 대/소문자를 구분하지 않는 문자열 비교를 지원하지 않으므로 원치 않는 데이터 정렬 동작이 있습니다. 일부 Linux 배포판에서는 "C" 로캘을 기본 로캘로 설정하므로 사용자가 예기치 않은 동작을 경험했습니다.

변경 내용 설명

.NET Core 3.0부터 "C" 로캘 매핑이 en_US_POSIX 대신 고정 로캘을 사용하도록 변경되었습니다. "C" 로캘의 고정 로캘 매핑은 일관성을 위해 Windows에도 적용됩니다.

en_US_POSIX는 대/소문자를 구분하지 않는 정렬/검색 문자열 작업을 지원하지 않기 때문에 "C"를 en_US_POSIX 문화권에 매핑 시 고객의 혼란이 발생했습니다. "C" 로캘이 일부 Linux 배포판에서 기본 로캘로 사용되므로 고객이 이러한 운영 체제에서 이 원치 않는 동작을 경험했습니다.

도입된 버전

3.0

권장 작업

이 변경 내용을 아는 것 외에 특별히 다른 것은 없습니다. 이러한 변경 내용은 "C" 로캘을 사용하는 애플리케이션에만 영향을 줍니다.

범주

전역화

영향을 받는 API

모든 데이터 정렬 및 문화권 API는 이 변경 내용의 영향을 받습니다.

MSBuild

- 리소스 매니페스트 파일 이름 변경

리소스 매니페스트 파일 이름 변경

.NET Core 3.0부터 기본 사례에서 MSBuild는 리소스 파일에 대해 다른 매니페스트 파일 이름을 생성합니다.

도입된 버전

3.0

변경 내용 설명

.NET Core 3.0보다 이전 버전에서는 프로젝트 파일의 `LogicalName` 항목에 대해 `ManifestResourceName`, `DependentUpon` 또는 `EmbeddedResource` 메타데이터가 지정되지 않은 경우 MSBuild는 `<RootNamespace>.<ResourceFilePathFromProjectRoot>.resources` 패턴으로 매니페스트 파일 이름을 생성했습니다. 프로젝트 파일에 `RootNamespace`가 정의되어 있지 않은 경우 기본적으로 프로젝트 이름으로 설정됩니다. 예를 들어, 루트 프로젝트 디렉터리의 *Form1.resx*라는 리소스 파일에 대해 생성된 매니페스트 이름은 *MyProject.Form1.resources*였습니다.

.NET Core 3.0부터 리소스 파일이 동일한 이름의 소스 파일과 공동 배치되는 경우(예: *Form1.resx*와 *Form1.cs*) MSBuild는 소스 파일의 형식 정보를 사용하여 `<Namespace>.<ClassName>.resources` 패턴으로 매니페스트 파일 이름을 생성합니다. 네임스페이스 및 클래스 이름은 공동 배치된 소스 파일의 첫 번째 형식에서 추출됩니다. 예를 들어 *Form1.cs*라는 소스 파일과 공동 배치된 *Form1.resx*라는 리소스 파일에 대해 생성된 매니페스트 이름은 *MyNamespace.Form1.resources*

입니다. 중요한 점은 파일 이름의 첫 부분이 이전 버전의 .NET Core와 다르다는 것입니다 (*MyProject*가 아니라 *MyNamespace*).

> **참고**
>
> 프로젝트 파일의 `LogicalName` 항목에 `ManifestResourceName`, `DependentUpon` 또는 `EmbeddedResource` 메타데이터가 지정되어 있는 경우 이 변경 내용은 해당 리소스 파일에 영향을 주지 않습니다.

이 주요 변경 내용은 .NET Core 프로젝트에 `EmbeddedResourceUseDependentUponConvention` 속성을 추가하여 도입되었습니다. 기본적으로 리소스 파일은 .NET Core 프로젝트 파일에 명시적으로 나열되지 않으므로 생성된 `DependentUpon` 파일의 이름을 지정하는 방법을 지정하는 메타데이터가 없습니다. `EmbeddedResourceUseDependentUponConvention`이 기본값 `true`로 설정된 경우 MSBuild는 공동 배치된 소스 파일을 찾아 해당 파일에서 네임스페이스 및 클래스 이름을 추출합니다. `EmbeddedResourceUseDependentUponConvention`을 `false`로 설정하면 MSBuild는 이전 동작에 따라, 즉 `RootNamespace`와 상대 파일 경로를 결합하여 매니페스트 이름을 생성합니다.

권장 작업

대부분의 경우 개발자에게는 아무 작업도 필요하지 않으며 앱은 계속 작동할 것입니다. 그러나 이 변경으로 앱이 중단되는 경우 다음 중 하나를 수행할 수 있습니다.

- 새 매니페스트 이름을 예상하도록 코드를 변경합니다.
- 프로젝트 파일에서 `EmbeddedResourceUseDependentUponConvention`을 `false`로 설정하여 새 명명 규칙을 옵트아웃합니다.

```xml
<PropertyGroup>

<EmbeddedResourceUseDependentUponConvention>false</EmbeddedResourceUseDependentUponConvention>
</PropertyGroup>
```

범주

MSBuild

영향을 받는 API

해당 없음

네트워킹

- HttpRequestMessage.Version의 기본값은 1.1로 변경되었습니다.

HttpRequestMessage.Version의 기본값은 1.1로 변경되었습니다.

System.Net.Http.HttpRequestMessage.Version 속성의 기본값은 2.0에서 1.1로 변경되었습니다.

도입된 버전

3.0

변경 내용 설명

.NET Core 1.0 ~ 2.0에서 System.Net.Http.HttpRequestMessage.Version 속성의 기본값은 1.1입니다. .NET Core 2.1부터 2.1로 변경되었습니다.

.NET Core 3.0부터 System.Net.Http.HttpRequestMessage.Version 속성에 의해 반환되는 기본 버전 번호는 다시 1.1입니다.

권장 작업

2.0 기본값을 반환하는 System.Net.Http.HttpRequestMessage.Version 속성에 따라 달라지는 경우 코드를 업데이트합니다.

범주

네트워킹

영향을 받는 API

- System.Net.Http.HttpRequestMessage.Version

윈도우 폼즈 (Windows Forms)

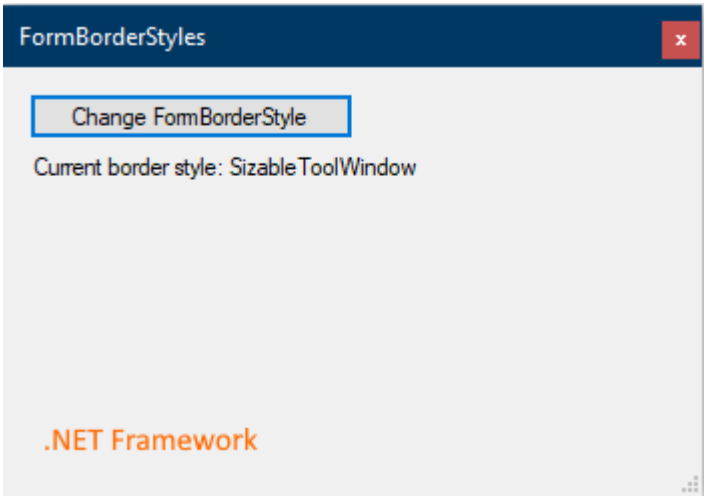
- Control.DefaultFont가 Segoe UI 9 pt로 변경되었습니다
- FolderBrowserDialog의 현대화
- 일부 Windows Forms 형식에서 SerializableAttribute가 제거됨
- AllowUpdateChildControlIndexForTabControls 호환성 스위치는 지원되지 않습니다.
- DomainUpDown.UseLegacyScrolling 호환성 스위치는 지원되지 않습니다
- DoNotLoadLatestRichEditControl 호환성 스위치는 지원되지 않습니다
- DoNotSupportSelectAllShortcutInMultilineTextBox 호환성 스위치는 지원되지 않습니다
- DontSupportReentrantFilterMessage 호환성 스위치는 지원되지 않습니다
- EnableVisualStyleValidation 호환성 스위치는 지원되지 않습니다
- UseLegacyContextMenuStripSourceControlValue 호환성 스위치는 지원되지 않습니다
- UseLegacyImages 호환성 스위치가 지원되지 않습니다
- Visual Basic의 About 및 SplashScreen 템플릿이 손상되었습니다.

기본 컨트롤 글꼴이 Segoe UI 9 pt로 변경됨

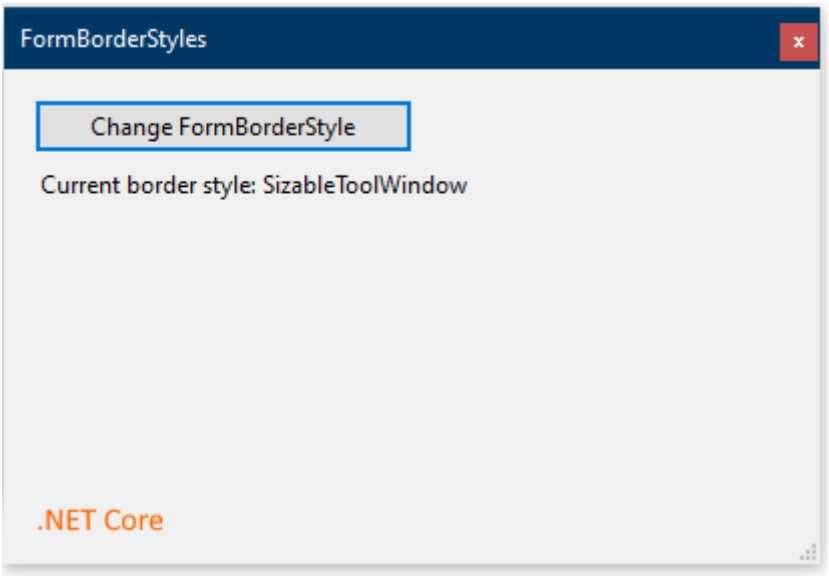
변경 내용 설명

.NET Framework에서 Control.DefaultFont 속성은 `Microsoft Sans Serif 8.25 pt`으로 설정되었습니다. 다음 이미지는 기본 글꼴을 사용하는 창을 보여줍니다.

.NET Framework의 기본 컨트롤 글꼴



.NET Core 3.0부터 기본 글꼴은 `Segoe UI 9 pt`(SystemFonts.MessageBoxFont동일한 글꼴)로 설정됩니다. 이러한 변경으로 인해 폼과 컨트롤의 크기가 약 27% 더 커져, 새 기본 글꼴의 크기 증가를 반영합니다. 예시:



이 변경은 Windows 사용자 경험(UX) 지침에 맞춰졌습니다.

도입된 버전

3.0

권장 작업

양식 및 컨트롤의 크기가 변경되어 애플리케이션이 올바르게 렌더링되는지 확인합니다.

단일 폼의 원래 글꼴을 유지하려면 기본 글꼴을 `Microsoft Sans Serif 8.25 pt`으로 설정하세요. 예시:

C#

```
public MyForm()
{
    InitializeComponent();
    Font = new Font(new FontFamily("Microsoft Sans Serif"), 8.25f);
}
```

또는 다음 방법 중 하나를 사용하여 전체 애플리케이션의 기본 글꼴을 변경할 수 있습니다.

- `ApplicationDefaultFont` MSBuild 속성을 "Microsoft Sans Serif, 8.25pt"로 설정합니다. Visual Studio에서 디자이너의 새 설정을 사용할 수 있으므로 이 방법을 사용하는 것이 좋습니다.

  XML

  ```
  <PropertyGroup>
    <ApplicationDefaultFont>Microsoft Sans Serif,
  ```

```
8.25pt</ApplicationDefaultFont>
</PropertyGroup>
```

- Application.SetDefaultFont(Font)을 호출하여.

```
C#

class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.SetHighDpiMode(HighDpiMode.SystemAware);
        Application.SetDefaultFont(new Font(new FontFamily("Microsoft Sans
Serif"), 8.25f));
        Application.Run(new Form1());
    }
}
```

범주

- 윈도우 폼즈 (Windows Forms)

영향을 받는 API

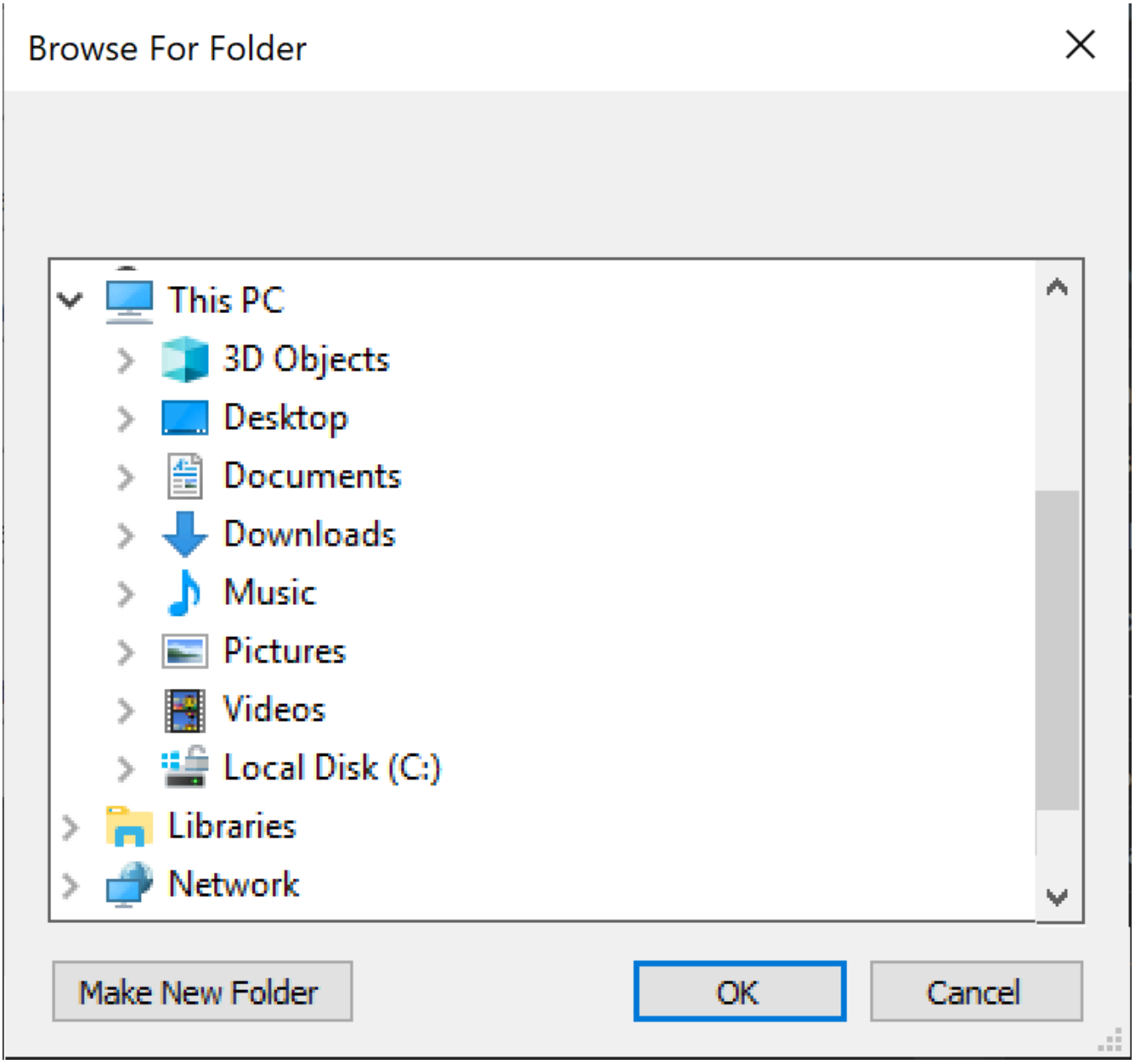
없음

FolderBrowserDialog의 현대화

.NET Core용 Windows Forms 애플리케이션에서 FolderBrowserDialog 컨트롤이 변경되었습니다.

변경 내용 설명

.NET Framework에서 Windows Forms는 FolderBrowserDialog 컨트롤에 다음 대화 상자를 사용합니다.



.NET Core 3.0에서 Windows Forms는 Windows Vista에서 도입된 최신 COM 기반 컨트롤을 사용합니다.

.NET Core에서의 FolderBrowserDialogControl

도입된 버전

3.0

권장 작업

대화 상자가 자동으로 업그레이드됩니다.

원래 대화 상자를 유지하려면 다음 코드 조각과 같이 대화 상자를 표시하기 전에 FolderBrowserDialog.AutoUpgradeEnabled 속성을 `false` 설정합니다.

C#

```
var dialog = new FolderBrowserDialog();
dialog.AutoUpgradeEnabled = false;
dialog.ShowDialog();
```

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- FolderBrowserDialog

Windows Forms의 일부 형식에서 SerializableAttribute 제거됨

일부 Windows Forms 클래스에서, 알려진 이진 직렬화 시나리오가 없는 경우에 SerializableAttribute이 제거되었습니다.

변경 내용 설명

다음 형식들은 .NET Framework에서는 SerializableAttribute으로 데코레이팅되었지만, .NET Core에서는 이 특성이 제거되었습니다.

- `System.InvariantComparer`
- System.ComponentModel.Design.ExceptionCollection
- System.ComponentModel.Design.Serialization.CodeDomSerializerException
- `System.ComponentModel.Design.Serialization.CodeDomComponentSerializationService.CodeDomSerializationStore`
- System.Drawing.Design.ToolboxItem
- `System.Resources.ResXNullRef`
- `System.Resources.ResXDataNode`
- `System.Resources.ResXFileRef`
- System.Windows.Forms.Cursor
- `System.Windows.Forms.NativeMethods.MSOCRINFOSTRUCT`
- `System.Windows.Forms.NativeMethods.MSG`

지금까지 이 serialization 메커니즘에는 심각한 유지 관리 및 보안 문제가 있었습니다. 형식에 대한 `SerializableAttribute` 유지 관리한다는 것은 해당 형식이 버전 간 직렬화 변경 및 잠재적으로 프레임워크 간 직렬화 변경에 대해 테스트되어야 한다는 것을 의미합니다. 이렇게 하면 이러

한 형식을 발전하기가 더 어려워지고 유지 관리 비용이 많이 들 수 있습니다. 이러한 형식에는 알려진 이진 직렬화 시나리오가 없으므로 속성을 제거해도 영향이 최소화됩니다.

자세한 내용은 이진 직렬화을 참조하세요.

도입된 버전

3.0

권장 작업

직렬화 가능으로 표시된 이러한 형식에 의존하는 코드를 업데이트하십시오.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- 없음

AllowUpdateChildControlIndexForTabControls 호환성 스위치가 지원되지 않음

`Switch.System.Windows.Forms.AllowUpdateChildControlIndexForTabControls` 호환성 스위치는 .NET Framework 4.6 이상 버전의 Windows Forms에서 지원되지만 .NET Core 또는 .NET 5.0 이상에서는 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.6 이상 버전에서 탭을 선택하면 컨트롤 컬렉션이 다시 정렬됩니다. `Switch.System.Windows.Forms.AllowUpdateChildControlIndexForTabControls` 호환성 스위치를 사용하면 이 동작이 바람직하지 않은 경우 애플리케이션에서 이 순서를 건너뛸 수 있습니다.

.NET Core 및 .NET 5.0 이상에서는 `Switch.System.Windows.Forms.AllowUpdateChildControlIndexForTabControls` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- 없음

DomainUpDown.UseLegacyScrolling 호환성 스위치가 지원되지 않습니다.

.NET Framework 4.7.1에서 도입된 `Switch.System.Windows.Forms.DomainUpDown.UseLegacyScrolling` 호환성 스위치는 .NET Core 또는 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.7.1부터 `Switch.System.Windows.Forms.DomainUpDown.UseLegacyScrolling` 호환성 스위치를 통해 개발자는 독립적인 DomainUpDown.DownButton() 및 DomainUpDown.UpButton() 작업에서 옵트아웃할 수 있습니다. 스위치는 컨텍스트 텍스트가 있는 경우 DomainUpDown.UpButton() 무시되는 레거시 동작을 복원했으며, 개발자는 DomainUpDown.DownButton() 작업 전에 컨트롤에서 DomainUpDown.UpButton() 작업을 사용해야 합니다. 더 자세한 정보를 보려면 <AppContextSwitchOverrides> 요소를 참조하시기 바랍니다.

.NET Core 및 .NET 5.0 이상에서는 `Switch.System.Windows.Forms.DomainUpDown.UseLegacyScrolling` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- DomainUpDown.DownButton()
- DomainUpDown.UpButton()

DoNotLoadLatestRichEditControl 호환성 스위치가 지원되지 않음

.NET Framework 4.7.1에서 도입된 `Switch.System.Windows.Forms.UseLegacyImages` 호환성 스위치는 .NET Core 또는 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.6.2 및 이전 버전에서 RichTextBox 컨트롤은 Win32 RichEdit 컨트롤 v3.0을 인스턴스화하고.NET Framework 4.7.1을 대상으로 하는 애플리케이션의 경우 RichTextBox 컨트롤은 RichEdit v4.1(*msftedit.dll*)을 인스턴스화합니다.
`Switch.System.Windows.Forms.DoNotLoadLatestRichEditControl` 호환성 스위치는 .NET Framework 4.7.1 이상 버전을 대상으로 하는 애플리케이션이 새 RichEdit v4.1 컨트롤을 옵트아웃하고 이전 RichEdit v3 컨트롤을 대신 사용할 수 있도록 하기 위해 도입되었습니다.

.NET Core 및 .NET 5.0 이상 버전에서는 `Switch.System.Windows.Forms.DoNotLoadLatestRichEditControl` 스위치가 지원되지 않습니다. RichTextBox 컨트롤의 새 버전만 지원됩니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- System.Windows.Forms.RichTextBox

DoNotSupportSelectAllShortcutInMultilineTextBox 호환성 전환이 지원되지 않음

.NET Framework 4.6.1에서 도입된 `Switch.System.Windows.Forms.DoNotSupportSelectAllShortcutInMultilineTextBox` 호환성 스위치는 .NET Core 및 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.6.1부터 `Ctrl` + `A` 바로 가기 키를 선택하면 TextBox 컨트롤에서 모든 텍스트가 선택되도록 설정되었습니다. .NET Framework 4.6 및 이전 버전에서는 `Textbox.ShortcutsEnabled`와 + 속성이 모두 ○로 설정된 경우, CtrlTextBox.Multiline `true` 바로 가기 키를 사용해도 모든 텍스트가 선택되지 않았습니다.
`Switch.System.Windows.Forms.DoNotSupportSelectAllShortcutInMultilineTextBox` 호환성 스위치는 원래 동작을 유지하기 위해 .NET Framework 4.6.1에 도입되었습니다. 자세한 내용은 TextBox.ProcessCmdKey참조하세요.

.NET Core 및 .NET 5.0 이상 버전에서는 `Switch.System.Windows.Forms.DoNotSupportSelectAllShortcutInMultilineTextBox` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- 없음

DontSupportReentrantFilterMessage 호환성 스위치가 지원되지 않음

.NET Framework 4.6.1에서 도입된 `Switch.System.Windows.Forms.DontSupportReentrantFilterMessage` 호환성 스위치는 .NET Core 및 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.6.1부터 `Switch.System.Windows.Forms.DontSupportReentrantFilterMessage` 호환성 스위치는 사용자 지정 IndexOutOfRangeException 구현을 사용하여 Application.FilterMessage 메시지를 호출할 때 가능한 IMessageFilter.PreFilterMessage 예외를 해결합니다. 자세한 내용은 문제 해결: 사용자 지정 IMessageFilter.PreFilterMessage 구현을 참조하세요.

.NET Core 및 .NET 5.0 이상에서는 `Switch.System.Windows.Forms.DontSupportReentrantFilterMessage` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- Application.FilterMessage

EnableVisualStyleValidation 호환성 스위치가 지원되지 않음

`Switch.System.Windows.Forms.EnableVisualStyleValidation` 호환성 스위치는 .NET Core 또는 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework에서 `Switch.System.Windows.Forms.EnableVisualStyleValidation` 호환성 스위치를 사용하면 애플리케이션이 숫자 형식으로 제공된 시각적 스타일의 유효성 검사를 옵트아웃할 수 있습니다.

.NET Core 및 .NET 5.0 이상에서는 `Switch.System.Windows.Forms.EnableVisualStyleValidation` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- 없음

UseLegacyContextMenuStripSourceControlValue 호환성 전환이 지원되지 않음

.NET Framework 4.7.2에서 도입된 `Switch.System.Windows.Forms.UseLegacyContextMenuStripSourceControlValue` 호환성 스위치는 .NET Core 또는 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.7.2부터 `Switch.System.Windows.Forms.UseLegacyContextMenuStripSourceControlValue` 호환성 스위치를 사용하면 개발자가 소스 제어에 대한 참조를 반환하는 ContextMenuStrip.SourceControl 속성의 새 동작을 옵트아웃할 수 있습니다. 속성의 이전 동작은 `null`반환하는 것이었습니다. 더 자세한 정보를 보려면 <AppContextSwitchOverrides> 요소를 참조하시기 바랍니다.

.NET Core 및 .NET 5.0 이상에서는 `Switch.System.Windows.Forms.UseLegacyContextMenuStripSourceControlValue` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- ContextMenuStrip.SourceControl

UseLegacyImages 호환성 스위치가 지원되지 않음

.NET Framework 4.8에서 도입된 `Switch.System.Windows.Forms.UseLegacyImages` 호환성 스위치는 .NET Core 또는 .NET 5.0 이상의 Windows Forms에서 지원되지 않습니다.

변경 내용 설명

.NET Framework 4.8부터 `Switch.System.Windows.Forms.UseLegacyImages` 호환성 스위치는 높은 DPI 환경의 ClickOnce 시나리오에서 가능한 이미지 크기 조정 문제를 해결했습니다. `true`설정하면 스위치를 사용하여 스케일이 100%을 초과하는 것으로 설정된 DPI 스케일이 높은 디스플레이에서 레거시 이미지 스케일링을 복원할 수 있습니다. 자세한 내용은 GitHub의 .NET Framework 4.8 릴리스 정보 참조하세요.

.NET Core 및 .NET 5.0 이상에서는 `Switch.System.Windows.Forms.UseLegacyImages` 스위치가 지원되지 않습니다.

도입된 버전

3.0

권장 작업

스위치를 제거합니다. 스위치는 지원되지 않으며 대체 기능을 사용할 수 없습니다.

범주

윈도우 폼즈 (Windows Forms)

영향을 받는 API

- 없음

About 및 SplashScreen 템플릿이 손상됨

Visual Studio에서 생성된 `About.vb` 및 `SplashScreen.vb` 파일에는 .NET Core 3.0 및 3.1을 사용할 수 없는 `My` 네임스페이스의 형식에 대한 참조가 포함되어 있습니다.

도입된 버전

3.0

변경 내용 설명

.NET Core 3.0 및 3.1에는 전체 Visual Basic `My` 지원이 포함되어 있지 않습니다. Visual Studio의 Visual Basic Windows Forms 앱에서 **About** 및 **SplashScreen** 양식 템플릿은 사용할 수 없는 `My.Application.Info` 유형의 속성을 참조합니다.

권장 작업

Visual Basic `My` 지원이 .NET 5에서 개선되어 프로젝트를 .NET 5 이상으로 업그레이드했습니다.

-또는-

앱의 **About** 및 **SplashScreen** 유형에서 컴파일러 오류를 수정합니다.
`System.Reflection.Assembly` 클래스를 사용하여 `My.Application.Info` 형식에서 제공하는 정보를 가져옵니다. 두 형태의 직접 포트는 여기에서 사용할 수 있습니다.

> **팁**
>
> 샘플 코드이며 최적이 아닌 코드입니다. 양식 로드 시간을 줄이기 위해 특성 목록을 캐시해야 합니다.

정보

VB

```
Imports System.Reflection

Public NotInheritable Class About

    Private Sub about_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        ' Set the title of the form.
        Dim applicationTitle As String =
Assembly.GetExecutingAssembly().GetCustomAttribute(Of AssemblyTitleAttribute)
()?.Title

        If String.IsNullOrEmpty(applicationTitle) Then
            applicationTitle =
System.IO.Path.GetFileNameWithoutExtension(Assembly.GetExecutingAssembly().GetName
().Name)
        End If

        Me.Text = String.Format("About {0}", applicationTitle)
        ' Initialize all of the text displayed on the About Box.
        ' TODO: Customize the application's assembly information in the
"Application" pane of the project
        '    properties dialog (under the "Project" menu).
        Me.LabelProductName.Text =
If(Assembly.GetExecutingAssembly().GetCustomAttribute(Of AssemblyProductAttribute)
()?.Product, "")
        Me.LabelVersion.Text = String.Format("Version {0}",
Assembly.GetExecutingAssembly().GetName().Version)
        Me.LabelCopyright.Text =
If(Assembly.GetExecutingAssembly().GetCustomAttribute(Of
AssemblyCopyrightAttribute)()?.Copyright, "")
        Me.LabelCompanyName.Text =
If(Assembly.GetExecutingAssembly().GetCustomAttribute(Of AssemblyCompanyAttribute)
()?.Company, "")
        Me.TextBoxDescription.Text =
If(Assembly.GetExecutingAssembly().GetCustomAttribute(Of
AssemblyDescriptionAttribute)()?.Description, "")
    End Sub
```

```
    Private Sub OKButton_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles OKButton.Click
        Me.Close()
    End Sub

End Class
```

스플래시 스크린

VB

```
Imports System.Reflection

Public NotInheritable Class SplashScreen

    Private Sub SplashScreen1_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load
        'Set up the dialog text at runtime according to the application's assembly
information.

        'TODO: Customize the application's assembly information in the
"Application" pane of the project
        '  properties dialog (under the "Project" menu).

        'Application title
        Dim appTitle As String =
Assembly.GetExecutingAssembly().GetCustomAttribute(Of AssemblyTitleAttribute)
()?.Title

        If String.IsNullOrEmpty(appTitle) Then
            appTitle =
System.IO.Path.GetFileNameWithoutExtension(Assembly.GetExecutingAssembly().GetName
().Name)
        End If

        ApplicationTitle.Text = appTitle

        Dim versionValue = Assembly.GetExecutingAssembly().GetName().Version

        'Format the version information using the text set into the Version
control at design time as the
        '  formatting string.  This allows for effective localization if desired.
        '  Build and revision information could be included by using the following
code and changing the
        '  Version control's designtime text to "Version {0}.{1:00}.{2}.{3}" or
something similar.  See
        '  String.Format() in Help for more information.
        '
        '    Version.Text = System.String.Format(Version.Text, versionValue.Major,
versionValue.Minor, versionValue.Build, versionValue.Revision)

        Version.Text = System.String.Format(Version.Text, versionValue.Major,
versionValue.Minor)
```

```
        'Copyright info
        Copyright.Text = If(Assembly.GetExecutingAssembly().GetCustomAttribute(Of
AssemblyCopyrightAttribute)()?.Copyright, "")
    End Sub

End Class
```

범주

비주얼 베이직 윈도우즈 폼 (Visual Basic Windows Forms)

영향을 받는 API

없음

WPF (Windows Presentation Foundation, 윈도우 프레젠테이션 파운데이션)

- 텍스트 편집기에서 변경된 끌어서 놓기 동작

텍스트 편집기에서 변경된 끌어서 놓기 동작

.NET Core 3.0에서는 텍스트를 다른 컨트롤로 끌 때 텍스트 편집기 컨트롤이 System.Windows.DataObject 만드는 방식이 변경되었습니다. 변경으로 인해 자동 변환이 비활성화되어 작업이 데이터를 변환DataFormats.Text하는 대신 그대로 DataFormats.UnicodeText 유지됩니다DataFormats.StringFormat.

도입된 버전

.NET Core 3.0

범주

윈도우즈 프레젠테이션 파운데이션 (Windows Presentation Foundation)

이전 동작

텍스트 편집기 컨트롤에서 텍스트를 끌 때의 데이터 형식 System.Windows.DataObject 은 .입니다 DataFormats.StringFormat.

새 동작

텍스트 편집기 컨트롤에서 텍스트를 끌 때의 데이터 형식 System.Windows.DataObject 은 DataFormats.Text> 입니다DataFormats.UnicodeText.

호환성이 손상되는 변경의 형식

이 변경 사항은 동작 변경입니다.

변경 이유

변경 내용은 의도하지 않았습니다.

권장 작업

이 변경 내용은 .NET 7에서 되돌렸습니다. .NET 7 이상으로 업그레이드합니다.

영향을 받는 API

- System.Windows.DataObject

참고 항목

- .NET Core 3.0의 새로운 기능

.NET Core 2.2의 새로운 기능

아티클 • 2025. 04. 30.

.NET Core 2.2에는 애플리케이션 배포, 런타임 서비스에 대한 이벤트 처리, Azure SQL 데이터베이스에 대한 인증, JIT 컴파일러 성능 및 메서드 실행 전에 코드 삽입의 `Main` 향상된 기능이 포함되어 있습니다.

새 배포 모드

.NET Core 2.2부터는.dll 파일 대신 **.exe** 파일인 **프레임워크 종속 실행 파일을** 배포할 수 있습니다. 프레임워크 종속 배포와 기능적으로 유사하게, 프레임워크 종속 실행 파일(FDE)은 여전히 실행하기 위해 .NET Core 시스템 전역 공유 버전에 의존합니다. 앱에는 코드와 타사 종속성만 포함됩니다. 프레임워크 종속 배포와 달리 FDE는 플랫폼별로 다릅니다.

이 새 배포 모드는 라이브러리 대신 실행 파일을 빌드하는 고유한 장점이 있습니다. 즉, 먼저 호출 `dotnet` 하지 않고 직접 앱을 실행할 수 있습니다.

코어

런타임 서비스에서 이벤트 처리

GC, JIT 및 ThreadPool과 같은 애플리케이션의 런타임 서비스 사용을 모니터링하여 애플리케이션에 미치는 영향을 파악하는 경우가 많습니다. Windows 시스템에서는 일반적으로 현재 프로세스의 ETW 이벤트를 모니터링하여 수행됩니다. 이 작업은 계속 잘 작동하지만 낮은 권한 환경 또는 Linux 또는 macOS에서 실행되는 경우 ETW를 항상 사용할 수 있는 것은 아닙니다.

.NET Core 2.2부터 시작하여, System.Diagnostics.Tracing.EventListener 클래스를 사용하여 CoreCLR 이벤트를 사용할 수 있습니다. 이러한 이벤트는 GC, JIT, ThreadPool 및 interop과 같은 런타임 서비스의 동작을 설명합니다. 이러한 이벤트는 CoreCLR ETW 공급자의 일부로 노출되는 것과 동일한 이벤트입니다. 이렇게 하면 애플리케이션에서 이러한 이벤트를 사용하거나 전송 메커니즘을 사용하여 원격 분석 집계 서비스로 보낼 수 있습니다. 다음 코드 샘플에서 이벤트를 구독하는 방법을 확인할 수 있습니다.

C#

```
internal sealed class SimpleEventListener : EventListener
{
    // Called whenever an EventSource is created.
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        // Watch for the .NET runtime EventSource and enable all of its events.
        if (eventSource.Name.Equals("Microsoft-Windows-DotNETRuntime"))
        {
            EnableEvents(eventSource, EventLevel.Verbose, (EventKeywords)(-1));
```

```
        }
    }

    // Called whenever an event is written.
    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // Write the contents of the event to the console.
        Console.WriteLine($"ThreadID = {eventData.OSThreadId} ID =
{eventData.EventId} Name = {eventData.EventName}");
        for (int i = 0; i < eventData.Payload.Count; i++)
        {
            string payloadString = eventData.Payload[i]?.ToString() ??
string.Empty;
            Console.WriteLine($"\tName = \"{eventData.PayloadNames[i]}\" Value =
\"{payloadString}\"");
        }
        Console.WriteLine("\n");
    }
}
```

또한 .NET Core 2.2는 ETW 이벤트에 대한 추가 정보를 제공하기 위해 EventWrittenEventArgs 클래스에 다음 두 속성을 추가합니다.

- EventWrittenEventArgs.OSThreadId
- EventWrittenEventArgs.TimeStamp

데이터

SqlConnection.AccessToken 속성을 사용하여 Azure SQL 데이터베이스에 대한 AAD 인증

.NET Core 2.2부터 Azure Active Directory에서 발급한 액세스 토큰을 사용하여 Azure SQL 데이터베이스에 인증할 수 있습니다. 액세스 토큰을 지원하기 위해 속성이 AccessToken 클래스에 SqlConnection 추가되었습니다. AAD 인증을 활용하려면 System.Data.SqlClient NuGet 패키지 버전 4.6을 다운로드합니다. 이 기능을 사용하려면 NuGet 패키지에 포함된 Microsoft.IdentityModel.Clients.ActiveDirectory 를 사용하여 액세스 토큰 값을 가져올 수 있습니다.

JIT 컴파일러 개선 사항

계층화된 컴파일은 옵트인 기능으로 유지됩니다.

.NET Core 2.1에서 JIT 컴파일러는 새로운 컴파일러 기술인 *계층화된 컴파일*을 옵트인 기능으로 구현했습니다. 계층화된 컴파일의 목표는 성능 향상입니다. JIT 컴파일러에서 수행하는 중요한 작업 중 하나는 코드 실행을 최적화하는 것입니다. 그러나 거의 사용되지 않는 코드 경로의 경우 컴파일러는 런타임이 최적화되지 않은 코드를 실행하는 데 소요되는 것보다 코드를 최적화하는

데 더 많은 시간을 할애할 수 있습니다. 계층화된 컴파일은 JIT 컴파일에 다음 두 단계를 도입합니다.

- 가능한 한 빨리 코드를 생성하는 **첫 번째 계층**입니다.
- 자주 실행되는 메서드에 대해 최적화된 코드를 생성하는 **두 번째 계층**입니다. 두 번째 컴파일 계층은 성능 향상을 위해 병렬로 수행됩니다.

계층화된 컴파일에서 발생할 수 있는 성능 향상에 대한 자세한 내용은 .NET Core 2.2 미리 보기 2 발표를 참조하세요.

계층화된 컴파일을 옵트인하는 방법에 대한 자세한 내용은 .NET Core 2.1의 새로운 기능에서Jit 컴파일러 개선 사항을 참조하세요.

실행 시간

Main 메서드를 실행하기 전에 코드 삽입

.NET Core 2.2부터 시작 후크를 사용하여 애플리케이션의 Main 메서드를 실행하기 전에 코드를 삽입할 수 있습니다. 시작 후크를 사용하면 호스트가 애플리케이션을 다시 컴파일하거나 변경할 필요 없이 배포된 후 애플리케이션의 동작을 사용자 지정할 수 있습니다.

호스팅 공급자는 동작과 같이 주 진입점의 부하 동작에 영향을 줄 수 있는 설정을 포함하여 사용자 지정 구성 및 정책을 정의할 System.Runtime.Loader.AssemblyLoadContext 것으로 예상합니다. 후크는 추적 또는 원격 분석 주입을 설정하거나, 처리를 위한 콜백을 설정하거나, 다른 환경 종속 동작을 정의하는 데 사용할 수 있습니다. 후크는 진입점과 별개이므로 사용자 코드를 수정할 필요가 없습니다.

자세한 내용은 호스트 시작 후크를 참조하세요.

참고하십시오

- .NET Core 3.1의 새로운 기능
- ASP.NET Core 2.2의 새로운 기능
- EF Core 2.2의 새로운 기능

.NET Core 2.1의 새로운 기능

아티클 • 2023. 05. 09.

.NET Core 2.1은 다음 영역에서 향상된 기능 및 새로운 기능을 포함합니다.

- 도구
- 롤포워드
- 배포
- Windows 호환 기능 팩
- JIT 컴파일 개선
- API 변경 내용

도구

.NET Core 2.1에 포함된 도구인 .NET Core 2.1 SDK(v 2.1.300)에는 다음과 같은 변경 내용과 향상된 기능이 포함됩니다.

빌드 성능 개선

.NET Core 2.1은 특히 증분 빌드의 빌드 시간 성능을 개선하는 데 주로 초점을 맞춥니다. 이러한 성능 개선은 `dotnet build`를 사용하는 명령줄 빌드와 Visual Studio의 빌드에 모두 적용됩니다. 개선된 일부 개별 영역은 다음과 같습니다.

- 패키지 자산 해결의 경우 모든 자산이 아닌 빌드에 사용되는 자산만 해결.
- 어셈블리 참조의 캐싱.
- 여러 개별 `dotnet build` 호출에서 사용되는 프로세스인 장기 실행 SDK 빌드 서버의 사용. `dotnet build`가 실행될 때마다 코드의 큰 블록을 JIT 컴파일할 필요가 없습니다. 다음 명령을 사용하여 빌드 서버 프로세스를 자동으로 종료할 수 있습니다.

  .NET CLI

  ```
  dotnet buildserver shutdown
  ```

새 CLI 명령

`DotnetCliToolReference`를 사용하여 프로젝트별로 사용할 수 있었던 많은 도구를 이제 .NET Core SDK의 일부로 사용할 수 있습니다. 사용 가능한 도구는 다음과 같습니다.

- `dotnet watch`는 지정된 명령 집합을 실행하기 전에 파일이 변경될 때까지 대기하는 파일 시스템 감시자를 제공합니다. 예를 들어 다음 명령은 자동으로 현재 프로젝트를 다시 빌드하고 파일이 변경될 때마다 자세한 정보 출력을 생성합니다.

  .NET CLI

  ```
  dotnet watch -- --verbose build
  ```

  `--verbose` 옵션 앞에 있는 `--` 옵션에 주목하세요. 이 옵션은 `dotnet watch` 명령으로 직접 전달되는 옵션을 자식 `dotnet` 프로세스에 전달되는 인수와 구분합니다. 이 옵션을 사용하지 않으면 `--verbose` 옵션이 `dotnet build` 명령 대신 `dotnet watch` 명령에 적용됩니다.

  자세한 내용은 dotnet watch를 사용한 ASP.NET Core 앱 개발을 참조하세요.

- `dotnet dev-certs`는 ASP.NET Core 애플리케이션에서 개발하는 동안 사용되는 인증서를 생성하고 관리합니다.

- `dotnet user-secrets`는 ASP.NET Core 애플리케이션의 사용자 비밀 저장소에서 비밀을 관리합니다.

- `dotnet sql-cache`는 분산 캐싱에 사용할 Microsoft SQL Server 데이터베이스에 테이블 및 인덱스를 만듭니다.

- `dotnet ef`는 Entity Framework Core 애플리케이션에서 데이터베이스, DbContext 개체 및 마이그레이션을 관리하기 위한 도구입니다. 자세한 내용은 EF Core .NET 명령줄 도구를 참조하세요.

전역 도구

.NET Core 2.1은 명령줄에서 전역으로 사용할 수 있는 사용자 지정 도구인 '전역 도구'를 지원합니다. 이전 버전 .NET Core의 확장성 모델은 `DotnetCliToolReference`를 사용하여 프로젝트별로만 사용 가능한 사용자 지정 도구를 만들었습니다.

전역 도구를 설치하려면 dotnet tool install 명령을 사용합니다. 예를 들어:

.NET CLI

```
dotnet tool install -g dotnetsay
```

설치된 도구는 도구 이름을 지정하여 명령줄에서 실행할 수 있습니다. 자세한 내용은 .NET Core 전역 도구 개요를 참조하세요.

`dotnet tool` 명령을 사용한 도구 관리

.NET Core 2.1 SDK에서 모든 도구 작업은 `dotnet tool` 명령을 사용합니다. 다음 옵션을 사용할 수 있습니다.

- dotnet tool install - 도구를 설치합니다.
- dotnet tool update - 도구를 제거하고 다시 설치하여 효과적으로 업데이트합니다.
- dotnet tool list - 현재 설치된 도구를 나열합니다.
- dotnet tool uninstall - 현재 설치된 도구를 제거합니다.

롤포워드

.NET Core 2.0부터 모든 .NET Core 애플리케이션은 시스템에 설치된 최신 *부 버전*으로 자동으로 롤포워드됩니다.

.NET Core 2.0부터 애플리케이션 빌드에 사용된 .NET Core 버전이 런타임에 없는 경우에는 애플리케이션이 .NET Core의 설치된 최신 '부 버전'에 대해 자동으로 실행됩니다. 즉, 애플리케이션이 .NET Core 2.0을 사용하여 빌드되고 .NET Core 2.0이 호스트 시스템에 없지만 .NET Core 2.1이 있는 경우 애플리케이션은 .NET Core 2.1을 사용하여 실행됩니다.

> **ⓘ 중요**
>
> 이 롤포워드 동작은 미리 보기 릴리스에는 적용되지 않습니다. 기본적으로 주요 릴리스에도 적용되지 않지만 아래 설정으로 변경할 수 있습니다.

후보 공유 프레임워크에서 롤포워드 설정을 변경하여 이 동작을 수정할 수 있습니다. 사용 가능한 설정은 다음과 같습니다.

- `0` - 부 버전 롤포워드 동작을 비활성화합니다. 이 설정을 사용하면 .NET Core 2.0.0용으로 빌드된 애플리케이션이 .NET Core 2.0.1 롤포워드되지만 .NET Core 2.2.0 또는 .NET Core 3.0.0에서는 롤포워드되지 않습니다.
- `1` - 부 버전 롤포워드 동작을 활성화합니다. 이는 설정의 기본값입니다. 이 설정을 사용하면 .NET Core 2.0.0용으로 빌드된 애플리케이션은 설치된 .NET Core 2.0.1 또는 .NET Core 2.2.0에 따라 하나를 롤포워드하지만 .NET Core 3.0.0으로 롤포워드되지 않습니다.
- `2` - 부 버전 및 주 버전 롤포워드 동작을 활성화합니다. 설정한 경우 다른 주 버전도 고려되므로 .NET Core 2.0.0용으로 빌드된 애플리케이션은 NET Core 3.0.0으로 롤포워드됩니다.

다음 세 가지 방법 중 하나로 이 설정을 수정할 수 있습니다.

- `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` 환경 변수를 원하는 값으로 설정합니다.
- 원하는 값이 포함된 다음 줄을 *.runtimeconfig.json* 파일에 추가합니다.

  JSON

  ```
  "rollForwardOnNoCandidateFx" : 0
  ```

- .NET Core CLI를 사용하는 경우 `run`과 같은 .NET Core 명령에 원하는 값을 포함하여 다음 옵션을 추가합니다.

  .NET CLI

  ```
  dotnet run --rollForwardOnNoCandidateFx=0
  ```

패치 버전 롤포워드는 이 설정과는 무관하며 잠재적인 부 버전 롤포워드가 적용된 후에 수행됩니다.

배포

자체 포함 애플리케이션 제공

`dotnet publish`는 이제 서비스 런타임 버전이 포함된 자체 포함 애플리케이션을 게시합니다. .NET Core 2.1 SDK(v 2.1.300)를 사용하여 자체 포함 애플리케이션을 게시하면 애플리케이션에는 해당 SDK가 인식하는 최신 서비스 런타임 버전이 포함됩니다. 최신 SDK로 업그레이드하면 최신 .NET Core 런타임 버전으로 게시됩니다. 이는 .NET Core 1.0 런타임 이상에 적용됩니다.

자체 포함 게시에는 NuGet.org의 런타임 버전을 사용합니다. 컴퓨터에 서비스 런타임이 있어야 할 필요는 없습니다.

.NET Core 2.0 SDK를 사용하면 `RuntimeFrameworkVersion` 속성을 통해 다른 버전이 지정되지 않는 한 자체 포함 애플리케이션은 .NET Core 2.0.0 런타임으로 게시됩니다. 이 새 동작을 사용하면 더 이상 자체 포함 애플리케이션의 상위 런타임 버전을 선택하기 위해 이 속성을 설정할 필요가 없습니다. 앞으로 가장 쉬운 방법은 항상 .NET Core 2.1 SDK(v 2.1.300)로 게시하는 것입니다.

자세한 내용은 자체 포함 배포 런타임 롤포워드를 참조하세요.

Windows 호환 기능 팩

.NET Framework에서 .NET Core로 기존 코드를 포팅할 경우 Windows 호환 기능 팩 을 사용할 수 있습니다. .NET Core에서 사용할 수 있는 것보다 20,000개 많은 API에 대한 액세스를 제공합니다. 이러한 API에는 System.Drawing 네임스페이스, EventLog 클래스, WMI, 성능 카운터, Windows 서비스 및 Windows 레지스트리 형식/멤버의 형식이 포함됩니다.

JIT 컴파일러 개선

.NET Core는 성능을 상당히 개선할 수 있는 '계층화된 컴파일'('적응형 최적화'라고도 함)이라는 새로운 JIT 컴파일러 기술을 통합합니다. 계층화된 컴파일은 옵트인 설정입니다.

JIT 컴파일러가 수행하는 중요한 작업 중 하나는 코드 실행을 최적화하는 것입니다. 그러나 자주 사용되지 않는 코드 경로의 경우 컴파일러가 코드 최적화에 사용하는 시간이 런타임이 최적화되지 않은 코드 실행에 사용하는 시간보다 길 수 있습니다. 계층화된 컴파일은 JIT 컴파일에 다음과 같은 두 단계를 도입합니다.

- 가능한 한 빨리 코드를 생성하는 **첫 번째 계층**.
- 자주 실행되는 메서드에 대한 최적화된 코드를 생성하는 **두 번째 계층**. 컴파일의 두 번째 계층은 성능 향상을 위해 병렬로 수행됩니다.

두 가지 방법 중 하나로 계층화된 컴파일을 옵트인할 수 있습니다.

- .NET Core 2.1 SDK를 사용하는 모든 프로젝트에서 계층화된 컴파일을 사용하려면 다음 환경 변수를 설정합니다.

  콘솔

  ```
  COMPlus_TieredCompilation="1"
  ```

- 프로젝트별로 계층화된 컴파일을 사용하려면 다음 예제처럼 MSBuild 프로젝트 파일의 `<PropertyGroup>` 섹션에 `<TieredCompilation>` 속성을 추가합니다.

  XML

  ```
  <PropertyGroup>
      <!-- other property definitions -->

      <TieredCompilation>true</TieredCompilation>
  </PropertyGroup>
  ```

API 변경 내용

`Span<T>` 및 `Memory<T>`

.NET Core 2.1에는 배열 및 다른 유형의 메모리 관련 작업을 훨씬 더 효율적으로 만드는 몇 가지 새로운 형식이 포함됩니다. 새 형식은 다음과 같습니다.

- System.Span<T>와 System.ReadOnlySpan<T>을 참조하세요.
- System.Memory<T>와 System.ReadOnlyMemory<T>을 참조하세요.

이러한 형식이 없다면 배열의 일부 또는 메모리 버퍼의 섹션과 같은 항목을 전달할 경우 데이터의 일부를 복사한 다음, 메서드에 전달해야 합니다. 이러한 형식은 추가 메모리 할당 및 복사 작업이 없어도 해당 데이터의 가상 보기를 제공합니다.

다음 예제에서는 Span<T> 및 Memory<T> 인스턴스를 사용하여 배열의 10개 요소에 대한 가상 보기를 제공합니다.

C#

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[100];
        for (int i = 0; i < 100; i++)
        {
            numbers[i] = i * 2;
        }

        var part = new Span<int>(numbers, start: 10, length: 10);
        foreach (var value in part)
            Console.Write($"{value}  ");
    }
}
// The example displays the following output:
//     20  22  24  26  28  30  32  34  36  38
```

Brotli 압축

.NET Core 2.1은 Brotli 압축 및 압축 풀기에 대한 지원을 추가합니다. Brotli는 RFC 7932 에 정의되어 있고 대부분의 웹 브라우저와 주요 웹 서버에서 지원되는 범용 무손실 압축 알고리즘입니다. 스트림 기반 System.IO.Compression.BrotliStream 클래스 또는 고성능 범위 기반 System.IO.Compression.BrotliEncoder 및

System.IO.Compression.BrotliDecoder 클래스를 사용할 수 있습니다. 다음 예제는 BrotliStream 클래스를 사용한 압축을 보여 줍니다.

C#

```
public static Stream DecompressWithBrotli(Stream toDecompress)
{
    MemoryStream decompressedStream = new MemoryStream();
    using (BrotliStream decompressionStream = new BrotliStream(toDecompress,
CompressionMode.Decompress))
    {
        decompressionStream.CopyTo(decompressedStream);
    }
    decompressedStream.Position = 0;
    return decompressedStream;
}
```

BrotliStream 동작은 DeflateStream 및 GZipStream과 동일하므로, 이러한 API를 호출하는 코드를 BrotliStream으로 쉽게 변환할 수 있습니다.

새 암호화 API 및 암호화 개선

.NET Core 2.1에는 암호화 API에 대한 다양한 개선 사항이 포함되어 있습니다.

- System.Security.Cryptography.Pkcs.SignedCms는 System.Security.Cryptography.Pkcs 패키지에서 사용할 수 있습니다. 구현은 .NET Framework의 SignedCms 클래스와 동일합니다.
- X509Certificate.GetCertHash 및 X509Certificate.GetCertHashString 메서드의 새 오버로드는 해시 알고리즘 식별자를 허용하므로 호출자가 SHA-1 이외의 알고리즘을 사용하여 인증서 지문 값을 가져올 수 있습니다.
- 새 Span<T> 기반 암호화 API는 해시, HMAC, 암호화 난수 생성, 비대칭 시그니처 생성, 비대칭 시그니처 처리 및 RSA 암호화에 사용할 수 있습니다.
- System.Security.Cryptography.Rfc2898DeriveBytes 성능은 Span<T> 기반 구현을 사용함으로써 15% 정도 향상되었습니다.
- 새 System.Security.Cryptography.CryptographicOperations 클래스에는 다음 두 가지 새 메서드가 포함됩니다.
    - FixedTimeEquals는 동일한 길이의 두 입력에 대해 반환하는 데 고정된 시간을 사용하므로, 타이밍 부채널 정보에 영향을 주지 않기 위해 암호화 확인에 사용하기에 적합합니다.
    - ZeroMemory는 최적화할 수 없는 메모리 정리 루틴입니다.

- RandomNumberGenerator.Fill 정적 메서드는 Span<T>를 임의 값으로 채웁니다.
- System.Security.Cryptography.Pkcs.EnvelopedCms는 이제 Linux 및 macOS에서 지원됩니다.
- ECDH(타원 곡선 Diffie-Hellman)는 이제 System.Security.Cryptography.ECDiffieHellman 클래스 제품군에서 사용할 수 있습니다. 노출 영역은 .NET Framework의 것과 같습니다.
- RSA.Create에서 반환된 인스턴스는 SHA-2 다이제스트를 사용하여 OAEP에서 암호화 또는 암호 해독되고 RSA-PSS를 사용하여 시그니처를 생성하거나 유효성 검사할 수 있습니다.

소켓 개선

.NET Core에는 더 높은 수준의 네트워킹 API 기반을 형성하는 새 형식 System.Net.Http.SocketsHttpHandler 및 다시 작성된 System.Net.Http.HttpMessageHandler가 포함됩니다. 예를 들어 System.Net.Http.SocketsHttpHandler는 HttpClient 구현의 기반입니다. 이전 버전의 .NET Core에서 더 높은 수준의 API는 기본 네트워킹 구현을 기반으로 했습니다.

.NET Core 2.1에 도입된 소켓 구현에는 다음과 같은 여러 가지 이점이 있습니다.

- 이전 구현보다 월등히 향상된 성능.
- 배포 및 서비스를 단순화하는 플랫폼 종속성 제거.
- 모든 .NET Core 플랫폼에서 일관된 동작.

SocketsHttpHandler는 .NET Core 2.1의 기본 구현입니다. 그러나 AppContext.SetSwitch 메서드를 호출하여 이전 HttpClientHandler 클래스를 사용하도록 애플리케이션을 구성할 수 있습니다.

```
C#

AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

환경 변수를 사용하여 SocketsHttpHandler를 기반으로 한 소켓 구현 사용을 옵트아웃할 수도 있습니다. 이를 수행하려면 `DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER`를 `false` 또는 0으로 설정합니다.

Windows에서는 네이티브 구현을 사용하는 System.Net.Http.WinHttpHandler을 사용하거나 클래스의 인스턴스를 HttpClient 생성자에 전달하여 SocketsHttpHandler 클래스를 사용하도록 선택할 수도 있습니다.

Linux 및 macOS에서는 프로세스별로 HttpClient만 구성할 수 있습니다. Linux에서 이전 HttpClient 구현을 사용하려면 libcurl 을 배포해야 합니다. (.NET Core 2.0과 함께 설치됩니다.)

호환성이 손상되는 변경

호환성이 손상되는 변경에 대한 자세한 내용은 버전 2.0에서 2.1로 마이그레이션 시 호환성이 손상되는 변경을 참조하세요.

참조

- .NET Core 3.1의 새로운 기능
- EF Core 2.1의 새로운 기능
- ASP.NET Core 2.1의 새로운 기능

.NET Core 2.1의 호환성이 손상되는 변경

아티클 • 2023. 03. 18.

.NET Core 버전 2.1로 마이그레이션하는 경우 이 문서에 나열된 호환성이 손상되는 변경이 앱에 영향을 줄 수 있습니다.

핵심 .NET 라이브러리

- 경로 API가 잘못된 문자에 대해 예외를 throw하지 않음
- 기본 제공 구조체 형식에 추가된 프라이빗 필드
- macOS의 OpenSSL 버전

경로 API가 잘못된 문자에 대해 예외를 throw하지 않음

파일 경로와 관련된 API는 잘못된 문자가 발견된 경우 더 이상 경로 문자의 유효성을 검사하거나 ArgumentException을 throw하지 않습니다.

변경 내용 설명

.NET Framework 및 .NET Core 1.0~2.0에서는 경로 인수에 잘못된 경로 문자가 포함된 경우 영향을 받는 API 섹션에 나열된 메서드가 ArgumentException을 throw합니다. .NET Core 2.1부터 잘못된 문자가 발견된 경우 이 메서드는 더 이상 잘못된 경로 문자를 검사하거나 예외를 throw하지 않습니다.

변경 이유

경로 문자의 적극적 유효성 검사는 일부 플랫폼 간 시나리오를 차단합니다. .NET에서 운영 체제 API 호출 결과를 복제하거나 예측하지 않도록 이 변경 내용이 도입되었습니다. 자세한 내용은 .NET Core 2.1의 System.IO 살펴보기 블로그 게시물을 참조하세요.

도입된 버전

.NET Core 2.1

권장 조치

코드가 이 API를 사용하여 잘못된 문자를 검사한 경우 Path.GetInvalidPathChars 호출을 추가할 수 있습니다.

영향을 받는 API

- System.IO.Directory.CreateDirectory
- System.IO.Directory.Delete
- System.IO.Directory.EnumerateDirectories
- System.IO.Directory.EnumerateFiles
- System.IO.Directory.EnumerateFileSystemEntries
- System.IO.Directory.GetCreationTime(String)
- System.IO.Directory.GetCreationTimeUtc(String)
- System.IO.Directory.GetDirectories
- System.IO.Directory.GetDirectoryRoot(String)
- System.IO.Directory.GetFiles
- System.IO.Directory.GetFileSystemEntries
- System.IO.Directory.GetLastAccessTime(String)
- System.IO.Directory.GetLastAccessTimeUtc(String)
- System.IO.Directory.GetLastWriteTime(String)
- System.IO.Directory.GetLastWriteTimeUtc(String)
- System.IO.Directory.GetParent(String)
- System.IO.Directory.Move(String, String)
- System.IO.Directory.SetCreationTime(String, DateTime)
- System.IO.Directory.SetCreationTimeUtc(String, DateTime)
- System.IO.Directory.SetCurrentDirectory(String)
- System.IO.Directory.SetLastAccessTime(String, DateTime)
- System.IO.Directory.SetLastAccessTimeUtc(String, DateTime)
- System.IO.Directory.SetLastWriteTime(String, DateTime)
- System.IO.Directory.SetLastWriteTimeUtc(String, DateTime)
- System.IO.DirectoryInfo ctor
- System.IO.Directory.GetDirectories
- System.IO.Directory.GetFiles
- System.IO.DirectoryInfo.GetFileSystemInfos
- System.IO.File.AppendAllText
- System.IO.File.AppendAllTextAsync
- System.IO.File.Copy
- System.IO.File.Create
- System.IO.File.CreateText
- System.IO.File.Decrypt
- System.IO.File.Delete
- System.IO.File.Encrypt
- System.IO.File.GetAttributes(String)
- System.IO.File.GetCreationTime(String)
- System.IO.File.GetCreationTimeUtc(String)

- System.IO.File.GetLastAccessTime(String)
- System.IO.File.GetLastAccessTimeUtc(String)
- System.IO.File.GetLastWriteTime(String)
- System.IO.File.GetLastWriteTimeUtc(String)
- System.IO.File.Move
- System.IO.File.Open
- System.IO.File.OpenRead(String)
- System.IO.File.OpenText(String)
- System.IO.File.OpenWrite(String)
- System.IO.File.ReadAllBytes(String)
- System.IO.File.ReadAllBytesAsync(String, CancellationToken)
- System.IO.File.ReadAllLines(String)
- System.IO.File.ReadAllLinesAsync(String, CancellationToken)
- System.IO.File.ReadAllText(String)
- System.IO.File.ReadAllTextAsync(String, CancellationToken)
- System.IO.File.SetAttributes(String, FileAttributes)
- System.IO.File.SetCreationTime(String, DateTime)
- System.IO.File.SetCreationTimeUtc(String, DateTime)
- System.IO.File.SetLastAccessTime(String, DateTime)
- System.IO.File.SetLastAccessTimeUtc(String, DateTime)
- System.IO.File.SetLastWriteTime(String, DateTime)
- System.IO.File.SetLastWriteTimeUtc(String, DateTime)
- System.IO.File.WriteAllBytes(String, Byte[])
- System.IO.File.WriteAllBytesAsync(String, Byte[], CancellationToken)
- System.IO.File.WriteAllLines
- System.IO.File.WriteAllLinesAsync
- System.IO.File.WriteAllText
- System.IO.FileInfo ctor
- System.IO.FileInfo.CopyTo
- System.IO.FileInfo.MoveTo
- System.IO.FileStream ctor
- System.IO.Path.GetFullPath(String)
- System.IO.Path.IsPathRooted(String)
- System.IO.Path.GetPathRoot(String)
- System.IO.Path.ChangeExtension(String, String)
- System.IO.Path.GetDirectoryName(String)
- System.IO.Path.GetExtension(String)
- System.IO.Path.HasExtension(String)
- System.IO.Path.Combine

추가 정보

- .NET Core 2.1의 System.IO 살펴보기

기본 제공 구조체 형식에 추가된 프라이빗 필드

프라이빗 필드가 참조 어셈블리의 특정 구조체 형식에 추가되었습니다. 그 결과 C#에서 이러한 구조체 형식은 항상 new 연산자 또는 기본 리터럴을 사용하여 인스턴스화해야 합니다.

변경 내용 설명

.NET Core 2.0 및 이전 버전에서 제공되는 일부 구조체 형식(예: ConsoleKeyInfo)은 C#에서 `new` 연산자 또는 기본 리터럴을 사용하지 않고도 인스턴스화할 수 있습니다. 이는 C# 컴파일러에서 사용하는 참조 어셈블리에 구조체에 대한 프라이빗 필드가 포함되지 않았기 때문입니다. .NET 구조체 형식에 대한 모든 프라이빗 필드는 .NET Core 2.1에서 시작하는 참조 어셈블리에 추가됩니다.

예를 들어 다음 C# 코드는 .NET Core 2.0에서 컴파일되지만 .NET Core 2.1에서는 컴파일되지 않습니다.

```
C#

ConsoleKeyInfo key;    // Struct type

if (key.ToString() == "y")
{
    Console.WriteLine("Yes!");
}
```

.NET Core 2.1의 이전 코드에서는 다음과 같은 컴파일러 오류가 발생합니다. **CS0165 - 할당되지 않은 지역 변수 'key'를 사용했습니다.**

도입된 버전

2.1

권장 조치

`new` 연산자 또는 기본 리터럴을 사용하여 구조체 형식을 인스턴스화합니다.

예를 들어:

C#

```
ConsoleKeyInfo key = new ConsoleKeyInfo();    // Struct type.

if (key.ToString() == "y")
    Console.WriteLine("Yes!");
```

C#

```
ConsoleKeyInfo key = default;    // Struct type.

if (key.ToString() == "y")
    Console.WriteLine("Yes!");
```

범주

핵심 .NET 라이브러리

영향을 받는 API

- System.ArraySegment<T>.Enumerator
- System.ArraySegment<T>
- System.Boolean
- System.Buffers.MemoryHandle
- System.Buffers.StandardFormat
- System.Byte
- System.Char
- System.Collections.DictionaryEntry
- System.Collections.Generic.Dictionary<TKey,TValue>.Enumerator
- System.Collections.Generic.Dictionary<TKey,TValue>.KeyCollection.Enumerator
- System.Collections.Generic.Dictionary<TKey,TValue>.ValueCollection.Enumerator
- System.Collections.Generic.HashSet<T>.Enumerator
- System.Collections.Generic.KeyValuePair<TKey,TValue>
- System.Collections.Generic.LinkedList<T>.Enumerator
- System.Collections.Generic.List<T>.Enumerator
- System.Collections.Generic.Queue<T>.Enumerator
- System.Collections.Generic.SortedDictionary<TKey,TValue>.Enumerator
- System.Collections.Generic.SortedDictionary<TKey,TValue>.KeyCollection.Enumerator
- System.Collections.Generic.SortedDictionary<TKey,TValue>.ValueCollection.Enumerator
- System.Collections.Generic.SortedSet<T>.Enumerator

- System.Collections.Generic.Stack<T>.Enumerator
- System.Collections.Immutable.ImmutableArray<T>.Enumerator
- System.Collections.Immutable.ImmutableArray<T>
- System.Collections.Immutable.ImmutableDictionary<TKey,TValue>.Enumerator
- System.Collections.Immutable.ImmutableHashSet<T>.Enumerator
- System.Collections.Immutable.ImmutableList<T>.Enumerator
- System.Collections.Immutable.ImmutableQueue<T>.Enumerator
- System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>.Enumerator
- System.Collections.Immutable.ImmutableSortedSet<T>.Enumerator
- System.Collections.Immutable.ImmutableStack<T>.Enumerator
- System.Collections.Specialized.BitVector32.Section
- System.Collections.Specialized.BitVector32
- LazyMemberInfo
- System.ComponentModel.Design.Serialization.MemberRelationship
- System.ConsoleKeyInfo
- System.Data.SqlTypes.SqlBinary
- System.Data.SqlTypes.SqlBoolean
- System.Data.SqlTypes.SqlByte
- System.Data.SqlTypes.SqlDateTime
- System.Data.SqlTypes.SqlDecimal
- System.Data.SqlTypes.SqlDouble
- System.Data.SqlTypes.SqlGuid
- System.Data.SqlTypes.SqlInt16
- System.Data.SqlTypes.SqlInt32
- System.Data.SqlTypes.SqlInt64
- System.Data.SqlTypes.SqlMoney
- System.Data.SqlTypes.SqlSingle
- System.Data.SqlTypes.SqlString
- System.DateTime
- System.DateTimeOffset
- System.Decimal
- System.Diagnostics.CounterSample
- System.Diagnostics.SymbolStore.SymbolToken
- System.Diagnostics.Tracing.EventSource.EventData
- System.Diagnostics.Tracing.EventSourceOptions
- System.Double
- System.Drawing.CharacterRange
- System.Drawing.Point
- System.Drawing.PointF

- System.Drawing.Rectangle
- System.Drawing.RectangleF
- System.Drawing.Size
- System.Drawing.SizeF
- System.Guid
- System.HashCode
- System.Int16
- System.Int32
- System.Int64
- System.IntPtr
- System.IO.Pipelines.FlushResult
- System.IO.Pipelines.ReadResult
- System.IO.WaitForChangedResult
- System.Memory<T>
- System.ModuleHandle
- System.Net.Security.SslApplicationProtocol
- System.Net.Sockets.IPPacketInformation
- System.Net.Sockets.SocketInformation
- System.Net.Sockets.UdpReceiveResult
- System.Net.WebSockets.ValueWebSocketReceiveResult
- System.Nullable<T>
- System.Numerics.BigInteger
- System.Numerics.Complex
- System.Numerics.Vector<T>
- System.ReadOnlyMemory<T>
- System.ReadOnlySpan<T>.Enumerator
- System.ReadOnlySpan<T>
- System.Reflection.CustomAttributeNamedArgument
- System.Reflection.CustomAttributeTypedArgument
- System.Reflection.Emit.Label
- System.Reflection.Emit.OpCode
- System.Reflection.Metadata.ArrayShape
- System.Reflection.Metadata.AssemblyDefinition
- System.Reflection.Metadata.AssemblyDefinitionHandle
- System.Reflection.Metadata.AssemblyFile
- System.Reflection.Metadata.AssemblyFileHandle
- System.Reflection.Metadata.AssemblyFileHandleCollection.Enumerator
- System.Reflection.Metadata.AssemblyFileHandleCollection
- System.Reflection.Metadata.AssemblyReference
- System.Reflection.Metadata.AssemblyReferenceHandle

- System.Reflection.Metadata.AssemblyReferenceHandleCollection.Enumerator
- System.Reflection.Metadata.AssemblyReferenceHandleCollection
- System.Reflection.Metadata.Blob
- System.Reflection.Metadata.BlobBuilder.Blobs
- System.Reflection.Metadata.BlobContentId
- System.Reflection.Metadata.BlobHandle
- System.Reflection.Metadata.BlobReader
- System.Reflection.Metadata.BlobWriter
- System.Reflection.Metadata.Constant
- System.Reflection.Metadata.ConstantHandle
- System.Reflection.Metadata.CustomAttribute
- System.Reflection.Metadata.CustomAttributeHandle
- System.Reflection.Metadata.CustomAttributeHandleCollection.Enumerator
- System.Reflection.Metadata.CustomAttributeHandleCollection
- System.Reflection.Metadata.CustomAttributeNamedArgument<TType>
- System.Reflection.Metadata.CustomAttributeTypedArgument<TType>
- System.Reflection.Metadata.CustomAttributeValue<TType>
- System.Reflection.Metadata.CustomDebugInformation
- System.Reflection.Metadata.CustomDebugInformationHandle
- System.Reflection.Metadata.CustomDebugInformationHandleCollection.Enumerator
- System.Reflection.Metadata.CustomDebugInformationHandleCollection
- System.Reflection.Metadata.DeclarativeSecurityAttribute
- System.Reflection.Metadata.DeclarativeSecurityAttributeHandle
- System.Reflection.Metadata.DeclarativeSecurityAttributeHandleCollection.Enumerator
- System.Reflection.Metadata.DeclarativeSecurityAttributeHandleCollection
- System.Reflection.Metadata.Document
- System.Reflection.Metadata.DocumentHandle
- System.Reflection.Metadata.DocumentHandleCollection.Enumerator
- System.Reflection.Metadata.DocumentHandleCollection
- System.Reflection.Metadata.DocumentNameBlobHandle
- System.Reflection.Metadata.Ecma335.ArrayShapeEncoder
- System.Reflection.Metadata.Ecma335.BlobEncoder
- System.Reflection.Metadata.Ecma335.CustomAttributeArrayTypeEncoder
- System.Reflection.Metadata.Ecma335.CustomAttributeElementTypeEncoder
- System.Reflection.Metadata.Ecma335.CustomAttributeNamedArgumentsEncoder
- System.Reflection.Metadata.Ecma335.CustomModifiersEncoder
- System.Reflection.Metadata.Ecma335.EditAndContinueLogEntry
- System.Reflection.Metadata.Ecma335.ExceptionRegionEncoder

- System.Reflection.Metadata.Ecma335.FixedArgumentsEncoder
- System.Reflection.Metadata.Ecma335.GenericTypeArgumentsEncoder
- System.Reflection.Metadata.Ecma335.InstructionEncoder
- System.Reflection.Metadata.Ecma335.LabelHandle
- System.Reflection.Metadata.Ecma335.LiteralEncoder
- System.Reflection.Metadata.Ecma335.LiteralsEncoder
- System.Reflection.Metadata.Ecma335.LocalVariablesEncoder
- System.Reflection.Metadata.Ecma335.LocalVariableTypeEncoder
- System.Reflection.Metadata.Ecma335.MethodBodyStreamEncoder.MethodBody
- System.Reflection.Metadata.Ecma335.MethodBodyStreamEncoder
- System.Reflection.Metadata.Ecma335.MethodSignatureEncoder
- System.Reflection.Metadata.Ecma335.NamedArgumentsEncoder
- System.Reflection.Metadata.Ecma335.NamedArgumentTypeEncoder
- System.Reflection.Metadata.Ecma335.NameEncoder
- System.Reflection.Metadata.Ecma335.ParametersEncoder
- System.Reflection.Metadata.Ecma335.ParameterTypeEncoder
- System.Reflection.Metadata.Ecma335.PermissionSetEncoder
- System.Reflection.Metadata.Ecma335.ReturnTypeEncoder
- System.Reflection.Metadata.Ecma335.ScalarEncoder
- System.Reflection.Metadata.Ecma335.SignatureDecoder<TType,TGenericContext>
- System.Reflection.Metadata.Ecma335.SignatureTypeEncoder
- System.Reflection.Metadata.Ecma335.VectorEncoder
- System.Reflection.Metadata.EntityHandle
- System.Reflection.Metadata.EventAccessors
- System.Reflection.Metadata.EventDefinition
- System.Reflection.Metadata.EventDefinitionHandle
- System.Reflection.Metadata.EventDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.EventDefinitionHandleCollection
- System.Reflection.Metadata.ExceptionRegion
- System.Reflection.Metadata.ExportedType
- System.Reflection.Metadata.ExportedTypeHandle
- System.Reflection.Metadata.ExportedTypeHandleCollection.Enumerator
- System.Reflection.Metadata.ExportedTypeHandleCollection
- System.Reflection.Metadata.FieldDefinition
- System.Reflection.Metadata.FieldDefinitionHandle
- System.Reflection.Metadata.FieldDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.FieldDefinitionHandleCollection
- System.Reflection.Metadata.GenericParameter
- System.Reflection.Metadata.GenericParameterConstraint
- System.Reflection.Metadata.GenericParameterConstraintHandle

- System.Reflection.Metadata.GenericParameterConstraintHandleCollection.Enumerator
- System.Reflection.Metadata.GenericParameterConstraintHandleCollection
- System.Reflection.Metadata.GenericParameterHandle
- System.Reflection.Metadata.GenericParameterHandleCollection.Enumerator
- System.Reflection.Metadata.GenericParameterHandleCollection
- System.Reflection.Metadata.GuidHandle
- System.Reflection.Metadata.Handle
- System.Reflection.Metadata.ImportDefinition
- System.Reflection.Metadata.ImportDefinitionCollection.Enumerator
- System.Reflection.Metadata.ImportDefinitionCollection
- System.Reflection.Metadata.ImportScope
- System.Reflection.Metadata.ImportScopeCollection.Enumerator
- System.Reflection.Metadata.ImportScopeCollection
- System.Reflection.Metadata.ImportScopeHandle
- System.Reflection.Metadata.InterfaceImplementation
- System.Reflection.Metadata.InterfaceImplementationHandle
- System.Reflection.Metadata.InterfaceImplementationHandleCollection.Enumerator
- System.Reflection.Metadata.InterfaceImplementationHandleCollection
- System.Reflection.Metadata.LocalConstant
- System.Reflection.Metadata.LocalConstantHandle
- System.Reflection.Metadata.LocalConstantHandleCollection.Enumerator
- System.Reflection.Metadata.LocalConstantHandleCollection
- System.Reflection.Metadata.LocalScope
- System.Reflection.Metadata.LocalScopeHandle
- System.Reflection.Metadata.LocalScopeHandleCollection.ChildrenEnumerator
- System.Reflection.Metadata.LocalScopeHandleCollection.Enumerator
- System.Reflection.Metadata.LocalScopeHandleCollection
- System.Reflection.Metadata.LocalVariable
- System.Reflection.Metadata.LocalVariableHandle
- System.Reflection.Metadata.LocalVariableHandleCollection.Enumerator
- System.Reflection.Metadata.LocalVariableHandleCollection
- System.Reflection.Metadata.ManifestResource
- System.Reflection.Metadata.ManifestResourceHandle
- System.Reflection.Metadata.ManifestResourceHandleCollection.Enumerator
- System.Reflection.Metadata.ManifestResourceHandleCollection
- System.Reflection.Metadata.MemberReference
- System.Reflection.Metadata.MemberReferenceHandle
- System.Reflection.Metadata.MemberReferenceHandleCollection.Enumerator
- System.Reflection.Metadata.MemberReferenceHandleCollection

- System.Reflection.Metadata.MetadataStringComparer
- System.Reflection.Metadata.MethodDebugInformation
- System.Reflection.Metadata.MethodDebugInformationHandle
- System.Reflection.Metadata.MethodDebugInformationHandleCollection.Enumerator
- System.Reflection.Metadata.MethodDebugInformationHandleCollection
- System.Reflection.Metadata.MethodDefinition
- System.Reflection.Metadata.MethodDefinitionHandle
- System.Reflection.Metadata.MethodDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.MethodDefinitionHandleCollection
- System.Reflection.Metadata.MethodImplementation
- System.Reflection.Metadata.MethodImplementationHandle
- System.Reflection.Metadata.MethodImplementationHandleCollection.Enumerator
- System.Reflection.Metadata.MethodImplementationHandleCollection
- System.Reflection.Metadata.MethodImport
- System.Reflection.Metadata.MethodSignature<TType>
- System.Reflection.Metadata.MethodSpecification
- System.Reflection.Metadata.MethodSpecificationHandle
- System.Reflection.Metadata.ModuleDefinition
- System.Reflection.Metadata.ModuleDefinitionHandle
- System.Reflection.Metadata.ModuleReference
- System.Reflection.Metadata.ModuleReferenceHandle
- System.Reflection.Metadata.NamespaceDefinition
- System.Reflection.Metadata.NamespaceDefinitionHandle
- System.Reflection.Metadata.Parameter
- System.Reflection.Metadata.ParameterHandle
- System.Reflection.Metadata.ParameterHandleCollection.Enumerator
- System.Reflection.Metadata.ParameterHandleCollection
- System.Reflection.Metadata.PropertyAccessors
- System.Reflection.Metadata.PropertyDefinition
- System.Reflection.Metadata.PropertyDefinitionHandle
- System.Reflection.Metadata.PropertyDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.PropertyDefinitionHandleCollection
- System.Reflection.Metadata.ReservedBlob<THandle>
- System.Reflection.Metadata.SequencePoint
- System.Reflection.Metadata.SequencePointCollection.Enumerator
- System.Reflection.Metadata.SequencePointCollection
- System.Reflection.Metadata.SignatureHeader
- System.Reflection.Metadata.StandaloneSignature
- System.Reflection.Metadata.StandaloneSignatureHandle

- System.Reflection.Metadata.StringHandle
- System.Reflection.Metadata.TypeDefinition
- System.Reflection.Metadata.TypeDefinitionHandle
- System.Reflection.Metadata.TypeDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.TypeDefinitionHandleCollection
- System.Reflection.Metadata.TypeLayout
- System.Reflection.Metadata.TypeReference
- System.Reflection.Metadata.TypeReferenceHandle
- System.Reflection.Metadata.TypeReferenceHandleCollection.Enumerator
- System.Reflection.Metadata.TypeReferenceHandleCollection
- System.Reflection.Metadata.TypeSpecification
- System.Reflection.Metadata.TypeSpecificationHandle
- System.Reflection.Metadata.UserStringHandle
- System.Reflection.ParameterModifier
- System.Reflection.PortableExecutable.CodeViewDebugDirectoryData
- System.Reflection.PortableExecutable.DebugDirectoryEntry
- System.Reflection.PortableExecutable.PEMemoryBlock
- System.Reflection.PortableExecutable.SectionHeader
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncVoidMethodBuilder
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable<TResult>.ConfiguredTaskAwaiter
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable<TResult>
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable.ConfiguredTaskAwaiter
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable
- System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>
- System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>.ConfiguredValueTaskAwaiter
- System.Runtime.CompilerServices.TaskAwaiter<TResult>
- System.Runtime.CompilerServices.TaskAwaiter
- System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>
- System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>
- System.Runtime.InteropServices.ArrayWithOffset
- System.Runtime.InteropServices.GCHandle
- System.Runtime.InteropServices.HandleRef
- System.Runtime.InteropServices.OSPlatform
- System.Runtime.InteropServices.WindowsRuntime.EventRegistrationToken

- System.Runtime.Serialization.SerializationEntry
- System.Runtime.Serialization.StreamingContext
- System.RuntimeArgumentHandle
- System.RuntimeFieldHandle
- System.RuntimeMethodHandle
- System.RuntimeTypeHandle
- System.SByte
- System.Security.Cryptography.CngProperty
- System.Security.Cryptography.ECCurve
- System.Security.Cryptography.HashAlgorithmName
- System.Security.Cryptography.X509Certificates.X509ChainStatus
- System.Security.Cryptography.Xml.X509IssuerSerial
- System.ServiceProcess.SessionChangeDescription
- System.Single
- System.Span<T>.Enumerator
- System.Span<T>
- System.Threading.AsyncFlowControl
- System.Threading.AsyncLocalValueChangedArgs<T>
- System.Threading.CancellationToken
- System.Threading.CancellationTokenRegistration
- System.Threading.LockCookie
- System.Threading.SpinLock
- System.Threading.SpinWait
- System.Threading.Tasks.Dataflow.DataflowMessageHeader
- System.Threading.Tasks.ParallelLoopResult
- System.Threading.Tasks.ValueTask<TResult>
- System.TimeSpan
- System.TimeZoneInfo.TransitionTime
- System.Transactions.TransactionOptions
- System.TypedReference
- System.TypedReference
- System.UInt16
- System.UInt32
- System.UInt64
- System.UIntPtr
- System.Windows.Forms.ColorDialog.Color
- System.Windows.Media.Animation.KeyTime
- System.Windows.Media.Animation.RepeatBehavior
- System.Xml.Serialization.XmlDeserializationEvents
- Windows.Foundation.Point

- Windows.Foundation.Rect
- Windows.Foundation.Size
- Windows.UI.Color
- Windows.UI.Xaml.Controls.Primitives.GeneratorPosition
- Windows.UI.Xaml.CornerRadius
- Windows.UI.Xaml.Duration
- Windows.UI.Xaml.GridLength
- Windows.UI.Xaml.Media.Matrix
- Windows.UI.Xaml.Media.Media3D.Matrix3D
- Windows.UI.Xaml.Thickness

---

macOS의 OpenSSL 버전

macOS에서 .NET Core 3.0 이상 런타임은 이제 AesCcm, AesGcm, DSAOpenSsl, ECDiffieHellmanOpenSsl, ECDsaOpenSsl, RSAOpenSsl 및 SafeEvpPKeyHandle 형식에 대해 OpenSSL 1.0.x 버전보다 OpenSSL 1.1.x 버전을 선호합니다.

.NET Core 2.1 런타임은 이제 OpenSSL 1.1.x 버전을 지원하지만 여전히 OpenSSL 1.0.x 버전을 선호합니다.

변경 내용 설명

이전에는 .NET Core 런타임이 macOS에서 OpenSSL과 상호 작용하는 형식에 대해 OpenSSL 1.0.x 버전을 사용했습니다. 최신 OpenSSL 1.0.x 버전인 OpenSSL 1.0.2는 이제 지원되지 않습니다. 지원되는 버전의 OpenSSL에 대해 OpenSSL을 사용하는 형식을 유지하기 위해 .NET Core 3.0 이상 런타임이 이제 macOS에서 최신 버전의 OpenSSL을 사용합니다.

이러한 변경을 통해 macOS에서 .NET Core 런타임에 대한 동작은 다음과 같습니다.

- .NET Core 3.0 이상 버전 런타임은 OpenSSL 1.1.x 버전(사용 가능한 경우)을 사용하고 사용 가능한 1.1.x 버전이 없는 경우에만 OpenSSL 1.0.x 버전으로 대체됩니다.

  사용자 지정 P/Invokes에 OpenSSL interop 형식을 사용하는 호출자의 경우 SafeEvpPKeyHandle.OpenSslVersion 설명의 지침을 따릅니다. OpenSslVersion 값을 확인하지 않으면 앱의 작동이 중단될 수도 있습니다.

- .NET Core 2.1 런타임은 OpenSSL 1.0.x 버전(사용 가능한 경우)을 사용하고 사용 가능한 1.0.x 버전이 없는 경우 OpenSSL 1.1.x 버전으로 대체됩니다.

  .NET Core 2.1에 SafeEvpPKeyHandle.OpenSslVersion 속성이 없어 런타임에 신뢰성 있게 OpenSSL 버전을 확인할 수 없기 때문에 2.1 런타임은 이전 버전의 OpenSSL을

선호합니다.

도입된 버전

- .NET Core 2.1.16
- .NET Core 3.0.3
- .NET Core 3.1.2

권장 조치

- 더 이상 필요하지 않으면 OpenSSL 1.0.2 버전을 제거합니다.
- AesCcm, AesGcm, DSAOpenSsl, ECDiffieHellmanOpenSsl, ECDsaOpenSsl, RSAOpenSsl 또는 SafeEvpPKeyHandle 형식을 사용하는 경우 OpenSSL 1.1.x 버전을 설치합니다.
- 사용자 지정 P/Invokes에 OpenSSL interop 형식을 사용하는 경우 SafeEvpPKeyHandle.OpenSslVersion 설명의 지침을 따릅니다.

범주

핵심 .NET 라이브러리

영향을 받는 API

- System.Security.Cryptography.AesCcm
- System.Security.Cryptography.AesGcm
- System.Security.Cryptography.DSAOpenSsl
- System.Security.Cryptography.ECDiffieHellmanOpenSsl
- System.Security.Cryptography.ECDsaOpenSsl
- System.Security.Cryptography.RSAOpenSsl
- System.Security.Cryptography.SafeEvpPKeyHandle

MSBuild

- 이제 프로젝트 도구가 SDK에 포함됨

이제 프로젝트 도구가 SDK에 포함됨

이제 .NET Core 2.1 SDK에 일반적인 CLI 도구가 포함되며 더 이상 프로젝트에서 해당 도구를 참조할 필요가 없습니다.

변경 내용 설명

.NET Core 2.0에서 프로젝트는 `<DotNetCliToolReference>` 프로젝트 설정을 사용하여 외부 .NET 도구를 참조합니다. .NET Core 2.1에서 해당 도구 중 일부는 .NET Core SDK에 포함되며 더 이상 설정이 필요하지 않습니다. 프로젝트에 해당 도구에 대한 참조를 포함하는 경우 다음과 같은 오류가 표시됩니다. **'Microsoft.EntityFrameworkCore.Tools.DotNet' 도구는 이제 .NET Core SDK에 포함됩니다.**

이제 도구가 .NET Core 2.1 SDK에 포함됨:

<DotNetCliToolReference> 값	도구
`Microsoft.DotNet.Watcher.Tools`	`dotnet-watch`
`Microsoft.Extensions.SecretManager.Tools`	dotnet-user-secrets
`Microsoft.Extensions.Caching.SqlConfig.Tools`	dotnet-sql-cache
`Microsoft.EntityFrameworkCore.Tools.DotNet`	dotnet-ef

도입된 버전

.NET Core SDK 2.1.300

권장 조치

프로젝트에서 `<DotNetCliToolReference>` 설정을 제거합니다.

범주

MSBuild

영향을 받는 API

N/A

참고 항목

- .NET Core 2.1의 새로운 기능

.NET Core 2.0의 새로운 기능

아티클 • 2025. 01. 17.

.NET Core 2.0은 다음 영역에서 향상된 기능 및 새로운 기능을 포함합니다.

- 도구
- 언어 지원
- 플랫폼 개선
- API 변경 내용
- Visual Studio 통합
- 설명서 개선

도구

dotnet restore runs implicitly

이전 버전의 .NET Core에서 dotnet new 명령을 사용하여 새 프로젝트를 만든 이후뿐만 아니라 프로젝트에 새 종속성을 추가할 때마다 dotnet restore 명령을 실행하여 즉시 종속성을 다운로드합니다.

dotnet restore, `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test` 및 `dotnet publish` 등 복원이 필요한 모든 명령에 의해 암시적으로 실행되므로 `dotnet pack`를 실행할 필요가 없습니다. 암시적 복원을 사용하지 않으려면 `--no-restore` 옵션을 사용합니다.

`dotnet restore` 명령은 Azure DevOps Services의 연속 통합 빌드 또는 복원 발생 시점을 명시적으로 제어해야 하는 빌드 시스템과 같이 명시적으로 복원이 가능한 특정 시나리오에서 여전히 유용합니다.

NuGet 피드를 관리하는 방법에 대한 자세한 내용은 dotnet restore 설명서를 참조하세요.

`dotnet restore` 스위치를 `--no-restore`, `new`, `run`, `build`, `publish` 및 `pack` 명령으로 전달하여 `test` 자동 호출을 사용하지 않도록 설정할 수 있습니다.

.NET Core 2.0로 대상 다시 지정

.NET Core 2.0 SDK가 설치되면 .NET Core 1.x을 대상으로 지정하는 프로젝트는 .NET Core 2.0의 대상으로 다시 지정될 수 있습니다.

.NET Core 2.0을 대상으로 지정하려면 1.x에서 2.0까지 `<TargetFramework>` 요소(또는 프로젝트 파일에서 대상이 둘 이상 있는 경우 `<TargetFrameworks>` 요소)의 값을 변경하여 프

로젝트 파일을 편집합니다.

```XML
<PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
 </PropertyGroup>
```

또한 같은 방식으로 .NET 표준 2.0으로 .NET 표준 라이브러리의 대상을 다시 지정할 수 있습니다.

```XML
<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
 </PropertyGroup>
```

프로젝트를 .NET Core 2.0으로 마이그레이션하는 방법에 대한 자세한 정보는 ASP.NET Core 1.x에서 ASP.NET Core 2.0으로 마이그레이션을 참조하세요.

언어 지원

C# 및 F#를 지원할 뿐만 아니라 .NET Core 2.0도 Visual Basic을 지원합니다.

Visual Basic

이제 버전 2.0,.NET Core는 Visual Basic 2017을 지원합니다. Visual Basic을 사용하여 다음과 같은 프로젝트 형식을 만들 수 있습니다.

- .NET Core 콘솔 앱
- .NET Core 클래스 라이브러리
- .NET 표준 클래스 라이브러리
- .NET Core 단위 테스트 프로젝트
- .NET Core xUnit 테스트 프로젝트

예를 들어 Visual Basic "Hello World" 애플리케이션을 만들려면 명령줄에서 다음 단계를 수행합니다.

1. 콘솔 창을 열고 프로젝트에 대한 디렉터리를 만들고 현재 디렉터리로 설정합니다.
2. `dotnet new console -lang vb` 명령을 입력합니다.

   이 명령은 `.vbproj`라는 Visual Basic 소스 코드와 함께 파일 확장인 프로젝트 파일을 만듭니다. 이 파일은 콘솔 창에 문자열 "Hello World!"를 작성하는 소스 코드를 포함

합니다.

3. `dotnet run` 명령을 입력합니다. .NET CLI는 콘솔 창에 "Hello World!" 메시지를 표시하는 애플리케이션을 자동으로 컴파일하고 실행합니다.

C# 7.1에 대한 지원

.NET Core 2.0은 다음을 비롯하여 다양하고 새로운 기능을 추가하는 C# 7.1을 지원합니다.

- 애플리케이션 진입점인 `Main` 메서드는 비동기 키워드로 표시될 수 있습니다.
- 유추된 튜플 이름입니다.
- 기본 식입니다.

플랫폼 개선

.NET Core 2.0은 .NET Core를 쉽게 설치하고 지원되는 운영 체제에서 사용할 수 있는 다양한 기능을 포함합니다.

Linux용 .NET core는 단일 구현입니다.

.NET Core 2.0은 여러 Linux 배포에서 작동하는 단일 Linux 구현을 제공합니다. .NET Core 1.x은 배포 관련 Linux 구현을 다운로드하는 데 필요합니다.

또한 단일 운영 체제로 Linux를 대상으로 하는 앱을 개발할 수 있습니다. .NET Core 1.x는 별도로 각 Linux 배포를 대상으로 하는 데 필요합니다.

Apple 암호화 라이브러리에 대한 지원

.NET Core 1.x는 OpenSSL 도구 키트의 암호화 라이브러리에 필요합니다. .NET Core 2.0은 Apple 암호화 라이브러리를 사용하고 OpenSSL을 필요로 하지 않으므로 더 이상 설치할 필요가 없습니다.

API 변경 내용 및 라이브러리 지원

.NET Standard 2.0에 대한 지원

.NET Standard는 해당 버전의 표준에 부합하는 .NET 구현에서 사용할 수 있어야 하는 버전이 지정된 API 집합을 정의합니다. .NET Standard는 라이브러리 개발자에서 대상으로 지정됩니다. 각 .NET 구현에 .NET Standard의 버전을 대상으로 지정하는 라이브러리에 사

용할 수 있는 기능을 보장하려고 합니다. .NET Core 1.x는 .NET Standard 버전 1.6을 지원하고 .NET Core 2.0은 최신 버전인 .NET Standard 2.0을 지원합니다. 자세한 내용은 .NET 표준을 참조하세요.

.NET Standard 2.0에는 .NET Standard 1.6에서 사용할 수 있었던 20,000개가 넘는 API가 포함됩니다. 이 확장된 노출 영역 중 상당 부분은 .NET Framework 및 Xamarin에 공통되는 API를 .NET 표준에 통합한 결과입니다.

.NET Standard 2.0 클래스 라이브러리는 .NET Framework 클래스 라이브러리를 참조할 수도 있고 .NET Standard 2.0에 표시되는 API를 호출하도록 제공됩니다. .NET Framework 라이브러리를 다시 컴파일하지 않아도 됩니다.

마지막 버전인 .NET Standard 1.6 이후 .NET Standard에 추가된 API 목록은 .NET Standard 2.0 및 1.6 을 참조하세요.

확장된 노출 영역

.NET Core 2.0에서는 사용할 수 있는 API의 총수는 .NET Core 1.1에 비교하여 두 배 이상 증가했습니다.

그리고 Windows 호환 기능 팩 덕분에 .NET Framework에서 이식하는 작업이 훨씬 더 간편해졌습니다.

.NET Framework 라이브러리에 대한 지원

.NET Core 코드는 기존 NuGet 패키지를 비롯한 기존 .NET Framework 라이브러리를 참조할 수 있습니다. 라이브러리는 표준.NET에서 발견되는 API를 사용해야 합니다.

Visual Studio 통합

Visual Studio 2017 버전 15.3은 .NET Core 개발자를 위한 다양한 향상된 기능을 제공합니다.

.NET Core 앱 및 .NET 표준 라이브러리를 대상으로 다시 지정

.NET Core 2.0 SDK가 설치되는 경우 .NET Core 1.x 프로젝트를 .NET Core 2.0으로, .NET 표준 1.x 라이브러리를 .NET 표준 2.0으로 대상을 변경할 수 있습니다.

Visual Studio에서 프로젝트의 대상을 변경하려면 프로젝트 속성 대화 상자의 **애플리케이션** 탭을 열고 **대상 프레임 워크** 값을 **.NET Core 2.0** 또는 **.NET 표준 2.0**으로 변경합니다.

또한 프로젝트를 마우스 오른쪽 단추로 클릭하고 ***.csproj 파일 편집** 옵션을 선택하여 변경할 수 있습니다. 자세한 내용은 이 항목 앞 부분의 도구 섹션을 참조하세요.

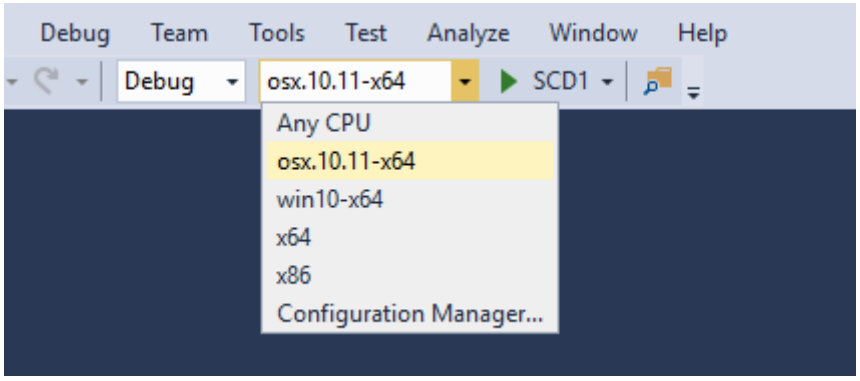
.NET Core에 대한 Live Unit Testing 지원

Live Unit Testing이 코드를 수정하려면 언제든지 자동으로 백그라운드에서 영향을 받는 단위 테스트를 실행하고 Visual Studio 환경에서 결과 및 코드 검사 라이브를 표시합니다. 이제 .NET Core 2.0은 Live Unit Testing을 지원합니다. 이전에 Live Unit Testing은 .NET Framework 애플리케이션에만 사용할 수 있었습니다.

자세한 내용은 Visual Studio에서 Live Unit Testing 및 Live Unit Testing FAQ를 참조하세요.

다양한 대상 프레임워크에 대한 지원 향상

다양한 대상 프레임워크에 대한 프로젝트를 빌드하는 경우 이제 최상위 메뉴에서 대상 플랫폼을 선택할 수 있습니다. 다음 그림에서 SCD1이라는 프로젝트는 64비트 macOS X 10.11(`osx.10.11-x64`) 및 64비트 Windows 10/Windows Server 2016(`win10-x64`)을 대상으로 지정합니다. 디버그 빌드를 실행하는 경우에 프로젝트 단추를 선택하기 전에 대상 프레임워크를 선택할 수 있습니다.



.NET Core SDK에 대한 병렬 지원

이제 Visual Studio와 독립적으로 .NET Core SDK를 설치할 수 있습니다. 따라서 단일 버전의 Visual Studio에서 다른 버전의 .NET Core를 대상으로 지정하는 프로젝트를 빌드할 수 있습니다. 이전에 Visual Studio 및 .NET Core SDK는 밀접하게 연결되어 있었습니다. 특정 버전의 SDK는 특정 버전의 Visual Studio와 함께 제공되었습니다.

설명서 개선

.NET 애플리케이션 아키텍처

빌드하는 데 .NET을 사용하는 경우 .NET 애플리케이션 아키텍처는 지침, 모범 사례 및 애플리케이션 예제를 제공하는 eBook 집합에 액세스 권한을 부여합니다.

- 마이크로 서비스 및 Docker 컨테이너
- ASP.NET을 사용하여 개발한 웹 애플리케이션
- Azure에서 클라우드에 배포되는 애플리케이션

참고 항목

- ASP.NET Core 2.0의 새로운 기능

.NET Standard의 새로운 기능

아티클 • 2024. 03. 11.

.NET Standard는 해당 버전의 표준에 부합하는 .NET 구현체에서 사용할 수 있어야 하는 버전이 지정된 API 세트를 정의합니다. .NET Standard는 라이브러리 개발자에서 대상으로 지정됩니다. .NET Standard 버전을 대상으로 하는 라이브러리는 해당 버전의 표준을 지원하는 모든 .NET 또는 Xamarin 구현에서 사용할 수 있습니다.

NET 표준은 .NET SDK에 포함되어 있습니다. .NET 워크로드를 선택하는 경우에도 Visual Studio에 포함됩니다.

.NET Standard 2.1은 마지막으로 릴리스되는 .NET Standard 버전입니다. 자세한 내용은 .NET 5 이상 및 .NET Standard를 참조하세요.

지원되는 .NET 구현체

.NET Standard 2.1은 다음 .NET 구현에서 지원됩니다.

- .NET Core 3.0 이상(.NET 5 이상 포함)
- Mono 6.4 이상
- Xamarin.iOS 12.16 이상
- Xamarin.Android 10.0 이상

.NET Standard 2.0은 다음 .NET 구현체에서 지원됩니다.

- .NET Core 2.0 이상(.NET 5 이상 포함)
- .NET Framework 4.6.1 이상
- Mono 5.4 이상
- Xamarin.iOS 10.14 이상
- Xamarin.Mac 3.8 이상
- Xamarin.Android 8.0 이상
- 유니버설 Windows 플랫폼 10.0.16299 이상

.NET Standard 2.1의 새로운 기능

.NET Standard 2.1은 표준에 많은 API를 추가합니다. 그 중 일부는 새로운 API이고 다른 일부는 .NET 구현을 더욱 통합하는 데 도움이 되는 기존 API입니다. .NET Standard 2.1에 추가된 API의 목록은 .NET Standard 2.1 및 2.0 을 참조하세요.

자세한 내용은 .NET Standard 2.1 발표 블로그 게시물을 참조하세요.

.NET Standard 2.0의 새로운 기능

.NET Standard 2.0에는 다음과 같은 새로운 기능이 포함되어 있습니다.

매우 폭넓은 API 집합

버전 1.6에는 비교적 적은 .NET Standard API가 포함되었습니다. 여기에는 .NET Framework 또는 Xamarin에서 일반적으로 사용된 많은 API가 제외되었습니다. 이로 인해 개발이 복잡해집니다. 개발자들이 여러 .NET 구현을 대상으로 하는 애플리케이션과 라이브러리를 개발할 때 친숙한 API에 대한 적합한 대체 API를 찾아야 하기 때문입니다. .NET Standard 2.0은 이전 Standard 버전인 .NET Standard 1.6에서 제공된 것보다 20,000개 더 많은 API를 추가하여 이 제약 사항을 해결합니다. .NET Standard 2.0에 추가된 API의 목록은 .NET Standard 2.0 및 1.6 을 참조하세요.

.NET Standard 2.0에서 System 네임스페이스에 대해 추가된 기능의 일부는 다음과 같습니다.

- AppDomain 클래스에 대한 지원.
- Array 클래스에 있는 추가 멤버의 배열을 사용하기 위한 향상된 지원.
- Attribute 클래스에 있는 추가 멤버의 속성을 사용하기 위한 향상된 지원.
- DateTime 값에 대한 향상된 달력 지원 및 추가 서식 옵션.
- 추가 Decimal 반올림 기능.
- Environment 클래스의 추가 기능.
- GC 클래스를 통해 가비지 수집기 제어 강화.
- String 클래스의 문자열 비교, 열거 및 정규화를 위한 향상된 지원.
- TimeZoneInfo.AdjustmentRule 및 TimeZoneInfo.TransitionTime 클래스의 일광 절약 조정 및 전환 시간에 대한 지원.
- Type 클래스의 크게 향상된 기능.
- SerializationInfo 및 StreamingContext 매개 변수를 통해 예외 생성자를 추가하여 예외 개체를 역직렬화하는 기능에 대한 향상된 지원.

.NET Framework 라이브러리에 대한 지원

많은 라이브러리는 .NET Standard가 아닌 .NET Framework를 대상으로 합니다. 그러나 이러한 라이브러리에서 대부분의 호출은 .NET Standard 2.0에 포함된 API를 대상으로 합니다. .NET Standard 2.0부터는 호환성 shim 을 사용하여 .NET Standard 라이브러리에서 .NET Framework 라이브러리에 액세스할 수 있습니다. 이 호환성 레이어는 개발자에게 투명하며, .NET Framework 라이브러리를 이용하기 위해 아무것도 할 필요가 없습니다.

한 가지 요구 사항은 .NET Framework 클래스 라이브러리에서 호출하는 API가 .NET Standard 2.0에 포함되어야 한다는 것입니다.

Visual Basic에 대한 지원

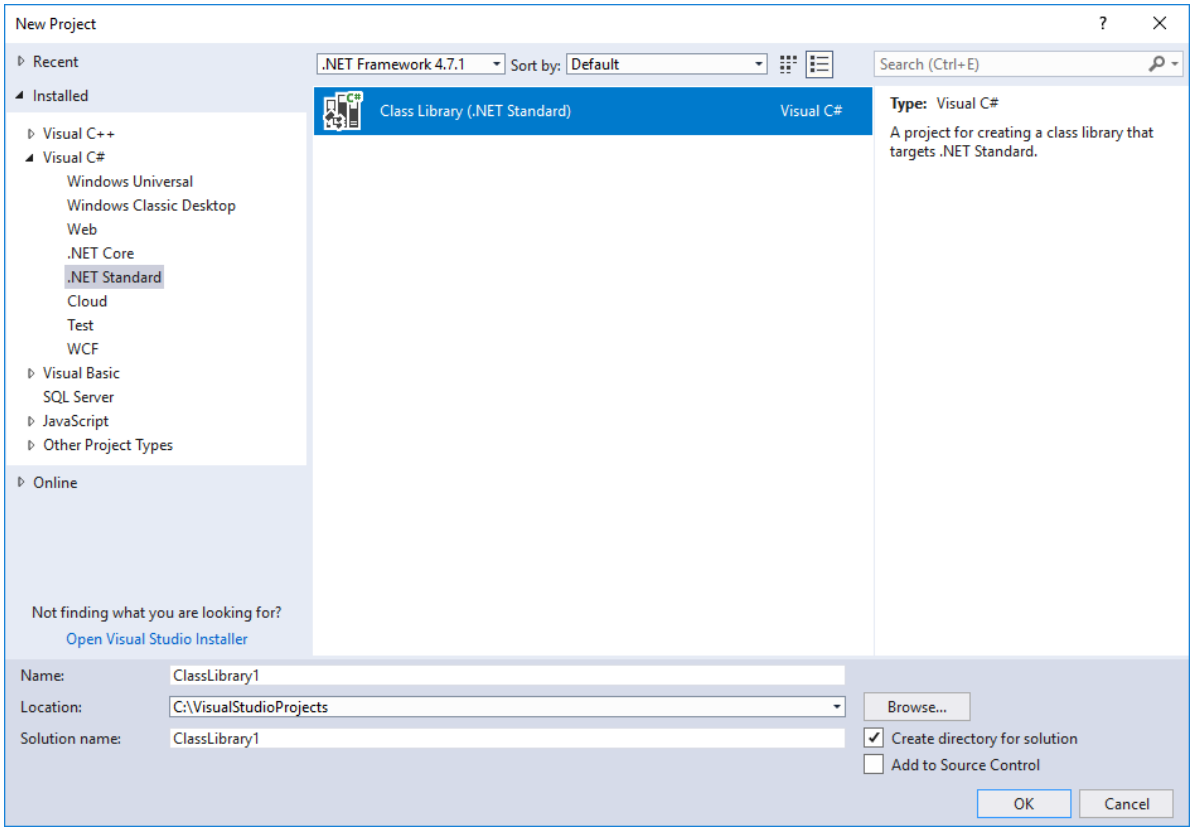
이제 Visual Basic에서 .NET Standard 라이브러리를 개발할 수 있습니다. .NET Core 워크로드가 설치된 Visual Studio 2019 및 Visual Studio 2017 버전 15.3 이상에는 .NET Standard 클래스 라이브러리 템플릿이 포함되어 있습니다. 다른 개발 도구와 환경을 사용하는 Visual Basic 개발자의 경우 dotnet new 명령을 사용하여 .NET Standard 라이브러리 프로젝트를 만들 수 있습니다. 자세한 내용은 .NET Standard 라이브러리에 대한 도구 지원을 참조하세요.

.NET Standard 라이브러리의 도구 지원

.NET Core 2.0 및 .NET Standard 2.0의 출시와 함께 Visual Studio 2017과 .NET CLI 모두 .NET Standard 라이브러리 생성을 위한 도구 지원이 포함됩니다.

.NET Core 플랫폼 간 개발 워크로드를 통해 Visual Studio를 설치하는 경우 다음 그림과 같이 프로젝트 템플릿을 사용하여 .NET Standard 2.0 라이브러리 프로젝트를 만들 수 있습니다.

C#



.NET CLI를 사용하는 경우 다음과 같이 dotnet new 명령을 통해 .NET Standard 2.0을 대상으로 하는 클래스 라이브러리 프로젝트를 만듭니다.

.NET CLI

```
dotnet new classlib
```

참고 항목

- .NET Standard
- .NET Standard 소개
- .NET SDK 다운로드

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

CTS(공용 형식 시스템)

아티클 • 2024. 01. 03.

공용 형식 시스템은 공용 언어 런타임에 형식을 선언하고 사용 및 관리하는 방법을 정의할 뿐 아니라 언어 간 통합에 대한 런타임 지원의 중요한 부분을 차지합니다. 공용 형식 시스템은 다음과 같은 기능을 수행합니다.

- 언어 간 통합, 형식 안전성 및 고성능 코드 실행이 가능한 프레임워크를 만듭니다.
- 다양한 프로그래밍 언어를 완벽하게 구현하는 개체 지향 모델을 제공합니다.
- 다양한 언어로 작성된 개체 간에 상호 작용할 수 있도록 언어에서 따라야 할 규칙을 정의합니다.
- 애플리케이션 개발에 사용되는 기본 데이터 형식(예: Boolean, Byte, Char, Int32 및 UInt64)이 포함된 라이브러리를 제공합니다.

.NET의 형식

.NET의 모든 형식은 값 형식 또는 참조 형식입니다.

값 형식은 해당 형식의 개체가 개체의 실제 값으로 나타나는 데이터 형식입니다. 값 형식의 인스턴스가 변수에 할당되면 해당 변수에는 값의 새 사본이 부여됩니다.

참조 형식은 해당 형식의 개체가 개체의 실제 값에 대한 참조(포인터와 유사)로 나타나는 데이터 형식입니다. 참조 형식이 변수에 할당되면 해당 변수는 원래 값을 참조하거나 가리키며 복사는 수행되지 않습니다.

.NET의 공용 형식 시스템에서는 다음과 같은 다섯 가지 범주의 형식을 지원합니다.

- 클래스
- 구조체
- 열거형
- 인터페이스
- 대리자

클래스

클래스는 다른 클래스에서 직접 파생될 수 있거나 System.Object에서 암시적으로 파생되는 참조 형식입니다. 클래스는 개체(클래스의 인스턴스)가 수행할 수 있는 작업(메서드, 이벤트 또는 속성)과 개체에 포함된 데이터(필드)를 정의합니다. 구현 없이 정의만이 포함

된 인터페이스와는 달리, 클래스에는 대개 정의와 구현이 모두 포함되어 있지만 구현이 없는 멤버가 하나 이상 있을 수도 있습니다.

다음 표에는 클래스가 가질 수 있는 일부 특성에 대한 설명이 나와 있습니다. 런타임을 지원하는 각 언어에는 클래스나 클래스 멤버가 이들 특성 중 하나 이상을 갖고 있음을 표시하는 방식이 있습니다. 그러나 .NET을 대상으로 하는 개별 프로그래밍 언어에서 이 특성들을 모두 사용할 수 있는 것은 아닙니다.

테이블 확장

특성	설명
sealed	이 형식에서 다른 클래스를 파생할 수 없도록 지정합니다.
구현	인터페이스 멤버의 구현을 제공하여 클래스에서 하나 이상의 인터페이스를 사용할 수 있음을 나타냅니다.
abstract	클래스를 인스턴스화할 수 없음을 나타냅니다. 이러한 클래스를 사용하려면 다른 클래스를 파생해야 합니다.
상속	기본 클래스를 지정한 곳에서 클래스의 인스턴스를 사용할 수 있음을 나타냅니다. 기본 클래스에서 상속된 파생 클래스에서는 기본 클래스에서 제공하는 공용 멤버의 구현을 사용할 수도 있고, 파생 클래스의 고유한 구현을 사용하여 공용 멤버의 구현을 재정의할 수도 있습니다.
exported 또는 not exported	클래스를 정의한 어셈블리 밖에서 클래스를 볼 수 있는지의 여부를 지정하며 이 특성은 중첩 클래스에는 적용되지 않고 최상위 클래스에만 적용됩니다.

> **참고**
>
> 클래스는 부모 클래스 또는 구조체 내에 중첩될 수도 있습니다. 중첩된 클래스에도 멤버 특성이 있습니다. 자세한 내용은 **중첩 형식**을 참조하세요.

구현이 없는 클래스 멤버는 추상 멤버입니다. 하나 이상의 멤버가 있는 클래스는 그 자체가 추상적이기 때문에 이 클래스의 새 인스턴스를 만들 수 없습니다. 런타임을 대상으로 하는 일부 언어에서는 추상 멤버가 없는 클래스를 추상으로 표시할 수 있습니다. 파생 클래스에서 상속하거나 재정의할 수 있는 기본 기능의 집합을 캡슐화하려는 경우 상황에 따라 추상 클래스를 사용할 수 있습니다. 추상이 아닌 클래스를 구체적인 클래스라고 합니다.

클래스는 인터페이스를 여러 개 구현할 수 있지만 모든 클래스가 암시적으로 상속되는 System.Object와 함께 단 하나의 기본 클래스에서만 상속될 수 있습니다. 모든 클래스에는 클래스의 새 인스턴스를 초기화하는 생성자가 최소한 하나는 있어야 합니다. 생성자를 명시적으로 정의하지 않으면 대부분의 컴파일러는 자동으로 매개 변수가 없는 생성자를 제공합니다.

구조체

구조체는 System.ValueType에서 암시적으로 파생된 뒤 System.Object에서 파생되는 값 형식입니다. 구조체는 적은 메모리를 요구하는 값을 나타낼 때 유용할 뿐 아니라 강력한 형식의 매개 변수를 갖는 메서드에 값으로 전달되는 매개 변수로 값을 전달할 때도 유용합니다. .NET에서 모든 기본 데이터 형식(Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32 및 UInt64)은 구조체로 정의됩니다.

클래스와 마찬가지로 구조체는 데이터(구조체의 필드)와 해당 데이터에 대해 수행할 수 있는 작업(구조체의 메서드)을 정의합니다. 이는 System.Object 및 System.ValueType 클래스에 정의된 가상 메서드와 값 형식 자체에 정의된 메서드를 비롯하여 구조체에서 메서드를 호출할 수 있음을 의미합니다. 다시 말하면 구조체에는 필드, 속성, 이벤트뿐 아니라 정적 및 비정적 메서드가 포함될 수 있습니다. 구조체의 인스턴스를 만들어 매개 변수로 전달하고 지역 변수로 저장하거나 다른 값 형식 또는 참조 형식의 필드에 저장할 수 있습니다. 구조체는 인터페이스를 구현할 수도 있습니다.

또한 값 형식이 몇 가지 측면에서 클래스와 다릅니다. 먼저, 값 형식은 System.ValueType에서 암시적으로 상속되긴 하지만 형식에서 직접 상속될 수는 없습니다. 마찬가지로 모든 값 형식은 봉인되어 있으므로 이 형식에서 다른 형식을 파생할 수 없습니다. 값 형식에는 생성자도 필요하지 않습니다.

공용 언어 런타임에서는 각 값 형식에 대해 해당 boxed 형식 클래스를 제공합니다. 이 클래스는 값 형식과 동일한 상태 및 동작을 가집니다. 값 형식의 인스턴스가 System.Object 형식의 매개 변수를 허용하는 메서드에 전달될 때는 boxing됩니다. 하지만 참조로 전달되는 매개 변수로 값 형식을 허용하는 메서드 호출에서 컨트롤이 반환될 때는 이 인스턴스가 unboxing됩니다. 즉, 클래스의 인스턴스에서 값 형식의 인스턴스로 다시 변환됩니다. 일부 언어에서는 boxed 형식이 필요할 때 별도의 구문을 사용해야 하지만 그 밖의 언어에서는 필요할 때 자동으로 boxed 형식을 사용합니다. 값 형식을 정의하면 boxed 형식과 unboxed 형식이 모두 정의됩니다.

열거형

열거형은 System.Enum에서 직접 상속되고 내부 기본 형식의 값에 대한 대체 이름을 제공하는 값 형식입니다. 열거형 형식은 이름, 내부 형식 및 필드 집합으로 구성됩니다. 내부 형식은 기본으로 제공되는 부호 있는 정수나 부호 없는 정수 형식(예: Byte, Int32 또는 UInt64) 중 하나여야 합니다. 각 필드는 상수를 나타내는 정적 리터럴 필드입니다. 여러 필드에 동일한 값을 할당할 수 있습니다. 이 경우 리플렉션 및 문자열 변환을 위해 값 중 하나를 기본 열거형 값으로 표시해야 합니다.

내부 형식의 값을 열거형에 할당하거나 반대로도 할당할 수 있으며 이때 런타임에서 캐스팅할 필요가 없습니다. 열거형의 인스턴스를 만들고 열거형의 내부 형식에 정의된 메서

드뿐 아니라 System.Enum의 메서드도 호출할 수 있습니다. 그러나 일부 언어에서는 내부 형식의 인스턴스가 필요한 경우 열거형을 매개 변수로 전달할 수 없고 그 반대로도 전달할 수 없습니다.

다음은 열거형에 적용되는 추가 제약 조건입니다.

- 고유의 메서드를 정의할 수 없습니다.
- 인터페이스를 구현할 수 없습니다.
- 속성 또는 이벤트를 정의할 수 없습니다.
- 열거형은 제네릭 형식 내에 중첩되어 있으므로 단독 제네릭이 아니면 제네릭이 될 수 없습니다. 즉, 열거형에는 자체의 형식 매개 변수를 사용할 수 없습니다.

> **참고**
>
> Visual Basic, C# 및 C++를 사용하여 만들어진 중첩 형식(열거형 포함)은 모든 바깥쪽 제네릭 형식의 형식 매개 변수를 포함하므로 자체의 형식 매개 변수가 없더라도 제네릭입니다. 자세한 내용은 Type.MakeGenericType 참조 항목의 "중첩 형식"을 참조하십시오.

FlagsAttribute 특성은 비트 필드라는 특수한 열거형을 나타냅니다. 런타임 자체에서는 기존의 열거형과 비트 필드를 구분하지 않지만 언어에 따라서는 이를 구분할 수도 있습니다. 이를 구분할 경우 열거형이 아니라 비트 필드에 비트 연산자를 적용하여 명명되지 않은 값을 생성할 수 있습니다. 일반적으로 열거형은 요일, 국가 또는 지역 이름 등과 같은 고유한 요소의 목록에 사용하고, 비트 필드는 `Red And Big And Fast` 같은 조합에서 나타날 수 있는 특성 및 수량 목록에 사용합니다.

다음 예제에서는 비트 필드와 기존의 열거형을 둘 다 사용하는 방법을 보여 줍니다.

C#

```
using System;
using System.Collections.Generic;

// A traditional enumeration of some root vegetables.
public enum SomeRootVegetables
{
    HorseRadish,
    Radish,
    Turnip
}

// A bit field or flag enumeration of harvesting seasons.
[Flags]
public enum Seasons
```

```
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}

public class Example
{
   public static void Main()
   {
       // Hash table of when vegetables are available.
       Dictionary<SomeRootVegetables, Seasons> AvailableIn = new
Dictionary<SomeRootVegetables, Seasons>();

       AvailableIn[SomeRootVegetables.HorseRadish] = Seasons.All;
       AvailableIn[SomeRootVegetables.Radish] = Seasons.Spring;
       AvailableIn[SomeRootVegetables.Turnip] = Seasons.Spring |
            Seasons.Autumn;

       // Array of the seasons, using the enumeration.
       Seasons[] theSeasons = new Seasons[] { Seasons.Summer,
Seasons.Autumn,
            Seasons.Winter, Seasons.Spring };

       // Print information of what vegetables are available each season.
       foreach (Seasons season in theSeasons)
       {
          Console.Write(String.Format(
              "The following root vegetables are harvested in {0}:\n",
              season.ToString("G")));
          foreach (KeyValuePair<SomeRootVegetables, Seasons> item in
AvailableIn)
          {
             // A bitwise comparison.
             if (((Seasons)item.Value & season) > 0)
                 Console.Write(String.Format("  {0:G}\n",
                      (SomeRootVegetables)item.Key));
          }
       }
   }
}
// The example displays the following output:
//    The following root vegetables are harvested in Summer:
//      HorseRadish
//    The following root vegetables are harvested in Autumn:
//      Turnip
//      HorseRadish
//    The following root vegetables are harvested in Winter:
//      HorseRadish
//    The following root vegetables are harvested in Spring:
//      Turnip
```

```
//      Radish
//      HorseRadish
```

인터페이스

인터페이스는 "실행 가능" 관계나 "소유" 관계를 지정하는 계약을 정의합니다. 인터페이스는 비교 및 정렬(IComparable 및 IComparable<T> 인터페이스), 같음 테스트(IEquatable<T> 인터페이스) 또는 컬렉션의 항목 열거(IEnumerable 및 IEnumerable<T> 인터페이스)와 같이 기능을 구현하는 데 사용되는 경우가 많습니다. 인터페이스에는 속성, 메서드 및 이벤트가 있을 수 있으며 이러한 멤버는 모두 추상 멤버입니다. 즉, 인터페이스는 멤버와 시그니처를 정의하지만 각 인터페이스 멤버의 기능은 인터페이스를 구현하는 형식에서 정의해야 합니다. 따라서 인터페이스를 구현하는 모든 클래스나 구조체는 인터페이스에 선언된 추상 멤버에 대한 정의를 제공해야 합니다. 인터페이스를 구현하는 클래스나 구조체에서 하나 이상의 다른 인터페이스를 함께 구현해야 할 수도 있습니다.

인터페이스에는 다음과 같은 제약 조건이 따릅니다.

- 인터페이스의 액세스 가능성은 원하는 대로 선언할 수 있지만 모든 인터페이스 멤버는 공용 액세스 가능성을 가져야 합니다.
- 인터페이스는 생성자를 정의할 수 없습니다.
- 인터페이스는 필드를 정의할 수 없습니다.
- 인터페이스에서는 인스턴스 멤버만을 정의할 수 있습니다. 정적 멤버는 정의할 수 없습니다.

둘 이상의 인터페이스에서 동일한 시그니처의 멤버를 선언할 수 있고 이들 멤버가 별도의 구현을 가질 수 있으므로 각 언어에서는 멤버를 요구하는 인터페이스에 구현을 매핑하기 위한 규칙을 제공해야 합니다.

대리자

대리자는 C++의 함수 포인터와 비슷한 목적으로 사용되는 참조 형식입니다. 대리자는 .NET의 이벤트 처리기와 콜백 함수에 사용됩니다. 함수 포인터와 달리 대리자는 안전하고, 확인할 수 있으며, 형식이 안전합니다. 대리자 형식은 호환되는 시그니처가 포함된 정적 메서드 또는 인스턴스 메서드를 나타낼 수 있습니다.

대리자 매개 변수의 형식이 메서드 매개 변수의 형식보다 제한적인 경우 대리자의 매개 변수는 메서드의 해당 매개 변수와 호환됩니다. 이 경우 대리자로 전달된 인수를 안전하게 메서드로 전달할 수 있습니다.

마찬가지로 메서드의 반환 형식이 대리자의 반환 형식보다 제한적인 경우 대리자의 반환 형식은 메서드의 반환 형식과 호환됩니다. 이 경우 메서드의 반환 값을 안전하게 대리자의 반환 형식으로 캐스팅할 수 있습니다.

예를 들어 IEnumerable 형식의 매개 변수와 Object의 반환 형식을 갖는 대리자는 Object 형식의 매개 변수와 IEnumerable 형식의 반환 값을 갖는 메서드를 나타낼 수 있습니다. 자세한 내용과 예제 코드는 Delegate.CreateDelegate(Type, Object, MethodInfo)를 참조하십시오.

대리자는 이 대리자가 나타내는 메서드에 바인딩됩니다. 대리자는 메서드에 바인딩될 뿐 아니라 개체에도 바인딩될 수 있습니다. 개체는 메서드의 첫 번째 매개 변수를 나타내며 대리자가 호출될 때마다 메서드에 전달됩니다. 메서드가 인스턴스 메서드이면 바인딩된 개체가 암시적 `this` 매개 변수(Visual Basic의 경우 `Me`)로 전달됩니다. 메서드가 정적이면 개체는 메서드의 첫 번째 정식 매개 변수로 전달되며 대리자 시그니처가 나머지 매개 변수와 일치해야 합니다. 자세한 내용과 예제 코드는 System.Delegate를 참조하십시오.

모든 대리자는 System.MulticastDelegate에서 상속받는 System.Delegate에서 상속됩니다. C#, Visual Basic 및 C++ 언어에서는 이러한 형식에서 상속받는 것을 허용하지 않으며, 대신 대리자를 선언하기 위한 키워드를 제공합니다.

대리자는 MulticastDelegate에서 상속하기 때문에 대리자에는 호출 목록이 있습니다. 이 목록에는 대리자가 나타내는 메서드와 대리자가 호출될 때 실행되는 메서드가 표시됩니다. 이 목록의 모든 메서드는 대리자가 호출될 때 제공되는 인수를 받습니다.

> **참고**
>
> 호출 목록에 하나 이상의 메서드가 있는 대리자에 대해서는 반환 형식을 가지고 있더라도 반환 값이 정의되지 않습니다.

대부분의 경우 콜백 메서드와 마찬가지로 대리자는 하나의 메서드만 나타내며, 대리자를 만들고 호출하는 것 외에는 필요한 작업이 없습니다.

.NET에서는 여러 메서드를 나타내는 대리자에 대해 Delegate 및 MulticastDelegate 대리자 클래스의 메서드를 제공하여 대리자 호출 목록에 메서드 추가(Delegate.Combine 메서드), 메서드 제거(Delegate.Remove 메서드), 호출 목록 가져오기 (Delegate.GetInvocationList 메서드) 등과 같은 작업을 지원합니다.

> **참고**
>
> C#, C++ 및 Visual Basic에서는 이벤트 처리기 대리자에 대해 이러한 메서드를 사용할 필요가 없습니다. 이들 언어에서는 이벤트 처리기를 추가하고 제거하기 위한 구문을 제공합니다.

형식 정의입니다.

형식 정의에는 다음과 같은 요소가 포함됩니다.

- 형식에 정의되어 있는 특성
- 형식의 액세스 가능성(표시 여부)
- 형식의 이름
- 형식의 기본 형식
- 형식에서 구현하는 인터페이스
- 형식의 멤버 각각에 대한 정의

특성

특성에서는 추가적인 사용자 정의 메타데이터를 제공합니다. 특성은 어셈블리의 형식에 대한 추가 정보를 저장하거나 디자인 타임 또는 런타임 환경에서 형식 멤버의 동작을 수정하는 데 주로 사용됩니다.

특성은 그 자체가 System.Attribute에서 상속된 클래스입니다. 특성 사용을 지원하는 언어 각각에는 특성을 언어 요소에 적용하기 위한 자체 구문이 있습니다. 특성은 거의 대부분의 언어 요소에 적용될 수 있으며, 특성이 적용될 수 있는 특정 요소는 해당 특성 클래스에 적용되는 AttributeUsageAttribute에 의해 정의됩니다.

형식 액세스 가능성

모든 형식에는 다른 형식에서 액세스할 수 있는지를 제어하는 한정자가 있습니다. 다음 표에서는 런타임에서 지원하는 형식 액세스 가능성에 대해 설명합니다.

테이블 확장

액세스 가능성	설명
public	모든 어셈블리에서 액세스할 수 있는 형식입니다.
어셈블리	해당 어셈블리 안에서만 액세스할 수 있는 형식입니다.

중첩된 형식의 액세스 가능성은 액세스 가능 도메인에 따라 다릅니다. 액세스 가능 도메인은 멤버에 대해 선언된 액세스 가능성 및 한 수준 위 형식의 액세스 가능 도메인에 의해 결정됩니다. 그러나 중첩 형식의 액세스 가능 도메인은 포함하는 형식의 액세스 가능 도메인을 벗어날 수는 없습니다.

프로그램 `M`의 형식 `T`에 선언된 중첩된 멤버 `P`에 대한 액세스 가능 도메인은 다음과 같이 정의됩니다. `M` 자체가 형식일 수도 있습니다.

- `M`에 대해 선언된 액세스 가능성이 `public`인 경우 `M`의 액세스 가능 도메인은 `T`의 액세스 가능 도메인입니다.

- `M`에 대해 선언된 액세스 가능성이 `protected internal`인 경우 `M`의 액세스 가능 도메인은 `T`의 프로그램 텍스트를 가진 `P`의 액세스 가능 도메인과 `T` 외부에 선언된 `P`에서 파생된 모든 형식의 프로그램 텍스트 사이의 교집합 부분입니다.
- `M`에 대해 선언된 액세스 가능성이 `protected`인 경우 `M`의 액세스 가능 도메인은 `T`의 프로그램 텍스트를 가진 `T`의 액세스 가능 도메인과 `T`에서 파생된 모든 형식 사이의 교집합 부분입니다.
- `M`에 대해 선언된 액세스 가능성이 `internal`인 경우 `M`의 액세스 가능 도메인은 `T`의 프로그램 텍스트를 가진 `P`의 액세스 가능 도메인의 교집합 부분입니다.
- `M`에 대해 선언된 액세스 가능성이 `private`인 경우 `M`의 액세스 가능 도메인은 `T`의 프로그램 텍스트입니다.

형식 이름

공용 형식 시스템에서 형식 이름에는 두 가지 제약 조건이 있습니다.

- 모든 이름은 유니코드(16비트) 문자의 문자열로 인코딩됩니다.
- 이름에 값 0x0000(16비트)을 포함할 수 없습니다.

그러나 대부분의 언어에서는 형식 이름에 대해 추가적인 제한을 적용합니다. 비교 연산은 바이트 단위로 수행되므로 대/소문자를 구분하며 로캘에 종속되지 않습니다.

형식은 다른 모듈 및 어셈블리의 형식을 참조할 수 있지만 하나의 .NET 모듈 내에서 완전히 정의되어야 합니다. 컴파일러 지원에 따라 차이가 있지만 형식은 여러 소스 코드 파일로 분리될 수 있습니다. 형식 이름은 네임스페이스 내에서만 고유하면 됩니다. 형식을 완전하게 식별하려면 형식의 구현을 포함하는 네임스페이스를 통해 형식 이름을 정규화해야 합니다.

기본 형식 및 인터페이스

형식은 다른 형식에서 값과 동작을 상속할 수 있습니다. 공용 형식 시스템에서는 둘 이상의 기본 형식에서 형식을 상속할 수 없습니다.

형식에서 구현할 수 있는 인터페이스의 수는 제한이 없습니다. 인터페이스를 구현하려면 형식에서 해당 인터페이스의 모든 가상 멤버를 구현해야 합니다. 가상 메서드는 파생된 형식에서 구현할 수 있으며 정적으로 또는 동적으로 호출할 수 있습니다.

형식 멤버

런타임을 사용하면 형식의 동작과 상태를 지정하는 형식의 멤버를 정의할 수 있습니다. 다음과 같은 형식 멤버가 있습니다.

- 필드
- 속성
- 메서드
- 생성자
- 이벤트
- 중첩 형식

필드

필드는 형식의 상태를 설명하고 상태의 일부를 포함합니다. 필드는 런타임에서 지원하는 모든 형식일 수 있습니다. 필드는 대개 `private` 또는 `protected`이므로 클래스 내부나 파생 클래스에서만 액세스할 수 있습니다. 필드의 값을 형식 외부에서 수정할 수 있는 경우 일반적으로 속성 집합 접근자가 사용됩니다. 공개적으로 노출된 필드는 대개 읽기 전용이며 다음 두 형식 중 하나일 수 있습니다.

- 상수. 해당 값은 디자인 타임에 할당됩니다. 상수는 `static`(Visual Basic의 경우 `Shared`) 키워드를 통해 정의되지는 않지만 클래스의 정적 멤버입니다.
- 읽기 전용 변수. 해당 값은 클래스 생성자에서만 할당될 수 있습니다.

다음 예제에서는 이 두 가지 읽기 전용 필드를 사용하는 방법을 보여 줍니다.

C#

```
using System;

public class Constants
{
   public const double Pi = 3.1416;
   public readonly DateTime BirthDate;

   public Constants(DateTime birthDate)
   {
      this.BirthDate = birthDate;
   }
}

public class Example
{
   public static void Main()
   {
      Constants con = new Constants(new DateTime(1974, 8, 18));
      Console.Write(Constants.Pi + "\n");
      Console.Write(con.BirthDate.ToString("d") + "\n");
   }
```

```
}
// The example displays the following output if run on a system whose
current
// culture is en-US:
//    3.1416
//    8/18/1974
```

속성

속성은 형식의 값 또는 상태에 이름을 지정하고 속성의 값을 가져오거나 설정하는 데 사용하는 메서드를 정의합니다. 속성은 기본 형식, 기본 형식의 컬렉션, 사용자 정의 형식 또는 사용자 정의 형식의 컬렉션일 수 있습니다. 속성은 주로 형식의 공용 인터페이스를 형식의 실제 표시와 무관하게 유지하기 위해 사용합니다. 따라서 속성은 클래스에 직접 저장되지 않은 값을 반영하거나(예를 들어 속성이 계산된 값을 반환하는 경우) 값이 전용 필드에 할당되기 전에 유효성 검사를 수행할 수 있습니다. 다음 예제에서는 후자의 패턴을 보여 줍니다.

C#

```
using System;

public class Person
{
   private int m_Age;

   public int Age
   {
      get { return m_Age; }
      set {
         if (value < 0 || value > 125)
         {
            throw new ArgumentOutOfRangeException("The value of the Age
property must be between 0 and 125.");
         }
         else
         {
            m_Age = value;
         }
      }
   }
}
```

읽기 가능한 속성이 포함된 형식의 MSIL(Microsoft Intermediate Language)에는 속성 자체뿐만 아니라 `get_`*propertyname* 메서드도 포함되어 있으며 쓰기 가능한 속성이 포함된 형식의 MSIL에는 `set_`*propertyname* 메서드가 포함되어 있습니다.

메서드

메서드는 형식에 대해 수행할 수 있는 작업을 설명합니다. 메서드의 시그니처에서는 모든 매개 변수 및 반환 값에 허용되는 형식을 지정합니다.

대부분의 메서드는 메서드 호출에 필요한 매개 변수 개수를 정확하게 정의하지만, 일부 메서드는 일정하지 않은 수의 매개 변수를 지원합니다. 이러한 메서드에 대해 선언된 최종 매개 변수는 ParamArrayAttribute 특성으로 표시됩니다. 일반적으로 언어 컴파일러는 C#의 `params` 및 Visual Basic의 `ParamArray` 등과 같이 ParamArrayAttribute를 명시적으로 사용할 필요가 없게 설정하는 키워드를 제공합니다.

생성자

생성자는 클래스 또는 구조체의 새 인스턴스를 만드는 특수한 메서드입니다. 다른 메서드와 마찬가지로 생성자는 매개 변수를 포함할 수 있지만 반환 값은 가지지 않습니다. 즉, `void`를 반환합니다.

클래스의 소스 코드에서 생성자를 명시적으로 정의하지 않은 경우 컴파일러는 매개 변수가 없는 생성자를 포함합니다. 그러나 클래스의 소스 코드에서 매개 변수가 있는 생성자만 정의하는 경우 Visual Basic 및 C# 컴파일러는 매개 변수가 없는 생성자를 생성하지 않습니다.

구조의 소스 코드에서 생성자를 정의하는 경우 해당 생성자에는 매개 변수가 있어야 합니다. 구조에서는 생성자(매개 변수가 없는 생성자)를 정의할 수 없으며 컴파일러는 구조 또는 다른 값 형식에 대해 매개 변수가 없는 생성자를 생성하지 않습니다. 모든 값 형식에는 암시적 매개 변수 없는 생성자가 있습니다. 이 생성자는 공용 언어 런타임에서 구현되며 구조의 모든 필드를 기본값으로 초기화합니다.

이벤트

이벤트는 응답할 수 있는 인시던트를 정의하고 이벤트를 구독, 구독 취소 및 발생시키기 위한 메서드를 정의합니다. 이벤트는 주로 다른 형식에 상태 변경 내용을 알려주는 데 사용합니다. 자세한 내용은 이벤트를 참조하세요.

중첩 형식

중첩 형식은 다른 형식의 멤버인 형식을 나타냅니다. 중첩 형식은 포함하는 형식과 밀접하게 결합되어야 하고 일반 용도의 형식으로 사용하면 안 됩니다. 중첩 형식은 해당 선언 형식에서 중첩 형식의 인스턴스를 만들고 사용하며 중첩 형식의 사용이 공용 멤버에게 노출되지 않는 경우에 유용합니다.

중첩 형식은 일부 개발자에게 혼란을 줄 수 있으므로 불가피한 경우를 제외하고는 공개적으로 표시하지 않아야 합니다. 잘 디자인된 라이브러리에서 개발자는 개체를 인스턴스화하거나 변수를 선언할 때 중첩 형식을 거의 사용하지 않아야 합니다.

형식 멤버의 특성

공용 형식 시스템에서 형식 멤버는 다양한 특성을 가질 수 있지만 언어에서 이런 특성을 모두 지원할 필요는 없습니다. 다음 표에서는 멤버 특성에 대해 설명합니다.

테이블 확장

특성	적용 대상	설명
abstract	메서드, 속성 및 이벤트	형식에서 메서드 구현을 제공하지 않습니다. 추상 메서드를 구현하거나 상속하는 형식에서 메서드에 대한 구현을 제공해야 합니다. 그러나 파생된 형식 자체가 추상 형식인 경우만은 예외입니다. 모든 추상 메서드는 가상 메서드입니다.
private, family, assembly, family와 assembly, family나 assembly 또는 public	모두	멤버의 액세스 가능성을 정의합니다. private 멤버와 동일한 형식 또는 중첩된 형식 내에서만 액세스할 수 있습니다. family 멤버와 동일한 형식 내에서, 그리고 그 멤버에서 상속된 파생된 형식에서 액세스할 수 있습니다. 어셈블리 형식이 정의된 어셈블리에서만 액세스할 수 있습니다. family and assembly 패밀리와 어셈블리 모두에 대한 액세스 자격이 있는 형식에서만 액세스할 수 있습니다. family or assembly 패밀리 또는 어셈블리에 대한 액세스 자격이 있는 형식에서만 액세스할 수 있습니다. public 모든 형식에서 액세스할 수 있습니다.
final	메서드, 속성 및 이벤트	가상 메서드는 파생된 형식에서 재정의할 수 없습니다.
initialize-only	필드	값을 초기화할 수만 있고 초기화 이후에는 작성할 수 없습니다.

특성	적용 대상	설명
인스턴스	필드, 메서드, 속성 및 이벤트	`static`(C# 및 C++), `Shared`(Visual Basic), `virtual`(C# 및 C++) 또는 `Overridable`(Visual Basic)로 표시되지 않은 멤버는 instance 키워드가 없는 인스턴스 멤버입니다. 멤버를 사용하는 개체 수만큼의 멤버 복사본이 메모리에 있습니다.
리터럴	필드	필드에 할당되는 값은 컴파일 타임에 알려지는 기본 제공 값 형식의 고정 값입니다. 리터럴 필드를 때로는 상수라고도 합니다.
newslot 또는 override	모두	멤버가 시그니처가 같은 상속된 멤버와 상호 작용하는 방법을 정의합니다. 새 슬롯 시그니처가 같은 상속된 멤버를 숨깁니다. override 상속된 가상 메서드의 정의를 바꿉니다. 기본값은 새 슬롯입니다.
정적	필드, 메서드, 속성 및 이벤트	멤버는 형식의 특정 인스턴스가 아니라 멤버가 정의된 형식에 속합니다. 멤버는 형식의 인스턴스가 작성되지 않은 경우에도 존재하며 형식의 모든 인스턴스 간에 공유됩니다.
virtual	메서드, 속성 및 이벤트	메서드는 파생된 형식에서 구현할 수 있으며 정적으로 또는 동적으로 호출할 수 있습니다. 동적 호출을 사용하는 경우, 메서드의 어떤 구현을 호출할지 결정하는 것은 컴파일 타임에 알려진 형식이 아니라 런타임에 호출을 수행하는 인스턴스의 형식입니다. 가상 메서드를 정적으로 호출하려면 원하는 버전의 메서드를 사용하는 형식으로 변수를 캐스팅해야 합니다.

오버로딩

각 형식 멤버에는 고유의 시그니처가 있습니다. 메서드 시그니처는 메서드 이름과 매개 변수 목록(메서드 인수의 순서 및 형식)으로 구성됩니다. 형식 내에서 시그니처를 다르게 하여 이름이 같은 여러 개의 메서드를 정의할 수 있습니다. 이름이 같은 둘 이상의 메서드를 정의하면 메서드가 오버로드됩니다. 예를 들어, System.Char에서는 IsDigit 메서드가 오버로드됩니다. 한 메서드는 Char을 사용하고, 다른 메서드는 String 및 Int32를 사용합니다.

> **참고**
>
> 반환 형식은 메서드 시그니처의 일부로 간주되지 않습니다. 즉, 메서드는 반환 형식만 다를 경우에는 오버로드될 수 없습니다.

멤버 상속, 재정의 및 숨기기

파생된 형식에서는 기본 형식의 멤버를 모두 상속합니다. 즉, 파생된 형식에서 이러한 멤버를 정의하고 사용할 수 있습니다. 상속된 멤버의 동작 또는 품질을 다음과 같은 두 가지 방법으로 수정할 수 있습니다.

- 파생된 형식에서 동일한 시그니처를 가진 새 멤버를 정의하여 상속 멤버를 숨길 수 있습니다. 이는 이전의 공용 멤버를 전용 멤버로 변경하거나 `final`로 표시된 상속된 메서드에 대해 새로운 동작을 정의하기 위해 수행될 수 있습니다.
- 상속된 가상 메서드를 파생된 형식에서 재정의할 수 있습니다. 메서드를 재정의하면 컴파일 타임에 알려진 변수의 형식이 아니라 런타임에 값의 형식을 기반으로 호출되는 메서드에 대한 새 정의가 제공됩니다. 가상 메서드가 `final`로 표시되어 있지 않고 새 메서드에서 최소한 가상 메서드와 동일한 수준의 액세스 가능성을 갖는 경우에만 가상 메서드를 재정의할 수 있습니다.

참조

- .NET API 브라우저
- 공용 언어 런타임
- .NET에서 형식 변환

언어 독립성 및 언어 독립적 구성 요소

아티클 • 2025. 03. 30.

.NET은 언어 독립적입니다. 즉, 개발자는 C#, F# 및 Visual Basic과 같은 .NET 구현을 대상으로 하는 여러 언어 중 하나로 개발할 수 있습니다. 원래 작성된 언어를 알 필요 없이 원래 언어의 규칙을 따르지 않고도 .NET 구현을 위해 개발된 클래스 라이브러리의 형식 및 멤버에 액세스할 수 있습니다. 구성 요소 개발자인 경우 해당 언어에 관계없이 모든 .NET 앱에서 구성 요소에 액세스할 수 있습니다.

> **참고**
>
> 이 문서의 첫 번째 부분에서는 언어 독립적 구성 요소, 즉 모든 언어로 작성된 앱에서 사용할 수 있는 구성 요소를 만드는 방법에 대해 설명합니다. 여러 언어로 작성된 소스 코드에서 단일 구성 요소 또는 앱을 만들 수도 있습니다. 이 문서의 두 번째 부분에서 **언어 간 상호 운용성** 참조하세요.

모든 언어로 작성된 다른 개체와 완전히 상호 작용하려면 개체가 모든 언어에 공통적인 기능만 호출자에게 노출해야 합니다. 이러한 일반적인 기능 집합은 생성된 어셈블리에 적용되는 규칙 집합인 CLS(*공용 언어 사양*)에 의해 정의됩니다. 공용 언어 사양은 ECMA-335 Standard: 공용 언어 인프라 파티션 I, 절 7~11에 정의되어 있습니다.

구성 요소가 공용 언어 사양을 준수하는 경우 CLS 규격으로 보장되며 CLS를 지원하는 프로그래밍 언어로 작성된 어셈블리의 코드에서 액세스할 수 있습니다.
CLSCompliantAttribute 특성을 소스 코드에 적용하여 구성 요소가 컴파일 시간에 공용 언어 사양을 준수하는지 여부를 확인할 수 있습니다. 자세한 내용은 CLSCompliantAttribute 특성참조하세요.

CLS 규정 준수 규칙

이 섹션에서는 CLS 규격 구성 요소를 만드는 규칙에 대해 설명합니다. 규칙의 전체 목록은 파티션 I, ECMA-335 표준의 절 11: 공용 언어 인프라 참조하세요.

> **참고**
>
> 공용 언어 사양은 소비자(CLS 규격 구성 요소에 프로그래밍 방식으로 액세스하는 개발자), 프레임워크(언어 컴파일러를 사용하여 CLS 규격 라이브러리를 만드는 개발자) 및 확장기(언어 컴파일러 또는 CLS 규격 구성 요소를 만드는 코드 파서와 같은 도구를 만드는 개발자)에 적용되는 CLS 규정 준수에 대한 각 규칙을 설명합니다. 이 문

서에서는 프레임워크에 적용되는 규칙에 중점을 둡니다. 그러나 extender에 적용되는 일부 규칙은 Reflection.Emit사용하여 만든 어셈블리에도 적용될 수 있습니다.

언어 독립적 구성 요소를 디자인하려면 CLS 규정 준수 규칙을 구성 요소의 공용 인터페이스에만 적용하면 됩니다. 프라이빗 구현은 사양을 준수할 필요가 없습니다.

> ⓘ **중요**
>
> CLS 규정 준수 규칙은 프라이빗 구현이 아닌 구성 요소의 공용 인터페이스에만 적용됩니다.

예를 들어 Byte 이외의 부호 없는 정수는 CLS 규격이 아닙니다. 다음 예제의 `Person` 클래스는 UInt16형식의 `Age` 속성을 노출하므로 다음 코드는 컴파일러 경고를 표시합니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private UInt16 personAge = 0;

   public UInt16 Age
   { get { return personAge; } }
}
// The attempt to compile the example displays the following compiler
warning:
//    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-
compliant
```

`Age` 속성의 형식을 UInt16에서 CLS 규격의 16비트 부호 있는 정수인 Int16로 변경하여 `Person` 클래스를 CLS 규격으로 만들 수 있습니다. 프라이빗 `personAge` 필드의 형식을 변경할 필요가 없습니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private Int16 personAge = 0;

   public Int16 Age
```

```
    { get { return personAge; } }
}
```

라이브러리의 공용 인터페이스는 다음으로 구성됩니다.

- 공용 클래스의 정의입니다.
- 공용 클래스의 공용 멤버 정의 및 파생 클래스(즉, 보호된 멤버)에 액세스할 수 있는 멤버의 정의입니다.
- public 클래스의 public 메서드의 매개 변수 및 반환 형식, 파생 클래스에 액세스할 수 있는 메서드의 매개 변수 및 반환 형식입니다.

CLS 준수에 대한 규칙은 다음 표에 나와 있습니다. 규칙의 텍스트는 에크마 인터내셔널의 2012 저작권을 가진 ECMA-335 표준: 공용 언어 인프라 에서 문자 그대로 가져온 것입니다. 이러한 규칙에 대한 자세한 내용은 다음 섹션에서 확인할 수 있습니다.

테이블 확장

카테고리	보십시오	규칙	규칙 번호
접근성	멤버 접근성	상속된 메서드를 재정의할 때는 접근성이 변경되지 않아야 하지만, 다른 어셈블리에서 상속된 메서드의 경우 접근성 `family-or-assembly`을 가진 메서드를 재정의할 때는 예외입니다. 이 경우, 재정의에는 접근성 `family`가 적용됩니다.	10
접근성	멤버 접근성	형식과 멤버의 가시성과 접근성은 멤버 자체가 가시적이고 접근 가능할 때, 그 멤버의 시그니처에 있는 형식이 항상 가시적이고 접근 가능하도록 해야 합니다. 예를 들어 어셈블리 외부에서 보이는 public 메서드에는 어셈블리 내부에서만 보이는 형식의 인수가 있어서는 안 됩니다. 멤버의 서명에 사용되는 인스턴스화된 제네릭 형식을 구성하는 형식의 표시 유형과 접근성은 멤버 자체가 표시되고 액세스할 수 있을 때마다 표시되고 액세스할 수 있어야 합니다. 예를 들어 어셈블리 외부에 표시되는 멤버의 서명에 있는 인스턴스화된 제네릭 형식에는 해당 형식이 어셈블리 내에서만 표시되는 제네릭 인수가 있어서는 안 됩니다.	12
배열	배열	배열에는 CLS 규격 형식의 요소가 있어야 하며 배열의 모든 차원은 하한이 0입니다. 항목이 배열이고 배열의 요소 형식이 오버로드를 구분해야 한다는 사실만 있으면 됩니다. 오버로드가 둘 이상의 배열 형식을 기반으로 하는 경우 요소 형식은 명명된 형식이어야 합니다.	16
특성	특성	특성은 System.Attribute형식이거나 상속되는 형식이어야 합니다.	41
특성	특성	CLS는 사용자 지정 특성의 인코딩 하위 집합만 허용합니다. 이러한 인코딩에 표시되는 유일한 형식은 (파티션 IV 참조) System.Type, System.String,	34

카테고리	보십시오	규칙	규칙 번호
		System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double및 CLS 규격 기본 정수 형식을 기반으로 하는 열거형 형식입니다.	
특성	특성	CLS는 공개적으로 표시되는 필수 한정자(`modreq`, 파티션 II 참조)를 허용하지 않지만, 이해하지 못하는 선택적 한정자(`modopt`, 파티션 II 참조)를 허용합니다.	35
생성자	생성자	개체 생성자는 상속된 인스턴스 데이터에 액세스하기 전에 해당 기본 클래스의 일부 인스턴스 생성자를 호출해야 합니다. (생성자가 필요하지 않은 값 형식에는 적용되지 않습니다.)	21
생성자	생성자	개체 생성자는 개체 생성의 일부로 호출되지 않으며 개체는 두 번 초기화되지 않습니다.	22
열거	열거형	열거형의 기본 형식은 기본 제공 CLS 정수 형식이어야 하며 필드 이름은 "value__"이어야 하며 해당 필드는 `RTSpecialName`표시되어야 합니다.	7
열거	열거형	System.FlagsAttribute 사용자 지정 특성의 유무(파티션 IV 라이브러리 참조)에 의해 구분되는 두 가지 종류의 열거형이 있습니다. 하나는 명명된 정수 값을 나타냅니다. 다른 하나는 명명되지 않은 값을 생성하기 위해 결합할 수 있는 명명된 비트 플래그를 나타냅니다. `enum` 값은 지정된 값으로 제한되지 않습니다.	8
열거	열거형	열거형의 리터럴 정적 필드에는 열거형 자체의 형식이 있어야 합니다.	9
이벤트	이벤트	이벤트를 구현하는 메서드는 메타데이터에 `SpecialName` 표시되어야 합니다.	29
이벤트	이벤트	이벤트 및 해당 접근자의 접근성은 동일해야 합니다.	30
이벤트	이벤트	이벤트에 대한 `add` 및 `remove` 메서드는 둘 다 존재해야 하거나 존재하지 않아야 합니다.	31
이벤트	이벤트	이벤트에 대한 `add` 및 `remove` 메서드는 각각 이벤트 형식을 정의하는 하나의 매개 변수를 가져와야 하며, 이 매개 변수는 System.Delegate파생되어야 합니다.	32
이벤트	이벤트	이벤트는 특정 명명 패턴을 준수해야 합니다. CLS 규칙 29에서 참조되는 SpecialName 특성은 적절한 이름 비교에서 무시되어야 하며 식별자 규칙을 준수해야 합니다.	33

카테고리	보십시오	규칙	규칙 번호
예외	예외	throw되는 개체는 System.Exception 형식이거나 상속되는 형식이어야 합니다. 그럼에도 불구하고 CLS 규격 메서드는 다른 유형의 예외 전파를 차단할 필요가 없습니다.	40
일반	CLS 준수 규정	CLS 규칙은 정의 어셈블리 외부에서 액세스할 수 있거나 표시되는 형식의 해당 부분에만 적용됩니다.	1
일반	CLS 준수 기준	비 CLS 규격 형식의 멤버는 CLS 규격으로 표시되지 않습니다.	2
제네릭	제네릭 형식 및 멤버	중첩 형식에는 바깥쪽 형식에 비해 적어도 같은 수의 제네릭 매개 변수가 있어야 합니다. 중첩 형식의 제네릭 매개 변수는 바깥쪽 형식의 제네릭 매개 변수에 대한 위치에 해당합니다.	42
제네릭	제네릭 형식 및 멤버	제네릭 형식의 이름은 위에 정의된 규칙에 따라 중첩되지 않은 형식에 선언되거나 중첩된 경우 새로 형식에 도입된 형식 매개 변수 수를 인코딩해야 합니다.	43
제네릭	제네릭 형식 및 멤버	제네릭 형식은 기본 형식 또는 인터페이스에 대한 제약 조건이 제네릭 형식 제약 조건에 의해 충족되도록 충분한 제약 조건을 다시 묶어야 합니다.	44
제네릭	제네릭 형식 및 멤버	제네릭 매개 변수의 제약 조건으로 사용되는 형식은 CLS 규격이어야 합니다.	45
제네릭	제네릭 형식 및 멤버	인스턴스화된 제네릭 형식의 멤버(중첩 형식 포함)의 표시 유형 및 접근성은 제네릭 형식 선언 전체가 아닌 특정 인스턴스화로 범위가 지정된 것으로 간주되어야 합니다. 이 경우 CLS 규칙 12의 표시 유형 및 접근성 규칙이 계속 적용됩니다.	46
제네릭	제네릭 형식 및 멤버	각 추상 또는 가상 제네릭 메서드에 대해 기본 구체적인(nonabstract) 구현이 있어야 합니다.	47
인터페	인터페이스	CLS 규격 인터페이스를 구현하기 위해 CLS 규격이 아닌 메서드의 정의가 필요하지 않습니다.	18

카테고리	보십시오	규칙	규칙 번호
이스			
인터페이스	인터페이스	CLS 규격 인터페이스는 정적 메서드를 정의하거나 필드를 정의하지 않습니다.	19
구성원	일반 형식 멤버	전역 정적 필드 및 메서드는 CLS 규격이 아닙니다.	36
구성원	--	리터럴 정적 값은 필드 초기화 메타데이터를 사용하여 지정됩니다. CLS 규격 리터럴은 필드 초기화 메타데이터에 지정된 값이 리터럴과 정확히 동일한 형식(또는 해당 리터럴이 `enum`경우 기본 형식)이어야 합니다.	13
구성원	일반 유형 멤버	vararg 제약 조건은 CLS의 일부가 아니며 CLS에서 지원하는 유일한 호출 규칙은 표준 관리되는 호출 규칙입니다.	15
명명 규칙	명명 규칙	어셈블리는 유니코드 표준3.0의 기술 보고서 15 부록 7을 따라 시작 및 식별자에 포함할 수 있는 문자 집합을 제어해야 하며, 유니코드 정규화 양식 온라인으로 제공됩니다. 식별자는 유니코드 정규화 양식 C로 정의된 정식 형식이어야 합니다. CLS의 경우 소문자 매핑(유니코드 로캘을 구분하지 않는 일 대 일 소문자 매핑으로 지정됨)이 동일한 경우 두 식별자가 동일합니다. 즉, CLS에서 두 식별자가 서로 다른 것으로 간주되려면 단순히 해당 사례보다 더 많이 달라야 합니다. 그러나 상속된 정의를 재정의하려면 CLI를 사용하려면 원래 선언의 정확한 인코딩을 사용해야 합니다.	4
오버로딩	명명 규칙	CLS 규격 범위에 도입된 모든 이름은 이름이 동일하고 오버로드를 통해 확인되는 경우를 제외하고 종류에 관계없이 고유해야 합니다. 즉, CTS를 사용하면 단일 형식이 메서드와 필드에 대해 동일한 이름을 사용할 수 있지만 CLS는 사용하지 않습니다.	5
오버로딩	명명 규칙	CTS에서 고유 서명을 구분할 수 있더라도 필드와 중첩 형식은 식별자 비교만으로 고유해야 합니다. 이름이 같은 메서드, 속성 및 이벤트(식별자 비교 기준)는 CLS 규칙 39에 지정된 경우를 제외하고 반환 형식보다 더 많이 달라야 합니다.	6
오버로딩	오버로드	속성 및 메서드만 오버로드할 수 있습니다.	37

카테고리	보십시오	규칙	규칙 번호
오버로딩	오버로드	속성 및 메서드는 매개 변수의 수와 형식에 따라 오버로드할 수 있습니다. 단, `op_Implicit` 및 `op_Explicit`변환 연산자는 반환 형식에 따라 오버로드될 수도 있습니다.	38
오버로딩	--	형식에 선언된 둘 이상의 CLS 규격 메서드의 이름이 같고 특정 형식 인스턴스화 집합의 경우 매개 변수와 반환 형식이 같으면 이러한 모든 메서드는 해당 형식 인스턴스화에서 의미 체계적으로 동일해야 합니다.	48
속성	속성	속성의 getter 및 setter 메서드를 구현하는 메서드는 메타데이터에 `SpecialName` 표시되어야 합니다.	24
속성	속성	속성의 접근자는 모두 정적이거나, 모두 가상이거나, 모두 인스턴스여야 합니다.	26
속성	속성	속성의 형식은 getter의 반환 형식과 setter의 마지막 인수 형식이어야 합니다. 속성의 매개 변수 형식은 getter에 대한 매개 변수의 형식과 setter의 최종 매개 변수를 제외한 모든 매개 변수의 형식이어야 합니다. 이러한 형식은 모두 CLS 규격이어야 하며 관리 포인터가 아니어야 합니다(즉, 참조로 전달되어서는 안 됩니다).	27
속성	속성	속성은 특정 명명 패턴을 준수해야 합니다. CLS 규칙 24에 언급된 `SpecialName` 특성은 적절한 이름 비교에서 무시되어야 하며 식별자 규칙을 준수해야 합니다. 속성에는 getter 메서드, setter 메서드 또는 둘 다 있어야 합니다.	28
형식 변환	형식 변환	op_Implicit 또는 op_Explicit이 제공된 경우, 강제 변환을 위한 대체 방법이 제공되어야 합니다.	39
유형	형식 및 형식 멤버 서명	Boxed 값 형식은 CLS 규격을 따르지 않습니다.	3
유형	형식 및 형식 멤버 서명	서명에 표시되는 모든 형식은 CLS 규격이어야 합니다. 인스턴스화된 제네릭 형식을 구성하는 모든 형식은 CLS 규격이어야 합니다.	11
유형	형식 및 형식 멤	형식화된 참조는 CLS 규격이 아닙니다.	14

카테고리	보십시오	규칙	규칙 번호
	버 서명		
유형	형식 및 형식 멤버 서명	관리되지 않는 포인터 형식은 CLS 규격이 아닙니다.	17
유형	형식 및 형식 멤버 서명	CLS 규격 클래스, 값 형식 및 인터페이스는 CLS 규격이 아닌 멤버를 구현할 필요가 없습니다.	20
유형	형식 및 형식 멤버 서명	System.Object CLS 규격입니다. 다른 CLS 준수하는 클래스는 반드시 CLS 준수하는 클래스에서 상속되어야 합니다.	23

하위 섹션에 대한 인덱스:

- 형식과 멤버 서명
- 명명 규칙
- 형식 변환
- 배열
- 인터페이스
- 열거형
- 일반 형식 멤버
- 멤버 접근성
- 제네릭 형식 및 멤버
- 생성자
- 속성
- 이벤트
- 오버로드
- 예외
- 특성

형식 및 형식 멤버 서명

System.Object 형식은 CLS 규격이며 .NET 형식 시스템의 모든 개체 형식의 기본 형식입니다. .NET의 상속은 암시적(예: String 클래스가 `Object` 클래스에서 암시적으로 상속됨) 또는 명시적(예: CultureNotFoundException 클래스는 Exception 클래스에서 명시적으로 상속되는 ArgumentException 클래스에서 명시적으로 상속됨)입니다. 파생 형식이 CLS 규격이 되려면 해당 기본 형식도 CLS 규격이어야 합니다.

다음 예제에서는 기본 형식이 CLS 규격이 아닌 파생 형식을 보여 줍니다. 부호 없는 32비트 정수는 카운터로 사용하는 기본 `Counter` 클래스를 정의합니다. 클래스는 부호 없는 정수 래핑을 통해 카운터 기능을 제공하므로 클래스는 CLS 규격이 아닌 것으로 표시됩니다. 따라서 파생 클래스인 `NonZeroCounter`CLS 규격도 아닙니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
   UInt32 ctr;

   public Counter()
   {
      ctr = 0;
   }

   protected Counter(UInt32 ctr)
   {
      this.ctr = ctr;
   }

   public override string ToString()
   {
      return String.Format("{0}). ", ctr);
   }

   public UInt32 Value
   {
      get { return ctr; }
   }

   public void Increment()
   {
      ctr += (uint) 1;
   }
}

public class NonZeroCounter : Counter
{
   public NonZeroCounter(int startIndex) : this((uint) startIndex)
```

```
    {
    }

    private NonZeroCounter(UInt32 startIndex) : base(startIndex)
    {
    }
}
// Compilation produces a compiler warning like the following:
//    Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter'
is not
//            CLS-compliant
//    Type3.cs(7,14): (Location of symbol related to previous warning)
```

메서드의 반환 형식 또는 속성 형식을 포함하여 멤버 서명에 표시되는 모든 형식은 CLS 규격이어야 합니다. 또한 제네릭 형식의 경우:

- 인스턴스화된 제네릭 형식을 구성하는 모든 형식은 CLS 규격이어야 합니다.
- 제네릭 매개 변수의 제약 조건으로 사용되는 모든 형식은 CLS 규격이어야 합니다.

.NET 공용 형식 시스템 공용 언어 런타임에서 직접 지원되고 어셈블리의 메타데이터에 특별히 인코딩되는 많은 기본 제공 형식을 포함합니다. 이러한 내장 형식 중 다음 표에 나열된 형식은 CLS 규격입니다.

테이블 확장

CLS 규격 유형	설명
바이트	부호 없는 8비트 정수
int16	부호 있는 16비트 정수
int32	부호가 있는 32비트 정수
Int64	부호 있는 64비트 정수
절반	반정밀도 부동 소수점 값
단일	단정밀도 부동 소수점 값
이중	배정밀도 부동 소수점 값
Boolean	true 또는 false 값 형식
문자	UTF-16으로 인코딩된 코드 단위
10진수	부동 소수점이 아닌 십진수
IntPtr	플랫폼 정의 크기의 포인터 또는 핸들
String	0개, 하나 또는 그 이상의 문자 개체 모음

다음 표에 나열된 내장 형식은 CLS 규격이 아닙니다.

테이블 확장

비준수 형식	설명	CLS 규격 준수 대안
SByte	부호 있는 8비트 정수 데이터 유형	int16
UInt16	부호 없는 16비트 정수	int32
UInt32	부호 없는 32비트 정수	Int64
UInt64	부호 없는 64비트 정수	Int64(오버플로할 수 있음), BigInteger또는 Double
UIntPtr	서명되지 않은 포인터 또는 핸들	IntPtr

.NET 클래스 라이브러리 또는 다른 클래스 라이브러리에는 CLS 규격이 아닌 다른 형식이 포함될 수 있습니다. 예를 들면 다음과 같습니다.

- 박싱된 값 타입. 다음 C# 예제에서는 `Value` 이름이 `int*` 형식의 public 속성이 있는 클래스를 만듭니다. `int*` boxed 값 형식이므로 컴파일러는 CLS 규격이 아닌 것으로 플래그를 지정합니다.

```
C#
```

```
using System;

[assembly:CLSCompliant(true)]

public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this
example:
//       warning CS3003: Type of 'TestClass.Value' is not CLS-
compliant
```

- 형식화된 참조는 개체에 대한 참조와 형식에 대한 참조를 포함하는 특수 구문입니다. 형식화된 참조는 .NET에서 TypedReference 클래스로 표시됩니다.

형식이 CLS 규격이 아닌 경우 CLSCompliantAttribute 특성을 `isCompliant` 값이 `false`로 적용해야 합니다. 자세한 내용은 CLSCompliantAttribute 특성 섹션을 참조하세요.

다음 예제에서는 메서드 서명 및 제네릭 형식 인스턴스화에서 CLS 준수 문제를 보여 줍니다. UInt32형식의 속성, `Nullable<UInt32>`형식의 속성 및 형식 UInt32 및 `Nullable<UInt32>`매개 변수가 있는 생성자를 사용하여 `InvoiceItem` 클래스를 정의합니다. 이 예제를 컴파일하려고 하면 4개의 컴파일러 경고가 표시됩니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<uint> qty;

   public InvoiceItem(uint sku, Nullable<uint> quantity)
   {
      itemId = sku;
      qty = quantity;
   }

   public Nullable<uint> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public uint InvoiceId
   {
      get { return invId; }
      set { invId = value; }
   }
}
// The attempt to compile the example displays the following output:
//    Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-
compliant
//    Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-
compliant
//    Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not
CLS-compliant
//    Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is
not CLS-compliant
```

컴파일러 경고를 제거하려면 `InvoiceItem` 공용 인터페이스의 CLS 규격이 아닌 형식을 규격 형식으로 바꿉니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<int> qty;

   public InvoiceItem(int sku, Nullable<int> quantity)
   {
      if (sku <= 0)
         throw new ArgumentOutOfRangeException("The item number is zero or
negative.");
      itemId = (uint) sku;

      qty = quantity;
   }

   public Nullable<int> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public int InvoiceId
   {
      get { return (int) invId; }
      set {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is
zero or negative.");
         invId = (uint) value; }
   }
}
```

나열된 특정 형식 외에도 일부 형식 범주는 CLS 규격이 아닙니다. 여기에는 관리되지 않는 포인터 형식 및 함수 포인터 형식이 포함됩니다. 다음 예제에서는 정수에 대한 포인터를 사용하여 정수 배열을 만들기 때문에 컴파일러 경고를 생성합니다.

C#

```
using System;

[assembly: CLSCompliant(true)]
```

```
public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not
CLS-compliant
```

CLS 규격 추상 클래스(즉, C#에서 `abstract` 또는 Visual Basic에서 `MustInherit` 표시된 클래스)의 경우 클래스의 모든 멤버도 CLS 규격이어야 합니다.

명명 규칙

일부 프로그래밍 언어는 대/소문자를 구분하지 않으므로 식별자(예: 네임스페이스, 형식 및 멤버의 이름)는 대/소문자 이상이어야 합니다. 소문자 매핑이 동일한 경우 두 식별자는 동등한 것으로 간주됩니다. 다음 C# 예제에서는 `Person` 및 `person` 두 개의 공용 클래스를 정의합니다. 대/소문자만 다르기 때문에 C# 컴파일러는 이는 CLS 규격 준수 대상이 아님을 표시합니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{
}

public class person
{
}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

네임스페이스, 형식 및 멤버의 이름과 같은 프로그래밍 언어 식별자는 유니코드 표준 준수해야 합니다. 즉, 다음을 의미합니다.

- 식별자의 첫 번째 문자는 유니코드 대문자, 소문자, 제목 대/소문자, 한정자 문자, 기타 문자 또는 문자 번호일 수 있습니다. 유니코드 문자 범주에 대한 자세한 내용은 System.Globalization.UnicodeCategory 열거형을 참조하세요.
- 후속 문자는 첫 번째 문자와 동일한 범주에 속할 수 있으며, 비간격 문자, 간격 조합 문자, 십진수, 연결 부호, 서식 코드를 포함할 수도 있습니다.

식별자를 비교하기 전에 여러 UTF-16으로 인코딩된 코드 단위로 단일 문자를 나타낼 수 있으므로 서식 코드를 필터링하고 식별자를 유니코드 정규화 양식 C로 변환해야 합니다. 유니코드 정규화 양식 C에서 동일한 코드 단위를 생성하는 문자 시퀀스는 CLS 규격이 아닙니다. 다음 예제에서는 ANGSTROM SIGN(U+212B) 문자로 구성된 `Å`속성과 LATIN CAPITAL LETTER A WITH RING ABOVE(U+00C5)로 구성된 `Å`라는 두 번째 속성을 정의합니다. C# 컴파일러와 Visual Basic 컴파일러 모두 소스 코드에 CLS 규격이 아닌 것으로 플래그를 지정합니다.

C#

```
public class Size
{
   private double a1;
   private double a2;

   public double Å
   {
       get { return a1; }
       set { a1 = value; }
   }

   public double Å
   {
       get { return a2; }
       set { a2 = value; }
   }
}
// Compilation produces a compiler warning like the following:
//    Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only
in case is not
//            CLS-compliant
//    Naming2a.cs(10,18): (Location of symbol related to previous warning)
//    Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing
only in case is not
//            CLS-compliant
//    Naming2a.cs(12,8): (Location of symbol related to previous warning)
//    Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing
only in case is not
```

```
//            CLS-compliant
//    Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

특정 범위 내의 멤버 이름(예: 어셈블리 내의 네임스페이스, 네임스페이스 내의 형식 또는 형식 내의 멤버)은 오버로드를 통해 확인되는 이름을 제외하고 고유해야 합니다. 이 요구 사항은 공통 형식 시스템의 요구 사항보다 더 엄격하므로 범위 내의 여러 멤버가 서로 다른 종류의 멤버인 경우 동일한 이름을 가질 수 있습니다(예: 하나는 메서드이고 다른 멤버는 필드임). 특히 타입 멤버의 경우:

- 필드 및 중첩 형식은 이름만으로 구분됩니다.
- 이름이 같은 메서드, 속성 및 이벤트는 반환 형식뿐만 아니라 더 많은 차이가 있어야 합니다.

다음 예제에서는 멤버 이름이 해당 범위 내에서 고유해야 한다는 요구 사항을 보여 줍니다. `Conversion`이라는 이름을 가진 네 명의 멤버를 포함하는 `Converter` 클래스를 정의합니다. 세 가지는 메서드이고, 하나는 속성입니다. Int64 매개 변수를 포함하는 메서드의 이름은 고유하지만 반환 값은 멤버 서명의 일부로 간주되지 않으므로 Int32 매개 변수가 있는 두 메서드는 이름이 지정되지 않습니다. `Conversion` 속성은 오버로드된 메서드와 동일한 이름을 가질 수 없으므로 이 요구 사항을 위반합니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
   public double Conversion(int number)
   {
      return (double) number;
   }

   public float Conversion(int number)
   {
      return (float) number;
   }

   public double Conversion(long number)
   {
      return (double) number;
   }

   public bool Conversion
   {
      get { return true; }
   }
}
```

```
// Compilation produces a compiler error like the following:
//    Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a
member called
//            'Conversion' with the same parameter types
//    Naming3.cs(8,18): (Location of symbol related to previous error)
//    Naming3.cs(23,16): error CS0102: The type 'Converter' already contains
a definition for
//            'Conversion'
//    Naming3.cs(8,18): (Location of symbol related to previous error)
```

개별 언어에는 고유 키워드가 포함되므로 공용 언어 런타임을 대상으로 하는 언어는 키워드와 일치하는 식별자(예: 형식 이름)를 참조하기 위한 몇 가지 메커니즘도 제공해야 합니다. 예를 들어 `case` C# 및 Visual Basic의 키워드입니다. 그러나 다음 Visual Basic 예제에서는 여는 중괄호와 닫는 중괄호를 사용하여 `case` 키워드에서 `case` 클래스를 명확하게 구분할 수 있습니다. 그렇지 않으면 예제에서 "키워드가 식별자로 유효하지 않습니다."라는 오류 메시지를 생성하고 컴파일에 실패합니다.

VB

```
Public Class [case]
    Private _id As Guid
    Private name As String

    Public Sub New(name As String)
        _id = Guid.NewGuid()
        Me.name = name
    End Sub

    Public ReadOnly Property ClientName As String
        Get
            Return name
        End Get
    End Property
End Class
```

다음 C# 예제에서는 `@` 기호를 사용하여 언어 키워드에서 식별자를 구분하여 `case` 클래스를 인스턴스화할 수 있습니다. 이 메시지가 없으면 C# 컴파일러에 "예상 형식" 및 "잘못된 식 용어 'case'라는 두 개의 오류 메시지가 표시됩니다."

C#

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
```

```
    }
}
```

형식 변환

공용 언어 사양은 두 가지 변환 연산자를 정의합니다.

- `op_Implicit` - 데이터 또는 정밀도의 손실을 초래하지 않는 변환 확대에 사용됩니다. 예를 들어 Decimal 구조체에는 정수 계열 형식의 값과 Char 값을 Decimal 값으로 변환하는 오버로드된 `op_Implicit` 연산자가 포함됩니다.

- `op_Explicit` 크기(값이 더 작은 범위의 값으로 변환됨) 또는 정밀도의 손실을 초래할 수 있는 축소 변환에 사용됩니다. 예를 들어 Decimal 구조에는 Double 및 Single 값을 Decimal 변환하고 Decimal 값을 정수 값, Double, Single및 Char변환하는 오버로드된 `op_Explicit` 연산자가 포함됩니다.

그러나 모든 언어가 연산자 오버로드 또는 사용자 지정 연산자의 정의를 지원하지는 않습니다. 이러한 변환 연산자를 구현하도록 선택하는 경우 변환을 수행하는 다른 방법도 제공해야 합니다. `From` *Xxx* 및 `To`*Xxx* 메서드를 제공하는 것이 좋습니다.

다음 예제에서는 CLS 규격 암시적 및 명시적 변환을 정의합니다. 부호 없는 배정밀도 부동 소수점 숫자를 나타내는 `UDouble` 클래스를 만듭니다. `UDouble`에서 Double로의 암시적 변환과 `UDouble`에서 Single로, Double에서 `UDouble`로, Single에서 `UDouble`로의 명시적 변환을 제공합니다. 또한 `ToDouble` 메서드를 암시적 변환 연산자 및 `ToSingle`, `FromDouble`및 `FromSingle` 메서드를 명시적 변환 연산자의 대안으로 정의합니다.

C#

```
using System;

public struct UDouble
{
   private double number;

   public UDouble(double value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

      number = value;
   }

   public UDouble(float value)
   {
      if (value < 0)
```

```
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

         number = value;
      }

      public static readonly UDouble MinValue = (UDouble) 0.0;
      public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

      public static explicit operator Double(UDouble value)
      {
         return value.number;
      }

      public static implicit operator Single(UDouble value)
      {
         if (value.number > (double) Single.MaxValue)
            throw new InvalidCastException("A UDouble value is out of range of
the Single type.");

         return (float) value.number;
      }

      public static explicit operator UDouble(double value)
      {
         if (value < 0)
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

         return new UDouble(value);
      }

      public static implicit operator UDouble(float value)
      {
         if (value < 0)
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

         return new UDouble(value);
      }

      public static Double ToDouble(UDouble value)
      {
         return (Double) value;
      }

      public static float ToSingle(UDouble value)
      {
         return (float) value;
      }

      public static UDouble FromDouble(double value)
      {
         return new UDouble(value);
      }
```

```
    public static UDouble FromSingle(float value)
    {
       return new UDouble(value);
    }
}
```

배열

CLS 규격 배열은 다음 규칙을 준수합니다.

- 배열의 모든 차원은 하한이 0이어야 합니다. 다음 예제에서는 하한이 1인 CLS 규격이 아닌 배열을 만듭니다. CLSCompliantAttribute 특성이 있음에도 불구하고 컴파일러는 `Numbers.GetTenPrimes` 메서드에서 반환된 배열이 CLS 규격이 아님을 감지하지 않습니다.

  C#

  ```
  [assembly: CLSCompliant(true)]

  public class Numbers
  {
     public static Array GetTenPrimes()
     {
        Array arr = Array.CreateInstance(typeof(Int32), new int[] {10},
  new int[] {1});
        arr.SetValue(1, 1);
        arr.SetValue(2, 2);
        arr.SetValue(3, 3);
        arr.SetValue(5, 4);
        arr.SetValue(7, 5);
        arr.SetValue(11, 6);
        arr.SetValue(13, 7);
        arr.SetValue(17, 8);
        arr.SetValue(19, 9);
        arr.SetValue(23, 10);

        return arr;
     }
  }
  ```

- 모든 배열 요소는 CLS 규격 형식으로 구성되어야 합니다. 다음 예제에서는 CLS 규격이 아닌 배열을 반환하는 두 가지 메서드를 정의합니다. 첫 번째 값은 UInt32 값의 배열을 반환합니다. 두 번째는 Int32 및 UInt32 값을 포함하는 Object 배열을 반환합니다. 컴파일러는 UInt32 형식 때문에 첫 번째 배열을 비준수로 식별하지만 두 번째 배열에 CLS 규격이 아닌 요소가 포함되어 있음을 인식하지 못합니다.

  C#

```
using System;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static UInt32[] GetTenPrimes()
   {
      uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
      return arr;
   }

   public static Object[] GetFivePrimes()
   {
      Object[] arr = { 1, 2, 3, 5u, 7u };
      return arr;
   }
}
// Compilation produces a compiler warning like the following:
//    Array2.cs(8,27): warning CS3002: Return type of
'Numbers.GetTenPrimes()' is not
//            CLS-compliant
```

- 배열 매개 변수가 있는 메서드에 대한 오버로드 확인은 배열 및 해당 요소 형식에 대한 사실을 기반으로 합니다. 이러한 이유로 오버로드된 `GetSquares` 메서드의 다음 정의는 CLS 규격입니다.

C#

```
using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static byte[] GetSquares(byte[] numbers)
   {
      byte[] numbersOut = new byte[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++) {
         int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
         if (square <= Byte.MaxValue)
            numbersOut[ctr] = (byte) square;
         // If there's an overflow, assign MaxValue to the
corresponding
         // element.
         else
            numbersOut[ctr] = Byte.MaxValue;
      }
      return numbersOut;
   }
```

```
   public static BigInteger[] GetSquares(BigInteger[] numbers)
   {
      BigInteger[] numbersOut = new BigInteger[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         numbersOut[ctr] = numbers[ctr] * numbers[ctr];

      return numbersOut;
   }
}
```

인터페이스

CLS 규격 인터페이스는 속성, 이벤트 및 가상 메서드(구현이 없는 메서드)를 정의할 수 있습니다. CLS 규격 인터페이스에는 다음 중 어느 것이라도 있을 수 없습니다.

- 정적 메서드 또는 정적 필드입니다. 인터페이스에서 정적 멤버를 정의하는 경우 C# 컴파일러와 Visual Basic 컴파일러 모두 컴파일러 오류를 생성합니다.
- 필드. 인터페이스에서 필드를 정의하는 경우 C# 컴파일러와 Visual Basic 컴파일러 모두 컴파일러 오류를 생성합니다.
- CLS 규격이 아닌 메서드입니다. 예를 들어 다음 인터페이스 정의에는 CLS 규격이 아닌 것으로 표시된 메서드 `INumber.GetUnsigned`포함됩니다. 이 예제에서는 컴파일러 경고를 생성합니다.

C#

```
using System;

[assembly:CLSCompliant(true)]

public interface INumber
{
   int Length();
   [CLSCompliant(false)] ulong GetUnsigned();
}
// Attempting to compile the example displays output like the
following:
//    Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()':
CLS-compliant interfaces
//            must have only CLS-compliant members
```

이 규칙 때문에 CLS 규격 형식은 CLS 규격이 아닌 멤버를 구현할 필요가 없습니다. CLS 규격 프레임워크가 비 CLS 규격 인터페이스를 구현하는 클래스를 노출하는 경우 CLS 규격이 아닌 모든 멤버의 구체적인 구현도 제공해야 합니다.

또한 CLS 규격 언어 컴파일러는 클래스가 여러 인터페이스에서 이름과 서명이 같은 멤버의 별도 구현을 제공하도록 허용해야 합니다. C# 및 Visual Basic은 동일한 명명된 메서드의 다른 구현을 제공하기 위해 명시적 인터페이스 구현을 지원합니다. 또한 Visual Basic은 특정 멤버가 구현하는 인터페이스와 멤버를 명시적으로 지정할 수 있는 `Implements` 키워드를 지원합니다. 다음 예제에서는 `ICelsius` 및 `IFahrenheit` 인터페이스를 명시적 인터페이스 구현으로 구현하는 `Temperature` 클래스를 정의하여 이 시나리오를 보여 줍니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
   decimal GetTemperature();
}

public interface ICelsius
{
   decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
   private decimal _value;

   public Temperature(decimal value)
   {
      // We assume that this is the Celsius value.
      _value = value;
   }

   decimal IFahrenheit.GetTemperature()
   {
      return _value * 9 / 5 + 32;
   }

   decimal ICelsius.GetTemperature()
   {
      return _value;
   }
}
public class Example
{
   public static void Main()
   {
      Temperature temp = new Temperature(100.0m);
      ICelsius cTemp = temp;
      IFahrenheit fTemp = temp;
      Console.WriteLine($"Temperature in Celsius: {cTemp.GetTemperature()}
```

```
degrees");
      Console.WriteLine($"Temperature in Fahrenheit:
{fTemp.GetTemperature()} degrees");
   }
}
// The example displays the following output:
//       Temperature in Celsius: 100.0 degrees
//       Temperature in Fahrenheit: 212.0 degrees
```

열거

CLS 규격 열거형은 다음 규칙을 따라야 합니다.

- 열거형의 기본 형식은 내장 CLS 규격 정수(Byte, Int16, Int32또는 Int64)여야 합니다. 예를 들어 다음 코드는 기본 형식이 UInt32 컴파일러 경고를 생성하는 열거형을 정의하려고 합니다.

  C#

  ```
  using System;

  [assembly: CLSCompliant(true)]

  public enum Size : uint {
     Unspecified = 0,
     XSmall = 1,
     Small = 2,
     Medium = 3,
     Large = 4,
     XLarge = 5
  };

  public class Clothing
  {
     public string Name;
     public string Type;
     public string Size;
  }
  // The attempt to compile the example displays a compiler warning like
  the following:
  //    Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not
  CLS-compliant
  ```

- 열거형 형식에는 FieldAttributes.RTSpecialName 특성으로 표시된 `Value__`이라는 단일 인스턴스 필드가 있어야 합니다. 이렇게 하면 필드 값을 암시적으로 참조할 수 있습니다.

- 열거형에는 열거형 자체의 형식과 일치하는 형식이 있는 리터럴 정적 필드가 포함됩니다. 예를 들어 `State.On` 및 `State.Off`값으로 `State` 열거형을 정의하면 `State.On` 및 `State.Off`는 모두 형식이 `State`인 리터럴 정적 필드입니다.

- 열거형에는 두 가지 종류가 있습니다.

  - 서로 배타적인 명명된 정수 값 집합을 나타내는 열거형입니다. 이 유형의 열거형은 System.FlagsAttribute 사용자 지정 특성이 없는 것으로 표시됩니다.

  - 명명되지 않은 값을 생성하기 위해 결합할 수 있는 비트 플래그 집합을 나타내는 열거형입니다. 이 유형의 열거형은 System.FlagsAttribute 사용자 지정 특성의 존재로 표시됩니다.

  자세한 내용은 Enum 구조에 대한 설명서를 참조하세요.

- 열거형의 값은 지정된 값의 범위로 제한되지 않습니다. 즉, 열거형의 값 범위는 기본 값의 범위입니다. Enum.IsDefined 메서드를 사용하여 지정된 값이 열거형의 멤버인지 여부를 확인할 수 있습니다.

일반적인 타입 멤버

공용 언어 사양을 사용하려면 모든 필드와 메서드에 특정 클래스의 멤버로 액세스해야 합니다. 따라서 전역 정적 필드 및 메서드(즉, 정적 필드 또는 형식과 별도로 정의된 메서드)는 CLS 규격이 아닙니다. 소스 코드에 전역 필드 또는 메서드를 포함하려는 경우 C# 컴파일러와 Visual Basic 컴파일러 모두 컴파일러 오류를 생성합니다.

공용 언어 사양은 표준 관리되는 호출 규칙만 지원합니다. `varargs` 키워드로 표시된 변수 인수 목록을 사용하여 관리되지 않는 호출 규칙 및 메서드를 지원하지 않습니다. 표준 관리되는 호출 규칙과 호환되는 변수 인수 목록의 경우 ParamArrayAttribute 특성 또는 C#의 `params` 키워드 및 Visual Basic의 `ParamArray` 키워드와 같은 개별 언어 구현을 사용합니다.

멤버 접근성

상속된 멤버를 재정의하면 해당 멤버의 접근성을 변경할 수 없습니다. 예를 들어 기본 클래스의 public 메서드는 파생 클래스의 private 메서드로 재정의할 수 없습니다. 한 가지 예외는 한 어셈블리에서 다른 어셈블리의 형식에 의해 재정의되는 `protected internal`(C#에서의 경우) 또는 `Protected Friend`(Visual Basic에서의 경우) 멤버입니다. 이 경우 재정의의 접근성은 `Protected`.

다음 예제에서는 CLSCompliantAttribute 특성이 `true`설정되고 `Animal`파생된 클래스인 `Human` `Species` 속성의 접근성을 public에서 private으로 변경하려고 할 때 발생하는 오류

를 보여 줍니다. 이 예제에서는 접근성이 public으로 변경되면 성공적으로 컴파일됩니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
   private string _species;

   public Animal(string species)
   {
      _species = species;
   }

   public virtual string Species
   {
      get { return _species; }
   }

   public override string ToString()
   {
      return _species;
   }
}

public class Human : Animal
{
   private string _name;

   public Human(string name) : base("Homo Sapiens")
   {
      _name = name;
   }

   public string Name
   {
      get { return _name; }
   }

   private override string Species
   {
      get { return base.Species; }
   }

   public override string ToString()
   {
      return _name;
   }
}

public class Example
```

```
{
   public static void Main()
   {
      Human p = new Human("John");
      Console.WriteLine(p.Species);
      Console.WriteLine(p.ToString());
   }
}
// The example displays the following output:
//    error CS0621: 'Human.Species': virtual or abstract members cannot be
private
```

멤버 서명의 형식은 해당 멤버에 액세스할 수 있을 때마다 액세스할 수 있어야 합니다. 예를 들어 공용 멤버는 형식이 private, protected 또는 internal인 매개 변수를 포함할 수 없습니다. 다음 예제에서는 `StringWrapper` 클래스 생성자가 문자열 값을 래핑하는 방법을 결정하는 내부 `StringOperationType` 열거형 값을 노출할 때 발생하는 컴파일러 오류를 보여 줍니다.

C#

```
using System;
using System.Text;

public class StringWrapper
{
   string internalString;
   StringBuilder internalSB = null;
   bool useSB = false;

   public StringWrapper(StringOperationType type)
   {
      if (type == StringOperationType.Normal) {
         useSB = false;
      }
      else {
         useSB = true;
         internalSB = new StringBuilder();
      }
   }

   // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//    error CS0051: Inconsistent accessibility: parameter type
//              'StringOperationType' is less accessible than method
//              'StringWrapper.StringWrapper(StringOperationType)'
```

제네릭 형식 및 멤버

중첩 형식에는 항상 묶는 형식만큼 많은 제네릭 매개 변수가 있습니다. 이 값은 바깥쪽 형식의 제네릭 매개 변수에 대한 위치에 해당합니다. 제네릭 형식에는 새 제네릭 매개 변수도 포함될 수 있습니다.

포함하는 형식의 제네릭 형식 매개 변수와 해당 중첩 형식 간의 관계는 개별 언어의 구문에 의해 숨겨질 수 있습니다. 다음 예제에서 제네릭 형식 `Outer<T>` `Inner1A` 및 `Inner1B<U>` 중첩된 두 클래스를 포함합니다. 각 클래스가 Object.ToString()상속하는 `ToString` 메서드에 대한 호출은 각 중첩 클래스에 포함된 클래스의 형식 매개 변수가 포함되어 있음을 보여 줍니다.

C#

```
using System;

[assembly:CLSCompliant(true)]

public class Outer<T>
{
   T value;

   public Outer(T value)
   {
      this.value = value;
   }

   public class Inner1A : Outer<T>
   {
      public Inner1A(T value) : base(value)
      {  }
   }

   public class Inner1B<U> : Outer<T>
   {
      U value2;

      public Inner1B(T value1, U value2) : base(value1)
      {
         this.value2 = value2;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var inst1 = new Outer<String>("This");
      Console.WriteLine(inst1);
```

```
      var inst2 = new Outer<String>.Inner1A("Another");
      Console.WriteLine(inst2);

      var inst3 = new Outer<String>.Inner1B<int>("That", 2);
      Console.WriteLine(inst3);
   }
}
// The example displays the following output:
//       Outer`1[System.String]
//       Outer`1+Inner1A[System.String]
//       Outer`1+Inner1B`1[System.String,System.Int32]
```

제네릭 형식 이름은 *name*'*n*형식으로 인코딩됩니다. 여기서 *이름* 형식 이름입니다. '는 문자 리터럴이고, *n* 형식에 선언된 매개 변수의 수이거나 중첩된 제네릭 형식의 경우 새로 도입된 형식 매개 변수의 수입니다. 제네릭 형식 이름의 인코딩은 리플렉션을 사용하여 라이브러리의 CLS 불만 제네릭 형식에 액세스하는 개발자에게 주로 중요합니다.

제약 조건이 제네릭 형식에 적용되는 경우 제약 조건으로 사용되는 모든 형식도 CLS 규격이어야 합니다. 다음 예제에서는 CLS 규격이 아닌 `BaseClass`라는 클래스와 형식 매개 변수가 `BaseClass`에서 파생된 `BaseCollection`이라는 제네릭 클래스를 정의합니다. 그러나 `BaseClass` CLS 규격이 아니므로 컴파일러는 경고를 내보낸다.

C#

```
using System;

[assembly:CLSCompliant(true)]

[CLSCompliant(false)] public class BaseClass
{}

public class BaseCollection<T> where T : BaseClass
{}
// Attempting to compile the example displays the following output:
//    warning CS3024: Constraint type 'BaseClass' is not CLS-compliant
```

제네릭 형식이 제네릭 기본 형식에서 파생된 경우 기본 형식의 제약 조건도 충족되도록 제약 조건을 다시 표시해야 합니다. 다음 예제에서는 숫자 형식을 나타낼 수 있는 `Number<T>` 정의합니다. 또한 부동 소수점 값을 나타내는 `FloatingPoint<T>` 클래스를 정의합니다. 그러나 소스 코드는 `FloatingPoint<T>` `Number<T>` 제약 조건(T는 값 형식이어야 합니다)을 적용하지 않으므로 컴파일에 실패합니다.

C#

```
using System;

[assembly:CLSCompliant(true)]
```

```
public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.",
e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value),
typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value),
typeof(T));
   }
}

public class FloatingPoint<T> : Number<T>
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a
floating-point number.");
   }
}
// The attempt to compile the example displays the following output:
//       error CS0453: The type 'T' must be a non-nullable value type in
//                order to use it as parameter 'T' in the generic type or
method 'Number<T>'
```

이 예제에서는 제약 조건이 `FloatingPoint<T>` 클래스에 추가되면 성공적으로 컴파일됩니다.

C#

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.",
e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value),
typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value),
typeof(T));
   }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a
floating-point number.");
```

```
    }
}
```

공용 언어 사양은 중첩된 형식 및 보호된 멤버에 대해 인스턴스화별 보수적 모델을 적용합니다. 열린 제네릭 형식은 중첩된 보호된 제네릭 형식의 특정 인스턴스화를 포함하는 서명이 있는 필드 또는 멤버를 노출할 수 없습니다. 제네릭 기본 클래스 또는 인터페이스의 특정 인스턴스화를 확장하는 제네릭이 아닌 형식은 중첩된 보호된 제네릭 형식의 다른 인스턴스화를 포함하는 서명이 있는 필드 또는 멤버를 노출할 수 없습니다.

다음 예제에서는 제네릭 형식, `C1<T>`(또는 Visual Basic의 `C1(Of T)`) 및 보호된 클래스 `C1<T>.N`(또는 Visual Basic의 `C1(Of T).N`)를 정의합니다. `C1<T>` `M1` 및 `M2`두 가지 메서드가 있습니다. 그러나 `M1` C1<T>(또는 `C1(Of T)`)에서 `C1<int>.N`(또는 `C1(Of Integer).N`) 개체를 반환하려고 하기 때문에 CLS 규격이 아닙니다. `C2`두 번째 클래스는 `C1<long>`(또는 `C1(Of Long)`)에서 파생됩니다. `M3` 및 `M4`두 가지 메서드가 있습니다. `M3` `C1<long>`서브클래스에서 `C1<int>.N`(또는 `C1(Of Integer).N`) 개체를 반환하려고 하기 때문에 CLS 규격이 아닙니다. 언어 컴파일러는 훨씬 더 제한적일 수 있습니다. 이 예제에서 Visual Basic은 `M4`컴파일하려고 할 때 오류를 표시합니다.

C#

```
using System;

[assembly:CLSCompliant(true)]

public class C1<T>
{
   protected class N { }

   protected void M1(C1<int>.N n) { } // Not CLS-compliant - C1<int>.N not
                                      // accessible from within C1<T> in all
                                      // languages
   protected void M2(C1<T>.N n) { }   // CLS-compliant – C1<T>.N accessible
                                      // inside C1<T>
}

public class C2 : C1<long>
{
   protected void M3(C1<int>.N n) { }  // Not CLS-compliant – C1<int>.N is
not
                                       // accessible in C2 (extends
C1<long>)

   protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                       // accessible in C2 (extends
C1<long>)
}
// Attempting to compile the example displays output like the following:
//       Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is
```

```
not CLS-compliant
//       Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is
not CLS-compliant
```

생성자

CLS 규격 클래스 및 구조체의 생성자는 다음 규칙을 따라야 합니다.

- 파생 클래스의 생성자는 상속된 인스턴스 데이터에 액세스하기 전에 해당 기본 클래스의 인스턴스 생성자를 호출해야 합니다. 이 요구 사항은 기본 클래스 생성자가 파생 클래스에서 상속되지 않기 때문입니다. 이 규칙은 직접 상속을 지원하지 않는 구조체에는 적용되지 않습니다.

  일반적으로 컴파일러는 다음 예제와 같이 CLS 규정 준수와 독립적으로 이 규칙을 적용합니다. `Person` 클래스에서 파생된 `Doctor` 클래스를 만들지만 `Doctor` 클래스는 상속된 인스턴스 필드를 초기화하기 위해 `Person` 클래스 생성자를 호출하지 못합니다.

  C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private string fName, lName, _id;

   public Person(string firstName, string lastName, string id)
   {
      if (String.IsNullOrEmpty(firstName + lastName))
         throw new ArgumentNullException("Either a first name or a last
name must be provided.");

      fName = firstName;
      lName = lastName;
      _id = id;
   }

   public string FirstName
   {
      get { return fName; }
   }

   public string LastName
   {
      get { return lName; }
   }
```

```
    public string Id
    {
       get { return _id; }
    }

    public override string ToString()
    {
       return String.Format("{0}{1}{2}", fName,
                            String.IsNullOrEmpty(fName) ?  "" : " ",
                            lName);
    }
}

public class Doctor : Person
{
    public Doctor(string firstName, string lastName, string id)
    {
    }

    public override string ToString()
    {
       return "Dr. " + base.ToString();
    }
}
// Attempting to compile the example displays output like the
following:
//    ctor1.cs(45,11): error CS1729: 'Person' does not contain a
constructor that takes 0
//             arguments
//    ctor1.cs(10,11): (Location of symbol related to previous error)
```

- 개체를 만드는 것 외에는 개체 생성자를 호출할 수 없습니다. 또한 개체를 두 번 초기화할 수 없습니다. 예를 들어 이는 Object.MemberwiseClone 및 역직렬화 메서드가 생성자를 호출하지 않아야 함을 의미합니다.

속성

CLS 규격 형식의 속성은 다음 규칙을 따라야 합니다.

- 속성에는 setter, getter 또는 둘 다 있어야 합니다. 어셈블리에서 이러한 메서드는 특수 메서드로 구현됩니다. 즉, 어셈블리의 메타데이터에서 `SpecialName` 표시된 별도의 메서드(getter는 `get_`*속성 이름* 이름이 지정되고 setter는 속성 이름`set_`)로 표시됩니다. C# 및 Visual Basic 컴파일러는 CLSCompliantAttribute 특성을 적용할 필요 없이 이 규칙을 자동으로 적용합니다.

- 속성의 형식은 속성 getter의 반환 형식 및 setter의 마지막 인수입니다. 이러한 형식은 CLS 규격이어야 하며 인수는 참조로 속성에 할당할 수 없습니다(즉, 관리 포인터일 수 없음).

- 속성에 getter와 setter가 모두 있는 경우 둘 다 가상, 정적 또는 두 인스턴스 모두여야 합니다. C# 및 Visual Basic 컴파일러는 속성 정의 구문을 통해 이 규칙을 자동으로 적용합니다.

이벤트

이벤트는 이름과 형식으로 정의됩니다. 이벤트 유형은 이벤트를 나타내는 데 사용되는 대리자입니다. 예를 들어 AppDomain.AssemblyResolve 이벤트는 ResolveEventHandler형식입니다. 이벤트 자체 외에도 이벤트 이름에 따라 이름이 있는 세 가지 메서드는 이벤트의 구현을 제공하며 어셈블리의 메타데이터에서 `SpecialName` 표시됩니다.

- `add_` *이벤트 처리기*를 추가하는 메서드입니다. 예를 들어 AppDomain.AssemblyResolve 이벤트에 대한 이벤트 구독 메서드의 이름은 `add_AssemblyResolve`.
- `remove_` *eventName*이벤트 처리기를 제거하는 메서드입니다. 예를 들어 AppDomain.AssemblyResolve 이벤트에 대한 제거 메서드의 이름은 `remove_AssemblyResolve`.
- 이벤트가 발생했음을 나타내는 메서드로, `raise_` *이벤트 이름*이입니다.

> **참고**
>
> 이벤트에 대한 공용 언어 사양 규칙의 대부분은 언어 컴파일러에 의해 구현되며 구성 요소 개발자에게 투명합니다.

이벤트를 추가, 제거 및 발생시키는 메서드는 동일한 접근성을 가져야 합니다. 또한 모두 정적, 인스턴스 또는 가상이어야 합니다. 이벤트를 추가하고 제거하는 메서드에는 해당 형식이 이벤트 대리자 형식인 매개 변수가 하나 있습니다. add 및 remove 메서드가 모두 있거나 둘 다 없어야 합니다.

다음 예제에서는 두 판독값 사이의 온도 변화가 임계값과 같거나 초과할 경우 `TemperatureChanged` 이벤트를 발생시키는 `Temperature` 명명된 CLS 규격 클래스를 정의합니다. `Temperature` 클래스는 이벤트 처리기를 선택적으로 실행할 수 있도록 `raise_TemperatureChanged` 메서드를 명시적으로 정의합니다.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;

[assembly: CLSCompliant(true)]
```

```
public class TemperatureChangedEventArgs : EventArgs
{
   private Decimal originalTemp;
   private Decimal newTemp;
   private DateTimeOffset when;

   public TemperatureChangedEventArgs(Decimal original, Decimal @new,
DateTimeOffset time)
   {
      originalTemp = original;
      newTemp = @new;
      when = time;
   }

   public Decimal OldTemperature
   {
      get { return originalTemp; }
   }

   public Decimal CurrentTemperature
   {
      get { return newTemp; }
   }

   public DateTimeOffset Time
   {
      get { return when; }
   }
}

public delegate void TemperatureChanged(Object sender,
TemperatureChangedEventArgs e);

public class Temperature
{
   private struct TemperatureInfo
   {
      public Decimal Temperature;
      public DateTimeOffset Recorded;
   }

   public event TemperatureChanged TemperatureChanged;

   private Decimal previous;
   private Decimal current;
   private Decimal tolerance;
   private List<TemperatureInfo> tis = new List<TemperatureInfo>();

   public Temperature(Decimal temperature, Decimal tolerance)
   {
      current = temperature;
      TemperatureInfo ti = new TemperatureInfo();
      ti.Temperature = temperature;
      tis.Add(ti);
```

```
      ti.Recorded = DateTimeOffset.UtcNow;
      this.tolerance = tolerance;
   }

   public Decimal CurrentTemperature
   {
      get { return current; }
      set {
         TemperatureInfo ti = new TemperatureInfo();
         ti.Temperature = value;
         ti.Recorded = DateTimeOffset.UtcNow;
         previous = current;
         current = value;
         if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new
TemperatureChangedEventArgs(previous, current, ti.Recorded));
      }
   }

   public void raise_TemperatureChanged(TemperatureChangedEventArgs
eventArgs)
   {
      if (TemperatureChanged == null)
         return;

      foreach (TemperatureChanged d in
TemperatureChanged.GetInvocationList()) {
         if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not
executed.");
         else
            d.Invoke(this, eventArgs);
      }
   }
}

public class Example
{
   public Temperature temp;

   public static void Main()
   {
      Example ex = new Example();
   }

   public Example()
   {
      temp = new Temperature(65, 3);
      temp.TemperatureChanged += this.TemperatureNotification;
      RecordTemperatures();
      Example ex = new Example(temp);
      ex.RecordTemperatures();
   }

   public Example(Temperature t)
```

```
    {
        temp = t;
        RecordTemperatures();
    }

    public void RecordTemperatures()
    {
        temp.TemperatureChanged += this.DuplicateTemperatureNotification;
        temp.CurrentTemperature = 66;
        temp.CurrentTemperature = 63;
    }

    internal void TemperatureNotification(Object sender,
TemperatureChangedEventArgs e)
    {
        Console.WriteLine($"Notification 1: The temperature changed from
{e.OldTemperature} to {e.CurrentTemperature}");
    }

    public void DuplicateTemperatureNotification(Object sender,
TemperatureChangedEventArgs e)
    {
        Console.WriteLine($"Notification 2: The temperature changed from
{e.OldTemperature} to {e.CurrentTemperature}");
    }
}
```

과부하

공용 언어 사양은 오버로드된 멤버에 다음 요구 사항을 적용합니다.

- 멤버는 매개 변수 수 및 매개 변수의 형식에 따라 오버로드할 수 있습니다. 호출 규칙, 반환 형식, 메서드 또는 해당 매개 변수에 적용된 사용자 지정 한정자 및 매개 변수가 값으로 전달되는지 또는 참조로 전달되는지 여부는 오버로드를 구분할 때 고려되지 않습니다. 예를 들어 이름이 명명 규칙 섹션의 범위 내에서 고유해야 한다는 요구 사항에 대한 코드를 참조하세요.
- 속성 및 메서드만 오버로드할 수 있습니다. 필드 및 이벤트는 오버로드할 수 없습니다.
- 제네릭 메서드는 제네릭 매개 변수의 수에 따라 오버로드할 수 있습니다.

> ⓘ **참고**
>
> `op_Explicit` 및 `op_Implicit` 연산자는 반환 값이 오버로드 확인을 위한 메서드 서명의 일부로 간주되지 않는 규칙의 예외입니다. 이러한 두 연산자는 매개 변수와 반환 값 모두에 따라 오버로드할 수 있습니다.

예외

예외 개체는 System.Exception 또는 System.Exception파생된 다른 형식에서 파생되어야 합니다. 다음 예제에서는 예외 처리에 `ErrorClass` 명명된 사용자 지정 클래스를 사용할 때 발생하는 컴파일러 오류를 보여 줍니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the
string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
// Compilation produces a compiler error like the following:
//    Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be
derived from
//            System.Exception
```

이 오류를 해결하려면 `ErrorClass` 클래스가 System.Exception상속되어야 합니다. 또한 `Message` 속성을 재정의해야 합니다. 다음 예제에서는 이러한 오류를 수정하여 CLS 규격인 `ErrorClass` 클래스를 정의합니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public override string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the
string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
```

특성

.NET 어셈블리에서 사용자 지정 특성은 사용자 지정 특성을 저장하고 어셈블리, 형식, 멤버 및 메서드 매개 변수와 같은 프로그래밍 개체에 대한 메타데이터를 검색하기 위한 확장 가능한 메커니즘을 제공합니다. 사용자 지정 특성은 System.Attribute 또는 `System.Attribute`System.Attribute에서 파생된 형식에서 파생되어야 합니다.

다음 예제에서는 이 규칙을 위반합니다. System.Attribute파생되지 않는 `NumericAttribute` 클래스를 정의합니다. 컴파일러 오류는 클래스가 정의되는 경우가 아니라 CLS 규격이 아닌 특성이 적용되는 경우에만 발생합니다.

C#

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
   private bool _isNumeric;

   public NumericAttribute(bool isNumeric)
   {
      _isNumeric = isNumeric;
   }

   public bool IsNumeric
   {
      get { return _isNumeric; }
   }
}

[Numeric(true)] public struct UDouble
{
   double Value;
}
// Compilation produces a compiler error like the following:
//    Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an
attribute class
//    Attribute1.cs(7,14): (Location of symbol related to previous error)
```

CLS 규격 특성의 생성자 또는 속성은 다음 형식만 노출할 수 있습니다.

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type

- 어떤 열거형이든 기본 형식이 Byte, Int16, Int32또는 Int64인 경우.

다음 예제에서는 특성파생되는 `DescriptionAttribute` 클래스를 정의합니다. 클래스 생성자에는 `Descriptor`형식의 매개 변수가 있으므로 클래스는 CLS 규격이 아닙니다. C# 컴파일러는 경고를 내보내지만 성공적으로 컴파일되지만 Visual Basic 컴파일러는 경고 또는 오류를 내보내지 않습니다.

C#

```
using System;

[assembly:CLSCompliantAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
   public DescriptorType Type;
   public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
   private Descriptor desc;

   public DescriptionAttribute(Descriptor d)
   {
      desc = d;
   }

   public Descriptor Descriptor
   { get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//       warning CS3015: 'DescriptionAttribute' has no accessible
//                constructors which use only CLS-compliant types
```

CLSCompliantAttribute 속성

CLSCompliantAttribute 특성은 프로그램 요소가 공용 언어 사양을 준수하는지 여부를 나타내는 데 사용됩니다. CLSCompliantAttribute(Boolean) 생성자에는 프로그램 요소가 CLS 규격인지 여부를 나타내는 단일 필수 매개 변수 *isCompliant*포함됩니다.

컴파일 시 컴파일러는 CLS 규격으로 추정되는 비규격 요소를 검색하고 경고를 내보냅니다. 컴파일러는 비규격으로 명시적으로 선언된 형식 또는 멤버에 대한 경고를 내보내지 않습니다.

구성 요소 개발자는 다음 두 가지 방법으로 `CLSCompliantAttribute` 특성을 사용할 수 있습니다.

- CLS 규격인 구성 요소 및 CLS 규격이 아닌 구성 요소에 의해 노출되는 공용 인터페이스의 부분을 정의합니다. 특성이 특정 프로그램 요소를 CLS 규격으로 표시하는 데 사용되는 경우 해당 요소를 사용하여 .NET을 대상으로 하는 모든 언어 및 도구에서 해당 요소에 액세스할 수 있습니다.

- 구성 요소 라이브러리의 공용 인터페이스가 CLS 규격인 프로그램 요소만 노출하도록 합니다. 요소가 CLS 규격이 아닌 경우 컴파일러는 일반적으로 경고를 실행합니다.

> **⚠ 경고**
>
> 경우에 따라 언어 컴파일러는 `CLSCompliantAttribute` 특성이 사용되는지 여부에 관계없이 CLS 규격 규칙을 적용합니다. 예를 들어 인터페이스에서 정적 멤버를 정의하면 CLS 규칙이 위반됩니다. 이와 관련하여 인터페이스에서 `static`(C#) 또는 `Shared`(Visual Basic) 멤버를 정의하는 경우 C# 및 Visual Basic 컴파일러 모두 오류 메시지를 표시하고 앱을 컴파일하지 못합니다.

CLSCompliantAttribute 특성은 AttributeUsageAttribute 특성이 AttributeTargets.All의 값으로 표시됩니다. 이 값을 사용하면 어셈블리, 모듈, 형식(클래스, 구조체, 열거형, 인터페이스 및 대리자), 형식 멤버(생성자, 메서드, 속성, 필드 및 이벤트), 매개 변수, 제네릭 매개 변수 및 반환 값을 비롯한 모든 프로그램 요소에 CLSCompliantAttribute 특성을 적용할 수 있습니다. 그러나 실제로는 어셈블리, 형식 및 형식 멤버에만 특성을 적용해야 합니다. 그렇지 않으면 컴파일러는 특성을 무시하고 라이브러리의 공용 인터페이스에서 비준수 매개 변수, 제네릭 매개 변수 또는 반환 값이 발생할 때마다 컴파일러 경고를 계속 생성합니다.

CLSCompliantAttribute 특성의 값은 포함된 프로그램 요소에 의해 상속됩니다. 예를 들어 어셈블리가 CLS 규격으로 표시된 경우 해당 형식도 CLS 규격입니다. 형식이 CLS 규격으로 표시된 경우 중첩된 형식 및 멤버도 CLS 규격입니다.

포함된 프로그램 요소에 CLSCompliantAttribute 특성을 적용하여 상속된 규정 준수를 명시적으로 재정의할 수 있습니다. 예를 들어, 규격 어셈블리에서 비준수 형식을 정의하려면 CLSCompliantAttribute 특성에 `isCompliant` 값을 `false`으로 사용할 수 있으며, 비준수 어셈블리에서 규격 형식을 정의하려면 `isCompliant` 값이 있는 `true` 특성을 사용할 수 있습니다. 규격 준수 유형 내에서 규격에 맞지 않는 멤버를 정의할 수도 있습니다. 그러나 비준수 형식은 규격 멤버를 가질 수 없으므로 `true`의 값을 가진 `isCompliant` 특성을 사용하여 비준수 형식에서의 상속을 재정의할 수 없습니다.

구성 요소를 개발할 때는 항상 CLSCompliantAttribute 특성을 사용하여 어셈블리, 해당 형식 및 해당 멤버가 CLS 규격인지 여부를 나타내야 합니다.

CLS 규격 구성 요소를 만들려면 다음을 수행합니다.

1. CLSCompliantAttribute 사용하여 어셈블리를 CLS 규격으로 표시합니다.
2. CLS 규격이 아닌 어셈블리에서 공개적으로 노출된 형식을 비준수로 표시합니다.
3. CLS 규격에 부합하는 형식에서 공개적으로 노출된 멤버들을 비규격으로 표시하십시오.
4. CLS 규격이 아닌 멤버에 대한 CLS 규격 대안을 제공합니다.

모든 비준수 형식 및 멤버를 성공적으로 표시한 경우 컴파일러는 비준수 경고를 내보내지 않아야 합니다. 그러나 CLS 규격이 아닌 멤버를 나타내고 제품 설명서에 CLS 규격 대안을 나열해야 합니다.

다음 예제에서는 CLSCompliantAttribute 특성을 사용하여 CLS 규격 어셈블리와 CLS 규격이 아닌 두 멤버가 있는 형식(`CharacterUtilities`)을 정의합니다. 두 멤버 모두 `CLSCompliant(false)` 특성으로 태그가 지정되므로 컴파일러는 경고를 생성하지 않습니다. 또한 이 클래스는 두 메서드 모두에 대한 CLS 규격 대안을 제공합니다. 일반적으로 CLS 규격 대안을 제공하기 위해 `ToUTF16` 메서드에 두 오버로드를 추가하기만 하면 됩니다. 그러나 반환 값에 따라 메서드를 오버로드할 수 없으므로 CLS 규격 메서드의 이름은 비준수 메서드의 이름과 다릅니다.

C#

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
   [CLSCompliant(false)] public static ushort ToUTF16(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return Convert.ToUInt16(s[0]);
   }

   [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
   {
      return Convert.ToUInt16(ch);
   }

   // CLS-compliant alternative for ToUTF16(String).
   public static int ToUTF16CodeUnit(String s)
   {
```

```
      s = s.Normalize(NormalizationForm.FormC);
      return (int) Convert.ToUInt16(s[0]);
   }

   // CLS-compliant alternative for ToUTF16(Char).
   public static int ToUTF16CodeUnit(Char ch)
   {
      return Convert.ToInt32(ch);
   }

   public bool HasMultipleRepresentations(String s)
   {
      String s1 = s.Normalize(NormalizationForm.FormC);
      return s.Equals(s1);
   }

   public int GetUnicodeCodePoint(Char ch)
   {
      if (Char.IsSurrogate(ch))
         throw new ArgumentException("ch cannot be a high or low
surrogate.");

      return Char.ConvertToUtf32(ch.ToString(), 0);
   }

   public int GetUnicodeCodePoint(Char[] chars)
   {
      if (chars.Length > 2)
         throw new ArgumentException("The array has too many characters.");

      if (chars.Length == 2) {
         if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a
high surrogate.");
         else
            return Char.ConvertToUtf32(chars[0], chars[1]);
      }
      else {
         return Char.ConvertToUtf32(chars.ToString(), 0);
      }
   }
}
```

라이브러리가 아닌 앱을 개발하는 경우(즉, 다른 앱 개발자가 사용할 수 있는 형식 또는 멤버를 노출하지 않는 경우) 앱에서 사용하는 프로그램 요소의 CLS 준수는 언어가 지원하지 않는 경우에만 중요합니다. 이 경우 CLS 규격이 아닌 요소를 사용하려고 하면 언어 컴파일러에서 오류가 발생합니다.

언어 간 상호 운용성

언어 독립에는 몇 가지 가능한 의미가 있습니다. 한 가지 의미는 다른 언어로 작성된 앱에서 한 언어로 작성된 형식을 원활하게 사용하는 것입니다. 이 문서의 핵심인 두 번째 의미는 여러 언어로 작성된 코드를 단일 .NET 어셈블리로 결합하는 것입니다.

다음 예제에서는 `NumericLib` 및 `StringLib` 두 클래스를 포함하는 Utilities.dll 클래스 라이브러리를 만들어 언어 간 상호 운용성을 보여 줍니다. `NumericLib` 클래스는 C#으로 작성되고 `StringLib` 클래스는 Visual Basic으로 작성됩니다. 다음은 `StringLib` 클래스에 단일 멤버 `ToTitleCase` 포함하는 `StringUtil.vb` 소스 코드입니다.

VB

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
    Private exclusions As List(Of String)

    Sub New()
        Dim words() As String = {"a", "an", "and", "of", "the"}
        exclusions = New List(Of String)
        exclusions.AddRange(words)
    End Sub

    <Extension()> _
    Public Function ToTitleCase(title As String) As String
        Dim words() As String = title.Split()
        Dim result As String = String.Empty

        For ctr As Integer = 0 To words.Length - 1
            Dim word As String = words(ctr)
            If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
                result += word.Substring(0, 1).ToUpper() + _
                          word.Substring(1).ToLower()
            Else
                result += word.ToLower()
            End If
            If ctr <= words.Length - 1 Then
                result += " "
            End If
        Next
        Return result
    End Function
End Module
```

`IsEven` `NearZero` 두 멤버가 있는 `NumericLib` 클래스를 정의하는 NumberUtil.cs 소스 코드는 다음과 같습니다.

C#

```
using System;

public static class NumericLib
{
   public static bool IsEven(this IConvertible number)
   {
      if (number is Byte ||
          number is SByte ||
          number is Int16 ||
          number is UInt16 ||
          number is Int32 ||
          number is UInt32 ||
          number is Int64)
         return Convert.ToInt64(number) % 2 == 0;
      else if (number is UInt64)
         return ((ulong) number) % 2 == 0;
      else
         throw new NotSupportedException("IsEven called for a non-integer
value.");
   }

   public static bool NearZero(double number)
   {
      return Math.Abs(number) < .00001;
   }
}
```

두 클래스를 단일 어셈블리에 패키지하려면 모듈로 컴파일해야 합니다. Visual Basic 소스 코드 파일을 모듈로 컴파일하려면 다음 명령을 사용합니다.

콘솔

```
vbc /t:module StringUtil.vb
```

Visual Basic 컴파일러의 명령줄 구문에 대한 자세한 내용은 명령줄에서 빌드하기를 참조하세요.

C# 소스 코드 파일을 모듈로 컴파일하려면 다음 명령을 사용합니다.

콘솔

```
csc /t:module NumberUtil.cs
```

그런 다음 링커 옵션 사용하여 두 모듈을 어셈블리로 컴파일합니다.

콘솔

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

다음 예제에서는 `NumericLib.NearZero` 및 `StringLib.ToTitleCase` 메서드를 호출합니다. Visual Basic 코드와 C# 코드는 모두 두 클래스의 메서드에 액세스할 수 있습니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      Double dbl = 0.0 - Double.Epsilon;
      Console.WriteLine(NumericLib.NearZero(dbl));

      string s = "war and peace";
      Console.WriteLine(s.ToTitleCase());
   }
}
// The example displays the following output:
//       True
//       War and Peace
```

Visual Basic 코드를 컴파일하려면 다음 명령을 사용합니다.

콘솔

```
vbc example.vb /r:UtilityLib.dll
```

C#으로 컴파일하려면 컴파일러의 이름을 `vbc` `csc`변경하고 파일 확장명을 *.vb.cs*변경합니다.

콘솔

```
csc example.cs /r:UtilityLib.dll
```

.NET의 형식 변환

아티클 • 2025. 04. 01.

모든 값에는 값에 할당된 공간의 양, 값에 사용할 수 있는 가능한 값 범위 및 사용할 수 있는 멤버와 같은 특성을 정의하는 연결된 형식이 있습니다. 많은 값을 둘 이상의 형식으로 표현할 수 있습니다. 예를 들어 값 4는 정수 또는 부동 소수점 값으로 표현할 수 있습니다. 형식 변환은 이전 형식의 값과 동일하지만 원래 개체의 ID(또는 정확한 값)를 반드시 유지하지는 않는 새 형식의 값을 만듭니다.

.NET은 다음 변환을 자동으로 지원합니다.

- 파생 클래스에서 기본 클래스로 변환합니다. 예를 들어 클래스 또는 구조체의 인스턴스를 Object 인스턴스로 변환할 수 있습니다. 이 변환에는 캐스팅 또는 변환 연산자가 필요하지 않습니다.
- 기본 클래스에서 원래 파생 클래스로 다시 변환합니다. C#에서 이 변환에는 캐스팅 연산자가 필요합니다. Visual Basic에서는 `Option Strict` 있는 경우 `CType` 연산자가 필요합니다.
- 인터페이스를 구현하는 형식에서 해당 인터페이스를 나타내는 인터페이스 개체로 변환합니다. 이 변환에는 캐스팅 또는 변환 연산자가 필요하지 않습니다.
- 인터페이스 개체에서 해당 인터페이스를 구현하는 원래 형식으로 다시 변환합니다. C#에서 이 변환에는 캐스팅 연산자가 필요합니다. Visual Basic에서는 `Option Strict` 있는 경우 `CType` 연산자가 필요합니다.

이러한 자동 변환 외에도 .NET은 사용자 지정 형식 변환을 지원하는 몇 가지 기능을 제공합니다. 여기에는 다음이 포함되었습니다.

- 형식 간의 사용 가능한 확대 변환을 정의하는 `Implicit` 연산자입니다. 자세한 내용은 암시적 연산자 섹션을 사용하여 암시적 변환을 참조하세요.
- 형식 간의 사용 가능한 축소 변환을 정의하는 `Explicit` 연산자입니다. 자세한 내용은 명시적 연산자 섹션을 사용하여 명시적 변환을 참조하세요.
- 각 기본 .NET 데이터 형식에 대한 변환을 정의하는 IConvertible 인터페이스입니다. 자세한 내용은 IConvertible Interface 섹션을 참조하세요.
- IConvertible 인터페이스에서 메서드를 구현하는 메서드 집합을 제공하는 Convert 클래스입니다. 자세한 내용은 클래스 변환 섹션을 참조하세요.
- 지정된 형식을 다른 형식으로 변환할 수 있도록 확장할 수 있는 기본 클래스인 TypeConverter 클래스입니다. 자세한 내용은 TypeConverter 클래스 섹션을 참조하

세요.

암시적 연산자를 사용한 암시적 변환

확대 변환에는 대상 형식보다 더 제한적인 범위 또는 더 제한된 멤버 목록이 있는 기존 형식의 값에서 새 값을 만드는 작업이 포함됩니다. 변환이 확대되면 데이터가 손실될 수 없습니다(정밀도 손실이 발생할 수 있음). 데이터를 손실할 수 없으므로 컴파일러는 명시적 변환 메서드 또는 캐스팅 연산자를 사용하지 않고도 변환을 암시적 또는 투명하게 처리할 수 있습니다.

> **참고**
>
> 암시적 변환을 수행하는 코드는 변환 메서드를 호출하거나 캐스팅 연산자를 사용할 수 있지만 암시적 변환을 지원하는 컴파일러에서는 이 메서드를 사용할 필요가 없습니다.

예를 들어 Decimal 형식은 Byte, Char, Int16, Int32, Int64, SByte, UInt16, UInt32및 UInt64 값에서 암시적 변환을 지원합니다. 다음 예제에서는 Decimal 변수에 값을 할당할 때 이러한 암시적 변환 중 일부를 보여 줍니다.

C#

```
  byte byteValue = 16;
  short shortValue = -1024;
  int intValue = -1034000;
  long longValue = 1152921504606846976;
  ulong ulongValue = UInt64.MaxValue;

  decimal decimalValue;

  decimalValue = byteValue;
  Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
                    byteValue.GetType().Name, decimalValue);

  decimalValue = shortValue;
  Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
                    shortValue.GetType().Name, decimalValue);

  decimalValue = intValue;
  Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
                    intValue.GetType().Name, decimalValue);

  decimalValue = longValue;
  Console.WriteLine("After assigning a {0} value, the Decimal value is
```

```
{1}.",
                      longValue.GetType().Name, decimalValue);

decimalValue = ulongValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  ulongValue.GetType().Name, decimalValue);
// The example displays the following output:
//    After assigning a Byte value, the Decimal value is 16.
//    After assigning a Int16 value, the Decimal value is -1024.
//    After assigning a Int32 value, the Decimal value is -1034000.
//    After assigning a Int64 value, the Decimal value is
1152921504606846976.
//    After assigning a UInt64 value, the Decimal value is
18446744073709551615.
```

특정 언어 컴파일러가 사용자 지정 연산자를 지원하는 경우 사용자 고유의 사용자 지정 형식으로 암시적 변환을 정의할 수도 있습니다. 다음 예제는 부호-크기 표현 방식을 사용하는 `ByteWithSign`라는 이름의 부호 있는 바이트 데이터 유형의 부분 구현을 제공합니다. Byte 및 SByte 값을 `ByteWithSign` 값으로의 암시적 변환을 지원합니다.

C#

```
public struct ByteWithSign
{
    private SByte signValue;
    private Byte value;

    public static implicit operator ByteWithSign(SByte value)
    {
        ByteWithSign newValue;
        newValue.signValue = (SByte)Math.Sign(value);
        newValue.value = (byte)Math.Abs(value);
        return newValue;
    }

    public static implicit operator ByteWithSign(Byte value)
    {
        ByteWithSign newValue;
        newValue.signValue = 1;
        newValue.value = value;
        return newValue;
    }

    public override string ToString()
    {
        return (signValue * value).ToString();
    }
}
```

클라이언트 코드는 다음 예제처럼 명시적 변환이나 캐스팅 연산자를 사용하지 않고 `ByteWithSign` 변수를 선언한 후 Byte 및 SByte 값을 할당할 수 있습니다.

C#

```
SByte sbyteValue = -120;
ByteWithSign value = sbyteValue;
Console.WriteLine(value);
value = Byte.MaxValue;
Console.WriteLine(value);
// The example displays the following output:
//       -120
//       255
```

명시적 연산자를 사용하여 명시적 변환

축소 변환에는 대상 형식보다 범위가 크거나 멤버 목록이 큰 기존 형식의 값에서 새 값을 만드는 작업이 포함됩니다. 축소 변환으로 인해 데이터가 손실될 수 있으므로 컴파일러는 변환 메서드 또는 캐스팅 연산자를 호출하여 변환을 명시적으로 수행해야 하는 경우가 많습니다. 즉, 변환은 개발자 코드에서 명시적으로 처리되어야 합니다.

> **참고**
>
> 변환 범위를 좁히기 위해 변환 방법 또는 캐스팅 연산자를 요구하는 주요 목적은 개발자가 코드에서 처리할 수 있도록 데이터 손실 또는 OverflowException 가능성을 인식하도록 하는 것입니다. 그러나 일부 컴파일러는 이 요구 사항을 완화할 수 있습니다. 예를 들어 Visual Basic에서 `Option Strict` 꺼져 있는 경우(기본 설정) Visual Basic 컴파일러는 축소 변환을 암시적으로 수행하려고 합니다.

예를 들어 UInt32, Int64및 UInt64 데이터 형식에는 다음 표와 같이 Int32 데이터 형식을 초과하는 범위가 있습니다.

테이블 확장

유형	Int32 범위와 비교
Int64	Int64.MaxValue Int32.MaxValue보다 크고 Int64.MinValueInt32.MinValue보다 작습니다(음수 범위가 더 큼).
UInt32	UInt32.MaxValue이 Int32.MaxValue보다 큽니다.
UInt64	UInt64.MaxValue이 Int32.MaxValue보다 크다.

이러한 축소 변환을 처리하기 위해 .NET에서는 형식이 `Explicit` 연산자를 정의할 수 있습니다. 개별 언어 컴파일러는 고유한 구문을 사용하여 이 연산자를 구현하거나 변환을 수행하기 위해 Convert 클래스의 멤버를 호출할 수 있습니다. (Convert 클래스에 대한 자세한 내용은 이 항목의 뒷부분에 있는 Convert 클래스 참조하세요.) 다음 예제에서는 언어 기능을 사용하여 잠재적으로 범위를 벗어난 정수 값을 Int32 값으로 명시적으로 변환하는 방법을 보여 줍니다.

C#

```
long number1 = int.MaxValue + 20L;
uint number2 = int.MaxValue - 1000;
ulong number3 = int.MaxValue;

int intNumber;

try
{
    intNumber = checked((int)number1);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number1.GetType().Name, intNumber);
}
catch (OverflowException)
{
    if (number1 > int.MaxValue)
        Console.WriteLine($"Conversion failed: {number1} exceeds
{int.MaxValue}.");
    else
        Console.WriteLine($"Conversion failed: {number1} is less than
{int.MinValue}.");
}

try
{
    intNumber = checked((int)number2);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number2.GetType().Name, intNumber);
}
catch (OverflowException)
{
    Console.WriteLine($"Conversion failed: {number2} exceeds
{int.MaxValue}.");
}

try
{
    intNumber = checked((int)number3);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number3.GetType().Name, intNumber);
}
```

```
catch (OverflowException)
{
    Console.WriteLine($"Conversion failed: {number1} exceeds
{int.MaxValue}.");
}

// The example displays the following output:
//    Conversion failed: 2147483667 exceeds 2147483647.
//    After assigning a UInt32 value, the Integer value is 2147482647.
//    After assigning a UInt64 value, the Integer value is 2147483647.
```

명시적 변환은 언어마다 다른 결과를 생성할 수 있으며, 이러한 결과는 해당 Convert 메서드에서 반환되는 값과 다를 수 있습니다. 예를 들어 Double 값 12.63251이 Int32변환되는 경우 Visual Basic `CInt` 메서드와 .NET Convert.ToInt32(Double) 메서드는 모두 Double 반올림하여 값을 13으로 반환하지만 C# `(int)` 연산자는 12 값을 반환하기 위해 Double 자른다. 마찬가지로 C# `(int)` 연산자는 부울-정수 변환을 지원하지 않지만 Visual Basic `CInt` 메서드는 `true` 값을 -1로 변환합니다. 반면에 Convert.ToInt32(Boolean) 메서드는 `true` 값을 1로 변환합니다.

대부분의 컴파일러에서는 명시적 변환을 선택되거나 선택되지 않은 방식으로 수행할 수 있습니다. 확인된 변환이 수행되면 변환할 형식의 값이 대상 형식 범위를 벗어나면 예외 OverflowException이 발생합니다. 같은 조건에서 확인되지 않은 변환이 수행되면 변환에서 예외가 발생하지 않을 수 있으나 정확한 동작이 정의되지 않아 잘못된 결과가 나올 수 있습니다.

> **참고**
>
> C#에서는 캐스팅 연산자와 함께 `checked` 키워드를 사용하거나 `/checked+` 컴파일러 옵션을 지정하여 확인된 변환을 수행할 수 있습니다. 반대로 선택되지 않은 변환은 캐스팅 연산자와 함께 `unchecked` 키워드를 사용하거나 `/checked-` 컴파일러 옵션을 지정하여 수행할 수 있습니다. 기본적으로 명시적 변환은 선택되지 않습니다. Visual Basic에서는 프로젝트의 **고급 컴파일러 설정** 대화 상자에서 **정수 오버플로 제거 확인** 확인란을 선택 취소하거나 `/removeintchecks-` 컴파일러 옵션을 지정하여 선택한 변환을 수행할 수 있습니다. 반대로, 프로젝트의 **고급 컴파일러 설정** 대화 상자에서 **정수 오버플로 제거 확인** 확인란을 선택하거나 `/removeintchecks+` 컴파일러 옵션을 지정하여 선택되지 않은 변환을 수행할 수 있습니다. 기본적으로 명시적 변환이 선택됩니다.

다음 C# 예제에서는 `checked` 및 `unchecked` 키워드를 사용하여 Byte 범위를 벗어난 값이 Byte변환되는 경우 동작의 차이를 보여 줍니다. 검사된 변환은 예외를 발생시키지만, 비검사 변환은 Byte.MaxValue를 Byte 변수에 할당합니다.

C#

```
int largeValue = Int32.MaxValue;
byte newValue;

try
{
    newValue = unchecked((byte)largeValue);
    Console.WriteLine($"Converted the {largeValue.GetType().Name} value
{largeValue} to the {newValue.GetType().Name} value {newValue}.");
}
catch (OverflowException)
{
    Console.WriteLine($"{largeValue} is outside the range of the Byte data
type.");
}

try
{
    newValue = checked((byte)largeValue);
    Console.WriteLine($"Converted the {largeValue.GetType().Name} value
{largeValue} to the {newValue.GetType().Name} value {newValue}.");
}
catch (OverflowException)
{
    Console.WriteLine($"{largeValue} is outside the range of the Byte data
type.");
}
// The example displays the following output:
//    Converted the Int32 value 2147483647 to the Byte value 255.
//    2147483647 is outside the range of the Byte data type.
```

특정 언어 컴파일러가 사용자 지정 오버로드된 연산자를 지원하는 경우 사용자 고유의 사용자 지정 형식으로 명시적 변환을 정의할 수도 있습니다. 다음 예제는 부호 및 크기 표현을 사용하는 `ByteWithSign`이라는 이름의 부호 있는 바이트 데이터 형식의 부분 구현을 제공합니다. Int32 및 UInt32 값을 `ByteWithSign` 값으로의 명시적 변환을 지원합니다.

C#

```
public struct ByteWithSignE
{
    private SByte signValue;
    private Byte value;

    private const byte MaxValue = byte.MaxValue;
    private const int MinValue = -1 * byte.MaxValue;

    public static explicit operator ByteWithSignE(int value)
    {
        // Check for overflow.
        if (value > ByteWithSignE.MaxValue || value <
ByteWithSignE.MinValue)
```

```
            throw new OverflowException(String.Format("'{0}' is out of range
of the ByteWithSignE data type.",
                                                      value));

        ByteWithSignE newValue;
        newValue.signValue = (SByte)Math.Sign(value);
        newValue.value = (byte)Math.Abs(value);
        return newValue;
    }

    public static explicit operator ByteWithSignE(uint value)
    {
        if (value > ByteWithSignE.MaxValue)
            throw new OverflowException(String.Format("'{0}' is out of range
of the ByteWithSignE data type.",
                                                      value));

        ByteWithSignE newValue;
        newValue.signValue = 1;
        newValue.value = (byte)value;
        return newValue;
    }

    public override string ToString()
    {
        return (signValue * value).ToString();
    }
}
```

클라이언트 코드는 다음 예제와 같이 `ByteWithSign` 변수를 선언하고, 할당에 캐스팅 연산자 또는 변환 메서드가 포함되는 경우 Int32 및 UInt32 값을 할당할 수 있습니다.

C#

```
ByteWithSignE value;

try
{
    int intValue = -120;
    value = (ByteWithSignE)intValue;
    Console.WriteLine(value);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
    uint uintValue = 1024;
    value = (ByteWithSignE)uintValue;
    Console.WriteLine(value);
}
```

```
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}
// The example displays the following output:
//       -120
//       '1024' is out of range of the ByteWithSignE data type.
```

IConvertible 인터페이스

모든 형식을 공용 언어 런타임 기본 형식으로 변환할 수 있도록 .NET은 IConvertible 인터페이스를 제공합니다. 구현 형식은 다음을 제공하는 데 필요합니다.

- 구현 형식의 TypeCode 반환하는 메서드입니다.
- 구현 형식을 각 공용 언어 런타임 기본 형식(Boolean, Byte, DateTime, Decimal, Double등)으로 변환하는 메서드입니다.
- 구현 형식의 인스턴스를 지정된 다른 형식으로 변환하는 일반화된 변환 방법입니다. 지원되지 않는 변환은 InvalidCastException을 발생시켜야 합니다.

각 공용 언어 런타임 기본 형식(즉, Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, String, UInt16, UInt32및 UInt64)과 DBNull 및 Enum 형식은 물론 IConvertible 인터페이스를 구현합니다. 그러나 명시적 인터페이스 구현은 다음과 같습니다. 변환 메서드는 다음 예제와 같이 IConvertible 인터페이스 변수를 통해서만 호출할 수 있습니다. 다음은 Int32 값을 해당 Char 값으로 변환하는 예제입니다.

C#

```
int codePoint = 1067;
IConvertible iConv = codePoint;
char ch = iConv.ToChar(null);
Console.WriteLine($"Converted {codePoint} to {ch}.");
```

구현 형식이 아닌 인터페이스에서 변환 메서드를 호출해야 하므로 명시적 인터페이스 구현에 상대적으로 비용이 많이 듭니다. 대신 Convert 클래스의 적절한 멤버를 호출하여 공용 언어 런타임 기본 형식 간에 변환하는 것이 좋습니다. 자세한 내용은 다음 섹션 Convert 클래스참조하세요.

> ⓘ **참고**
>
> IConvertible 인터페이스 및 .NET에서 제공하는 Convert 클래스 외에도 개별 언어는 변환을 수행하는 방법을 제공할 수 있습니다. 예를 들어 C#에서는 캐스팅 연산자를

사용합니다. Visual Basic은 `CType`, `CInt` 및 `DirectCast` 같은 컴파일러 구현 변환 함수를 사용합니다.

대부분의 경우 IConvertible 인터페이스는 .NET의 기본 형식 간 변환을 지원하도록 설계되었습니다. 그러나 해당 형식을 다른 사용자 지정 형식으로 변환할 수 있도록 사용자 지정 형식에서 인터페이스를 구현할 수도 있습니다. 자세한 내용은 이 항목의 뒷부분에 있는 ChangeType 메서드 사용자 지정 변환 섹션을 참조하세요.

Convert 클래스

각 기본 형식의 IConvertible 인터페이스 구현을 호출하여 형식 변환을 수행할 수 있지만 System.Convert 클래스의 메서드를 호출하는 것은 한 기본 형식에서 다른 기본 형식으로 변환하는 데 권장되는 언어 중립적 방법입니다. 또한 Convert.ChangeType(Object, Type, IFormatProvider) 메서드를 사용하여 지정된 사용자 지정 형식에서 다른 형식으로 변환할 수 있습니다.

기본 형식 간 변환

Convert 클래스는 기본 형식 간의 변환을 수행하는 언어 중립적 방법을 제공하며 공용 언어 런타임을 대상으로 하는 모든 언어에서 사용할 수 있습니다. 확장 및 축소 변환을 모두 위한 전체 메서드 세트를 제공하며, 지원되지 않는 변환(예: DateTime 값을 정수 값으로 변환)에 대해서는 InvalidCastException 예외를 발생시킵니다. 축소 변환은 확인된 컨텍스트에서 수행되며 변환에 실패하면 OverflowException throw됩니다.

> **ⓘ 중요**
>
> Convert 클래스에는 각 기본 형식으로 변환하는 메서드가 포함되어 있으므로 각 기본 형식의 IConvertible 명시적 인터페이스 구현을 호출할 필요가 없습니다.

다음 예제에서는 System.Convert 클래스를 사용하여 .NET 기본 형식 간에 몇 가지 확대 및 축소 변환을 수행하는 방법을 보여 줍니다.

C#

```
// Convert an Int32 value to a Decimal (a widening conversion).
int integralValue = 12534;
decimal decimalValue = Convert.ToDecimal(integralValue);
Console.WriteLine($"Converted the {integralValue.GetType().Name} value
{integralValue} to " +
                                  "the {decimalValue.GetType().Name} value
{decimalValue:N2}.");
// Convert a Byte value to an Int32 value (a widening conversion).
```

```
byte byteValue = Byte.MaxValue;
int integralValue2 = Convert.ToInt32(byteValue);
Console.WriteLine($"Converted the {byteValue.GetType().Name} value
{byteValue} to " +
                                  "the {integralValue2.GetType().Name} value
{integralValue2:G}.");

// Convert a Double value to an Int32 value (a narrowing conversion).
double doubleValue = 16.32513e12;
try
{
    long longValue = Convert.ToInt64(doubleValue);
    Console.WriteLine($"Converted the {doubleValue.GetType().Name} value
{doubleValue:E} to " +
                                      "the {longValue.GetType().Name} value
{longValue:N0}.");
}
catch (OverflowException)
{
    Console.WriteLine($"Unable to convert the {doubleValue.GetType().Name:E}
value {doubleValue}.");
}

// Convert a signed byte to a byte (a narrowing conversion).
sbyte sbyteValue = -16;
try
{
    byte byteValue2 = Convert.ToByte(sbyteValue);
    Console.WriteLine($"Converted the {sbyteValue.GetType().Name} value
{sbyteValue} to " +
                                      "the {byteValue2.GetType().Name} value
{byteValue2:G}.");
}
catch (OverflowException)
{
    Console.WriteLine($"Unable to convert the {sbyteValue.GetType().Name}
value {sbyteValue}.");
}
// The example displays the following output:
//       Converted the Int32 value 12534 to the Decimal value 12,534.00.
//       Converted the Byte value 255 to the Int32 value 255.
//       Converted the Double value 1.632513E+013 to the Int64 value
16,325,130,000,000.
//       Unable to convert the SByte value -16.
```

경우에 따라, 특히 부동 소수점 값으로 변환할 때 변환은 OverflowExceptionthrow하지 않더라도 정밀도 손실을 포함할 수 있습니다. 다음 예제에서는 이러한 정밀도 손실을 보여 줍니다. 첫 번째 경우 Decimal 값은 Double변환할 때 정밀도(유효 자릿수가 적음)가 적습니다. 두 번째 경우 변환을 완료하기 위해 Double 값이 42.72에서 43으로 반올림됩니다.

```
C#
```

```
double doubleValue;

// Convert a Double to a Decimal.
decimal decimalValue = 13956810.967028881234514711m;
doubleValue = Convert.ToDouble(decimalValue);
Console.WriteLine($"{decimalValue} converted to {doubleValue}.");

doubleValue = 42.72;
try
{
    int integerValue = Convert.ToInt32(doubleValue);
    Console.WriteLine($"{doubleValue} converted to {integerValue}.");
}
catch (OverflowException)
{
    Console.WriteLine($"Unable to convert {doubleValue} to an integer.");
}
// The example displays the following output:
//       13956810.96702881234514711 converted to 13956810.9670289.
//       42.72 converted to 43.
```

Convert 클래스에서 지원하는 확대 및 축소 변환을 모두 나열하는 테이블은 형식 변환 테이블참조하세요.

ChangeType 메서드를 사용하여 사용자 지정 변환

각 기본 형식으로의 변환을 지원하는 것 외에도 Convert 클래스를 사용하여 사용자 지정 형식을 하나 이상의 미리 정의된 형식으로 변환할 수 있습니다. 이 변환은 Convert.ChangeType(Object, Type, IFormatProvider) 메서드에 의해 수행되며, `value` 매개 변수의 IConvertible.ToType 메서드에 대한 호출을 래핑합니다. 즉, `value` 매개 변수로 표시되는 개체는 IConvertible 인터페이스의 구현을 제공해야 합니다.

> ⓘ **참고**
>
> Convert.ChangeType(Object, Type) 및 Convert.ChangeType(Object, Type, IFormatProvider) 메서드는 Type 개체를 사용하여 `value` 변환되는 대상 형식을 지정하기 때문에 컴파일 시간에 형식을 알 수 없는 개체로 동적 변환을 수행하는 데 사용할 수 있습니다. 그러나 IConvertible의 `value` 구현은 여전히 이 변환을 지원해야 합니다.

다음 예제에서는 `TemperatureCelsius` 개체를 `TemperatureFahrenheit` 개체로 변환할 수 있도록 하는 IConvertible 인터페이스의 가능한 구현을 보여 줍니다. 이 예제에서는 IConvertible 인터페이스를 구현하고 Object.ToString 메서드를 재정의하는 기본 클래스

`Temperature`정의합니다. 파생된 `TemperatureCelsius` 및 `TemperatureFahrenheit` 클래스는 각각 기본 클래스의 `ToType` 및 `ToString` 메서드를 재정의합니다.

C#

```
using System;

public abstract class Temperature : IConvertible
{
    protected decimal temp;

    public Temperature(decimal temperature)
    {
        this.temp = temperature;
    }

    public decimal Value
    {
        get { return this.temp; }
        set { this.temp = value; }
    }

    public override string ToString()
    {
        return temp.ToString(null as IFormatProvider) + "º";
    }

    // IConvertible implementations.
    public TypeCode GetTypeCode()
    {
        return TypeCode.Object;
    }

    public bool ToBoolean(IFormatProvider provider)
    {
        throw new InvalidCastException(String.Format("Temperature-to-Boolean
conversion is not supported."));
    }

    public byte ToByte(IFormatProvider provider)
    {
        if (temp < Byte.MinValue || temp > Byte.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the Byte data type.", temp));
        else
            return (byte)temp;
    }

    public char ToChar(IFormatProvider provider)
    {
        throw new InvalidCastException("Temperature-to-Char conversion is
not supported.");
    }
```

```
    public DateTime ToDateTime(IFormatProvider provider)
    {
        throw new InvalidCastException("Temperature-to-DateTime conversion
is not supported.");
    }

    public decimal ToDecimal(IFormatProvider provider)
    {
        return temp;
    }

    public double ToDouble(IFormatProvider provider)
    {
        return (double)temp;
    }

    public short ToInt16(IFormatProvider provider)
    {
        if (temp < Int16.MinValue || temp > Int16.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the Int16 data type.", temp));
        else
            return (short)Math.Round(temp);
    }

    public int ToInt32(IFormatProvider provider)
    {
        if (temp < Int32.MinValue || temp > Int32.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the Int32 data type.", temp));
        else
            return (int)Math.Round(temp);
    }

    public long ToInt64(IFormatProvider provider)
    {
        if (temp < Int64.MinValue || temp > Int64.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the Int64 data type.", temp));
        else
            return (long)Math.Round(temp);
    }

    public sbyte ToSByte(IFormatProvider provider)
    {
        if (temp < SByte.MinValue || temp > SByte.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the SByte data type.", temp));
        else
            return (sbyte)temp;
    }

    public float ToSingle(IFormatProvider provider)
    {
```

```
        return (float)temp;
    }

    public virtual string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°";
    }

    // If conversionType is implemented by another IConvertible method, call
it.
    public virtual object ToType(Type conversionType, IFormatProvider
provider)
    {
        switch (Type.GetTypeCode(conversionType))
        {
            case TypeCode.Boolean:
                return this.ToBoolean(provider);
            case TypeCode.Byte:
                return this.ToByte(provider);
            case TypeCode.Char:
                return this.ToChar(provider);
            case TypeCode.DateTime:
                return this.ToDateTime(provider);
            case TypeCode.Decimal:
                return this.ToDecimal(provider);
            case TypeCode.Double:
                return this.ToDouble(provider);
            case TypeCode.Empty:
                throw new NullReferenceException("The target type is
null.");
            case TypeCode.Int16:
                return this.ToInt16(provider);
            case TypeCode.Int32:
                return this.ToInt32(provider);
            case TypeCode.Int64:
                return this.ToInt64(provider);
            case TypeCode.Object:
                // Leave conversion of non-base types to derived classes.
                throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                                conversionType.Name));
            case TypeCode.SByte:
                return this.ToSByte(provider);
            case TypeCode.Single:
                return this.ToSingle(provider);
            case TypeCode.String:
                IConvertible iconv = this;
                return iconv.ToString(provider);
            case TypeCode.UInt16:
                return this.ToUInt16(provider);
            case TypeCode.UInt32:
                return this.ToUInt32(provider);
            case TypeCode.UInt64:
                return this.ToUInt64(provider);
            default:
```

```
                throw new InvalidCastException("Conversion not supported.");
        }
    }

    public ushort ToUInt16(IFormatProvider provider)
    {
        if (temp < UInt16.MinValue || temp > UInt16.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt16 data type.", temp));
        else
            return (ushort)Math.Round(temp);
    }

    public uint ToUInt32(IFormatProvider provider)
    {
        if (temp < UInt32.MinValue || temp > UInt32.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt32 data type.", temp));
        else
            return (uint)Math.Round(temp);
    }

    public ulong ToUInt64(IFormatProvider provider)
    {
        if (temp < UInt64.MinValue || temp > UInt64.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt64 data type.", temp));
        else
            return (ulong)Math.Round(temp);
    }
}

public class TemperatureCelsius : Temperature, IConvertible
{
    public TemperatureCelsius(decimal value) : base(value)
    {
    }

    // Override ToString methods.
    public override string ToString()
    {
        return this.ToString(null);
    }

    public override string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°C";
    }

    // If conversionType is a implemented by another IConvertible method,
call it.
    public override object ToType(Type conversionType, IFormatProvider
provider)
    {
        // For non-objects, call base method.
```

```
        if (Type.GetTypeCode(conversionType) != TypeCode.Object)
        {
            return base.ToType(conversionType, provider);
        }
        else
        {
            if (conversionType.Equals(typeof(TemperatureCelsius)))
                return this;
            else if (conversionType.Equals(typeof(TemperatureFahrenheit)))
                return new TemperatureFahrenheit((decimal)this.temp * 9 / 5
+ 32);
            else
                throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                                conversionType.Name));
        }
    }
}

public class TemperatureFahrenheit : Temperature, IConvertible
{
    public TemperatureFahrenheit(decimal value) : base(value)
    {
    }

    // Override ToString methods.
    public override string ToString()
    {
        return this.ToString(null);
    }

    public override string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°F";
    }

    public override object ToType(Type conversionType, IFormatProvider
provider)
    {
        // For non-objects, call base method.
        if (Type.GetTypeCode(conversionType) != TypeCode.Object)
        {
            return base.ToType(conversionType, provider);
        }
        else
        {
            // Handle conversion between derived classes.
            if (conversionType.Equals(typeof(TemperatureFahrenheit)))
                return this;
            else if (conversionType.Equals(typeof(TemperatureCelsius)))
                return new TemperatureCelsius((decimal)(this.temp - 32) * 5
/ 9);
            // Unspecified object type: throw an InvalidCastException.
            else
                throw new InvalidCastException(String.Format("Cannot convert
```

```
from Temperature to {0}.",
                                                  conversionType.Name));
        }
    }
}
```

다음 예제에서는 `TemperatureCelsius` 개체를 `TemperatureFahrenheit` 개체로 변환하는 이러한 IConvertible 구현에 대한 몇 가지 호출을 보여 줍니다.

C#

```
TemperatureCelsius tempC1 = new TemperatureCelsius(0);
TemperatureFahrenheit tempF1 =
(TemperatureFahrenheit)Convert.ChangeType(tempC1,
typeof(TemperatureFahrenheit), null);
Console.WriteLine($"{tempC1} equals {tempF1}.");
TemperatureCelsius tempC2 = (TemperatureCelsius)Convert.ChangeType(tempC1,
typeof(TemperatureCelsius), null);
Console.WriteLine($"{tempC1} equals {tempC2}.");
TemperatureFahrenheit tempF2 = new TemperatureFahrenheit(212);
TemperatureCelsius tempC3 = (TemperatureCelsius)Convert.ChangeType(tempF2,
typeof(TemperatureCelsius), null);
Console.WriteLine($"{tempF2} equals {tempC3}.");
TemperatureFahrenheit tempF3 =
(TemperatureFahrenheit)Convert.ChangeType(tempF2,
typeof(TemperatureFahrenheit), null);
Console.WriteLine($"{tempF2} equals {tempF3}.");
// The example displays the following output:
//       0°C equals 32°F.
//       0°C equals 0°C.
//       212°F equals 100°C.
//       212°F equals 212°F.
```

TypeConverter 클래스

또한 .NET을 사용하면 System.ComponentModel.TypeConverter 클래스를 확장하고 System.ComponentModel.TypeConverterAttribute 특성을 통해 형식 변환기를 형식과 연결하여 사용자 지정 형식에 대한 형식 변환기를 정의할 수 있습니다. 다음 표에서는 이 접근 방식과 사용자 지정 형식에 대한 IConvertible 인터페이스 구현 간의 차이점을 강조 표시합니다.

> ⓘ **참고**
>
> 사용자 지정 형식에 대해 정의된 형식 변환기가 있는 경우에만 디자인 타임 지원을 제공할 수 있습니다.

테이블 확장

TypeConverter를 사용한 변환	IConvertible을 사용한 변환
사용자 지정 형식에 대해 TypeConverter에서 별도의 클래스를 파생시켜 구현합니다. 이 파생 클래스는 TypeConverterAttribute 특성을 적용하여 사용자 지정 형식과 연결됩니다.	변환을 수행하기 위해 사용자 지정 형식에 의해 구현됩니다. 형식의 사용자는 형식에 대한 IConvertible 변환 메서드를 호출합니다.
디자인 타임과 런타임에 모두 사용할 수 있습니다.	런타임에만 사용할 수 있습니다.
리플렉션을 사용하기 때문에 IConvertible로 활성화된 변환보다 느립니다.	리플렉션을 사용하지 않습니다.
사용자 지정 형식에서 다른 데이터 형식으로, 다른 데이터 형식에서 사용자 지정 형식으로 양방향 형식 변환을 허용합니다. 예를 들어, `MyType`에 대해 정의된 TypeConverter은 `MyType`에서 String으로의 변환과 String에서 `MyType`으로의 변환을 허용합니다.	사용자 지정 형식에서 다른 데이터 형식으로 변환할 수 있지만 다른 데이터 형식에서 사용자 지정 형식으로 변환할 수는 없습니다.

형식 변환기를 사용하여 변환을 수행하는 방법에 대한 자세한 내용은 System.ComponentModel.TypeConverter참조하세요.

참고하십시오

- System.Convert
- IConvertible
- 형식 변환 테이블

.NET의 형식 변환 표

아티클 • 2024. 03. 18.

확대 변환은 한 형식의 값을 크기가 같거나 더 큰 다른 형식으로 변환할 때 발생합니다. 축소 변환은 한 형식의 값을 크기가 더 작은 다른 형식의 값으로 변환할 때 발생합니다. 이 항목의 표에서는 두 가지 유형의 변환에서 모두 나타나는 동작을 설명합니다.

확대 변환

다음 표에서는 정보 손실 없이 수행할 수 있는 확대 변환에 대해 설명합니다.

테이블 확장

Type	데이터 손실 없이 다음 형식으로 변환할 수 있음
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
SByte	Int16, Int32, Int64, Single, Double, Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64, UInt64, Double, Decimal
Int64	Decimal
UInt64	Decimal
Single	Double

Single 또는 Double로의 일부 확대 변환에서는 정밀도 손실이 발생할 수 있습니다. 다음 표에서는 때때로 정보 손실이 발생할 수 있는 확대 변환에 대해 설명합니다.

테이블 확장

Type	다음 형식으로 변환할 수 있음
Int32	Single
UInt32	Single

Type	다음 형식으로 변환할 수 있음
Int64	Single, Double
UInt64	Single, Double
Decimal	Single, Double

축소 변환

Single 또는 Double로의 축소 변환에서는 정보 손실이 발생할 수 있습니다. 대상 형식이 소스 크기를 제대로 표시할 수 없는 경우 결과 형식이 상수 `PositiveInfinity` 또는 `NegativeInfinity`로 설정됩니다. `PositiveInfinity`는 양수를 0으로 나눈 결과이며 Single 또는 Double 값이 `MaxValue` 필드 값을 초과하는 경우에도 반환됩니다. `NegativeInfinity`는 음수를 0으로 나눈 결과이며 Single 또는 Double 값이 `MinValue` 필드 값보다 작은 경우에도 반환됩니다. Double에서 Single로 변환하면 `PositiveInfinity` 또는 `NegativeInfinity`가 발생할 수 있습니다.

축소 변환 시 다른 데이터 형식에 대한 정보 손실도 발생할 수 있습니다. 그러나 변환 중인 형식의 값이 대상 형식의 `MaxValue` 및 `MinValue` 필드에 지정된 범위를 벗어나면 OverflowException이 throw되며, 런타임에서 변환을 검사하여 대상 형식의 값이 해당 `MaxValue` 또는 `MinValue`를 초과하지 않는지 확인합니다. System.Convert 클래스로 수행하는 변환은 항상 이런 방식으로 검사됩니다.

다음 표에서는 System.Convert 사용 시 OverflowException을 throw하는 변환 또는 변환 중인 형식의 값이 정의된 결과 형식 범위를 벗어나는지 확인한 모든 변환을 보여줍니다.

테이블 확장

Type	다음 형식으로 변환할 수 있음
Byte	SByte
SByte	Byte, UInt16, UInt32, UInt64
Int16	Byte, SByte, UInt16
UInt16	Byte, SByte, Int16
Int32	Byte, SByte, Int16, UInt16,UInt32
UInt32	Byte, SByte, Int16, UInt16, Int32
Int64	Byte, SByte, Int16, UInt16, Int32,UInt32,UInt64

Type	다음 형식으로 변환할 수 있음
UInt64	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64
Decimal	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Single	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Double	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

참고 항목

- System.Convert
- .NET에서 형식 변환

System.Convert 클래스

아티클 • 2025. 04. 04.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

정적 Convert 클래스에는 .NET의 기본 데이터 형식으로의 변환을 지원하는 데 주로 사용되는 메서드가 포함되어 있습니다. 지원되는 기본 형식은 Boolean,Char, ,SByte, ByteInt16, Int32, Int64UInt16, UInt32UInt64SingleDoubleDecimalDateTime , 및 .String 또한 클래스에는 Convert 다른 종류의 변환을 지원하는 메서드가 포함되어 있습니다.

기본 형식으로의 변환 및 기본 형식에서의 변환

모든 기본 형식을 다른 모든 기본 형식으로 변환하는 변환 메서드가 있습니다. 그러나 특정 변환 메서드에 대한 실제 호출은 런타임에 기본 형식의 값과 대상 기본 형식에 따라 5가지 결과 중 하나를 생성할 수 있습니다. 이 다섯 가지 결과는 다음과 같습니다.

- 변환이 없습니다. 형식에서 그 자체로 변환을 시도할 때 발생합니다(예를 들어, Convert.ToInt32(Int32)을(를) 인수로 하여 Int32을(를) 호출하는 경우). 이 경우 메서드는 단순히 원래 형식의 인스턴스를 반환합니다.

- InvalidCastException입니다. 특정 변환이 지원되지 않는 경우에 발생합니다. InvalidCastException 다음 변환에 대해 throw됩니다.
  - 변환에서 Char 또는 Boolean, Single, Double, Decimal, DateTime.
  - Boolean, Single, Double, Decimal, 또는 DateTime에서 Char로 변환합니다.
  - DateTime를 String를 제외한 다른 형식으로 변환합니다.
  - String을(를) 제외한 다른 형식에서 DateTime로의 변환.

- FormatException. 문자열이 적절한 형식이 아니므로 문자열 값을 다른 기본 형식으로 변환하는 시도가 실패할 때 발생합니다. 다음 변환에서는 예외가 발생됩니다.
  - Boolean 값으로 변환될 문자열이 Boolean.TrueString이나 Boolean.FalseString와 같지 않습니다.
  - 값으로 변환할 문자열은 Char 여러 문자로 구성됩니다.
  - 숫자 형식으로 변환할 문자열은 유효한 숫자로 인식되지 않습니다.
  - 변환 DateTime 할 문자열은 유효한 날짜 및 시간 값으로 인식되지 않습니다.

- 성공적인 변환입니다. 이전 결과에 나열되지 않은 두 개의 서로 다른 기본 형식 간의 변환의 경우 모든 확대 변환과 데이터 손실을 초래하지 않는 모든 축소 변환이 성공하고 메서드는 대상 기본 형식의 값을 반환합니다.

- OverflowException입니다. 이는 축소 변환으로 인해 데이터가 손실되는 경우에 발생합니다. 예를 들어 값이 10000인 Int32 인스턴스를 Byte 형식으로 변환하려고 하면, 10000이 Byte 데이터 형식의 범위를 벗어났기 때문에 OverflowException가 발생합니다.

숫자 형식을 변환할 때 유효 자릿수의 일부가 손실되더라도 예외가 발생하지 않습니다. 그러나 결과가 특정 변환 메서드의 반환 값 형식으로 나타낼 수 있는 것보다 큰 경우 예외가 throw됩니다.

예를 들어 Double가 Single로 변환될 때 정밀도 손실이 발생할 수 있지만 예외가 발생하지 않습니다. 그러나 Double의 크기(정도)가 너무 커서 Single로 표현할 수 없는 경우, 오버플로 예외가 발생합니다.

10진수가 아닌 숫자

이 클래스에는 Convert 정수 값을 소수점이 아닌 문자열 표현으로 변환하고 소수가 아닌 숫자를 나타내는 문자열을 정수 값으로 변환하기 위해 호출할 수 있는 정적 메서드가 포함되어 있습니다. 이러한 각 변환 메서드에는 숫자 시스템, 이진(base 2), 8진수(base 8), 16진수(base 16) 및 10진수(base 10)를 지정할 수 있는 인수가 포함 `base` 됩니다. 각 CLS 규격 기본 정수 형식을 문자열로 변환하는 메서드 집합과 문자열을 각 기본 정수 계열 형식으로 변환하는 메서드 집합이 있습니다.

- ToString(Byte, Int32) 및 ToByte(String, Int32)를 사용하여 지정된 기준에서 문자열과 바이트 값 간에 변환합니다.
- ToString(Int16, Int32) 및 ToInt16(String, Int32)를 사용하여 지정된 기준의 문자열과 16비트 부호 있는 정수 간에 변환합니다.
- ToString(Int32, Int32) 및 ToInt32(String, Int32)를 사용하여 지정된 기준의 문자열과 32비트 부호 있는 정수 간에 변환합니다.
- ToString(Int64, Int32) 및 ToInt64(String, Int32)를 사용하여 지정된 밑의 문자열로 또는 문자열에서 64비트 부호 있는 정수로 변환합니다.
- ToSByte(String, Int32)지정된 형식의 바이트 값의 문자열 표현을 서명된 바이트로 변환합니다.
- ToUInt16(String, Int32)지정된 형식의 정수 문자열 표현을 부호 없는 16비트 정수로 변환합니다.
- ToUInt32(String, Int32)지정된 형식의 정수 문자열 표현을 부호 없는 32비트 정수로 변환합니다.
- ToUInt64(String, Int32)지정된 형식의 정수 문자열 표현을 부호 없는 64비트 정수로 변환합니다.

다음 예제에서는 지원되는 모든 숫자 형식의 Int16.MaxValue 문자열로 값을 변환합니다. 그런 다음 문자열 값을 Int16 값으로 다시 변환합니다.

```
```

C#

```
using System;

public class Example
{
   public static void Main()
   {
      int[] baseValues = { 2, 8, 10, 16 };
      short value = Int16.MaxValue;
      foreach (var baseValue in baseValues) {
         String s = Convert.ToString(value, baseValue);
         short value2 = Convert.ToInt16(s, baseValue);

         Console.WriteLine($"{value} --> {s} (base {baseValue}) --> {value2}");
      }
   }
}
// The example displays the following output:
//     32767 --> 111111111111111 (base 2) --> 32767
//     32767 --> 77777 (base 8) --> 32767
//     32767 --> 32767 (base 10) --> 32767
//     32767 --> 7fff (base 16) --> 32767
```

사용자 지정 개체에서 기본 형식으로 변환

이 메서드는 기본 형식 Convert 간의 변환을 지원하는 것 외에도 모든 사용자 지정 형식을 모든 기본 형식으로 변환할 수 있도록 지원합니다. 이렇게 하려면 사용자 지정 형식은 구현 형식을 각 기본 형식으로 변환하는 메서드를 정의하는 인터페이스를 구현 IConvertible 해야 합니다. 특정 형식에서 지원되지 않는 변환은 InvalidCastException.을 throw해야 합니다.

메서드가 첫 번째 매개 변수로 사용자 지정 형식을 전달받거나, `Convert.To`*Type* 메서드(예: Convert.ToInt32(Object) 또는 Convert.ToDouble(Object, IFormatProvider))가 호출되고 첫 번째 매개 변수로 사용자 지정 형식의 인스턴스를 전달받으면, Convert 메서드는 변환을 수행하기 위해 사용자 지정 형식의 IConvertible 구현을 호출합니다. 자세한 내용은 .NET의 형식 변환을 참조하세요.

문화권별 서식 지정 정보

모든 기본 형식 변환 메서드와 ChangeType 메서드에는 형식의 매개 변수가 있는 오버로드가 포함됩니다 IFormatProvider. 예를 들어 메서드에는 Convert.ToBoolean 다음 두 오버로드가 있습니다.

- Convert.ToBoolean(Object, IFormatProvider)
- Convert.ToBoolean(String, IFormatProvider)

매개 변수는 IFormatProvider 변환 프로세스를 지원하기 위해 문화권별 서식 정보를 제공할 수 있습니다. 그러나 대부분의 기본 형식 변환 메서드에서는 무시됩니다. 다음 기본 형식 변환 메서드에서만 사용됩니다. 인수가 `null`IFormatProvider 이러한 메서드 CultureInfo 에 전달되면 현재 문화권을 나타내는 개체가 사용됩니다.

- 값을 숫자 형식으로 변환하는 메서드입니다. 매개 변수는 IFormatProvider 형식 String 및 IFormatProvider.의 매개 변수가 있는 오버로드에서 사용됩니다. 또한 형식 Object 의 매개 변수가 있는 오버로드에서 사용되며 IFormatProvider 개체의 런타임 형식이 String.
- 값을 날짜 및 시간으로 변환하는 메서드입니다. 매개 변수는 IFormatProvider 형식 String 및 IFormatProvider.의 매개 변수가 있는 오버로드에서 사용됩니다. 형식 Object의 매개변수를 갖는 오버로드에서도 사용되며, 객체의 실행 시간 형식이 String인 경우 IFormatProvider에 사용됩니다.
- DateTime 숫자 값이나 IFormatProvider 값을 문자열로 변환하는 Convert.ToString 매개 변수를 포함한 오버로드에 의해.

그러나 IConvertible를 구현하는 모든 사용자 정의 형식은 IFormatProvider 매개변수를 사용할 수 있습니다.

Base64 인코딩

Base64 인코딩은 이진 데이터를 문자열로 변환합니다. base-64 숫자로 표현된 데이터는 7비트 문자만 전송할 수 있는 데이터 채널을 통해 쉽게 전달할 수 있습니다. 클래스에는 base64 인코딩을 지원하는 다음 메서드가 포함됩니다. 이 메서드 집합은 바이트 배열을 String로 변환하거나, 반대로 밑 64 문자로 구성된 유니코드 문자 배열 간에 바이트 배열을 변환하는 기능을 지원합니다.

- ToBase64String- 바이트 배열을 base64로 인코딩된 문자열로 변환합니다.
- ToBase64CharArray- 바이트 배열을 base64로 인코딩된 문자 배열로 변환합니다.
- FromBase64String- base64로 인코딩된 문자열을 바이트 배열로 변환합니다.
- FromBase64CharArray- base64로 인코딩된 문자 배열을 바이트 배열로 변환합니다.

기타 일반 변환

다른 .NET 클래스를 사용하여 클래스의 정적 메서드에서 지원되지 않는 일부 변환을 Convert 수행할 수 있습니다. 여기에는 다음이 포함됩니다.

- 바이트 배열로 변환

  이 클래스는 BitConverter 기본 숫자 형식(포함 Boolean)을 바이트 배열로 변환하고 바이트 배열에서 기본 데이터 형식으로 다시 변환하는 메서드를 제공합니다.

- 문자 인코딩 및 디코딩

  Encoding 클래스 및 파생 클래스(예: UnicodeEncoding 및UTF8Encoding)는 문자 배열 또는 문자열을 인코딩하는 메서드(즉, 특정 인코딩에서 바이트 배열로 변환)하고 인코딩된 바이트 배열(즉, 바이트 배열을 UTF16으로 인코딩된 유니코드 문자로 다시 변환)을 디코딩하는 메서드를 제공합니다. 자세한 내용은 .NET의 문자 인코딩을 참조하세요.

예제

다음 예제에서는 클래스의 변환 메서드 Convert 중 일부(예 ToInt32: , ToBoolean및 ToString)를 보여 줍니다.

C#

```
double dNumber = 23.15;

try {
    // Returns 23
    int    iNumber = System.Convert.ToInt32(dNumber);
}
catch (System.OverflowException) {
    System.Console.WriteLine(
                "Overflow in double to int conversion.");
}
// Returns True
bool   bNumber = System.Convert.ToBoolean(dNumber);

// Returns "23.15"
string strNumber = System.Convert.ToString(dNumber);

try {
    // Returns '2'
    char chrNumber = System.Convert.ToChar(strNumber[0]);
}
catch (System.ArgumentNullException) {
    System.Console.WriteLine("String is null");
}
catch (System.FormatException) {
    System.Console.WriteLine("String length is greater than 1.");
}

// System.Console.ReadLine() returns a string and it
// must be converted.
int newInteger = 0;
try {
    System.Console.WriteLine("Enter an integer:");
    newInteger = System.Convert.ToInt32(
                        System.Console.ReadLine());
}
catch (System.ArgumentNullException) {
    System.Console.WriteLine("String is null.");
```

```
}
catch (System.FormatException) {
    System.Console.WriteLine("String does not consist of an " +
                    "optional sign followed by a series of digits.");
}
catch (System.OverflowException) {
    System.Console.WriteLine(
    "Overflow in string to int conversion.");
}

System.Console.WriteLine($"Your integer as a double is
{System.Convert.ToDouble(newInteger)}");
```

무명 형식과 튜플 형식 중에서 선택

2025. 06. 17.

적절한 형식을 선택하려면 다른 형식에 비해 유용성, 성능 및 절충을 고려해야 합니다. 익명 형식은 C# 3.0부터 사용할 수 있으며 제네릭 System.Tuple<T1,T2> 형식은 .NET Framework 4.0에서 도입되었습니다. 그 이후로 이름에서 알 수 있듯이 익명 형식의 유연성으로 값 형식을 제공하는 것과 같은 System.ValueTuple<T1,T2> 언어 수준 지원과 함께 새로운 옵션이 도입되었습니다. 이 문서에서는 한 형식을 다른 형식보다 선택하는 것이 적절한 경우를 알아봅니다.

유용성 및 기능

익명 형식은 LINQ(Language-Integrated Query) 식을 사용하여 C# 3.0에서 도입되었습니다. LINQ를 사용하면 개발자는 종종 쿼리 결과를 작업 중인 개체의 몇 가지 선택 속성을 포함하는 익명 형식으로 프로젝팅합니다. 다음 예제에서는 DateTime 개체 배열을 인스턴스화한 후, 두 속성을 가진 익명 형식으로 프로젝션하여 이 개체들을 반복하여 처리합니다.

C#

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var anonymous in
             dates.Select(
                 date => new { Formatted = $"{date:MMM dd, yyyy hh:mm zzz}",
date.Ticks }))
{
    Console.WriteLine($"Ticks: {anonymous.Ticks}, formatted:
{anonymous.Formatted}");
}
```

무명 형식은 연산자를 new 사용하여 인스턴스화되고 속성 이름과 형식은 선언에서 유추됩니다. 동일한 어셈블리에 있는 둘 이상의 무명 개체 이니셜라이저가 동일한 순서에 있고 이름과 형식이 같은 속성 시퀀스를 지정하는 경우 컴파일러는 개체를 동일한 형식의 인스턴스로 처리합니다. 이러한 개체는 컴파일러에서 생성된 동일한 형식 정보를 공유합니다.

이전 C# 코드 조각은 다음 컴파일러에서 생성된 C# 클래스와 마찬가지로 두 개의 속성을 사용하여 익명 형식을 프로젝션합니다.

C#

```
internal sealed class f__AnonymousType0
{
    public string Formatted { get; }
    public long Ticks { get; }

    public f__AnonymousType0(string formatted, long ticks)
    {
        Formatted = formatted;
        Ticks = ticks;
    }
}
```

자세한 내용은 익명 형식을 참조하세요. LINQ 쿼리로 프로젝팅할 때 튜플과 동일한 기능이 존재하므로 속성을 튜플로 선택할 수 있습니다. 이러한 튜플은 익명 형식과 마찬가지로 쿼리를 통해 흐릅니다. 이제 `System.Tuple<string, long>`를 사용하는 다음 예제를 고려해보십시오.

C#

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var tuple in
            dates.Select(
                date => new Tuple<string, long>($"{date:MMM dd, yyyy hh:mm zzz}",
date.Ticks)))
{
    Console.WriteLine($"Ticks: {tuple.Item2}, formatted: {tuple.Item1}");
}
```

이 System.Tuple<T1,T2>경우 인스턴스는 번호가 매겨진 항목 속성(예: `Item1` 및 `Item2`.)을 노출합니다. 이러한 속성 이름은 서수만 제공하여 속성 값의 의도를 이해하기 어렵게 만들 수 있습니다. 또한 `System.Tuple` 형식은 참조 `class` 형식입니다. System.ValueTuple<T1,T2> 그러나 값 `struct` 형식입니다. 다음 C# 코드 조각은 `ValueTuple<string, long>` 에 투영하는 데 사용됩니다. 이렇게 하면 리터럴 구문을 사용하여 할당합니다.

C#

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};
```

```
foreach (var (formatted, ticks) in
            dates.Select(
                date => (Formatted: $"{date:MMM dd, yyyy at hh:mm zzz}",
date.Ticks)))
{
    Console.WriteLine($"Ticks: {ticks}, formatted: {formatted}");
}
```

튜플에 대한 자세한 내용은 튜플 형식(C# 참조) 또는 튜플(Visual Basic)을 참조하세요.

이전 예제는 모두 기능적으로 동일합니다. 그러나 유용성과 기본 구현에는 약간의 차이가 있습니다.

장단점

항상 ValueTuple를 Tuple 대신 사용하는 것과 익명 형식을 사용하는 것이 좋을 수 있지만, 고려해야 할 장단점이 있습니다. 형식은 ValueTuple 변경 가능한 반면 Tuple 읽기 전용입니다. 익명 형식은 식 트리에서 사용할 수 있지만 튜플은 사용할 수 없습니다. 다음 표는 몇 가지 주요 차이점에 대한 개요입니다.

주요 차이점

테이블 확장

이름	액세스 한정자	유형	사용자 지정 멤버 이름	분해 지원	식 트리 지원
무명 형식	`internal`	`class`	✔	✘	✔
Tuple	`public`	`class`	✘	✘	✔
ValueTuple	`public`	`struct`	✔	✔	✘

직렬화

형식을 선택할 때 중요한 고려 사항 중 하나는 직렬화해야 하는지 여부입니다. serialization은 개체의 상태를 유지하거나 전송할 수 있는 폼으로 변환하는 프로세스입니다. 자세한 내용은 serialization을 참조하세요. serialization이 중요한 경우에는 익명 형식이나 튜플 형식을 사용하는 것보다 `class` 또는 `struct`를 만들어 사용하는 것이 좋습니다.

성능

이러한 형식 간의 성능은 시나리오에 따라 달라집니다. 주요 영향으로는 할당과 복사 간의 절충이 포함됩니다. 대부분의 시나리오에서는 영향이 작습니다. 중대한 영향이 발생할 수 있는 경우

결정을 알리기 위해 측정을 수행해야 합니다.

결론

개발자는 튜플과 무명 형식 중에서 선택할 때 고려해야 할 몇 가지 요소가 있습니다. 일반적으로 식 트리를 사용하지 않고 튜플 문법에 익숙하다면, 속성을 이름 지을 수 있는 유연성을 가진 값 형식을 제공하므로 ValueTuple를 선택하세요. 식 트리를 사용 중이고 속성 이름을 지정하려는 경우 익명 형식을 선택합니다. 그렇지 않으면 Tuple를 사용합니다.

참고하십시오

- 익명 형식
- 표현식 트리
- 튜플 형식(C# 참조)
- 튜플(Visual Basic)
- 형식 디자인 지침

핵심 .NET 라이브러리 개요

.NET API에는 개발 프로세스를 신속하게 처리 및 최적화하고 시스템 기능에 대한 액세스를 제공하는 클래스, 인터페이스, 대리자 및 값 형식이 포함됩니다. 언어 간의 상호 운용성을 용이하게 하기 위해 대부분의 .NET 형식은 CLS 규격이므로 컴파일러가 CLS(공용 언어 사양)를 준수하는 모든 프로그래밍 언어에서 사용할 수 있습니다.

.NET 형식은 .NET 애플리케이션, 구성 요소 및 컨트롤이 빌드되는 기반입니다. .NET에는 다음 함수를 수행하는 형식이 포함됩니다.

- 기본 데이터 형식 및 예외를 나타냅니다.
- 데이터 구조를 캡슐화합니다.
- I/O를 수행합니다.
- 로드된 형식에 대한 정보에 액세스합니다.
- .NET 보안 검사를 호출합니다.
- 데이터 액세스, 풍부한 클라이언트 쪽 GUI 및 서버 제어 클라이언트 쪽 GUI를 제공합니다.

.NET은 다양한 인터페이스 집합과 추상 및 구체적(비 추상) 클래스를 제공합니다. as-is 구체적인 클래스를 사용하거나 대부분의 경우 해당 클래스에서 고유한 클래스를 파생시킬 수 있습니다. 인터페이스의 기능을 사용하려면 인터페이스를 구현하는 클래스를 만들거나 인터페이스를 구현하는 .NET 클래스 중 하나에서 클래스를 파생시킬 수 있습니다.

명명 규칙

.NET 형식은 점 구문 명명 체계를 사용하여 계층 구조를 나타냅니다. 관련 형식은 보다 쉽게 검색 및 참조할 수 있도록 네임스페이스로 그룹화됩니다. 전체 이름의 첫 번째 부분은 네임스페이스 이름입니다. 이름의 마지막 부분은 형식 또는 멤버 이름입니다. 예를 들어, `System.Collections.Generic.List<T>`는 List<T> 네임스페이스에 속하는 `System.Collections.Generic` 형식을 나타냅니다. System.Collections.Generic 형식을 사용하여 제네릭 컬렉션을 사용할 수 있습니다.

이 명명 체계를 사용하면 .NET을 확장하는 라이브러리 개발자가 계층적 형식 그룹을 만들고 일관되고 유익한 방식으로 이름을 지정할 수 있습니다. 또한 형식 이름 충돌을 방지하는 전체 이름(즉, 네임스페이스 및 형식 이름)으로 형식을 명확하게 식별할 수 있습니다.

명명 패턴을 사용하여 관련 형식을 네임스페이스로 그룹화하면 클래스 라이브러리를 빌드하고 문서화하는 데 유용합니다. 그러나 이 명명 체계는 표시 유형, 멤버 액세스, 상속, 보안 또는 바인딩에 영향을 주지 않습니다. 네임스페이스는 여러 어셈블리에서 분할할 수 있으며 단일 어셈블리에는 여러 네임스페이스의 형식이 포함될 수 있습니다. 어셈블리는 공용 언어 런타임의 버전 관리, 배포, 보안, 로드 및 표시 유형에 대한 공식적인 구조를 제공합니다.

네임스페이스 및 형식 이름에 대한 자세한 내용은 공용 형식 시스템을 참조하세요.

시스템 네임스페이스

System 네임스페이스는 .NET의 기본 형식에 대한 루트 네임스페이스입니다. 이 네임스페이스에는 모든 애플리케이션에서 사용하는 기본 데이터 형식(예: Object 상속 계층의 루트), Byte, , CharArrayInt32및 String클래스를 나타내는 클래스가 포함됩니다.

이러한 형식의 대부분은 프로그래밍 언어에서 사용하는 기본 데이터 형식에 해당합니다. .NET 형식을 사용하여 코드를 작성할 때 .NET 기본 데이터 형식이 필요한 경우 해당 언어 키워드를 사용할 수 있습니다. 자세한 내용은 다음을 참조하세요.

- 기본 제공 형식(C# 참조)
- 기본 형식(Visual Basic)
- 기본 형식(F#)

기본 데이터 형식 System 외에도 네임스페이스는 예외를 처리하는 클래스부터 가비지 수집과 같은 핵심 런타임 개념을 다루는 클래스에 이르기까지 100개가 넘는 클래스를 포함합니다. 네임스페이스 System는 여러 하위 수준 네임스페이스도 포함합니다.

.NET API 참조 설명서는 각 네임스페이스, 해당 형식 및 해당 멤버에 대한 설명서를 제공합니다.

데이터 구조체

.NET에는 많은 .NET 앱의 작동 주체인 데이터 구조 집합이 포함되어 있습니다. 주로 컬렉션이지만 다른 형식도 포함됩니다.

- Array - 인덱스로 액세스할 수 있는 강력한 형식의 개체 배열을 나타냅니다. 구성에 따라 고정 크기가 있습니다.
- List<T> - 인덱스로 액세스할 수 있는 강력한 형식의 개체 목록을 나타냅니다. 필요에 따라 자동으로 크기가 조정됩니다.
- Dictionary<TKey,TValue> - 키로 인덱싱되는 값의 컬렉션을 나타냅니다. 키를 통해 값에 액세스할 수 있습니다. 필요에 따라 자동으로 크기가 조정됩니다.
- Uri - URI(Uniform Resource Identifier)의 개체 표현을 제공하고 URI 부분에 쉽게 액세스할 수 있도록 합니다.
- DateTime - 일반적으로 날짜 및 시간으로 표현되는 시간(인스턴트)을 나타냅니다.

유틸리티 API

.NET에는 많은 중요한 작업에 대한 기능을 제공하는 유틸리티 API 집합이 포함되어 있습니다.

- System.Net.Http.HttpClient - URI로 식별된 리소스에서 HTTP 요청을 보내고 HTTP 응답을 수신하기 위한 API입니다.

- System.Diagnostics.Debug 및 System.Diagnostics.Trace: 애플리케이션 실행 중에 로깅 및 디버깅 정보를 작성하기 위한 API입니다.
- System.IO.StreamReader 및 System.IO.StreamWriter - 파일 읽기 및 쓰기를 위한 API입니다.
- System.Text.Json.JsonSerializer - 개체 또는 값 형식을 JSON으로 직렬화하고 JSON을 개체 또는 값 형식으로 역직렬화하는 API입니다.

앱 모델 API

.NET과 함께 사용할 수 있는 여러 앱 모델이 있습니다. 예를 들면 다음과 같습니다.

- ASP.NET Core - 웹 사이트 및 서비스를 빌드하기 위한 웹 프레임워크입니다. Windows, Linux 및 macOS에서 지원됩니다.
- .NET MAUI - C#을 사용하여 Windows, macOS, iOS 및 Android에서 실행되는 네이티브 앱을 빌드하기 위한 앱 플랫폼입니다.
- Windows Desktop - WPF(Windows Presentation Foundation) 및 Windows Forms를 포함합니다.

참고하십시오

- 런타임 라이브러리 개요
- 일반 형식 시스템
- .NET API 브라우저

Last updated on 2025. 11. 07.

System.Object 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스는 Object 모든 .NET 클래스의 궁극적인 기본 클래스이며 형식 계층의 루트입니다.

.NET의 모든 클래스는 파생 Object되므로 클래스에 Object 정의된 모든 메서드는 시스템의 모든 개체에서 사용할 수 있습니다. 파생 클래스는 다음을 포함한 이러한 메서드 중 일부를 실제로 재정의할 수 있습니다.

- Equals: 개체 간의 비교를 지원합니다.
- Finalize: 개체가 자동으로 회수되기 전에 정리 작업을 수행합니다.
- GetHashCode: 해시 테이블 사용을 지원하기 위해 개체 값에 해당하는 숫자를 생성합니다.
- ToString: 클래스의 인스턴스를 설명하는 사람이 읽을 수 있는 텍스트 문자열을 만듭니다.

일반적으로 언어는 상속이 암시적이기 때문에 상속 Object 을 선언하는 클래스가 필요하지 않습니다.

성능 고려 사항

모든 형식의 개체를 처리해야 하는 컬렉션과 같은 클래스를 디자인하는 경우 클래스의 인스턴스를 허용하는 클래스 멤버를 Object 만들 수 있습니다. 그러나 데이터 형식을 박싱 및 언박싱하는 프로세스에는 성능상의 비용이 발생합니다. 새 클래스가 특정 값 형식을 자주 처리하는 것을 알고 있는 경우 두 가지 전술 중 하나를 사용하여 boxing 비용을 최소화할 수 있습니다.

- 형식을 Object 허용하는 일반 메서드와 클래스가 자주 처리할 것으로 예상되는 각 값 형식을 허용하는 형식별 메서드 오버로드 집합을 만듭니다. 호출 매개 변수 형식을 허용하는 형식별 메서드가 있는 경우 boxing이 발생하지 않고 형식별 메서드가 호출됩니다. 호출 매개 변수 형식과 일치하는 메서드 인수가 없으면 매개 변수가 boxed되고 일반 메서드가 호출됩니다.
- 제네릭을 사용하도록 형식 및 해당 멤버를 디자인합니다. 공용 언어 런타임은 클래스의 인스턴스를 만들고 제네릭 형식 인수를 지정할 때 닫힌 제네릭 형식을 만듭니다. 제네릭 메서드는 형식 특화되며 호출 매개변수를 박싱하지 않고 호출할 수 있습니다.

형식을 허용하고 반환 Object 하는 범용 클래스를 개발해야 하는 경우도 있지만 자주 사용되는 형식을 처리하는 형식별 클래스를 제공하여 성능을 향상시킬 수 있습니다. 예를 들어 부울 값을 설정 및 가져오기에 특정한 클래스를 제공하면 부울 값의 박싱 및 언박싱 비용이 제거됩니다.

System.Object.Equals 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 문서는 Object.Equals(Object) 메서드와 관련이 있습니다.

현재 인스턴스와 `obj` 매개 변수 간의 비교 유형은 현재 인스턴스가 참조 형식인지 값 형식인지에 따라 달라집니다.

- 현재 인스턴스가 참조 형식인 경우 메서드는 Equals(Object) 참조 같음을 테스트하고 메서드에 Equals(Object) 대한 호출은 메서드 호출 ReferenceEquals 과 동일합니다. 참조 같음은 비교된 개체 변수가 동일한 개체를 참조한다는 것을 의미합니다. 다음 예제에서는 이러한 비교의 결과를 보여 줍니다. 클래스 참조 형식인 `Person`을 정의하고 클래스 생성자 `Person`를 호출하여 같은 값을 가진 두 개의 새 `Person` 개체인 `person1a`와 `person2`를 인스턴스화합니다. `person1a`을(를) 다른 개체 변수`person1b`에 할당합니다. 예제의 출력에서 보이는 것처럼 `person1a`와 `person1b`는 동일한 개체를 참조하므로 동일합니다. 그러나 `person1a` `person2` 값이 동일하지만 같지는 않습니다.

C#

```
using System;

// Define a reference type that does not override Equals.
public class Person
{
   private string personName;

   public Person(string name)
   {
      this.personName = name;
   }

   public override string ToString()
   {
      return this.personName;
   }
}

public class Example1
{
   public static void Main()
   {
      Person person1a = new Person("John");
      Person person1b = person1a;
      Person person2 = new Person(person1a.ToString());

      Console.WriteLine("Calling Equals:");
```

```
      Console.WriteLine($"person1a and person1b:
{person1a.Equals(person1b)}");
      Console.WriteLine($"person1a and person2: {person1a.Equals(person2)}");

      Console.WriteLine("\nCasting to an Object and calling Equals:");
      Console.WriteLine($"person1a and person1b: {((object)
person1a).Equals((object) person1b)}");
      Console.WriteLine($"person1a and person2: {((object)
person1a).Equals((object) person2)}");
   }
}
// The example displays the following output:
//       person1a and person1b: True
//       person1a and person2: False
//
//       Casting to an Object and calling Equals:
//       person1a and person1b: True
//       person1a and person2: False
```

- 현재 인스턴스가 값 형식인 경우 메서드는 Equals(Object) 값 같음을 테스트합니다. 값 같음은 다음을 의미합니다.
  - 두 개체의 형식이 같습니다. 다음 예제가 보여주듯이, 값이 12인 Byte 개체는 런타임 형식이 다르기 때문에 값이 12인 Int32 개체와 같지 않습니다.

    C#

    ```
    byte value1 = 12;
    int value2 = 12;

    object object1 = value1;
    object object2 = value2;

    Console.WriteLine($"{object1} ({object1.GetType().Name}) = {object2}
    ({object2.GetType().Name}): {object1.Equals(object2)}");

    // The example displays the following output:
    //        12 (Byte) = 12 (Int32): False
    ```

  - 두 개체의 public 및 private 필드 값은 같습니다. 다음 예제에서는 값 같음을 테스트합니다. 값 형식인 `Person` 구조를 정의하고 `Person` 클래스 생성자를 호출하여 값이 같은 두 개의 새 `Person` 객체 `person1` 및 `person2`를 인스턴스화합니다. 예제의 출력에서와 같이 두 개체 변수는 서로 다른 개체 `person1` 를 참조하지만 `person2` 개인 `personName` 필드에 대한 값이 같기 때문에 같습니다.

    C#

    ```
    using System;
    ```

```
// Define a value type that does not override Equals.
public struct Person3
{
   private string personName;

   public Person3(string name)
   {
      this.personName = name;
   }

   public override string ToString()
   {
      return this.personName;
   }
}

public struct Example3
{
   public static void Main()
   {
      Person3 person1 = new Person3("John");
      Person3 person2 = new Person3("John");

      Console.WriteLine("Calling Equals:");
      Console.WriteLine(person1.Equals(person2));

      Console.WriteLine("\nCasting to an Object and calling Equals:");
      Console.WriteLine(((object) person1).Equals((object) person2));
   }
}
// The example displays the following output:
//       Calling Equals:
//       True
//
//       Casting to an Object and calling Equals:
//       True
```

.NET에서 Object 클래스가 모든 형식의 기본 클래스이므로, Object.Equals(Object) 메서드는 다른 모든 형식에 대한 기본적인 동등성 비교를 제공합니다. 그러나 형식은 값 같음을 달성하기 위해 Equals을 재정의하는 경우가 많습니다. 자세한 내용은 호출자에 대한 참고 사항 및 상속자에 대한 참고 섹션을 참조하세요.

Windows 런타임에 대한 참고 사항

Windows 런타임의 클래스에서 Equals(Object) 메서드 오버로드를 호출하면, Equals(Object)를 재정의하지 않는 클래스에 대한 기본 동작을 제공합니다. 이는 .NET이 Windows 런타임에 대해 제공하는 지원의 일부입니다(Windows 스토어 앱 및 Windows 런타임에 대한 .NET 지원 참조). Windows 런타임의 클래스는 Object를 상속되지 않으며 현재 Equals(Object) 메서드를 구현하지 않습니다. 그러나 C# 또는 Visual Basic 코드에서 사용할 때 ToString, Equals(Object),

GetHashCode 방법이 있는 것처럼 보이며, .NET은 이러한 메서드에 대한 기본 동작을 제공합니다.

> ⓘ **참고**
>
> C# 또는 Visual Basic으로 작성된 Windows 런타임 클래스는 Equals(Object) 메서드 오버로드를 재정의할 수 있습니다.

발신자에 대한 참고 사항

파생 클래스는 값의 동등성을 구현하기 위해 Object.Equals(Object) 메서드를 재정의하는 경우가 많습니다. 형식은 종종 `Equals` 인터페이스를 구현함으로써 IEquatable<T> 메서드에 대해 강력한 형식의 추가 오버로드를 제공합니다. 메서드 `Equals`를 호출하여 같음을 테스트할 때, 현재 인스턴스가 Object.Equals를 재정의하는지 여부와 특정 `Equals` 메서드 호출이 해결되는 방법을 이해해야 합니다. 그렇지 않으면 의도한 것과 다른 같음 테스트를 수행하고 메서드가 예기치 않은 값을 반환할 수 있습니다.

다음 예제에서 이에 대해 설명합니다. 동일한 문자열을 사용하여 세 StringBuilder 개의 개체를 인스턴스화한 다음 메서드를 `Equals` 네 번 호출합니다. 첫 번째 메서드 호출은 `true`를 반환하고, 나머지 세 번은 `false`를 반환합니다.

C#

```
using System;
using System.Text;

public class Example5
{
   public static void Main()
   {
      StringBuilder sb1 = new StringBuilder("building a string...");
      StringBuilder sb2 = new StringBuilder("building a string...");

      Console.WriteLine($"sb1.Equals(sb2): {sb1.Equals(sb2)}");
      Console.WriteLine($"((Object) sb1).Equals(sb2): {((Object)
sb1).Equals(sb2)}");
      Console.WriteLine($"Object.Equals(sb1, sb2): {Object.Equals(sb1, sb2)}");

      Object sb3 = new StringBuilder("building a string...");
      Console.WriteLine($"\nsb3.Equals(sb2): {sb3.Equals(sb2)}");
   }
}
// The example displays the following output:
//       sb1.Equals(sb2): True
//       ((Object) sb1).Equals(sb2): False
//       Object.Equals(sb1, sb2): False
```

```
//
//       sb3.Equals(sb2): False
```

첫 번째 경우 값 같음을 테스트하는 강력한 형식 StringBuilder.Equals(StringBuilder) 의 메서드 오버로드가 호출됩니다. 두 StringBuilder 개체에 할당된 문자열이 같으므로 메서드는 반환합니다 `true`. 그러나 StringBuilder은 Object.Equals(Object)을 재정의하지 않습니다. 이 때문에 StringBuilder 개체가 Object로 캐스팅될 때, StringBuilder 인스턴스가 Object 형식의 변수에 할당될 때, 그리고 Object.Equals(Object, Object) 메서드에 두 개의 StringBuilder 개체가 전달될 때, 기본 Object.Equals(Object) 메서드가 호출됩니다. StringBuilder 참조 형식이므로 두 StringBuilder 개체 ReferenceEquals 를 메서드에 전달하는 것과 같습니다. 세 StringBuilder 개체 모두 동일한 문자열을 포함하지만 세 개의 고유 개체를 참조합니다. 따라서 이러한 세 가지 메서드 호출은 반환됩니다 `false`.

메서드를 호출 ReferenceEquals 하여 참조 같음을 위해 현재 개체를 다른 개체와 비교할 수 있습니다. Visual Basic에서는 `is` 키워드를 사용할 수도 있습니다(예: `If Me Is otherObject Then ...`).

상속자에 대한 참고 사항

고유한 형식을 정의할 때 해당 형식은 기본 형식의 메서드에 `Equals` 정의된 기능을 상속합니다. 다음 표에서는 .NET의 `Equals` 주요 형식 범주에 대한 메서드의 기본 구현을 나열합니다.

테이블 확장

유형 범주	평등의 정의 방식	코멘트
에서 직접 파생된 클래스 Object	Object.Equals(Object)	참조 같음; 호출 Object.ReferenceEquals에 해당합니다.
구조	ValueType.Equals	값 같음; 직접 바이트 바이트 비교 또는 리플렉션을 사용한 필드별 비교 중 하나.
열거	Enum.Equals	값은 동일한 열거형 형식과 동일한 기본 값을 가져야 합니다.
대리인	MulticastDelegate.Equals	대리자는 유형이 동일하고 호출 목록이 일치해야 합니다.
인터페이스	Object.Equals(Object)	참조 동등성

값 형식의 경우 항상 Equals를 재정의해야 합니다. 왜냐하면 리플렉션에 의존하는 동등성 테스트는 성능이 떨어지기 때문입니다. 기본 구현 Equals을 재정의하여 참조 형식에 대해 참조 동일성이 아닌 값 동일성을 테스트하고, 값 동일성의 정확한 의미를 참조 형식에서 정의할 수도 있습니다. 두 개체가 동일한 인스턴스가 아니더라도 값이 같으면 Equals 구현은 `true`을 반환합니다.

형식의 구현자는 개체의 값을 구성하는 항목을 결정하지만 일반적으로 개체의 인스턴스 변수에 저장된 데이터의 일부 또는 전부입니다. 예를 들어, String 객체의 값은 문자열의 문자에 기반합니다. String.Equals(Object) 메서드는 Object.Equals(Object) 메서드를 재정의하여 동일한 순서로 동일한 문자를 포함하는 두 문자열 인스턴스에 대해 `true`을 반환합니다.

다음 예제는 값 같음을 테스트하기 위해 Object.Equals(Object) 메서드를 어떻게 재정의하는지를 보여줍니다. 클래스 Equals의 메서드 `Person`를 재정의합니다. 같음의 기본 클래스 구현을 수락한 경우 `Person` 두 `Person` 개체는 단일 개체를 참조하는 경우에만 동일합니다. 그러나 이 경우 두 `Person` 개체는 `Person.Id` 속성값이 같을 때 동일합니다.

C#

```
public class Person6
{
   private string idNumber;
   private string personName;

   public Person6(string name, string id)
   {
      this.personName = name;
      this.idNumber = id;
   }

   public override bool Equals(Object obj)
   {
      Person6 personObj = obj as Person6;
      if (personObj == null)
         return false;
      else
         return idNumber.Equals(personObj.idNumber);
   }

   public override int GetHashCode()
   {
      return this.idNumber.GetHashCode();
   }
}

public class Example6
{
   public static void Main()
   {
      Person6 p1 = new Person6("John", "63412895");
      Person6 p2 = new Person6("Jack", "63412895");
      Console.WriteLine(p1.Equals(p2));
      Console.WriteLine(Object.Equals(p1, p2));
   }
}
// The example displays the following output:
```

```
//       True
//       True
```

Equals을 재정의하는 것 외에도 IEquatable<T> 인터페이스를 구현하여 정적 타입의 같음 테스트를 제공할 수 있습니다.

다음 문장은 Equals(Object) 메서드의 모든 구현에 대해 참이어야 합니다. 목록에서 `x` `y` `z`null이 아닌 개체 참조를 나타냅니다.

- `x.Equals(x)`은(는) `true`을(를) 반환합니다.
- `x.Equals(y)`는 `y.Equals(x)`와 동일한 값을 반환합니다.
- `x.Equals(y)`는 `true`와 `x`가 둘 다 `y`인 경우 `NaN`를 반환합니다.
- `(x.Equals(y) && y.Equals(z))`가 `true`를 반환하면 `x.Equals(z)`가 `true`를 반환합니다.
- `x.Equals(y)` 및 `x`가 참조하는 개체가 수정되지 않는 한, `y`의 연속 호출은 동일한 값을 반환합니다.
- `x.Equals(null)`은(는) `false`을(를) 반환합니다.

구현은 예외를 발생시켜서는 안 되며, 항상 값을 반환해야 합니다. 예를 들어, `obj`이 `null`인 경우, Equals 메서드는 `false`를 반환해야 하며, ArgumentNullException을(를) 발생시키지 않아야 합니다.

오버라이드할 때 다음 지침을 따르십시오:Equals(Object)

- 구현하는 IComparable 형식은 Equals(Object)을 재정의해야 합니다.
- Equals(Object)를 재정의하는 형식은 GetHashCode 또한 재정의해야 합니다. 그렇지 않으면 해시 테이블이 제대로 작동하지 않을 수 있습니다.
- IEquatable<T> 인터페이스를 구현하여 같음을 위한 강력한 형식의 테스트를 지원하는 것을 고려하십시오. IEquatable<T>.Equals 구현 시 Equals와 일치하는 결과를 반환해야 합니다.
- 프로그래밍 언어가 연산자 오버로드를 지원하고 지정된 형식에 대해 동등 연산자를 오버로드하는 경우, 해당 동등 연산자와 동일한 결과를 반환하도록 Equals(Object) 메서드를 재정의해야 합니다. 이렇게 하면 사용하는 Equals 클래스 라이브러리 코드(예: ArrayList 및 Hashtable)가 애플리케이션 코드에서 같음 연산자를 사용하는 방식과 일치하는 방식으로 동작하도록 할 수 있습니다.

참조 형식에 대한 지침

Equals(Object)이(가) 참조 유형에 대해 재정의될 때 다음 지침이 적용됩니다.

- 타입의 의미 체계가 해당 타입이 어떤 값(들)을 나타낸다는 사실을 기반으로 한다면, Equals를 재정의하는 것이 좋습니다.
- 대부분의 참조 형식은 같음 연산자를 오버로드하면 안 됩니다, 비록 Equals를 재정의할지라도. 그러나 복소수 형식과 같은 값 의미 체계를 갖도록 의도된 참조 형식을 구현하는 경우 같음 연산자를 재정의해야 합니다.
- 변경 가능한 참조 유형에서는 Equals를 오버라이드해서는 안 됩니다. 이는 이전 섹션에서 설명한 대로 Equals을(를) 재정의하려면 GetHashCode 메서드도 재정의해야 하기 때문입니다. 즉, 변경 가능한 참조 형식 인스턴스의 해시 코드는 수명 동안 변경될 수 있으므로 해시 테이블에서 개체가 손실될 수 있습니다.

값 형식에 대한 지침

값 타입 Equals(Object) 를 재정의하는 경우 다음 지침이 적용됩니다.

- 값이 참조 형식인 하나 이상의 필드를 포함하는 값 형식을 정의하는 경우 Equals(Object)을(를) 재정의해야 합니다. 제공된 Equals(Object) 구현은 ValueType 필드가 모두 값 형식인 값 형식에 대해 바이트 바이트 비교를 수행하지만 리플렉션을 사용하여 필드에 참조 형식이 포함된 값 형식의 필드별 비교를 수행합니다.
- 재정의 Equals를 하고 당신의 개발 언어가 연산자 오버로드를 지원하는 경우 같음 연산자를 오버로드해야 합니다.
- 당신은 IEquatable<T> 인터페이스를 구현해야 합니다. 강력한 형식의 IEquatable<T>.Equals 메서드를 호출하면 `obj` 인수가 박싱되지 않습니다.

예시

다음 예제는 값 같음을 제공하기 위해 `Point` 메서드를 재정의하는 Equals 클래스를 보여주며, `Point3D`로부터 파생된 `Point` 클래스를 설명합니다. `Point`는 값 같음을 테스트하기 위해 Object.Equals(Object)를 재정의하므로, Object.Equals(Object) 메서드는 호출되지 않습니다. 그러나 `Point3D.Equals`는 값 같음을 제공하는 방식으로 `Point.Equals`를 구현하므로 `Point`가 Object.Equals(Object)를 호출합니다.

C#

```
using System;

class Point2
{
    protected int x, y;
```

```
    public Point2() : this(0, 0)
    { }

    public Point2(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override bool Equals(Object obj)
    {
        //Check for null and compare run-time types.
        if ((obj == null) || !this.GetType().Equals(obj.GetType()))
        {
            return false;
        }
        else
        {
            Point2 p = (Point2)obj;
            return (x == p.x) && (y == p.y);
        }
    }

    public override int GetHashCode()
    {
        return (x << 2) ^ y;
    }

    public override string ToString()
    {
        return String.Format("Point2({0}, {1})", x, y);
    }
}

sealed class Point3D : Point2
{
    int z;

    public Point3D(int x, int y, int z) : base(x, y)
    {
        this.z = z;
    }

    public override bool Equals(Object obj)
    {
        Point3D pt3 = obj as Point3D;
        if (pt3 == null)
            return false;
        else
            return base.Equals((Point2)obj) && z == pt3.z;
    }

    public override int GetHashCode()
    {
```

```
        return (base.GetHashCode() << 2) ^ z;
    }

    public override String ToString()
    {
        return String.Format("Point2({0}, {1}, {2})", x, y, z);
    }
}

class Example7
{
    public static void Main()
    {
        Point2 point2D = new Point2(5, 5);
        Point3D point3Da = new Point3D(5, 5, 2);
        Point3D point3Db = new Point3D(5, 5, 2);
        Point3D point3Dc = new Point3D(5, 5, -1);

        Console.WriteLine($"{point2D} = {point3Da}: {point2D.Equals(point3Da)}");
        Console.WriteLine($"{point2D} = {point3Db}: {point2D.Equals(point3Db)}");
        Console.WriteLine($"{point3Da} = {point3Db}:
{point3Da.Equals(point3Db)}");
        Console.WriteLine($"{point3Da} = {point3Dc}:
{point3Da.Equals(point3Dc)}");
    }
}
// The example displays the following output:
//       Point2(5, 5) = Point2(5, 5, 2): False
//       Point2(5, 5) = Point2(5, 5, 2): False
//       Point2(5, 5, 2) = Point2(5, 5, 2): True
//       Point2(5, 5, 2) = Point2(5, 5, -1): False
```

메서드는 `Point.Equals` 인수가 `obj`이 아니고 이 개체와 동일한 형식의 인스턴스를 참조하는지 확인합니다. 확인 중 하나가 실패하면 메서드가 반환됩니다 `false`.

메서드는 `Point.Equals` 메서드를 GetType 호출하여 두 개체의 런타임 형식이 동일한지 여부를 확인합니다. 메서드가 C#에서 `obj is Point` 형태로 또는 Visual Basic에서 `TryCast(obj, Point)` 형태로 검사를 사용한 경우, `true`와 현재 인스턴스의 런타임 형식이 동일하지 않더라도 `true`가 `Point`의 파생 클래스인 경우 `obj` 검사가 를 반환합니다. 두 개체가 동일한 형식임을 확인한 후 메서드는 형식 `obj` 으로 캐스팅 `Point` 되고 두 개체의 인스턴스 필드를 비교한 결과를 반환합니다.

`Point3D.Equals`에서 상속된 `Point.Equals` 메서드는 Object.Equals(Object)를 재정의하며 다른 작업이 수행되기 전에 호출됩니다. `Point3D`는 봉인된 클래스(`NotInheritable` Visual Basic)이므로, C#의 `obj is Point` 형식이나 Visual Basic의 `TryCast(obj, Point)` 형식으로 확인하면 `obj`가 `Point3D` 객체인지 확인하기에 충분합니다. `Point3D` 객체인 경우 `Point` 객체로 캐스팅되어 Equals의 기본 클래스 구현에 전달됩니다. 상속된 `Point.Equals` 메서드가 `true` 반환될 때만, 메서드는 파생 클래스에서 도입된 `z` 인스턴스 필드를 비교합니다.

다음 예제에서는 내부적으로 사각형을 `Rectangle` 두 개체 `Point` 로 구현하는 클래스를 정의합니다. `Rectangle` 클래스는 값 같음을 구현하기 위해 Object.Equals(Object)를 재정의합니다.

C#

```
using System;

class Rectangle
{
   private Point a, b;

   public Rectangle(int upLeftX, int upLeftY, int downRightX, int downRightY)
   {
      this.a = new Point(upLeftX, upLeftY);
      this.b = new Point(downRightX, downRightY);
   }

   public override bool Equals(Object obj)
   {
      // Perform an equality check on two rectangles (Point object pairs).
      if (obj == null || GetType() != obj.GetType())
          return false;
      Rectangle r = (Rectangle)obj;
      return a.Equals(r.a) && b.Equals(r.b);
   }

   public override int GetHashCode()
   {
      return Tuple.Create(a, b).GetHashCode();
   }

    public override String ToString()
    {
       return String.Format("Rectangle({0}, {1}, {2}, {3})",
                            a.x, a.y, b.x, b.y);
    }
}

class Point
{
  internal int x;
  internal int y;

  public Point(int X, int Y)
  {
     this.x = X;
     this.y = Y;
  }

  public override bool Equals (Object obj)
  {
     // Performs an equality check on two points (integer pairs).
     if (obj == null || GetType() != obj.GetType()) return false;
```

```
      Point p = (Point)obj;
      return (x == p.x) && (y == p.y);
   }

   public override int GetHashCode()
   {
      return Tuple.Create(x, y).GetHashCode();
   }
}

class Example
{
   public static void Main()
   {
      Rectangle r1 = new Rectangle(0, 0, 100, 200);
      Rectangle r2 = new Rectangle(0, 0, 100, 200);
      Rectangle r3 = new Rectangle(0, 0, 150, 200);

      Console.WriteLine($"{r1} = {r2}: {r1.Equals(r2)}");
      Console.WriteLine($"{r1} = {r3}: {r1.Equals(r3)}");
      Console.WriteLine($"{r2} = {r3}: {r2.Equals(r3)}");
   }
}
// The example displays the following output:
//    Rectangle(0, 0, 100, 200) = Rectangle(0, 0, 100, 200): True
//    Rectangle(0, 0, 100, 200) = Rectangle(0, 0, 150, 200): False
//    Rectangle(0, 0, 100, 200) = Rectangle(0, 0, 150, 200): False
```

일부 언어, 예를 들어 C# 및 Visual Basic은 연산자 오버로딩을 지원합니다. 형식이 같음 연산자를 오버로드하는 경우, 동일한 기능을 제공하기 위해 EqualsEquals(Object) 메서드를 반드시 재정의해야 합니다. 이 작업은 일반적으로 다음 예제와 같이 오버로드된 같음 연산자를 기준으로 메서드를 작성 Equals(Object) 하여 수행됩니다.

C#

```
using System;

public struct Complex
{
   public double re, im;

   public override bool Equals(Object obj)
   {
      return obj is Complex && this == (Complex)obj;
   }

   public override int GetHashCode()
   {
      return Tuple.Create(re, im).GetHashCode();
   }

   public static bool operator ==(Complex x, Complex y)
```

```
    {
       return x.re == y.re && x.im == y.im;
    }

    public static bool operator !=(Complex x, Complex y)
    {
       return !(x == y);
    }

     public override String ToString()
     {
        return String.Format("({0}, {1})", re, im);
     }
}

class MyClass
{
  public static void Main()
  {
    Complex cmplx1, cmplx2;

    cmplx1.re = 4.0;
    cmplx1.im = 1.0;

    cmplx2.re = 2.0;
    cmplx2.im = 1.0;

    Console.WriteLine($"{cmplx1} <> {cmplx2}: {cmplx1 != cmplx2}");
    Console.WriteLine($"{cmplx1} = {cmplx2}: {cmplx1.Equals(cmplx2)}");

    cmplx2.re = 4.0;

    Console.WriteLine($"{cmplx1} = {cmplx2}: {cmplx1 == cmplx2}");
    Console.WriteLine($"{cmplx1} = {cmplx2}: {cmplx1.Equals(cmplx2)}");
  }
}
// The example displays the following output:
//       (4, 1) <> (2, 1): True
//       (4, 1) = (2, 1): False
//       (4, 1) = (4, 1): True
//       (4, 1) = (4, 1): True
```

`Complex` 값 형식이므로 파생될 수 없습니다. 따라서 Equals(Object) 메서드에 대한 재정의는 각 개체의 정확한 런타임 형식을 결정하기 위해 GetType을 호출할 필요가 없으며, 대신 C#의 `is` 연산자나 Visual Basic의 `TypeOf` 연산자를 사용하여 `obj` 매개 변수의 형식을 확인할 수 있습니다.

System.Object.Finalize 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 Finalize 메서드는 개체가 제거되기 전에 현재 개체가 보유한 관리되지 않는 리소스에 대한 정리 작업을 수행하는 데 사용됩니다. 이 메서드는 보호되므로 이 클래스를 통해서만 또는 파생 클래스를 통해서만 액세스할 수 있습니다.

최종화 작동 방법

클래스는 Object 클래스에 대한 구현을 제공하지 않으며, Finalize 메서드를 재정의하지 않는 한 가비지 수집기는 파생된 Object 형식을 종료 대상으로 표시하지 않습니다.

형식이 Finalize 메서드를 재정의하는 경우, 가비지 수집기는 형식의 각 인스턴스를 종료 대기열이라 불리는 내부 구조에 추가합니다. 종료 큐에는 가비지 수집기가 메모리를 회수하기 전에 종료 코드를 실행해야 하는 관리되는 힙의 모든 개체에 대한 항목이 포함됩니다. 가비지 수집기는 다음 조건에서 Finalize 메서드를 자동으로 호출합니다.

- 가비지 수집기가 개체가 접근 불가능하다는 것을 발견한 후, 메서드를 호출하여 개체를 종료에서 제외하지 않는 한 해당 개체에 액세스할 수 없습니다.
- **.NET Framework에서만** 개체가 종료에서 제외되지 않는 한 애플리케이션 도메인을 종료하는 동안에만 적용됩니다. 종료하는 동안 여전히 액세스할 수 있는 개체도 처리됩니다.

Finalize 와 같은 GC.ReRegisterForFinalize 메커니즘을 사용하여 개체를 다시 등록하고 메서드가 이후에 호출되지 않는 한 지정된 인스턴스에서 GC.SuppressFinalize 한 번만 자동으로 호출됩니다.

Finalize 작업에는 다음과 같은 제한 사항이 있습니다.

- 종료자가 실행되는 정확한 시간은 정의되지 않습니다. 클래스 인스턴스에 대한 리소스의 결정적 릴리스를 보장하려면 메서드를 `Close` 구현하거나 구현을 IDisposable.Dispose 제공합니다.
- 물체 하나가 다른 물체를 참조하더라도, 두 물체의 파이널라이저는 특정 순서로 실행될 것이라는 보장은 없습니다. 즉, 개체 A에 개체 B에 대한 참조가 있고 둘 다 종료자가 있는 경우 개체 A의 종료자가 시작될 때 개체 B가 이미 종료되었을 수 있습니다.
- 종료자가 실행되는 스레드는 지정되지 않습니다.

메서드가 Finalize 완료될 때 실행되지 않거나 다음과 같은 예외적인 상황에서 전혀 실행되지 않을 수 있습니다.

- 다른 종료자가 무기한으로 차단되는 경우(예: 무한 루프에 빠지거나 절대 획득할 수 없는 잠금을 얻으려고 시도하는 경우 등). 런타임이 완료될 때 종료자를 실행하려고 하기 때문에 종료자가 무기한 차단되면 다른 종료자가 호출되지 않을 수 있습니다.
- 런타임에 정리 기회를 주지 않고 프로세스가 종료되는 경우 이 경우 런타임의 첫 번째 프로세스 종료 알림은 DLL_PROCESS_DETACH 알림입니다.

런타임은 종료 시, 종료할 수 있는 개체 수가 계속 감소하는 경우에만 개체를 계속해서 정리합니다.

Finalize 또는 Finalize의 재정의가 예외를 throw하고, 런타임이 기본 정책을 재정의하는 애플리케이션에 의해 호스팅되지 않는 경우, 런타임은 프로세스를 종료하며, 활성 `try`/`finally` 블록이나 종료자가 실행되지 않습니다. 이 동작은 종료자가 리소스를 해제하거나 제거할 수 없는 경우 프로세스 무결성을 보장합니다.

Finalize 메서드 재정의

클래스가 파일 핸들이나 데이터베이스 연결과 같은 비관리 리소스를 사용하는 경우, 가비지 수집 중에 이를 사용하는 관리 개체가 폐기될 때 해당 리소스를 해제해야 하므로, Finalize를 재정의해야 합니다. 가비지 수집기가 관리되는 리소스를 자동으로 해제하므로, 관리되는 개체에 대해 별도의 메서드를 구현할 필요가 없습니다.

> **ⓘ 중요**
>
> 관리되지 않는 리소스를 SafeHandle 래핑하는 개체를 사용할 수 있는 경우, Finalize을 재정의하지 않고 안전한 핸들로 Dispose 패턴을 구현하는 것이 좋습니다. 자세한 내용은 SafeHandle 대체 섹션을 참조하세요.

이 Object.Finalize 메서드는 기본적으로 아무 작업도 수행하지 않지만, 필요한 경우에만 Finalize를 오버라이드하여 관리되지 않는 리소스를 해제해야 합니다. 메모리 재할당은 종료 작업이 실행되는 경우 최소 두 번의 가비지 수집이 필요하기 때문에 훨씬 더 오래 걸리는 경향이 있습니다. 또한 참조 형식에 대해서만 Finalize 메서드를 재정의해야 합니다. 공용 언어 런타임은 참조 형식만 완료합니다. 값 형식의 종료자는 무시됩니다.

메서드의 범위는 Object.Finalize `protected`입니다. 클래스에서 메서드를 재정의할 때 이 제한된 범위를 유지해야 합니다. 메서드를 Finalize 보호하여 애플리케이션 사용자가 개체의 Finalize 메서드를 직접 호출하지 못하게 합니다.

파생 형식의 Finalize 모든 구현은 기본 형식의 구현 Finalize을 호출해야 합니다. 애플리케이션 코드를 호출 Finalize할 수 있는 유일한 경우입니다. 개체의 Finalize 메서드는 기본 클래스 이외의 개체에서 메서드를 호출하면 안 됩니다. 이는 공용 언어 런타임 종료의 경우와 같이 호출되는 다른 개체를 호출 개체와 동시에 수집할 수 있기 때문입니다.

> ⓘ **참고**
>
> C# 컴파일러는 **Finalize** 메서드를 재정의할 수 없도록 합니다. 대신, 클래스에 대해 소멸자를 구현하여 종료 처리를 제공합니다. C# 소멸자가 해당 기본 클래스의 소멸자를 자동으로 호출합니다.
>
> 또한 Visual C++는 메서드를 구현하기 **Finalize** 위한 고유한 구문을 제공합니다. 자세한 내용은 **방법: 클래스 및 구조체 정의 및 사용(C++/CLI)**의 "소멸자 및 종료자" 섹션을 참조하세요.

가비지 수집은 비결정적이므로 가비지 수집기가 종료를 수행하는 시기를 정확하게 알 수 없습니다. 리소스를 즉시 해제하려면 삭제 패턴 및 IDisposable 인터페이스를 구현하도록 선택할 수도 있습니다. 클래스의 소비자는 IDisposable.Dispose 구현을 호출하여 관리되지 않는 리소스를 해제할 수 있으며, Finalize 메서드를 사용하여 Dispose 메서드가 호출되지 않을 경우 관리되지 않는 리소스를 해제할 수 있습니다.

Finalize 가비지 수집 중에 개체를 정리한 후 개체를 다시 액세스할 수 있도록 하는 작업을 포함하여 거의 모든 작업을 수행할 수 있습니다. 그러나 개체는 한 번만 부활할 수 있습니다. Finalize 가비지 수집 중에는 부활된 개체에 대해 호출할 수 없습니다.

SafeHandle 대안

신뢰할 수 있는 종료자를 만드는 것은 종종 어려운데, 그 이유는 애플리케이션의 상태를 가정할 수 없고 OutOfMemoryException 및 StackOverflowException와 같은 처리되지 않은 시스템 예외가 종료자를 종료시키기 때문입니다. 클래스에서 관리되지 않는 리소스를 해제하기 위해 종료자를 구현하는 대신 클래스에서 System.Runtime.InteropServices.SafeHandle 파생된 개체를 사용하여 관리되지 않는 리소스를 래핑한 다음 종료자 없이 삭제 패턴을 구현할 수 있습니다. .NET Framework는 Microsoft.Win32에서 파생된 System.Runtime.InteropServices.SafeHandle 네임스페이스의 다음 클래스를 제공합니다.

- SafeFileHandle 는 파일 핸들에 대한 래퍼 클래스입니다.
- SafeMemoryMappedFileHandle 는 메모리 매핑된 파일 핸들에 대한 래퍼 클래스입니다.
- SafeMemoryMappedViewHandle 는 관리되지 않는 메모리 블록에 대한 포인터에 대한 래퍼 클래스입니다.
- SafeNCryptKeyHandle, SafeNCryptProviderHandle및 SafeNCryptSecretHandle 암호화 핸들에 대한 래퍼 클래스입니다.
- SafePipeHandle는 파이프 핸들에 대한 래퍼(wrapper) 클래스입니다.
- SafeRegistryHandle 는 레지스트리 키에 대한 핸들에 대한 래퍼 클래스입니다.
- SafeWaitHandle 는 대기 핸들에 대한 래퍼 클래스입니다.

다음 예제에서는 메서드를 재정의하는 대신 안전한 핸들과 함께 dispose 패턴을 사용합니다. 특정 파일 확장자를 사용하여 `FileAssociation` 파일을 처리하는 애플리케이션에 대한 레지스트리 정보를 래핑하는 클래스를 정의합니다. Windows `out` 함수 호출에 의해 매개 변수로 반환된 SafeRegistryHandle 두 개의 레지스트리 핸들이 생성자에 전달됩니다. 형식의 보호된 `Dispose` 메서드는 `SafeRegistryHandle.Dispose` 메서드를 호출하여 이 두 핸들을 해제합니다.

C#

```
using Microsoft.Win32.SafeHandles;
using System;
using System.ComponentModel;
using System.IO;
using System.Runtime.InteropServices;

public class FileAssociationInfo : IDisposable
{
   // Private variables.
   private String ext;
   private String openCmd;
   private String args;
   private SafeRegistryHandle hExtHandle, hAppIdHandle;

   // Windows API calls.
   [DllImport("advapi32.dll", CharSet= CharSet.Auto, SetLastError=true)]
   private static extern int RegOpenKeyEx(IntPtr hKey,
                  String lpSubKey, int ulOptions, int samDesired,
                  out IntPtr phkResult);
   [DllImport("advapi32.dll", CharSet= CharSet.Unicode, EntryPoint =
"RegQueryValueExW",
               SetLastError=true)]
   private static extern int RegQueryValueEx(IntPtr hKey,
                  string lpValueName, int lpReserved, out uint lpType,
                  string lpData, ref uint lpcbData);
   [DllImport("advapi32.dll", SetLastError = true)]
   private static extern int RegSetValueEx(IntPtr hKey,
[MarshalAs(UnmanagedType.LPStr)] string lpValueName,
                  int Reserved, uint dwType, [MarshalAs(UnmanagedType.LPStr)]
string lpData,
                  int cpData);
   [DllImport("advapi32.dll", SetLastError=true)]
   private static extern int RegCloseKey(UIntPtr hKey);

   // Windows API constants.
   private const int HKEY_CLASSES_ROOT = unchecked((int) 0x80000000);
   private const int ERROR_SUCCESS = 0;

    private const int KEY_QUERY_VALUE = 1;
    private const int KEY_SET_VALUE = 0x2;

   private const uint REG_SZ = 1;

   private const int MAX_PATH = 260;
```

```
   public FileAssociationInfo(String fileExtension)
   {
      int retVal = 0;
      uint lpType = 0;

      if (!fileExtension.StartsWith("."))
            fileExtension = "." + fileExtension;
      ext = fileExtension;

      IntPtr hExtension = IntPtr.Zero;
      // Get the file extension value.
      retVal = RegOpenKeyEx(new IntPtr(HKEY_CLASSES_ROOT), fileExtension, 0,
KEY_QUERY_VALUE, out hExtension);
      if (retVal != ERROR_SUCCESS)
         throw new Win32Exception(retVal);
      // Instantiate the first SafeRegistryHandle.
      hExtHandle = new SafeRegistryHandle(hExtension, true);

      string appId = new string(' ', MAX_PATH);
      uint appIdLength = (uint) appId.Length;
      retVal = RegQueryValueEx(hExtHandle.DangerousGetHandle(), String.Empty, 0,
out lpType, appId, ref appIdLength);
      if (retVal != ERROR_SUCCESS)
         throw new Win32Exception(retVal);
      // We no longer need the hExtension handle.
      hExtHandle.Dispose();

      // Determine the number of characters without the terminating null.
      appId = appId.Substring(0, (int) appIdLength / 2 - 1) +
@"\shell\open\Command";

      // Open the application identifier key.
      string exeName = new string(' ', MAX_PATH);
      uint exeNameLength = (uint) exeName.Length;
      IntPtr hAppId;
      retVal = RegOpenKeyEx(new IntPtr(HKEY_CLASSES_ROOT), appId, 0,
KEY_QUERY_VALUE | KEY_SET_VALUE,
                            out hAppId);
       if (retVal != ERROR_SUCCESS)
         throw new Win32Exception(retVal);

      // Instantiate the second SafeRegistryHandle.
      hAppIdHandle = new SafeRegistryHandle(hAppId, true);

      // Get the executable name for this file type.
      string exePath = new string(' ', MAX_PATH);
      uint exePathLength = (uint) exePath.Length;
      retVal = RegQueryValueEx(hAppIdHandle.DangerousGetHandle(), String.Empty, 0,
out lpType, exePath, ref exePathLength);
      if (retVal != ERROR_SUCCESS)
         throw new Win32Exception(retVal);

      // Determine the number of characters without the terminating null.
      exePath = exePath.Substring(0, (int) exePathLength / 2 - 1);
```

```
      // Remove any environment strings.
      exePath = Environment.ExpandEnvironmentVariables(exePath);

      int position = exePath.IndexOf('%');
      if (position >= 0) {
         args = exePath.Substring(position);
         // Remove command line parameters ('%0', etc.).
         exePath = exePath.Substring(0, position).Trim();
      }
      openCmd = exePath;
   }

   public String Extension
   { get { return ext; } }

   public String Open
   { get { return openCmd; }
     set {
        if (hAppIdHandle.IsInvalid | hAppIdHandle.IsClosed)
           throw new InvalidOperationException("Cannot write to registry key.");
        if (! File.Exists(value)) {
           string message = String.Format("'{0}' does not exist", value);
           throw new FileNotFoundException(message);
        }
        string cmd = value + " %1";
        int retVal = RegSetValueEx(hAppIdHandle.DangerousGetHandle(),
String.Empty, 0,
                                   REG_SZ, value, value.Length + 1);
        if (retVal != ERROR_SUCCESS)
           throw new Win32Exception(retVal);
     } }

   public void Dispose()
   {
      Dispose(disposing: true);
      GC.SuppressFinalize(this);
   }

   protected void Dispose(bool disposing)
   {
      // Ordinarily, we release unmanaged resources here;
      // but all are wrapped by safe handles.

      // Release disposable objects.
      if (disposing) {
         if (hExtHandle != null) hExtHandle.Dispose();
         if (hAppIdHandle != null) hAppIdHandle.Dispose();
      }
   }
}
```

System.Object.GetHashCode 메서드

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 메서드는 GetHashCode 개체 같음의 빠른 검사가 필요한 알고리즘에 대한 해시 코드를 제공합니다. 해시 코드는 클래스, Dictionary<TKey,TValue> 클래스 또는 클래스에서 파생된 형식과 같은 Hashtable 해시 기반 컬렉션에서 DictionaryBase 개체를 삽입하고 식별하는 데 사용되는 숫자 값입니다.

> ⓘ **참고 항목**
>
> 해시 테이블에서 해시 코드를 사용하는 방법과 몇 가지 추가 해시 코드 알고리즘에 대한 자세한 내용은 Wikipedia의 **해시 함수** 항목을 참조하세요.

동일한 두 개체는 동일한 해시 코드를 반환합니다. 그러나 반대는 true가 아닙니다. 다른(같지 않은) 개체에는 동일한 해시 코드가 있을 수 있기 때문에 동일한 해시 코드가 개체 같음을 의미하지는 않습니다. 또한 .NET은 메서드의 GetHashCode 기본 구현을 보장하지 않으며, 이 메서드가 반환하는 값은 .NET Framework 및 .NET Core의 다른 버전과 같은 .NET 구현과 32비트 및 64비트 플랫폼과 같은 플랫폼 간에 다를 수 있습니다. 이러한 이유로 이 메서드의 기본 구현을 해시 용도로 고유한 개체 식별자로 사용하지 마세요. 두 가지 결과는 다음과 같습니다.

- 동일한 해시 코드가 개체 같음을 암시한다고 가정해서는 안 됩니다.
- 동일한 개체가 애플리케이션 도메인, 프로세스 및 플랫폼에서 해시할 수 있으므로 생성된 애플리케이션 도메인 외부에서 해시 코드를 유지하거나 사용하면 안 됩니다.

> ⚠ **Warning**
>
> 해시 코드는 해시 테이블을 기반으로 하는 컬렉션에서 효율적인 삽입 및 조회를 위한 것입니다. 해시 코드는 영구 값이 아닙니다. 이러한 이유로 다음을 수행합니다.
>
> - 해시 코드 값을 직렬화하거나 데이터베이스에 저장하지 마세요.
> - 키 컬렉션에서 개체를 검색하는 키로 해시 코드를 사용하지 마세요.
> - 애플리케이션 도메인 또는 프로세스 간에 해시 코드를 보내지 마세요. 경우에 따라 해시 코드는 프로세스별 또는 애플리케이션별 도메인 기준으로 계산될 수 있습니다.
> - 암호화에 강력한 해시가 필요한 경우 암호화 해시 함수에서 반환하는 값 대신 해시 코드를 사용하지 마세요. 암호화 해시의 경우, System.Security.Cryptography.HashAlgorithm 클래스 또는

System.Security.Cryptography.KeyedHashAlgorithm 클래스에서 파생된 클래스를 사용합니다.

- 두 개체가 같은지 여부를 확인하기 위해 해시 코드의 같음을 테스트하지 마세요. (같지 않은 개체는 동일한 해시 코드를 가질 수 있습니다.) 같음을 테스트하려면 메서드 ReferenceEquals 또는 메서드 Equals를 호출하세요.

GetHashCode 메서드는 파생된 형식에 의해 재정의될 수 있습니다. 재정의되지 않은 경우 GetHashCode 참조 형식에 대한 해시 코드는 개체의 참조를 기반으로 해시 코드를 계산하는 기본 클래스의 메서드를 호출 Object.GetHashCode 하여 계산됩니다. 자세한 내용은 다음을 참조하세요 RuntimeHelpers.GetHashCode. 즉, 메서드가 반환 ReferenceEquals 하는 두 개체에는 `true` 동일한 해시 코드가 있습니다. 값 형식이 GetHashCode을(를) 재정의하지 않으면 기본 클래스의 ValueType.GetHashCode 메서드는 리플렉션을 사용하여 형식 필드의 값을 기반으로 해시 코드를 계산합니다. 즉, 필드 값이 같은 값이 있는 값 형식에는 해시 코드가 같습니다. GetHashCode를 재정의하는 방법에 대한 자세한 내용은 "상속자 참고사항" 섹션을 참조하세요.

> ⚠ **Warning**
>
> GetHashCode 메서드를 재정의하는 경우 Equals도 재정의해야 하며, 그 반대의 경우도 마찬가지입니다. 두 객체가 동등성에 대해 테스트되는 경우 재정의된 Equals 메서드가 `true` 를 반환하면, 재정의된 GetHashCode 메서드는 두 객체에 대해 동일한 값을 반환해야 합니다.

해시 테이블에서 키로 사용되는 개체가 유용한 구현GetHashCode을 제공하지 않는 경우 클래스 생성자의 오버로드 IEqualityComparer 중 하나에 구현을 Hashtable 제공하여 해시 코드 공급자를 지정할 수 있습니다.

Windows 런타임에 대한 참고 사항

Windows 런타임의 클래스에서 GetHashCode 메서드를 호출할 때, GetHashCode를 재정의하지 않는 클래스에 대한 기본 동작을 제공합니다. 이는 .NET이 Windows 런타임에 대해 제공하는 지원의 일부입니다(Windows 스토어 앱 및 Windows 런타임에 대한 .NET 지원 참조). Windows 런타임의 클래스는 Object을(를) 상속받지 않으며 현재 GetHashCode을(를) 구현하지 않고 있습니다. 그러나 C# 또는 Visual Basic 코드에서 사용할 때는 ToString, Equals(Object), GetHashCode 메서드가 있는 것처럼 보이며, .NET Framework는 이러한 메서드에 대한 기본 동작을 제공합니다.

> ⓘ **참고 항목**

C# 또는 Visual Basic으로 작성된 Windows 런타임 클래스는 GetHashCode 메서드를 재정의할 수 있습니다.

예시

형식과 Int32 같거나 작은 범위의 숫자 값에 대한 해시 코드를 계산하는 가장 간단한 방법 중 하나는 해당 값을 반환하는 것입니다. 다음 예제는 `Number` 구조체의 이러한 구현을 설명합니다.

C#

```
using System;

public struct Number
{
   private int n;

   public Number(int value)
   {
      n = value;
   }

   public int Value
   {
      get { return n; }
   }

   public override bool Equals(Object obj)
   {
      if (obj == null || ! (obj is Number))
         return false;
      else
         return n == ((Number) obj).n;
   }

   public override int GetHashCode()
   {
      return n;
   }

   public override string ToString()
   {
      return n.ToString();
   }
}

public class Example1
{
   public static void Main()
   {
      Random rnd = new Random();
```

```
      for (int ctr = 0; ctr <= 9; ctr++) {
         int randomN = rnd.Next(Int32.MinValue, Int32.MaxValue);
         Number n = new Number(randomN);
         Console.WriteLine("n = {0,12}, hash code = {1,12}", n, n.GetHashCode());
      }
   }
}
// The example displays output like the following:
//       n =   -634398368, hash code =   -634398368
//       n =   2136747730, hash code =   2136747730
//       n =  -1973417279, hash code =  -1973417279
//       n =   1101478715, hash code =   1101478715
//       n =   2078057429, hash code =   2078057429
//       n =   -334489950, hash code =   -334489950
//       n =    -68958230, hash code =    -68958230
//       n =   -379951485, hash code =   -379951485
//       n =    -31553685, hash code =    -31553685
//       n =   2105429592, hash code =   2105429592
```

형식에는 해시 코드 생성에 참여할 수 있는 여러 데이터 필드가 있는 경우가 많습니다. 해시 코드를 생성하는 한 가지 방법은 다음 예제와 같이 작업을 사용하여 `XOR (eXclusive OR)` 이러한 필드를 결합하는 것입니다.

C#

```
using System;

// A type that represents a 2-D point.
public struct Point2
{
    private int x;
    private int y;

    public Point2(int x, int y)
    {
       this.x = x;
       this.y = y;
    }

    public override bool Equals(Object obj)
    {
       if (! (obj is Point2)) return false;

       Point2 p = (Point2) obj;
       return x == p.x & y == p.y;
    }

    public override int GetHashCode()
    {
        return x ^ y;
    }
}
```

```
public class Example3
{
   public static void Main()
   {
      Point2 pt = new Point2(5, 8);
      Console.WriteLine(pt.GetHashCode());

      pt = new Point2(8, 5);
      Console.WriteLine(pt.GetHashCode());
   }
}
// The example displays the following output:
//       13
//       13
```

이전 예제에서는 (n1, n2) 및 (n2, n1)에 대해 동일한 해시 코드를 반환하므로 바람직한 것보다 더 많은 충돌을 생성할 수 있습니다. .NET 5 이상에서 권장되는 솔루션은 HashCode.Combine. 대칭 문제를 피하고 `Tuple` 개체의 오버헤드 없이 잘 분산된 해시 코드를 생성합니다.

C#

```
using System;

public struct Point3
{
    private int x;
    private int y;

    public Point3(int x, int y)
    {
       this.x = x;
       this.y = y;
    }

    public override bool Equals(Object obj)
    {
        if (obj is Point3)
        {
            Point3 p = (Point3) obj;
            return x == p.x & y == p.y;
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(x, y);
    }
}
```

```
public class Example
{
   public static void Main()
   {
        Point3 pt = new Point3(5, 8);
        Console.WriteLine(pt.GetHashCode());

        pt = new Point3(8, 5);
        Console.WriteLine(pt.GetHashCode());
   }
}
// The example displays output similar to the following.
// Note: HashCode.Combine results are not stable across .NET versions.
//       185727722
//       -363254492
```

.NET Framework의 대안은 연속 필드의 해시 코드를 두 비트 이상 왼쪽으로 이동하여 개별 해시 코드의 가중치를 지정하는 것입니다. 최적으로 비트 31 이상으로 이동한 비트는 삭제하지 않고 래핑해야 합니다. C# 및 Visual Basic 모두에서 왼쪽 시프트 연산자가 비트를 삭제하므로 다음과 같이 왼쪽 시프트 및 래핑 메서드를 만들어야 합니다.

C#

```
public int ShiftAndWrap(int value, int positions)
{
    positions = positions & 0x1F;

    // Save the existing bit pattern, but interpret it as an unsigned integer.
    uint number = BitConverter.ToUInt32(BitConverter.GetBytes(value), 0);
    // Preserve the bits to be discarded.
    uint wrapped = number >> (32 - positions);
    // Shift and wrap the discarded bits.
    return BitConverter.ToInt32(BitConverter.GetBytes((number << positions) |
wrapped), 0);
}
```

다음 예제에서는 이 shift-and-wrap 메서드를 사용하여 이전 예제에서 사용된 구조체의 `Point` 해시 코드를 계산합니다.

C#

```
using System;

public struct Point
{
    private int x;
    private int y;

    public Point(int x, int y)
    {
```

```
        this.x = x;
        this.y = y;
    }

    public override bool Equals(Object obj)
    {
        if (!(obj is Point)) return false;

        Point p = (Point) obj;
        return x == p.x & y == p.y;
    }

    public override int GetHashCode()
    {
        return ShiftAndWrap(x.GetHashCode(), 2) ^ y.GetHashCode();
    }

    private int ShiftAndWrap(int value, int positions)
    {
        positions = positions & 0x1F;

        // Save the existing bit pattern, but interpret it as an unsigned integer.
        uint number = BitConverter.ToUInt32(BitConverter.GetBytes(value), 0);
        // Preserve the bits to be discarded.
        uint wrapped = number >> (32 - positions);
        // Shift and wrap the discarded bits.
        return BitConverter.ToInt32(BitConverter.GetBytes((number << positions) |
wrapped), 0);
    }
}

public class Example2
{
   public static void Main()
   {
        Point pt = new Point(5, 8);
        Console.WriteLine(pt.GetHashCode());

        pt = new Point(8, 5);
        Console.WriteLine(pt.GetHashCode());
   }
}
// The example displays the following output:
//       28
//       37
```

Last updated on 2026. 03. 18.

System.Object.ToString 메서드

아티클 • 2025. 04. 05.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Object.ToString 는 .NET의 일반적인 서식 지정 메서드입니다. 표시에 적합하도록 개체를 문자열 표현으로 변환합니다. (.NET의 서식 지원에 대한 자세한 내용은 형식 서식 지정을 참조하세요.) 메서드의 Object.ToString 기본 구현은 개체 형식의 정규화된 이름을 반환합니다.

> ⓘ **중요**
>
> 다른 유형의 멤버 목록에서 링크를 따라 이 페이지에 도달했을 수 있습니다. 해당 유형이 Object.ToString을(를) 오버라이드하지 않기 때문입니다. 대신 메서드의 Object.ToString 기능을 상속합니다.

형식은 특정 형식의 Object.ToString 보다 적합한 문자열 표현을 제공하기 위해 메서드를 재정의하는 경우가 많습니다. 형식은 메서드 Object.ToString를 오버로드하여 형식 문자열이나 문화권 구분 서식 지정을 지원하는 경우가 많습니다.

기본 Object.ToString() 메서드

메서드의 ToString 기본 구현은 다음 예제에서 보여주는 것처럼 Object의 형식에 대한 완전한 이름을 반환합니다.

C#

```
Object obj = new Object();
Console.WriteLine(obj.ToString());

// The example displays the following output:
//       System.Object
```

Object는 .NET의 모든 참조 형식의 기본 클래스이므로, ToString 메서드를 재정의하지 않는 참조 형식에 의해 이 동작이 상속됩니다. 다음 예제에서는 이를 보여 줍니다. 모든 Object 멤버의 기본 구현을 허용하는 명명된 `Object1` 클래스를 정의합니다. 해당 메서드는 ToString 개체의 정규화된 형식 이름을 반환합니다.

C#

```
using System;
using Examples;

namespace Examples
```

```
{
   public class Object1
   {
   }
}

public class Example5
{
   public static void Main()
   {
      object obj1 = new Object1();
      Console.WriteLine(obj1.ToString());
   }
}
// The example displays the following output:
//    Examples.Object1
```

Object.ToString() 메서드 재정의

형식은 일반적으로 개체 인스턴스를 Object.ToString 나타내는 문자열을 반환하도록 메서드를 재정의합니다. 예를 들어, Char, Int32, 및 String와 같은 기본 형식은 개체가 나타내는 값의 문자열 형식을 반환하는 ToString 구현을 제공합니다. 다음 예제에서는 ToString 메서드를 재정의하여 해당 값과 함께 형식 이름을 반환하도록 하는 클래스 `Object2` 를 정의합니다.

C#

```
using System;

public class Object2
{
   private object value;

   public Object2(object value)
   {
      this.value = value;
   }

   public override string ToString()
   {
      return base.ToString() + ": " + value.ToString();
   }
}

public class Example6
{
   public static void Main()
   {
      Object2 obj2 = new Object2('a');
      Console.WriteLine(obj2.ToString());
   }
```

```
}
// The example displays the following output:
//       Object2: a
```

다음 표에서는 .NET의 형식 범주를 나열하고 Object.ToString 메서드를 재정의하는지 여부를 나타냅니다.

테이블 확장

유형 범주	Object.ToString()을 재정의합니다.	행동
클래스	n/a	n/a
구조	예(ValueType.ToString)	`Object.ToString()` 동일
열거	예(Enum.ToString())	멤버 이름
인터페이스	아니오	n/a
대리인	아니오	n/a

상속자에 대한 참고 섹션에서 ToString 오버라이딩에 대한 추가 정보를 참조하세요.

ToString 메서드 오버로드

매개 변수 없는 Object.ToString() 메서드를 재정의하는 것 외에도 많은 형식이 `ToString` 메서드를 오버로드하여 매개 변수를 허용하는 메서드의 버전을 제공합니다. 가장 일반적으로 이 작업은 변수 서식 지정 및 문화권 구분 서식을 지원하기 위해 수행됩니다.

다음 예제에서는 `ToString` 메서드를 오버로드하여 `Automobile` 클래스의 다양한 필드 값을 포함하는 결과 문자열을 반환합니다. 모델 이름과 연도를 반환하는 4개의 형식 문자열인 G를 정의합니다. D: 모델 이름, 연도 및 문 수를 반환합니다. C는 모델 이름, 연도 및 실린더 수를 반환합니다. 및 4개의 필드 값이 모두 있는 문자열을 반환하는 A입니다.

C#

```
using System;

public class Automobile
{
   private int _doors;
   private string _cylinders;
   private int _year;
   private string _model;

   public Automobile(string model, int year , int doors,
                     string cylinders)
```

```
    {
        _model = model;
        _year = year;
        _doors = doors;
        _cylinders = cylinders;
    }

    public int Doors
    { get { return _doors; } }

    public string Model
    { get { return _model; } }

    public int Year
    { get { return _year; } }

    public string Cylinders
    { get { return _cylinders; } }

    public override string ToString()
    {
        return ToString("G");
    }

    public string ToString(string fmt)
    {
        if (string.IsNullOrEmpty(fmt))
            fmt = "G";

        switch (fmt.ToUpperInvariant())
        {
            case "G":
                return string.Format("{0} {1}", _year, _model);
            case "D":
                return string.Format("{0} {1}, {2} dr.",
                                     _year, _model, _doors);
            case "C":
                return string.Format("{0} {1}, {2}",
                                     _year, _model, _cylinders);
            case "A":
                return string.Format("{0} {1}, {2} dr. {3}",
                                     _year, _model, _doors, _cylinders);
            default:
                string msg = string.Format("'{0}' is an invalid format string",
                                           fmt);
                throw new ArgumentException(msg);
        }
    }
}

public class Example7
{
    public static void Main()
    {
        var auto = new Automobile("Lynx", 2016, 4, "V8");
```

```
      Console.WriteLine(auto.ToString());
      Console.WriteLine(auto.ToString("A"));
   }
}
// The example displays the following output:
//       2016 Lynx
//       2016 Lynx, 4 dr. V8
```

다음 예제에서는 오버로드된 Decimal.ToString(String, IFormatProvider) 메서드를 호출하여 통화 값의 문화권 구분 서식을 표시합니다.

C#

```
using System;
using System.Globalization;

public class Example8
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "en-GB", "fr-FR",
                                "hr-HR", "ja-JP" };
      Decimal value = 1603.49m;
      foreach (var cultureName in cultureNames) {
         CultureInfo culture = new CultureInfo(cultureName);
         Console.WriteLine($"{culture.Name}: {value.ToString("C2", culture)}");
      }
   }
}
// The example displays the following output:
//       en-US: $1,603.49
//       en-GB: £1,603.49
//       fr-FR: 1 603,49 €
//       hr-HR: 1.603,49 kn
//       ja-JP: ¥1,603.49
```

서식 문자열 및 문화권 구분 서식에 대한 자세한 내용은 형식 서식 지정을 참조하세요. 숫자 값에서 지원하는 형식 문자열은 표준 숫자 형식 문자열 및 사용자 지정 숫자 형식 문자열을 참조하세요. 날짜 및 시간 값에서 지원하는 형식 문자열은 표준 날짜 및 시간 형식 문자열과사용자 지정 날짜 및 시간 형식 문자열을 참조하세요.

Object.ToString 메서드 확장

형식은 기본 Object.ToString 메서드를 상속하므로 해당 동작이 바람직하지 않고 변경할 수 있습니다. 배열 및 컬렉션 클래스의 경우 특히 그렇습니다. 배열 또는 컬렉션 클래스의 `ToString` 메서드는 멤버의 값을 표시할 것으로 예상할 수 있지만, 대신 다음 예제에서 보듯이 완전히 정규화된 형식 이름을 표시합니다.

C#

```
int[] values = { 1, 2, 4, 8, 16, 32, 64, 128 };
Console.WriteLine(values.ToString());

List<int> list = new List<int>(values);
Console.WriteLine(list.ToString());

// The example displays the following output:
//       System.Int32[]
//       System.Collections.Generic.List`1[System.Int32]
```

원하는 결과 문자열을 생성하는 몇 가지 옵션이 있습니다.

- 형식이 배열, 컬렉션 개체 또는 IEnumerable 및 IEnumerable<T> 인터페이스를 구현하는 개체인 경우 C#의 `foreach` 문 또는 Visual Basic의 `For Each...Next` 구문을 사용하여 해당 요소를 열거할 수 있습니다.

- 클래스가 `sealed` (C#) 또는 `NotInheritable` (Visual Basic) 형식이 아닌 경우, 메서드를 사용자 정의하려는 기본 클래스를 상속하는 Object.ToString 래퍼 클래스를 개발할 수 있습니다. 최소한 다음을 수행해야 합니다.

  1. 필요한 생성자를 구현합니다. 파생 클래스는 기본 클래스 생성자를 상속하지 않습니다.
  2. 원하는 결과 문자열을 Object.ToString 반환하도록 메서드를 재정의합니다.

  다음 예제에서는 List<T> 클래스를 위한 래퍼 클래스를 정의합니다. 컬렉션의 각 메서드 값을 표시하기 위해 완전한 형식 이름 대신 Object.ToString 메서드를 재정의합니다.

  C#

  ```
  using System;
  using System.Collections.Generic;

  public class CList<T> : List<T>
  {
     public CList(IEnumerable<T> collection) : base(collection)
     { }

     public CList() : base()
     {}

     public override string ToString()
     {
        string retVal = string.Empty;
        foreach (T item in this) {
           if (string.IsNullOrEmpty(retVal))
              retVal += item.ToString();
           else
  ```

```
            retVal += string.Format(", {0}", item);
        }
        return retVal;
    }
}

public class Example2
{
    public static void Main()
    {
        var list2 = new CList<int>();
        list2.Add(1000);
        list2.Add(2000);
        Console.WriteLine(list2.ToString());
    }
}
// The example displays the following output:
//    1000, 2000
```

- 원하는 결과 문자열을 반환하는 확장 메서드 를 개발합니다. 이러한 방식으로 기본 Object.ToString 메서드를 재정의할 수 없습니다. 즉, 확장 클래스(C#) 또는 모듈(Visual Basic의 경우)에는 원래 형식의 `ToString` 메서드 대신 호출되는 매개 변수가 없는 메서드 `ToString` 를 사용할 수 없습니다. 매개 변수 없는 `ToString` 대체를 위해 다른 이름을 제공해야 합니다.

  다음 예제에서는 클래스를 확장하는 List<T> 두 가지 메서드, 즉 매개 변수가 없는 `ToString2` 메서드와 `ToString` 형식 문자열을 String 나타내는 매개 변수가 있는 메서드를 정의합니다.

C#

```
using System;
using System.Collections.Generic;

public static class StringExtensions
{
    public static string ToString2<T>(this List<T> l)
    {
        string retVal = string.Empty;
        foreach (T item in l)
            retVal += string.Format("{0}{1}", string.IsNullOrEmpty(retVal) ?
                                                "" : ", ",
                                    item);
        return string.IsNullOrEmpty(retVal) ? "{}" : "{ " + retVal + " }";
    }

    public static string ToString<T>(this List<T> l, string fmt)
    {
        string retVal = string.Empty;
        foreach (T item in l) {
            IFormattable ifmt = item as IFormattable;
```

```
            if (ifmt != null)
               retVal += string.Format("{0}{1}",
                                       string.IsNullOrEmpty(retVal) ?
                                          "" : ", ", ifmt.ToString(fmt, null));
            else
               retVal += ToString2(l);
         }
         return string.IsNullOrEmpty(retVal) ? "{}" : "{ " + retVal + " }";
      }
   }

   public class Example3
   {
      public static void Main()
      {
         List<int> list = new List<int>();
         list.Add(1000);
         list.Add(2000);
         Console.WriteLine(list.ToString2());
         Console.WriteLine(list.ToString("N0"));
      }
   }
   // The example displays the following output:
   //       { 1000, 2000 }
   //       { 1,000, 2,000 }
```

Windows 런타임에 대한 참고 사항

Windows 런타임의 클래스에서 ToString 메서드를 호출할 때, ToString를 재정의하지 않는 클래스에 대한 기본 동작을 제공합니다. 이는 .NET이 Windows 런타임에 대해 제공하는 지원의 일부입니다(Windows 스토어 앱 및 Windows 런타임에 대한 .NET 지원 참조). Windows 런타임의 클래스는 Object을(를) 상속하지 않으며, 항상 ToString을(를) 구현하는 것도 아닙니다. 그러나 C# 또는 Visual Basic 코드에서 사용할 때 항상 해당 메서드와 GetHashCode 메서드가 있는 것처럼 보이며 ToStringEquals(Object).NET은 이러한 메서드에 대한 기본 동작을 제공합니다.

공용 언어 런타임은 Windows 런타임 개체에서 IStringable.ToString을 사용한 후에 Object.ToString의 기본 구현으로 되돌아갑니다.

> ⓘ **참고**
>
> C# 또는 Visual Basic으로 작성된 Windows 런타임 클래스는 ToString 메서드를 재정의할 수 있습니다.

Windows 런타임 및 IStringable 인터페이스

Windows 런타임에는 단일 메서드인 IStringable.ToString이 제공하는 것과 비슷한 기본 서식 지원을 제공하는 IStringable 인터페이스가 Object.ToString포함되어 있습니다. 모호성을 방지하기 위해 관리되는 형식에서 IStringable을 구현해서는 안 됩니다.

관리되는 개체가 네이티브 코드 또는 JavaScript 또는 C++/CX와 같은 언어로 작성된 코드로 호출되면 IStringable을 구현하는 것처럼 보입니다. 공용 언어 런타임은 IStringable이 관리 개체에 구현되지 않은 경우 IStringable.ToString에서 Object.ToString로 호출을 자동으로 라우팅합니다.

> **⚠ 경고**
>
> 공용 언어 런타임은 Windows 스토어 앱의 모든 관리되는 형식에 대해 **IStringable** 을 자동으로 구현하므로 고유한 `IStringable` 구현을 제공하지 않는 것이 좋습니다. 구현하면 `IStringable` Windows 런타임, C++/CX 또는 JavaScript에서 호출 `ToString` 할 때 의도하지 않은 동작이 발생할 수 있습니다.

Windows 런타임 구성 요소에서 내보낸 공용 관리형 형식으로 IStringable 을 구현하도록 선택하는 경우 다음 제한 사항이 적용됩니다.

- 다음과 같이 "클래스 구현" 관계에서만 IStringable 인터페이스를 정의할 수 있습니다.

  C#

  ```
  public class NewClass : IStringable
  ```

- 인터페이스에서 IStringable 을 구현할 수 없습니다.
- 매개 변수를 IStringable 형식으로 선언할 수 없습니다.
- IStringable 은 메서드, 속성 또는 필드의 반환 형식일 수 없습니다.
- 다음과 같은 메서드 정의를 사용하여 기본 클래스에서 IStringable 구현을 숨길 수 없습니다.

  C#

  ```
  public class NewClass : IStringable
  {
     public new string ToString()
     {
        return "New ToString in NewClass";
     }
  }
  ```

대신 IStringable.ToString 구현은 항상 기본 클래스 구현을 재정의해야 합니다. 강력한 형식의 `ToString` 클래스 인스턴스에서 구현을 호출해야만 구현을 숨길 수 있습니다.

다양한 조건에서 네이티브 코드에서 관리되는 형식으로 호출하면, 해당 형식이 IStringable을 구현하거나 ToString 구현을 숨길 경우, 예기치 않은 동작이 발생할 수 있습니다.

System.Nullable 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스는 Nullable 할당 `null`할 수 있는 값 형식을 지원합니다.

형식은 값을 할당하거나 `null`로 할당할 수 있는 경우 null을 허용한다고 합니다. 즉, `null`이란 형식에 값이 전혀 없음을 의미합니다. 기본적으로 String과 같은 모든 참조 형식은 null을 허용하지만, Int32과 같은 모든 값 형식은 허용하지 않습니다.

C# 및 Visual Basic에서는 값 형식 뒤의 표기법을 사용하여 값 형식을 `?` nullable로 표시합니다. 예를 들어 `int?` C# 또는 `Integer?` Visual Basic에서는 할당 `null`할 수 있는 정수 값 형식을 선언합니다.

클래스는 Nullable 구조에 대한 Nullable<T> 보완적인 지원을 제공합니다. 클래스는 Nullable nullable 형식의 기본 형식을 가져오고 기본 값 형식이 제네릭 비교 및 같음 연산을 지원하지 않는 nullable 형식 쌍에 대한 비교 및 같음 연산을 지원합니다.

박싱 및 언박싱

nullable 형식이 박싱될 때, 공용 언어 런타임은 Nullable<T> 개체 자체가 아닌 개체의 기본 값을 자동으로 박싱합니다. 즉, HasValue 속성이 `true`이면 Value 속성의 내용이 "박스화"됩니다.

`HasValue` nullable 형식의 속성이`false`면 boxing 작업의 결과는 다음과 입니다`null`. nullable 형식의 기본 값이 언박싱되면 공용 언어 런타임은 기본 값을 생성하여 초기화된 새로운 Nullable<T> 구조를 만듭니다.

Last updated on 2026. 02. 12.

.NET의 제네릭

아티클 • 2025. 04. 03.

제네릭을 사용하면 메서드, 클래스, 구조 또는 인터페이스를 정확한 데이터 형식에 맞게 조정할 수 있습니다. 예를 들어 키와 값이 Hashtable 모든 형식일 수 있도록 하는 클래스를 사용하는 대신 제네릭 클래스를 Dictionary<TKey,TValue> 사용하고 키와 값에 허용되는 형식을 지정할 수 있습니다. 제네릭의 이점 중에는 향상된 코드 재사용성 및 형식 안전성이 있습니다.

제네릭 정의 및 사용

제네릭은 저장하거나 사용하는 하나 이상의 형식에 대한 자리 표시자(형식 매개 변수)가 있는 클래스, 구조체, 인터페이스 및 메서드입니다. 제네릭 컬렉션 클래스는 형식 매개 변수를 저장하는 개체의 형식에 대한 자리 표시자로 사용할 수 있습니다. 형식 매개 변수는 해당 필드의 형식 및 해당 메서드의 매개 변수 형식으로 나타납니다. 제네릭 메서드는 해당 형식 매개 변수를 반환 값의 형식 또는 형식 매개 변수 중 하나의 형식으로 사용할 수 있습니다.

다음 코드에서는 간단한 제네릭 클래스 정의를 보여 줍니다.

C#

```
public class SimpleGenericClass<T>
{
    public T Field;
}
```

제네릭 클래스의 인스턴스를 만들 때 형식 매개 변수를 대체할 실제 형식을 지정합니다. 이렇게 하면 생성된 제네릭 클래스라고 하는 새 제네릭 클래스가 설정되며, 선택한 형식은 형식 매개 변수가 표시되는 모든 곳에서 대체됩니다. 결과는 다음 코드와 같이 선택한 형식에 맞게 조정된 형식 안전 클래스입니다.

C#

```
public static void Main()
{
    SimpleGenericClass<string> g = new SimpleGenericClass<string>();
    g.Field = "A string";
    //...
    Console.WriteLine($"SimpleGenericClass.Field          = \"
{g.Field}\"");
    Console.WriteLine($"SimpleGenericClass.Field.GetType() =
```

```
{g.Field.GetType().FullName}");
}
```

용어

다음 용어는 .NET에서 제네릭을 설명하는 데 사용됩니다.

- *제네릭 형식 정의*는 템플릿으로 작동하며 포함하거나 사용할 수 있는 형식에 대한 자리 표시자를 포함하는 클래스, 구조 또는 인터페이스 선언입니다. 예를 들어 클래스에는 System.Collections.Generic.Dictionary<TKey,TValue> 키와 값의 두 가지 형식이 포함될 수 있습니다. 제네릭 형식 정의는 템플릿일 뿐이므로 제네릭 형식 정의인 클래스, 구조체 또는 인터페이스의 인스턴스를 만들 수 없습니다.

- *제네릭 형식 매개 변수* 또는 *형식 매개 변수*는 제네릭 형식 또는 메서드 정의의 자리 표시자입니다. System.Collections.Generic.Dictionary<TKey,TValue> 제네릭 형식에는 키와 값의 형식을 나타내는 두 개의 형식 매개 변수 `TKey` 가 있습니다`TValue`.

- *생성된 제네릭 형식* 또는 *생성된 형식*은 제네릭 형식 정의의 제네릭 형식 매개 변수에 대한 형식을 지정한 결과입니다.

- *제네릭 형식 인수*는 제네릭 형식 매개 변수로 대체되는 모든 형식입니다.

- 일반 용어 *제네릭 형식* 에는 생성된 형식과 제네릭 형식 정의가 모두 포함됩니다.

- 제네릭 형식 매개 변수의 *공변성* 및 *반공변성*으로 인해 생성된 제네릭 형식을 사용할 수 있습니다. 이 제네릭 형식의 형식 인수는 대상 생성 형식보다 파생(공변성) 또는 파생(반공변성)이 적습니다. 공변성 및 반공변성(contravariance)을 통칭하여 *분산*이라고 합니다. 자세한 내용은 공변성 및 반공변성(contravariance)을 참조하세요.

- *제약 조건은* 제네릭 형식 매개 변수에 대한 제한입니다. 예를 들어 형식의 인스턴스를 정렬할 수 있도록 형식 매개 변수를 제네릭 인터페이스를 구현 System.Collections.Generic.IComparer<T> 하는 형식으로 제한할 수 있습니다. 특정 기본 클래스가 있거나, 매개 변수가 없는 생성자가 있거나, 참조 형식 또는 값 형식인 형식으로 형식 매개 변수를 제한할 수도 있습니다. 제네릭 형식의 사용자는 제약 조건을 충족하지 않는 형식 인수를 대체할 수 없습니다.

- *제네릭 메서드 정의*는 제네릭 형식 매개 변수 목록과 정식 매개 변수 목록이라는 두 개의 매개 변수 목록이 있는 메서드입니다. 형식 매개 변수는 다음 코드와 같이 반환 형식 또는 정식 매개 변수의 형식으로 표시할 수 있습니다.

  ```
  C#

  T MyGenericMethod<T>(T arg)
  {
  ```

```
    T temp = arg;
    //...
    return temp;
}
```

제네릭 메서드는 제네릭 또는 비제네릭 형식에 나타날 수 있습니다. 메서드가 제네릭 형식에 속해 있거나 바깥쪽 형식의 제네릭 매개 변수와 같은 형식을 가진 정식 매개 변수를 포함하고 있다고 해서 반드시 제네릭인 것은 아니라는 점을 유의해야 합니다. 메서드는 고유한 형식 매개 변수 목록이 있는 경우에만 제네릭입니다. 다음 코드에서는 메서드 `G` 만 제네릭입니다.

C#

```
class A
{
    T G<T>(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
class MyGenericClass<T>
{
    T M(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
```

제네릭의 장점 및 단점

제네릭 컬렉션 및 대리자를 사용할 때는 다음과 같은 여러 가지 이점이 있습니다.

- 형식 안전성. 제네릭은 형식 안전의 부담을 컴파일러로 옮깁니다. 컴파일 시간에 적용되므로 올바른 데이터 형식을 테스트하기 위해 코드를 작성할 필요가 없습니다. 형식 캐스팅의 필요성과 런타임 오류 가능성이 줄어듭니다.

- 코드가 적을수록 더 쉽게 재사용할 수 있습니다. 기본 형식에서 상속하고 멤버를 재정의할 필요가 없습니다. 예를 들어 즉시 LinkedList<T> 사용할 준비가 된 것입니다. 예를 들어 다음 변수 선언을 사용하여 문자열의 연결된 목록을 만들 수 있습니다.

C#

```
LinkedList<string> llist = new LinkedList<string>();
```

- 성능 향상. 일반적으로 제네릭 컬렉션 형식은 값 형식을 상자에 입력할 필요가 없으므로 값 형식을 저장하고 조작하는 데 더 적합합니다.
- 제네릭 대리자는 여러 대리자 클래스를 만들 필요 없이 형식이 안전한 콜백을 사용하도록 설정합니다. 예를 들어, Predicate<T> 제네릭 대리자를 사용하면 특정 형식에 대한 사용자의 검색 조건을 구현하는 메서드를 만들 수 있으며, Array, Find, FindLast와 같은 FindAll 형식의 메서드와 함께 그 메서드를 사용할 수 있습니다.
- 제네릭은 동적으로 생성된 코드를 간소화합니다. 동적으로 생성된 코드와 함께 제네릭을 사용하는 경우 형식을 생성할 필요가 없습니다. 이렇게 하면 전체 어셈블리를 생성하는 대신 간단한 동적 메서드를 사용할 수 있는 시나리오 수가 증가합니다. 자세한 내용은 방법: 동적 메서드 정의 및 실행 및 DynamicMethod.을 참조하세요.

다음은 제네릭의 몇 가지 제한 사항입니다.

- 제네릭 형식은 대부분의 기본 클래스 MarshalByRefObject 에서 파생될 수 있습니다(및 제약 조건은 제네릭 형식 매개 변수가 같은 MarshalByRefObject기본 클래스에서 파생되도록 요구하는 데 사용할 수 있습니다). 그러나 .NET은 컨텍스트 바인딩된 제네릭 형식을 지원하지 않습니다. 제네릭 형식은 ContextBoundObject에서 파생될 수 있지만, 해당 형식의 인스턴스를 만들려고 하면 TypeLoadException가 발생합니다.
- 열거형에는 제네릭 형식 매개 변수가 있을 수 없습니다. 열거형은 부수적으로만 제네릭일 수 있습니다(예: Visual Basic, C#또는 C++를 사용하여 정의된 제네릭 형식에 중첩되어 있기 때문). 자세한 내용은 공용 형식 시스템의 "열거형"을 참조하세요.
- 경량 동적 메서드는 제네릭일 수 없습니다.
- Visual Basic, C# 및 C++에서는 모든 바깥쪽 형식의 형식 매개 변수에 형식이 할당되지 않은 한 제네릭 형식으로 묶인 중첩 형식을 인스턴스화할 수 없습니다. 이를 말하는 또 다른 방법은 리플렉션에서 이러한 언어를 사용하여 정의된 중첩된 형식에 모든 바깥쪽 형식의 형식 매개 변수가 포함된다는 것입니다. 바깥쪽 형식의 형식 매개 변수를 중첩된 형식의 멤버 정의에서 사용할 수 있게 합니다. 자세한 내용은 MakeGenericType의 "중첩 형식"을 참조하세요.

> **참고**
>
> 동적 어셈블리에서 코드를 내보내거나 Ilasm.exe(IL 어셈블러) 를 사용하여 정의된 중첩 형식은 바깥쪽 형식의 형식 매개 변수를 포함할 필요가 없습니다. 그

러나 형식 매개 변수를 포함하지 않는 경우 형식 매개 변수는 중첩 클래스의 범위에 없습니다.

자세한 내용은 MakeGenericType의 "중첩 형식"을 참조하세요.

클래스 라이브러리 및 언어 지원

.NET은 다음 네임스페이스에 여러 제네릭 컬렉션 클래스를 제공합니다.

- 네임스페이스에는 System.Collections.Generic .NET에서 제공하는 대부분의 제네릭 컬렉션 형식(예: List<T>Dictionary<TKey,TValue> 제네릭 클래스)이 포함됩니다.
- 네임스페이스에는 System.Collections.ObjectModel 개체 모델을 클래스 사용자에게 노출하는 데 유용한 제네릭 클래스와 같은 ReadOnlyCollection<T> 추가 제네릭 컬렉션 형식이 포함되어 있습니다.

정렬 및 같음 비교를 구현하기 위한 제네릭 인터페이스는 이벤트 처리기, 변환 및 검색 조건자의 제네릭 대리자 형식과 함께 네임스페이스에 제공됩니다 System .

네임스페이 System.Numerics 스는 수학 기능을 위한 제네릭 인터페이스를 제공합니다(.NET 7 이상 버전에서 사용 가능). 자세한 내용은 제네릭 수학을 참조하세요.

제네릭 형식과 제네릭 메서드를 검사하기 위한 System.Reflection 네임스페이스, 제네릭 형식과 메서드를 포함하는 동적 어셈블리를 내보내기 위한 System.Reflection.Emit, 그리고 제네릭을 포함하는 원본 그래프를 생성하기 위한 System.CodeDom에 대한 지원이 추가되었습니다.

공용 언어 런타임은 제네릭 형식(CIL, 공용 중간 언어)을 지원하기 위해 Stelem, Ldelem, Unbox_Any, Constrained, 및 Readonly과 같은 새로운 opcode 및 접두사를 제공합니다.

Visual C++, C# 및 Visual Basic은 모두 제네릭을 정의하고 사용하기 위한 모든 지원을 제공합니다. 언어 지원에 대한 자세한 내용은 Visual Basic의 제네릭 형식, 제네릭 소개 및 Visual C++의 제네릭 개요를 참조하세요.

중첩 형식 및 제네릭

제네릭 형식에 중첩된 형식은 바깥쪽 제네릭 형식의 형식 매개 변수에 따라 달라질 수 있습니다. 공용 언어 런타임은 고유한 제네릭 형식 매개 변수가 없는 경우에도 중첩된 형식을 제네릭으로 간주합니다. 중첩 형식의 인스턴스를 만들 때는 모든 묶은 제네릭 형식에 대한 형식 인수를 지정해야 합니다.

관련 문서

테이블 확장

제목	설명
.NET의 제네릭 컬렉션	.NET의 제네릭 컬렉션 클래스 및 기타 제네릭 형식에 대해 설명합니다.
배열 및 목록을 조작하기 위한 제네릭 대리자	배열 또는 컬렉션의 요소에 대해 수행할 변환, 검색 조건자 및 작업에 대한 제네릭 대리자를 설명합니다.
제네릭 수학	수학 연산을 일반적으로 수행하는 방법을 설명합니다.
제네릭 인터페이스	제네릭 형식의 제품군에서 공통 기능을 제공하는 제네릭 인터페이스에 대해 설명합니다.
공변성 및 반공변성	제네릭 형식 매개 변수의 공변성 및 반공변에 대해 설명합니다.
일반적으로 사용되는 컬렉션 형식	제네릭 형식을 포함하여 .NET에서 컬렉션 형식의 특성 및 사용 시나리오에 대한 요약 정보를 제공합니다.
제네릭 컬렉션을 사용하는 경우	제네릭 컬렉션 형식을 사용하는 시기를 결정하는 일반적인 규칙에 대해 설명합니다.
방법: 리플렉션 내보내기를 사용하여 제네릭 형식 정의	제네릭 형식 및 메서드를 포함하는 동적 어셈블리를 생성하는 방법을 설명합니다.
Visual Basic의 제네릭 형식	제네릭 형식을 사용하고 정의하는 방법 항목을 포함하여 Visual Basic 사용자의 제네릭 기능에 대해 설명합니다.
제네릭 소개	C# 사용자에 대한 제네릭 형식 정의 및 사용에 대한 개요를 제공합니다.
Visual C++의 제네릭 개요	제네릭과 템플릿 간의 차이점을 포함하여 C++ 사용자의 제네릭 기능에 대해 설명합니다.

참고 문헌

- System.Collections.Generic
- System.Collections.ObjectModel
- System.Reflection.Emit.OpCodes

제네릭 형식 개요

2025. 06. 17.

개발자는 암시적이든 명시적으로든 .NET에서 항상 제네릭을 사용합니다. .NET에서 LINQ를 사용할 때 IEnumerable<T>을(를) 작업하고 있다는 것을 알아차린 적이 있었나요? 또는 Entity Framework를 사용하여 데이터베이스와 통신하기 위한 "제네릭 리포지토리"의 온라인 샘플을 본 적이 있다면 대부분의 메서드가 반환 `IQueryable<T>`되는 것을 보셨나요? 이러한 예제에서 **T**가 무엇이고 왜 거기에 있는지 궁금했을 것입니다.

.NET Framework 2.0에서 처음 도입된 제네릭은 기본적으로 개발자가 실제 데이터 형식에 커밋하지 않고 형식이 안전한 데이터 구조를 정의할 수 있는 "코드 템플릿"입니다. 예를 들어, List<T>는 , `List<int>`, 또는 `List<string>` 같은 모든 형식과 함께 선언하고 사용할 수 있는 `List<Person>`입니다.

제네릭이 유용한 이유를 이해하려면 제네릭 ArrayList을 추가하기 전과 후에 특정 클래스를 살펴보겠습니다. .NET Framework 1.0에서는 `ArrayList` 요소가 형식 Object이었습니다. 컬렉션에 추가된 모든 요소가 자동으로 .로 `Object`변환되었습니다. 목록에서 요소를 읽을 때도 마찬가지입니다. 이 과정은 박싱 및 언박싱으로 알려져 있으며, 성능에 영향을 미칩니다. 그러나 성능 외에도 컴파일 시간에 목록의 데이터 형식을 확인할 수 있는 방법이 없으므로 일부 취약한 코드가 생성됩니다. 제네릭은 목록의 각 인스턴스가 포함할 데이터 형식을 정의하여 이 문제를 해결합니다. 오직 `List<int>`에는 정수만 추가할 수 있고 `List<Person>`에는 오직 Persons만 추가할 수 있습니다.

제네릭은 런타임에도 사용할 수 있습니다. 런타임은 사용 중인 데이터 구조의 유형을 알고 메모리에 보다 효율적으로 저장할 수 있습니다.

다음 예제는 런타임에 데이터 구조 형식을 아는 효율성을 보여 주는 작은 프로그램입니다.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

namespace GenericsExample {
  class Program {
    static void Main(string[] args) {
      //generic list
      List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };
      //non-generic list
      ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
      // timer for generic list sort
      Stopwatch s = Stopwatch.StartNew();
      ListGeneric.Sort();
```

```
            s.Stop();
            Console.WriteLine($"Generic Sort: {ListGeneric}  \n Time taken:
{s.Elapsed.TotalMilliseconds}ms");

            //timer for non-generic list sort
            Stopwatch s2 = Stopwatch.StartNew();
            ListNonGeneric.Sort();
            s2.Stop();
            Console.WriteLine($"Non-Generic Sort: {ListNonGeneric}  \n Time taken:
{s2.Elapsed.TotalMilliseconds}ms");
            Console.ReadLine();
        }
    }
  }
```

이 프로그램은 다음과 유사한 출력을 생성합니다.

콘솔

```
Generic Sort: System.Collections.Generic.List`1[System.Int32]
 Time taken: 0.0034ms
Non-Generic Sort: System.Collections.ArrayList
 Time taken: 0.2592ms
```

여기에서 가장 먼저 알 수 있는 것은 제네릭 목록을 정렬하는 것이 제네릭이 아닌 목록을 정렬하는 것보다 훨씬 빠르다는 것입니다. 제네릭 목록의 형식은 고유하지만([System.Int32]), 제네릭이 아닌 목록의 형식은 일반화됩니다. 런타임은 제네릭 `List<int>` 이 형식 Int32임을 알고 있으므로 메모리의 기본 정수 배열에 목록 요소를 저장할 수 있지만, 제네릭 `ArrayList` 이 아닌 사용자는 각 목록 요소를 개체로 캐스팅해야 합니다. 이 예제에서와 같이 추가 캐스트는 시간이 걸리고 목록 정렬 속도가 느려집니다.

제네릭 유형을 아는 런타임의 추가적인 이점은 더 나은 디버깅 환경입니다. C#에서 제네릭을 디버깅할 때 각 요소가 데이터 구조에 있는 형식을 알 수 있습니다. 제네릭이 없으면 각 요소의 형식을 알 수 없습니다.

참고하십시오

- C# 프로그래밍 가이드 - 제네릭

.NET의 제네릭 컬렉션

아티클 • 2025. 05. 01.

.NET 클래스 라이브러리는 System.Collections.Generic 및 System.Collections.ObjectModel 네임스페이스에 여러 제네릭 컬렉션 클래스를 제공합니다. 이러한 클래스에 대한 자세한 내용은 일반적으로 사용되는 컬렉션 형식을 참조하세요.

System.Collections.Generic

제네릭 컬렉션 형식의 대부분은 비제네릭 형식의 직접적인 유사체입니다. Dictionary<TKey,TValue>는 Hashtable의 제네릭 버전입니다. 열거형 DictionaryEntry 대신 제네릭 구조 KeyValuePair<TKey,TValue>를 사용합니다.

List<T>는 ArrayList의 제네릭 버전입니다. 제네릭이 아닌 버전에 해당하는 제네릭 Queue<T> 및 Stack<T> 클래스가 있습니다.

SortedList<TKey,TValue>의 제네릭 버전과 비제네릭 버전이 있습니다. 두 버전 모두 사전과 목록의 하이브리드입니다. SortedDictionary<TKey,TValue> 제네릭 클래스는 순수 사전이며 제네릭이 아닌 사전이 없습니다.

LinkedList<T> 제네릭 클래스는 실제 연결된 목록입니다. 비제네릭 대응 항목이 없습니다.

System.Collections.ObjectModel

제네릭 클래스는 Collection<T> 고유한 제네릭 컬렉션 형식을 파생하기 위한 기본 클래스를 제공합니다. 이 클래스는 ReadOnlyCollection<T> 제네릭 인터페이스를 구현 IList<T> 하는 모든 형식에서 읽기 전용 컬렉션을 생성하는 쉬운 방법을 제공합니다. 제네릭 클래스는 KeyedCollection<TKey,TItem> 자체 키를 포함하는 개체를 저장하는 방법을 제공합니다.

기타 제네릭 형식

Nullable<T> 제네릭 구조를 사용하면 값 형식을 할당`null`할 수 있는 것처럼 사용할 수 있습니다. 이 기능은 값 형식이 포함된 필드가 누락될 수 있는 데이터베이스 쿼리를 사용할 때 유용할 수 있습니다. 제네릭 형식 매개 변수는 모든 값 형식일 수 있습니다.

> **참고**
>
> C# 및 Visual Basic에서는 언어에 nullable 형식에 대한 구문이 있으므로 명시적으로 사용할 Nullable<T> 필요가 없습니다. Nullable 값 형식(C# 참조) 및 Nullable 값 형식(Visual Basic)을 참조하세요.

제네릭 구조는 ArraySegment<T> 모든 형식의 1차원 0부터 시작하는 배열 내에서 요소 범위를 구분하는 방법을 제공합니다. 제네릭 형식 매개 변수는 배열 요소의 형식입니다.

EventHandler<TEventArgs> 제네릭 대리자는 이벤트가 .NET에서 사용하는 이벤트 처리 패턴을 따르는 경우 이벤트를 처리하기 위해 대리자 형식을 선언할 필요가 없습니다. 예를 들어, 이벤트에 대한 데이터를 보관하기 위해 EventArgs에서 파생된 `MyEventArgs` 클래스를 만들었다고 가정해 봅시다. 그런 다음 다음과 같이 이벤트를 선언할 수 있습니다.

C#

```
public event EventHandler<MyEventArgs> MyEvent;
```

참고하십시오

- System.Collections.Generic
- System.Collections.ObjectModel
- 제네릭
- 배열 및 목록을 조작하기 위한 제네릭 대리자
- 제네릭 인터페이스

배열 및 목록을 다루기 위한 일반 대리자

2025. 06. 22.

이 항목에서는 배열 또는 컬렉션의 요소에 대해 수행할 변환, 검색 조건자 및 작업에 대한 일반 대리자의 개요를 제공합니다.

배열 및 목록을 다루기 위한 일반 대리자

Action<T> 제네릭 대리자는 지정된 형식의 요소에 대해 일부 작업을 수행하는 메서드를 나타냅니다. 요소에 대해 원하는 작업을 수행하는 메서드를 만들고, 해당 메서드를 나타내는 대리자의 Action<T> 인스턴스를 만든 다음, 배열과 대리자를 정적 제네릭 메서드에 Array.ForEach 전달할 수 있습니다. 배열의 모든 요소에 대해 메서드가 호출됩니다.

제네릭 클래스는 List<T> 대리자를 ForEach 사용하는 Action<T> 메서드도 제공합니다. 이 메서드는 제네릭이 아닙니다.

> ⓘ **참고**
>
> 이렇게 하면 제네릭 형식 및 메서드에 대한 흥미로운 점이 있습니다. 메서드는 Array.ForEach가 제네릭 형식이 아니기 때문에 정적(`Shared` Visual Basic)이고 제네릭이어야 합니다. Array에서 작동할 형식을 지정할 수 있는 유일한 이유는 메서드 자체에 고유한 형식 매개변수 목록이 있기 때문입니다. 반면, 제네릭 List<T>.ForEach 이 아닌 메서드는 제네릭 클래스 List<T>에 속하므로 클래스의 형식 매개 변수만 사용합니다. 클래스는 강력한 형식이므로 메서드는 인스턴스 메서드가 될 수 있습니다.

Predicate<T> 제네릭 대리자는 특정 요소가 정의한 조건을 충족하는지 여부를 결정하는 메서드를 나타냅니다. Array의 다음 정적 제네릭 메서드와 함께 사용하여 요소 또는 요소 집합을 검색할 수 있습니다: Exists, Find, FindAll, FindIndex, FindLast, FindLastIndex, 및 TrueForAll.

Predicate<T> 또한 제네릭 클래스의 해당 비제네릭 인스턴스 메서드와 List<T> 함께 작동합니다.

Comparison<T> 제네릭 대리자를 사용하면 네이티브 정렬 순서가 없는 배열 또는 목록 요소에 대한 정렬 순서를 제공하거나 네이티브 정렬 순서를 재정의할 수 있습니다. 비교를 수행하는 메서드를 만들고, 메서드를 나타내는 대리자의 Comparison<T> 인스턴스를 만든 다음, 배열과 대리자를 정적 제네릭 메서드에 Array.Sort<T>(T[], Comparison<T>) 전달합니다. 제네릭 클래스는 List<T> 해당 인스턴스 메서드 오버로드를 List<T>.Sort(Comparison<T>)제공합니다.

Converter<TInput,TOutput> 제네릭 대리자를 사용하면 두 형식 간의 변환을 정의하고 한 형식의 배열을 다른 형식의 배열로 변환하거나 한 형식의 목록을 다른 형식 목록으로 변환할 수 있습니다. 기존 목록의 요소를 새 형식으로 변환하는 메서드를 만들고, 메서드를 나타내는 대리자 인

스턴스를 만들고, 제네릭 정적 메서드를 사용하여 Array.ConvertAll 원래 배열에서 새 형식의 배열을 생성하거나 List<T>.ConvertAll<TOutput>(Converter<T,TOutput>) , 원래 목록에서 새 형식의 목록을 생성하는 제네릭 인스턴스 메서드를 만듭니다.

대리자 연결

이러한 대리자를 사용하는 대부분의 메서드는 배열 또는 목록을 반환하며 다른 메서드에 전달할 수 있습니다. 예를 들어 배열의 특정 요소를 선택하고, 해당 요소를 새 형식으로 변환하고, 새 배열에 저장하려는 경우 제네릭 메서드에서 반환된 FindAll 배열을 제네릭 메서드에 ConvertAll 전달할 수 있습니다. 새 요소 형식에 자연 정렬 순서가 없는 경우 제네릭 메서드에서 반환된 배열을 ConvertAll 제네릭 메서드에 Sort<T>(T[], Comparison<T>) 전달할 수 있습니다.

참고하십시오

- System.Collections.Generic
- System.Collections.ObjectModel
- 제네릭
- .NET의 제네릭 컬렉션
- 제네릭 인터페이스
- 공변성 및 반공변성

제네릭 수학

2025. 06. 17.

.NET 7은 기본 클래스 라이브러리에 새로운 수학 관련 제네릭 인터페이스를 도입했습니다. 이러한 인터페이스의 가용성은 제네릭 형식 또는 메서드의 형식 매개 변수를 "숫자와 유사한"으로 제한할 수 있습니다. 또한 C# 11 이상에서는 인터페이스 멤버를 정의static virtual할 수 있습니다. 연산자를 로 `static`선언해야 하므로 이 새로운 C# 기능을 사용하면 숫자와 유사한 형식에 대한 새 인터페이스에서 연산자를 선언할 수 있습니다.

이러한 혁신을 통해 일반적으로 수학 연산을 수행할 수 있습니다. 즉, 작업 중인 정확한 형식을 알 필요 없이 수행할 수 있습니다. 예를 들어 두 개의 숫자를 추가하는 메서드를 작성하려는 경우 이전에는 각 형식에 대한 메서드 오버로드(예: `static int Add(int first, int second)` 및 `static float Add(float first, float second)`)를 추가해야 했습니다. 이제 형식 매개 변수가 숫자와 유사한 형식으로 제한되는 단일 제네릭 메서드를 작성할 수 있습니다. 다음은 그 예입니다.

C#

```
static T Add<T>(T left, T right)
    where T : INumber<T>
{
    return left + right;
}
```

이 메서드에서 형식 매개 변수 `T` 는 새 INumber<TSelf> 인터페이스를 구현하는 형식으로 제한됩니다. INumber<TSelf> 는 IAdditionOperators<TSelf,TOther,TResult>+ 연산자를 포함하는 인터페이스를 구현합니다. 이를 통해 메서드는 일반적으로 두 숫자를 추가할 수 있습니다. 메서드는 모든 .NET의 기본 제공 숫자 형식과 함께 사용할 수 있으며, 이는 모두 .NET 7에서 INumber<TSelf>를 구현하도록 업데이트되었기 때문입니다.

라이브러리 작성자는 "중복" 오버로드를 제거하여 코드 베이스를 간소화할 수 있으므로 제네릭 수학 인터페이스의 이점을 가장 활용할 수 있습니다. 다른 개발자는 사용하는 API가 더 많은 형식을 지원하기 시작할 수 있으므로 간접적으로 이점을 얻을 수 있습니다.

인터페이스

인터페이스는 사용자가 자신의 인터페이스를 정의할 수 있을 만큼 정밀하게 설계되었으며, 또한 사용하기에 쉽게 이해할 수 있게끔 세분화되었습니다. 이 정도까지 대부분의 사용자가 상호 작용하는 몇 가지 핵심 숫자 인터페이스(예: INumber<TSelf> 및 IBinaryInteger<TSelf>.)가 있습니다. 보다 세분화된 IAdditionOperators<TSelf,TOther,TResult> 및 ITrigonometricFunctions<TSelf>와 같은 인터페이스는 이러한 유형을 지원하고, 고유의 도메인별 숫자 인터페이스를 정의하는 개발자가 사용할 수 있습니다.

- 숫자 인터페이스
- 연산자 인터페이스
- 함수 인터페이스
- 인터페이스 구문 분석 및 서식 지정

숫자 인터페이스

이 섹션에서는 숫자와 유사한 형식 및 사용할 수 있는 System.Numerics 기능을 설명하는 인터페이스에 대해 설명합니다.

테이블 확장

인터페이스 이름	설명
IBinaryFloatingPointIeee754<TSelf>	IEEE 754 표준을 구현하는 *이진* 부동 소수점 형식[1] 에 공통적인 API를 노출합니다.
IBinaryInteger<TSelf>	이진 정수[2]에 공통적인 API를 노출합니다.
IBinaryNumber<TSelf>	이진 번호에 공통적인 API를 노출합니다.
IFloatingPoint<TSelf>	부동 소수점 형식에 공통적인 API를 노출합니다.
IFloatingPointIeee754<TSelf>	IEEE 754 표준을 구현하는 부동 소수점 형식에 공통적인 API를 노출합니다.
INumber<TSelf>	비교 가능한 숫자 형식(사실상 "실제" 숫자 도메인)에 공통적인 API를 노출합니다.
INumberBase<TSelf>	모든 숫자 형식(사실상 "복합" 숫자 도메인)에 공통적인 API를 노출합니다.
ISignedNumber<TSelf>	모든 부호 있는 숫자 유형(예: `NegativeOne` 개념)에 공통적인 API를 제공합니다.
IUnsignedNumber<TSelf>	서명되지 않은 모든 숫자 형식에 공통적인 API를 노출합니다.
IAdditiveIdentity<TSelf,TResult>	`(x + T.AdditiveIdentity) == x` 개념을 설명합니다.
IMinMaxValue<TSelf>	`T.MinValue` 및 `T.MaxValue`의 개념을 드러냅니다.
IMultiplicativeIdentity<TSelf,TResult>	`(x * T.MultiplicativeIdentity) == x` 개념을 설명합니다.

[1]이진 부동 소수점 형식은 Double(`double`), Half, 및 Single(`float`)입니다.

[2]이진 정수 타입은 Byte (`byte`), Int16 (`short`), Int32 (`int`), Int64 (`long`), Int128, IntPtr (`nint`), SByte (`sbyte`), UInt16 (`ushort`), UInt32 (`uint`), UInt64 (`ulong`), UInt128, 및 UIntPtr (`nuint`)입니다.

직접 사용할 가능성이 가장 큰 인터페이스는 INumber<TSelf>에 해당하는 인터페이스입니다. 형식이 이 인터페이스를 구현하는 경우 값에 기호가 있고(`unsigned` 양수로 간주되는 형식 포함) 동일한 형식의 다른 값과 비교할 수 있음을 의미합니다. INumberBase<TSelf> 에서는 *복잡한* 숫자와 *허수* 등의 고급 개념(예: 음수의 제곱근)을 제공합니다. 모든 숫자 형식에 모든 연산이 적합한 것은 아니므로 같은 IFloatingPointIeee754<TSelf>다른 인터페이스가 생성되었습니다. 예를 들어 숫자의 바닥 계산은 부동 소수점 형식에만 적합합니다. .NET 기본 클래스 라이브러리에서 부동 소수점 형식 Double 은 IFloatingPointIeee754<TSelf> 구현하지만 Int32 구현하지 않습니다.

여러 인터페이스는 Char, DateOnly, DateTime, DateTimeOffset, Decimal, Guid, TimeOnly, 및 TimeSpan를 포함한 다양한 형식에 의해 구현됩니다.

다음 표에서는 각 인터페이스에서 노출하는 핵심 API 중 일부를 보여 줍니다.

테이블 확장

인터페이스	API 이름	설명
IBinaryInteger<TSelf>	`DivRem`	몫과 나머지를 동시에 계산합니다.
	`LeadingZeroCount`	이진 표현에서 선행 0비트 수를 계산합니다.
	`PopCount`	이진 표현의 집합 비트 수를 계산합니다.
	`RotateLeft`	비트를 왼쪽으로 회전하는데, 이는 원형 왼쪽 시프트라고도 합니다.
	`RotateRight`	비트를 오른쪽으로 회전시키며, 이를 원형 우측 시프트라고도 합니다.
	`TrailingZeroCount`	이진 표현의 후행 0비트 수를 계산합니다.
IFloatingPoint<TSelf>	`Ceiling`	값을 양수 무한대로 반올림합니다. +4.5는 +5가 되고 -4.5는 -4가 됩니다.
	`Floor`	값을 음의 무한대로 반올림합니다. +4.5는 +4가 되고 -4.5는 -5가 됩니다.
	`Round`	지정된 반올림 모드를 사용하여 값을 반올림합니다.
	`Truncate`	값을 0으로 반올림합니다. +4.5는 +4가 되고 -4.5는 -4가 됩니다.
IFloatingPointIeee754<TSelf>	`E`	형식에 대한 Euler의 번호를 나타내는 값을 가져옵니다.
	`Epsilon`	형식에 대해 0보다 큰 가장 작은 표현 가능 값을 가져옵니다.

인터페이스	API 이름	설명
	`NaN`	형식을 나타내는 `NaN` 값을 가져옵니다.
	`NegativeInfinity`	형식을 나타내는 `-Infinity` 값을 가져옵니다.
	`NegativeZero`	형식을 나타내는 `-Zero` 값을 가져옵니다.
	`Pi`	형식을 나타내는 `Pi` 값을 가져옵니다.
	`PositiveInfinity`	형식을 나타내는 `+Infinity` 값을 가져옵니다.
	`Tau`	형식을 나타내는 `Tau` 값(`2 * Pi`)을 가져옵니다.
	(기타)	(함수 인터페이스 아래에 나열된 전체 인터페이스 집합을 구현합니다.)
INumber<TSelf>	`Clamp`	값을 지정된 최소값 및 최대값보다 작지 않은 값으로 제한합니다.
	`CopySign`	지정된 값의 부호를 지정된 다른 값과 동일하게 설정합니다.
	`Max`	두 값 중 더 큰 값을 반환하고 `NaN` 입력이 두 값 중 하나일 경우 반환합니다 `NaN`.
	`MaxNumber`	두 값 중 더 큰 값을 반환하고 입력이 1 `NaN`개인 경우 숫자를 반환합니다.
	`Min`	두 값 중 더 작은 값을 반환하고 입력 중 하나가 `NaN` 이면 반환합니다 `NaN`.
	`MinNumber`	두 값 중 더 작은 값을 반환하고 입력 `NaN`이 1개인 경우 숫자를 반환합니다.
	`Sign`	음수 값에 대한 -1, 0의 경우 0, 양수 값의 경우 +1을 반환합니다.
INumberBase<TSelf>	`One`	형식의 값 1을 가져옵니다.
	`Radix`	형식의 radix 또는 base를 가져옵니다. Int32 는 2를 반환합니다. Decimal 10을 반환합니다.
	`Zero`	형식의 값 0을 가져옵니다.
	`CreateChecked`	입력이 맞지 않을 경우 OverflowException을(를) 던져 값을 생성합니다.[1]
	`CreateSaturating`	입력이 맞지 않을 경우 `T.MinValue` 또는 `T.MaxValue`로 고정하여 값을 만듭니다.[1]

인터페이스	API 이름	설명
	`CreateTruncating`	입력이 맞지 않는 경우 래핑하여 다른 값에서 값을 만듭니다. [1]
	`IsComplexNumber`	값에 0이 아닌 실제 부분과 0이 아닌 허수 부분이 있으면 true를 반환합니다.
	`IsEvenInteger`	값이 짝수이면 true를 반환합니다. 2.0은 `true`을 반환하고, 2.2는 `false`을 반환합니다.
	`IsFinite`	값이 무한이 아닌 `NaN`경우 true를 반환합니다.
	`IsImaginaryNumber`	값에 실제 부분이 0이면 true를 반환합니다. 즉 `0` , 허수이고 `1 + 1i` 그렇지 않습니다.
	`IsInfinity`	값이 무한대를 나타내는 경우 true를 반환합니다.
	`IsInteger`	값이 정수이면 true를 반환합니다. 2.0 및 3.0 반환 `true`및 2.2 및 3.1 반환 `false`.
	`IsNaN`	값이 `NaN`을 나타내면 true를 반환합니다.
	`IsNegative`	값이 음수이면 true를 반환합니다. 여기에는 -0.0이 포함됩니다.
	`IsPositive`	값이 양수이면 true를 반환합니다. 여기에는 0 및 +0.0이 포함됩니다.
	`IsRealNumber`	값에 허수 부분이 0이면 true를 반환합니다. 즉, 0은 모든 `INumber<T>` 형식과 마찬가지로 실제입니다.
	`IsZero`	값이 0을 나타내는 경우 true를 반환합니다. 여기에는 0, +0.0 및 -0.0이 포함됩니다.
	`MaxMagnitude`	절대값의 크기를 비교하여 더 큰 값을 반환하며, 만약 어느 한 입력이 `NaN`라면 `NaN`을 반환합니다.
	`MaxMagnitudeNumber`	절대값이 더 큰 값을 반환하고 입력이 `NaN`1인 경우 숫자를 반환합니다.
	`MinMagnitude`	입력 값 중 하나가 `NaN`일 경우 `NaN`을 반환하며, 그렇지 않으면 절대값이 더 작은 값을 반환합니다.
	`MinMagnitudeNumber`	절대값이 더 작은 값을 반환하고 입력 `NaN`이 1인 경우 숫자를 반환합니다.
ISignedNumber<TSelf>	`NegativeOne`	형식에 대한 -1 값을 가져옵니다.

[1]세 `Create*` 가지 메서드의 동작을 이해하려면 다음 예제를 고려하세요.

너무 큰 값이 지정된 경우의 예:

- `byte.CreateChecked(384)`이(가) OverflowException을(를) 발생시킵니다.
- `byte.CreateSaturating(384)` 는 384가 255보다 Byte.MaxValue 크므로 255를 반환합니다.
- `byte.CreateTruncating(384)` 는 가장 낮은 8비트를 사용하므로 128을 반환합니다(384에는 16진수 표현 `0x0180`이 있고 가장 낮은 8비트는 `0x80` 128).

너무 작은 값이 지정된 경우의 예:

- `byte.CreateChecked(-384)`이(가) OverflowException을(를) 발생시킵니다.
- `byte.CreateSaturating(-384)` 는 -384 (0)보다 Byte.MinValue 작기 때문에 0을 반환합니다.
- `byte.CreateTruncating(-384)` 는 가장 낮은 8비트를 사용하므로 128을 반환합니다(384에는 16진수 표현 `0xFE80`이 있고 가장 낮은 8비트는 `0x80` 128).

또한 `Create*` 메서드는 `float` 및 `double`와 같은 IEEE 754 부동 소수점 형식에 대해 `PositiveInfinity`, `NegativeInfinity`, `NaN` 등의 특수 값을 사용할 때 몇 가지 특별한 고려 사항이 있습니다. 모든 세 개의 `Create*` API는 `CreateSaturating`로 동작합니다. 또한 `MinValue` 및 `MaxValue`는 가장 큰 음수/양수 "표준" 숫자를 나타내지만, 실제 최소값과 최대값은 `NegativeInfinity` 및 `PositiveInfinity`이므로, 대신 이 값으로 고정됩니다.

연산자 인터페이스

연산자 인터페이스는 C# 언어에서 사용할 수 있는 다양한 연산자에 해당합니다.

- 모든 형식에 대해 올바르지 않으므로 곱하기 및 나누기와 같은 작업을 명시적으로 페어링하지 않습니다. 예를 들어 `Matrix4x4 * Matrix4x4` 유효하지만 `Matrix4x4 / Matrix4x4` 유효하지는 않습니다.
- 일반적으로 입력 및 결과 형식은 두 정수로 나누어 정`double` `3 / 2 = 1.5`수 집합의 평균을 계산하는 등의 시나리오를 지원하기 위해 다를 수 있습니다.

테이블 확장

인터페이스 이름	정의된 연산자
IAdditionOperators<TSelf,TOther,TResult>	`x + y`
IBitwiseOperators<TSelf,TOther,TResult>	`x & y`, 'x | y', `x ^ y` 및 `~x`
IComparisonOperators<TSelf,TOther,TResult>	`x < y`, `x > y`, `x <= y` 및 `x >= y`
IDecrementOperators<TSelf>	`--x` 및 `x--`
IDivisionOperators<TSelf,TOther,TResult>	`x / y`

인터페이스 이름	정의된 연산자
IEqualityOperators<TSelf,TOther,TResult>	`x == y` 및 `x != y`
IIncrementOperators<TSelf>	`++x` 및 `x++`
IModulusOperators<TSelf,TOther,TResult>	`x % y`
IMultiplyOperators<TSelf,TOther,TResult>	`x * y`
IShiftOperators<TSelf,TOther,TResult>	`x << y` 및 `x >> y`
ISubtractionOperators<TSelf,TOther,TResult>	`x - y`
IUnaryNegationOperators<TSelf,TResult>	`-x`
IUnaryPlusOperators<TSelf,TResult>	`+x`

> **참고**
>
> 일부 인터페이스는 선택되지 않은 일반 연산자 외에 확인된 연산자를 정의합니다. 확인된 연산자는 확인된 컨텍스트에서 호출되며 사용자 정의 형식이 오버플로 동작을 정의할 수 있도록 허용합니다. 예를 들어 CheckedSubtraction(TSelf, TOther)확인된 연산자를 구현하는 경우 예를 들어 Subtraction(TSelf, TOther)선택되지 않은 연산자도 구현해야 합니다.

함수 인터페이스

함수 인터페이스는 특정 숫자 인터페이스보다 더 광범위하게 적용되는 일반적인 수학 API를 정의합니다. 이러한 인터페이스는 모두 에 의해 IFloatingPointIeee754<TSelf>구현되며 나중에 다른 관련 형식에 의해 구현될 수 있습니다.

테이블 확장

인터페이스 이름	설명
IExponentialFunctions<TSelf>	`e^x`, `e^x - 1`, `2^x`, `2^x - 1`, `10^x`, 및 `10^x - 1`를 지원하는 지수 함수를 제공합니다.
IHyperbolicFunctions<TSelf>	`acosh(x)`, `asinh(x)`, `atanh(x)`, `cosh(x)`, `sinh(x)`, 및 `tanh(x)`를 지원하는 하이퍼볼릭 함수를 노출합니다.
ILogarithmicFunctions<TSelf>	`ln(x)`, `ln(x + 1)`, `log2(x)`, `log2(x + 1)`, `log10(x)` 및 `log10(x + 1)`를 지원하는 로그 함수를 제공합니다.
IPowerFunctions<TSelf>	를 지원하는 전원 함수를 노출합니다 `x^y`.

인터페이스 이름	설명
IRootFunctions<TSelf>	루트 기능을 노출하여 `cbrt(x)` 및 `sqrt(x)`를 지원합니다.
ITrigonometricFunctions<TSelf>	`acos(x)`, `asin(x)`, `atan(x)`, `cos(x)`, `sin(x)`, 및 `tan(x)`를 지원하는 삼각 함수를 제공합니다.

인터페이스 구문 분석 및 서식 지정

구문 분석 및 서식 지정은 프로그래밍의 핵심 개념입니다. 사용자 입력을 지정된 형식으로 변환하거나 사용자에게 형식을 표시할 때 일반적으로 사용됩니다. 이러한 인터페이스는 네임스페이스에 System 있습니다.

테이블 확장

인터페이스 이름	설명
IParsable<TSelf>	`T.Parse(string, IFormatProvider)` 및 `T.TryParse(string, IFormatProvider, out TSelf)`에 대한 지원을 제공합니다.
ISpanParsable<TSelf>	`T.Parse(ReadOnlySpan<char>, IFormatProvider)` 및 `T.TryParse(ReadOnlySpan<char>, IFormatProvider, out TSelf)`에 대한 지원을 제공합니다.
IFormattable [1]	에 대한 `value.ToString(string, IFormatProvider)`지원을 노출합니다.
ISpanFormattable [1]	에 대한 `value.TryFormat(Span<char>, out int, ReadOnlySpan<char>, IFormatProvider)`지원을 노출합니다.

[1]이 인터페이스는 새로운 인터페이스가 아니며 제네릭도 아닙니다. 그러나 모든 숫자 형식에 의해 구현되며 역 연산 `IParsable`을 나타냅니다.

예를 들어 다음 프로그램은 형식 매개 변수가 제한 IParsable<TSelf>되는 제네릭 메서드를 사용하여 콘솔에서 읽는 두 숫자를 입력으로 사용합니다. 입력 및 결과 값의 형식 매개 변수가 제한 INumber<TSelf>되는 제네릭 메서드를 사용하여 평균을 계산한 다음 결과를 콘솔에 표시합니다.

C#

```
using System.Globalization;
using System.Numerics;

static TResult Average<T, TResult>(T first, T second)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    return TResult.CreateChecked( (first + second) / T.CreateChecked(2) );
```

```
}

static T ParseInvariant<T>(string s)
    where T : IParsable<T>
{
    return T.Parse(s, CultureInfo.InvariantCulture);
}

Console.Write("First number: ");
var left = ParseInvariant<float>(Console.ReadLine());

Console.Write("Second number: ");
var right = ParseInvariant<float>(Console.ReadLine());

Console.WriteLine($"Result: {Average<float, float>(left, right)}");

/* This code displays output similar to:

First number: 5.0
Second number: 6
Result: 5.5
*/
```

참고하십시오

- .NET 7의 일반 수학(블로그 게시물)

.NET의 제네릭 인터페이스

2025. 06. 17.

이 문서에서는 제네릭 형식의 제품군에서 공통 기능을 제공하는 .NET의 제네릭 인터페이스에 대해 개요를 제공합니다.

제네릭 인터페이스는 데이터 형식의 안전성을 보장하면서, 순서 비교 및 같음 비교를 위한 비제네릭 인터페이스의 안전한 대안이 되며, 제네릭 컬렉션 형식에서 공유하는 기능을 제공합니다. .NET 7에는 숫자와 유사한 형식에 대한 제네릭 인터페이스가 도입되었습니다. 예를 들면 System.Numerics.INumber<TSelf>다음과 같습니다. 이러한 인터페이스를 사용하면 제네릭 형식 매개 변수가 제네릭 숫자 인터페이스를 구현하는 형식으로 제한되는 수학 기능을 제공하는 제네릭 메서드를 정의할 수 있습니다.

> ⓘ **참고**
>
> 여러 제네릭 인터페이스의 형식 매개 변수는 공변성 또는 반공변으로 표시되어 이러한 인터페이스를 구현하는 형식을 할당하고 사용하는 데 더 큰 유연성을 제공합니다. 자세한 내용은 **공변성 및 반공변성(Contravariance)**을 참조하세요.

같음 및 순서 비교

- 네임스페이스 System에는 제네릭 인터페이스인 **System.IComparable<T>**와 **System.IEquatable<T>**가 있으며, 이들은 비제네릭 인터페이스와 같이 각각 정렬 비교와 같음 비교를 위한 메서드를 정의합니다. 형식은 이러한 인터페이스를 구현하여 이러한 비교를 수행하는 기능을 제공합니다.

- 네임스페이스 System.Collections.Generic 안에서 **IComparer<T>** 및 **IEqualityComparer<T>** 제네릭 인터페이스는 System.IComparable<T> 또는 System.IEquatable<T> 인터페이스를 구현하지 않는 형식에 대한 순서 또는 동등성 비교를 정의할 수 있는 방법을 제공합니다. 관계를 재정의할 수 있는 유형에 대해 방법을 제공하기도 합니다.

  이러한 인터페이스는 많은 제네릭 컬렉션 클래스의 메서드 및 생성자에서 사용됩니다. 예를 들어 제네릭 개체를 클래스의 IComparer<T> 생성자에 전달하여 제네릭 SortedDictionary<TKey,TValue>System.IComparable<T>을 구현하지 않는 형식의 정렬 순서를 지정할 수 있습니다. 제네릭 정적 메서드의 Array.Sort 오버로드와 List<T>.Sort 제네릭 IComparer<T> 구현을 사용하여 배열 및 목록을 정렬하는 인스턴스 메서드가 있습니다.

  Comparer<T>와 EqualityComparer<T> 제네릭 클래스는 IComparer<T> 및 IEqualityComparer<T> 제네릭 인터페이스의 구현을 위한 기본 클래스를 제공하며, 각각의

Comparer<T>.Default 및 EqualityComparer<T>.Default 속성을 통해 기본 순서 및 같음 비교를 제공합니다.

컬렉션 기능

- **ICollection<T>** 제네릭 인터페이스는 제네릭 컬렉션 형식에 대한 기본 인터페이스입니다. 요소를 추가, 제거, 복사 및 열거하기 위한 기본 기능을 제공합니다. ICollection<T> 는 제네릭 및 비제네릭 IEnumerable<T>IEnumerable에서 상속됩니다.

- **IList<T>** 제네릭 인터페이스는 인덱스 검색을 위한 메서드로 ICollection<T> 제네릭 인터페이스를 확장합니다.

- 제네릭 인터페이스 **IDictionary<TKey,TValue>**는 ICollection<T> 제네릭 인터페이스에 키 검색 메서드를 추가하여 확장합니다. .NET 기본 클래스 라이브러리의 제네릭 사전 형식도 비제네릭 IDictionary 인터페이스를 구현합니다.

- 제네릭 인터페이스는 **IEnumerable<T>** 제네릭 열거자 구조를 제공합니다. 제네릭 열거자가 구현하는 제네릭 인터페이스는 IEnumerator<T> 제네릭 IEnumerator 이 아닌 인터페이스 MoveNext 를 상속합니다. 형식 매개 변수Reset에 의존하지 않는 멤버 및 `T` 멤버는 비제네릭 인터페이스에만 나타납니다. 즉, 제네릭이 아닌 인터페이스의 소비자도 제네릭 인터페이스를 사용할 수 있습니다.

수학 기능

.NET 7에는 숫자와 유사한 형식 및 사용할 수 있는 System.Numerics 기능을 설명하는 제네릭 인터페이스가 네임스페이스에 도입되었습니다. .NET 기본 클래스 라이브러리가 제공하는 20개의 숫자 형식은 예를 들어 Int32 및 Double를 포함하여 이러한 인터페이스를 구현하도록 업데이트되었습니다. 이 인터페이스 중에서 가장 두드러지는 것은 INumber<TSelf>이며, 이는 "실수" 수와 대략적으로 해당합니다.

이러한 인터페이스에 대한 자세한 내용은 제네릭 수학을 참조하세요.

참고하십시오

- System.Collections.Generic
- System.Collections.ObjectModel
- 제네릭
- .NET의 제네릭 컬렉션
- 배열 및 목록을 조작하기 위한 제네릭 대리자
- 공변성(Covariance) 및 반공변성(Contravariance)

제네릭의 공변성 및 반공변성

공변성 및 *반공변성(contravariance)*은 형식 인수가 상속에 의해 관련될 때 생성된 제네릭 형식 간의 참조 변환이 작동하는 방식(예: `IEnumerable<Derived>` 및 `IEnumerable<Base>` 사이)을 설명합니다. 분산은 제네릭 인터페이스 또는 대리자 형식의 형식 매개 변수의 속성이며 다른 형식 인수를 사용하는 생성된 형식 간에 존재하는 암시적 변환을 제어합니다. 기본적으로 제네릭 형식 매개 변수는 *고정* 형식입니다. 한 형식 인수가 다른 형식에서 파생되더라도 형식 매개 변수가 `List<Derived>` 또는 `List<Base>`으로 명시적으로 선언되지 않는 한 생성된 제네릭 형식(예: 및)은 관련이 없습니다.

다음 정의 및 예제에서는 명명된 기본 클래스와 이름이 `Base`지정된 `Derived` 파생 클래스를 가정합니다.

- *고정* 형식 인수를 사용하려면 지정된 형식을 정확히 사용해야 합니다. 파생 형식이나 기본 형식을 대체할 수 없습니다.

  예를 들어 형식 `List<Base>`변수에 인스턴스 `List<Derived>` 를 할당하거나 그 반대로 할당할 수 없습니다.

- *공변* 형식 매개 변수를 사용하면 원래 형식 인수에 더 많은 파생 형식을 대체할 수 있습니다.

  예를 들어 형식 `IEnumerable<Derived>`변수에 `IEnumerable<Base>` 인스턴스를 할당할 수 있습니다.

- *반공변* 형식 매개 변수를 사용하면 원래 형식 대신 파생 형식 인수의 기본 형식을 대체할 수 있습니다.

  예를 들어 형식 `Action<Base>`변수에 `Action<Derived>` 인스턴스를 할당할 수 있습니다.

공변 형식 매개 변수를 사용하면 다음 코드와 같이 일반적인 다형성처럼 보이는 할당을 만들 수 있습니다.

C#

```
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

List<T> 클래스는 IEnumerable<T> 인터페이스를 구현하므로, `List<Derived>`는 `List(Of Derived)` (Visual Basic에서) `IEnumerable<Derived>`를 구현합니다. 공변 형식 매개 변수는 나머지를 수행합니다.

반면에 반공변성은 직관에 어긋나는 것처럼 보입니다. 다음 예제에서는 Visual Basic에서 형식 `Action<Base>` `Action(Of Base)` 의 대리자를 만든 다음 해당 대리자를 형식`Action<Derived>`의 변

수에 할당합니다.

C#

```
Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

거꾸로 보일 수 있지만, 이는 컴파일되고 실행되는 타입 안전 코드입니다. 람다 식은 할당된 대리자와 일치하므로 하나의 형식 `Base` 매개 변수를 사용하고 반환 값이 없는 메서드를 정의합니다. 형식 매개 변수 `Action<Derived>`가 반공변성이므로, `T` 대리자의 생성된 대리자를 형식 Action<T> 변수에 할당할 수 있습니다. 매개 변수 형식을 지정하기 때문에 `T` 코드는 형식이 안전합니다. 형식의 대리자가 형식 `Action<Base>` `Action<Derived>`의 대리자인 것처럼 호출되는 경우 해당 인수는 형식 `Derived`이어야 합니다. 메서드의 매개변수가 `Base` 형식이기 때문에 이 인수는 항상 기본 메서드에 안전하게 전달할 수 있습니다.

일반적으로 공변 형식 매개 변수는 대리자의 반환 형식으로 사용할 수 있으며 반공변 형식 매개 변수를 매개 변수 형식으로 사용할 수 있습니다. 인터페이스의 경우 공변 형식 매개 변수를 인터페이스 메서드의 반환 형식으로 사용할 수 있으며 반공변 형식 매개 변수를 인터페이스 메서드의 매개 변수 형식으로 사용할 수 있습니다.

공변성 및 반공변성(contravariance)을 통칭하여 *분산*이라고 합니다. 공변성 또는 반공변으로 표시되지 않은 제네릭 형식 매개 변수를 *고정*이라고 합니다. 공용 언어 런타임의 분산에 대한 팩트의 간략한 요약:

- 변형 형식 매개 변수는 제네릭 인터페이스 및 제네릭 대리자 형식으로 제한됩니다.
- 제네릭 인터페이스 또는 제네릭 대리자 형식에는 공변 및 반공변 형식 매개 변수가 모두 있을 수 있습니다.
- 변형성은 참조 형식에만 적용됩니다. 변형 형식 매개 변수에 값 형식을 지정하면 해당 형식 매개 변수는 결과로 생성된 형식에서 불변이 됩니다.
- 대리자 조합에는 분산이 적용되지 않습니다. 즉, `Action<Derived>` 및 `Action<Base>` 형식의 대리자 두 개(`Action(Of Derived)` 및 `Action(Of Base)`는 Visual Basic에서 사용)는 결과가 형식적으로 안전하더라도 두 번째 대리자를 첫 번째 대리자와 결합할 수 없습니다. 분산을 사용하면 두 번째 대리자를 형식 변수에 할당할 수 있지만 대리자는 형식 `Action<Derived>`이 정확히 일치하는 경우에만 결합할 수 있습니다.
- C# 9부터 공변 반환 형식이 지원됩니다. 재정의 메서드는 재정의하는 메서드를 더 파생된 반환 형식으로 선언할 수 있으며 재정의되는 읽기 전용 속성은 더 파생된 형식을 선언할 수 있습니다.

공변 형식 매개 변수를 사용하는 제네릭 인터페이스

여러 제네릭 인터페이스에는 공변 형식 매개 변수(예: IEnumerable<T>, IEnumerator<T>, IQueryable<T> 및)가 있습니다 IGrouping<TKey,TElement>. 이러한 인터페이스의 모든 형식 매개 변수는 공변성이므로 형식 매개 변수는 멤버의 반환 형식에만 사용됩니다.

다음 예제에서는 공변 형식 매개 변수를 보여 줍니다. 이 예제는 두 가지 형식을 정의합니다: `Base`에는 `PrintBases`이라는 정적 메서드가 있으며, 이 메서드는 `IEnumerable<Base>` (`IEnumerable(Of Base)`은 Visual Basic) 매개변수를 받아 요소를 출력합니다. `Derived`는 `Base`로부터 상속받습니다. 이 예제에서는 빈 `List<Derived>` 형식(`List(Of Derived)` Visual Basic)을 만들고 이 형식을 캐스팅하지 않고 형식 `PrintBases` 변수에 `IEnumerable<Base>` 전달하고 할당할 수 있음을 보여 줍니다. List<T>는 단일 공변 형식 매개 변수가 있는 IEnumerable<T>을 구현합니다. 공변 형식 매개 변수는 인스턴스 `IEnumerable<Derived>` 를 대신 사용할 `IEnumerable<Base>` 수 있는 이유입니다.

C#

```
using System;
using System.Collections.Generic;

class Base
{
    public static void PrintBases(IEnumerable<Base> bases)
    {
        foreach(Base b in bases)
        {
            Console.WriteLine(b);
        }
    }
}

class Derived : Base
{
    public static void Main()
    {
        List<Derived> dlist = new List<Derived>();

        Derived.PrintBases(dlist);
        IEnumerable<Base> bIEnum = dlist;
    }
}
```

반공변 형식 매개 변수를 사용하는 제네릭 인터페이스

여러 제네릭 인터페이스에는 반공변 형식 매개 변수가 있습니다. 예: IComparer<T>, IComparable<T> 및 IEqualityComparer<T>. 이러한 인터페이스에는 반공변 형식 매개 변수만 있으므로 형식 매개 변수는 인터페이스의 멤버에서 매개 변수 형식으로만 사용됩니다.

다음 예제에서는 반공변 형식 매개 변수를 보여 줍니다. 이 예제에서는 `MustInherit` Visual Basic에서 추상 `Shape` 클래스를 정의하고, `Area` 속성을 포함합니다. 이 예제에서는 `ShapeAreaComparer`을(를) 구현하는 `IComparer<Shape>` 클래스도 Visual Basic(`IComparer(Of Shape)`)에서 정의합니다. 메서드의 IComparer<T>.Compare 구현은 속성 값을 `Area` 기반으로 하므로 `ShapeAreaComparer` 영역별로 개체를 정렬 `Shape` 하는 데 사용할 수 있습니다.

`Circle` 클래스는 `Shape`를 상속하고 `Area`를 재정의합니다. 이 예제는 SortedSet<T>의 `Circle` 개체를 생성하기 위해, `IComparer<Circle>`(Visual Basic에서는 `IComparer(Of Circle)`)을 받는 생성자를 사용합니다. 그러나 이 예제에서는 `IComparer<Circle>`을(를) 전달하는 대신 `ShapeAreaComparer`을(를) 구현하는 `IComparer<Shape>` 개체를 전달합니다. 이 예제에서는 제네릭 인터페이스의 형식 매개 변수가`Shape` 반공변성이므로 코드가 더 파생된 형식()의 비교자를 호출할 때 덜 파생된 형식(`Circle`)의 IComparer<T> 비교자를 전달할 수 있습니다.

새 `Circle` 개체가 `SortedSet<Circle>`에 추가되면, 새 요소를 기존 요소와 비교할 때마다 `IComparer<Shape>.Compare` 개체의 `IComparer(Of Shape).Compare` 메서드(`ShapeAreaComparer` 메서드는 Visual Basic의 경우)가 호출됩니다. 메서드(`Shape`)의 매개 변수 형식은 전달되는 형식(`Circle`)보다 덜 파생되므로 호출은 형식이 안전합니다. 반공변성을 사용하면 `ShapeAreaComparer`에서 파생된 단일 형식의 컬렉션뿐만 아니라 `Shape`에서 파생된 혼합 형식의 컬렉션도 정렬할 수 있습니다.

C#

```
using System;
using System.Collections.Generic;

abstract class Shape
{
    public virtual double Area { get { return 0; }}
}

class Circle : Shape
{
    private double r;
    public Circle(double radius) { r = radius; }
    public double Radius { get { return r; }}
    public override double Area { get { return Math.PI * r * r; }}
}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>
{
    int IComparer<Shape>.Compare(Shape a, Shape b)
    {
```

```
        if (a == null) return b == null ? 0 : -1;
        return b == null ? 1 : a.Area.CompareTo(b.Area);
    }
}

class Program
{
    static void Main()
    {
        // You can pass ShapeAreaComparer, which implements IComparer<Shape>,
        // even though the constructor for SortedSet<Circle> expects
        // IComparer<Circle>, because type parameter T of IComparer<T> is
        // contravariant.
        SortedSet<Circle> circlesByArea =
            new SortedSet<Circle>(new ShapeAreaComparer())
                { new Circle(7.2), new Circle(100), null, new Circle(.01) };

        foreach (Circle c in circlesByArea)
        {
            Console.WriteLine(c == null ? "null" : "Circle with area " + c.Area);
        }
    }
}

/* This code example produces the following output:

null
Circle with area 0.000314159265358979
Circle with area 162.860163162095
Circle with area 31415.9265358979
 */
```

변형 형식 매개 변수가 있는 제네릭 대리자

`Func` 제네릭 대리자, 예를 들어 Func<T,TResult>, 는 공변 반환 형식과 반공변 매개 변수 형식을 가집니다. `Action` 같은 제네릭 대리자는 Action<T1,T2>와 같은 반공변 매개 변수 형식을 가집니다. 즉, 파생 매개 변수 형식이 더 많고(제네릭 대리자의 `Func` 경우) 파생 반환 형식이 적은 변수에 대리자를 할당할 수 있습니다.

> ⓘ **참고 항목**
>
> 제네릭 대리자의 `Func` 마지막 제네릭 형식 매개 변수는 대리자 서명에 있는 반환 값의 형식을 지정합니다. 다른 제네릭 형식 매개 변수는 반공변(`out` 키워드)인 반면 공변성(`in` 키워드)입니다.

다음 코드에서는 이를 보여 줍니다. 코드의 첫 번째 부분은 `Base`라는 이름의 클래스, `Derived`를 상속하는 `Base`라는 클래스, 그리고 `static`라는 메서드(`Shared`는 Visual Basic에서의 명칭)를 포

함하는 다른 클래스를 정의합니다. 메서드는 인스턴스 `Base` 를 가져와서 인스턴스를 `Derived`반환합니다. 인수가 인스턴스 `Derived` `MyMethod` 인 경우 해당 인수를 반환하고 인수가 인스턴스 `Base` `MyMethod` 인 경우 .의 `Derived`새 인스턴스를 반환합니다. 예제에서는 `Main()`(Visual Basic에서) 나타내는 인스턴스 `Func<Base, Derived>` 를 만들고 변수`Func(Of Base, Derived)`에 저장합니다`MyMethod` `f1`.

C#

```
public class Base {}
public class Derived : Base {}

public class Program
{
    public static Derived MyMethod(Base b)
    {
        return b as Derived ?? new Derived();
    }

    static void Main()
    {
        Func<Base, Derived> f1 = MyMethod;
```

두 번째 코드 조각에서는 반환 형식이 공변성이므로 대리자를 형식 `Func<Base, Base>` 변수(`Func(Of Base, Base)` Visual Basic의 경우)에 할당할 수 있음을 보여줍니다.

C#

```
// Covariant return type.
Func<Base, Base> f2 = f1;
Base b2 = f2(new Base());
```

세 번째 코드 조각에서는 매개 변수 형식이 반공변이므로 대리자를 Visual Basic의 형식 `Func<Derived, Derived>` `Func(Of Derived, Derived)` 변수에 할당할 수 있음을 보여줍니다.

C#

```
// Contravariant parameter type.
Func<Derived, Derived> f3 = f1;
Derived d3 = f3(new Derived());
```

코드의 마지막 부분에서는 반공변성 매개 변수 형식과 공변성 반환 형식의 효과를 결합하여 대리자를 형식 `Func<Derived, Base>` 변수(`Func(Of Derived, Base)` Visual Basic의 경우)에 할당할 수 있음을 보여 줍니다.

C#

```
// Covariant return type and contravariant parameter type.
Func<Derived, Base> f4 = f1;
Base b4 = f4(new Derived());
```

제네릭이 아닌 대리자에서의 변동

앞의 코드에서 `MyMethod`의 시그니처는 생성된 제네릭 대리자 `Func<Base, Derived>`의 시그니처와 정확히 일치합니다 (`Func(Of Base, Derived)` in Visual Basic). 이 예제에서는 모든 대리자 형식이 제네릭 대리자 형식에서 생성되는 한 파생된 매개 변수 형식이 더 많고 파생된 반환 형식이 적은 변수 또는 메서드 매개 변수에 이 제네릭 대리 Func<T,TResult>자를 저장할 수 있음을 보여줍니다.

이것은 중요한 점입니다. 제네릭 대리자의 형식 매개 변수에서 공변성 및 반공변성의 효과는 일반 대리자 바인딩에서 공변성 및 반공변성의 영향과 유사합니다(대리자의 차이(C#) 및 대리자의 차이 참조(Visual Basic)) 그러나 대리자 바인딩의 분산은 변형 형식 매개 변수가 있는 제네릭 대리자 형식뿐만 아니라 모든 대리자 형식에서 작동합니다. 또한 대리자 바인딩의 분산을 사용하면 더 제한적인 매개 변수 형식과 덜 제한적인 반환 형식이 있는 대리자에서 메서드를 바인딩할 수 있지만 제네릭 대리자의 할당은 두 대리자 형식이 동일한 제네릭 형식 정의에서 생성된 경우에만 작동합니다.

다음 예제에서는 대리자 바인딩에서 분산의 결합된 효과와 제네릭 형식 매개 변수의 분산을 보여 줍니다. 이 예제에서는 최소 파생()에서 가장`Type1` 많이 파생된(`Type3`)에 이르는 세 가지 형식을 포함하는 형식 계층 구조를 정의합니다. 일반 대리자 바인딩의 분산은 매개 변수 형식과 반환 형식이 `Type1` 있는 메서드를 매개 변수 `Type3` 형식과 반환 `Type2`형식이 `Type2` 있는 제네릭 대리자로 바인딩하는 데 사용됩니다. 그런 다음 제네릭 형식 매개 변수의 공변성과 반공변성을 사용하여 형식이 `Type3`이고 반환 형식이 `Type1`인 제네릭 대리자가 다른 변수에 할당됩니다. 두 번째 할당에서는 변수 형식과 대리자 형식을 모두 동일한 제네릭 형식 정의(이 경우 Func<T,TResult>)에서 생성해야 합니다.

C#

```
using System;

public class Type1 {}
public class Type2 : Type1 {}
public class Type3 : Type2 {}

public class Program
{
    public static Type3 MyMethod(Type1 t)
    {
        return t as Type3 ?? new Type3();
    }

    static void Main()
```

```
    {
        Func<Type2, Type2> f1 = MyMethod;

        // Covariant return type and contravariant parameter type.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}
```

변형 제네릭 인터페이스 및 대리자 정의

Visual Basic 및 C#에는 인터페이스 및 대리자의 제네릭 형식 매개 변수를 공변 또는 반공변으로 표시할 수 있는 키워드가 있습니다.

공변 형식 매개 변수는 키워드(`out`Visual Basic의 `Out` 키워드)로 표시됩니다. 공변 형식 매개 변수를 인터페이스에 속하는 메서드의 반환 값으로 사용하거나 대리자의 반환 형식으로 사용할 수 있습니다. 공변 형식 매개 변수를 인터페이스 메서드에 대한 제네릭 형식 제약 조건으로 사용할 수 없습니다.

> ⓘ **참고 항목**
>
> 인터페이스의 메서드에 제네릭 대리자 형식인 매개 변수가 있는 경우 인터페이스 형식의 공변 형식 매개 변수를 사용하여 대리자 형식의 반공변 형식 매개 변수를 지정할 수 있습니다.

반공변 형식 매개 변수는 키워드(`in`Visual Basic의 `In` 키워드)로 표시됩니다. 반공변 형식 매개 변수를 인터페이스에 속하는 메서드의 매개 변수 형식 또는 대리자의 매개 변수 형식으로 사용할 수 있습니다. 반공변성 형식 매개 변수를 인터페이스 메서드에 대한 제네릭 형식 제약 조건으로 사용할 수 있습니다.

인터페이스 형식 및 대리자 형식만 변형 형식 매개 변수를 가질 수 있습니다. 인터페이스 또는 대리자 형식에는 공변 및 반공변 형식 매개 변수가 모두 있을 수 있습니다.

Visual Basic 및 C#에서는 공변 및 반공변성 형식 매개 변수를 사용하는 규칙을 위반하거나 인터페이스 및 대리자가 아닌 형식의 형식 매개 변수에 공변성 및 반공변성 주석을 추가할 수 없습니다.

자세한 내용 및 예제 코드는 제네릭 인터페이스의 분산(C#) 및 제네릭 인터페이스의 분산(Visual Basic)을 참조하세요.

형식 목록

다음 인터페이스 및 대리자 형식에는 공변 및/또는 반공변 형식 매개 변수가 있습니다.

테이블 확장

유형	공변 형식 매개 변수	반공변 형식 매개 변수
Action<T> Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>		예
Comparison<T>		예
Converter<TInput,TOutput>	예	예
Func<TResult>	예	
Func<T,TResult> Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>	예	예
IComparable<T>		예
Predicate<T>		예
IComparer<T>		예
IEnumerable<T>	예	
IEnumerator<T>	예	
IEqualityComparer<T>		예
IGrouping<TKey,TElement>	예	
IOrderedEnumerable<TElement>	예	
IOrderedQueryable<T>	예	
IQueryable<T>	예	

참고하십시오

- 공변성 및 반공변성(C#)
- 공변성 및 반공변성(Visual Basic)
- 대리자의 변이(C#)
- 대리자 차이(Visual Basic)

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 31.

컬렉션 및 데이터 구조

컬렉션으로 저장하고 조작할 때 유사한 데이터를 보다 효율적으로 처리할 수 있는 경우가 많습니다. 클래스 또는 System.Array 클래스와 System.Collections, System.Collections.Generic, System.Collections.Concurrent, System.Collections.Immutable 네임스페이스의 클래스를 사용하여 컬렉션에서 개별 요소 또는 요소 범위를 추가, 제거 및 수정할 수 있습니다.

컬렉션에는 두 가지 주요 형식이 있습니다. 제네릭 컬렉션 및 제네릭이 아닌 컬렉션입니다. 제네릭 컬렉션은 컴파일 시 형식이 안전합니다. 따라서 제네릭 컬렉션은 일반적으로 더 나은 성능을 제공합니다. 제네릭 컬렉션은 생성될 때 형식 매개 변수를 허용합니다. 컬렉션에서 항목을 추가하거나 제거할 때 형식에서 Object 캐스팅할 필요가 없습니다. 또한 대부분의 일반 컬렉션은 Windows 스토어 앱에서 지원됩니다. 제네릭이 아닌 컬렉션은 항목을 저장할 때 Object 캐스팅이 필요하며, 이로 인해 대부분이 Windows 스토어 앱 개발에 지원되지 않습니다. 그러나 이전 코드에는 제네릭이 아닌 컬렉션이 표시될 수 있습니다.

.NET Framework 4 이상 버전에서 네임스페이스의 System.Collections.Concurrent 컬렉션은 여러 스레드에서 컬렉션 항목에 액세스하기 위한 효율적인 스레드로부터 안전한 작업을 제공합니다. 네임스페이스 (System.Collections.Immutable)의 변경할 수 없는 컬렉션 클래스는 원래 컬렉션의 복사본에서 작업을 수행하고 원래 컬렉션을 수정할 수 없으므로 기본적으로 스레드로부터 안전합니다.

일반적인 컬렉션 기능

모든 컬렉션은 컬렉션에서 항목을 추가, 제거 또는 찾기 위한 메서드를 제공합니다. 또한 ICollection 인터페이스 또는 ICollection<T> 인터페이스를 직접 또는 간접적으로 구현하는 모든 컬렉션은 이러한 기능을 공유합니다.

- **컬렉션을 열거하는 기능**

  .NET 컬렉션은 컬렉션을 반복할 수 있도록 System.Collections.IEnumerable 또는 System.Collections.Generic.IEnumerable<T>을 구현합니다. 열거자는 컬렉션의 모든 요소에 대한 이동 가능한 포인터로 생각할 수 있습니다. foreach, in statement 및 For Each... Next 문은 GetEnumerator 메서드에 의해 노출되는 열거자를 사용하고, 열거자를 조작하는 복잡성을 숨깁니다. 또한 구현 System.Collections.Generic.IEnumerable<T> 하는 모든 컬렉션은 *쿼리 가능한 형식* 으로 간주되며 LINQ를 사용하여 쿼리할 수 있습니다. LINQ 쿼리는 데이터에 액세스하기 위한 일반적인 패턴을 제공합니다. 일반적으로 표준 `foreach` 루프보다 간결하고 읽기 가능하며 필터링, 순서 지정 및 그룹화 기능을 제공합니다. LINQ 쿼리는 성능을 향상시킬 수도 있습니다. 자세한 내용은 LINQ to Objects(C#), LINQ to Objects(Visual Basic), PLINQ(병렬 LINQ),LINQ 쿼리 소개(C#) 및 기본 쿼리 작업(Visual Basic)을 참조하세요.

- **컬렉션 콘텐츠를 배열에 복사하는 기능**

  모든 컬렉션은 메서드를 사용하여 `CopyTo` 배열에 복사할 수 있습니다. 그러나 새 배열의 요소 순서는 열거자가 반환하는 시퀀스를 기반으로 합니다. 결과 배열은 항상 하한이 0인 1차원입니다.

또한 많은 컬렉션 클래스에는 다음과 같은 기능이 포함되어 있습니다.

- **용량 및 개수 속성**

  컬렉션의 용량은 포함할 수 있는 요소의 수입니다. 컬렉션의 수는 실제로 포함된 요소의 수입니다. 일부 컬렉션은 용량 또는 개수 또는 둘 다를 숨깁니다.

  현재 용량에 도달하면 대부분의 컬렉션이 자동으로 용량으로 확장됩니다. 메모리가 다시 할당되고 요소가 이전 컬렉션에서 새 컬렉션으로 복사됩니다. 이 디자인은 컬렉션을 사용하는 데 필요한 코드를 줄입니다. 그러나 컬렉션의 성능에 부정적인 영향을 미칠 수 있습니다. 예를 들어 List<T>에서, Count이 Capacity보다 작으면 항목을 추가하는 것은 O(1) 작업입니다. 새 요소를 수용하기 위해 용량을 늘려야 하는 경우 항목을 추가하면 O(`n`) 작업이

nCount됩니다. 여러 재할당으로 인한 성능 저하를 방지하는 가장 좋은 방법은 초기 용량을 컬렉션의 예상 크기로 설정하는 것입니다.

A BitArray 는 특별한 경우입니다. 용량은 길이와 같으며 개수와 동일합니다.

- **일관된 하한**

  컬렉션의 하한은 첫 번째 요소의 인덱스입니다. 네임스페이스의 System.Collections 모든 인덱싱된 컬렉션은 하한이 0이므로 0으로 인덱싱됩니다. Array에는 기본적으로 하한이 0이지만 , 을 사용하여 Array.CreateInstance 클래스의 인스턴스를 만들 때 다른 하한을 정의할 수 있습니다.

- **여러 스레드에서 액세스하기 위한 동기화** (System.Collections 클래스에만 해당).

  네임스페이스 System.Collections의 제네릭이 아닌 컬렉션 형식은 동기화를 통해 일부 스레드 보안을 제공하며, 일반적으로 SyncRoot 및 IsSynchronized 멤버를 통해 노출됩니다. 이러한 컬렉션은 기본적으로 스레드로부터 안전하지 않습니다. 컬렉션에 확장 가능하고 효율적인 다중 스레드 액세스가 필요한 경우 네임스페이스의 System.Collections.Concurrent 클래스 중 하나를 사용하거나 변경할 수 없는 컬렉션을 사용하는 것이 좋습니다. 자세한 내용은 Thread-Safe 컬렉션을 참조하세요.

컬렉션 선택

일반적으로 제네릭 컬렉션을 사용해야 합니다. 다음 표에서는 이러한 시나리오에 사용할 수 있는 몇 가지 일반적인 컬렉션 시나리오 및 컬렉션 클래스에 대해 설명합니다. 제네릭 컬렉션을 처음으로 사용하는 경우 다음 표는 작업에 가장 적합한 제네릭 컬렉션을 선택하는 데 도움이 됩니다.

테이블 확장

하고 싶어요...	제네릭 컬렉션 옵션	제네릭이 아닌 컬렉션 옵션	스레드에 안전하거나 불변하는 컬렉션 옵션
키별 빠른 조회를 위해 항목을 키/값 쌍으로 저장	Dictionary<TKey,TValue>	Hashtable (키의 해시 코드를 기반으로 구성된 키/값 쌍의 컬렉션입니다.)	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
인덱스별 항목 액세스	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
아이템을 선입선출 방식으로 사용하세요	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
후입선출(LIFO) 방식으로First-Out 데이터를 사용하기	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
순차적으로 항목 액세스	LinkedList<T>	권장 사항 없음	권장 사항 없음
항목이 제거되거나 컬렉션에 추가될 때 알림을 받습니다. (구현 INotifyPropertyChanged 그리고 INotifyCollectionChanged)	ObservableCollection<T>	권장 사항 없음	권장 사항 없음
정렬된 컬렉션	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue>

하고 싶어요...	제네릭 컬렉션 옵션	제네릭이 아닌 컬렉션 옵션	스레드에 안전하거나 불변하는 컬렉션 옵션
			ImmutableSortedSet<T>
수학 함수 집합	HashSet<T>	권장 사항 없음	ImmutableHashSet<T>
	SortedSet<T>		ImmutableSortedSet<T>

컬렉션의 알고리즘 복잡성

컬렉션 클래스를 선택할 때 성능의 잠재적인 절충을 고려할 가치가 있습니다. 다음 표를 사용하여 알고리즘 복잡성에서 다양한 변경 가능한 컬렉션 형식이 해당 변경할 수 없는 컬렉션 형식과 비교하는 방법을 참조합니다. 변경할 수 없는 컬렉션 형식은 성능이 떨어지지만 불변성을 제공하는 경우가 많으며 이는 종종 유효한 비교 이점입니다.

테이블 확장

변경 가능	상환	최악의 경우	변경 불가능	복잡성
`Stack<T>.Push`	O(1)	O(`n`)	`ImmutableStack<T>.Push`	O(1)
`Queue<T>.Enqueue`	O(1)	O(`n`)	`ImmutableQueue<T>.Enqueue`	O(1)
`List<T>.Add`	O(1)	O(`n`)	`ImmutableList<T>.Add`	O(log `n`)
`List<T>.Item[Int32]`	O(1)	O(1)	`ImmutableList<T>.Item[Int32]`	O(log `n`)
`List<T>.Enumerator`	O(`n`)	O(`n`)	`ImmutableList<T>.Enumerator`	O(`n`)
`HashSet<T>.Add` 조회	O(1)	O(`n`)	`ImmutableHashSet<T>.Add`	O(log `n`)
`SortedSet<T>.Add`	O(log `n`)	O(`n`)	`ImmutableSortedSet<T>.Add`	O(log `n`)
`Dictionary<T>.Add`	O(1)	O(`n`)	`ImmutableDictionary<T>.Add`	O(log `n`)
`Dictionary<T>` 조회	O(1)	O(1) – 또는 엄밀하게 O(`n`)	`ImmutableDictionary<T>` 조회	O(log `n`)
`SortedDictionary<T>.Add`	O(log `n`)	O(`n` log `n`)	`ImmutableSortedDictionary<T>.Add`	O(log `n`)

`List<T>`는 `for` 루프 또는 `foreach` 루프를 사용하여 효율적으로 열거할 수 있습니다. 그러나 인덱서의 O(로그 `ImmutableList<T>`) 시간으로 인해 `for` 루프 안에서는 효율적이지 않습니다. `ImmutableList<T>`를 `foreach` 루프를 사용하여 열거하는 것은 `ImmutableList<T>`가 데이터를 저장할 때 `List<T>` 배열 대신 이진 트리를 사용하기 때문에 효율적입니다. 배열을 빠르게 인덱싱할 수 있는 반면, 원하는 인덱스가 있는 노드가 발견될 때까지 이진 트리를 아래로 이동해야 합니다.

`SortedSet<T>` 또한 둘 다 이진 트리를 사용하기 때문에 복잡성이 `ImmutableSortedSet<T>` 동일합니다. 중요한 차이점은 변경할 수 없는 이진 트리를 사용한다는 `ImmutableSortedSet<T>` 것입니다. `ImmutableSortedSet<T>` 또한 변형을 허용하는 클래스를 System.Collections.Immutable.ImmutableSortedSet<T>.Builder 제공하므로 불변성과 성능을 모두 가질 수 있습니다.

관련 문서

테이블 확장

제목	설명
컬렉션 클래스 선택	다양한 컬렉션을 설명하고 시나리오에 맞게 컬렉션을 선택하는 데 도움이 됩니다.
일반적으로 사용되는 컬렉션 형식	일반적으로 사용되는 제네릭 및 제네릭이 아닌 컬렉션 형식(예: System.Array, System.Collections.Generic.List<T> 및 System.Collections.Generic.Dictionary<TKey,TValue>)에 대해 설명합니다.
제네릭 컬렉션을 사용하는 경우	제네릭 컬렉션 형식의 사용에 대해 설명합니다.
컬렉션 내의 비교 및 정렬	컬렉션에서 같음 비교 및 정렬 비교의 사용에 대해 설명합니다.
정렬된 컬렉션 형식	정렬된 컬렉션 성능 및 특성을 설명합니다.
해시 테이블 및 사전 컬렉션 형식	제네릭 및 제네릭이 아닌 해시 기반 사전 형식의 기능을 설명합니다.
Thread-Safe 모음집	여러 스레드에서 안전하고 효율적인 동시 액세스를 지원하는 컬렉션 System.Collections.Concurrent.BlockingCollection<T> System.Collections.Concurrent.ConcurrentBag<T> 형식에 대해 설명합니다.
System.Collections.Immutable	변경할 수 없는 컬렉션을 소개하고 컬렉션 형식에 대한 링크를 제공합니다.

참고 문헌

- System.Array
- System.Collections
- System.Collections.Concurrent
- System.Collections.Generic
- System.Collections.Specialized
- System.Linq
- System.Collections.Immutable

Last updated on 2025. 10. 20.

컬렉션 클래스 선택

2025. 06. 17.

컬렉션 클래스를 신중하게 선택해야 합니다. 잘못된 형식을 사용하면 컬렉션 사용을 제한할 수 있습니다.

> ⓘ **중요**
>
> System.Collections 네임스페이스의 유형을 사용하지 마십시오. 컬렉션의 제네릭 및 동시 버전은 더 큰 형식 안전성 및 기타 개선 사항 때문에 권장됩니다.

다음 질문을 살펴보세요.

- 값이 검색된 후 요소가 일반적으로 삭제되는 순차 목록이 필요한가요?
  - 그렇다면 선입선출(FIFO) 동작이 필요한 경우 Queue 클래스 또는 Queue<T> 제네릭 클래스를 고려하세요. 마지막으로 들어온 것이 먼저 나가는(LIFO, Last-in, First-out) 동작이 필요한 경우, Stack 클래스 또는 Stack<T> 제네릭 클래스를 사용하는 것이 좋습니다. 여러 스레드에서 안전하게 액세스하려면 동시 실행 가능한 버전 ConcurrentQueue<T> 및 ConcurrentStack<T>을 사용하십시오. 불변성을 위해 불변 버전인 ImmutableQueue<T> 및 ImmutableStack<T>를 고려하세요.
  - 그렇지 않은 경우 다른 컬렉션을 사용하는 것이 좋습니다.
- FIFO, LIFO 또는 임의와 같은 특정 순서로 요소에 액세스해야 합니까?
  - Queue 클래스 및 Queue<T>ConcurrentQueue<T>ImmutableQueue<T> 제네릭 클래스는 모두 FIFO 액세스를 제공합니다. 자세한 내용은 Thread-Safe 컬렉션을 사용하는 경우를 참조하세요.
  - Stack 클래스 및 Stack<T>ConcurrentStack<T>ImmutableStack<T> 제네릭 클래스는 모두 LIFO 액세스를 제공합니다. 자세한 내용은 Thread-Safe 컬렉션을 사용하는 경우를 참조하세요.
  - 제네릭 클래스를 LinkedList<T> 사용하면 머리에서 꼬리로 또는 꼬리에서 머리까지 순차적으로 액세스할 수 있습니다.
- 인덱스별로 각 요소에 액세스해야 합니까?
  - ArrayList 및 StringCollection 클래스와 List<T> 제네릭 클래스는 0부터 시작하는 인덱스를 기반으로 해당 요소에 대한 액세스를 제공합니다. 불변성을 위해 불변의 제네릭 버전 ImmutableArray<T> 및 ImmutableList<T>을 고려하세요.

  - Hashtable, SortedList, ListDictionary, 및 StringDictionary 클래스와 Dictionary<TKey,TValue> 및 SortedDictionary<TKey,TValue> 제네릭 클래스는 요소의 키로 해당 요소에 대한 액세스를 제공합니다. 또한 다음과 같은 몇 가지 해당 형식의 변경할 수 없는 버전이 ImmutableHashSet<T>ImmutableDictionary<TKey,TValue>ImmutableSortedSet<T>있습니다. ImmutableSortedDictionary<TKey,TValue>

  - NameObjectCollectionBase 및 NameValueCollection 클래스와 KeyedCollection<TKey,TItem> 및 SortedList<TKey,TValue> 제네릭 클래스는 0부터 시작하는 인덱스나 요소의 키를 사용하여 그 요소에 액세스할 수 있습니다.

- 각 요소에는 하나의 값, 하나의 키와 하나의 값의 조합 또는 하나의 키와 여러 값의 조합이 포함하나요?

  - 하나의 값: IList 인터페이스 또는 IList<T> 제네릭 인터페이스를 기반으로 하는 컬렉션 중 하나를 사용하십시오. 변경할 수 없는 옵션의 경우 제네릭 인터페이스를 IImmutableList<T> 고려합니다.

  - 하나의 키와 하나의 값: IDictionary 인터페이스 또는 IDictionary<TKey,TValue> 제네릭 인터페이스를 기반으로 하는 컬렉션 중 하나를 사용합니다. 변경할 수 없는 옵션의 경우 IImmutableSet<T> 또는 IImmutableDictionary<TKey,TValue> 제네릭 인터페이스를 고려합니다.

  - 포함된 키가 있는 값 1개: 제네릭 클래스를 KeyedCollection<TKey,TItem> 사용합니다.

  - 하나의 키 및 여러 값: 클래스를 NameValueCollection 사용합니다.

- 요소를 입력한 방법과 다르게 정렬해야 합니까?

  - 클래스는 Hashtable 해당 해시 코드를 기준으로 해당 요소를 정렬합니다.

  - SortedList 클래스 및 SortedList<TKey,TValue>SortedDictionary<TKey,TValue> 제네릭 클래스는 해당 요소를 키별로 정렬합니다. 정렬 순서는 IComparer 클래스에 대한 SortedList 인터페이스의 구현 및 IComparer<T>와 SortedList<TKey,TValue> 제네릭 클래스에 대한 SortedDictionary<TKey,TValue> 제네릭 인터페이스의 구현을 기반으로 합니다. 두 제네릭 형식 중, SortedDictionary<TKey,TValue>은 SortedList<TKey,TValue>보다 성능이 더 좋고, SortedList<TKey,TValue>는 메모리를 덜 소비합니다.

  - ArrayList에서는 Sort 구현을 매개변수로 취하는 IComparer 메서드를 제공합니다. 제네릭 클래스인 List<T>에는 Sort 제네릭 인터페이스의 구현을 매개 변수로 받는 IComparer<T> 메서드가 있습니다.

- 빠른 검색 및 정보 검색이 필요한가요?
  - ListDictionary 는 작은 컬렉션(항목 10개 이하)보다 Hashtable 빠릅니다. 제네릭 클래스 Dictionary<TKey,TValue>는 제네릭 클래스 SortedDictionary<TKey,TValue>보다 더 빠

른 조회를 제공합니다. 다중 스레드 구현은 ConcurrentDictionary<TKey,TValue>. ConcurrentBag<T> 는 정렬되지 않은 데이터에 대해 빠른 다중 스레드 삽입을 제공합니다. 두 다중 스레드 형식에 대한 자세한 내용은 Thread-Safe 컬렉션을 사용하는 경우를 참조하세요.

- 문자열만 허용하는 컬렉션이 필요한가요?
  - StringCollection (IList을/를 기반으로 한) 및 StringDictionary (IDictionary을/를 기반으로 한)는 System.Collections.Specialized 네임스페이스에 있습니다.
  - 또한, 제네릭 형식 인수에 대해 System.Collections.Generic 클래스를 지정함으로써 String 네임스페이스의 제네릭 컬렉션 클래스를 강력한 형식의 문자열 컬렉션으로 사용할 수 있습니다. 예를 들어 변수를 List<String> 또는 Dictionary<String,String> 형식으로 선언할 수 있습니다.

LINQ to Objects 및 PLINQ

ko-KR: LINQ to Objects를 사용하면, 개발자는 개체 형식이 IEnumerable 또는 IEnumerable<T> 을 구현하는 한, LINQ 쿼리를 사용하여 메모리 내 개체에 액세스할 수 있습니다. LINQ 쿼리는 데이터에 액세스하기 위한 일반적인 패턴을 제공하고, 일반적으로 표준 `foreach` 루프보다 간결하고 읽기 가능하며, 필터링, 순서 지정 및 그룹화 기능을 제공합니다. 자세한 내용은 LINQ to Objects(C#) 및 LINQ to Objects(Visual Basic)를 참조하세요.

PLINQ는 다중 코어 컴퓨터를 보다 효율적으로 사용하여 많은 시나리오에서 더 빠른 쿼리 실행을 제공할 수 있는 LINQ to Objects의 병렬 구현을 제공합니다. 자세한 내용은 PLINQ(병렬 LINQ)를 참조하세요.

참고하십시오

- System.Collections
- System.Collections.Specialized
- System.Collections.Generic
- Thread-Safe 모음집

일반적으로 사용되는 컬렉션 형식

2025. 06. 17.

컬렉션 형식은 해시 테이블, 큐, 스택, 모음, 사전 및 목록과 같은 데이터를 수집하는 다양한 방법을 나타냅니다.

모든 컬렉션은 ICollection 인터페이스 또는 ICollection<T> 인터페이스를 직접 또는 간접적으로 기반으로 합니다. IList 및 IDictionary 제네릭 대응은 모두 이러한 두 인터페이스에서 파생됩니다.

IList 또는 ICollection에 직접 기반을 둔 컬렉션에서는 모든 요소에 값만 포함됩니다. 이러한 형식은 다음과 같습니다.

- Array
- ArrayList
- List<T>
- Queue
- ConcurrentQueue<T>
- Stack
- ConcurrentStack<T>
- LinkedList<T>

인터페이스를 기반으로 IDictionary 하는 컬렉션에서 모든 요소에는 키와 값이 모두 포함됩니다. 이러한 형식은 다음과 같습니다.

- Hashtable
- SortedList
- SortedList<TKey,TValue>
- Dictionary<TKey,TValue>
- ConcurrentDictionary<TKey,TValue>

클래스는 KeyedCollection<TKey,TItem> 값 내에 포함된 키가 있는 값 목록이므로 고유합니다. 결과적으로 목록과 사전처럼 동작합니다.

효율적인 다중 스레드 컬렉션 액세스가 필요한 경우 네임스페이스에서 System.Collections.Concurrent 제네릭 컬렉션을 사용합니다.

Queue 및 Queue<T> 클래스는 선입선출 목록을 제공합니다. Stack 및 Stack<T> 클래스는 후입선출 목록을 제공합니다.

강력한 입력

제네릭 컬렉션은 강력한 입력에 가장 적합한 솔루션입니다. 예를 들어, Int32가 아닌 다른 형식의 요소를 `List<Int32>` 컬렉션에 추가하면 컴파일 시 오류가 발생합니다. 그러나 언어에서 제네릭을 System.Collections 지원하지 않는 경우 네임스페이스에는 강력한 형식의 컬렉션 클래스를 만들기 위해 확장할 수 있는 추상 기본 클래스가 포함됩니다. 이러한 기본 클래스는 다음과 같습니다.

- CollectionBase
- ReadOnlyCollectionBase
- DictionaryBase

컬렉션의 차이

컬렉션은 요소를 저장, 정렬 및 비교하는 방법과 검색을 수행하는 방법에 따라 다릅니다.

SortedList 클래스와 SortedList<TKey,TValue> 제네릭 클래스는 Hashtable 클래스와 Dictionary<TKey,TValue> 제네릭 클래스의 정렬된 버전을 제공합니다.

모든 컬렉션은 0부터 시작하는 인덱스를 사용하지만, Array은 0이 아닌 인덱스를 사용하는 배열을 허용합니다.

SortedList 또는 KeyedCollection<TKey,TItem>의 요소에 키나 요소의 인덱스로 접근할 수 있습니다. Hashtable 또는 Dictionary<TKey,TValue>의 요소는 해당 요소의 키로만 접근할 수 있습니다.

컬렉션 형식에서 LINQ 사용

LINQ to Objects 기능은 구현 IEnumerable 하거나 IEnumerable<T>구현하는 모든 형식의 메모리 내 개체에 액세스하기 위한 일반적인 패턴을 제공합니다. LINQ 쿼리는 루프와 같은 `foreach` 표준 구문에 비해 몇 가지 이점이 있습니다.

- 간결하고 이해하기 쉽습니다.
- 데이터를 필터링, 정렬 및 그룹화할 수 있습니다.
- 성능을 향상시킬 수 있습니다.

자세한 내용은 LINQ to Objects(C#),LINQ to Objects(Visual Basic) 및 PLINQ(병렬 LINQ)를 참조하세요.

관련 항목

테이블 확장

제목	설명
컬렉션 및 데이터 구조	스택, 큐, 목록, 배열 및 사전을 포함하여 .NET에서 사용할 수 있는 다양한 컬렉션 형식에 대해 설명합니다.
해시 테이블 및 사전 컬렉션 형식	제네릭 및 비제네릭 해시 기반 사전 형식의 기능을 설명합니다.
정렬된 컬렉션 형식	목록 및 집합에 대한 정렬 기능을 제공하는 클래스에 대해 설명합니다.
제네릭	.NET에서 제공하는 제네릭 컬렉션, 대리자 및 인터페이스를 포함하여 제네릭 기능에 대해 설명합니다. C#, Visual Basic 및 Visual C++에 대한 기능 설명서와 리플렉션과 같은 지원 기술에 대한 링크를 제공합니다.

참고 문헌

System.Collections

System.Collections.Generic

System.Collections.ICollection

System.Collections.Generic.ICollection<T>

System.Collections.IList

System.Collections.Generic.IList<T>

System.Collections.IDictionary

System.Collections.Generic.IDictionary<TKey,TValue>

제네릭 컬렉션 사용 기준

2025. 06. 17.

제네릭 컬렉션을 사용하면 기본 컬렉션 형식에서 파생되고 형식별 멤버를 구현하지 않고도 형식 안전성의 자동 이점을 얻을 수 있습니다. 제네릭 컬렉션 형식은 일반적으로 컬렉션 요소가 값 형식일 경우 요소들을 박싱할 필요가 없으므로 해당 비제네릭 컬렉션 형식(및 비제네릭 기본 컬렉션 형식에서 파생된 형식)보다 성능이 뛰어납니다.

.NET Standard 1.0 이상을 대상으로 하는 프로그램의 경우 여러 스레드가 컬렉션에서 System.Collections.Concurrent 항목을 동시에 추가하거나 제거할 수 있는 경우 네임스페이스에서 제네릭 컬렉션 클래스를 사용합니다. 또한 불변성이 필요한 경우 네임스페이스의 제네릭 컬렉션 클래스를 System.Collections.Immutable 고려합니다.

다음 제네릭 형식은 기존 컬렉션 형식에 해당합니다.

- List<T>는 에 해당하는 제네릭 클래스입니다.ArrayList
- Dictionary<TKey,TValue>와 ConcurrentDictionary<TKey,TValue>는 Hashtable에 해당하는 제네릭 클래스입니다.
- Collection<T>는 에 해당하는 제네릭 클래스입니다.CollectionBase Collection<T> 은 기본 클래스로 사용할 수 있지만, 달리 CollectionBase추상적이지 않으므로 훨씬 쉽게 사용할 수 있습니다.
- ReadOnlyCollection<T>는 에 해당하는 제네릭 클래스입니다.ReadOnlyCollectionBase ReadOnlyCollection<T> 가 추상이 아니고 기존 List<T> 항목을 읽기 전용 컬렉션으로 쉽게 노출할 수 있는 생성자가 있습니다.
- , Queue<T>, ConcurrentQueue<T>, ImmutableQueue<T>ImmutableArray<T>및 SortedList<TKey,TValue> 제네릭 클래스는 ImmutableSortedSet<T>이름이 같은 각각의 비제네릭 클래스에 해당합니다.

추가 형식

제네릭 컬렉션 형식 중에서 일부는 대응하는 비제네릭 컬렉션 형식이 없습니다. 여기에는 다음과 같은 항목이 포함됩니다.

- LinkedList<T> 는 O(1) 삽입 및 제거 작업을 제공하는 범용 연결된 목록입니다.
- SortedDictionary<TKey,TValue> 은 O(log `n`) 삽입 및 검색 작업이 있는 정렬된 사전이므로 유용한 대안이 됩니다 SortedList<TKey,TValue>.

- KeyedCollection<TKey,TItem> 는 자체 키를 포함하는 개체를 저장하는 방법을 제공하는 목록과 사전 간의 하이브리드입니다.
- BlockingCollection<T> 는 경계 및 차단 기능이 있는 컬렉션 클래스를 구현합니다.
- ConcurrentBag<T> 는 순서가 지정되지 않은 요소의 빠른 삽입 및 제거를 제공합니다.

변경할 수 없는 빌더

앱에서 불변성 기능을 원하는 경우 네임스페이 System.Collections.Immutable 스는 사용할 수 있는 제네릭 컬렉션 형식을 제공합니다. 변경할 수 없는 모든 컬렉션 형식은 여러 변형을 수행할 때 성능을 최적화할 수 있는 클래스를 제공합니다 `Builder` . 클래스는 `Builder` 변경 가능한 상태로 작업을 일괄 처리합니다. 모든 변형이 완료되면 메서드를 호출 `ToImmutable` 하여 모든 노드를 "고정"하고 변경할 수 없는 제네릭 컬렉션(예: .)을 ImmutableList<T>만듭니다.

제네릭이 아닌 `Builder` 메서드를 호출하여 객체를 만들 수 있습니다. 인스턴스`Builder`에서 `ToImmutable()`을(를) 호출할 수 있습니다. 마찬가지로, `Immutable*` 컬렉션에서 `ToBuilder()`을 호출하여 제네릭 불변 컬렉션으로부터 빌더 인스턴스를 생성할 수 있습니다. 다음은 다양한 `Builder` 형식입니다.

- ImmutableArray<T>.Builder
- ImmutableDictionary<TKey,TValue>.Builder
- ImmutableHashSet<T>.Builder
- ImmutableList<T>.Builder
- ImmutableSortedDictionary<TKey,TValue>.Builder
- ImmutableSortedSet<T>.Builder

LINQ to Objects

LINQ to Objects 기능을 사용하면 개체 형식이 System.Collections.IEnumerable 또는 System.Collections.Generic.IEnumerable<T> 인터페이스를 구현하고 있는 한 LINQ 쿼리를 사용하여 메모리 내 개체에 액세스할 수 있습니다. LINQ 쿼리는 데이터에 액세스하기 위한 일반적인 패턴을 제공합니다. 는 일반적으로 표준 `foreach` 루프보다 간결하고 읽기 가능하며 필터링, 순서 지정 및 그룹화 기능을 제공합니다. LINQ 쿼리는 성능을 향상시킬 수도 있습니다. 자세한 내용은 LINQ to Objects(C#),LINQ to Objects(Visual Basic) 및 PLINQ(병렬 LINQ)를 참조하세요.

추가 기능

일부 제네릭 형식에는 비제네릭 컬렉션 형식에서 찾을 수 없는 기능이 있습니다. 예를 들어 List<T> 비제너릭 ArrayList 클래스에 해당하는 클래스에는 목록 검색을 위한 메서드를 지정할 수 있는 대리자, 목록의 각 요소에서 작동하는 메서드를 나타내는 대리자, Predicate<T> 형식 간

변환을 정의할 수 있는 대리자 등 Action<T>Converter<TInput,TOutput> 제네릭 대리자를 허용하는 여러 메서드가 있습니다.

이 List<T> 클래스를 사용하면 목록을 정렬하고 검색하기 위한 고유한 IComparer<T> 제네릭 인터페이스 구현을 지정할 수 있습니다. 클래스 SortedDictionary<TKey,TValue>와 SortedList<TKey,TValue>도 이 기능을 갖추고 있습니다. 또한 이러한 클래스를 사용하면 컬렉션을 만들 때 비교자를 지정할 수 있습니다. 비슷한 방식으로 Dictionary<TKey,TValue> 및 KeyedCollection<TKey,TItem> 클래스는 고유한 같음 비교자를 지정할 수 있게 합니다.

참고하십시오

- 컬렉션 및 데이터 구조
- 일반적으로 사용되는 컬렉션 형식
- 제네릭

컬렉션 내에서 비교 및 정렬

2025. 06. 17.

클래스는 System.Collections 제거할 요소를 검색하거나 키 및 값 쌍의 값을 반환하는지 여부에 관계없이 컬렉션 관리와 관련된 거의 모든 프로세스에서 비교를 수행합니다.

컬렉션은 일반적으로 같음 비교자 및/또는 순서 비교자를 사용합니다. 비교에는 두 개의 구문이 사용됩니다.

동등성 확인

와 같은 `Contains` IndexOfLastIndexOf메서드는 `Remove` 컬렉션 요소에 대해 같음 비교자를 사용합니다. 컬렉션이 제네릭이면 다음 지침에 따라 항목이 같음으로 비교됩니다.

- T 형식이 제네릭 인터페이스 IEquatable<T>을(를) 구현할 경우, 해당 인터페이스의 Equals 메서드가 같음 비교자로 사용됩니다.
- T 형식이 IEquatable<T>을(를) 구현하지 않으면 Object.Equals이(가) 사용됩니다.

또한 사전 컬렉션에 대한 일부 생성자 오버로드는 키의 동일성을 비교하기 위한 구현을 IEqualityComparer<T>로 허용합니다. 예를 들어 Dictionary<TKey,TValue> 생성자를 참조하십시오.

정렬 순서 결정

`BinarySearch` 및 `Sort`과 같은 메서드는 컬렉션 요소에 대한 순서 비교자를 사용합니다. 비교는 컬렉션의 요소 간 또는 요소와 지정된 값 사이일 수 있습니다. 개체를 비교할 때는 `default comparer`과/와 `explicit comparer`의 개념이 있습니다.

기본 비교자는 비교되는 개체 중 하나 이상을 사용하여 **IComparable** 인터페이스를 구현합니다. 목록 컬렉션의 값으로 사용되거나 사전 컬렉션의 키로 사용되는 모든 클래스에서 **IComparable**을 구현하는 것이 좋습니다. 제네릭 컬렉션의 경우 같음 비교는 다음에 따라 결정됩니다.

- T 형식이 제네릭 인터페이스를 System.IComparable<T> 구현하는 경우 기본 비교자는 해당 인터페이스의 IComparable<T>.CompareTo(T) 메서드입니다.
- T 형식이 제네릭 System.IComparable 이 아닌 인터페이스를 구현하는 경우 기본 비교자는 해당 인터페이스의 IComparable.CompareTo(Object) 메서드입니다.
- T 형식이 두 인터페이스 중 하나를 구현하지 않는 경우 기본 비교자가 없으며 비교자 또는 비교 대리자를 명시적으로 제공해야 합니다.

명시적 비교를 제공하기 위해 일부 메서드는 **IComparer 구현을** 매개 변수로 허용합니다. 예를 들어 메서드는 List<T>.Sort 구현을 System.Collections.Generic.IComparer<T> 허용합니다.

시스템의 현재 문화권 설정은 컬렉션 내의 비교 및 정렬에 영향을 줄 수 있습니다. 기본적으로 **컬렉션** 클래스의 비교 및 정렬은 문화권을 구분합니다. 문화권 설정을 무시하여 일관된 비교 및 정렬 결과를 얻으려면 InvariantCulture를 허용하는 멤버 오버로드와 함께 CultureInfo를 사용하십시오. 자세한 내용은 컬렉션에서 문화권을 구분하지 않는 문자열 작업 수행 및 배열에서 문화권을 구분하지 않는 문자열 작업 수행을 참조하세요.

같음 및 정렬 예제

다음 코드에서는 간단한 비즈니스 개체의 IEquatable<T> 구현 및 IComparable<T> 구현을 보여 줍니다. 또한 개체가 목록에 저장되고 정렬될 때 Sort() 메서드를 호출하면, 형식 `Part`에 대한 기본 비교자를 사용하고 익명 메서드를 사용하여 구현된 Sort(Comparison<T>) 메서드를 볼 수 있습니다.

C#

```
using System;
using System.Collections.Generic;

// Simple business object. A PartId is used to identify the
// type of part but the part name can change.
public class Part : IEquatable<Part>, IComparable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString() =>
        $"ID: {PartId}   Name: {PartName}";

    public override bool Equals(object obj) =>
        (obj is Part part)
                ? Equals(part)
                : false;

    public int SortByNameAscending(string name1, string name2) =>
        name1?.CompareTo(name2) ?? 1;

    // Default comparer for Part type.
    // A null value means that this object is greater.
    public int CompareTo(Part comparePart) =>
        comparePart == null ? 1 : PartId.CompareTo(comparePart.PartId);

    public override int GetHashCode() => PartId;

    public bool Equals(Part other) =>
        other is null ? false : PartId.Equals(other.PartId);
```

```
    // Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        var parts = new List<Part>
        {
            // Add parts to the list.
            new Part { PartName = "regular seat", PartId = 1434 },
            new Part { PartName = "crank arm", PartId = 1234 },
            new Part { PartName = "shift lever", PartId = 1634 },
            // Name intentionally left null.
            new Part { PartId = 1334 },
            new Part { PartName = "banana seat", PartId = 1444 },
            new Part { PartName = "cassette", PartId = 1534 }
        };

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        parts.ForEach(Console.WriteLine);

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        parts.ForEach(Console.WriteLine);

        // This shows calling the Sort(Comparison<T> comparison) overload using
        // a lambda expression as the Comparison<T> delegate.
        // This method treats null as the lesser of two values.
        parts.Sort((Part x, Part y) =>
            x.PartName == null && y.PartName == null
                ? 0
                : x.PartName == null
                    ? -1
                    : y.PartName == null
                        ? 1
                        : x.PartName.CompareTo(y.PartName));

        Console.WriteLine("\nAfter sort by name:");
        parts.ForEach(Console.WriteLine);

        /*

            Before sort:
        ID: 1434   Name: regular seat
        ID: 1234   Name: crank arm
        ID: 1634   Name: shift lever
```

```
        ID: 1334   Name:
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette

        After sort by part number:
        ID: 1234   Name: crank arm
        ID: 1334   Name:
        ID: 1434   Name: regular seat
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette
        ID: 1634   Name: shift lever

        After sort by name:
        ID: 1334   Name:
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette
        ID: 1234   Name: crank arm
        ID: 1434   Name: regular seat
        ID: 1634   Name: shift lever

         */
    }
}
```

참고하십시오

- IComparer
- IEquatable<T>
- IComparer<T>
- IComparable
- IComparable<T>

정렬된 컬렉션 형식

2025. 06. 17.

System.Collections.SortedList 클래스, System.Collections.Generic.SortedList<TKey,TValue> 제네릭 클래스 및 System.Collections.Generic.SortedDictionary<TKey,TValue> 제네릭 클래스는 Hashtable 클래스와 Dictionary<TKey,TValue> 제네릭 클래스와 유사하게 IDictionary 인터페이스를 구현한다는 점에서 비슷하지만, 그 들은 키별로 정렬된 순서로 요소를 유지 관리하며 해시 테이블의 O(1) 삽입 및 검색 특성이 없습니다. 세 클래스에는 다음과 같은 몇 가지 기능이 있습니다.

- 세 클래스 모두 인터페이스를 구현합니다 System.Collections.IDictionary . 또한 두 개의 제네릭 클래스는 제네릭 인터페이스를 System.Collections.Generic.IDictionary<TKey,TValue> 구현합니다.
- 각 요소는 열거용 키/값 쌍입니다.

  > **참고**
  >
  > 제네릭이 아닌 SortedList 클래스는 열거될 때 DictionaryEntry 개체를 반환하지만, 반면 두 제네릭 형식은 KeyValuePair<TKey,TValue> 개체를 반환합니다.

- 요소는 비제네릭인 경우에는 System.Collections.IComparer 구현에 따라, 두 제네릭 클래스의 경우에는 SortedList 구현에 따라 정렬됩니다.
- 각 클래스는 키만 포함하거나 값만 포함하는 컬렉션을 반환하는 속성을 제공합니다.

다음 표에서는 정렬된 두 목록 클래스와 SortedDictionary<TKey,TValue> 클래스와의 몇 가지 차이점을 나열합니다.

테이블 확장

SortedList 비제네릭 클래스 및 SortedList<TKey,TValue> 제네릭 클래스	**SortedDictionary<TKey,TValue> 제네릭 클래스**
키와 값을 반환하는 속성이 인덱싱되어 효율적인 인덱싱된 검색이 가능합니다.	인덱싱된 검색이 없습니다.
검색은 O(로그 `n`)입니다.	검색은 O(로그 `n`)입니다.
삽입 및 제거는 일반적으로 O(`n`)이지만 삽입은 정렬 순서에 이미 있는 데이터의 O(로그 `n`)이므로 각 요소가 목록의 끝에 추가됩니다. (크기 조정이 필요하지 않다고 가정합니다.)	삽입 및 제거는 O(로그 `n`)입니다.

SortedList 비제네릭 클래스 및 SortedList<TKey,TValue> 제네릭 클래스	**SortedDictionary<TKey,TValue> 제네릭 클래스**
SortedDictionary<TKey,TValue>보다 적은 메모리를 사용합니다.	제네릭 클래스 SortedList 및 비제네릭 클래스 SortedList<TKey,TValue>보다 더 많은 메모리를 사용합니다.

여러 스레드에서 동시에 액세스할 수 있어야 하는 정렬된 목록 또는 사전의 경우 파생되는 클래스 ConcurrentDictionary<TKey,TValue>에 정렬 논리를 추가할 수 있습니다. 불변성을 고려할 때 다음과 같은 해당 변경할 수 없는 형식은 유사한 정렬 의미 체계를 따릅니다. ImmutableSortedSet<T>ImmutableSortedDictionary<TKey,TValue>

> ⓘ **참고**
>
> 사용자 고유의 키(예: 직원 ID 번호가 포함된 직원 레코드)를 포함하는 값의 경우 제네릭 클래스에서 KeyedCollection<TKey,TItem> 파생하여 목록의 특성과 사전의 일부 특징이 있는 키 컬렉션을 만들 수 있습니다.

.NET Framework 4부터 SortedSet<T> 클래스는 삽입, 삭제 및 검색 후 정렬된 순서로 데이터를 유지하는 자체 균형 트리를 제공합니다. 이 클래스와 HashSet<T> 클래스는 ISet<T> 인터페이스를 구현합니다.

참고하십시오

- System.Collections.IDictionary
- System.Collections.Generic.IDictionary<TKey,TValue>
- ConcurrentDictionary<TKey,TValue>
- 일반적으로 사용되는 컬렉션 형식

해시 테이블 및 사전 컬렉션 형식

2025. 06. 17.

System.Collections.Hashtable 클래스 및 System.Collections.Generic.Dictionary<TKey,TValue>System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue> 제네릭 클래스는 인터페이스를 구현합니다 System.Collections.IDictionary. 제네릭 클래스는 Dictionary<TKey,TValue> 제네릭 인터페이스도 구현합니다 IDictionary<TKey,TValue> . 따라서 이러한 컬렉션의 각 요소는 키 및 값 쌍입니다.

개체는 Hashtable 컬렉션의 요소를 포함하는 버킷으로 구성됩니다. 버킷은 대부분의 컬렉션보다 더 쉽고 빠르게 검색할 수 있는 요소 Hashtable의 가상 하위 그룹입니다. 각 버킷은 해시 함수를 사용하여 생성되고 요소의 키를 기반으로 하는 해시 코드와 연결됩니다.

제네릭 HashSet<T> 클래스는 고유 요소를 포함하는 순서가 지정되지 않은 컬렉션입니다.

해시 함수는 키를 기반으로 숫자 해시 코드를 반환하는 알고리즘입니다. 키는 저장되는 개체의 일부 속성 값입니다. 해시 함수는 항상 동일한 키에 대해 동일한 해시 코드를 반환해야 합니다. 해시 함수는 서로 다른 두 키에 대해 동일한 해시 코드를 생성할 수 있지만, 각 고유 키에 대해 고유한 해시 코드를 생성하는 해시 함수는 해시 테이블에서 요소를 검색할 때 성능이 향상됩니다.

요소 Hashtable 로 사용되는 각 개체는 메서드의 GetHashCode 구현을 사용하여 자체에 대한 해시 코드를 생성할 수 있어야 합니다. 그러나 Hashtable의 모든 요소에 대해 해시 함수를 지정하려면 Hashtable 구현을 매개변수 중 하나로 받는 IHashCodeProvider 생성자를 또한 사용할 수 있습니다.

개체가 Hashtable에 추가되면, 해당 개체의 해시 코드와 일치하는 해시 코드에 연결된 버킷에 저장됩니다. 값을 검색할 때 해당 값에 대한 Hashtable해시 코드가 생성되고 해당 해시 코드와 연결된 버킷이 검색됩니다.

예를 들어 문자열에 대한 해시 함수는 문자열에 있는 각 문자의 ASCII 코드를 가져와 해시 코드를 생성하기 위해 함께 추가할 수 있습니다. 문자열 "picnic"에는 문자열 "basket"의 해시 코드와 다른 해시 코드가 있습니다. 따라서 문자열 "피크닉"과 "바구니"는 다른 버킷에 있을 것입니다. 반면, "스트레스"와 "디저트"는 동일한 해시 코드를 가지며 동일한 버킷에 있습니다.

Dictionary<TKey,TValue> 및 ConcurrentDictionary<TKey,TValue> 클래스는 클래스와 동일한 기능을 갖습니다Hashtable. 특정 유형의 Dictionary<TKey,TValue>는 Object 외의 경우, 값 유형에 관한 Hashtable보다 더 나은 성능을 제공합니다. 이는 Hashtable의 요소가 Object 형식이기 때문입니다. 따라서 일반적으로 값 형식을 저장하거나 검색할 때 박싱과 언박싱이 발생합니다. ConcurrentDictionary<TKey,TValue> 여러 스레드가 컬렉션에 동시에 액세스할 수 있는 경우 클래스를 사용해야 합니다.

참고하십시오

- Hashtable
- IDictionary
- IHashCodeProvider
- Dictionary<TKey,TValue>
- System.Collections.Generic.IDictionary<TKey,TValue>
- System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>
- 일반적으로 사용되는 컬렉션 형식

스레드로부터 안전한 컬렉션

2025. 10. 21.

네임스페이스에는 System.Collections.Concurrent 스레드로부터 안전하고 확장 가능한 여러 컬렉션 클래스가 포함되어 있습니다. 여러 스레드는 사용자 코드에서 추가 동기화를 요구하지 않고도 이러한 컬렉션에서 항목을 안전하고 효율적으로 추가하거나 제거할 수 있습니다. 새 코드를 작성할 때 동시 컬렉션 클래스를 사용하여 컬렉션에 여러 스레드를 동시에 작성합니다. 공유 컬렉션에서만 읽는 경우 네임스페이스의 클래스를 System.Collections.Generic 사용합니다.

System.Collections 및 System.Collections.Generic

System.Collections 네임스페이스의 컬렉션 클래스에는 ArrayList 및 Hashtable가 포함되어 있습니다. 이러한 클래스는 컬렉션 주위를 감싸는 스레드 안전한 래퍼를 반환하는 `Synchronized` 속성을 통해 일부 스레드 보안을 제공합니다. 래퍼는 모든 추가 또는 제거 작업에서 전체 컬렉션을 잠그는 작업을 수행합니다. 따라서 컬렉션에 액세스하려는 각 스레드는 해당 순서가 하나의 잠금을 획득할 때까지 기다려야 합니다. 이 프로세스는 확장할 수 없으며 큰 컬렉션의 성능이 크게 저하될 수 있습니다. 또한, 디자인은 경합 조건에서 보호되지 않습니다. 자세한 내용은 제네릭 컬렉션의 동기화를 참조하세요.

System.Collections.Generic 네임스페이스의 컬렉션 클래스에는 List<T> 및 Dictionary<TKey,TValue>가 포함되어 있습니다. 이 클래스들은 System.Collections 클래스에 비해 향상된 형식 안전성 및 성능을 제공합니다. 그러나 클래스는 System.Collections.Generic 스레드 동기화를 제공하지 않습니다. 사용자 코드는 여러 스레드에서 항목을 동시에 추가하거나 제거할 때 모든 동기화를 제공해야 합니다.

네임스페이스에서 동시 컬렉션 클래스는 System.Collections.Concurrent 형식 안전성과 보다 효율적이고 완전한 스레드 보안을 제공하기 때문에 사용하는 것이 좋습니다.

세분화된 잠금 및 잠금 없는 메커니즘

일부 동시 컬렉션 형식은 SpinLock, SpinWait, SemaphoreSlim, CountdownEvent 같은 간단한 동기화 메커니즘을 사용합니다. 이러한 동기화 형식은 일반적으로 스레드를 실제 상태로 전환하기 전에 짧은 기간 동안 `Wait`를 사용합니다. 대기 시간이 짧을 것으로 예상되는 경우 회전은 대기보다 계산 비용이 훨씬 적게 들며, 여기에는 비용이 많이 드는 커널 전환이 포함됩니다. 회전을 사용하는 컬렉션 클래스의 경우 이 효율성은 여러 스레드가 높은 속도로 항목을 추가하고 제거할 수 있음을 의미합니다. 회전 및 차단에 대한 자세한 내용은 SpinLock 및 SpinWait을 참조하세요.

및 ConcurrentQueue<T> 클래스는 ConcurrentStack<T> 잠금을 전혀 사용하지 않습니다. 대신 스레드 안전을 달성하기 위해 Interlocked 작업에 의존합니다.

> **참고**
>
> 동시에 존재하는 컬렉션 클래스는 ICollection을 지원하므로, IsSynchronized 및 SyncRoot 속성에 대한 구현을 제공하며, 이러한 속성은 관련이 없습니다. `IsSynchronized`는 항상 `false`를 반환하고, `SyncRoot`는 항상 `null`입니다 (`Nothing`는 Visual Basic에서).

다음 표에서는 네임스페이스의 컬렉션 형식을 System.Collections.Concurrent 나열합니다.

테이블 확장

유형	설명
BlockingCollection<T>	IProducerConsumerCollection<T>을(를) 구현하는 모든 형식에 대한 경계 및 제한 기능을 제공합니다. 자세한 내용은 BlockingCollection 개요를 참조하세요.
ConcurrentDictionary<TKey,TValue>	키-값 쌍 사전의 스레드 안전한 구현.
ConcurrentQueue<T>	FIFO(선입선출) 큐의 스레드 안전한 구현입니다.
ConcurrentStack<T>	후입선출 스택의 스레드 안전한 구현입니다.
ConcurrentBag<T>	정렬되지 않은 요소 컬렉션의 스레드 안전한 구현입니다.
IProducerConsumerCollection<T>	해당 형식이 `BlockingCollection`에서 사용되도록 구현해야 하는 인터페이스입니다.

관련 문서

테이블 확장

제목	설명
BlockingCollection의 개요	형식에서 제공하는 BlockingCollection<T> 기능을 설명합니다.
방법: ConcurrentDictionary에서 항목 추가 및 제거	에서 요소를 추가하고 제거하는 방법을 설명합니다. ConcurrentDictionary<TKey,TValue>
방법: BlockingCollection에서 개별적으로 항목 추가 및 가져오기	읽기 전용 열거자를 사용하지 않고 차단 컬렉션에서 항목을 추가하고 검색하는 방법을 설명합니다.
방법: 컬렉션에 경계 및 차단 기능 추가	컬렉션 클래스를 컬렉션의 기본 스토리지 메커니즘 IProducerConsumerCollection<T> 으로 사용하는 방법을 설명합니다.

제목	설명
방법: ForEach를 사용하여 BlockingCollection에서 항목 제거	`foreach`(`For Each` Visual Basic에서) 차단된 컬렉션의 모든 항목을 제거하는 방법을 설명합니다.
방법: 파이프라인에서 차단 컬렉션 배열 사용	여러 차단 컬렉션을 동시에 사용하여 파이프라인을 구현하는 방법을 설명합니다.
방법: ConcurrentBag을 사용하여 개체 풀 만들기	새 항목을 지속적으로 만드는 대신 개체를 다시 사용할 수 있는 시나리오에서 동시 모음을 사용하여 성능을 향상시키는 방법을 보여 줍니다.

참고 문헌

- System.Collections.Concurrent

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

대리자 및 람다 식

2025. 06. 17.

대리자는 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타내는 형식을 정의합니다. 매개 변수 목록과 반환 형식이 일치하는 메서드(정적 또는 인스턴스)를 해당 형식의 변수에 할당한 다음 직접 호출하거나(적절한 인수를 사용하여) 인수 자체로 다른 메서드에 전달한 다음 호출할 수 있습니다. 다음 예제에서는 대리자 사용을 보여 줍니다.

C#

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- 줄은 `public delegate string Reverse(string s);` 문자열 매개 변수를 사용한 다음 문자열 매개 변수를 반환하는 메서드의 대리자 형식을 만듭니다.
- `static string ReverseString(string s)` 정의된 대리자 형식과 정확히 동일한 매개 변수 목록 및 반환 형식을 포함하는 메서드는 대리자를 구현합니다.
- 줄은 `Reverse rev = ReverseString;` 해당 대리자 형식의 변수에 메서드를 할당할 수 있음을 보여줍니다.
- 이 줄은 `Console.WriteLine(rev("a string"));` 대리자 형식의 변수를 사용하여 대리자를 호출하는 방법을 보여 줍니다.

개발 프로세스를 간소화하기 위해 .NET에는 프로그래머가 재사용할 수 있고 새 형식을 만들 필요가 없는 대리자 형식 집합이 포함되어 있습니다. 이러한 형식은 `Func<>`, `Action<>`, `Predicate<>`이며, 새 대리자 형식을 정의하지 않고도 사용할 수 있습니다. 사용하려는 방식과 관련이 있는 세 가지 형식 간에는 몇 가지 차이점이 있습니다.

- `Action<>` 는 대리자의 인수를 사용하여 작업을 수행해야 하는 경우에 사용됩니다. 캡슐화하는 메서드는 값을 반환하지 않습니다.
- `Func<>` 는 일반적으로 변환이 있을 때 사용됩니다. 즉, 대리자의 인수를 다른 결과로 변환해야 합니다. 프로젝션이 좋은 예입니다. 캡슐화하는 메서드는 지정된 값을 반환합니다.
- `Predicate<>` 는 인수가 대리자의 조건을 충족하는지 확인해야 할 때 사용됩니다. `Func<T, bool>`로도 작성할 수 있으며, 이는 메서드가 불린 값을 반환함을 의미합니다.

이제 위의 예제를 가져와서 사용자 지정 형식 대신 대리자를 `Func<>` 사용하여 다시 작성할 수 있습니다. 프로그램은 정확히 동일하게 계속 실행됩니다.

C#

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

이 간단한 예제에서는 `Main` 메서드 외부에서 메서드를 정의하는 것이 약간 불필요한 것처럼 보입니다. .NET Framework 2.0에서는 추가 형식이나 메서드를 지정하지 않고도 "인라인" 대리자를 만들 수 있는 *익명 대리*자의 개념을 도입했습니다.

다음 예제에서 익명 대리자는 목록을 짝수로 필터링한 다음 콘솔에 출력합니다.

C#

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
```

```
            list.Add(i);
        }

        List<int> result = list.FindAll(
          delegate (int no)
          {
              return (no % 2 == 0);
          }
        );

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

보시다시피 대리자의 본문은 다른 대리자처럼 여러 표현의 집합입니다. 그러나 별도의 정의가 아니라 메서드 호출 에서 List<T>.FindAll로 도입했습니다.

그러나 이 방법을 사용하더라도 버릴 수 있는 코드는 여전히 많습니다. *람다 식이* 활용되는 곳입니다. 람다 식 또는 간단히 말해서 "람다"는 C# 3.0에서 LINQ(언어 통합 쿼리)의 핵심 구성 요소 중 하나로 도입되었습니다. 대리자를 사용하기 위한 보다 편리한 구문일 뿐입니다. 매개 변수 목록 및 메서드 본문을 선언하지만 대리자에게 할당되지 않는 한 자체의 공식 ID가 없습니다. 대리자와 달리, 이들을 이벤트 등록 시 또는 다양한 LINQ 절 및 메서드에서 직접 할당할 수 있습니다.

람다 식은 대리자를 지정하는 또 다른 방법이기 때문에 위 샘플을 다시 작성하여 익명 대리자 대신 람다 식을 사용할 수 있어야 합니다.

C#

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(i => i % 2 == 0);

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
```

```
    }
}
```

앞의 예제에서 사용된 람다 식은 .입니다 `i => i % 2 == 0`. 다시 말하지만, 대리자를 사용하기 위한 편리한 구문일 뿐입니다. 내부에서 일어나는 작업은 익명 대리자에서 발생하는 작업과 유사합니다.

다시 말하지만, 람다는 대리자일 뿐입니다. 즉, 다음 코드 조각과 같이 문제 없이 이벤트 처리기로 사용할 수 있습니다.

C#

```
public MainWindow()
{
    InitializeComponent();

    Loaded += (o, e) =>
    {
        this.Title = "Loaded";
    };
}
```

`+=` 이 컨텍스트의 연산자는 이벤트를 구독하는 데 사용됩니다. 자세한 내용은 이벤트구독 및 구독 취소하는 방법을 참조하세요.

추가 읽기 및 리소스

- 대표자
- 람다 식

System.Delegate.CreateDelegate 메서드들

아티클 • 2025. 04. 03.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

메서드는 CreateDelegate 지정된 형식의 대리자를 만듭니다.

CreateDelegate(Type, MethodInfo) 메서드

이 메서드 오버로드는 CreateDelegate(Type, MethodInfo, Boolean) 메서드 오버로드를 호출하고 `true`에 대해 `throwOnBindFailure`를 지정하는 것과 같습니다.

예시

이 섹션에는 두 가지 코드 예제가 포함되어 있습니다. 첫 번째 예제에서는 이 메서드 오버로드를 사용하여 만들 수 있는 두 가지 유형의 대리자를 보여 줍니다. 인스턴스 메서드를 통해 열고 정적 메서드를 통해 엽니다.

두 번째 코드 예제에서는 호환되는 매개 변수 형식 및 반환 형식을 보여 줍니다.

예제 1

다음 코드 예제에서는 메서드의 이 오버로드 CreateDelegate 를 사용하여 대리자를 만들 수 있는 두 가지 방법을 보여 줍니다.

> **참고**
>
> CreateDelegate 메서드에는 MethodInfo을(를) 지정하지만 첫 번째 인수를 지정하지 않는 두 가지 오버로드가 있습니다. 이들 두 오버로드는 기능은 동일하지만, 하나는 바인딩 실패 시 예외를 던질지 여부를 선택할 수 있고, 다른 하나는 항상 예외를 던진다는 차이점이 있습니다. 이 코드 예제에서는 두 오버로드를 모두 사용합니다.

이 예제에서는 클래스 `C`를 선언하고 정적 메서드 `M2`와 인스턴스 메서드 `M1`를 포함합니다. 두 개의 대리자 형식을 선언하는데, `D1`는 `C`의 인스턴스와 문자열을 받고, `D2`는 문자열을 받습니다.

명명된 `Example` 두 번째 클래스에는 대리자를 만드는 코드가 포함됩니다.

- 열려 있는 인스턴스 메서드를 나타내는 형식 `D1`의 대리자가 인스턴스 메서드 `M1`에 대해 만들어집니다. 대리자를 호출할 때 인스턴스를 전달해야 합니다.
- 정적 `M2`메서드에 대해 열려 있는 정적 메서드를 나타내는 형식`D2`의 대리자가 만들어집니다.

C#

```
using System;
using System.Reflection;

// Declare three delegate types for demonstrating the combinations
// of static versus instance methods and open versus closed
// delegates.
//
public delegate void D1(C c, string s);
public delegate void D2(string s);
public delegate void D3();

// A sample class with an instance method and a static method.
//
public class C
{
    private int id;
    public C(int id) { this.id = id; }

    public void M1(string s)
    {
        Console.WriteLine("Instance method M1 on C:  id = {0}, s = {1}",
            this.id, s);
    }

    public static void M2(string s)
    {
        Console.WriteLine($"Static method M2 on C:  s = {s}");
    }
}

public class Example2
{
    public static void Main()
    {
        C c1 = new C(42);

        // Get a MethodInfo for each method.
        //
        MethodInfo mi1 = typeof(C).GetMethod("M1",
            BindingFlags.Public | BindingFlags.Instance);
        MethodInfo mi2 = typeof(C).GetMethod("M2",
            BindingFlags.Public | BindingFlags.Static);

        D1 d1;
        D2 d2;
        D3 d3;
```

```
        Console.WriteLine("\nAn instance method closed over C.");
        // In this case, the delegate and the
        // method must have the same list of argument types; use
        // delegate type D2 with instance method M1.
        //
        Delegate test =
            Delegate.CreateDelegate(typeof(D2), c1, mi1, false);

        // Because false was specified for throwOnBindFailure
        // in the call to CreateDelegate, the variable 'test'
        // contains null if the method fails to bind (for
        // example, if mi1 happened to represent a method of
        // some class other than C).
        //
        if (test != null)
        {
            d2 = (D2)test;

            // The same instance of C is used every time the
            // delegate is invoked.
            d2("Hello, World!");
            d2("Hi, Mom!");
        }

        Console.WriteLine("\nAn open instance method.");
        // In this case, the delegate has one more
        // argument than the instance method; this argument comes
        // at the beginning, and represents the hidden instance
        // argument of the instance method. Use delegate type D1
        // with instance method M1.
        //
        d1 = (D1)Delegate.CreateDelegate(typeof(D1), null, mi1);

        // An instance of C must be passed in each time the
        // delegate is invoked.
        //
        d1(c1, "Hello, World!");
        d1(new C(5280), "Hi, Mom!");

        Console.WriteLine("\nAn open static method.");
        // In this case, the delegate and the method must
        // have the same list of argument types; use delegate type
        // D2 with static method M2.
        //
        d2 = (D2)Delegate.CreateDelegate(typeof(D2), null, mi2);

        // No instances of C are involved, because this is a static
        // method.
        //
        d2("Hello, World!");
        d2("Hi, Mom!");

        Console.WriteLine("\nA static method closed over the first argument
(String).");
```

```
        // The delegate must omit the first argument of the method.
        // A string is passed as the firstArgument parameter, and
        // the delegate is bound to this string. Use delegate type
        // D3 with static method M2.
        //
        d3 = (D3)Delegate.CreateDelegate(typeof(D3),
            "Hello, World!", mi2);

        // Each time the delegate is invoked, the same string is
        // used.
        d3();
    }
}

/* This code example produces the following output:

An instance method closed over C.
Instance method M1 on C:  id = 42, s = Hello, World!
Instance method M1 on C:  id = 42, s = Hi, Mom!

An open instance method.
Instance method M1 on C:  id = 42, s = Hello, World!
Instance method M1 on C:  id = 5280, s = Hi, Mom!

An open static method.
Static method M2 on C:  s = Hello, World!
Static method M2 on C:  s = Hi, Mom!

A static method closed over the first argument (String).
Static method M2 on C:  s = Hello, World!
 */
```

예제 2

다음 코드 예제에서는 매개 변수 형식 및 반환 형식의 호환성을 보여 줍니다.

코드 예제에서는 `Base`라는 기본 클래스와 `Base`에서 파생된 `Derived` 클래스를 정의합니다. 파생 클래스에는 `Base` 형식의 매개 변수를 하나 가지고 `Derived`을(를) 반환하는 `MyMethod`라는 메서드가 있으며, Visual Basic에서는 `Shared`라고 합니다. 코드 예제에서는 형식 `Derived`의 매개 변수를 하나 가지고 반한 형식 `Base`의 대리자 `Example`도 정의합니다.

코드 예제에서는 명명 `Example` 된 대리자를 사용하여 메서드 `MyMethod`를 나타낼 수 있음을 보여 줍니다. 다음과 같은 이유로 메서드를 대리자로 바인딩할 수 있습니다.

- 대리자의 매개 변수 형식(`Derived`)은 (`Base`)의 `MyMethod` 매개 변수 형식보다 더 제한적이므로 항상 대리자의 인수를 전달해도 안전합니다`MyMethod`.

- (`Derived`)의 `MyMethod` 반환 형식은 대리자의 매개 변수 형식보다 더 제한적이므로 메서드의 반환 형식을 항상 대리`Base`자의 반환 형식으로 캐스팅하는 것이 안전합니다.

코드 예제에서는 출력을 생성하지 않습니다.

C#

```
using System;
using System.Reflection;

// Define two classes to use in the demonstration, a base class and
// a class that derives from it.
//
public class Base { }

public class Derived : Base
{
    // Define a static method to use in the demonstration. The method
    // takes an instance of Base and returns an instance of Derived.
    // For the purposes of the demonstration, it is not necessary for
    // the method to do anything useful.
    //
    public static Derived MyMethod(Base arg)
    {
        Base dummy = arg;
        return new Derived();
    }
}

// Define a delegate that takes an instance of Derived and returns an
// instance of Base.
//
public delegate Base Example5(Derived arg);

class Test
{
    public static void Main()
    {
        // The binding flags needed to retrieve MyMethod.
        BindingFlags flags = BindingFlags.Public | BindingFlags.Static;

        // Get a MethodInfo that represents MyMethod.
        MethodInfo minfo = typeof(Derived).GetMethod("MyMethod", flags);

        // Demonstrate contravariance of parameter types and covariance
        // of return types by using the delegate Example5 to represent
        // MyMethod. The delegate binds to the method because the
        // parameter of the delegate is more restrictive than the
        // parameter of the method (that is, the delegate accepts an
        // instance of Derived, which can always be safely passed to
        // a parameter of type Base), and the return type of MyMethod
        // is more restrictive than the return type of Example5 (that
```

```
        // is, the method returns an instance of Derived, which can
        // always be safely cast to type Base).
        //
        Example5 ex =
            (Example5)Delegate.CreateDelegate(typeof(Example5), minfo);

        // Execute MyMethod using the delegate Example5.
        //
        Base b = ex(new Derived());
    }
}
```

CreateDelegate(Type, Object, MethodInfo) 및 CreateDelegate(Type, Object, MethodInfo, Boolean) 메서드

두 오버로드의 기능은 동일하며, 차이점은 하나는 바인딩 실패 시 예외를 발생시킬지 여부를 지정할 수 있다는 것이고, 다른 하나는 항상 예외를 발생시킨다는 것입니다.

대리자 형식과 메서드에는 호환되는 반환 형식이 있어야 합니다. 즉, `method`의 반환 유형은 `type`의 반환 유형으로 할당 가능해야 합니다.

`firstArgument`이러한 오버로드에 대한 두 번째 매개 변수는 대리자가 나타내는 메서드의 첫 번째 인수입니다. `firstArgument`이 제공되면, 대리자가 호출될 때마다 `method`로 전달됩니다. `firstArgument`는 대리자에게 바인딩되었다고 하며, 대리자는 첫 번째 인수를 통해 닫혀 있다고 합니다. `method`가 `static`(`Shared`는 Visual Basic에서)인 경우, 대리자를 호출할 때 제공되는 인수 목록에는 첫 번째 매개 변수를 제외한 모든 매개 변수가 포함됩니다. `method`가 인스턴스 메서드인 경우 `firstArgument`는 숨겨진 인스턴스 매개 변수(`this`는 C#에서, `Me`는 Visual Basic에서)로 전달됩니다.

제공된 경우 `firstArgument` 첫 번째 매개 변수는 `method` 참조 형식이어야 하며 해당 형식과 `firstArgument` 호환되어야 합니다.

> ⓘ **중요**
>
> `method`가 `static`일 경우 (`Shared`는 Visual Basic의 경우) 첫 번째 매개 변수가 Object 또는 ValueType인 경우, `firstArgument`는 값 형식일 수 있습니다. 이 경우 `firstArgument`은 자동으로 박싱됩니다. C# 또는 Visual Basic 함수 호출에서와 마찬가지로 다른 인수에는 자동 boxing이 발생하지 않습니다.

`firstArgument`이(가) null 참조이고 `method`이(가) 인스턴스 메서드인 경우, 결과는 대리자 형식 `type` 및 `method`의 서명에 따라 달라집니다.

- 시그니처에 `type` 숨겨진 첫 번째 매개 변수 `method`가 명시적으로 포함된 경우 대리자는 열려 있는 인스턴스 메서드를 나타낸다. 대리자를 호출하면 인수 목록의 첫 번째 인수가 숨겨진 인스턴스 매개 변수 `method`로 전달됩니다.
- 서명 `method` 과 `type` 일치하는 경우(즉, 모든 매개 변수 형식이 호환됨) 대리자는 null 참조를 통해 닫혀 있다고 합니다. 대리자를 호출하는 것은 null 인스턴스에서 인스턴스 메서드를 호출하는 것과 같으며 특히 유용하지는 않습니다.

`firstArgument`이(가) null 참조인 경우, `method`이(가) 정적이면 결과는 대리자 형식 `type` 및 `method`의 서명에 따라 달라집니다.

- 시그니처 `method` 와 `type` 일치(즉, 모든 매개 변수 형식이 호환됨)이면 대리자는 개방형 정적 메서드를 나타낸다. 정적 메서드에 가장 일반적인 경우입니다. 이 경우 메서드 오버로드를 사용하여 CreateDelegate(Type, MethodInfo) 약간 더 나은 성능을 얻을 수 있습니다.
- 시그니처가 `type` 두 번째 매개 변수로 시작하고 나머지 매개 변수 `method` 형식이 호환되는 경우 대리자는 null 참조를 통해 닫힌 것으로 표시됩니다. 대리자를 호출하면 null 참조가 첫 번째 매개 변수 `method`에 전달됩니다.

예시

다음 코드 예제에서는 단일 대리자형이 나타낼 수 있는 모든 메서드를 보여 줍니다: 인스턴스 메서드에 닫힌 형식, 인스턴스 메서드에 열린 형식, 정적 메서드에 열린 형식, 정적 메서드에 닫힌 형식.

코드 예제에서는 두 개의 클래스 `C` 를 정의하고 `F`형식의 인수가 하나 있는 대리자 형식 `D C`을 정의합니다. 클래스에는 일치하는 정적 메서드와 인스턴스 메서드 `M1`가 `M3`있으며 `M4`, 클래스 `C` 에는 인수가 없는 인스턴스 메서드 `M2` 도 있습니다.

명명된 `Example` 세 번째 클래스에는 대리자를 만드는 코드가 포함됩니다.

- 대리자는 `C` 및 `F` 형식의 인스턴스 메서드 `M1`에 대해 생성되며, 각각은 해당 형식의 인스턴스를 위해 닫힙니다. 형식 `C` 의 메서드 `M1` 는 `ID` 바인딩된 인스턴스 및 인수의 속성을 표시합니다.
- 형식 `C`의 메서드 `M2`에 대한 대리자가 만들어집니다. 대리자의 인수가 인스턴스 메서드의 숨겨진 첫 번째 인수를 나타내는 열린 인스턴스 대리자입니다. 메서드에 다른 인수가 없습니다. 정적 메서드인 것처럼 호출됩니다.
- 정적 메서드 `M3` 및 형식 `C`와 `F`의 대리자가 생성됩니다. 이는 열린 정적 대리자입니다.

- 마지막으로 형식 `C` 및 형식 `F` 의 정적 메서드 `M4` 에 대해 대리자가 만들어집니다. 각 메서드는 선언 형식을 첫 번째 인수로 사용하고 형식의 인스턴스를 제공하므로 대리자는 첫 번째 인수를 통해 닫힙니다. 형식 `C` 의 메서드 `M4` 는 `ID` 바인딩된 인스턴스 및 인수의 속성을 표시합니다.

C#

```
using System;
using System.Reflection;

// Declare a delegate type. The object of this code example
// is to show all the methods this delegate can bind to.
//
public delegate void D(C1 c);

// Declare two sample classes, C1 and F. Class C1 has an ID
// property so instances can be identified.
//
public class C1
{
    private int id;
    public int ID { get { return id; } }
    public C1(int id) { this.id = id; }

    public void M1(C1 c)
    {
        Console.WriteLine("Instance method M1(C1 c) on C1:  this.id = {0},
c.ID = {1}",
            this.id, c.ID);
    }

    public void M2()
    {
        Console.WriteLine($"Instance method M2() on C1:  this.id =
{this.id}");
    }

    public static void M3(C1 c)
    {
        Console.WriteLine($"Static method M3(C1 c) on C1:  c.ID = {c.ID}");
    }

    public static void M4(C1 c1, C1 c2)
    {
        Console.WriteLine("Static method M4(C1 c1, C1 c2) on C1:  c1.ID =
{0}, c2.ID = {1}",
            c1.ID, c2.ID);
    }
}

public class F
{
    public void M1(C1 c)
```

```
    {
        Console.WriteLine($"Instance method M1(C1 c) on F:  c.ID = {c.ID}");
    }

    public static void M3(C1 c)
    {
        Console.WriteLine($"Static method M3(C1 c) on F:  c.ID = {c.ID}");
    }

    public static void M4(F f, C1 c)
    {
        Console.WriteLine($"Static method M4(F f, C1 c) on F:  c.ID =
{c.ID}");
    }
}

public class Example
{
    public static void Main()
    {
        C1 c1 = new C1(42);
        C1 c2 = new C1(1491);
        F f1 = new F();

        D d;

        // Instance method with one argument of type C1.
        MethodInfo cmi1 = typeof(C1).GetMethod("M1");
        // Instance method with no arguments.
        MethodInfo cmi2 = typeof(C1).GetMethod("M2");
        // Static method with one argument of type C1.
        MethodInfo cmi3 = typeof(C1).GetMethod("M3");
        // Static method with two arguments of type C1.
        MethodInfo cmi4 = typeof(C1).GetMethod("M4");

        // Instance method with one argument of type C1.
        MethodInfo fmi1 = typeof(F).GetMethod("M1");
        // Static method with one argument of type C1.
        MethodInfo fmi3 = typeof(F).GetMethod("M3");
        // Static method with an argument of type F and an argument
        // of type C1.
        MethodInfo fmi4 = typeof(F).GetMethod("M4");

        Console.WriteLine("\nAn instance method on any type, with an
argument of type C1.");
        // D can represent any instance method that exactly matches its
        // signature. Methods on C1 and F are shown here.
        //
        d = (D)Delegate.CreateDelegate(typeof(D), c1, cmi1);
        d(c2);
        d = (D)Delegate.CreateDelegate(typeof(D), f1, fmi1);
        d(c2);

        Console.WriteLine("\nAn instance method on C1 with no arguments.");
        // D can represent an instance method on C1 that has no arguments;
```

```
        // in this case, the argument of D represents the hidden first
        // argument of any instance method. The delegate acts like a
        // static method, and an instance of C1 must be passed each time
        // it is invoked.
        //
        d = (D)Delegate.CreateDelegate(typeof(D), null, cmi2);
        d(c1);

        Console.WriteLine("\nA static method on any type, with an argument
of type C1.");
        // D can represent any static method with the same signature.
        // Methods on F and C1 are shown here.
        //
        d = (D)Delegate.CreateDelegate(typeof(D), null, cmi3);
        d(c1);
        d = (D)Delegate.CreateDelegate(typeof(D), null, fmi3);
        d(c1);

        Console.WriteLine("\nA static method on any type, with an argument
of");
        Console.WriteLine("    that type and an argument of type C1.");
        // D can represent any static method with one argument of the
        // type the method belongs and a second argument of type C1.
        // In this case, the method is closed over the instance of
        // supplied for the its first argument, and acts like an instance
        // method. Methods on F and C1 are shown here.
        //
        d = (D)Delegate.CreateDelegate(typeof(D), c1, cmi4);
        d(c2);
        Delegate test =
            Delegate.CreateDelegate(typeof(D), f1, fmi4, false);

        // This final example specifies false for throwOnBindFailure
        // in the call to CreateDelegate, so the variable 'test'
        // contains Nothing if the method fails to bind (for
        // example, if fmi4 happened to represent a method of
        // some class other than F).
        //
        if (test != null)
        {
            d = (D)test;
            d(c2);
        }
    }
}

/* This code example produces the following output:

An instance method on any type, with an argument of type C1.
Instance method M1(C1 c) on C1:  this.id = 42, c.ID = 1491
Instance method M1(C1 c) on F:  c.ID = 1491

An instance method on C1 with no arguments.
Instance method M2() on C1:  this.id = 42
```

```
A static method on any type, with an argument of type C1.
Static method M3(C1 c) on C1:  c.ID = 42
Static method M3(C1 c) on F:  c.ID = 42

A static method on any type, with an argument of
    that type and an argument of type C1.
Static method M4(C1 c1, C1 c2) on C1:  c1.ID = 42, c2.ID = 1491
Static method M4(F f, C1 c) on F:  c.ID = 1491
*/
```

호환되는 매개 변수 형식 및 반환 형식

이 메서드 오버로드를 사용하여 만든 대리자의 매개 변수 형식 및 반환 형식은 대리자가 나타내는 메서드의 매개 변수 형식 및 반환 형식과 호환되어야 합니다. 형식은 정확히 일치할 필요가 없습니다.

대리자의 매개 변수는 대리자 매개 변수의 형식이 메서드 매개 변수의 형식보다 더 제한적인 경우 메서드의 해당 매개 변수와 호환됩니다. 대리자 매개 변수에 전달된 인수를 메서드에 안전하게 전달할 수 있기 때문입니다.

마찬가지로 메서드의 반환 형식이 대리자의 반환 형식보다 더 제한적인 경우 대리자의 반환 형식이 메서드의 반환 형식과 호환됩니다. 이 경우 메서드의 반환 값이 대리자의 반환 형식으로 안전하게 캐스팅될 수 있기 때문입니다.

예를 들어, Hashtable 형식의 매개 변수를 가지고 Object 형식의 반환 값을 가지는 대리자는, Object 형식의 매개 변수와 Hashtable 형식의 반환 값을 가지는 메서드를 나타낼 수 있습니다.

대리자가 표현할 수 있는 메서드를 결정하다

오버로드에서 제공하는 CreateDelegate(Type, Object, MethodInfo) 유연성을 생각하는 또 다른 유용한 방법은 지정된 대리자가 메서드 시그니처와 메서드 종류(정적 및 인스턴스)의 네 가지 조합을 나타낼 수 있다는 것입니다. 하나의 인수가 있는 대리자 형식 `D` 을 고려합니다 `C`. 다음은 모든 경우에 일치해야 하므로 반환 형식을 무시하고 메서드가 나타낼 수 있는 방법을 `D` 설명합니다.

- `D` 는 인스턴스 메서드가 속한 형식에 관계없이 정확히 하나의 형식 `C` 인수가 있는 인스턴스 메서드를 나타낼 수 있습니다. CreateDelegate가 호출될 때, `firstArgument` 은 `method` 형식에 속하는 인스턴스이며, 그 결과 대리자는 해당 인스턴스에 대해 닫혀 있다고 합니다. (간단히 말해서, `D` 가 null 참조인 경우, `firstArgument` 도 null 참조를 통해 닫을 수 있습니다.)

- `D`는 `C`의 인수가 없는 인스턴스 메서드를 나타낼 수 있습니다. CreateDelegate가 호출될 때, `firstArgument`은 null 참조입니다. 결과 대리자는 열린 인스턴스 메서드를 나타내며 호출될 때마다 인스턴스 `C` 를 제공해야 합니다.

- `D` 는 형식의 인수 하나를 사용하는 정적 메서드를 나타낼 수 있으며 해당 메서드는 모든 형식 `C`에 속할 수 있습니다. CreateDelegate가 호출될 때, `firstArgument`은 null 참조입니다. 결과 대리자는 열린 정적 메서드를 나타내며 호출될 때마다 인스턴스 `C` 를 제공해야 합니다.

- `D`는 `F` 형식에 속하는 정적 메서드를 나타낼 수 있으며, 인수로는 `F` 형식과 `C` 형식의 두 가지가 있습니다. CreateDelegate이(가) 호출되면, `firstArgument`은(는) `F`의 인스턴스입니다. 결과 대리자는 해당 인스턴스 `F`에 대해 닫혀 있는 정적 메서드를 나타냅니다. 형식이 동일한 경우 `F` `C` 정적 메서드에는 해당 형식의 두 인수가 있습니다. (이 경우 `firstArgument`가 null 참조일 때 `D`는 null 참조를 통해 닫힙니다.)

System.Enum 클래스

2025. 06. 22.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

열거형은 기본 형식이 모든 정수 형식인 명명된 상수 집합입니다. 명시적으로 선언된 Int32 기본 형식이 없으면 사용됩니다. Enum 는 .NET의 모든 열거형에 대한 기본 클래스입니다. 열거형 형식은 C#의 `enum` 키워드, Visual Basic의 `Enum`...`End Enum` 구문 및 F#의 `type` 키워드로 정의됩니다.

Enum 에서는 이 클래스의 인스턴스를 비교하고, 인스턴스 값을 문자열 표현으로 변환하고, 숫자의 문자열 표현을 이 클래스의 인스턴스로 변환하고, 지정된 열거형 및 값의 인스턴스를 만드는 메서드를 제공합니다.

열거형을 비트 필드로 처리할 수도 있습니다. 자세한 내용은 비독점 멤버 및 Flags 특성 섹션을 참조하세요 FlagsAttribute.

열거형 형식 만들기

프로그래밍 언어는 일반적으로 명명된 상수 집합과 해당 값으로 구성된 열거형을 선언하는 구문을 제공합니다. 다음 예제에서는 C#, F# 및 Visual Basic에서 열거형을 정의하는 데 사용하는 구문을 보여 줍니다. `ArrivalStatus`라는 이름이 붙은 열거형에는 세 개의 멤버인 `ArrivalStatus.Early`, `ArrivalStatus.OnTime`, 및 `ArrivalStatus.Late`가 있습니다. 모든 경우에 열거형은 Enum로부터 명시적으로 상속되지 않으며, 상속 관계는 컴파일러에서 암시적으로 처리됩니다.

C#

```
public enum ArrivalStatus { Unknown=-3, Late=-1, OnTime=0, Early=1 };
```

> ⚠ **경고**
>
> 내부 형식이 정수 형식이 아니거나 Char열거형 형식을 만들면 안 됩니다. 리플렉션을 사용하여 이러한 열거형 형식을 만들 수 있지만 결과 형식을 사용하는 메서드 호출은 신뢰할 수 없으며 추가 예외를 throw할 수도 있습니다.

열거형 형식 인스턴스화

변수를 선언하고 열거형의 상수 중 하나를 할당하여 다른 값 형식을 인스턴스화하는 것처럼 열거형 형식을 인스턴스화할 수 있습니다. 다음 예에서는 값이 `ArrivalStatus`인 `ArrivalStatus.OnTime`를 인스턴스화합니다.

C#

```
public class Example
{
   public static void Main()
   {
      ArrivalStatus status = ArrivalStatus.OnTime;
      Console.WriteLine($"Arrival Status: {status} ({status:D})");
   }
}
// The example displays the following output:
//       Arrival Status: OnTime (0)
```

다음과 같은 방법으로 열거형 값을 인스턴스화할 수도 있습니다.

- 특정 프로그래밍 언어의 기능을 사용하여 C#에서와 같이 캐스팅하거나(Visual Basic에서와 같이) 정수 값을 열거형 값으로 변환합니다. 다음 예제에서는 값을 `ArrivalStatus`로 갖는 `ArrivalStatus.Early` 개체를 이런 방식으로 만듭니다.

  C#

  ```
  ArrivalStatus status2 = (ArrivalStatus)1;
  Console.WriteLine($"Arrival Status: {status2} ({status2:D})");
  // The example displays the following output:
  //       Arrival Status: Early (1)
  ```

- 암시적 매개 변수 없는 생성자를 호출합니다. 다음 예제와 같이 이 경우 열거형 인스턴스의 기본 값은 0입니다. 그러나 열거형에서 유효한 상수의 값이 반드시 있는 것은 아닙니다.

  C#

  ```
  ArrivalStatus status1 = new ArrivalStatus();
  Console.WriteLine($"Arrival Status: {status1} ({status1:D})");
  // The example displays the following output:
  //       Arrival Status: OnTime (0)
  ```

- 상수의 이름을 포함하는 문자열을 구문 분석하기 위해 열거형의 Parse 또는 TryParse 메서드를 호출합니다. 구문 분석 열거형 값 섹션을 참조하여 더 많은 정보를 확인하세요.

- 정수 값을 열거형 형식으로 변환하는 메서드를 호출 ToObject 합니다. 자세한 내용은 변환 수행 섹션을 참조하세요 .

열거형 모범 사례

열거형 형식을 정의할 때는 다음 모범 사례를 사용하는 것이 좋습니다.

- 값이 0인 열거형 멤버를 정의하지 않은 경우 열거형 상수 만들기를 `None` 고려합니다. 기본적으로 열거형에 사용되는 메모리는 공용 언어 런타임에 의해 0으로 초기화됩니다. 따라서 값이 0인 상수는 정의하지 않으면 열거형을 만들 때 잘못된 값이 포함됩니다.
- 애플리케이션이 나타내야 하는 명백한 기본 사례가 있는 경우 값을 나타내는 데 값이 0인 열거형 상수 사용을 고려합니다. 기본 사례가 없는 경우, 다른 열거형 상수로 표현되지 않는 경우를 지정하기 위해 값이 0인 열거형 상수를 사용하는 것이 좋습니다.
- 나중에 사용하도록 예약된 열거형 상수를 지정하지 마세요.
- 열거형 상수 값을 값으로 사용하는 메서드 또는 속성을 정의할 때 값의 유효성을 검사하는 것이 좋습니다. 그 이유는 해당 숫자 값이 열거형에 정의되지 않은 경우에도 숫자 값을 열거형 형식으로 캐스팅할 수 있기 때문입니다.

비독점 멤버 및 Flags 특성 섹션에 상수가 비트 필드인 열거형 형식에 대한 추가 모범 사례입니다.

열거형을 사용하여 작업 수행

열거형을 만들 때는 새 메서드를 정의할 수 없습니다. 그러나 열거형 형식은 클래스에서 정적 및 인스턴스 메서드의 전체 집합을 상속합니다 Enum . 다음 섹션에서는 열거형 값을 사용할 때 일반적으로 사용되는 몇 가지 다른 메서드 외에도 이러한 메서드의 대부분을 조사합니다.

변환 수행

캐스팅(C# 및 F#) 또는 변환(Visual Basic) 연산자를 사용하여 열거형 멤버와 해당 기본 형식 간에 변환할 수 있습니다. F#에서도 `enum` 함수가 사용됩니다. 다음 예제에서는 캐스팅 또는 변환 연산자를 사용하여 정수에서 열거형 값으로, 열거형 값에서 정수로 변환을 수행합니다.

C#

```
int value3 = 2;
ArrivalStatus status3 = (ArrivalStatus)value3;

int value4 = (int)status3;
```

Enum 클래스에는 모든 정수 계열 형식의 값을 열거형 값으로 변환하는 메서드 ToObject도 포함됩니다. 다음 예제에서는 ToObject(Type, Int32) 메서드를 사용하여 Int32을 `ArrivalStatus` 값으

로 변환합니다. ToObject 형식 값을 반환하므로 개체를 열거형 형식Object으로 캐스팅하는 데 캐스팅 또는 변환 연산자를 사용해야 할 수도 있습니다.

C#

```
int number = -1;
ArrivalStatus arrived =
(ArrivalStatus)ArrivalStatus.ToObject(typeof(ArrivalStatus), number);
```

정자를 열거형 값으로 변환할 때 실제로 열거형의 멤버가 아닌 값을 할당할 수 있습니다. 이를 방지하기 위해 변환을 수행하기 전에 정수를 IsDefined 메서드에 전달할 수 있습니다. 다음 예제에서는 이 메서드를 사용하여 정수 값 배열의 요소를 값으로 변환 `ArrivalStatus` 할 수 있는지 여부를 확인합니다.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        int[] values = { -3, -1, 0, 1, 5, Int32.MaxValue };
        foreach (var value in values)
        {
            ArrivalStatus status;
            if (Enum.IsDefined(typeof(ArrivalStatus), value))
                status = (ArrivalStatus)value;
            else
                status = ArrivalStatus.Unknown;
            Console.WriteLine($"Converted {value:N0} to {status}");
        }
    }
}
// The example displays the following output:
//       Converted -3 to Unknown
//       Converted -1 to Late
//       Converted 0 to OnTime
//       Converted 1 to Early
//       Converted 5 to Unknown
//       Converted 2,147,483,647 to Unknown
```

클래스는 Enum 열거형 값에서 정수 형식으로 변환하기 위한 인터페이스의 IConvertible 명시적 인터페이스 구현을 제공하지만 이러한 변환을 수행하려면 클래스의 Convert 메서드(예: ToInt32 메서드)를 사용해야 합니다. 다음 예제에서는 GetUnderlyingType 메서드와 Convert.ChangeType 메서드를 함께 사용하여 열거형 값을 기본 형식으로 변환하는 방법을 보여 줍니다. 이 예제에서는 컴파일 시간에 열거형의 기본 형식을 알 필요가 없습니다.

C#

```
ArrivalStatus status = ArrivalStatus.Early;
var number = Convert.ChangeType(status,
Enum.GetUnderlyingType(typeof(ArrivalStatus)));
Console.WriteLine($"Converted {status} to {number}");
// The example displays the following output:
//       Converted Early to 1
```

열거형 값 분석하기

Parse 및 TryParse 메서드를 사용하면 열거형 값의 문자열 표현을 해당 값으로 변환할 수 있습니다. 문자열 표현은 열거형 상수의 이름 또는 기본 값일 수 있습니다. 문자열을 열거형의 기본 형식 값으로 변환할 수 있는 경우 구문 분석 메서드는 특정 열거형의 멤버가 아닌 숫자의 문자열 표현을 성공적으로 변환합니다. 이를 IsDefined 방지하기 위해 구문 분석 메서드의 결과가 유효한 열거형 값인지 확인하기 위해 메서드를 호출할 수 있습니다. 이 예제는 이 접근 방식을 설명하고 Parse(Type, String) 메서드와 Enum.TryParse<TEnum>(String, TEnum) 메서드에 대한 호출을 보여줍니다. 제네릭이 아닌 구문 분석 메서드는 C# 및 F#에서 캐스팅하거나 Visual Basic에서는 적절한 열거형 형식으로 변환해야 할 수 있는 개체를 반환합니다.

C#

```
string number = "-1";
string name = "Early";

try
{
    ArrivalStatus status1 = (ArrivalStatus)Enum.Parse(typeof(ArrivalStatus),
number);
    if (!(Enum.IsDefined(typeof(ArrivalStatus), status1)))
        status1 = ArrivalStatus.Unknown;
    Console.WriteLine($"Converted '{number}' to {status1}");
}
catch (FormatException)
{
    Console.WriteLine($"Unable to convert '{number}' to an ArrivalStatus value.");
}

ArrivalStatus status2;
if (Enum.TryParse<ArrivalStatus>(name, out status2))
{
    if (!(Enum.IsDefined(typeof(ArrivalStatus), status2)))
        status2 = ArrivalStatus.Unknown;
    Console.WriteLine($"Converted '{name}' to {status2}");
}
else
{
    Console.WriteLine($"Unable to convert '{number}' to an ArrivalStatus value.");
}
```

```
// The example displays the following output:
//       Converted '-1' to Late
//       Converted 'Early' to Early
```

열거형 값의 서식 지정

정적 Format 메서드와 인스턴스 ToString 메서드의 오버로드를 호출하여 열거형 값을 문자열 표현으로 변환할 수 있습니다. 형식 문자열을 사용하여 열거형 값이 문자열로 표시되는 정확한 방법을 제어할 수 있습니다. 자세한 내용은 열거형 형식 문자열참조하세요. 다음 예제에서는 지원되는 각 열거형 형식 문자열("G" 또는 "g", "D" 또는 "d", "X" 또는 "x" 및 "F" 또는 "f")을 사용하여 열거형의 `ArrivalStatus` 멤버를 해당 문자열 표현으로 변환합니다.

C#

```
string[] formats = { "G", "F", "D", "X" };
ArrivalStatus status = ArrivalStatus.Late;
foreach (var fmt in formats)
    Console.WriteLine(status.ToString(fmt));

// The example displays the following output:
//       Late
//       Late
//       -1
//       FFFFFFFF
```

열거형 멤버 반복

Enum 형식은 IEnumerable 또는 IEnumerable<T> 인터페이스를 구현하지 않아, C#의 `foreach`, F#의 `for..in`, 또는 Visual Basic의 `For Each` 구문을 사용하여 컬렉션의 멤버를 반복할 수 없습니다. 그러나 두 가지 방법 중 하나를 사용하여 멤버를 열거할 수 있습니다.

- 메서드를 GetNames 호출하여 열거형 멤버의 이름을 포함하는 문자열 배열을 검색할 수 있습니다. 다음으로 문자열 배열의 각 요소에 대해 메서드를 Parse 호출하여 문자열을 해당하는 열거형 값으로 변환할 수 있습니다. 다음 예제에서는 이 방법을 보여 줍니다.

  C#

  ```
  string[] names = Enum.GetNames(typeof(ArrivalStatus));
  Console.WriteLine($"Members of {typeof(ArrivalStatus).Name}:");
  Array.Sort(names);
  foreach (var name in names)
  {
      ArrivalStatus status = (ArrivalStatus)Enum.Parse(typeof(ArrivalStatus),
  name);
      Console.WriteLine($"   {status} ({status:D})");
  }
  ```

```
// The example displays the following output:
//       Members of ArrivalStatus:
//          Early (1)
//          Late (-1)
//          OnTime (0)
//          Unknown (-3)
```

- 메서드를 GetValues 호출하여 열거형의 기본 값이 포함된 배열을 검색할 수 있습니다. 다음으로 배열의 각 요소에 대해 메서드를 ToObject 호출하여 정수를 해당하는 열거형 값으로 변환할 수 있습니다. 다음 예제에서는 이 방법을 보여 줍니다.

C#

```
var values = Enum.GetValues(typeof(ArrivalStatus));
Console.WriteLine($"Members of {typeof(ArrivalStatus).Name}:");
foreach (ArrivalStatus status in values)
{
    Console.WriteLine($"   {status} ({status:D})");
}
// The example displays the following output:
//       Members of ArrivalStatus:
//          OnTime (0)
//          Early (1)
//          Unknown (-3)
//          Late (-1)
```

비독점 멤버 및 플래그 속성

열거형의 일반적인 용도 중 하나는 상호 배타적인 값 집합을 나타내는 것입니다. 예를 들어 인스턴스가 `ArrivalStatus` 값으로 `Early`, `OnTime`, 또는 `Late`를 가질 수 있습니다. 인스턴스 값 `ArrivalStatus` 이 둘 이상의 열거형 상수에 반영되는 것은 의미가 없습니다.

그러나 열거형 개체의 값에는 여러 열거형 멤버가 포함될 수 있으며 각 멤버는 열거형 값의 비트 필드를 나타냅니다. 이 FlagsAttribute 특성을 사용하여 열거형이 비트 필드로 구성됨을 나타낼 수 있습니다. 예를 들어, 명명 `Pets` 된 열거형을 사용하여 가정에서 애완 동물의 종류를 나타낼 수 있습니다. 다음과 같이 정의할 수 있습니다.

C#

```
[Flags]
public enum Pets
{
    None = 0, Dog = 1, Cat = 2, Bird = 4, Rodent = 8,
    Reptile = 16, Other = 32
};
```

`Pets` 그런 다음, 다음 예제와 같이 열거형을 사용할 수 있습니다.

C#

```
Pets familyPets = Pets.Dog | Pets.Cat;
Console.WriteLine($"Pets: {familyPets:G} ({familyPets:D})");
// The example displays the following output:
//       Pets: Dog, Cat (3)
```

비트 열거형을 정의하고 특성을 적용 FlagsAttribute 할 때는 다음 모범 사례를 사용해야 합니다.

- FlagsAttribute 숫자 값에 대해 비트 연산(AND, OR, EXCLUSIVE OR)을 수행해야 하는 경우에만 열거형에 사용자 지정 특성을 사용합니다.
- 열거형 상수를 정의할 때 2의 제곱수로, 즉 1, 2, 4, 8 등으로 정의합니다. 즉, 결합된 열거형 상수의 개별 플래그가 겹치지 않습니다.
- 일반적으로 사용되는 플래그 조합에 대해 열거형 상수 만들기를 고려합니다. 열거된 상수 `Read = 1` 및 `Write = 2`이 포함된 파일 I/O 작업에 사용되는 열거형이 있는 경우, `ReadWrite = Read OR Write` 및 `Read` 플래그를 결합한 열거형 상수 `Write`를 만드는 것을 고려하십시오. 또한 플래그를 결합하는 데 사용되는 비트 OR 연산은 간단한 작업에 필요하지 않은 경우에 고급 개념으로 간주될 수 있습니다.
- 많은 플래그 위치가 1로 설정될 수 있으므로 음수를 플래그 열거 상수로 정의하면 코드가 혼동되고 코딩 오류가 발생할 수 있으므로 주의해야 합니다.
- 플래그가 숫자 값에 설정되어 있는지 여부를 테스트하는 편리한 방법은 다음 예제와 같이 인스턴스 HasFlag 메서드를 호출하는 것입니다.

  C#

  ```
  Pets familyPets = Pets.Dog | Pets.Cat;
  if (familyPets.HasFlag(Pets.Dog))
      Console.WriteLine("The family has a dog.");
  // The example displays the following output:
  //       The family has a dog.
  ```

  숫자 값과 플래그 열거형 상수 간에 비트 AND 연산을 수행하는 것과 같습니다. 이 상수는 숫자 값의 모든 비트를 플래그에 해당하지 않는 0으로 설정한 다음, 해당 작업의 결과가 플래그 열거 상수와 같은지 여부를 테스트합니다. 다음 예에서 확인할 수 있습니다.

  C#

  ```
  Pets familyPets = Pets.Dog | Pets.Cat;
  if ((familyPets & Pets.Dog) == Pets.Dog)
      Console.WriteLine("The family has a dog.");
  ```

```
// The example displays the following output:
//       The family has a dog.
```

- 값이 0인 플래그 열거형 상수의 이름으로 사용합니다 `None` . 결과가 항상 0이므로 비트 AND 연산에서 열거형 상수로 플래그를 테스트할 수 없습니다 `None` . 그러나 숫자 값과 `None` 열거된 상수 간의 비트 비교가 아닌 논리적 비교를 수행하여 숫자 값의 비트가 설정되었는지 여부를 확인할 수 있습니다. 다음 예에서 확인할 수 있습니다.

C#

```
Pets familyPets = Pets.Dog | Pets.Cat;
if (familyPets == Pets.None)
    Console.WriteLine("The family has no pets.");
else
    Console.WriteLine("The family has pets.");
// The example displays the following output:
//       The family has pets.
```

- 열거형 자체의 상태를 미러링하기 위해서만 열거형 값을 정의하지 마세요. 예를 들어 열거형의 끝을 표시하는 열거형 상수만 정의하지 마세요. 열거형의 마지막 값을 확인해야 하는 경우 해당 값을 명시적으로 확인합니다. 또한 범위 내의 모든 값이 유효한 경우 첫 번째 및 마지막 열거 상수에 대한 범위 검사를 수행할 수 있습니다.

열거형 메서드 추가

열거형 형식은 (C#) 및 (Visual Basic) 같은 `enum` `Enum` 언어 구조에 의해 정의되므로 클래스에서 Enum 상속된 메서드 이외의 열거형 형식에 대한 사용자 지정 메서드를 정의할 수 없습니다. 그러나 확장 메서드를 사용하여 특정 열거형 형식에 기능을 추가할 수 있습니다.

다음 예제 `Grades` 에서 열거형은 학생이 수업에서 받을 수 있는 가능한 문자 성적을 나타냅니다. 형식 `Passing` 에 이름이 `Grades` 인 확장 메서드가 추가되어 해당 형식의 각 인스턴스가 이제 통과 성적인지 아닌지를 "알 수 있게 되었습니다". 클래스에는 `Extensions` 최소 통과 등급을 정의하는 정적 읽기-쓰기 변수도 포함됩니다. 확장 메서드의 `Passing` 반환 값은 해당 변수의 현재 값을 반영합니다.

C#

```
using System;

// Define an enumeration to represent student grades.
public enum Grades { F = 0, D = 1, C = 2, B = 3, A = 4 };

// Define an extension method for the Grades enumeration.
public static class Extensions
{
```

```
    public static Grades minPassing = Grades.D;

    public static bool Passing(this Grades grade)
    {
        return grade >= minPassing;
    }
}

class Example8
{
    static void Main()
    {
        Grades g1 = Grades.D;
        Grades g2 = Grades.F;
        Console.WriteLine($"{g1} {(g1.Passing() ? "is" : "is not")} a passing
grade.");
        Console.WriteLine($"{g2} {(g2.Passing() ? "is" : "is not")} a passing
grade.");

        Extensions.minPassing = Grades.C;
        Console.WriteLine("\nRaising the bar!\n");
        Console.WriteLine($"{g1} {(g1.Passing() ? "is" : "is not")} a passing
grade.");
        Console.WriteLine($"{g2} {(g2.Passing() ? "is" : "is not")} a passing
grade.");
    }
}
// The exmaple displays the following output:
//       D is a passing grade.
//       F is not a passing grade.
//
//       Raising the bar!
//
//       D is not a passing grade.
//       F is not a passing grade.
```

예시

다음 예제에서는 열거형을 사용하여 명명된 값을 나타내고 다른 열거형을 사용하여 명명된 비트 필드를 나타내는 방법을 보여 줍니다.

C#

```
using System;

public class EnumTest {
    enum Days { Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday };
    enum BoilingPoints { Celsius = 100, Fahrenheit = 212 };
    [Flags]
    enum Colors { Red = 1, Green = 2, Blue = 4, Yellow = 8 };
```

```
    public static void Main() {

        Type weekdays = typeof(Days);
        Type boiling = typeof(BoilingPoints);

        Console.WriteLine("The days of the week, and their corresponding values in
the Days Enum are:");

        foreach ( string s in Enum.GetNames(weekdays) )
            Console.WriteLine( "{0,-11}= {1}", s, Enum.Format( weekdays,
Enum.Parse(weekdays, s), "d"));

        Console.WriteLine();
        Console.WriteLine("Enums can also be created which have values that
represent some meaningful amount.");
        Console.WriteLine("The BoilingPoints Enum defines the following items, and
corresponding values:");

        foreach ( string s in Enum.GetNames(boiling) )
            Console.WriteLine( "{0,-11}= {1}", s, Enum.Format(boiling,
Enum.Parse(boiling, s), "d"));

        Colors myColors = Colors.Red | Colors.Blue | Colors.Yellow;
        Console.WriteLine();
        Console.WriteLine($"myColors holds a combination of colors. Namely:
{myColors}");
    }
}
```

System.FlagsAttribute 클래스

아티클 • 2025. 04. 05.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

특성은 FlagsAttribute 열거형을 비트 필드, 즉 플래그 집합으로 처리할 수 있음을 나타냅니다.

비트 필드는 일반적으로 함께 발생할 수 있는 요소 목록에 사용되는 반면 열거형 상수는 일반적으로 상호 배타적 요소 목록에 사용됩니다. 따라서 비트 필드는 비트 `OR` 연산과 결합하여 명명되지 않은 값을 생성하도록 디자인된 반면 열거된 상수는 생성되지 않습니다. 언어는 열거형 상수에 비해 비트 필드 사용에 따라 다릅니다.

FlagsAttribute의 특성

AttributeUsageAttribute 가 이 클래스에 적용되고 해당 속성이 Inherited 지정됩니다 `false`. 이 특성은 열거형에만 적용할 수 있습니다.

FlagsAttribute 및 열거형에 대한 지침

- FlagsAttribute 숫자 값에 대해 비트 연산(AND, OR, EXCLUSIVE OR)을 수행해야 하는 경우에만 열거형에 사용자 지정 특성을 사용합니다.
- 열거형 상수를 정의할 때 2의 제곱수로, 즉 1, 2, 4, 8 등으로 정의합니다. 즉, 결합된 열거형 상수의 개별 플래그가 겹치지 않습니다.
- 일반적으로 사용되는 플래그 조합에 대해 열거형 상수 만들기를 고려합니다. 열거된 상수 `Read = 1` 및 `Write = 2`이 포함된 파일 I/O 작업에 사용되는 열거형이 있는 경우, `Read` 및 `Write` 플래그를 결합한 열거형 상수 `ReadWrite = Read OR Write`를 만드는 것을 고려하십시오. 또한 플래그를 결합하는 데 사용되는 비트 OR 연산은 간단한 작업에 필요하지 않은 경우에 고급 개념으로 간주될 수 있습니다.
- 많은 플래그 위치가 1로 설정될 수 있으므로 음수를 플래그 열거 상수로 정의하면 코드가 혼동되고 코딩 오류가 발생할 수 있으므로 주의해야 합니다.
- 플래그가 숫자 값에 설정되어 있는지 여부를 테스트하는 편리한 방법은 숫자 값과 플래그 열거 상수 간에 비트 AND 연산을 수행하는 것입니다. 이 연산은 숫자 값의 모든 비트를 플래그에 해당하지 않는 0으로 설정한 다음 해당 작업의 결과가 플래그 열거 상수와 같은지 여부를 테스트합니다.
- 값이 0인 플래그 열거형 상수의 이름으로 사용합니다 `None` . 결과가 항상 0이므로 비트 AND 연산에서 열거형 상수로 플래그를 테스트할 수 없습니다 `None` . 그러나 숫자 값과

`None` 열거된 상수 간의 비트 비교가 아닌 논리적 비교를 수행하여 숫자 값의 비트가 설정되었는지 여부를 확인할 수 있습니다.

플래그 열거형 대신 값 열거형을 만드는 경우에도 열거형 상수는 `None` 만드는 것이 좋습니다. 그 이유는 기본적으로 열거형에 사용되는 메모리가 공용 언어 런타임에 의해 0으로 초기화하기 때문입니다. 따라서 값이 0인 상수는 정의하지 않으면 열거형을 만들 때 잘못된 값이 포함됩니다.

애플리케이션에서 표시해야 하는 명백한 기본 사례가 있는 경우 값이 0인 열거형 상수로 기본값을 나타내는 것이 좋습니다. 기본 사례가 없는 경우, 다른 열거형 상수로 표현되지 않는 경우를 나타내기 위해 값이 0인 열거형 상수를 사용하는 것을 고려하십시오.

- 열거형 자체의 상태를 미러링하기 위해서만 열거형 값을 정의하지 마세요. 예를 들어 열거형의 끝을 표시하는 열거형 상수만 정의하지 마세요. 열거형의 마지막 값을 확인해야 하는 경우 해당 값을 명시적으로 확인합니다. 또한 범위 내의 모든 값이 유효한 경우 첫 번째 및 마지막 열거 상수에 대한 범위 검사를 수행할 수 있습니다.
- 나중에 사용하도록 예약된 열거형 상수를 지정하지 마세요.
- 열거형 상수 값을 값으로 사용하는 메서드 또는 속성을 정의할 때 값의 유효성을 검사하는 것이 좋습니다. 그 이유는 해당 숫자 값이 열거형에 정의되지 않은 경우에도 숫자 값을 열거형 형식으로 캐스팅할 수 있기 때문입니다.

예시

다음 예제는 `FlagsAttribute` 특성의 사용을 보여주며, Enum 선언에서 `FlagsAttribute`를 사용하는 ToString 방법에 미치는 영향을 보여줍니다.

C#

```
using System;

class Example
{
    // Define an Enum without FlagsAttribute.
    enum SingleHue : short
    {
        None = 0,
        Black = 1,
        Red = 2,
        Green = 4,
        Blue = 8
    };

    // Define an Enum with FlagsAttribute.
    [Flags]
    enum MultiHue : short
```

```
    {
        None = 0,
        Black = 1,
        Red = 2,
        Green = 4,
        Blue = 8
    };

    static void Main()
    {
        // Display all possible combinations of values.
        Console.WriteLine(
             "All possible combinations of values without FlagsAttribute:");
        for (int val = 0; val <= 16; val++)
            Console.WriteLine("{0,3} - {1:G}", val, (SingleHue)val);

        // Display all combinations of values, and invalid values.
        Console.WriteLine(
             "\nAll possible combinations of values with FlagsAttribute:");
        for (int val = 0; val <= 16; val++)
            Console.WriteLine("{0,3} - {1:G}", val, (MultiHue)val);
    }
}
// The example displays the following output:
//       All possible combinations of values without FlagsAttribute:
//         0 - None
//         1 - Black
//         2 - Red
//         3 - 3
//         4 - Green
//         5 - 5
//         6 - 6
//         7 - 7
//         8 - Blue
//         9 - 9
//        10 - 10
//        11 - 11
//        12 - 12
//        13 - 13
//        14 - 14
//        15 - 15
//        16 - 16
//
//       All possible combinations of values with FlagsAttribute:
//         0 - None
//         1 - Black
//         2 - Red
//         3 - Black, Red
//         4 - Green
//         5 - Black, Green
//         6 - Red, Green
//         7 - Black, Red, Green
//         8 - Blue
//         9 - Black, Blue
//        10 - Red, Blue
```

```
//        11 - Black, Red, Blue
//        12 - Green, Blue
//        13 - Black, Green, Blue
//        14 - Red, Green, Blue
//        15 - Black, Red, Green, Blue
//        16 - 16
```

앞의 예제에서는 두 가지 색 관련 열거형, `SingleHue`와 `MultiHue`를 정의합니다. 후자에는 특성이 `FlagsAttribute` 있습니다. 전자는 그렇지 않습니다. 이 예제에서는 열거형 형식의 기본 값을 나타내지 않는 정수 등 정수 범위가 열거형 형식 및 해당 문자열 표현으로 캐스팅될 때의 동작 차이를 보여 줍니다. 예를 들어, 3은 `SingleHue` 멤버의 기본 값이 아니기 때문에 `SingleHue` 값으로 나타낼 수 없습니다. 그러나 `FlagsAttribute` 특성을 사용하면 3을 `Black, Red`의 `MultiHue` 값으로 나타낼 수 있습니다.

다음 예제에서는 특성을 사용하여 다른 열거형 `FlagsAttribute` 을 정의하고 비트 논리 및 같음 연산자를 사용하여 하나 이상의 비트 필드가 열거형 값에 설정되어 있는지 여부를 확인하는 방법을 보여 줍니다. 메서드를 Enum.HasFlag 사용하여 이 작업을 수행할 수도 있지만 이 예제에는 표시되지 않습니다.

C#

```
using System;

[Flags]
public enum PhoneService
{
    None = 0,
    LandLine = 1,
    Cell = 2,
    Fax = 4,
    Internet = 8,
    Other = 16
}

public class Example1
{
    public static void Main()
    {
        // Define three variables representing the types of phone service
        // in three households.
        var household1 = PhoneService.LandLine | PhoneService.Cell |
                         PhoneService.Internet;
        var household2 = PhoneService.None;
        var household3 = PhoneService.Cell | PhoneService.Internet;

        // Store the variables in an array for ease of access.
        PhoneService[] households = { household1, household2, household3 };

        // Which households have no service?
        for (int ctr = 0; ctr < households.Length; ctr++)
```

```
            Console.WriteLine($"Household {ctr + 1} has phone service:" +
                $"{(households[ctr] == PhoneService.None ? "No" : "Yes")}");
        Console.WriteLine();

        // Which households have cell phone service?
        for (int ctr = 0; ctr < households.Length; ctr++)
            Console.WriteLine($"Household {ctr + 1} has cell phone service: " +
                $"{((households[ctr] & PhoneService.Cell) == PhoneService.Cell ?
"Yes" : "No")}");
        Console.WriteLine();

        // Which households have cell phones and land lines?
        var cellAndLand = PhoneService.Cell | PhoneService.LandLine;
        for (int ctr = 0; ctr < households.Length; ctr++)
            Console.WriteLine($"Household {ctr + 1} has cell and land line
service: " +
                $"{((households[ctr] & cellAndLand) == cellAndLand ? "Yes" :
"No")}");
        Console.WriteLine();

        // List all types of service of each household?//
        for (int ctr = 0; ctr < households.Length; ctr++)
            Console.WriteLine($"Household {ctr + 1} has: {households[ctr]:G}");
        Console.WriteLine();
    }
}
// The example displays the following output:
//    Household 1 has phone service: Yes
//    Household 2 has phone service: No
//    Household 3 has phone service: Yes
//
//    Household 1 has cell phone service: Yes
//    Household 2 has cell phone service: No
//    Household 3 has cell phone service: Yes
//
//    Household 1 has cell and land line service: Yes
//    Household 2 has cell and land line service: No
//    Household 3 has cell and land line service: No
//
//    Household 1 has: LandLine, Cell, Internet
//    Household 2 has: None
//    Household 3 has: Cell, Internet
```

이벤트를 처리하고 발생시키기

.NET의 이벤트는 대리자 모델을 기반으로 합니다. 대리자 모델은 구독자가 공급자에 등록하고 공급자로부터 알림을 받을 수 있도록 하는 관찰자 디자인 패턴을 따릅니다. 이벤트 발신자는 이벤트가 발생할 때 알림을 푸시합니다. 이벤트 수신기는 응답을 정의합니다. 이 문서에서는 대리자 모델의 주요 구성 요소, 애플리케이션에서 이벤트를 사용하는 방법 및 코드에서 이벤트를 구현하는 방법을 설명합니다.

이벤트 발신자를 사용하여 이벤트를 트리거하십시오

이벤트는 동작 발생을 알리기 위해 개체에서 보낸 메시지입니다. 이 작업은 단추 누르기 등의 사용자 상호 작용이거나 속성 값 변경과 같은 다른 프로그램 논리에서 발생할 수 있습니다. 이벤트를 발생시키는 개체를 *이벤트 발신자*라고 합니다. 이벤트 발신자는 이벤트 발생 이벤트를 수신(처리)하는 개체 또는 메서드를 모릅니다. 이벤트는 보통 이벤트 송신자의 구성원입니다. 예를 들어 Elapsed 이벤트는 클래스의 Timer 멤버이며 PropertyChanged , 이벤트는 인터페이스를 구현 INotifyPropertyChanged 하는 클래스의 멤버입니다.

이벤트를 정의하려면 이벤트 클래스의 서명에 C# 이벤트 또는 Visual Basic Event 키워드를 사용하고 이벤트에 대한 대리자 유형을 지정합니다. 대리자는 다음 섹션에서 설명합니다.

이벤트를 발생시키려면 이벤트와 연결된 대리자를 호출합니다. 이벤트 발신자는 일반적으로 `On<EventName>`와 같은 이름의 메서드(`Protected Overridable`(Visual Basic))에서 호출을 감싸서 처리합니다. 예: `OnDataReceived`. 이 메서드는 하나의 매개 변수인 형식 EventArgs 의 이벤트 데이터 개체 또는 파생 형식을 사용합니다. 이러한 방식으로 호출을 래핑하면 파생 클래스가 raise 논리를 재정의할 수 있습니다. 이 메서드를 재정의하는 파생 클래스는 등록된 대리자가 이벤트를 수신하도록 기본 클래스 구현을 호출해야 합니다.

다음 예제에서는 이름이 지정된 `ThresholdReached`이벤트를 선언하는 방법을 보여 있습니다. 이 이벤트는 대리자인 EventHandler과 관련되어 있으며, `OnThresholdReached` 메서드에서 호출됩니다.

C#

```
class Counter
{
    public event EventHandler? ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        ThresholdReached?.Invoke(this, e);
    }

    // Provide remaining implementation for the class...
}
```

이벤트 처리기에 대한 대리자 서명 선언

대리자는 메서드에 대한 참조를 보유하는 형식입니다. 대리자는 참조하는 메서드에 대한 반환 형식 및 매개 변수를 보여 주는 서명으로 선언됩니다. 서명과 일치하는 메서드에 대한 참조만 보유할 수 있습니다. 대리자는 형식이 안전한 함수 포인터 또는 콜백과 동일합니다. 대리자 선언은 대리자 클래스를 정의하기에 충분합니다.

대리자는 .NET에서 많은 용도를 사용합니다. 이벤트의 컨텍스트에서 대리자는 이벤트 소스와 이벤트를 처리하는 코드 사이의 중간(또는 포인터와 유사한 메커니즘)입니다. 이전 섹션의 예제와 같이 이벤트 선언에 대리자 형식을 포함하여 대리자를 이벤트와 연결합니다. 대리자에 대한 자세한 내용은 클래스를 참조하세요 Delegate .

.NET은 대부분의 이벤트 시나리오를 지원하기 위해 EventHandler 및 EventHandler<TEventArgs> 대리자를 제공합니다. EventHandler 이벤트 데이터를 포함하지 않는 모든 이벤트에 대리자를 사용합니다. EventHandler<TEventArgs> 이벤트에 대한 데이터를 포함하는 이벤트에 대리자를 사용합니다. 이러한 대리자는 반환 형식 값이 없으며 두 개의 매개 변수(이벤트 원본의 개체 및 이벤트 데이터의 개체)를 사용합니다.

대리자는 멀티캐스트 클래스 개체입니다. 즉, 둘 이상의 이벤트 처리 메서드에 대한 참조를 보유할 수 있습니다. 자세한 내용은 참조 페이지를 참조 Delegate 하세요. 대리자는 이벤트 처리에서 유연성과 세분화된 제어를 제공합니다. 대리자는 이벤트에 대해 등록된 이벤트 처리기 목록을 유지 관리하여 이벤트를 발생시키는 클래스에 대한 이벤트 디스패처 역할을 합니다.

EventHandler 및 EventHandler<TEventArgs> 대리자 타입을 사용하여 필요한 대리자를 정의합니다. 선언에서 `delegate`의 형식 또는 `Delegate`의 형식으로 대리자를 표시합니다. 다음 예제에서는 명명 `ThresholdReachedEventHandler`된 대리자를 선언하는 방법을 보여줍니다.

C#

```
public delegate void ThresholdReachedEventHandler(object sender,
ThresholdReachedEventArgs e);
```

이벤트 데이터 클래스 작업

이벤트와 연결된 데이터는 이벤트 데이터 클래스를 통해 제공할 수 있습니다. .NET은 애플리케이션에서 사용할 수 있는 많은 이벤트 데이터 클래스를 제공합니다. 예를 들어, SerialDataReceivedEventArgs 클래스는 SerialPort.DataReceived 이벤트의 이벤트 데이터 클래스입니다. .NET은 모든 이벤트 데이터 클래스가 접미사로 `EventArgs` 끝나는 명명 패턴을 따릅니다. 이벤트에 대한 대리자를 확인하여 이벤트와 연결된 이벤트 데이터 클래스를 결정합니다. 예를 들어 대리자는 SerialDataReceivedEventHandler 클래스를 SerialDataReceivedEventArgs 매개 변수로 포함합니다.

EventArgs 이 클래스는 일반적으로 이벤트 데이터 클래스의 기본 형식입니다. 이벤트에 연결된 데이터가 없는 경우에도 이 클래스를 사용합니다. 구독자에게 추가 데이터 없이 발생한 일을 알리는 이벤트를 만들 때 클래스를 대리자의 EventArgs 두 번째 매개 변수로 포함합니다. 데이터가 제공되지 않으면 EventArgs.Empty 값을 전달할 수 있습니다. 대리자는 EventHandler 클래스를 EventArgs 매개 변수로 포함합니다.

EventArgs 클래스에서 파생되는 클래스를 생성하여 이벤트 관련 데이터를 전달하는 데 필요한 멤버를 제공할 수 있습니다. 일반적으로 .NET과 동일한 명명 패턴을 사용하고 접미사로 `EventArgs` 이벤트 데이터 클래스 이름을 종료해야 합니다.

다음 예제에서는 발생 중인 이벤트와 관련된 속성을 포함하는 명명 `ThresholdReachedEventArgs` 된 이벤트 데이터 클래스를 보여 줍니다.

C#

```
public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

처리기를 사용하여 이벤트에 응답

이벤트에 응답하려면 이벤트 수신기에서 이벤트 처리기 메서드를 정의합니다. 이 메서드는 처리 중인 이벤트에 대한 대리자의 서명과 일치해야 합니다. 이벤트 처리기에서 사용자가 단추를 누른 후 사용자 입력을 수집하는 등 이벤트가 발생할 때 필요한 작업을 수행합니다. 이벤트가 발생할 때 알림을 받으려면 이벤트 처리기 메서드가 이벤트를 구독해야 합니다.

다음 예제는 `c_ThresholdReached`라는 이벤트 처리기 메서드를 보여주며, 이는 EventHandler 대리자의 서명과 일치합니다. 메서드는 이벤트를 구독합니다.`ThresholdReached`

C#

```
class Program
{
    static void Main()
    {
        var c = new Counter();
        c.ThresholdReached += c_ThresholdReached;

        // Provide remaining implementation for the class...
    }

    static void c_ThresholdReached(object? sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
```

```
    }
}
```

정적 및 동적 이벤트 처리기 사용

.NET을 사용하면 구독자가 이벤트 알림을 정적으로 또는 동적으로 등록할 수 있습니다. 정적 이벤트 처리기는 이벤트가 처리되는 클래스의 전체 수명 동안 적용됩니다. 동적 이벤트 처리기는 일반적으로 일부 조건부 프로그램 논리에 대한 응답으로 프로그램 실행 중에 명시적으로 활성화되고 비활성화됩니다. 이벤트 알림이 특정 조건에서만 필요하거나 런타임 조건에서 호출할 특정 처리기가 결정되는 경우 동적 처리기를 사용할 수 있습니다. 이전 섹션의 예제에서는 이벤트 처리기를 동적으로 추가하는 방법을 보여 줍니다. 자세한 내용은 이벤트 (Visual Basic의 경우) 및 이벤트 (C#)를 참조하세요.

여러 이벤트를 발생시키다

클래스가 여러 이벤트를 발생하면 컴파일러는 이벤트 대리자 인스턴스당 하나의 필드를 생성합니다. 이벤트 수가 큰 경우 대리자당 하나의 필드의 스토리지 비용이 허용되지 않을 수 있습니다. 이러한 시나리오에서 .NET은 이벤트 대리자를 저장하기 위해 선택한 다른 데이터 구조와 함께 사용할 수 있는 이벤트 속성을 제공합니다.

이벤트 속성은 이벤트 접근자가 함께 포함된 이벤트 선언으로 구성됩니다. 이벤트 접근자는 스토리지 데이터 구조에서 이벤트 대리자 인스턴스를 추가하거나 제거하기 위해 정의하는 메서드입니다.

> ⓘ **참고 항목**
>
> 각 이벤트 대리자를 호출하기 전에 검색해야 하므로 이벤트 속성은 이벤트 필드보다 느립니다.

절충은 메모리와 속도 사이에 있습니다. 클래스가 자주 발생하지 않는 많은 이벤트를 정의하는 경우 이벤트 속성을 구현해야 합니다. 자세한 내용은 이벤트 속성을 사용하여 여러 이벤트 선언을 참조하세요.

관련 작업 살펴보기

다음 리소스는 이벤트 작업과 관련된 다른 작업 및 개념을 설명합니다.

- 이벤트 발생 및 사용: 이벤트를 발생시키고 사용하는 예제를 찾습니다.
- 이벤트 속성을 사용하여 여러 이벤트 선언: 이벤트 속성을 사용하여 여러 이벤트를 선언하는 방법을 알아봅니다.

- 관찰자 디자인 패턴 살펴보기: 구독자가 공급자에 등록하고 공급자로부터 알림을 받을 수 있도록 하는 디자인 패턴을 검토합니다.

사양 참조 검토

사양 참조 설명서는 이벤트 처리를 지원하는 API에 사용할 수 있습니다.

테이블 확장

API 이름	API 형식	참고 문헌
EventHandler	대리인	EventHandler
EventHandler<TEventArgs>	대리인	EventHandler<TEventArgs>
EventArgs	클래스	EventArgs
대리인	클래스	Delegate

관련 콘텐츠

- 이벤트(Visual Basic)
- 이벤트(C# 프로그래밍 가이드)
- 방법: 이벤트 발행 및 사용
- 방법: 이벤트 속성을 사용하여 여러 이벤트 선언

Last updated on 2026. 03. 23.

이벤트 발생 및 사용

이 문서에서는 대리자, EventHandler 대리자 및 사용자 지정 대리자를 사용하여 EventHandler<TEventArgs> .NET의 이벤트로 작업하는 방법을 보여 줍니다. 여기에는 데이터가 있는 이벤트와 없는 이벤트에 대한 예제가 있습니다.

사전 요구 사항

이벤트 문서의 개념을 숙지합니다.

데이터 없이 이벤트 발생시키기

이러한 단계는 실행 중인 합계가 임계값에 `Counter` 도달하거나 초과할 때 `ThresholdReached` 이벤트를 발생시키는 클래스를 만듭니다.

1. 대리자를 사용하여 이벤트를 선언합니다 EventHandler .

   이벤트가 처리기에 데이터를 전달하지 않는 경우에 사용합니다 `EventHandler` .

   C#

   ```
   public event EventHandler? ThresholdReached;
   ```

2. `protected virtual` 메서드를 추가하여 이벤트를 발생시키고, Visual Basic에서는 `Protected Overridable`를 사용합니다.

   이 패턴을 사용하면 파생 클래스가 대리자를 직접 호출하지 않고 이벤트 발생 동작을 재정의할 수 있습니다. C#에서 null 조건부 연산자(`?.`)를 사용하여 구독자 없음을 방지합니다 (Visual Basic `RaiseEvent` 에서는 자동으로 처리함).

   C#

   ```
   protected virtual void OnThresholdReached(EventArgs e)
   {
       ThresholdReached?.Invoke(this, e);
   }
   ```

3. 조건이 충족되면 raise 메서드를 호출합니다.

   이 이벤트는 데이터를 전달하지 않으므로 전달 Empty 합니다.

   C#

```
if (_total >= _threshold)
{
    OnThresholdReached(EventArgs.Empty);
}
```

4. 연산자를 사용하여 `+=` 이벤트를 구독합니다(Visual Basic의 `AddHandler`경우).

C#

```
c.ThresholdReached += c_ThresholdReached;
```

5. 이벤트 처리기 메서드를 정의합니다.

   해당 서명은 대리자와 EventHandler 일치해야 합니다. 첫 번째 매개 변수는 이벤트 원본이고 두 번째 매개 변수는 다음과 같습니다 EventArgs.

C#

```
static void c_ThresholdReached(object? sender, EventArgs e)
{
    Console.WriteLine("The threshold was reached.");
    Environment.Exit(0);
}
```

다음 예제에서는 전체 구현을 보여줍니다.

C#

```
class EventNoData
{
    static void Main()
    {
        Counter c = new(new Random().Next(10));
        c.ThresholdReached += c_ThresholdReached;

        Console.WriteLine("press 'a' key to increase total");
        while (Console.ReadKey(true).KeyChar == 'a')
        {
            Console.WriteLine("adding one");
            c.Add(1);
        }
    }

    static void c_ThresholdReached(object? sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
        Environment.Exit(0);
    }
}
```

```
class Counter(int passedThreshold)
{
    private readonly int _threshold = passedThreshold;
    private int _total;

    public void Add(int x)
    {
        _total += x;
        if (_total >= _threshold)
        {
            OnThresholdReached(EventArgs.Empty);
        }
    }

    protected virtual void OnThresholdReached(EventArgs e)
    {
        ThresholdReached?.Invoke(this, e);
    }

    public event EventHandler? ThresholdReached;
}
```

데이터를 사용하여 이벤트를 발생시키다

이러한 단계는 이전 `Counter` 예제를 확장하여 임계값과 도달한 시간을 포함하는 이벤트를 발생합니다.

1. 에서 상속되는 이벤트 데이터 클래스를 정의합니다.EventArgs

   처리기에 전달하려는 각 데이터 조각에 대한 속성을 추가합니다.

   C#

   ```
   public class ThresholdReachedEventArgs : EventArgs
   {
       public int Threshold { get; set; }
       public DateTime TimeReached { get; set; }
   }
   ```

2. 대리자를 사용하여 이벤트를 선언하고 EventHandler<TEventArgs> 이벤트 데이터 클래스를 형식 인수로 전달합니다.

   C#

   ```
   public event EventHandler<ThresholdReachedEventArgs>? ThresholdReached;
   ```

3. `protected virtual` 메서드를 추가하여 이벤트를 발생시킵니다 (`Protected Overridable`는 Visual Basic).

   이 패턴을 사용하면 파생 클래스가 대리자를 직접 호출하지 않고 이벤트 발생 동작을 재정의할 수 있습니다. C#에서 null 조건부 연산자(`?.`)를 사용하여 구독자 없음을 방지합니다 (Visual Basic `RaiseEvent` 에서는 자동으로 처리함).

   C#

   ```
   protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
   {
       ThresholdReached?.Invoke(this, e);
   }
   ```

4. 이벤트 데이터 개체를 채우고 조건이 충족되면 raise 메서드를 호출합니다.

   C#

   ```
   if (_total >= _threshold)
   {
       ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
       args.Threshold = _threshold;
       args.TimeReached = DateTime.Now;
       OnThresholdReached(args);
   }
   ```

5. 연산자를 사용하여 `+=` 이벤트를 구독합니다(Visual Basic의 `AddHandler`경우).

   C#

   ```
   c.ThresholdReached += c_ThresholdReached;
   ```

6. 이벤트 처리기를 정의합니다.

   두 번째 매개 변수 유형은 EventArgs 대신 `ThresholdReachedEventArgs`으로, 이는 처리기가 이벤트 데이터를 읽을 수 있게 해줍니다.

   C#

   ```
   static void c_ThresholdReached(object? sender, ThresholdReachedEventArgs e)
   {
       Console.WriteLine($"The threshold of {e.Threshold} was reached at
   {e.TimeReached}.");
       Environment.Exit(0);
   }
   ```

다음 예제에서는 전체 구현을 보여줍니다.

C#

```
class EventWithData
{
    static void Main()
    {
        CounterWithData c = new(new Random().Next(10));
        c.ThresholdReached += c_ThresholdReached;

        Console.WriteLine("press 'a' key to increase total");
        while (Console.ReadKey(true).KeyChar == 'a')
        {
            Console.WriteLine("adding one");
            c.Add(1);
        }
    }

    static void c_ThresholdReached(object? sender, ThresholdReachedEventArgs e)
    {
        Console.WriteLine($"The threshold of {e.Threshold} was reached at
{e.TimeReached}.");
        Environment.Exit(0);
    }
}

class CounterWithData(int passedThreshold)
{
    private readonly int _threshold = passedThreshold;
    private int _total;

    public void Add(int x)
    {
        _total += x;
        if (_total >= _threshold)
        {
            ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
            args.Threshold = _threshold;
            args.TimeReached = DateTime.Now;
            OnThresholdReached(args);
        }
    }

    protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
    {
        ThresholdReached?.Invoke(this, e);
    }

    public event EventHandler<ThresholdReachedEventArgs>? ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
```

```
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

이벤트에 대한 사용자 지정 대리자 선언

제네릭을 사용할 수 없는 레거시 코드에서 클래스를 사용할 수 있도록 하는 등 드문 시나리오에서만 사용자 지정 대리자를 선언합니다. 대부분의 경우 이전 섹션에 표시된 대로 사용합니다 EventHandler<TEventArgs> .

1. 사용자 지정 대리자 형식을 선언합니다.

   대리자 서명은 이벤트 처리기 서명과 일치해야 합니다. 두 매개 변수: 이벤트 원본 (`object`Visual Basic의 `Object`경우) 및 이벤트 데이터 클래스:

   C#

   ```
   public delegate void ThresholdReachedEventHandler(object sender,
   ThresholdReachedEventArgs e);
   ```

2. 사용자 지정 대리자 형식을 사용하여 이벤트를 선언하고 EventHandler<TEventArgs>대신 이를 활용하십시오.

   C#

   ```
   public event ThresholdReachedEventHandler? ThresholdReached;
   ```

3. `protected virtual` 메서드(`Protected Overridable`는 Visual Basic의 경우) 를 추가하여 이 벤트를 발생시킵니다.

   C#에서 null 조건부 연산자(`?.`)를 사용하여 구독자 없음을 방지합니다(Visual Basic `RaiseEvent` 에서는 자동으로 처리함).

   C#

   ```
   protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
   {
       ThresholdReached?.Invoke(this, e);
   }
   ```

4. 이벤트 데이터 개체를 채우고 조건이 충족되면 raise 메서드를 호출합니다.

   C#

```
if (_total >= _threshold)
{
    ThresholdReachedEventArgs args = new();
    args.Threshold = _threshold;
    args.TimeReached = DateTime.Now;
    OnThresholdReached(args);
}
```

5. 연산자를 사용하여 `+=` 이벤트를 구독합니다(Visual Basic의 `AddHandler`경우).

C#

```
c.ThresholdReached += c_ThresholdReached;
```

6. 이벤트 처리기를 정의합니다.

   처리기 서명은 보낸 사람 및 두 번째 매개 변수의 이벤트 데이터 클래스에 대한 사용자 지정 대리`object` 자와 일치해야 합니다.

C#

```
static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
{
    Console.WriteLine($"The threshold of {e.Threshold} was reached at
{e.TimeReached}.");
    Environment.Exit(0);
}
```

다음 예제에서는 전체 구현을 보여줍니다.

C#

```
class EventWithDelegate
{
    static void Main()
    {
        CounterWithDelegate c = new(new Random().Next(10));
        c.ThresholdReached += c_ThresholdReached;

        Console.WriteLine("press 'a' key to increase total");
        while (Console.ReadKey(true).KeyChar == 'a')
        {
            Console.WriteLine("adding one");
            c.Add(1);
        }
    }

    static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
    {
```

```
        Console.WriteLine($"The threshold of {e.Threshold} was reached at
{e.TimeReached}.");
        Environment.Exit(0);
    }
}

class CounterWithDelegate(int passedThreshold)
{
    private readonly int _threshold = passedThreshold;
    private int _total;

    public void Add(int x)
    {
        _total += x;
        if (_total >= _threshold)
        {
            ThresholdReachedEventArgs args = new();
            args.Threshold = _threshold;
            args.TimeReached = DateTime.Now;
            OnThresholdReached(args);
        }
    }

    protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
    {
        ThresholdReached?.Invoke(this, e);
    }

    public event ThresholdReachedEventHandler? ThresholdReached;
}

public delegate void ThresholdReachedEventHandler(object sender,
ThresholdReachedEventArgs e);
```

관련 콘텐츠

- 이벤트

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 31.

이벤트 속성을 사용하여 여러 이벤트 선언

클래스가 필드와 유사한 이벤트 선언을 사용하여 많은 이벤트를 정의하는 경우 컴파일러는 각각에 대한 지원 필드를 생성합니다. 해당 클래스의 모든 인스턴스는 구독되지 않은 이벤트에 대해서도 해당 필드의 모든 메모리를 할당합니다. 많은 이벤트로 인해 인스턴스당 이 오버헤드로 인해 메모리가 낭비됩니다.

이 메모리 오버헤드를 방지하려면 대리자 컬렉션에서 지원되는 이벤트 속성을 사용합니다. 컬렉션은 이벤트 키별로 대리자를 설정, 액세스 및 검색하는 메서드를 제공해야 합니다. EventHandlerList는 이 용도로 설계되었지만, Hashtable 또는 DictionaryBase에서 파생된 클래스를 사용할 수도 있습니다. 컬렉션의 구현 세부 정보를 비공개로 유지합니다.

각 이벤트 속성은 add 접근자와 remove 접근자를 정의합니다. add 접근자는 대리자 컬렉션에 입력 대리자를 추가하고 제거 접근자는 제거합니다. 두 접근자 모두 미리 정의된 키를 사용하여 컬렉션에서 인스턴스를 추가하고 제거합니다.

사전 요구 사항

이벤트 문서의 개념을 숙지합니다.

이벤트 속성을 사용하여 여러 이벤트 정의

이 단계에서는 10개의 이벤트 속성을 노출하는 `Sensor` 클래스를 생성하며, 이 모든 속성은 단일 `EventHandlerList`에 의해 지원됩니다.

1. 에서 상속되는 이벤트 데이터 클래스를 정의합니다.EventArgs

   처리기에 전달하려는 각 데이터 조각에 대한 속성을 추가합니다.

   C#

   ```
   public class SensorEventArgs(string sensorId, double value) : EventArgs
   {
       public string SensorId { get; } = sensorId;
       public double Value { get; } = value;
   }
   ```

2. EventHandlerList 대리자를 저장할 필드를 선언합니다.

   C#

   ```
   protected EventHandlerList listEventDelegates = new();
   ```

3. 각 이벤트에 대한 고유 키를 선언합니다.

   `EventHandlerList` 는 키별로 대리자를 저장합니다. 각 키에 대해 `static readonly` 개체(`Shared ReadOnly` Visual Basic에서는 개체)를 사용하세요. 개체 ID는 각 키의 고유성을 보장합니다.

   C#

```
static readonly object temperatureChangedKey = new();
static readonly object humidityChangedKey = new();
static readonly object pressureChangedKey = new();
static readonly object batteryLowKey = new();
static readonly object signalLostKey = new();
static readonly object signalRestoredKey = new();
static readonly object thresholdExceededKey = new();
static readonly object calibrationRequiredKey = new();
static readonly object dataReceivedKey = new();
static readonly object errorDetectedKey = new();
```

4. 사용자 지정 추가 및 제거 접근자를 사용하여 각 이벤트를 이벤트 속성으로 정의합니다.

   각 접근자는 해당 이벤트의 키를 사용하여 목록에서 `AddHandler` 또는 `RemoveHandler`을 호출합니다. C#에서는 키로 목록에서 대리자를 검색하고 호출하는 `protected virtual` raise 메서드를 추가합니다. Visual Basic에서, `Custom Event` 선언에는 이미 그러한 `RaiseEvent` 블록이 포함되어 있습니다.

   C#

```
public event EventHandler<SensorEventArgs> TemperatureChanged
{
    add    { listEventDelegates.AddHandler(temperatureChangedKey, value); }
    remove { listEventDelegates.RemoveHandler(temperatureChangedKey, value); }
}
protected virtual void OnTemperatureChanged(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[temperatureChangedKey])?.I
nvoke(this, e);
```

   해당 키를 사용하여 각 이벤트에 대해 이 패턴을 반복합니다.

5. 연산자를 사용하여 `+=` 이벤트를 구독합니다(Visual Basic의 `AddHandler`경우).

   C#

```
sensor.TemperatureChanged += Sensor_TemperatureChanged;
```

6. 이벤트 처리기를 정의합니다.

서명은 EventHandler<TEventArgs> 델리게이트와 일치해야 합니다—`object` sender와 이벤트 데이터 클래스가 두 번째 매개 변수로 사용됩니다.

C#

```
static void Sensor_TemperatureChanged(object? sender, SensorEventArgs e) =>
    Console.WriteLine($"Sensor {e.SensorId}: temperature changed to
{e.Value}.");
```

다음 예제에서는 전체 `Sensor` 클래스 구현을 보여줍니다.

C#

```
using System.ComponentModel;

public class SensorEventArgs(string sensorId, double value) : EventArgs
{
    public string SensorId { get; } = sensorId;
    public double Value { get; } = value;
}

// Sensor defines 10 event properties backed by a single EventHandlerList.
// EventHandlerList is memory-efficient for classes with many events: it only
// allocates storage for events that have active subscribers.
class Sensor
{
    protected EventHandlerList listEventDelegates = new();

    static readonly object temperatureChangedKey = new();
    static readonly object humidityChangedKey = new();
    static readonly object pressureChangedKey = new();
    static readonly object batteryLowKey = new();
    static readonly object signalLostKey = new();
    static readonly object signalRestoredKey = new();
    static readonly object thresholdExceededKey = new();
    static readonly object calibrationRequiredKey = new();
    static readonly object dataReceivedKey = new();
    static readonly object errorDetectedKey = new();

    public event EventHandler<SensorEventArgs> TemperatureChanged
    {
        add    { listEventDelegates.AddHandler(temperatureChangedKey, value); }
        remove { listEventDelegates.RemoveHandler(temperatureChangedKey, value); }
    }
    protected virtual void OnTemperatureChanged(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[temperatureChangedKey])?.Invoke
(this, e);

    public event EventHandler<SensorEventArgs> HumidityChanged
    {
        add    { listEventDelegates.AddHandler(humidityChangedKey, value); }
```

```
        remove { listEventDelegates.RemoveHandler(humidityChangedKey, value); }
    }
    protected virtual void OnHumidityChanged(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[humidityChangedKey])?.Invoke(th
is, e);

    public event EventHandler<SensorEventArgs> PressureChanged
    {
        add    { listEventDelegates.AddHandler(pressureChangedKey, value); }
        remove { listEventDelegates.RemoveHandler(pressureChangedKey, value); }
    }
    protected virtual void OnPressureChanged(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[pressureChangedKey])?.Invoke(th
is, e);

    public event EventHandler<SensorEventArgs> BatteryLow
    {
        add    { listEventDelegates.AddHandler(batteryLowKey, value); }
        remove { listEventDelegates.RemoveHandler(batteryLowKey, value); }
    }
    protected virtual void OnBatteryLow(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[batteryLowKey])?.Invoke(this,
e);

    public event EventHandler<SensorEventArgs> SignalLost
    {
        add    { listEventDelegates.AddHandler(signalLostKey, value); }
        remove { listEventDelegates.RemoveHandler(signalLostKey, value); }
    }
    protected virtual void OnSignalLost(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[signalLostKey])?.Invoke(this,
e);

    public event EventHandler<SensorEventArgs> SignalRestored
    {
        add    { listEventDelegates.AddHandler(signalRestoredKey, value); }
        remove { listEventDelegates.RemoveHandler(signalRestoredKey, value); }
    }
    protected virtual void OnSignalRestored(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[signalRestoredKey])?.Invoke(thi
s, e);

    public event EventHandler<SensorEventArgs> ThresholdExceeded
    {
        add    { listEventDelegates.AddHandler(thresholdExceededKey, value); }
        remove { listEventDelegates.RemoveHandler(thresholdExceededKey, value); }
    }
    protected virtual void OnThresholdExceeded(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[thresholdExceededKey])?.Invoke(
```

```
this, e);

    public event EventHandler<SensorEventArgs> CalibrationRequired
    {
        add    { listEventDelegates.AddHandler(calibrationRequiredKey, value); }
        remove { listEventDelegates.RemoveHandler(calibrationRequiredKey, value); }
    }
    protected virtual void OnCalibrationRequired(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[calibrationRequiredKey])?.Invok
e(this, e);

    public event EventHandler<SensorEventArgs> DataReceived
    {
        add    { listEventDelegates.AddHandler(dataReceivedKey, value); }
        remove { listEventDelegates.RemoveHandler(dataReceivedKey, value); }
    }
    protected virtual void OnDataReceived(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[dataReceivedKey])?.Invoke(this,
e);

    public event EventHandler<SensorEventArgs> ErrorDetected
    {
        add    { listEventDelegates.AddHandler(errorDetectedKey, value); }
        remove { listEventDelegates.RemoveHandler(errorDetectedKey, value); }
    }
    protected virtual void OnErrorDetected(SensorEventArgs e) =>

((EventHandler<SensorEventArgs>?)listEventDelegates[errorDetectedKey])?.Invoke(this
, e);
}
```

관련 콘텐츠

- System.ComponentModel.EventHandlerList
- Events

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 30.

관찰자 디자인 패턴

관찰자 디자인 패턴을 사용하면 구독자가 공급자에 등록하고 공급자로부터 알림을 받을 수 있습니다. 푸시 기반 알림이 필요한 모든 시나리오에 적합합니다. 이 패턴은 *공급자*(*주체* 또는 *관찰 가능* 개체라고도 함) 및 0개, 하나 이상의 *관찰자를 정의합니다*. 관찰자는 공급자에 등록하고 미리 정의된 조건, 이벤트 또는 상태 변경이 발생할 때마다 공급자는 대리자를 호출하여 모든 관찰자에게 자동으로 알깁니다. 이 메서드 호출에서 공급자는 관찰자에게 현재 상태 정보를 제공할 수도 있습니다. .NET에서 관찰자 디자인 패턴은 제네릭 System.IObservable<T> 및 System.IObserver<T> 인터페이스를 구현하여 적용됩니다. 제네릭 형식 매개 변수는 알림 정보를 제공하는 형식을 나타냅니다.

패턴을 적용하는 경우

관찰자 디자인 패턴은 데이터 원본(비즈니스 논리) 계층과 사용자 인터페이스(표시) 계층과 같은 서로 다른 두 구성 요소 또는 애플리케이션 계층 간의 깨끗한 분리를 지원하기 때문에 분산 푸시 기반 알림에 적합합니다. 공급자가 콜백을 사용하여 클라이언트에 현재 정보를 제공할 때마다 패턴을 구현할 수 있습니다.

패턴을 구현하려면 다음 세부 정보를 제공해야 합니다.

- 관찰자에게 알림을 보내는 개체인 공급자 또는 주체입니다. 공급자는 인터페이스를 구현하는 클래스 또는 구조체입니다 IObservable<T> . 공급자는 공급자로부터 알림을 받으려는 관찰자가 호출하는 단일 메서드 IObservable<T>.Subscribe를 구현해야 합니다.

- 공급자로부터 알림을 받는 개체인 관찰자입니다. 관찰자는 인터페이스를 구현하는 클래스 또는 구조체입니다 IObserver<T> . 관찰자는 세 가지 메서드를 구현해야 하며, 이 메서드는 모두 공급자가 호출합니다.
  - IObserver<T>.OnNext새 정보 또는 현재 정보를 관찰자에게 제공하는 입니다.
  - IObserver<T>.OnError- 관찰자에게 오류가 발생했음을 알릴 수 있습니다.
  - IObserver<T>.OnCompleted- 공급자가 알림 보내기를 완료했음을 나타냅니다.

- 공급자가 관찰자를 추적할 수 있도록 하는 메커니즘입니다. 일반적으로 공급자는 알림을 구독한 System.Collections.Generic.List<T> 구현에 대한 참조를 저장하기 위해 IObserver<T> 객체와 같은 컨테이너 객체를 사용합니다. 이 목적을 위해 스토리지 컨테이너를 사용하면 공급자가 0부터 무제한 관찰자까지 처리할 수 있습니다. 관찰자가 알림을 받는 순서는 정의되지 않았습니다. 공급자는 모든 메서드를 사용하여 주문을 결정할 수 있습니다.

- IDisposable 알림이 완료되면 공급자가 관찰자를 제거할 수 있도록 하는 구현입니다. 관찰자는 IDisposable 메서드를 통해 Subscribe 구현에 대한 참조를 받으므로, 공급자가 알림을 보내는 것을 완료하기 전에 IDisposable.Dispose 메서드를 호출하여 구독을 취소할 수도 있습니다.

- 공급자가 관찰자에게 보내는 데이터를 포함하는 개체입니다. 이 개체의 형식은 IObservable<T> 및 IObserver<T> 인터페이스의 제네릭 형식 매개 변수에 해당합니다. 이 개체는 IObservable<T> 구현과 같을 수도 있지만, 일반적으로는 별도의 타입입니다.

> ⓘ **참고 항목**
>
> 관찰자 디자인 패턴을 구현하는 것 외에도, IObservable<T> 및 IObserver<T> 인터페이스를 사용하여 구축된 라이브러리를 탐색하는 데 관심이 있을 수 있습니다. 예를 들어 Rx(.NET용 Reactive Extensions) 는 비동기 프로그래밍을 지원하는 일련의 확장 메서드와 LINQ 표준 시퀀스 연산자로 구성됩니다.

패턴 구현

다음 예제에서는 관찰자 디자인 패턴을 사용하여 공항 수하물 찾는 정보 시스템을 구현합니다. 클래스는 `BaggageInfo` 도착 항공편 및 각각의 항공편에서 수하물을 픽업할 수 있는 수하물 회전대에 대한 정보를 제공합니다. 다음 예제에 나와 있습니다.

C#

```
namespace Observables.Example;

public readonly record struct BaggageInfo(
    int FlightNumber,
    string From,
    int Carousel);
```

`BaggageHandler` 클래스는 도착 항공편 및 수하물 회수대에 대한 정보를 수신할 책임이 있습니다. 내부적으로 다음 두 컬렉션을 유지 관리합니다.

- `_observers`: 업데이트된 정보를 관찰하는 클라이언트의 컬렉션입니다.
- `_flights`: 항공편과 할당된 수하물 컨베이어 벨트의 모음입니다.

클래스의 `BaggageHandler` 소스 코드는 다음 예제에 나와 있습니다.

C#

```
namespace Observables.Example;

public sealed class BaggageHandler : IObservable<BaggageInfo>
{
    private readonly HashSet<IObserver<BaggageInfo>> _observers = new();
    private readonly HashSet<BaggageInfo> _flights = new();

    public IDisposable Subscribe(IObserver<BaggageInfo> observer)
    {
```

```
        // Check whether observer is already registered. If not, add it.
        if (_observers.Add(observer))
        {
            // Provide observer with existing data.
            foreach (BaggageInfo item in _flights)
            {
                observer.OnNext(item);
            }
        }

        return new Unsubscriber<BaggageInfo>(_observers, observer);
    }

    // Called to indicate all baggage is now unloaded.
    public void BaggageStatus(int flightNumber) =>
        BaggageStatus(flightNumber, string.Empty, 0);

    public void BaggageStatus(int flightNumber, string from, int carousel)
    {
        var info = new BaggageInfo(flightNumber, from, carousel);

        // Carousel is assigned, so add new info object to list.
        if (carousel > 0 && _flights.Add(info))
        {
            foreach (IObserver<BaggageInfo> observer in _observers)
            {
                observer.OnNext(info);
            }
        }
        else if (carousel is 0)
        {
            // Baggage claim for flight is done.
            if (_flights.RemoveWhere(
                flight => flight.FlightNumber == info.FlightNumber) > 0)
            {
                foreach (IObserver<BaggageInfo> observer in _observers)
                {
                    observer.OnNext(info);
                }
            }
        }
    }

    public void LastBaggageClaimed()
    {
        foreach (IObserver<BaggageInfo> observer in _observers)
        {
            observer.OnCompleted();
        }

        _observers.Clear();
    }
}
```

업데이트된 정보를 수신하려는 클라이언트는 메서드를 호출합니다 `BaggageHandler.Subscribe` . 클라이언트가 이전에 알림을 구독하지 않은 경우 클라이언트의 IObserver<T> 구현에 대한 참조가 컬렉션에 `_observers` 추가됩니다.

오버로드된 메서드를 `BaggageHandler.BaggageStatus` 호출하여 항공편의 수하물이 언로드 중이거나 더 이상 언로드되지 않음을 나타낼 수 있습니다. 첫 번째 경우, 메서드는 항공편 번호와 항공편이 출발한 공항, 수하물이 내려지는 회전목마를 전달합니다. 두 번째 경우 메서드는 플라이트 번호만 전달됩니다. 언로드되는 수하물을 위해, 메서드는 메서드에 전달된 `BaggageInfo` 정보가 `_flights` 컬렉션에 존재하는지 확인합니다. 그렇지 않은 경우 메서드는 정보를 추가하고 각 관찰자의 `OnNext` 메서드를 호출합니다. 수하물이 더 이상 언로드되지 않는 항공편의 경우, 이 메서드는 해당 항공편의 정보가 컬렉션에 저장 `_flights` 되어 있는지 여부를 확인합니다. 이 경우 메서드는 각 관찰자의 `OnNext` 메서드를 호출한 후 `BaggageInfo` 객체를 `_flights` 컬렉션에서 제거합니다.

하루의 마지막 비행이 착륙하고 수하물이 처리되면 메서드가 `BaggageHandler.LastBaggageClaimed` 호출됩니다. 이 메서드는 각 관찰자의 `OnCompleted` 메서드를 호출하여 모든 알림이 완료되었음을 나타내고 컬렉션을 지웁니다 `_observers` .

공급자의 Subscribe 메서드는 관찰자가 IDisposable 메서드가 호출되기 전에 알림 수신을 중지할 수 있도록 하는 OnCompleted 구현을 반환합니다. 이 `Unsubscriber(Of BaggageInfo)` 클래스의 소스 코드는 다음 예제에 나와 있습니다. 메서드에서 `BaggageHandler.Subscribe` 클래스가 인스턴스화되면 컬렉션에 대한 참조와 컬렉션에 `_observers` 추가된 관찰자에 대한 참조가 전달됩니다. 이러한 참조는 지역 변수에 할당됩니다. 개체의 `Dispose` 메서드가 호출되면 관찰자가 컬렉션에 `_observers` 여전히 존재하는지 확인하고, 이 경우 관찰자를 제거합니다.

C#

```
namespace Observables.Example;

internal sealed class Unsubscriber<BaggageInfo> : IDisposable
{
    private readonly ISet<IObserver<BaggageInfo>> _observers;
    private readonly IObserver<BaggageInfo> _observer;

    internal Unsubscriber(
        ISet<IObserver<BaggageInfo>> observers,
        IObserver<BaggageInfo> observer) => (_observers, _observer) = (observers,
observer);

    public void Dispose() => _observers.Remove(_observer);
}
```

다음 예제에서는 수하물 클레임 정보를 표시하는 기본 클래스인 IObserver<T> 구현인 `ArrivalsMonitor`을 제공합니다. 정보는 원래 도시의 이름으로 사전순으로 표시됩니다. 메서드

`ArrivalsMonitor`는 Visual Basic에서는 `overridable`으로, C#에서는 `virtual`로 표시되므로 파생 클래스에서 재정의할 수 있습니다.

C#

```
namespace Observables.Example;

public class ArrivalsMonitor : IObserver<BaggageInfo>
{
    private readonly string _name;
    private readonly List<string> _flights = new();
    private readonly string _format = "{0,-20} {1,5}  {2, 3}";
    private IDisposable? _cancellation;

    public ArrivalsMonitor(string name)
    {
        ArgumentException.ThrowIfNullOrEmpty(name);
        _name = name;
    }

    public virtual void Subscribe(BaggageHandler provider) =>
        _cancellation = provider.Subscribe(this);

    public virtual void Unsubscribe()
    {
        _cancellation?.Dispose();
        _flights.Clear();
    }

    public virtual void OnCompleted() => _flights.Clear();

    // No implementation needed: Method is not called by the BaggageHandler class.
    public virtual void OnError(Exception e)
    {
        // No implementation.
    }

    // Update information.
    public virtual void OnNext(BaggageInfo info)
    {
        bool updated = false;

        // Flight has unloaded its baggage; remove from the monitor.
        if (info.Carousel is 0)
        {
            string flightNumber = string.Format("{0,5}", info.FlightNumber);
            for (int index = _flights.Count - 1; index >= 0; index--)
            {
                string flightInfo = _flights[index];
                if (flightInfo.Substring(21, 5).Equals(flightNumber))
                {
                    updated = true;
                    _flights.RemoveAt(index);
                }
```

```
            }
        }
        else
        {
            // Add flight if it doesn't exist in the collection.
            string flightInfo = string.Format(_format, info.From, info.FlightNumber,
info.Carousel);
            if (_flights.Contains(flightInfo) is false)
            {
                _flights.Add(flightInfo);
                updated = true;
            }
        }

        if (updated)
        {
            _flights.Sort();
            Console.WriteLine($"Arrivals information from {_name}");
            foreach (string flightInfo in _flights)
            {
                Console.WriteLine(flightInfo);
            }

            Console.WriteLine();
        }
    }
}
```

클래스에는 `ArrivalsMonitor` 및 `Subscribe` 메서드가 포함됩니다. `Subscribe` 메서드를 사용하면, 클래스는 IDisposable 호출에서 반환된 구현을 프라이빗 변수에 저장할 Subscribe 수 있습니다. 이 `Unsubscribe` 메서드를 사용하면 공급자의 Dispose 구현을 호출하여 클래스가 알림에서 구독을 취소할 수 있습니다. `ArrivalsMonitor`는 OnNext, OnError, 및 OnCompleted 메서드의 구현도 제공합니다. 구현에 OnNext 만 상당한 양의 코드가 포함됩니다. 이 방법은 도착 항공편의 출발지 공항과 수하물을 받을 수 있는 회전식 컨베이어에 대한 정보를 유지하는 개인 정렬된 일반 List<T> 객체와 함께 작동합니다. 클래스가 `BaggageHandler` 새 플라이트 도착을 보고하는 경우 메서드 구현은 OnNext 해당 플라이트 정보를 목록에 추가합니다. 클래스가 `BaggageHandler` 항공편의 수하물이 언로드되었다고 OnNext 보고하는 경우, 이 메서드는 해당 항공편을 목록에서 제거합니다. 변경될 때마다 목록이 정렬되어 콘솔에 표시됩니다.

다음 예제에서는 `BaggageHandler` 클래스와 `ArrivalsMonitor` 클래스의 두 인스턴스를 인스턴스화하는 애플리케이션 진입점을 포함하고, `BaggageHandler.BaggageStatus` 메서드를 사용하여 도착 항공편에 대한 정보를 추가하고 제거합니다. 각 경우에 관찰자는 업데이트를 받고 수하물 클레임 정보를 올바르게 표시합니다.

C#

```
using Observables.Example;

BaggageHandler provider = new();
```

```
ArrivalsMonitor observer1 = new("BaggageClaimMonitor1");
ArrivalsMonitor observer2 = new("SecurityExit");

provider.BaggageStatus(712, "Detroit", 3);
observer1.Subscribe(provider);

provider.BaggageStatus(712, "Kalamazoo", 3);
provider.BaggageStatus(400, "New York-Kennedy", 1);
provider.BaggageStatus(712, "Detroit", 3);
observer2.Subscribe(provider);

provider.BaggageStatus(511, "San Francisco", 2);
provider.BaggageStatus(712);
observer2.Unsubscribe();

provider.BaggageStatus(400);
provider.LastBaggageClaimed();

// Sample output:
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//   Kalamazoo              712    3
//
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//   Kalamazoo              712    3
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
```

```
//   Arrivals information from BaggageClaimMonitor1
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from SecurityExit
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from BaggageClaimMonitor1
//   San Francisco          511    2
```

관련 문서

테이블 확장

제목	설명
관찰자 디자인 패턴 모범 사례	관찰자 디자인 패턴을 구현하는 애플리케이션을 개발할 때 채택할 모범 사례를 설명합니다.
방법: 공급자 구현하기	온도 모니터링 애플리케이션에 대한 공급자의 단계별 구현을 제공합니다.
방법: 관찰자 구현	온도 모니터링 애플리케이션에 대한 관찰자의 단계별 구현을 제공합니다.

Last updated on 2025. 10. 20.

관찰자 디자인 패턴 모범 사례

2025. 06. 17.

.NET에서 관찰자 디자인 패턴은 인터페이스 집합으로 구현됩니다. System.IObservable<T> 인터페이스는 데이터 공급자를 나타내며, 관찰자가 알림에서 구독을 취소할 수 있도록 하는 IDisposable 구현을 제공할 책임이 있습니다. 인터페이스는 System.IObserver<T> 관찰자를 나타냅니다. 이 항목에서는 이러한 인터페이스를 사용하여 관찰자 디자인 패턴을 구현할 때 개발자가 따라야 하는 모범 사례를 설명합니다.

스레드 처리

일반적으로 공급자는 일부 컬렉션 개체가 나타내는 구독자 목록에 특정 관찰자를 추가하여 메서드를 구현 IObservable<T>.Subscribe 하고 구독자 목록에서 특정 관찰자를 제거하여 메서드를 구현합니다 IDisposable.Dispose . 관찰자는 언제든지 이러한 메서드를 호출할 수 있습니다. 또한 공급자/관찰자 계약은 콜백 메서드 이후 IObserver<T>.OnCompleted 구독 취소를 담당하는 사용자를 지정하지 않으므로 공급자와 관찰자는 모두 목록에서 동일한 멤버를 제거하려고 시도할 수 있습니다. 이러한 가능성 때문에, Subscribe 메서드와 Dispose 메서드는 모두 스레드로부터 안전해야 합니다. 일반적으로 동시 컬렉션 또는 잠금을 사용하는 것이 포함됩니다. 스레드로부터 안전하지 않은 구현은 그렇지 않다는 것을 명시적으로 문서화해야 합니다.

추가 보증은 공급자/관찰자 계약 위에 있는 계층에 지정해야 합니다. 구현자는 관찰자 계약에 대한 사용자 혼동을 피하기 위해 추가 요구 사항을 적용할 때 명확하게 설명해야 합니다.

예외 처리

데이터 공급자와 관찰자 간의 느슨한 결합으로 인해 관찰자 디자인 패턴의 예외는 정보 제공을 위한 것입니다. 이는 공급자와 관찰자가 관찰자 디자인 패턴에서 예외를 처리하는 방식에 영향을 줍니다.

공급자 - OnError 메서드 호출

이 OnError 메서드는 메서드와 마찬가지로 IObserver<T>.OnNext 관찰자에게 정보를 전달하기 위한 것입니다. 그러나 OnNext 이 메서드는 관찰자에게 현재 또는 업데이트된 데이터를 제공하도록 디자인된 반면 OnError , 메서드는 공급자가 유효한 데이터를 제공할 수 없음을 나타내도록 설계되었습니다.

공급자는 예외를 처리하고 메서드를 호출 OnError 할 때 다음 모범 사례를 따라야 합니다.

- 공급자는 특정 요구 사항이 있는 경우 자체 예외를 처리해야 합니다.

- 공급자는 관찰자가 특정 방식으로 예외를 처리할 것을 기대하거나 요구하지 않아야 합니다.
- 공급자는 업데이트를 제공하는 기능을 손상시키는 예외를 처리할 때 메서드를 호출 OnError 해야 합니다. 이러한 예외에 대한 정보는 관찰자에게 전달될 수 있습니다. 다른 경우에는 관찰자에게 예외를 알릴 필요가 없습니다.

공급자가 OnError 메서드 또는 IObserver<T>.OnCompleted 메서드를 호출하면 더 이상 알림이 없어야 하며, 공급자는 관찰자 구독을 해지할 수 있습니다. 그러나 관찰자는 알림을 받기 OnErrorIObserver<T>.OnCompleted 전과 후에 모두 포함하여 언제든지 구독을 취소할 수 있습니다. 관찰자 디자인 패턴은 공급자 또는 관찰자가 구독 취소를 담당하는지 여부를 결정하지 않습니다. 따라서 둘 다 구독 취소를 시도할 수 있습니다. 일반적으로 관찰자가 구독을 취소하면 구독자 컬렉션에서 제거됩니다. 단일 스레드 애플리케이션 IDisposable.Dispose 에서 구현은 개체 참조가 유효하고 개체가 제거를 시도하기 전에 구독자 컬렉션의 멤버인지 확인해야 합니다. 다중 스레드 애플리케이션에서는 System.Collections.Concurrent.BlockingCollection<T>와 같은 스레드로부터 안전한 컬렉션 개체를 사용해야 합니다.

관찰자 - OnError 메서드 구현

관찰자가 공급자로부터 오류 알림을 받으면 관찰자는 예외를 정보로 처리해야 하며 특정 작업을 수행할 필요가 없습니다.

관찰자는 공급자의 메서드 호출에 응답할 OnError 때 다음 모범 사례를 따라야 합니다.

- 관찰자는 인터페이스 구현(예: OnNext 또는 OnError.)에서 예외를 throw해서는 안 됩니다. 그러나 관찰자가 예외를 throw하는 경우 이러한 예외가 처리되지 않을 것으로 예상해야 합니다.
- 호출 스택을 보존하려면, 자신의 Exception 메서드에 전달된 OnError 객체를 throw하려는 옵저버는 예외를 던지기 전에 예외를 래핑해야 합니다. 이 목적을 위해 표준 예외 개체를 사용해야 합니다.

추가 모범 사례

메서드에서 IObservable<T>.Subscribe 등록 취소를 시도하면 null 참조가 발생할 수 있습니다. 따라서 이 방법을 피하는 것이 좋습니다.

여러 공급자에 관찰자를 연결할 수 있지만, 권장되는 패턴은 IObserver<T> 인스턴스를 단 하나의 IObservable<T> 인스턴스에만 연결하는 것입니다.

참고하십시오

- 관찰자 디자인 패턴
- 방법: 관찰자 구현
- 방법: 공급자 구현하기

방법: 공급자 구현

아티클 • 2023. 04. 07.

관찰자 디자인 패턴은 데이터를 모니터링하고 알림을 보내는 공급자와 해당 공급자로부터 알림(콜백)을 받는 하나 이상의 관찰자 간에 구분이 필요합니다. 이 항목에서는 공급자를 만드는 방법을 설명합니다. 관련 항목인 방법: 관찰자 구현에서는 관찰자를 만드는 방법을 설명합니다.

공급자를 만들려면

1. 공급자가 관찰자에게 보내야 하는 데이터를 정의합니다. 공급자와 이 공급자가 관찰자로 보내는 데이터가 단일 형식일 수 있지만, 일반적으로 서로 다른 형식으로 표시됩니다. 예를 들어 온도 모니터링 애플리케이션에서 `Temperature` 구조체는 공급자(다음 단계에 정의된 `TemperatureMonitor` 클래스로 표시됨)가 모니터링하고 관찰자가 구독하는 데이터를 정의합니다.

   C#

   ```
   using System;

   public struct Temperature
   {
      private decimal temp;
      private DateTime tempDate;

      public Temperature(decimal temperature, DateTime dateAndTime)
      {
         this.temp = temperature;
         this.tempDate = dateAndTime;
      }

      public decimal Degrees
      { get { return this.temp; } }

      public DateTime Date
      { get { return this.tempDate; } }
   }
   ```

2. System.IObservable<T> 인터페이스를 구현하는 형식인 데이터 공급자를 정의합니다. 공급자의 제네릭 형식 인수는 공급자가 관찰자로 보내는 형식입니다. 다음 예제에서는 `Temperature`의 제네릭 형식 인수를 사용하여 구성된 System.IObservable<T> 구현인 `TemperatureMonitor` 클래스를 정의합니다.

   C#

```
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
```

3. 해당하는 경우 각 관찰자에게 알릴 수 있도록 공급자가 관찰자에 대한 참조를 저장할 방식을 결정합니다. 일반적으로 제네릭 List<T> 개체와 같은 컬렉션 개체가 이 용도로 사용됩니다. 다음 예제에서는 `TemperatureMonitor` 클래스 생성자에서 인스턴스화되는 private List<T> 개체를 정의합니다.

C#

```
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
   List<IObserver<Temperature>> observers;

   public TemperatureMonitor()
   {
      observers = new List<IObserver<Temperature>>();
   }
```

4. 구독자가 언제든지 알림 수신을 중지할 수 있도록 공급자가 해당 구독자에게 반환할 수 있는 IDisposable 구현을 정의합니다. 다음 예제에서는 클래스가 인스턴스화될 때 구독자 컬렉션 및 구독자에 대한 참조를 전달받는 중첩 `Unsubscriber` 클래스를 정의합니다. 이 코드를 통해 구독자가 개체의 IDisposable.Dispose 구현을 호출하여 구독자 컬렉션에서 자신을 제거할 수 있습니다.

C#

```
private class Unsubscriber : IDisposable
{
   private List<IObserver<Temperature>> _observers;
   private IObserver<Temperature> _observer;

   public Unsubscriber(List<IObserver<Temperature>> observers,
IObserver<Temperature> observer)
   {
      this._observers = observers;
      this._observer = observer;
   }

   public void Dispose()
   {
      if (! (_observer == null)) _observers.Remove(_observer);
```

```
   }
}
```

5. IObservable<T>.Subscribe 메서드를 구현합니다. 이 메서드는 System.IObserver<T> 인터페이스에 대한 참조를 전달받으며, 3단계에서 해당 용도로 디자인한 개체에 저장되어야 합니다. 그런 다음, 이 메서드는 4단계에서 개발한 IDisposable 구현을 반환해야 합니다. 다음 예제는 `TemperatureMonitor` 클래스의 Subscribe 메서드 구현을 보여줍니다.

C#

```
public IDisposable Subscribe(IObserver<Temperature> observer)
{
   if (! observers.Contains(observer))
      observers.Add(observer);

   return new Unsubscriber(observers, observer);
}
```

6. 해당 IObserver<T>.OnNext, IObserver<T>.OnError 및 IObserver<T>.OnCompleted 구현을 호출하여 적절하게 관찰자에게 알립니다. 일부 경우에 오류가 발생하면 공급자가 OnError 메서드를 호출하지 못할 수 있습니다. 예를 들어 다음 `GetTemperature` 메서드는 5초마다 온도 데이터를 읽는 모니터를 시뮬레이트하고 온도가 이전 값보다 .1도 이상 변경된 경우 관찰자에게 알립니다. 디바이스에서 온도를 보고하지 않으면(즉, 값이 null인 경우) 공급자는 전송이 완료되었음을 관찰자에게 알립니다. 각 관찰자의 OnCompleted 메서드를 호출할 뿐만 아니라 `GetTemperature` 메서드는 List<T> 컬렉션을 지웁니다. 이 경우 공급자는 해당 관찰자의 OnError 메서드를 호출하지 않습니다.

C#

```
public void GetTemperature()
{
   // Create an array of sample data to mimic a temperature device.
   Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m,
15.2m, 15.25m, 15.2m,
                                15.4m, 15.45m, null };
   // Store the previous temperature, so notification is only sent
after at least .1 change.
   Nullable<Decimal> previous = null;
   bool start = true;

   foreach (var temp in temps) {
      System.Threading.Thread.Sleep(2500);
      if (temp.HasValue) {
         if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m ))
{
```

```
            Temperature tempData = new Temperature(temp.Value,
 DateTime.Now);
            foreach (var observer in observers)
               observer.OnNext(tempData);
            previous = temp;
            if (start) start = false;
         }
      }
      else {
         foreach (var observer in observers.ToArray())
            if (observer != null) observer.OnCompleted();

         observers.Clear();
         break;
      }
   }
}
```

예제

다음 예제에는 온도 모니터링 애플리케이션에 대한 IObservable<T> 구현을 정의하기 위한 전체 소스 코드가 포함되어 있습니다. 여기에는 관찰자로 전송된 데이터인 `Temperature` 구조체 및 IObservable<T> 구현인 `TemperatureMonitor` 클래스가 포함됩니다.

C#

```
using System.Threading;
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
   List<IObserver<Temperature>> observers;

   public TemperatureMonitor()
   {
      observers = new List<IObserver<Temperature>>();
   }

   private class Unsubscriber : IDisposable
   {
      private List<IObserver<Temperature>> _observers;
      private IObserver<Temperature> _observer;

      public Unsubscriber(List<IObserver<Temperature>> observers,
IObserver<Temperature> observer)
      {
         this._observers = observers;
         this._observer = observer;
```

```
      }

      public void Dispose()
      {
         if (! (_observer == null)) _observers.Remove(_observer);
      }
   }

   public IDisposable Subscribe(IObserver<Temperature> observer)
   {
      if (! observers.Contains(observer))
         observers.Add(observer);

      return new Unsubscriber(observers, observer);
   }

   public void GetTemperature()
   {
      // Create an array of sample data to mimic a temperature device.
      Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m,
15.2m, 15.25m, 15.2m,
                                   15.4m, 15.45m, null };
      // Store the previous temperature, so notification is only sent after
at least .1 change.
      Nullable<Decimal> previous = null;
      bool start = true;

      foreach (var temp in temps) {
         System.Threading.Thread.Sleep(2500);
         if (temp.HasValue) {
            if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m )) {
               Temperature tempData = new Temperature(temp.Value,
DateTime.Now);
               foreach (var observer in observers)
                  observer.OnNext(tempData);
               previous = temp;
               if (start) start = false;
            }
         }
         else {
            foreach (var observer in observers.ToArray())
               if (observer != null) observer.OnCompleted();

            observers.Clear();
            break;
         }
      }
   }
}
```

참고 항목

- IObservable<T>
- 관찰자 디자인 패턴
- 방법: 관찰자 구현
- 관찰자 디자인 패턴 유용한 정보

방법: 관찰자 구현

아티클 • 2025. 04. 10.

관찰자 디자인 패턴에는 알림을 등록하는 관찰자와 데이터를 모니터링하고 하나 이상의 관찰자에게 알림을 보내는 공급자 간의 구분이 필요합니다. 이 항목에서는 관찰자를 만드는 방법에 대해 설명합니다. 관련 항목인 방법: 공급자구현에서는 공급자를 만드는 방법에 대해 설명합니다.

관찰자를 만들려면

1. System.IObserver<T> 인터페이스를 구현하는 형식인 관찰자를 정의합니다. 예를 들어 다음 코드는 `TemperatureReporter`제네릭 형식 인수를 사용하여 생성된 System.IObserver<T> 구현인 `Temperature` 형식을 정의합니다.

   C#

   ```
   public class TemperatureReporter : IObserver<Temperature>
   ```

2. 공급자가 IObserver<T>.OnCompleted 구현을 호출하기 전에 관찰자가 알림 수신을 중지할 수 있는 경우 공급자의 IDisposable 메서드에서 반환된 IObservable<T>.Subscribe 구현을 보유할 프라이빗 변수를 정의합니다. 또한 공급자의 Subscribe 메서드를 호출하고 반환된 IDisposable 개체를 저장하는 구독 메서드를 정의해야 합니다. 예를 들어 다음 코드는 `unsubscriber` 프라이빗 변수를 정의하고 공급자의 `Subscribe` 메서드를 호출하고 반환된 개체를 Subscribe 변수에 할당하는 `unsubscriber` 메서드를 정의합니다.

   C#

   ```
   public class TemperatureReporter : IObserver<Temperature>
   {
      private IDisposable unsubscriber;
      private bool first = true;
      private Temperature last;

      public virtual void Subscribe(IObservable<Temperature> provider)
      {
         unsubscriber = provider.Subscribe(this);
      }
   ```

3. 이 기능이 필요한 경우 공급자가 IObserver<T>.OnCompleted 구현을 호출하기 전에 관찰자가 알림 수신을 중지할 수 있도록 하는 메서드를 정의합니다. 다음 예제에서는 `Unsubscribe` 메서드를 정의합니다.

   C#

```
public virtual void Unsubscribe()
{
   unsubscriber.Dispose();
}
```

4. IObserver<T> 인터페이스에서 정의한 세 가지 메서드인 IObserver<T>.OnNext, IObserver<T>.OnError및 IObserver<T>.OnCompleted구현을 제공합니다. 공급자 및 애플리케이션의 요구 사항에 따라 OnError 및 OnCompleted 메서드는 스텁 구현일 수 있습니다. OnError 메서드는 전달된 Exception 개체를 예외로 처리해서는 안 되며 OnCompleted 메서드는 공급자의 IDisposable.Dispose 구현을 자유롭게 호출할 수 있습니다. 다음 예제에서는 IObserver<T> 클래스의 `TemperatureReporter` 구현을 보여줍니다.

C#

```
public virtual void OnCompleted()
{
   Console.WriteLine("Additional temperature data will not be transmitted.");
}

public virtual void OnError(Exception error)
{
   // Do nothing.
}

public virtual void OnNext(Temperature value)
{
   Console.WriteLine($"The temperature is {value.Degrees}°C at
{value.Date:g}");
   if (first)
   {
      last = value;
      first = false;
   }
   else
   {
      Console.WriteLine($"   Change: {value.Degrees - last.Degrees}° in
{value.Date.ToUniversalTime() - last.Date.ToUniversalTime():g}");
   }
}
```

예시

다음 예제에서는 온도 모니터링 애플리케이션에 대 한 `TemperatureReporter` 구현을 제공 하는 IObserver<T> 클래스에 대 한 전체 소스 코드를 포함 합니다.

C#

```
public class TemperatureReporter : IObserver<Temperature>
{
   private IDisposable unsubscriber;
   private bool first = true;
   private Temperature last;

   public virtual void Subscribe(IObservable<Temperature> provider)
   {
      unsubscriber = provider.Subscribe(this);
   }

   public virtual void Unsubscribe()
   {
      unsubscriber.Dispose();
   }

   public virtual void OnCompleted()
   {
      Console.WriteLine("Additional temperature data will not be transmitted.");
   }

   public virtual void OnError(Exception error)
   {
      // Do nothing.
   }

   public virtual void OnNext(Temperature value)
   {
      Console.WriteLine($"The temperature is {value.Degrees}°C at
{value.Date:g}");
      if (first)
      {
         last = value;
         first = false;
      }
      else
      {
         Console.WriteLine($"   Change: {value.Degrees - last.Degrees}° in
{value.Date.ToUniversalTime() - last.Date.ToUniversalTime():g}");
      }
   }
}
```

참고하십시오

- IObserver<T>
- 관찰자 디자인 패턴
- 방법: 공급자 구현하기
- 관찰자 디자인 패턴 모범 사례

.NET에서 예외 처리 및 예외 던지기

2025. 06. 17.

애플리케이션은 일관된 방식으로 실행 중에 발생하는 오류를 처리할 수 있어야 합니다. .NET은 애플리케이션에 오류를 균일한 방식으로 알리는 모델을 제공합니다. .NET 작업은 예외를 throw 하여 오류를 나타냅니다.

예외

예외는 실행 중인 프로그램에서 발생하는 오류 조건 또는 예기치 않은 동작입니다. 코드 사용 중 또는 직접 작성한 코드(예: 공유 라이브러리)의 오류, 사용할 수 없는 운영 체제 리소스, 런타임에서 발생하는 예상치 못한 상황(예: 확인할 수 없는 코드) 등으로 인해 예외가 발생할 수 있습니다. 애플리케이션은 이러한 조건 중 일부에서 복구할 수 있지만 다른 조건에서는 복구할 수 없습니다. 대부분의 애플리케이션 예외에서 복구할 수 있지만 대부분의 런타임 예외에서 복구할 수는 없습니다.

.NET에서 예외는 클래스 System.Exception를 상속하는 개체입니다. 문제가 발생한 코드 영역에서 예외가 throw됩니다. 예외는 애플리케이션이 처리하거나 프로그램이 종료될 때까지 스택 위로 전달됩니다.

예외 및 기존 오류 처리 메서드

일반적으로 언어의 오류 처리 모델은 오류를 검색하고 처리기를 찾는 언어의 고유한 방법 또는 운영 체제에서 제공하는 오류 처리 메커니즘에 의존했습니다. .NET에서 예외 처리를 구현하는 방법은 다음과 같은 이점을 제공합니다.

- 예외 발생 및 처리는 .NET 프로그래밍 언어에서 동일하게 작동합니다.
- 예외를 처리하기 위한 특정 언어 구문은 필요하지 않지만 각 언어에서 고유한 구문을 정의할 수 있습니다.
- 예외는 프로세스 및 기기 경계를 넘어 던져질 수 있습니다.
- 프로그램 안정성을 높이기 위해 예외 처리 코드를 애플리케이션에 추가할 수 있습니다.

예외는 반환 코드와 같은 다른 오류 알림 방법보다 이점을 제공합니다. 예외가 발생하고 이를 처리하지 않으면 런타임이 애플리케이션을 종료하므로 오류는 그냥 지나치지 않습니다. 잘못된 값은 오류 반환 코드를 확인하지 못하는 코드의 결과로 시스템을 통해 계속 전파되지 않습니다.

일반적인 예외

다음 표에는 몇 가지 일반적인 예외와 원인의 예가 나와 있습니다.

테이블 확장

예외 유형	설명	예시
Exception	모든 예외에 대한 기본 클래스입니다.	없음(이 예외의 파생 클래스 사용).
IndexOutOfRangeException	배열이 잘못 인덱싱된 경우에만 런타임에서 throw됩니다.	유효한 범위를 벗어난 배열 인덱싱: `arr[arr.Length+1]`
NullReferenceException	null 개체를 참조하는 경우에만 런타임에 의해 throw됩니다.	`object o = null;` `o.ToString();`
InvalidOperationException	잘못된 상태일 때 메서드에 의해 throw됩니다.	기본 컬렉션에서 항목을 제거한 후 호출 `Enumerator.MoveNext()` 합니다.
ArgumentException	모든 인수 예외에 대한 기본 클래스입니다.	없음(이 예외의 파생 클래스 사용).
ArgumentNullException	인수를 null로 허용하지 않는 메서드에 의해 throw됩니다.	`String s = null;` `"Calculate".IndexOf(s);`
ArgumentOutOfRangeException	인수가 특정 범위 안에 있는지를 확인하는 메서드에 의해 throw됩니다.	`String s = "string";` `s.Substring(s.Length+1);`

참고하십시오

- Exception 클래스 및 속성
- 방법: Try-Catch 블록을 사용하여 예외를 잡는
- 방법: 특정 예외를 Catch 블록에서 사용하는 방법
- 방법: 명시적으로 예외를 던지기
- 방법: User-Defined 예외 만들기
- User-Filtered 예외 처리기 사용
- 방법: Finally 블록 사용
- COM Interop 예외 처리
- 예외에 대한 모범 사례
- 모든 개발자가 런타임의 예외에 대해 알아야 할 사항

예외 클래스 및 속성

2025. 06. 17.

Exception 클래스는 예외가 상속되는 기본 클래스입니다. 예를 들어 InvalidCastException 클래스 계층 구조는 다음과 같습니다.

Object
  Exception
    SystemException
      InvalidCastException

클래스에는 Exception 예외를 더 쉽게 이해할 수 있도록 하는 다음과 같은 속성이 있습니다.

테이블 확장

속성 이름	설명
Data	IDictionary 키-값 쌍으로 임의의 데이터를 저장하는 객체입니다.
HelpLink	예외의 원인에 대한 광범위한 정보를 제공하는 도움말 파일에 URL(또는 URN)을 보관할 수 있습니다.
InnerException	이 속성은 예외를 처리하는 동안 일련의 예외를 만들고 보존하는 데 사용할 수 있습니다. 이를 사용하여 이전에 catch된 예외를 포함하는 새 예외를 만들 수 있습니다. 원래 예외는 속성의 두 번째 예외에 InnerException 의해 캡처될 수 있으므로 두 번째 예외를 처리하는 코드가 추가 정보를 검사할 수 있습니다. 예를 들어 형식이 잘못 지정된 인수를 받는 메서드가 있다고 가정해 보겠습니다. 코드는 인수를 읽으려고 시도하지만, 예외가 던져집니다. 메서드는 예외를 catch하고 FormatException을(를) 던집니다. 예외를 발생시키는 이유를 호출자가 더 잘 파악할 수 있도록, 메서드가 보조 루틴에서 발생한 예외를 잡아내고 나서 발생한 오류를 더 잘 나타내는 예외를 발생시키는 것이 바람직할 때가 있습니다. 내부 예외 참조를 원래 예외로 설정할 수 있는 더 의미 있는 새 예외를 만들 수 있습니다. 그러면 더 의미 있는 예외가 호출자에게 던져질 수 있습니다. 이 기능을 사용하면 먼저 throw된 예외로 끝나는 일련의 연결된 예외를 만들 수 있습니다.
Message	예외의 원인에 대한 세부 정보를 제공합니다.
Source	오류를 발생시키는 애플리케이션 또는 개체의 이름을 가져오거나 설정합니다.
StackTrace	오류가 발생한 위치를 확인하는 데 사용할 수 있는 스택 추적을 포함합니다. 디버깅 정보를 사용할 수 있는 경우 스택 추적에는 원본 파일 이름 및 프로그램 줄 번호가 포함됩니다.

대부분의 클래스는 Exception을(를) 상속하지만 추가 멤버를 구현하거나 추가 기능을 제공하지 않습니다. 그들은 단순히 Exception을(를) 상속할 뿐입니다. 따라서 예외에 대한 가장 중요한 정보는 예외 클래스의 계층 구조, 예외 이름 및 예외에 포함된 정보를 찾을 수 있습니다.

Exception에서 파생된 개체만 throw하고 catch하는 것이 좋지만, Object 클래스에서 파생된 개체는 예외로 throw할 수 있습니다. 모든 언어가 Exception에서 파생되지 않은 오브젝트를 throw하고 catch하는 것을 지원하지는 않습니다.

참고하십시오

- 예외

try/catch 블록을 사용하여 예외를 catch하는 방법

아티클 • 2025. 05. 09.

`try` 블록에는 예외를 발생시키거나 throw할 가능성이 있는 코드 문을 배치하고, `catch` 블록 아래에 `try` 블록 하나 이상을 두어 예외를 처리하는 문을 배치합니다. 각 `catch` 블록은 예외 형식을 포함하며 해당 예외 유형을 처리하는 데 필요한 추가 문을 포함할 수 있습니다.

다음 예제 StreamReader 에서는 *data.txt* 파일을 열고 파일에서 줄을 검색합니다. 코드는 세 가지 예외 중 어느 것을 발생시킬 수 있으므로 해당 `try` 블록에 배치됩니다. 세 개의 `catch` 블록은 예외를 포착하고 콘솔에 결과를 표시하여 처리합니다.

C#

```
using System;
using System.IO;

public class ProcessFile
{
    public static void Main()
    {
        try
        {
            using (StreamReader sr = File.OpenText("data.txt"))
            {
                Console.WriteLine($"The first line of this file is
{sr.ReadLine()}");
            }
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine($"The file was not found: '{e}'");
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine($"The directory was not found: '{e}'");
        }
        catch (IOException e)
        {
            Console.WriteLine($"The file could not be opened: '{e}'");
        }
    }
}
```

CLR(공용 언어 런타임)은 `catch` 블록으로 처리되지 않는 예외를 처리합니다. CLR에서 예외를 catch하는 경우 CLR 구성에 따라 다음 결과 중 하나가 발생할 수 있습니다.

- **디버그** 대화 상자가 나타납니다.

- 프로그램이 실행을 중지하고 예외 정보가 있는 대화 상자가 나타납니다.
- 오류가 표준 오류 출력 스트림에 출력됩니다.

> **참고**
>
> 대부분의 코드는 예외를 throw할 수 있으며, CLR 자체에 의해 언제든지 throw될 수 있는 예외(예: OutOfMemoryException일부 예외)가 있습니다. 애플리케이션에서 이러한 예외를 처리할 필요는 없지만 다른 사용자가 사용할 라이브러리를 작성할 때 발생할 수 있습니다. 블록에서 `try` 코드를 설정하는 시기에 대한 제안 사항은 **예외에 대한 모범 사례를 참조하세요**.

참고하십시오

- 예외
- .NET에서 I/O 오류 처리

catch 블록에서 특정 예외를 사용하는 방법

아티클 • 2025. 05. 01.

일반적으로 기본 `catch` 문을 사용하기보다는 특정 유형의 예외를 잡는 것이 좋은 프로그래밍 습관입니다.

예외가 발생하면 스택을 따라 전달되며 각 catch 블록은 그것을 처리할 기회를 갖습니다. catch 문의 순서가 중요합니다. 특정 예외를 대상으로 하는 catch 블록을 일반적인 예외 catch 블록 앞에 배치하지 않으면 컴파일러가 오류를 발생시킬 수 있습니다. 적절한 catch 블록은 예외의 형식을 catch 블록에 지정된 예외의 이름과 일치시켜 결정됩니다. 특정 catch 블록이 없으면, 일반 catch 블록이 존재할 경우 그 블록에 의해 예외가 처리됩니다.

다음 코드 예제에서는 `try`/`catch` 블록을 사용하여 InvalidCastException을(를) 포착합니다. 샘플은 직원 수준(`Emlevel`)이라는 단일 속성을 가진 `Employee` 클래스를 만듭니다. 메서드는 `PromoteEmployee`개체를 사용하고 직원 수준을 증가합니다. InvalidCastException 인스턴스가 DateTime 메서드에 전달될 때 `PromoteEmployee` 발생합니다.

C#

```
using System;

public class Employee
{
    //Create employee level property.
    public int Emlevel
    {
        get
        {
            return(emlevel);
        }
        set
        {
            emlevel = value;
        }
    }

    private int emlevel = 0;
}

public class Ex13
{
    public static void PromoteEmployee(Object emp)
    {
        // Cast object to Employee.
        var e = (Employee) emp;
        // Increment employee level.
        e.Emlevel = e.Emlevel + 1;
    }
```

```
    static void Main()
    {
        try
        {
            Object o = new Employee();
            DateTime newYears = new DateTime(2001, 1, 1);
            // Promote the new employee.
            PromoteEmployee(o);
            // Promote DateTime; results in InvalidCastException as newYears is
not an employee instance.
            PromoteEmployee(newYears);
        }
        catch (InvalidCastException e)
        {
            Console.WriteLine("Error passing data to PromoteEmployee method. " +
e.Message);
        }
    }
}
```

참고하십시오

- 예외

명시적으로 예외를 throw하는 방법

아티클 • 2024. 03. 14.

C# throw 또는 Visual Basic Throw 문을 사용하여 명시적으로 예외를 throw할 수 있습니다. `throw` 문을 사용하여 catch된 예외를 다시 throw할 수도 있습니다. 디버깅할 때 더 많은 정보를 제공하기 위해 다시 throw되는 예외에 정보를 추가하는 것이 좋은 코딩 습관입니다.

다음 코드 예제에서는 `try`/`catch` 블록을 사용하여 가능한 FileNotFoundException을 catch합니다. `try` 블록 뒤에는 FileNotFoundException을 catch하고 데이터 파일을 찾을 수 없는 경우 콘솔에 메시지를 쓰는 `catch` 블록이 있습니다. 다음 문은 새 FileNotFoundException을 throw하고 예외에 텍스트 정보를 추가하는 `throw` 문입니다.

C#

```
var fs = default(FileStream);
try
{
    // Opens a text tile.
    fs = new FileStream(@"C:\temp\data.txt", FileMode.Open);
    var sr = new StreamReader(fs);

    // A value is read from the file and output to the console.
    string? line = sr.ReadLine();
    Console.WriteLine(line);
}
catch (FileNotFoundException e)
{
    Console.WriteLine($"[Data File Missing] {e}");
    throw new FileNotFoundException(@"[data.txt not in c:\temp directory]",
e);
}
finally
{
    if (fs != null)
        fs.Close();
}
```

참고 항목

- 예외

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

사용자 정의 예외를 만드는 방법

아티클 • 2023. 05. 09.

.NET에서는 Exception base 클래스에서 최종적으로 파생되는 예외 클래스의 계층 구조를 제공합니다. 그러나 사용자 요구를 충족하는 미리 정의된 예외가 없는 경우 Exception 클래스에서 파생하여 사용자 고유의 예외 클래스를 만들 수 있습니다.

사용자 고유의 예외를 만드는 경우 다음 예제와 같이 사용자 정의 예외의 클래스 이름을 "Exception" 단어로 끝내고 세 가지 공통 생성자를 구현합니다. 예제에서는 `EmployeeListNotFoundException`이라는 새 예외 클래스를 정의합니다. 이 클래스는 Exception base 클래스에서 파생되며 세 가지 생성자를 포함합니다.

C#

```
using System;

public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

> ⓘ **참고**
>
> 원격을 사용하는 경우 사용자 정의 예외에 대한 메타데이터를 서버(호출 수신자) 및 클라이언트(프록시 개체 또는 호출자)에서 사용할 수 있는지 확인해야 합니다. 자세한 내용은 **예외에 대한 모범 사례**를 참조하세요.

참조

- 예외

방법: 지역화된 예외 메시지를 사용하여 사용자 정의 예외 생성

아티클 • 2024. 07. 31.

이 문서에서는 위성 어셈블리를 사용하여 지역화된 예외 메시지로 기본 Exception 클래스에서 상속되는 사용자 정의 예외를 만드는 방법을 설명합니다.

사용자 지정 예외 만들기

.NET에는 사용할 수 있는 다양한 예외가 포함되어 있습니다. 그러나 사용자의 요구를 충족하는 항목이 없는 경우에는 사용자 지정 예외를 직접 만들 수 있습니다.

`StudentName` 속성을 포함하는 `StudentNotFoundException`을 만들려고 한다고 가정해 보겠습니다. 사용자 지정 예외를 만들려면 다음 단계를 수행합니다.

1. Exception에서 상속된 클래스를 만듭니다. 클래스 이름은 "Exception"으로 끝나야 합니다.

   C#

   ```
   public class StudentNotFoundException : Exception { }
   ```

2. 기본 생성자를 추가합니다.

   C#

   ```
   public class StudentNotFoundException : Exception
   {
       public StudentNotFoundException() { }

       public StudentNotFoundException(string message)
           : base(message) { }

       public StudentNotFoundException(string message, Exception inner)
           : base(message, inner) { }
   }
   ```

3. 추가 속성 및 생성자를 정의합니다.

   C#

   ```
   public class StudentNotFoundException : Exception
   {
   ```

```
    public string StudentName { get; }

    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }

    public StudentNotFoundException(string message, string studentName)
        : this(message)
    {
        StudentName = studentName;
    }
}
```

지역화된 예외 메시지 만들기

사용자 지정 예외를 만들고 다음과 같은 코드를 사용하여 어디에서든 throw할 수 있습니다.

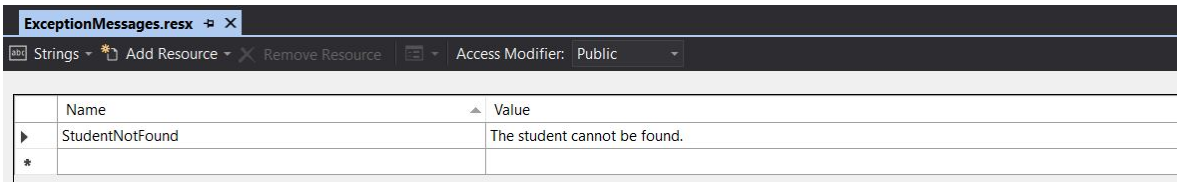
C#

```
throw new StudentNotFoundException("The student cannot be found.", "John");
```

이전 줄의 문제는 `"The student cannot be found."`이 상수 문자열에 불과하다는 것입니다. 지역화된 애플리케이션에서는 사용자 문화권에 따라 다른 메시지를 사용할 수 있습니다. 위성 어셈블리는 이 작업을 수행하는 좋은 방법입니다. 위성 어셈블리는 특정 언어에 대한 리소스를 포함하는 DLL입니다. 런타임에 특정 리소스를 요청하면 CLR은 사용자 문화권에 따라 해당 리소스를 찾습니다. 해당 문화권에 대한 위성 어셈블리를 찾을 수 없는 경우에는 기본 문화권의 리소스가 사용됩니다.

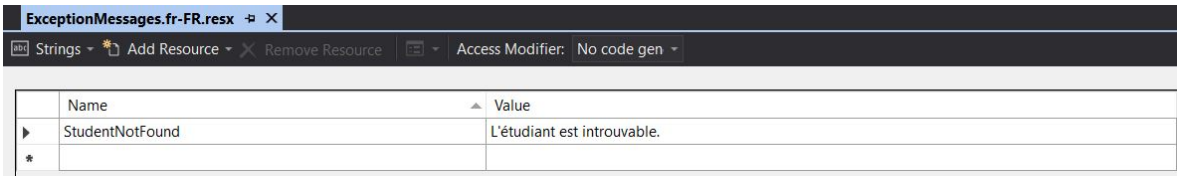
지역화된 예외 메시지를 만들려면

1. 리소스 파일을 저장한 *Resources*라는 새 폴더를 만듭니다.
2. 새 리소스 파일을 추가합니다. Visual Studio에서 이렇게 하려면 **솔루션 탐색기**에서 폴더를 마우스 오른쪽 단추로 클릭하고 **추가**>**새 항목**>**리소스 파일**을 선택합니다. 파일 이름을 *ExceptionMessages.resx*입니다. 이 파일은 기본 리소스 파일입니다.
3. 다음 그림에 표시된 것처럼 예외 메시지에 대한 이름/값 쌍을 추가합니다.



4. 프랑스어의 새 리소스 파일을 추가합니다. 이름을 *ExceptionMessages.fr-FR.resx*로 지정합니다.

5. 예외 메시지에 대한 이름/값 쌍을 다시 추가하지만 이번에는 다음과 같은 프랑스 값을 사용합니다.



6. 프로젝트를 빌드하면 빌드 출력 폴더에는 위성 어셈블리인 *.dll* 파일을 포함하는 *fr-FR* 폴더가 있어야 합니다.

7. 다음과 같은 코드를 사용하여 예외를 throw합니다.

C#

```
var resourceManager = new
ResourceManager("FULLY_QUALIFIED_NAME_OF_RESOURCE_FILE",
Assembly.GetExecutingAssembly());
throw new
StudentNotFoundException(resourceManager.GetString("StudentNotFound"),
"John");
```

> **참고**
>
> 프로젝트 이름이 `TestProject`이고 리소스 파일 *ExceptionMessages.resx*가 프로젝트의 *Resources* 폴더에 있는 경우 리소스 파일의 정규화된 이름은 `TestProject.Resources.ExceptionMessages`입니다.

참고 항목

- 사용자 정의 예외를 만드는 방법
- 위성 어셈블리 만들기
- base(C# 참조)
- this(C# 참조)

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

finally 블록을 사용하는 방법

아티클 • 2025. 05. 01.

예외가 발생하면 실행이 중지되고 적절한 예외 처리기에 제어가 제공됩니다. 이는 실행될 것으로 예상되는 코드 줄이 무시되는 경우가 많습니다. 파일 닫기와 같은 일부 리소스 정리는 예외가 throw된 경우에도 수행해야 합니다. 이를 위해 블록을 `finally` 사용할 수 있습니다. `finally` 예외가 throw되는지 여부에 관계없이 블록은 항상 실행됩니다.

다음 코드 예제에서는 `try`/`catch` 블록을 사용하여 ArgumentOutOfRangeException을/를 잡습니다. 이 메서드는 `Main` 두 개의 배열을 만들고 한 배열을 다른 배열에 복사하려고 시도합니다. `length`가 -1로 지정됨에 따라 ArgumentOutOfRangeException이 생성되고, 오류가 콘솔에 기록됩니다. 블록은 `finally` 복사 작업의 결과에 관계없이 실행됩니다.

C#

```
class ArgumentOutOfRangeExample
{
    public static void Main()
    {
        int[] array1 = {0, 0};
        int[] array2 = {0, 0};

        try
        {
            Array.Copy(array1, array2, -1);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Error: {e}");
            throw;
        }
        finally
        {
            Console.WriteLine("This statement is always executed.");
        }
    }
}
```

참고하십시오

- 예외

사용자 필터 예외 처리기 사용

아티클 • 2024. 03. 13.

사용자 필터 예외 처리기는 예외에 대해 정의한 요구 사항을 기반으로 하여 예외를 catch 하고 처리합니다. 이러한 처리기는 `when` 키워드(Visual Basic의 경우 `Catch` 및 `When`)와 함께 `catch` 문을 사용합니다.

이 기법은 특정 예외 개체가 여러 오류에 해당하는 경우에 유용합니다. 이 경우 개체는 일반적으로 오류와 관련된 특정 오류 코드를 포함하는 속성을 갖습니다. 식에서 해당 오류 코드 속성을 사용하여 `catch` 절에서 처리하려는 특정 오류만 선택할 수 있습니다.

다음 예에서는 `catch`/`when` 문을 보여 줍니다.

C#

```
try
{
    //Try statements.
}
catch (Exception ex) when (ex.Message.Contains("404"))
{
    //Catch statements.
}
```

사용자 필터 절의 식에는 어떤 제한도 적용되지 않습니다. 사용자 필터 식을 실행하는 중에 예외가 발생하면 예외는 무시되고 필터 식이 false로 계산된 것으로 간주됩니다. 이 경우 공용 언어 런타임은 현재 예외에 대한 처리기를 계속 검색합니다.

특정 예외 조항과 사용자 필터링 조항을 결합합니다.

`catch` 문은 특정 예외와 사용자 필터 절을 모두 포함할 수 있습니다. 런타임은 특정 예외를 먼저 테스트합니다. 특정 예외가 성공하면 런타임은 사용자 필터를 실행합니다. 일반 필터에는 클래스 필터에 선언된 변수에 대한 참조가 포함될 수 있습니다. 두 필터 절의 순서는 바꿀 수 없습니다.

다음 예에서는 **catch** 문의 특정 예외와 **when** 키워드를 사용하는 사용자 필터링 절을 보여 줍니다.

C#

```
try
{
```

```
    //Try statements.
}
catch (System.Net.Http.HttpRequestException ex) when
(ex.Message.Contains("404"))
{
    //Catch statements.
}
```

참고 항목

- 예외

COM Interop 예외 처리

관리 코드와 관리되지 않는 코드는 함께 작동하여 예외를 처리할 수 있습니다. 메서드가 관리 코드에서 예외를 throw하는 경우 공용 언어 런타임은 HRESULT를 COM 개체에 전달할 수 있습니다. 비관리 코드에서 오류 HRESULT를 반환하여 메서드가 실패할 경우, 런타임은 관리 코드에서 catch할 수 있는 예외를 던집니다.

런타임은 COM interop의 HRESULT를 보다 구체적인 예외에 자동으로 매핑합니다. 예를 들어, E_ACCESSDENIED는 UnauthorizedAccessException로, E_OUTOFMEMORY는 OutOfMemoryException로 변환됩니다.

HRESULT가 사용자 지정 결과이거나 런타임에 알 수 없는 경우 런타임은 제네릭 COMException을 클라이언트에 전달합니다. **COMException**의 **ErrorCode** 속성에는 HRESULT 값이 포함됩니다.

IErrorInfo를 사용하여 작업하기

COM에서 관리 코드로 오류가 전달되면 런타임은 예외 개체를 오류 정보로 채웁니다. IErrorInfo를 지원하고 HRESULTS를 반환하는 COM 개체는 관리 코드 예외에 이 정보를 제공합니다. 예를 들어 런타임은 COM 오류의 설명을 예외의 Message 속성에 매핑합니다. HRESULT에서 추가 오류 정보를 제공하지 않으면 런타임은 예외의 많은 속성을 기본값으로 채웁니다.

비관리 코드에서 메서드가 실패하는 경우 예외를 관리 코드 세그먼트에 전달할 수 있습니다. HRESULTS 및 예외 항목에는 HRESULTS가 런타임 예외 개체에 매핑되는 방법을 보여 주는 테이블이 포함되어 있습니다.

참고하십시오

- 예외

Last updated on 2026. 03. 31.

예외에 대한 모범 사례

적절한 예외 처리는 애플리케이션 안정성에 필수적입니다. 앱이 충돌하지 않도록 예상 예외를 의도적으로 처리할 수 있습니다. 그러나 크래시된 앱은 정의되지 않은 동작을 가진 앱보다 더 안정적이고 진단 가능합니다.

이 문서에서는 예외를 처리하고 만드는 모범 사례를 설명합니다.

예외 처리

다음 모범 사례는 예외를 처리하는 방법에 관한 것입니다.

- try/catch/finally 블록을 사용하여 오류를 처리하거나 리소스를 해제
- 예외 방지하기 위한 일반적인 조건 처리
- 취소 및 비동기 예외 캡처
- 예외를 방지할 수 있도록 클래스 디자인
- 예외 인해 메서드가 완료되지 않은 경우 복원 상태
- 예외를 적절히 캡처하고 재투척하기

try/catch/finally 블록을 사용하여 오류 복구 또는 리소스 해제

잠재적으로 예외를 생성할 수 있는 코드와 앱이 해당 예외에서 복구할 수 있는 경우 코드 주위에 `try`/`catch` 블록을 사용합니다. `catch` 블록에서 항상 가장 파생된 것에서 가장 적게 파생된 예외로 예외를 정렬합니다. (모든 예외는 Exception 클래스에서 파생됩니다. 파생된 예외는 기본 예외 클래스에 대한 `catch` 절 앞에 오는 `catch` 절에서 처리되지 않습니다.) 코드가 예외에서 복구할 수 없는 경우 해당 예외를 catch하지 마세요. 가능하면 메서드를 호출 스택 위로 이동하여 복구할 수 있도록 설정합니다.

`using` 문 또는 `finally` 블록으로 할당된 리소스를 정리합니다. 예외가 발생하면 `using` 문을 사용하여 리소스를 자동으로 해제하는 것이 좋습니다. `finally` 블록을 사용하여 IDisposable구현하지 않는 리소스를 정리합니다. `finally` 절의 코드는 예외가 throw되는 경우에도 거의 항상 실행됩니다.

예외를 방지하기 위한 일반적인 조건 처리

발생할 가능성이 있지만 예외를 트리거할 수 있는 조건의 경우 예외를 방지하는 방식으로 처리하는 것이 좋습니다. 예를 들어 이미 닫힌 연결을 닫으려고 하면 `InvalidOperationException`를 받게 됩니다. `if` 문을 사용하여 연결을 닫기 전에 연결 상태를 확인하여 방지할 수 있습니다.

```
C#
```

```
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}
```

닫기 전에 연결 상태를 확인하지 않으면 `InvalidOperationException` 예외를 포착할 수 있습니다.

C#

```
try
{
    conn.Close();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

선택하는 방법은 이벤트가 발생할 것으로 예상되는 빈도에 따라 달라집니다.

- 이벤트가 자주 발생하지 않는 경우 예외 처리를 사용합니다. 즉, 이벤트가 실제로 예외적이고 예기치 않은 파일 끝과 같은 오류를 나타내는 경우입니다. 예외 처리를 사용하는 경우 일반 조건에서 더 적은 코드가 실행됩니다.
- 이벤트가 정기적으로 발생하고 정상적인 실행의 일부로 간주될 수 있는 경우 코드에서 오류 조건을 확인합니다. 일반적인 오류 조건을 확인하면 예외를 방지하므로 코드가 더 적게 실행됩니다.

  > **참고**
  >
  > 선행 검사는 대부분의 경우 예외를 제거합니다. 그러나 검사와 작업 사이에 보호된 조건이 변경될 수 있는 경쟁 상태가 존재할 수 있으며, 이러한 경우에는 여전히 예외가 발생할 수 있습니다.

예외를 방지하기 위해 `Try*` 메서드 호출

예외의 성능 비용이 엄청나게 많은 경우 일부 .NET 라이브러리 메서드는 대체 형태의 오류 처리를 제공합니다. 예를 들어, 값이 너무 커서 Int32.Parse로 나타낼 수 없는 경우, OverflowException은 Int32을 던집니다. 그러나 Int32.TryParse 이 예외를 throw하지는 않습니다. 대신 논리값을 반환하며, 성공 시 구문 분석된 유효한 정수를 포함하는 `out` 매개 변수가 있습니다.

Dictionary<TKey,TValue>.TryGetValue 사전에서 값을 가져오는 것과 비슷한 동작을 가지고 있습니다.

취소 및 비동기 예외 처리하기

비동기 메서드를 호출할 때는 OperationCanceledException에서 파생되는 TaskCanceledException을 잡는 대신 `OperationCanceledException`을 catch하는 것이 좋습니다. 취소가 요청되면 많은 비동기 메서드가 OperationCanceledException 예외를 throw합니다. 이러한 예외를 사용하면 실행을 효율적으로 중지하고 취소 요청이 관찰되면 호출 스택을 해제할 수 있습니다.

비동기 메서드는 실행 중에 발생하는 예외를 반환하는 태스크에 저장합니다. 예외가 반환된 태스크에 저장될 경우, 태스크가 대기 중일 때 해당 예외가 throw됩니다. ArgumentException같은 사용 예외는 여전히 동기적으로 던져집니다. 자세한 내용은 비동기 예외를 참조하세요.

예외를 방지할 수 있도록 클래스 디자인

클래스는 예외를 트리거하는 호출을 방지할 수 있는 메서드 또는 속성을 제공할 수 있습니다. 예를 들어 FileStream 클래스는 파일의 끝에 도달했는지 여부를 확인하는 데 도움이 되는 메서드를 제공합니다. 이러한 메서드를 호출하여 파일의 끝을 지나 읽는 경우 throw되는 예외를 방지할 수 있습니다. 다음 예제에서는 예외를 트리거하지 않고 파일의 끝까지 읽는 방법을 보여줍니다.

C#

```
class FileRead
{
    public static void ReadAll(FileStream fileToRead)
    {
        ArgumentNullException.ThrowIfNull(fileToRead);

        int b;

        // Set the stream position to the beginning of the file.
        fileToRead.Seek(0, SeekOrigin.Begin);

        // Read each byte to the end of the file.
        for (int i = 0; i < fileToRead.Length; i++)
        {
            b = fileToRead.ReadByte();
            Console.Write(b.ToString());
            // Or do something else with the byte.
        }
    }
}
```

예외를 방지하는 또 다른 방법은 예외를 throw하는 대신 가장 일반적인 오류 사례에 대한 `null`(또는 기본값)를 반환하는 것입니다. 일반적인 오류 사례는 일반적인 제어 흐름으로 간주될 수 있습니다. 이러한 경우 `null`(또는 기본값)를 반환하면 앱에 대한 성능 영향을 최소화할 수 있습니다.

값 형식의 경우 앱의 오류 표시기로 `Nullable<T>` 또는 `default` 사용할지 여부를 고려합니다. `Nullable<Guid>`사용하면 `default` `null`대신 `Guid.Empty` 됩니다. 경우에 따라 `Nullable<T>` 추가하면 값이 있거나 없는 경우 더 명확하게 할 수 있습니다. `Nullable<T>` 추가하면 필요하지 않은지 확인하는 추가 사례가 생성되고 잠재적인 오류 원본을 만드는 데만 도움이 될 수 있습니다.

예외로 인해 메서드가 완료되지 않은 경우의 복원 상태

호출자는 메서드에서 예외가 발생할 때 부작용이 없다는 점을 가정할 수 있어야 합니다. 예를 들어 한 계좌에서 인출하고 다른 계좌에 입금하여 돈을 이체하는 코드가 있을 때, 입금 과정 중에 예외가 발생하는 경우 인출이 완료되지 않도록 해야 합니다.

C#

```
public void TransferFunds(Account from, Account to, decimal amount)
{
    from.Withdrawal(amount);
    // If the deposit fails, the withdrawal shouldn't remain in effect.
    to.Deposit(amount);
}
```

앞의 메서드는 예외를 직접 throw하지 않습니다. 그러나 입금 작업이 실패할 경우 인출이 취소되도록 메서드를 작성해야 합니다.

이 상황을 처리하는 한 가지 방법은 예금 거래에 의해 발생한 예외를 잡아내어 인출을 취소하는 것입니다.

C#

```
private static void TransferFunds(Account from, Account to, decimal amount)
{
    string withdrawalTrxID = from.Withdrawal(amount);
    try
    {
        to.Deposit(amount);
    }
    catch
    {
        from.RollbackTransaction(withdrawalTrxID);
        throw;
    }
}
```

이 예제에서는 `throw` 사용하여 원래 예외를 다시 throw하여 호출자가 InnerException 속성을 검사하지 않고도 문제의 실제 원인을 쉽게 확인할 수 있도록 합니다. 대안은 새 예외를 throw하고 원래 예외를 내부 예외로 포함하는 것입니다.

C#

```
catch (Exception ex)
{
    from.RollbackTransaction(withdrawalTrxID);
    throw new TransferFundsException("Withdrawal failed.", innerException: ex)
    {
        From = from,
        To = to,
        Amount = amount
    };
}
```

예외를 적절히 포착하고 다시 던집니다.

예외가 발생하면 포함된 정보 중 일부는 스택 추적입니다. 스택 추적은 예외를 throw하는 메서드에서 시작하여 예외를 catch하는 메서드로 끝나는 메서드 호출 계층 구조의 목록입니다. `throw` 문에서 예외를 지정하여 예외를 다시 throw하는 경우(예: `throw e`) 스택 추적은 현재 메서드에서 다시 시작되고 예외를 throw한 원래 메서드와 현재 메서드 간의 메서드 호출 목록이 손실됩니다. 예외를 제외하고 원래 스택 추적 정보를 유지하려면 예외를 다시 발생시키는 위치에 따라 두 가지 옵션이 있습니다.

- 예외 인스턴스를 catch한 처리기(`catch` 블록) 내에서 예외를 다시 throw하는 경우 예외를 지정하지 않고 `throw` 문을 사용합니다. CA2200 코드 분석 규칙은 실수로 스택 추적 정보를 잃을 수 있는 위치를 코드에서 찾는 데 도움이 됩니다.
- 처리기(예:`catch` 블록) 이외의 위치에서 예외를 다시 throw하는 경우, ExceptionDispatchInfo.Capture(Exception)을 사용하여 처리기에서 예외를 캡처하고, 다시 throw하려는 경우 ExceptionDispatchInfo.Throw()를 사용하세요. ExceptionDispatchInfo.SourceException 속성을 사용하여 캡처된 예외를 검사할 수 있습니다.

다음 예제에서는 ExceptionDispatchInfo 클래스를 사용할 수 있는 방법 및 출력의 모양을 보여줄 수 있습니다.

C#

```
ExceptionDispatchInfo? edi = null;
try
{
    var txt = File.ReadAllText(@"C:\temp\file.txt");
}
catch (FileNotFoundException e)
{
    edi = ExceptionDispatchInfo.Capture(e);
}

// ...
```

```
Console.WriteLine("I was here.");

if (edi is not null)
    edi.Throw();
```

예제 코드의 파일이 없으면 다음 출력이 생성됩니다.

출력

```
I was here.
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\temp\file.txt'.
File name: 'C:\temp\file.txt'
   at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String fullPath,
FileMode mode, FileAccess access, FileShare share, FileOptions options)
   at Microsoft.Win32.SafeHandles.SafeFileHandle.Open(String fullPath, FileMode
mode, FileAccess access, FileShare share, FileOptions options, Int64
preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.Strategies.OSFileStreamStrategy..ctor(String path, FileMode mode,
FileAccess access, FileShare share, FileOptions options, Int64 preallocationSize,
Nullable`1 unixCreateMode)
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(String path,
FileMode mode, FileAccess access, FileShare share, FileOptions options, Int64
preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.StreamReader.ValidateArgsAndOpenPath(String path, Encoding encoding,
Int32 bufferSize)
   at System.IO.File.ReadAllText(String path, Encoding encoding)
   at Example.ProcessFile.Main() in C:\repos\ConsoleApp1\Program.cs:line 12
--- End of stack trace from previous location ---
   at Example.ProcessFile.Main() in C:\repos\ConsoleApp1\Program.cs:line 24
```

예외를 던지기

다음 모범 사례는 예외를 throw하는 방법에 관한 것입니다.

- 미리 정의된 예외 형식 사용
- 예외 작성기 메서드 사용
- 지역화된 문자열 메시지 포함
- 적절한 문법 사용
- throw 문장을 적절히 배치하세요
- 절에서 예외를 발생하지 마세요.
- 예기치 않은 위치에서 예외를 발생시키지 마세요.
- 인수 유효성 검사 예외를 동기적으로 Throw

미리 정의된 예외 형식 사용

미리 정의된 예외 클래스가 적용되지 않는 경우에만 새 예외 클래스를 도입합니다. 예를 들어:

- 개체의 현재 상태를 고려할 때 속성 집합 또는 메서드 호출이 적절하지 않은 경우 InvalidOperationException 예외를 throw합니다.
- 잘못된 매개 변수가 전달되면 ArgumentException 예외를 발생시키거나 ArgumentException에서 파생된 미리 정의된 클래스 중 하나를 발생시킵니다.

> ⓘ **참고**
>
> 가능한 경우 미리 정의된 예외 형식을 사용하는 것이 가장 좋지만, , AccessViolationException, IndexOutOfRangeException 및 NullReferenceException같은 일부 StackOverflowException 예외 형식을 발생시켜서는 안 됩니다. 자세한 내용은 **CA2201: 예약된 예외 유형을 발생시키지 마십시오**.

예외 작성기 메서드 사용

클래스는 구현의 여러 위치에서 동일한 예외를 throw하는 것이 일반적입니다. 과도한 코드를 방지하려면 예외를 만들고 반환하는 도우미 메서드를 만듭니다. 예를 들어:

C#

```
class FileReader
{
    private readonly string _fileName;

    public FileReader(string path)
    {
        _fileName = path;
    }

    public byte[] Read(int bytes)
    {
        byte[] results = FileUtils.ReadFromFile(_fileName, bytes) ?? throw
NewFileIOException();
        return results;
    }

    static FileReaderException NewFileIOException()
    {
        string description = "My NewFileIOException Description";

        return new FileReaderException(description);
    }
}
```

일부 주요 .NET 예외 형식에는 예외를 할당하고 throw하는 정적 `throw` 도우미 메서드가 있습니다. 해당 예외 형식을 생성하고 throw하는 대신 다음 메서드를 호출해야 합니다.

- ArgumentNullException.ThrowIfNull
- ArgumentException.ThrowIfNullOrEmpty(String, String)
- ArgumentException.ThrowIfNullOrWhiteSpace(String, String)
- ArgumentOutOfRangeException.ThrowIfZero<T>(T, String)
- ArgumentOutOfRangeException.ThrowIfNegative<T>(T, String)
- ArgumentOutOfRangeException.ThrowIfEqual<T>(T, T, String)
- ArgumentOutOfRangeException.ThrowIfLessThan<T>(T, T, String)
- ArgumentOutOfRangeException.ThrowIfNotEqual<T>(T, T, String)
- ArgumentOutOfRangeException.ThrowIfNegativeOrZero<T>(T, String)
- ArgumentOutOfRangeException.ThrowIfGreaterThan<T>(T, T, String)
- ArgumentOutOfRangeException.ThrowIfLessThanOrEqual<T>(T, T, String)
- ArgumentOutOfRangeException.ThrowIfGreaterThanOrEqual<T>(T, T, String)
- ObjectDisposedException.ThrowIf

> **팁**
>
> 다음 코드 분석 규칙을 사용하면 이러한 정적 `throw` 도우미를 활용할 수 있는 위치를 코드에서 찾을 수 있습니다. **CA1510**, **CA1511**, **CA1512**및 **CA1513**.

비동기 메서드를 구현하는 경우 취소가 요청되었는지 확인한 다음 CancellationToken.ThrowIfCancellationRequested()생성 및 throw하는 대신 OperationCanceledException 호출합니다. 자세한 내용은 CA2250참조하세요.

지역화된 문자열 메시지 포함

사용자가 보는 오류 메시지는 예외 클래스의 이름이 아니라 throw된 예외의 Exception.Message 속성에서 파생됩니다. 일반적으로 Exception.Message `message` 인수에 메시지 문자열을 전달하여 속성에 값을 할당합니다.

지역화된 애플리케이션의 경우 애플리케이션에서 throw할 수 있는 모든 예외에 대해 지역화된 메시지 문자열을 제공해야 합니다. 리소스 파일을 사용하여 지역화된 오류 메시지를 제공합니다. 애플리케이션을 지역화하고 지역화된 문자열을 검색하는 방법에 대한 자세한 내용은 다음 문서를 참조하세요.

- 방법: 지역화된 예외 메시지를 사용하여 사용자 정의 예외 만들기
- .NET 앱의 리소스
- System.Resources.ResourceManager

적절한 문법 사용

명확한 문장을 작성하고 끝에 문장 부호를 포함하세요. Exception.Message 속성에 할당된 문자열의 각 문장은 마침표로 끝나야 합니다. 예를 들어, "로그 테이블이 오버플로되었습니다."라는 문장은 올바른 문법과 문장 부호를 사용합니다.

throw 문을 적절하게 배치하세요

스택 추적이 유용하도록 throw 문을 위치시킵니다. 스택 트레이스는 예외가 throw되는 문에서 시작해서 예외를 catch하는 `catch` 문에서 끝납니다.

finally 절에서는 예외를 발생시키지 마세요.

`finally` 절에서 예외를 발생시키지 마십시오. 자세한 내용은 CA2219 코드 분석 규칙을 참조하세요.

예기치 않은 위치에서 예외를 발생시키지 마세요.

`Equals`, `GetHashCode` 및 `ToString` 메서드, 정적 생성자 및 같음 연산자와 같은 일부 메서드는 예외를 throw해서는 안 됩니다. 자세한 내용은 CA1065 코드 분석 규칙을 참조하세요.

인수 유효성 검사 예외를 동기적으로 발생시키다

태스크 반환 메서드에서는 메서드의 비동기 부분을 입력하기 전에 인수의 유효성을 검사하고 ArgumentException 및 ArgumentNullException같은 해당 예외를 throw해야 합니다. 메서드의 비동기 부분에서 발생하는 예외는 반환된 작업에 저장되고, 작업이 대기될 때까지 나타나지 않는다. 자세한 내용은 작업 반환 메서드 예외를 참조하세요.

사용자 지정 예외 형식

다음 모범 사례는 사용자 지정 예외 유형과 관련이 있습니다.

- 예외 클래스 이름을 로 끝내고 Exception를 사용하세요.
- 세 개의 생성자 포함
- 필요에 따라 더 많은 속성 제공

`Exception` 사용하여 예외 클래스 이름 종료

사용자 지정 예외가 필요한 경우 적절하게 이름을 지정하고 Exception 클래스에서 파생합니다. 예를 들어:

```
C#
```

```
public class MyFileNotFoundException : Exception
{
}
```

3개의 생성자 포함

사용자 고유의 예외 클래스를 만들 때 매개 변수가 없는 생성자, 문자열 메시지를 사용하는 생성자, 문자열 메시지와 내부 예외를 사용하는 생성자 등 세 가지 이상의 공통 생성자를 사용합니다.

- 기본값을 사용하는 Exception().
- 문자열 메시지를 수락하는 Exception(String).
- Exception(String, Exception)- 문자열 메시지 및 내부 예외를 허용합니다.

예를 들어 방법:사용자 정의 예외 만들기를 참조하세요.

필요에 따라 더 많은 속성 제공

추가 정보가 유용한 프로그래밍 방식 시나리오가 있는 경우에만 예외에 대한 추가 속성(사용자 지정 메시지 문자열 외에)을 제공합니다. 예를 들어 FileNotFoundExceptionFileName 속성을 제공합니다.

참고

- 예외를 위한 디자인 지침

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

System.AccessViolationException 클래스

2025. 05. 29.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

코드가 할당되지 않았거나 액세스 권한이 없는 메모리를 읽거나 쓰려고 할 때 관리되지 않거나 안전하지 않은 코드에서 액세스 위반이 발생합니다. 포인터에 잘못된 값이 있기 때문에 일반적으로 발생합니다. 잘못된 포인터를 통한 모든 읽기 또는 쓰기가 액세스 위반으로 이어지는 것은 아니므로 액세스 위반은 일반적으로 잘못된 포인터를 통해 여러 읽기 또는 쓰기가 발생했으며 메모리가 손상되었을 수 있음을 나타냅니다. 따라서 액세스 위반은 거의 항상 심각한 프로그래밍 오류를 나타냅니다. 이러한 AccessViolationException 심각한 오류를 명확하게 식별합니다.

완전히 확인 가능한 관리 코드로 구성된 프로그램에서는 모든 참조가 유효하거나 null이며 액세스 위반은 불가능합니다. 확인 가능한 코드에서 null 참조를 참조하려는 모든 작업은 NullReferenceException 예외를 발생시킵니다. AccessViolationException 확인 가능한 관리 코드가 관리되지 않는 코드 또는 안전하지 않은 관리 코드와 상호 작용하는 경우에만 발생합니다.

AccessViolationException 예외 문제 해결

AccessViolationException 안전하지 않은 관리 코드 또는 확인 가능한 관리 코드가 관리되지 않는 코드와 상호 작용하는 경우에만 예외가 발생할 수 있습니다.

- 안전하지 않은 관리 코드에서 발생하는 액세스 위반은 플랫폼에 따라 NullReferenceException 예외나 AccessViolationException 예외로 표현될 수 있습니다.
- 제어되지 않는 코드에서의 접근 위반이 관리 코드로 전달될 때, 항상 AccessViolationException 예외로 래핑됩니다.

두 경우 모두 AccessViolationException 예외의 원인을 식별하고 수정하는 방법은 다음과 같습니다.

- 액세스하려는 메모리가 할당되었는지 확인합니다. AccessViolationException 예외는 보호된 메모리에 액세스하려는 시도에서 항상 발생합니다. 즉, 할당되지 않았거나 프로세스가 소유하지 않은 메모리에 접근할 때 발생합니다.

  자동 메모리 관리는 .NET 런타임에서 제공하는 서비스 중 하나입니다. 관리 코드가 관리되지 않는 코드와 동일한 기능을 제공하는 경우 관리 코드로 이동하여 이 기능을 활용하는 것이 좋습니다. 자세한 내용은 자동 메모리 관리를 참조하세요.

- 액세스하려는 메모리가 손상되지 않았는지 확인합니다. 잘못된 포인터를 통해 여러 읽기 또는 쓰기 작업이 발생한 경우 메모리가 손상되었을 수 있습니다. 일반적으로 미리 정의된 버퍼 외부의 주소를 읽거나 쓸 때 발생합니다.

AccessViolationException 및 try/catch 블록

AccessViolationException .NET 런타임에서 던져진 예외는, 런타임에 의해 예약된 메모리 외부에서 발생하는 경우, 구조적 예외 처리기의 `catch` 문에서 처리되지 않습니다.

.NET Framework만: 이러한 AccessViolationException 예외를 처리하려면, 예외가 발생되는 메서드에 HandleProcessCorruptedStateExceptionsAttribute 특성을 적용합니다. 이 변경은 사용자 코드에서 throw된 예외에 AccessViolationException 영향을 주지 않으며, 이러한 예외는 `catch` 문으로 계속해서 catch될 수 있습니다.

> **⊗ 주의**
>
> **HandleProcessCorruptedStateExceptions 특성**은 현재 .NET 버전에서 사용되지 않습니다. 손상된 프로세스 상태-예외에서 복구는 지원되지 않으며 특성이 있는 경우 무시됩니다.

System.Exception 클래스

아티클 • 2025. 03. 26.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Exception 클래스는 모든 예외의 기본 클래스입니다. 오류가 발생하면 시스템 또는 현재 실행 중인 애플리케이션이 오류에 대한 정보가 포함된 예외를 던져 보고를 합니다. 예외가 발생하면 애플리케이션 또는 디폴트 예외 처리기에서 처리됩니다.

오류 및 예외

런타임 오류는 다양한 이유로 발생할 수 있습니다. 그러나 코드에서 모든 오류를 예외로 처리해야 하는 것은 아닙니다. 다음은 런타임에 발생할 수 있는 오류의 몇 가지 범주와 이에 대응하는 적절한 방법입니다.

- **사용 오류입니다**. 사용 오류는 예외가 발생할 수 있는 프로그램 논리의 오류를 나타냅니다. 그러나 오류는 예외 처리를 통해서가 아니라 잘못된 코드를 수정하여 해결해야 합니다. 예를 들어 다음 예제에서 Object.Equals(Object) 메서드의 재정의는 `obj` 인수가 항상 null이 아니어야 한다고 가정합니다.

```
C#
```

```
using System;

public class Person1
{
   private string _name;

   public string Name
   {
      get { return _name; }
      set { _name = value; }
   }

   public override int GetHashCode()
   {
      return this.Name.GetHashCode();
   }

   public override bool Equals(object obj)
   {
      // This implementation contains an error in program logic:
      // It assumes that the obj argument is not null.
      Person1 p = (Person1) obj;
      return this.Name.Equals(p.Name);
   }
}
```

```
public class UsageErrorsEx1
{
   public static void Main()
   {
      Person1 p1 = new Person1();
      p1.Name = "John";
      Person1 p2 = null;

      // The following throws a NullReferenceException.
      Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
   }
}
```

`obj` `null` 때 발생하는 NullReferenceException 예외는 Object.Equals 재정의를 호출한 다음 다시 컴파일하기 전에 null을 명시적으로 테스트하도록 소스 코드를 수정하여 제거할 수 있습니다. 다음 예제에서는 `null` 인수를 처리 하는 수정 된 소스 코드를 포함 합니다.

C#

```
using System;

public class Person2
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        // This implementation handles a null obj argument.
        Person2 p = obj as Person2;
        if (p == null)
            return false;
        else
            return this.Name.Equals(p.Name);
    }
}

public class UsageErrorsEx2
{
    public static void Main()
```

```
    {
        Person2 p1 = new Person2();
        p1.Name = "John";
        Person2 p2 = null;

        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
}
// The example displays the following output:
//        p1 = p2: False
```

사용 오류에 예외 처리를 사용하는 대신 Debug.Assert 메서드를 사용하여 디버그 빌드에서 사용 오류를 식별하고 Trace.Assert 메서드를 사용하여 디버그 및 릴리스 빌드 모두에서 사용 오류를 식별할 수 있습니다. 자세한 내용은 관리 코드 어설션을 참조하세요.

- **프로그램 오류입니다.** 프로그램 오류는 버그 없는 코드를 작성하여 반드시 피할 수 없는 런타임 오류입니다.

  경우에 따라 프로그램 오류는 예상 또는 일상적인 오류 조건을 반영할 수 있습니다. 이 경우 예외 처리를 사용하여 프로그램 오류를 처리하지 않고 작업을 다시 시도하는 것이 좋습니다. 예를 들어 사용자가 특정 형식으로 날짜를 입력해야 하는 경우 날짜 문자열을 DateTime 값으로 변환할 수 없는 경우 FormatException 예외를 throw하는 DateTime.ParseExact 메서드를 사용하는 대신 구문 분석 작업이 성공했는지 여부를 나타내는 Boolean 값을 반환하는 DateTime.TryParseExact 메서드를 호출하여 날짜 문자열을 구문 분석할 수 있습니다. 마찬가지로 사용자가 존재하지 않는 파일을 열려고 하면 먼저 File.Exists 메서드를 호출하여 파일이 있는지 확인하고 파일이 없는 경우 만들 것인지 묻는 메시지를 표시할 수 있습니다.

  다른 경우에 프로그램 오류는 코드에서 처리할 수 있는 예기치 않은 오류 조건을 반영합니다. 예를 들어 파일이 있는지 확인했더라도 파일을 열기 전에 삭제하거나 손상되었을 수 있습니다. 이 경우 StreamReader 개체를 인스턴스화하거나 Open 메서드를 호출하여 파일을 열려고 하면 FileNotFoundException 예외가 발생할 수 있습니다. 이러한 경우 예외 처리를 사용하여 오류에서 복구해야 합니다.

- **시스템 오류.** 시스템 오류는 의미 있는 방식으로 프로그래밍 방식으로 처리할 수 없는 런타임 오류입니다. 예를 들어 공용 언어 런타임에서 추가 메모리를 할당할 수 없는 경우 모든 메서드가 OutOfMemoryException 예외를 throw할 수 있습니다. 일반적으로 시스템 오류는 예외 처리를 사용하여 처리되지 않습니다. 대신 AppDomain.UnhandledException 같은 이벤트를 사용하고 Environment.FailFast 메서드를 호출하여 예외 정보를 기록하고 애플리케이션이 종료되기 전에 사용자에게 오류를 알릴 수 있습니다.

블록 시도/catch

공용 언어 런타임은 예외를 개체로 표현하고 프로그램 코드와 예외 처리 코드를 `try` 블록 및 `catch` 블록으로 분리하는 것을 기반으로 하는 예외 처리 모델을 제공합니다. 각각 특정 유형의 예외를 처리하도록 설계된 하나 이상의 `catch` 블록 또는 다른 블록보다 더 구체적인 예외를 catch하도록 설계된 하나의 블록이 있을 수 있습니다.

애플리케이션이 애플리케이션 코드 블록을 실행하는 동안 발생하는 예외를 처리하는 경우 코드는 `try` 문 내에 배치되어야 하며 `try` 블록이라고 합니다. `try` 블록에서 throw된 예외를 처리하는 애플리케이션 코드는 `catch` 문 내에 배치되며 `catch` 블록이라고 합니다. 0개 이상의 `catch` 블록이 `try` 블록과 연결되고 각 `catch` 블록에는 처리되는 예외 유형을 결정하는 형식 필터가 포함됩니다.

`try` 블록에서 예외가 발생하면 시스템은 예외를 처리하는 `catch` 블록을 찾을 때까지 애플리케이션 코드에 표시되는 순서대로 연결된 `catch` 블록을 검색합니다. `catch` 블록은 catch 블록의 형식 필터가 `T` 또는 `T`으로부터 파생된 모든 형식을 지정할 경우, `T` 형식의 예외를 처리합니다. 시스템에서 예외를 처리하는 첫 번째 `catch` 블록을 찾은 후 검색을 중지합니다. 이러한 이유로 애플리케이션 코드에서 형식을 처리하는 `catch` 블록은 이 섹션의 예제에 설명된 대로 해당 기본 형식을 처리하는 `catch` 블록 앞에 지정되어야 합니다. `System.Exception` 처리하는 catch 블록이 마지막으로 지정됩니다.

현재 `try` 블록에 연결된 `catch` 블록 중 어느 것도 예외를 처리하지 않고, 현재 `try` 블록이 현재 호출의 다른 `try` 블록 내에 중첩되어 있는 경우, 다음 외부 `try` 블록에 연결된 `catch` 블록이 검색됩니다. 예외에 대한 `catch` 블록이 없으면 시스템은 현재 호출에서 이전 중첩 수준을 검색합니다. 현재 호출에서 예외에 대한 `catch` 블록이 없으면 예외가 호출 스택 위로 전달되고 이전 스택 프레임은 예외를 처리하는 `catch` 블록을 검색합니다. 호출 스택의 검색은 예외가 처리되거나 호출 스택에 더 이상 프레임이 없을 때까지 계속됩니다. 예외를 처리하는 `catch` 블록을 찾지 않고 호출 스택의 맨 위에 도달하면 기본 예외 처리기가 이를 처리하고 애플리케이션이 종료됩니다.

F# try.. with 식

F#은 `catch` 블록을 사용하지 않습니다. 대신, 발생한 예외는 단일 `with` 블록으로 패턴 매칭됩니다. 문이 아닌 식이므로 모든 경로가 동일한 유형을 반환해야 합니다. 더 알아보려면 시도...와 표현을 참조하세요.

예외 유형 기능

예외 유형은 다음 기능을 지원합니다.

- 오류를 설명하는 사람이 읽을 수 있는 텍스트입니다. 예외가 발생하면 런타임에서 사용자에게 오류의 특성을 알리고 문제를 해결하기 위한 작업을 제안하는 문자 메

시지를 사용할 수 있게 합니다. 이 문자 메시지는 예외 개체의 Message 속성에 보관됩니다. 예외 개체를 만드는 동안 해당 특정 예외의 세부 정보를 설명하는 텍스트 문자열을 생성자에 전달할 수 있습니다. 생성자에 오류 메시지 인수가 제공되지 않으면 기본 오류 메시지가 사용됩니다. 자세한 내용은 Message 속성을 참조하세요.

- 예외가 발생했을 때의 호출 스택 상태입니다. StackTrace 속성은 코드에서 오류가 발생하는 위치를 확인하는 데 사용할 수 있는 스택 추적을 전달합니다. 스택 추적에는 호출된 모든 메서드와 호출이 이루어지는 원본 파일의 줄 번호가 나열됩니다.

예외 클래스 속성

Exception 클래스에는 코드 위치, 형식, 도움말 파일 및 예외 이유를 식별하는 데 도움이 되는 여러 속성이 포함되어 있습니다. StackTrace, InnerException, Message, HelpLink, HResult, Source, TargetSite및 Data.

둘 이상의 예외 간에 인과 관계가 있는 경우 InnerException 속성은 이 정보를 유지 관리합니다. 외부 예외는 이 내부 예외에 대한 응답으로 발생합니다. 외부 예외를 처리하는 코드는 이전 내부 예외의 정보를 사용하여 오류를 보다 적절하게 처리할 수 있습니다. 예외에 대한 추가 정보는 Data 속성에 키/값 쌍의 컬렉션으로 저장할 수 있습니다.

예외 개체를 만드는 동안 생성자에 전달되는 오류 메시지 문자열은 지역화되어야 하며 ResourceManager 클래스를 사용하여 리소스 파일에서 제공할 수 있습니다. 지역화된 리소스에 대한 자세한 내용은 위성 어셈블리 만들기 및 리소스 패키징 및 배포 항목을 참조하세요.

사용자에게 예외가 발생한 이유에 대한 광범위한 정보를 제공하기 위해 HelpLink 속성은 도움말 파일에 대한 URL(또는 URN)을 보유할 수 있습니다.

Exception 클래스는 HRESULT `COR_E_EXCEPTION`, 값은 0x80131500, 를 사용합니다.

Exception 클래스 인스턴스의 초기 속성 값 목록은 Exception 생성자를 참조하세요.

성능 고려 사항

예외를 throw하거나 처리하면 상당한 양의 시스템 리소스와 실행 시간이 사용됩니다. 예측 가능한 이벤트 또는 흐름 제어를 처리하지 않고 진정으로 특별한 조건을 처리하기 위해서만 예외를 throw합니다. 예를 들어 클래스 라이브러리를 개발할 때와 같은 경우에 메서드 인수가 유효하지 않은 경우 유효한 매개 변수를 사용하여 메서드를 호출해야 하므로 예외를 throw하는 것이 합리적입니다. 잘못된 메서드 인수가 사용 오류의 결과가 아닌 경우 특별한 문제가 발생했음을 의미합니다. 반대로, 사용자가 잘못된 데이터를 입력할 것으로 예상할 수 있으므로 사용자 입력이 잘못된 경우 예외를 throw하지 마세요. 대신 사용자가 유효한 입력을 입력할 수 있도록 재시도 메커니즘을 제공합니다. 예외를 사용하

여 사용 오류를 처리해서는 안 됩니다. 대신 어설션 사용하여 사용 오류를 식별하고 수정합니다.

또한 반환 코드가 충분한 경우 예외를 throw하지 마세요. 반환 코드를 예외로 변환하지 마세요. 정기적으로 예외를 catch하여 무시하고 처리를 계속하지 마세요.

예외를 다시 발생시키기

대부분의 경우 예외 처리기는 예외를 호출자에게 전달하려고 합니다. 이 문제는 다음에서 가장 자주 발생합니다.

- .NET 클래스 라이브러리 또는 다른 클래스 라이브러리의 메서드에 대한 호출을 차례로 래핑하는 클래스 라이브러리입니다.
- 치명적인 예외가 발생하는 애플리케이션 또는 라이브러리입니다. 예외 처리기는 예외를 기록한 다음 예외를 다시 throw할 수 있습니다.

예외를 다시 throw하는 권장 방법은 C#의 throw 문, F#의 reraise 함수 및 식을 포함하지 않고 Visual Basic의 Throw 문을 사용하는 것입니다. 이렇게 하면 예외가 호출자에게 전파될 때 모든 호출 스택 정보가 유지됩니다. 다음 예제에서는 이를 보여 줍니다. 문자열 확장 메서드인 `FindOccurrences` 인수의 유효성을 미리 검사하지 않고 String.IndexOf(String, Int32) 대한 하나 이상의 호출을 래핑합니다.

C#

```
using System;
using System.Collections.Generic;

public static class Library1
{
    public static int[] FindOccurrences(this String s, String f)
    {
        var indexes = new List<int>();
        int currentIndex = 0;
        try
        {
            while (currentIndex >= 0 && currentIndex < s.Length)
            {
                currentIndex = s.IndexOf(f, currentIndex);
                if (currentIndex >= 0)
                {
                    indexes.Add(currentIndex);
                    currentIndex++;
                }
            }
        }
        catch (ArgumentNullException)
        {
```

```
            // Perform some action here, such as logging this exception.

            throw;
        }
        return indexes.ToArray();
    }
}
```

그런 다음 호출자가 `FindOccurrences` 두 번 호출합니다. `FindOccurrences`로의 두 번째 호출에서 호출자는 `null`을 검색 문자열로 전달하여 String.IndexOf(String, Int32) 메서드가 ArgumentNullException 예외를 발생시킵니다. 이 예외는 `FindOccurrences` 메서드에서 처리되고 호출자에게 다시 전달됩니다. throw 문은 식 없이 사용되므로 예제의 출력은 호출 스택이 유지됨을 보여줍니다.

C#

```
public class RethrowEx1
{
    public static void Main()
    {
        String s = "It was a cold day when...";
        int[] indexes = s.FindOccurrences("a");
        ShowOccurrences(s, "a", indexes);
        Console.WriteLine();

        String toFind = null;
        try
        {
            indexes = s.FindOccurrences(toFind);
            ShowOccurrences(s, toFind, indexes);
        }
        catch (ArgumentNullException e)
        {
            Console.WriteLine($"An exception ({e.GetType().Name})
occurred.");
            Console.WriteLine($"Message:{Environment.NewLine}   {e.Message}
{Environment.NewLine}");
            Console.WriteLine($"Stack Trace:{Environment.NewLine}
{e.StackTrace}{Environment.NewLine}");
        }
    }

    private static void ShowOccurrences(String s, String toFind, int[]
indexes)
    {
        Console.Write("'{0}' occurs at the following character positions: ",
                      toFind);
        for (int ctr = 0; ctr < indexes.Length; ctr++)
            Console.Write("{0}{1}", indexes[ctr],
                          ctr == indexes.Length - 1 ? "" : ", ");

        Console.WriteLine();
```

```
    }
}
// The example displays the following output:
//    'a' occurs at the following character positions: 4, 7, 15
//
//    An exception (ArgumentNullException) occurred.
//    Message:
//       Value cannot be null.
//    Parameter name: value
//
//    Stack Trace:
//          at System.String.IndexOf(String value, Int32 startIndex, Int32
count, Stri
//    ngComparison comparisonType)
//       at Library.FindOccurrences(String s, String f)
//       at Example.Main()
```

반면, 다음 문을 사용하여 예외가 다시 던져지는 경우:

C#

```
throw e;
```

... 그런 다음 전체 호출 스택이 유지되지 않고 예제에서 다음 출력을 생성합니다.

출력

```
'a' occurs at the following character positions: 4, 7, 15

An exception (ArgumentNullException) occurred.
Message:
   Value cannot be null.
Parameter name: value

Stack Trace:
      at Library.FindOccurrences(String s, String f)
   at Example.Main()
```

다소 더 번거로운 대안은 새로운 예외를 발생시키고, 내부 예외에 원래 예외의 호출 스택 정보를 보존하는 것입니다. 그런 다음 호출자는 새 예외의 InnerException 속성을 사용하여 스택 프레임 및 원래 예외에 대한 기타 정보를 검색할 수 있습니다. 이 경우 throw 문은 다음과 같이 작성됩니다.

C#

```
throw new ArgumentNullException("You must supply a search string.", e);
```

예외를 처리하는 사용자 코드는 다음 예외 처리기에서 보여 주듯이 InnerException 속성에 원래 예외에 대한 정보가 포함되어 있음을 알아야 합니다.

C#

```
try
{
    indexes = s.FindOccurrences(toFind);
    ShowOccurrences(s, toFind, indexes);
}
catch (ArgumentNullException e)
{
    Console.WriteLine($"An exception ({e.GetType().Name}) occurred.");
    Console.WriteLine($"   Message:{Environment.NewLine}{e.Message}");
    Console.WriteLine($"   Stack Trace:{Environment.NewLine}
{e.StackTrace}");
    Exception ie = e.InnerException;
    if (ie != null)
    {
        Console.WriteLine("   The Inner Exception:");
        Console.WriteLine($"      Exception Name: {ie.GetType().Name}");
        Console.WriteLine($"      Message: {ie.Message}
{Environment.NewLine}");
        Console.WriteLine($"      Stack Trace:{Environment.NewLine}
{ie.StackTrace}{Environment.NewLine}");
    }
}
// The example displays the following output:
//    'a' occurs at the following character positions: 4, 7, 15
//
//    An exception (ArgumentNullException) occurred.
//       Message: You must supply a search string.
//
//       Stack Trace:
//          at Library.FindOccurrences(String s, String f)
//       at Example.Main()
//
//       The Inner Exception:
//          Exception Name: ArgumentNullException
//          Message: Value cannot be null.
//    Parameter name: value
//
//          Stack Trace:
//          at System.String.IndexOf(String value, Int32 startIndex, Int32
count, Stri
//    ngComparison comparisonType)
//       at Library.FindOccurrences(String s, String f)
```

표준 예외 선택

예외를 throw해야 하는 경우 사용자 지정 예외를 구현하는 대신 .NET에서 기존 예외 형식을 사용할 수 있습니다. 다음 두 가지 조건에서 표준 예외 형식을 사용해야 합니다.

- 사용 오류(즉, 메서드를 호출하는 개발자가 만든 프로그램 논리의 오류)로 인해 예외를 발생시킵니다. 일반적으로 ArgumentException, ArgumentNullException, InvalidOperationException또는 NotSupportedException같은 예외를 throw합니다. 예외 개체를 인스턴스화할 때 예외 개체의 생성자에 제공하는 문자열은 개발자가 오류를 수정할 수 있도록 오류를 설명해야 합니다. 자세한 내용은 Message 속성을 참조하세요.

- 기존 .NET 예외를 사용하여 호출자에게 전달할 수 있는 오류를 처리하고 있습니다. 가능한 가장 파생된 예외를 던져야 합니다. 예를 들어 메서드에서 인수가 열거형 형식의 유효한 멤버여야 하는 경우 ArgumentException아닌 InvalidEnumArgumentException(가장 파생된 클래스)를 throw해야 합니다.

다음 표에서는 일반적인 예외 유형 및 예외를 throw할 조건을 나열합니다.

테이블 확장

예외	조건
ArgumentException	메서드에 전달되는 null이 아닌 인수가 잘못되었습니다.
ArgumentNullException	메서드에 전달되는 인수는 `null`.
ArgumentOutOfRangeException	인수가 유효한 값 범위를 벗어났습니다.
DirectoryNotFoundException	디렉터리 경로의 일부가 잘못되었습니다.
DivideByZeroException	정수 또는 Decimal 나누기 연산의 분모는 0입니다.
DriveNotFoundException	드라이브를 사용할 수 없거나 존재하지 않습니다.
FileNotFoundException	파일이 없습니다.
FormatException	값은 `Parse`같은 변환 메서드에 의해 문자열에서 변환할 적절한 형식이 아닙니다.
IndexOutOfRangeException	인덱스가 배열 또는 컬렉션의 범위를 벗어났습니다.
InvalidOperationException	개체의 현재 상태에서 메서드 호출이 잘못되었습니다.
KeyNotFoundException	컬렉션의 멤버에 액세스하기 위해 지정된 키를 찾을 수 없습니다.
NotImplementedException	메서드 또는 작업이 구현되지 않습니다.
NotSupportedException	메서드 또는 작업은 지원되지 않습니다.

예외	조건
ObjectDisposedException	삭제된 개체에 대해 작업이 수행됩니다.
OverflowException	산술, 캐스팅 또는 변환 연산으로 인해 오버플로가 발생합니다.
PathTooLongException	경로 또는 파일 이름이 시스템 정의 최대 길이를 초과합니다.
PlatformNotSupportedException	현재 플랫폼에서는 작업이 지원되지 않습니다.
RankException	잘못된 차원 수를 가진 배열이 메서드에 전달됩니다.
TimeoutException	작업에 할당된 시간 간격이 만료되었습니다.
UriFormatException	잘못된 URI(Uniform Resource Identifier)가 사용됩니다.

사용자 지정 예외 구현

다음 경우 기존 .NET 예외를 사용하여 오류 조건을 처리하는 것은 적절하지 않습니다.

- 예외가 기존 .NET 예외에 매핑할 수 없는 고유한 프로그램 오류를 반영하는 경우
- 예외에 기존 .NET 예외에 적합한 처리와 다른 처리가 필요하거나 유사한 예외에서 예외를 명확하게 구분해야 하는 경우 예를 들어, 대상 정수 형식의 범위를 벗어난 문자열을 숫자로 변환할 때 ArgumentOutOfRangeException 예외를 발생시키는 경우, 메서드를 호출할 때 호출자가 적절한 제한된 값을 제공하지 않아 발생하는 오류에 대해 같은 예외를 사용하지 않아야 합니다.

Exception 클래스는 .NET에서 모든 예외의 기본 클래스입니다. 많은 파생 클래스는 Exception 클래스의 멤버의 상속된 동작을 사용합니다. Exception멤버를 재정의하지 않으며 고유한 멤버를 정의하지도 않습니다.

고유한 예외 클래스를 정의하려면 다음을 수행합니다.

1. Exception상속되는 클래스를 정의합니다. 필요한 경우 예외에 대한 추가 정보를 제공하기 위해 클래스에 필요한 고유 멤버를 정의합니다. 예를 들어 ArgumentException 클래스에는 인수로 인해 예외가 발생한 매개 변수의 이름을 지정하는 ParamName 속성이 포함되며, RegexMatchTimeoutException 속성에는 제한 시간 간격을 나타내는 MatchTimeout 속성이 포함됩니다.
2. 필요한 경우 변경하거나 수정하려는 기능을 가진 상속된 멤버를 재정의하십시오. 대부분의 기존 파생 클래스 Exception는 상속된 멤버의 동작을 재정의하지 않는다는 점을 유의하세요.
3. 사용자 지정 예외 개체를 직렬화할 수 있는지 여부를 확인합니다. Serialization을 사용하면 예외에 대한 정보를 저장할 수 있으며 원격 컨텍스트에서 서버 및 클라이언

트 프록시에서 예외 정보를 공유할 수 있습니다. 예외 개체를 직렬화할 수 있도록 하려면 SerializableAttribute 특성으로 표시합니다.

4. 예외 클래스의 생성자를 정의합니다. 일반적으로 예외 클래스에는 다음 생성자 중 하나 이상이 있습니다.

    - Exception()- 기본값을 사용하여 새 예외 개체의 속성을 초기화합니다.
    - Exception(String)지정한 오류 메시지를 사용하여 새 예외 개체를 초기화합니다.
    - Exception(String, Exception)지정한 오류 메시지와 내부 예외를 사용하여 새 예외 개체를 초기화합니다.
    - Exception(SerializationInfo, StreamingContext)- 직렬화된 데이터에서 새 예외 개체를 초기화하는 `protected` 생성자입니다. 예외 개체를 직렬화할 수 있도록 선택한 경우 이 생성자를 구현해야 합니다.

다음 예제에서는 사용자 지정 예외 클래스의 사용을 보여 줍니다. 클라이언트가 소수가 아닌 시작 번호를 지정하여 소수 시퀀스를 검색하려고 할 때 throw되는 `NotPrimeException` 예외를 정의합니다. 예외는 예외를 발생시킨 소수가 아닌 숫자를 반환하는 새 속성 `NonPrime`정의합니다. serialization에 대한 SerializationInfo 및 StreamingContext 매개 변수가 있는 보호된 매개 변수 없는 생성자와 생성자를 구현하는 것 외에도 `NotPrimeException` 클래스는 `NonPrime` 속성을 지원하는 세 개의 추가 생성자를 정의합니다. 각 생성자는 소수가 아닌 값의 값을 유지하는 것 외에도 기본 클래스 생성자를 호출합니다. `NotPrimeException` 클래스도 SerializableAttribute 특성으로 표시됩니다.

C#

```
using System;
using System.Runtime.Serialization;

[Serializable()]
public class NotPrimeException : Exception
{
   private int notAPrime;

   protected NotPrimeException()
      : base()
   { }

   public NotPrimeException(int value) :
      base(String.Format("{0} is not a prime number.", value))
   {
      notAPrime = value;
   }
```

```
   public NotPrimeException(int value, string message)
      : base(message)
   {
      notAPrime = value;
   }

   public NotPrimeException(int value, string message, Exception
innerException) :
      base(message, innerException)
   {
      notAPrime = value;
   }

   protected NotPrimeException(SerializationInfo info,
                               StreamingContext context)
      : base(info, context)
   { }

   public int NonPrime
   { get { return notAPrime; } }
}
```

다음 예제에 표시된 `PrimeNumberGenerator` 클래스는 Eratosthenes의 Sieve를 사용하여 소수 시퀀스를 2에서 해당 클래스 생성자에 대한 호출에서 클라이언트가 지정한 제한으로 계산합니다. `GetPrimesFrom` 메서드는 지정한 하한보다 크거나 같은 소수를 모두 반환하지만 해당 하한이 소수가 아니면 `NotPrimeException` throw합니다.

C#

```
using System;
using System.Collections.Generic;

[Serializable]
public class PrimeNumberGenerator
{
   private const int START = 2;
   private int maxUpperBound = 10000000;
   private int upperBound;
   private bool[] primeTable;
   private List<int> primes = new List<int>();

   public PrimeNumberGenerator(int upperBound)
   {
      if (upperBound > maxUpperBound)
      {
         string message = String.Format(
                          "{0} exceeds the maximum upper bound of {1}.",
                          upperBound, maxUpperBound);
         throw new ArgumentOutOfRangeException(message);
      }
      this.upperBound = upperBound;
```

```
      // Create array and mark 0, 1 as not prime (True).
      primeTable = new bool[upperBound + 1];
      primeTable[0] = true;
      primeTable[1] = true;

      // Use Sieve of Eratosthenes to determine prime numbers.
      for (int ctr = START; ctr <= (int)Math.Ceiling(Math.Sqrt(upperBound));
            ctr++)
      {
         if (primeTable[ctr]) continue;

         for (int multiplier = ctr; multiplier <= upperBound / ctr;
multiplier++)
            if (ctr * multiplier <= upperBound) primeTable[ctr * multiplier]
= true;
      }
      // Populate array with prime number information.
      int index = START;
      while (index != -1)
      {
         index = Array.FindIndex(primeTable, index, (flag) => !flag);
         if (index >= 1)
         {
            primes.Add(index);
            index++;
         }
      }
   }

   public int[] GetAllPrimes()
   {
      return primes.ToArray();
   }

   public int[] GetPrimesFrom(int prime)
   {
      int start = primes.FindIndex((value) => value == prime);
      if (start < 0)
         throw new NotPrimeException(prime, String.Format("{0} is not a
prime number.", prime));
      else
         return primes.FindAll((value) => value >= prime).ToArray();
   }
}
```

다음 예제에서는 소수가 아닌 숫자로 `GetPrimesFrom` 메서드를 두 번 호출하며, 그 중 하나는 애플리케이션 도메인 경계를 넘습니다. 두 경우 모두 예외가 발생하여 클라이언트 코드에서 성공적으로 처리됩니다.

```
C#
```

```
using System;
using System.Reflection;

class Example1
{
    public static void Main()
    {
        int limit = 10000000;
        PrimeNumberGenerator primes = new PrimeNumberGenerator(limit);
        int start = 1000001;
        try
        {
            int[] values = primes.GetPrimesFrom(start);
            Console.WriteLine($"There are {start} prime numbers from {limit}
to {2}");
        }
        catch (NotPrimeException e)
        {
            Console.WriteLine($"{e.NonPrime} is not prime");
            Console.WriteLine(e);
            Console.WriteLine("--------");
        }

        AppDomain domain = AppDomain.CreateDomain("Domain2");
        PrimeNumberGenerator gen =
(PrimeNumberGenerator)domain.CreateInstanceAndUnwrap(
                                        typeof(Example).Assembly.FullName,
                                        "PrimeNumberGenerator", true,
                                        BindingFlags.Default, null,
                                        new object[] { 1000000 }, null,
null);
        try
        {
            start = 100;
            Console.WriteLine(gen.GetPrimesFrom(start));
        }
        catch (NotPrimeException e)
        {
            Console.WriteLine($"{e.NonPrime} is not prime");
            Console.WriteLine(e);
            Console.WriteLine("--------");
        }
    }
}
```

예시

다음 예제에서는 ArithmeticException 오류를 처리하도록 정의된 `catch`(F#의`with`) 블록을 보여 줍니다. ArithmeticException에서 파생된 DivideByZeroException 때문에

DivideByZeroException 오류에 대해 명시적으로 정의된 `catch` 블록이 없으므로 이 `catch` 블록도 DivideByZeroException 오류를 잡습니다.

C#

```
using System;

class ExceptionTestClass
{
   public static void Main()
   {
      int x = 0;
      try
      {
         int y = 100 / x;
      }
      catch (ArithmeticException e)
      {
         Console.WriteLine($"ArithmeticException Handler: {e}");
      }
      catch (Exception e)
      {
         Console.WriteLine($"Generic Exception Handler: {e}");
      }
   }
}
/*
This code example produces the following results:

ArithmeticException Handler: System.DivideByZeroException: Attempted to
divide by zero.
   at ExceptionTestClass.Main()

*/
```

System.Exception.Data 속성

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

속성에서 반환된 System.Collections.IDictionary 개체를 Data 사용하여 예외와 관련된 추가 정보를 저장하고 검색합니다. 정보는 임의의 수의 사용자 정의 키/값 쌍 형식입니다. 각 키/값 쌍의 키 구성 요소는 일반적으로 식별 문자열인 반면 쌍의 값 구성 요소는 모든 유형의 개체일 수 있습니다.

키/값 쌍 보안

속성에서 반환된 컬렉션에 저장된 키/값 쌍은 Data 안전하지 않습니다. 애플리케이션이 중첩된 일련의 루틴을 호출하고 각 루틴에 예외 처리기가 포함된 경우 결과 호출 스택에는 해당 예외 처리기의 계층이 포함됩니다. 하위 수준 루틴에서 예외를 throw하는 경우 호출 스택 계층 구조의 모든 상위 수준 예외 처리기는 다른 예외 처리기에서 컬렉션에 저장된 키/값 쌍을 읽거나 수정할 수 있습니다. 즉, 키/값 쌍의 정보가 기밀이 아니며 키/값 쌍의 정보가 손상된 경우 애플리케이션이 올바르게 작동한다는 것을 보장해야 합니다.

주요 충돌

키 충돌은 다른 예외 처리기가 키/값 쌍에 액세스하기 위해 동일한 키를 지정하는 경우에 발생합니다. 키 충돌의 결과로 하위 수준 예외 처리기가 실수로 더 높은 수준의 예외 처리기와 통신할 수 있고 이 통신으로 인해 미묘한 프로그램 오류가 발생할 수 있으므로 애플리케이션을 개발할 때는 주의해야 합니다. 그러나 신중한 경우 주요 충돌을 사용하여 애플리케이션을 향상시킬 수 있습니다.

키 충돌 방지

키/값 쌍에 대한 고유 키를 생성하는 명명 규칙을 채택하여 키 충돌을 방지합니다. 예를 들어 명명 규칙은 애플리케이션의 마침표로 구분된 이름, 쌍에 대한 추가 정보를 제공하는 메서드 및 고유 식별자로 구성된 키를 생성할 수 있습니다.

제품 및 공급업체라는 두 애플리케이션에 각각 Sales라는 메서드가 있다고 가정합니다. Products 애플리케이션의 Sales 메서드는 제품의 ID 번호(재고 유지 단위 또는 SKU)를 제공합니다. Suppliers 애플리케이션의 Sales 메서드는 공급업체의 ID 번호 또는 SID를 제공합니다. 따라서 이 예제의 명명 규칙은 "Products.Sales.SKU" 및 "Suppliers.Sales.SID" 키를 생성합니다.

주요 충돌 활용

하나 이상의 특수한 미리 정렬된 키를 사용하여 처리를 제어하여 키 충돌을 악용합니다. 한 시나리오에서 호출 스택 계층 구조의 가장 높은 수준의 예외 처리기가 하위 수준 예외 처리기에서 발생시킨 모든 예외를 잡는다고 가정합니다. 특수 키가 있는 키/값 쌍이 있는 경우 상위 수준 예외 처리기는 개체의 나머지 키/값 쌍 IDictionary 의 형식을 비표준 방식으로 지정하고, 그렇지 않으면 나머지 키/값 쌍의 형식이 일반적인 방식으로 지정됩니다.

이제 다른 시나리오에서는 호출 스택 계층 구조 내의 각 수준에 있는 예외 처리기가 다음 하위 수준의 예외 처리기에 의해 발생된 예외를 catch한다고 가정합니다. 또한 각 예외 처리기는 속성에서 반환된 Data 컬렉션에 미리 정렬된 키 집합으로 액세스할 수 있는 키/값 쌍 집합이 포함되어 있음을 알고 있습니다.

각 예외 처리기는 미리 구성된 키 집합을 사용하여 해당 키/값 쌍의 값 구성 요소를 해당 예외 처리기에 고유한 정보로 업데이트합니다. 업데이트 프로세스가 완료되면 예외 처리기가 다음 상위 수준 예외 처리기에 예외를 throw합니다. 마지막으로, 가장 높은 수준의 예외 처리기는 키/값 쌍에 액세스하고 모든 하위 수준 예외 처리기의 통합 업데이트 정보를 표시합니다.

System.Exception.Message 속성

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

오류 메시지는 예외를 처리하는 개발자를 대상으로 합니다. 속성의 Message 텍스트는 오류를 완전히 설명해야 하며 가능한 경우 오류를 수정하는 방법도 설명해야 합니다. 최상위 예외 처리기는 최종 사용자에게 메시지를 표시할 수 있으므로 문법적으로 올바르고 메시지의 각 문장이 마침표로 끝나는지 확인해야 합니다. 물음표나 느낌표를 사용하지 마세요. 애플리케이션에서 지역화된 예외 메시지를 사용하는 경우 정확하게 번역되었는지 확인해야 합니다.

> ⓘ **Important**
>
> 적절한 권한을 확인하지 않고 예외 메시지에 중요한 정보를 공개하지 마세요.

Message가 반환하는 정보에 ToString 속성 값이 포함됩니다. Message 속성은 Exception을 생성할 때만 설정됩니다. 현재 인스턴스에 대한 생성자에 메시지가 제공되지 않은 경우 시스템은 현재 시스템 문화권을 사용하여 형식이 지정된 기본 메시지를 제공합니다.

예시

다음 코드 예제는 Exception 예외를 throw한 후 catch하고, Message 속성을 사용하여 예외의 텍스트 메시지를 표시합니다.

C#

```
using System;

namespace NDP_UE_CS
{
    // Derive an exception; the constructor sets the HelpLink and
    // Source properties.
    class LogTableOverflowException : Exception
    {
        const string overflowMessage = "The log table has overflowed.";

        public LogTableOverflowException(
            string auxMessage, Exception inner) :
                base(String.Format("{0} - {1}",
                    overflowMessage, auxMessage), inner)
        {
            this.HelpLink = "https://learn.microsoft.com";
            this.Source = "Exception_Class_Samples";
```

```
    }
}

class LogTable
{
    public LogTable(int numElements)
    {
        logArea = new string[numElements];
        elemInUse = 0;
    }

    protected string[] logArea;
    protected int elemInUse;

    // The AddRecord method throws a derived exception if
    // the array bounds exception is caught.
    public int AddRecord(string newRecord)
    {
        try
        {
            logArea[elemInUse] = newRecord;
            return elemInUse++;
        }
        catch (Exception e)
        {
            throw new LogTableOverflowException(
                String.Format("Record \"{0}\" was not logged.",
                    newRecord), e);
        }
    }
}

class OverflowDemo
{
    // Create a log table and force an overflow.
    public static void Main()
    {
        LogTable log = new LogTable(4);

        Console.WriteLine(
            "This example of \n   Exception.Message, \n" +
            "   Exception.HelpLink, \n   Exception.Source, \n" +
            "   Exception.StackTrace, and \n   Exception." +
            "TargetSite \ngenerates the following output.");

        try
        {
            for (int count = 1; ; count++)
            {
                log.AddRecord(
                    String.Format(
                        "Log record number {0}", count));
            }
        }
        catch (Exception ex)
```

```
            {
                Console.WriteLine($"{Environment.NewLine}Message ---
{Environment.NewLine}{ex.Message}");
                Console.WriteLine($"{Environment.NewLine}HelpLink ---
{Environment.NewLine}{ex.HelpLink}");
                Console.WriteLine($"{Environment.NewLine}Source ---
{Environment.NewLine}{ex.Source}");
                Console.WriteLine($"{Environment.NewLine}StackTrace ---
{Environment.NewLine}{ex.StackTrace}");
                Console.WriteLine($"{Environment.NewLine}TargetSite ---
{Environment.NewLine}{ex.TargetSite}");
            }
        }
    }
}

/*
This example of
   Exception.Message,
   Exception.HelpLink,
   Exception.Source,
   Exception.StackTrace, and
   Exception.TargetSite
generates the following output.

Message ---
The log table has overflowed. - Record "Log record number 5" was not logged.

HelpLink ---
https://learn.microsoft.com

Source ---
Exception_Class_Samples

StackTrace ---
   at NDP_UE_CS.LogTable.AddRecord(String newRecord)
   at NDP_UE_CS.OverflowDemo.Main()

TargetSite ---
Int32 AddRecord(System.String)
*/
```

Last updated on 2026. 02. 12.

System.InvalidCastException 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

.NET은 파생 형식에서 기본 형식으로의 자동 변환과, 그리고 원래의 파생 형식으로 다시 변환을 지원합니다. 또한, 인터페이스를 제공하는 형식에서 인터페이스 개체로의 변환 및 그 반대 방향으로의 자동 변환도 지원합니다. 또한 사용자 지정 변환을 지원하는 다양한 메커니즘을 포함합니다. 자세한 내용은 .NET의 형식 변환을 참조하세요.

InvalidCastException 한 형식의 인스턴스를 다른 형식으로 변환할 수 없는 경우 예외가 throw됩니다. 예를 들어 Char 값을 DateTime 값으로 변환하려고 시도하면 InvalidCastException 예외가 발생합니다. 한 형식을 다른 형식으로 변환할 수 있는 경우에도, 원본 형식의 값이 대상 형식의 범위를 초과할 때 발생하는 예외와는 다릅니다 OverflowException. InvalidCastException 예외는 개발자 오류로 인해 발생하며 블록에서 `try/catch` 처리해서는 안 됩니다. 대신 예외의 원인을 제거해야 합니다.

시스템에서 지원하는 변환에 대한 자세한 내용은 클래스를 Convert 참조하세요. 대상 형식이 원본 형식 값을 저장할 수 있지만 특정 원본 값을 저장할 만큼 크지 않을 때 발생하는 오류는 예외를 OverflowException 참조하세요.

> **참고**
>
> 대부분의 경우 언어 컴파일러는 원본 형식과 대상 형식 간에 변환이 없음을 감지하고 컴파일러 오류를 발생합니다.

변환 시도가 예외 InvalidCastException를 발생시키는 조건 중 일부는 다음 섹션에서 설명합니다.

명시적 참조 변환에 성공하려면 원본 값이어야 `null` 합니다. 또는 소스 인수에서 참조하는 개체 형식을 암시적 참조 변환을 통해 대상 형식으로 변환할 수 있어야 합니다.

다음 중간 언어(IL) 명령은 InvalidCastException 예외를 throw합니다.

- `castclass`
- `refanyval`
- `unbox`

InvalidCastException 는 값이 0x80004002 HRESULT `COR_E_INVALIDCAST` 를 사용합니다.

InvalidCastException 인스턴스의 초기 속성 값 목록은 InvalidCastException 생성자를 참조하세요.

기본 형식 및 IConvertible

특정 변환을 지원하지 않는 기본 형식의 IConvertible 구현을 직접 또는 간접적으로 호출합니다. 예를 들어, Boolean 값을 Char로 변환하거나, DateTime 값을 Int32로 변환하려고 하면 InvalidCastException 예외가 발생합니다. 다음 예제에서는 Boolean.IConvertible.ToChar 값을 Convert.ToChar(Boolean)로 변환하기 위해 Boolean 메서드와 Char 메서드를 모두 호출합니다. 두 경우 모두 메서드 호출이 InvalidCastException 예외를 발생시킵니다.

C#

```
using System;

public class IConvertibleEx
{
    public static void Main()
    {
        bool flag = true;
        try
        {
            IConvertible conv = flag;
            Char ch = conv.ToChar(null);
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException)
        {
            Console.WriteLine("Cannot convert a Boolean to a Char.");
        }

        try
        {
            Char ch = Convert.ToChar(flag);
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException)
        {
            Console.WriteLine("Cannot convert a Boolean to a Char.");
        }
    }
}
// The example displays the following output:
//       Cannot convert a Boolean to a Char.
//       Cannot convert a Boolean to a Char.
```

변환이 지원되지 않으므로 해결 방법이 없습니다.

Convert.ChangeType 메서드

메서드를 Convert.ChangeType 호출하여 개체를 한 형식에서 다른 형식으로 변환했지만 하나 또는 두 형식 모두 인터페이스를 IConvertible 구현하지 않습니다.

대부분의 경우 변환이 지원되지 않으므로 해결 방법이 없습니다. 경우에 따라 가능한 해결 방법은 원본 형식의 속성 값을 대상 형식의 유사한 속성에 수동으로 할당하는 것입니다.

축소 변환 및 IConvertible 구현

축소 연산자는 형식에서 지원하는 명시적 변환을 정의합니다. 변환을 수행하려면 C#의 캐스팅 연산자 또는 `CType` Visual Basic의 변환 메서드(있는 경우 `Option Strict`)가 필요합니다.

그러나 원본 형식이나 대상 형식이 두 형식 간의 명시적 또는 축소 변환을 정의하지 않고 하나 또는 두 형식 IConvertible 의 구현이 원본 형식에서 대상 형식 InvalidCastException 으로의 변환을 지원하지 않는 경우 예외가 throw됩니다.

대부분의 경우 변환이 지원되지 않으므로 해결 방법이 없습니다.

다운캐스팅

다운캐스트는 기본 형식의 인스턴스를 파생 형식 중 하나로 변환하려고 하는 경우를 말합니다. 다음 예제에서는 `Person` 개체를 `PersonWithID` 개체로 변환하려고 하면 실패합니다.

C#

```
using System;

public class Person
{
   String _name;

   public String Name
   {
      get { return _name; }
      set { _name = value; }
   }
}

public class PersonWithId : Person
{
   String _id;

   public string Id
   {
      get { return _id; }
      set { _id = value; }
   }
}
```

```
public class Example
{
   public static void Main()
   {
      Person p = new Person();
      p.Name = "John";
      try {
         PersonWithId pid = (PersonWithId) p;
         Console.WriteLine("Conversion succeeded.");
      }
      catch (InvalidCastException) {
         Console.WriteLine("Conversion failed.");
      }

      PersonWithId pid1 = new PersonWithId();
      pid1.Name = "John";
      pid1.Id = "246";
      Person p1 = pid1;
      try {
         PersonWithId pid1a = (PersonWithId) p1;
         Console.WriteLine("Conversion succeeded.");
      }
      catch (InvalidCastException) {
         Console.WriteLine("Conversion failed.");
      }

      Person p2 = null;
      try {
         PersonWithId pid2 = (PersonWithId) p2;
         Console.WriteLine("Conversion succeeded.");
      }
      catch (InvalidCastException) {
         Console.WriteLine("Conversion failed.");
      }
   }
}
// The example displays the following output:
//       Conversion failed.
//       Conversion succeeded.
//       Conversion succeeded.
```

예제에서 알 수 있듯이, 다운캐스트는 `Person` 개체가 `PersonWithId` 개체에서 `Person` 개체로 업캐스트되는 과정에서 생성된 경우, 또는 `Person` 개체가 `null`인 경우에만 성공합니다.

인터페이스 객체에서 변환

인터페이스 개체를 해당 인터페이스를 구현하는 형식으로 변환하려고 하지만 대상 형식은 인터페이스 개체가 원래 파생된 형식의 동일한 형식 또는 기본 클래스가 아닙니다. 다음 예제에서는 InvalidCastException 개체를 IFormatProvider 개체로 변환하려고 할 때 DateTimeFormatInfo 예

외를 던집니다. 변환이 실패하는 이유는 DateTimeFormatInfo 클래스가 IFormatProvider 인터페이스를 구현했지만, DateTimeFormatInfo 개체가 인터페이스 개체가 파생된 CultureInfo 클래스와 관련이 없기 때문입니다.

C#

```
using System;
using System.Globalization;

public class InterfaceEx
{
    public static void Main()
    {
        var culture = CultureInfo.InvariantCulture;
        IFormatProvider provider = culture;

        DateTimeFormatInfo dt = (DateTimeFormatInfo)provider;
    }
}
// The example displays the following output:
//    Unhandled Exception: System.InvalidCastException:
//       Unable to cast object of type //System.Globalization.CultureInfo// to
//           type //System.Globalization.DateTimeFormatInfo//.
//       at Example.Main()
```

예외 메시지에서 알 수 있듯이 인터페이스 개체가 원래 형식의 인스턴스로 다시 변환되는 경우에만 변환이 성공합니다. 이 경우 .CultureInfo 인터페이스 개체가 원래 형식의 기본 형식 인스턴스로 변환되는 경우에도 변환이 성공합니다.

문자열 변환

C#에서 캐스팅 연산자를 사용하여 값 또는 개체를 문자열 표현으로 변환하려고 합니다. 다음 예제에서는 Char 값을 문자열로 변환하려는 시도와 정수를 문자열로 변환하려는 시도가 모두 InvalidCastException 예외를 발생시킵니다.

C#

```
public class StringEx
{
    public static void Main()
    {
        object value = 12;
        // Cast throws an InvalidCastException exception.
        string s = (string)value;
    }
}
```

> ⓘ **참고**
>
> Visual Basic `CStr` 연산자를 사용하여 기본 형식의 값을 문자열로 변환하는 데 성공합니다. 이 작업은 InvalidCastException 예외를 발생시키지 않습니다.

모든 형식의 인스턴스를 문자열 표현으로 성공적으로 변환하려면 다음 예제와 같이 해당 `ToString` 메서드를 호출합니다. `ToString` 메서드는 항상 존재합니다. 이는 ToString 메서드가 Object 클래스에 의해 정의되기 때문에, 모든 관리되는 형식에서 상속되거나 재정의됩니다.

C#

```
using System;

public class ToStringEx2
{
    public static void Main()
    {
        object value = 12;
        string s = value.ToString();
        Console.WriteLine(s);
    }
}
// The example displays the following output:
//       12
```

Visual Basic 6.0 마이그레이션

사용자 정의 이벤트를 사용자 정의 컨트롤에서 Visual Basic .NET으로 호출하여 Visual Basic 6.0 애플리케이션을 업그레이드하고 있으며 InvalidCastException " "지정된 캐스트가 유효하지 않습니다."라는 메시지와 함께 예외가 throw됩니다. 이 예외를 제거하려면 양식의 코드 줄을 변경합니다(예: `Form1`).

VB

```
Call UserControl11_MyCustomEvent(UserControl11, New
UserControl1.MyCustomEventEventArgs(5))
```

다음 코드 줄로 바꿉다.

VB

```
Call UserControl11_MyCustomEvent(UserControl11(0), New
UserControl1.MyCustomEventEventArgs(5))
```

System.InvalidOperationException 클래스

아티클 • 2025. 03. 25.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

InvalidOperationException 잘못된 인수 이외의 이유로 인해 메서드를 호출하지 못한 경우에 사용됩니다. 일반적으로, 개체의 상태가 메서드 호출을 지원할 수 없을 때 발생합니다. 예를 들어, InvalidOperationException 예외는 다음과 같은 메서드에 의해 throw됩니다.

- IEnumerator.MoveNext 열거자를 만든 후 컬렉션의 개체가 수정되는 경우입니다. 자세한 내용은 반복하는 동안 컬렉션 변경참조하세요.
- 메서드를 호출하기 전에 리소스 집합이 닫혀 있는지 ResourceSet.GetString.
- XContainer.Add추가할 개체가 잘못 구조화된 XML 문서를 생성할 경우
- 기본 또는 UI 스레드가 아닌 스레드에서 UI를 조작하려고 시도하는 메서드입니다.

> ⓘ **중요**
>
> InvalidOperationException 예외는 다양한 상황에서 throw될 수 있으므로 Message 속성에서 반환된 예외 메시지를 읽는 것이 중요합니다.

InvalidOperationException는 값이 0x80131509인 HRESULT `COR_E_INVALIDOPERATION`을 사용합니다.

InvalidOperationException인스턴스의 초기 속성 값 목록은 InvalidOperationException 생성자를 참조하세요.

InvalidOperationException 예외의 일반적인 원인

다음 섹션에서는 앱에서 InvalidOperationException 예외가 발생하는 몇 가지 일반적인 사례와 그 원인을 설명합니다. 문제를 처리하는 방법은 특정 상황에 따라 달라집니다. 그러나 가장 일반적으로 예외는 개발자 오류로 인해 발생하며 InvalidOperationException 예외를 예상 및 방지할 수 있습니다.

비 UI 스레드에서 UI 스레드 업데이트

종종 작업자 스레드는 애플리케이션의 사용자 인터페이스에 표시할 데이터 수집을 포함하는 일부 백그라운드 작업을 수행하는 데 사용됩니다. 하지만. Windows Forms 및 WPF(Windows Presentation Foundation)와 같은 .NET용 대부분의 GUI(그래픽 사용자 인터페이스) 애플리케이션 프레임워크를 사용하면 UI(기본 또는 UI 스레드)를 만들고 관리하는 스레드에서만 GUI 개체에 액세스할 수 있습니다. UI 스레드가 아닌 다른 스레드에서 UI 요소에 액세스하려고 하면 오류 InvalidOperationException이(가) 던져집니다. 예외 메시지의 텍스트는 다음 표에 나와 있습니다.

테이블 확장

애플리케이션 유형	메시지
WPF 앱	**호출 스레드는 다른 스레드가 소유하므로 이 개체에 액세스할 수 없습니다.**
UWP 앱	**애플리케이션은 다른 스레드에 대해 마샬링된 인터페이스를 호출했습니다.**
Windows Forms 앱	**스레드 간 작업이 유효하지 않습니다: 'TextBox1' 제어가 생성된 스레드가 아닌 다른 스레드에서 액세스되었습니다.**

.NET용 UI 프레임워크는 UI 요소의 멤버에 대한 호출이 UI 스레드에서 실행되고 있는지 여부를 확인하는 메서드와 UI 스레드에서 호출을 예약하는 다른 메서드를 포함하는 *디스패처* 패턴을 구현합니다.

- WPF 앱에서 Dispatcher.CheckAccess 메서드를 호출하여 메서드가 UI가 아닌 스레드에서 실행 중인지 확인합니다. 메서드가 UI 스레드에서 실행 중이면 `true` 반환하고, 그렇지 않으면 `false`. Dispatcher.Invoke 메서드의 오버로드 중 하나를 호출하여 UI 스레드에서 호출을 예약합니다.
- UWP 앱에서 CoreDispatcher.HasThreadAccess 속성을 확인하여 메서드가 UI가 아닌 스레드에서 실행 중인지 확인합니다. CoreDispatcher.RunAsync 메서드를 호출하여 UI 스레드를 업데이트하는 대리자를 실행합니다.
- Windows Forms 앱에서 Control.InvokeRequired 속성을 사용하여 메서드가 UI가 아닌 스레드에서 실행 중인지 확인합니다. Control.Invoke 메서드의 오버로드 중 하나를 호출하여 UI 스레드를 업데이트하는 대리자를 실행합니다.

다음 예제에서는 UI 요소를 만든 스레드가 아닌 스레드에서 UI 요소를 업데이트하려고 할 때 throw되는 InvalidOperationException 예외를 보여 줍니다. 각 예제에서는 두 개의 컨트롤을 만들어야 합니다.

- 이름이 `textBox1`텍스트 상자 컨트롤입니다. Windows Forms 앱에서는 Multiline 속성을 `true`로 설정해야 합니다.
- 이름이 `threadExampleBtn`인 버튼 컨트롤입니다. 이 예제에서는 단추의 `Click` 이벤트에 대한 처리기 `ThreadsExampleBtn_Click`제공합니다.

각 경우에서 `threadExampleBtn_Click` 이벤트 처리기는 `DoSomeWork` 메서드를 두 번 호출합니다. 첫 번째 호출은 동기적으로 실행되고 성공합니다. 그러나 두 번째 호출은 스레드 풀 스레드에서 비동기적으로 실행되므로 UI가 아닌 스레드에서 UI를 업데이트하려고 시도합니다. 이로 인해 InvalidOperationException 예외가 발생합니다.

WPF 앱

C#

```
private async void threadExampleBtn_Click(object sender, RoutedEventArgs e)
{
    textBox1.Text = String.Empty;

    textBox1.Text = "Simulating work on UI thread.\n";
    DoSomeWork(20);
    textBox1.Text += "Work completed...\n";

    textBox1.Text += "Simulating work on non-UI thread.\n";
    await Task.Run(() => DoSomeWork(1000));
    textBox1.Text += "Work completed...\n";
}

private async void DoSomeWork(int milliseconds)
{
    // Simulate work.
    await Task.Delay(milliseconds);

    // Report completion.
    var msg = String.Format("Some work completed in {0} ms.\n",
milliseconds);
    textBox1.Text += msg;
}
```

다음 버전의 `DoSomeWork` 메서드는 WPF 앱에서 예외를 제거합니다.

C#

```
private async void DoSomeWork(int milliseconds)
{
    // Simulate work.
    await Task.Delay(milliseconds);

    // Report completion.
    bool uiAccess = textBox1.Dispatcher.CheckAccess();
    String msg = String.Format("Some work completed in {0} ms. on {1}UI
thread\n",
                               milliseconds, uiAccess ? String.Empty : "non-
");
    if (uiAccess)
        textBox1.Text += msg;
```

```
    else
        textBox1.Dispatcher.Invoke(() => { textBox1.Text += msg; });
}
```

Windows Forms 앱

C#

```
List<String> lines = new List<String>();

private async void threadExampleBtn_Click(object sender, EventArgs e)
{
    textBox1.Text = String.Empty;
    lines.Clear();

    lines.Add("Simulating work on UI thread.");
    textBox1.Lines = lines.ToArray();
    DoSomeWork(20);

    lines.Add("Simulating work on non-UI thread.");
    textBox1.Lines = lines.ToArray();
    await Task.Run(() => DoSomeWork(1000));

    lines.Add("ThreadsExampleBtn_Click completes. ");
    textBox1.Lines = lines.ToArray();
}

private async void DoSomeWork(int milliseconds)
{
    // simulate work
    await Task.Delay(milliseconds);

    // report completion
    lines.Add(String.Format("Some work completed in {0} ms on UI thread.",
milliseconds));
    textBox1.Lines = lines.ToArray();
}
```

다음 버전의 `DoSomeWork` 메서드는 Windows Forms 앱에서 예외를 제거합니다.

C#

```
private async void DoSomeWork(int milliseconds)
{
    // simulate work
    await Task.Delay(milliseconds);

    // Report completion.
    bool uiMarshal = textBox1.InvokeRequired;
    String msg = String.Format("Some work completed in {0} ms. on {1}UI
```

```
thread\n",
                                milliseconds, uiMarshal ? String.Empty :
"non-");
    lines.Add(msg);

    if (uiMarshal) {
        textBox1.Invoke(new Action(() => { textBox1.Lines = lines.ToArray();
}));
    }
    else {
        textBox1.Lines = lines.ToArray();
    }
}
```

컬렉션을 반복하면서 변경하기

C#의 `foreach` 문, F#의 `for...in` 또는 Visual Basic의 `For Each` 문은 컬렉션의 멤버를 반복하고 개별 요소를 읽거나 수정하는 데 사용됩니다. 그러나 컬렉션에서 항목을 추가하거나 제거하는 데는 사용할 수 없습니다. 이렇게 하면 예외 InvalidOperationException이 발생하고, 메시지는 "**컬렉션이 수정되었습니다; 열거형 작업은 실행되지 않을 수 있습니다.**"와 비슷합니다.

다음 예제에서는 컬렉션에 각 정수의 제곱을 추가하려고 시도하는 정수 컬렉션을 반복합니다. 예제는 List<T>.Add 메서드에 대한 첫 번째 호출 시 InvalidOperationException을 throw합니다.

C#

```
using System;
using System.Collections.Generic;

public class IteratingEx1
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };
        foreach (var number in numbers)
        {
            int square = (int)Math.Pow(number, 2);
            Console.WriteLine($"{number}^{square}");
            Console.WriteLine($"Adding {square} to the collection...");
            Console.WriteLine();
            numbers.Add(square);
        }
    }
}
// The example displays the following output:
//    1^1
//    Adding 1 to the collection...
//
```

```
//
//    Unhandled Exception: System.InvalidOperationException: Collection was
modified;
//       enumeration operation may not execute.
//       at
System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource
resource)
//       at System.Collections.Generic.List`1.Enumerator.MoveNextRare()
//       at Example.Main()
```

애플리케이션 논리에 따라 두 가지 방법 중 하나로 예외를 제거할 수 있습니다.

- 요소를 반복하는 동안 컬렉션에 추가해야 하는 경우 `foreach`, `for...in` 또는 `For Each` 대신 `for` (F#의 `for..to`) 문을 사용하여 인덱스로 반복할 수 있습니다. 다음 예제에서는 for 문을 사용하여 컬렉션의 숫자 제곱을 컬렉션에 추가합니다.

  C#

```
using System;
using System.Collections.Generic;

public class IteratingEx2
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };

        int upperBound = numbers.Count - 1;
        for (int ctr = 0; ctr <= upperBound; ctr++)
        {
            int square = (int)Math.Pow(numbers[ctr], 2);
            Console.WriteLine($"{numbers[ctr]}^{square}");
            Console.WriteLine($"Adding {square} to the collection...");
            Console.WriteLine();
            numbers.Add(square);
        }

        Console.WriteLine("Elements now in the collection: ");
        foreach (var number in numbers)
            Console.Write("{0}    ", number);
    }
}
// The example displays the following output:
//    1^1
//    Adding 1 to the collection...
//
//    2^4
//    Adding 4 to the collection...
//
//    3^9
//    Adding 9 to the collection...
//
```

```
//    4^16
//    Adding 16 to the collection...
//
//    5^25
//    Adding 25 to the collection...
//
//    Elements now in the collection:
//    1    2    3    4    5    1    4    9    16    25
```

루프 내에서 루프를 적절하게 종료하는 카운터를 사용하거나, 뒤로 반복하거나, `Count` - 1에서 0으로 반복하거나, 예제와 같이 배열의 요소 수를 변수에 할당하고 이를 사용하여 루프의 상한을 설정하여 컬렉션을 반복하기 전에 반복 수를 설정해야 합니다. 그렇지 않으면 모든 반복에서 요소가 컬렉션에 추가되면 무한 루프가 발생합니다.

- 반복하는 동안 컬렉션에 요소를 추가할 필요가 없는 경우 컬렉션을 반복할 때 추가할 임시 컬렉션에 추가할 요소를 저장할 수 있습니다. 다음 예제에서는 이 방법을 사용하여 컬렉션의 숫자 제곱을 임시 컬렉션에 추가한 다음 컬렉션을 단일 배열 개체로 결합합니다.

C#

```
using System;
using System.Collections.Generic;

public class IteratingEx3
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };
        var temp = new List<int>();

        // Square each number and store it in a temporary collection.
        foreach (var number in numbers)
        {
            int square = (int)Math.Pow(number, 2);
            temp.Add(square);
        }

        // Combine the numbers into a single array.
        int[] combined = new int[numbers.Count + temp.Count];
        Array.Copy(numbers.ToArray(), 0, combined, 0, numbers.Count);
        Array.Copy(temp.ToArray(), 0, combined, numbers.Count,
temp.Count);

        // Iterate the array.
        foreach (var value in combined)
            Console.Write("{0}    ", value);
    }
}
```

```
// The example displays the following output:
//       1    2    3    4    5    1    4    9    16    25
```

개체를 비교할 수 없는 배열 또는 컬렉션 정렬

Array.Sort(Array) 메서드 또는 List<T>.Sort() 메서드와 같은 범용 정렬 메서드는 일반적으로 정렬할 개체 중 하나 이상이 IComparable<T> 또는 IComparable 인터페이스를 구현해야 합니다. 그렇지 않은 경우 컬렉션 또는 배열을 정렬할 수 없으며 메서드는 InvalidOperationException 예외를 throw합니다. 다음 예제에서는 `Person` 클래스를 정의하고, 제네릭 List<T> 개체에 두 개의 `Person` 개체를 저장하고, 정렬을 시도합니다. 예제의 출력 결과에서 보이듯이, List<T>.Sort() 메서드를 호출하면 InvalidOperationException 오류가 발생합니다.

C#

```
using System;
using System.Collections.Generic;

public class Person1
{
    public Person1(string fName, string lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class ListSortEx1
{
    public static void Main()
    {
        var people = new List<Person1>();

        people.Add(new Person1("John", "Doe"));
        people.Add(new Person1("Jane", "Doe"));
        people.Sort();
        foreach (var person in people)
            Console.WriteLine($"{person.FirstName} {person.LastName}");
    }
}
// The example displays the following output:
//    Unhandled Exception: System.InvalidOperationException: Failed to
compare two elements in the array. --->
//       System.ArgumentException: At least one object must implement
IComparable.
//       at System.Collections.Comparer.Compare(Object a, Object b)
//       at System.Collections.Generic.ArraySortHelper`1.SwapIfGreater(T[]
```

```
keys, IComparer`1 comparer, Int32 a, Int32 b)
//       at
System.Collections.Generic.ArraySortHelper`1.DepthLimitedQuickSort(T[] keys,
Int32 left, Int32 right, IComparer`1 comparer, Int32 depthLimit)
//       at System.Collections.Generic.ArraySortHelper`1.Sort(T[] keys,
Int32 index, Int32 length, IComparer`1 comparer)
//       --- End of inner exception stack trace ---
//       at System.Collections.Generic.ArraySortHelper`1.Sort(T[] keys,
Int32 index, Int32 length, IComparer`1 comparer)
//       at System.Array.Sort[T](T[] array, Int32 index, Int32 length,
IComparer`1 comparer)
//       at System.Collections.Generic.List`1.Sort(Int32 index, Int32 count,
IComparer`1 comparer)
//       at Example.Main()
```

다음 세 가지 방법 중에서 예외를 제거할 수 있습니다.

- 정렬하려는 형식을 소유할 수 있는 경우(즉, 소스 코드를 제어하는 경우) IComparable<T> 또는 IComparable 인터페이스를 구현하도록 수정할 수 있습니다. 이렇게 하려면 IComparable<T>.CompareTo 또는 CompareTo 메서드를 구현해야 합니다. 기존 형식에 인터페이스 구현을 추가하는 것은 호환성이 손상되는 변경이 아닙니다.

  다음 예제에서는 이 방법을 사용하여 `Person` 클래스에 대한 IComparable<T> 구현을 제공합니다. 컬렉션 또는 배열의 일반 정렬 메서드를 계속 호출할 수 있으며 예제의 출력에서 볼 수 있듯이 컬렉션이 성공적으로 정렬됩니다.

  C#

```
using System;
using System.Collections.Generic;

public class Person2 : IComparable<Person>
{
    public Person2(String fName, String lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public String FirstName { get; set; }
    public String LastName { get; set; }

    public int CompareTo(Person other)
    {
        return String.Format("{0} {1}", LastName, FirstName).
               CompareTo(String.Format("{0} {1}", other.LastName,
other.FirstName));
    }
}
```

```
public class ListSortEx2
{
    public static void Main()
    {
        var people = new List<Person2>();

        people.Add(new Person2("John", "Doe"));
        people.Add(new Person2("Jane", "Doe"));
        people.Sort();
        foreach (var person in people)
            Console.WriteLine($"{person.FirstName} {person.LastName}");
    }
}
// The example displays the following output:
//       Jane Doe
//       John Doe
```

- 정렬하려는 형식의 소스 코드를 수정할 수 없는 경우 IComparer<T> 인터페이스를 구현하는 특수 용도의 정렬 클래스를 정의할 수 있습니다. IComparer<T> 매개 변수를 포함하는 `Sort` 메서드의 오버로드를 호출할 수 있습니다. 이 방법은 여러 조건에 따라 개체를 정렬할 수 있는 특수 정렬 클래스를 개발하려는 경우에 특히 유용합니다.

  다음 예제에서는 `Person` 컬렉션을 정렬하는 데 사용되는 사용자 지정 `PersonComparer` 클래스를 개발하여 이 방법을 사용합니다. 그런 다음 이 클래스의 인스턴스를 List<T>.Sort(IComparer<T>) 메서드에 전달합니다.

C#

```
using System;
using System.Collections.Generic;

public class Person3
{
    public Person3(String fName, String lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public String FirstName { get; set; }
    public String LastName { get; set; }
}

public class PersonComparer : IComparer<Person3>
{
    public int Compare(Person3 x, Person3 y)
    {
        return String.Format("{0} {1}", x.LastName, x.FirstName).
               CompareTo(String.Format("{0} {1}", y.LastName,
y.FirstName));
```

```
    }
}

public class ListSortEx3
{
    public static void Main()
    {
        var people = new List<Person3>();

        people.Add(new Person3("John", "Doe"));
        people.Add(new Person3("Jane", "Doe"));
        people.Sort(new PersonComparer());
        foreach (var person in people)
            Console.WriteLine($"{person.FirstName} {person.LastName}");
    }
}
// The example displays the following output:
//       Jane Doe
//       John Doe
```

- 정렬하려는 형식의 소스 코드를 수정할 수 없는 경우 정렬을 수행할 Comparison<T> 대리자를 만들 수 있습니다. 대리자 서명은

```
C#

int Comparison<T>(T x, T y)
```

다음 예제에서는 Comparison<T> 대리자 서명과 일치하는 `PersonComparison` 메서드를 정의하여 이 방법을 사용합니다. 그런 다음 이 대리자를 List<T>.Sort(Comparison<T>) 메서드에 전달합니다.

```
C#

using System;
using System.Collections.Generic;

public class Person
{
   public Person(String fName, String lName)
   {
      FirstName = fName;
      LastName = lName;
   }

   public String FirstName { get; set; }
   public String LastName { get; set; }
}

public class ListSortEx4
{
   public static void Main()
```

```
    {
       var people = new List<Person>();

       people.Add(new Person("John", "Doe"));
       people.Add(new Person("Jane", "Doe"));
       people.Sort(PersonComparison);
       foreach (var person in people)
          Console.WriteLine($"{person.FirstName} {person.LastName}");
    }

    public static int PersonComparison(Person x, Person y)
    {
       return String.Format("{0} {1}", x.LastName, x.FirstName).
              CompareTo(String.Format("{0} {1}", y.LastName,
 y.FirstName));
    }
}
// The example displays the following output:
//       Jane Doe
//       John Doe
```

Nullable<T>가 null일 때 이를 기본 형식으로 캐스팅

기본 형식에 캐스팅되는 `null`Nullable<T> 값을 시도하면 InvalidOperationException 예외가 발생하고 오류 메시지로 "**널 허용 개체는 값을 가져야 합니다.**"가 표시됩니다.

다음 예제에서는 `Nullable(Of Integer)` 값을 포함하는 배열을 반복하려고 할 때 InvalidOperationException 예외를 throw합니다.

C#

```
using System;
using System.Linq;

public class NullableEx1
{
   public static void Main()
   {
      var queryResult = new int?[] { 1, 2, null, 4 };
      var map = queryResult.Select(nullableInt => (int)nullableInt);

      // Display list.
      foreach (var num in map)
         Console.Write("{0} ", num);
      Console.WriteLine();
   }
}
// The example displays the following output:
//    1 2
//    Unhandled Exception: System.InvalidOperationException: Nullable object
must have a value.
```

```
//       at
System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource
resource)
//       at Example.<Main>b__0(Nullable`1 nullableInt)
//       at System.Linq.Enumerable.WhereSelectArrayIterator`2.MoveNext()
//       at Example.Main()
```

예외를 방지하려면 다음을 수행합니다.

- Nullable<T>.HasValue 속성을 사용하여 `null`않은 요소만 선택합니다.
- Nullable<T>.GetValueOrDefault 오버로드 중 하나를 호출하여 `null` 값의 기본값을 제공합니다.

다음 예제에서는 둘 다 InvalidOperationException 예외를 방지합니다.

C#

```
using System;
using System.Linq;

public class NullableEx2
{
   public static void Main()
   {
      var queryResult = new int?[] { 1, 2, null, 4 };
      var numbers = queryResult.Select(nullableInt =>
(int)nullableInt.GetValueOrDefault());

      // Display list using Nullable<int>.HasValue.
      foreach (var number in numbers)
         Console.Write("{0} ", number);
      Console.WriteLine();

      numbers = queryResult.Select(nullableInt => (int)
(nullableInt.HasValue ? nullableInt : -1));
      // Display list using Nullable<int>.GetValueOrDefault.
      foreach (var number in numbers)
         Console.Write("{0} ", number);
      Console.WriteLine();
   }
}
// The example displays the following output:
//       1 2 0 4
//       1 2 -1 4
```

빈 컬렉션에서 System.Linq.Enumerable 메서드 호출

Enumerable.Aggregate, Enumerable.Average, Enumerable.First, Enumerable.Last, Enumerable.Max, Enumerable.Min, Enumerable.Single및 Enumerable.SingleOrDefault 메

서드는 시퀀스에 대한 작업을 수행하고 단일 결과를 반환합니다. 이러한 메서드의 일부 오버로드는 시퀀스가 비어 있을 때 InvalidOperationException 예외를 throw하고 다른 오버로드는 `null` 반환합니다. 또한 Enumerable.SingleOrDefault 메서드는 시퀀스에 둘 이상의 요소가 포함된 경우 InvalidOperationException 예외를 throw합니다.

> **참고**
>
> InvalidOperationException 예외를 throw하는 대부분의 메서드는 오버로드입니다. 선택한 오버로드의 동작을 이해해야 합니다.

다음 표에서는 일부 System.Linq.Enumerable 메서드를 호출하여 throw된 InvalidOperationException 예외 개체의 예외 메시지를 나열합니다.

테이블 확장

메서드	메시지

`Aggregate`
`Average`
`Last`
`Max`
`Min`
	시퀀스에 요소가 없습니다.

`First`	**시퀀스에 일치하는 요소** 없음
`Single` `SingleOrDefault`	**Sequence에는 일치하는 요소가 둘 이상**

예외를 제거하거나 처리하는 방법은 애플리케이션의 가정과 호출하는 특정 방법에 따라 달라집니다.

- 빈 시퀀스를 확인하지 않고 이러한 메서드 중 하나를 의도적으로 호출하는 경우 시퀀스가 비어 있지 않고 빈 시퀀스가 예기치 않은 것으로 가정합니다. 이 경우 예외를 포착하거나 다시 던지는 것이 적절합니다.
- 빈 시퀀스를 확인하지 못한 경우 Enumerable.Any 오버로드의 오버로드 중 하나를 호출하여 시퀀스에 요소가 포함되어 있는지 여부를 확인할 수 있습니다.

  > **팁**
  >
  > 시퀀스를 생성하기 전에 Enumerable.Any<TSource> (IEnumerable<TSource>, Func<TSource,Boolean>) 메서드를 호출하면 처리

할 데이터에 많은 요소가 포함되거나 시퀀스를 생성하는 작업이 비용이 많이 드는 경우 성능이 향상될 수 있습니다.

- Enumerable.First, Enumerable.Last또는 Enumerable.Single같은 메서드를 호출한 경우 시퀀스의 멤버 대신 기본값을 반환하는 Enumerable.FirstOrDefault, Enumerable.LastOrDefault또는 Enumerable.SingleOrDefault같은 대체 메서드를 대체할 수 있습니다.

이 예제에서는 추가 세부 정보를 제공합니다.

다음 예제에서는 Enumerable.Average 메서드를 사용하여 값이 4보다 큰 시퀀스의 평균을 계산합니다. 원래 배열의 값이 4를 초과하지 않으므로 시퀀스에 값이 포함되지 않으며 메서드가 InvalidOperationException 예외를 throw합니다.

C#

```
using System;
using System.Linq;

public class Example
{
   public static void Main()
   {
      int[] data = { 1, 2, 3, 4 };
      var average = data.Where(num => num > 4).Average();
      Console.Write("The average of numbers greater than 4 is {0}",
                    average);
   }
}
// The example displays the following output:
//    Unhandled Exception: System.InvalidOperationException: Sequence
contains no elements
//       at System.Linq.Enumerable.Average(IEnumerable`1 source)
//       at Example.Main()
```

다음 예제와 같이 시퀀스를 처리하는 메서드를 호출하기 전에 시퀀스에 요소가 포함되어 있는지 여부를 확인하기 위해 Any 메서드를 호출하여 예외를 제거할 수 있습니다.

C#

```
using System;
using System.Linq;

public class EnumerableEx2
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };
        var moreThan4 = dbQueryResults.Where(num => num > 4);
```

```
        if (moreThan4.Any())
            Console.WriteLine($"Average value of numbers greater than 4:
{moreThan4.Average()}:");
        else
            // handle empty collection
            Console.WriteLine("The dataset has no values greater than 4.");
    }
}
// The example displays the following output:
//       The dataset has no values greater than 4.
```

Enumerable.First 메서드는 시퀀스의 첫 번째 항목 또는 지정된 조건을 충족하는 시퀀스의 첫 번째 요소를 반환합니다. 시퀀스가 비어 있으므로 첫 번째 요소가 없으면 InvalidOperationException 예외가 throw됩니다.

다음 예제에서는 dbQueryResults 배열에 4보다 큰 요소가 포함되지 않으므로 Enumerable.First<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>) 메서드가 InvalidOperationException 예외를 throw합니다.

C#

```
using System;
using System.Linq;

public class EnumerableEx3
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var firstNum = dbQueryResults.First(n => n > 4);

        Console.WriteLine($"The first value greater than 4 is {firstNum}");
    }
}
// The example displays the following output:
//    Unhandled Exception: System.InvalidOperationException:
//       Sequence contains no matching element
//       at System.Linq.Enumerable.First[TSource](IEnumerable`1 source,
Func`2 predicate)
//       at Example.Main()
```

Enumerable.First 대신 Enumerable.FirstOrDefault 메서드를 호출하여 지정된 값이나 기본값을 반환할 수 있습니다. 메서드가 시퀀스에서 첫 번째 요소를 찾지 못하면 해당 데이터 형식의 기본값을 반환합니다. 기본값은 참조 형식에 대한 `null`, 숫자 데이터 형식의 경우 0, DateTime 형식에 대한 DateTime.MinValue.

> ⓘ 참고
>
> Enumerable.FirstOrDefault 메서드에서 반환된 값을 해석하는 것은 종종 형식의 기본값이 시퀀스에서 유효한 값일 수 있기 때문에 복잡합니다. 이 경우 Enumerable.Any 메서드를 호출하여 Enumerable.First 메서드를 호출하기 전에 시퀀스에 유효한 멤버가 있는지 확인합니다.

다음 예제에서는 Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>) 메서드를 호출하여 이전 예제에서 throw된 InvalidOperationException 예외를 방지합니다.

C#

```
using System;
using System.Linq;

public class EnumerableEx4
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var firstNum = dbQueryResults.FirstOrDefault(n => n > 4);

        if (firstNum == 0)
            Console.WriteLine("No value is greater than 4.");
        else
            Console.WriteLine($"The first value greater than 4 is
{firstNum}");
    }
}
// The example displays the following output:
//       No value is greater than 4.
```

하나의 요소 없이 시퀀스에서 Enumerable.Single 또는 Enumerable.SingleOrDefault를 호출합니다.

Enumerable.Single 메서드는 시퀀스의 유일한 요소 또는 지정된 조건을 충족하는 시퀀스의 유일한 요소를 반환합니다. 시퀀스에 요소가 없거나 둘 이상의 요소가 있는 경우 메서드는 InvalidOperationException 예외를 throw합니다.

시퀀스에 요소가 없는 경우 예외를 throw하는 대신 Enumerable.SingleOrDefault 메서드를 사용하여 기본값을 반환할 수 있습니다. 그러나 Enumerable.SingleOrDefault 메서드는 시퀀스에 둘 이상의 요소가 포함된 경우에도 InvalidOperationException 예외를 throw합니다.

다음 표에서는 Enumerable.Single 및 Enumerable.SingleOrDefault 메서드 호출에 의해 throw된 InvalidOperationException 예외 개체의 예외 메시지를 나열합니다.

테이블 확장

메서드	메시지
`Single`	**시퀀스에 일치하는 요소** 없음
`Single` `SingleOrDefault`	**Sequence에는 일치하는 요소가 둘 이상**

다음 예제에서는 시퀀스에 4보다 큰 요소가 없으므로 Enumerable.Single 메서드를 호출하면 InvalidOperationException 예외가 throw됩니다.

C#

```
using System;
using System.Linq;

public class EnumerableEx5
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var singleObject = dbQueryResults.Single(value => value > 4);

        // Display results.
        Console.WriteLine($"{singleObject} is the only value greater than
4");
    }
}
// The example displays the following output:
//    Unhandled Exception: System.InvalidOperationException:
//       Sequence contains no matching element
//       at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source,
Func`2 predicate)
//       at Example.Main()
```

다음 예제에서는 시퀀스가 비어 있을 때 InvalidOperationException 예외가 throw되지 않도록 Enumerable.SingleOrDefault 메서드를 호출합니다. 그러나 이 시퀀스는 값이 2보다 큰 여러 요소를 반환하므로 InvalidOperationException 예외도 throw합니다.

C#

```
using System;
using System.Linq;
```

```
public class EnumerableEx6
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var singleObject = dbQueryResults.SingleOrDefault(value => value >
2);

        if (singleObject != 0)
            Console.WriteLine($"{singleObject} is the only value greater
than 2");
        else
            // Handle an empty collection.
            Console.WriteLine("No value is greater than 2");
    }
}
// The example displays the following output:
//    Unhandled Exception: System.InvalidOperationException:
//       Sequence contains more than one matching element
//       at System.Linq.Enumerable.SingleOrDefault[TSource](IEnumerable`1
source, Func`2 predicate)
//       at Example.Main()
```

Enumerable.Single 메서드를 호출하면 하나의 요소만 포함된 시퀀스이거나, 지정된 조건을 충족하는 시퀀스에 하나의 요소만 포함되어 있다고 가정합니다. Enumerable.SingleOrDefault 0개 또는 1개의 결과가 있는 시퀀스를 가정하지만 더 이상 그렇지 않습니다. 이 가정이 의도적인 가정이고 이러한 조건이 충족되지 않는 경우 결과 InvalidOperationException 다시 throw하거나 잡는 것이 적절합니다. 그렇지 않은 경우 또는 일부 빈도로 잘못된 조건이 발생할 것으로 예상되는 경우 FirstOrDefault 또는 Where 같은 다른 Enumerable 메서드를 사용하는 것이 좋습니다.

동적 애플리케이션 간 도메인 필드 액세스

검색하려는 주소가 있는 필드가 포함된 개체가 코드가 실행되는 애플리케이션 도메인 내에 없는 경우 OpCodes.Ldflda CIL(공용 중간 언어) 명령은 InvalidOperationException 예외를 throw합니다. 필드의 주소는 필드가 있는 애플리케이션 도메인에서만 액세스할 수 있습니다.

InvalidOperationException 예외를 던지다

어떤 이유로 개체의 상태가 특정 메서드 호출을 지원하지 않는 경우에만 InvalidOperationException 예외를 throw해야 합니다. 즉, 메서드 호출은 일부 상황이나 컨텍스트에서 유효하지만 다른 경우에는 유효하지 않습니다.

메서드 호출 실패가 잘못된 인수로 인해 발생하는 경우 ArgumentException 또는 파생 클래스 중 하나(ArgumentNullException 또는 ArgumentOutOfRangeException)를 대신 throw해야 합니다.

System.NotImplementedException 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

특정 메서드, get 접근자 또는 set 접근자가 형식의 멤버로 존재하지만 구현되지 않은 경우, NotImplementedException 예외가 throw됩니다.

NotImplementedException 참조 같음을 지원하는 기본 Object.Equals 구현을 사용합니다. 인스턴스 NotImplementedException의 초기 값 목록은 NotImplementedException 생성자를 참조하십시오.

예외를 던지다

해당 멤버가 아직 개발 중이고 나중에 프로덕션 코드에서만 구현될 때 고유한 형식의 속성 또는 메서드에서 예외를 throw NotImplementedException 하도록 선택할 수 있습니다. 즉, 예외는 NotImplementedException "아직 개발 중"과 동의어여야 합니다.

예외 처리

예외는 NotImplementedException 호출하려는 메서드 또는 속성에 구현이 없으므로 기능이 없음을 나타냅니다. 따라서 `try/catch` 블록에서 이 오류를 처리해서는 안 됩니다. 대신 코드에서 멤버 호출을 제거해야 합니다. 라이브러리의 프로덕션 버전에서 구현될 때 멤버에 대한 호출을 포함할 수 있습니다.

경우에 따라 NotImplementedException 사전 프로덕션 라이브러리에서 아직 개발 중인 기능을 나타내는 데 예외를 사용하지 않을 수 있습니다. 그러나 여전히 기능을 사용할 수 없음을 나타내며 코드에서 멤버 호출을 제거해야 합니다.

NotImplementedException 및 기타 예외 형식

.NET에는 형식의 NotSupportedExceptionPlatformNotSupportedException특정 멤버에 대한 구현이 없음을 나타내는 두 가지 다른 예외 형식도 포함됩니다. 다음 조건에서는 NotImplementedException 예외 대신 이 중 하나를 throw해야 합니다.

- 플랫폼 또는 버전에 따라 일부 멤버는 사용할 수 있지만 다른 멤버는 사용할 수 없는 형식을 설계한 경우, 기능이 지원되지 않는 플랫폼에서 PlatformNotSupportedException 예외를 throw합니다.
- 인터페이스 멤버 구현이나 추상 기본 클래스 메서드 재정의가 불가능한 경우, NotSupportedException 예외를 throw합니다.

예를 들어 Convert.ToInt32(DateTime) 메서드는 날짜와 시간과 32비트 부호 있는 정수 간에 의미 있는 변환이 없기 때문에 NotSupportedException 예외를 throw합니다. 클래스가 인터페이스를 구현하기 때문에 이 경우 메서드가 ConvertIConvertible 있어야 합니다.

추상 기본 클래스를 NotSupportedException 구현한 경우 예외를 throw하고 파생 클래스에서 재정의해야 하는 새 멤버를 추가해야 합니다. 이 경우 멤버를 추상화하면 기존 서브클래스가 로드되지 않습니다.

System.NotSupportedException 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

NotSupportedException 는 호출된 메서드 또는 속성에 대한 구현이 없음을 나타냅니다.

NotSupportedException 는 값이 0x80131515 HRESULT `COR_E_NOTSUPPORTED`를 사용합니다.

NotSupportedException인스턴스의 초기 속성 값 목록은 NotSupportedException 생성자를 참조하세요.

NotSupportedException 예외를 발생시킵니다.

다음과 같은 경우에는 예외를 NotSupportedException 던지는 것을 추천합니다.

- 범용 인터페이스를 구현하고 있으며 메서드 수에는 의미 있는 구현이 없습니다. 예를 들어, IConvertible 인터페이스를 구현하는 날짜 및 시간 형식을 생성하는 경우, 대부분의 변환에 대해 NotSupportedException 예외를 발생시킵니다.
- 당신은 여러 메서드를 재정의해야 하는 추상 클래스에서 상속받았습니다. 그러나 이러한 하위 집합에 대한 구현만 제공할 준비가 된 것입니다. 구현하지 않기로 결정한 메서드의 경우, NotSupportedException을(를) 던지는 것을 선택할 수 있습니다.
- 조건에 따라 작업을 수행할 수 있도록 상태를 가지는 일반적인 형식을 정의하고 있습니다. 예를 들어 형식은 읽기 전용이거나 읽기/쓰기일 수 있습니다. 이 경우 다음을 수행합니다.
  - 개체가 읽기 전용인 경우, 인스턴스의 속성에 값을 할당하려 하거나 인스턴스 상태를 수정하는 메서드를 호출하면 NotSupportedException 예외가 발생해야 합니다.
  - 특정 기능을 사용할 수 있는지 여부를 나타내는 값을 반환 Boolean 하는 속성을 구현해야 합니다. 예를 들어, 읽기 전용이거나 읽기/쓰기가 가능한 형식의 경우 읽기/쓰기 메서드 집합을 사용할 수 있는지 여부를 나타내는 속성 `IsReadOnly`을 구현할 수 있습니다.

NotSupportedException 예외 처리

예외는 NotSupportedException 메서드에 구현이 없으며 메서드를 호출해서는 안 되었음을 나타냅니다. 예외를 처리해서는 안 됩니다. 예외의 원인에 따라서, 즉 구현이 완전히 없거나 아니면 멤버 호출이 개체의 목적과 일치하지 않는 경우(예: 읽기 전용 FileStream.Write 개체에서 FileStream 메서드를 호출하는 경우)에 따라 해야 할 일이 결정됩니다.

의미 있는 방식으로 작업을 수행할 수 없으므로 구현이 제공되지 않았습니다. 추상 기본 클래스의 메서드에 대한 구현을 제공하거나 범용 인터페이스를 구현하는 개체에서 메서드를 호출하는 경우와 메서드에 의미 있는 구현이 없는 경우 일반적인 예외입니다.

예를 들어 클래스는 Convert 인터페이스를 IConvertible 구현합니다. 즉, 모든 기본 형식을 다른 모든 기본 형식으로 변환하는 메서드를 포함해야 합니다. 그러나 이러한 변환의 대부분은 불가능합니다. 따라서 예를 들어, 메서드 Convert.ToBoolean(DateTime)에 대한 호출은 NotSupportedException와 DateTime 값 간에 변환이 불가능하기 때문에 Boolean 예외를 발생시킵니다.

예외를 제거하려면 메서드 호출을 제거해야 합니다.

개체의 상태가 지정된 경우 메서드 호출이 지원되지 않습니다. 개체의 상태 때문에 기능을 사용할 수 없는 멤버를 호출하려고 합니다. 다음 세 가지 방법 중 하나로 예외를 제거할 수 있습니다.

- 개체의 상태를 미리 알고 있지만 지원되지 않는 메서드 또는 속성을 호출했습니다. 이 경우 멤버 호출은 오류이며 제거할 수 있습니다.

- 개체의 상태를 미리 알고 있지만(일반적으로 코드에서 인스턴스화했기 때문) 개체가 잘못 구성되었습니다. 다음 예제에서는 이 문제를 보여 줍니다. 읽기 전용 FileStream 개체를 만든 다음 쓰기를 시도합니다.

```
C#
```

```
using System;
using System.IO;
using System.Text;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main()
    {
        Encoding enc = Encoding.Unicode;
        String value = "This is a string to persist.";
        Byte[] bytes = enc.GetBytes(value);

        FileStream fs = new FileStream(@".\TestFile.dat",
                                       FileMode.Open,
                                       FileAccess.Read);
        Task t = fs.WriteAsync(enc.GetPreamble(), 0,
enc.GetPreamble().Length);
        Task t2 = t.ContinueWith((a) => fs.WriteAsync(bytes, 0,
bytes.Length));
        await t2;
        fs.Close();
    }
}
// The example displays the following output:
//    Unhandled Exception: System.NotSupportedException: Stream does not
```

```
support writing.
//       at System.IO.Stream.BeginWriteInternal(Byte[] buffer, Int32 offset,
Int32 count, AsyncCallback callback, Object state
//    , Boolean serializeAsynchronously)
//       at System.IO.FileStream.BeginWrite(Byte[] array, Int32 offset, Int32
numBytes, AsyncCallback userCallback, Object sta
//    teObject)
//       at System.IO.Stream.<>c.<BeginEndWriteAsync>b__53_0(Stream stream,
ReadWriteParameters args, AsyncCallback callback,
//    Object state)
//       at
System.Threading.Tasks.TaskFactory`1.FromAsyncTrim[TInstance,TArgs](TInstance
thisRef, TArgs args, Func`5 beginMet
//    hod, Func`3 endMethod)
//       at System.IO.Stream.BeginEndWriteAsync(Byte[] buffer, Int32 offset,
Int32 count)
//       at System.IO.FileStream.WriteAsync(Byte[] buffer, Int32 offset,
Int32 count, CancellationToken cancellationToken)
//       at System.IO.Stream.WriteAsync(Byte[] buffer, Int32 offset, Int32
count)
//       at Example.Main()
```

인스턴스화된 개체가 원하는 기능을 지원하는지 확인하여 예외를 제거합니다. 다음 예제에서는 생성자에 올바른 인수를 제공하여 읽기 전용 FileStream 개체의 FileStream.FileStream(String, FileMode, FileAccess) 문제를 해결합니다.

- 개체의 상태를 미리 알지 못하며 개체가 특정 작업을 지원하지 않습니다. 대부분의 경우 개체는 특정 작업 집합을 지원하는지 여부를 나타내는 속성 또는 메서드를 포함해야 합니다. 개체의 값을 확인하고 적절한 경우에만 멤버를 호출하여 예외를 제거할 수 있습니다.

  다음 예제는 읽기를 지원하지 않는 스트림의 시작 부분에서 읽기를 시도할 때 `DetectEncoding` 예외를 throw하는 NotSupportedException 메서드를 정의합니다.

C#

```
using System;
using System.IO;
using System.Threading.Tasks;

public class TestPropEx1
{
    public static async Task Main()
    {
        String name = @".\TestFile.dat";
        var fs = new FileStream(name,
                                FileMode.Create,
                                FileAccess.Write);
        Console.WriteLine("Filename: {0}, Encoding: {1}",
                          name, await FileUtilities1.GetEncodingType(fs));
    }
}
```

```
public class FileUtilities1
{
    public enum EncodingType
    { None = 0, Unknown = -1, Utf8 = 1, Utf16 = 2, Utf32 = 3 }

    public async static Task<EncodingType> GetEncodingType(FileStream fs)
    {
        Byte[] bytes = new Byte[4];
        int bytesRead = await fs.ReadAsync(bytes, 0, 4);
        if (bytesRead < 2)
            return EncodingType.None;

        if (bytesRead >= 3 & (bytes[0] == 0xEF && bytes[1] == 0xBB &&
bytes[2] == 0xBF))
            return EncodingType.Utf8;

        if (bytesRead == 4)
        {
            var value = BitConverter.ToUInt32(bytes, 0);
            if (value == 0x0000FEFF | value == 0xFEFF0000)
                return EncodingType.Utf32;
        }

        var value16 = BitConverter.ToUInt16(bytes, 0);
        if (value16 == (ushort)0xFEFF | value16 == (ushort)0xFFFE)
            return EncodingType.Utf16;

        return EncodingType.Unknown;
    }
}
// The example displays the following output:
//    Unhandled Exception: System.NotSupportedException: Stream does not
support reading.
//       at System.IO.FileStream.BeginRead(Byte[] array, Int32 offset, Int32
numBytes, AsyncCallback callback, Object state)
//       at System.IO.Stream.<>c.<BeginEndReadAsync>b__46_0(Stream stream,
ReadWriteParameters args, AsyncCallback callback, Object state)
//       at System.Threading.Tasks.TaskFactory`1.FromAsyncTrim[TInstance,
TArgs](TInstance thisRef, TArgs args, Func`5 beginMethod, Func`3 endMethod)
//       at System.IO.Stream.BeginEndReadAsync(Byte[] buffer, Int32 offset,
Int32 count)
//       at System.IO.FileStream.ReadAsync(Byte[] buffer, Int32 offset, Int32
count, CancellationToken cancellationToken)
//       at System.IO.Stream.ReadAsync(Byte[] buffer, Int32 offset, Int32
count)
//       at FileUtilities.GetEncodingType(FileStream fs) in
C:\Work\docs\program.cs:line 26
//       at Example.Main() in C:\Work\docs\program.cs:line 13
//       at Example.<Main>()
```

속성 값을 FileStream.CanRead 검사하고 스트림이 읽기 전용인 경우 메서드를 종료하여 예외를 제거할 수 있습니다.

C#

```
    public static async Task<EncodingType> GetEncodingType(FileStream fs)
    {
        if (!fs.CanRead)
            return EncodingType.Unknown;

        Byte[] bytes = new Byte[4];
        int bytesRead = await fs.ReadAsync(bytes, 0, 4);
        if (bytesRead < 2)
            return EncodingType.None;

        if (bytesRead >= 3 & (bytes[0] == 0xEF && bytes[1] == 0xBB &&
bytes[2] == 0xBF))
            return EncodingType.Utf8;

        if (bytesRead == 4)
        {
            var value = BitConverter.ToUInt32(bytes, 0);
            if (value == 0x0000FEFF | value == 0xFEFF0000)
                return EncodingType.Utf32;
        }

        var value16 = BitConverter.ToUInt16(bytes, 0);
        if (value16 == (ushort)0xFEFF | value16 == (ushort)0xFFFE)
            return EncodingType.Utf16;

        return EncodingType.Unknown;
    }
}
// The example displays the following output:
//       Filename: .\TestFile.dat, Encoding: Unknown
```

관련 예외 유형

예외는 NotSupportedException 다른 두 가지 예외 유형과 밀접하게 관련되어 있습니다.

- NotImplementedException

  이 예외는 메서드를 구현할 수 있지만 멤버가 이후 버전에서 구현되거나, 멤버를 특정 플랫폼에서 사용할 수 없거나, 멤버가 추상 클래스에 속하며 파생 클래스가 구현을 제공해야 하기 때문에 throw됩니다.

- InvalidOperationException

  이 예외는 일반적으로 개체가 요청된 작업을 수행할 수 있는 시나리오에서 throw되며 개체 상태는 작업을 수행할 수 있는지 여부를 결정합니다.

.NET Compact Framework 참고 사항

.NET Compact Framework를 사용하고 네이티브 함수에서 P/Invoke를 사용하는 경우 다음과 같은 경우 이 예외가 throw될 수 있습니다.

- 관리 코드의 선언이 잘못되었습니다.
- .NET Compact Framework는 수행하려는 작업을 지원하지 않습니다.
- 내보내기에서 DLL 이름이 변형됩니다.

NotSupportedException 예외가 발생한 경우 다음을 확인합니다.

- .NET Compact Framework P/Invoke 제한을 위반하는 경우
- 미리 할당된 메모리가 필요한 인수의 경우 이러한 변수가 있는 경우 기존 변수에 대한 참조를 전달해야 합니다.
- 내보낸 함수의 이름이 올바른지 확인합니다. DumpBin.exe을 사용하여 확인할 수 있습니다.
- 너무 많은 인수를 전달하려고 시도하지 않는다는 것입니다.

System.TypeInitializationException 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스 초기화 도중 형식 초기화에 실패하면, 해당 형식의 클래스 초기자에서 던져진 예외에 대한 참조가 만들어져 TypeInitializationException에 전달됩니다. InnerException 속성은 기본 TypeInitializationException 예외를 보유합니다.

일반적으로 예외는 애플리케이션이 TypeInitializationException 계속되지 않도록 하는 치명적인 조건(런타임이 형식을 인스턴스화할 수 없음)을 반영합니다. 애플리케이션 실행 환경에서 일부 변경이 발생할 경우 가장 흔히 TypeInitializationException이(가) throw됩니다. 따라서 디버그 코드 문제를 해결하는 것 외에는 예외를 블록에서 `try`/`catch` 처리해서는 안 됩니다. 대신 예외의 원인을 조사하고 제거해야 합니다.

TypeInitializationException 는 값이 0x80131534 HRESULT `COR_E_TYPEINITIALIZATION`를 사용합니다.

TypeInitializationException인스턴스의 초기 속성 값 목록은 TypeInitializationException 생성자를 참조하세요.

다음 섹션에서는 TypeInitializationException 예외가 발생하는 어떤 상황에 대해 설명합니다.

정적 생성자

정적 생성자(있는 경우)는 형식의 새 인스턴스를 만들기 전에 런타임에서 자동으로 호출됩니다. 정적 생성자는 개발자가 명시적으로 정의할 수 있습니다. 정적 생성자가 명시적으로 정의되지 않은 경우, 컴파일러는 C# 또는 F#의 `static` 멤버나 Visual Basic의 `Shared` 멤버를 초기화하기 위해 자동으로 생성자를 만듭니다. 정적 생성자에 대한 자세한 내용은 정적 생성자를 참조하세요.

가장 일반적으로 정적 생성자가 형식을 TypeInitializationException 인스턴스화할 수 없는 경우 예외가 발생합니다. 이 속성은 InnerException 정적 생성자가 형식을 인스턴스화할 수 없는 이유를 나타냅니다. 예외의 일반적인 원인 TypeInitializationException 은 다음과 같습니다.

- 정적 생성자의 처리되지 않은 예외

  정적 생성자에서 예외가 throw 되면 해당 예외가 TypeInitializationException 예외로 래핑되고 해당 형식을 인스턴스화할 수 없습니다.

  종종 이 예외의 문제를 해결하기 어려운 이유는 정적 생성자가 소스 코드에서 항상 명시적으로 정의되지 않는다는 것입니다. 다음과 같은 경우 정적 생성자가 형식에 존재합니다.

- 유형의 멤버로 명시적으로 정의되었습니다.
- 형식에는 `static` 단일 문에서 선언되고 초기화된 (C# 또는 F#) 또는 `Shared` (Visual Basic의 경우) 변수가 있습니다. 이 경우 언어 컴파일러는 형식에 대한 정적 생성자를 생성합니다. IL 디스어셈블러와 같은 유틸리티를 사용하여 검사할 수 있습니다. 예를 들어 C# 및 VB 컴파일러가 다음 예제를 컴파일할 때 다음과 유사한 정적 생성자에 대한 IL을 생성합니다.

il

```
.method private specialname rtspecialname static
        void  .cctor() cil managed
{
   // Code size       12 (0xc)
   .maxstack  8
   IL_0000:  ldc.i4.3
   IL_0001:  newobj     instance void TestClass::.ctor(int32)
   IL_0006:  stsfld     class TestClass Example::test
   IL_000b:  ret
} // end of method Example::.cctor
```

다음 예제에서는 컴파일러가 생성한 정적 생성자가 던진 예외를 보여줍니다. 클래스에는 `Example` 형식의 `static`(C#) 또는 `Shared`(Visual Basic) 필드가 포함되어 있으며, 이 필드는 클래스 생성자에 3 값을 전달하여 인스턴스화됩니다. 그러나 이 값은 불법입니다. 0 또는 1의 값만 허용됩니다. 결과적으로 `TestClass` 클래스 생성자는 ArgumentOutOfRangeException을(를) 던집니다. 이 예외는 처리되지 않으므로 TypeInitializationException 예외에 감싸집니다.

C#

```
using System;

public class Example
{
   private static TestClass test = new TestClass(3);

   public static void Main()
   {
      Example ex = new Example();
      Console.WriteLine(test.Value);
   }
}

public class TestClass
{
   public readonly int Value;

   public TestClass(int value)
   {
```

```
      if (value < 0 || value > 1) throw new
ArgumentOutOfRangeException(nameof(value));
      Value = value;
   }
}
// The example displays the following output:
//    Unhandled Exception: System.TypeInitializationException:
//       The type initializer for 'Example' threw an exception. --->
//       System.ArgumentOutOfRangeException: Specified argument was out of
the range of valid values.
//       at TestClass..ctor(Int32 value)
//       at Example..cctor()
//       --- End of inner exception stack trace ---
//       at Example.Main()
```

예외 메시지에는 속성에 대한 InnerException 정보가 표시됩니다.

- 누락된 어셈블리 또는 데이터 파일

  예외의 일반적인 원인은 애플리케이션의 TypeInitializationException 개발 및 테스트 환경에 있던 어셈블리 또는 데이터 파일이 런타임 환경에서 누락된 것입니다. 예를 들어 다음 예제를 이 명령줄 구문을 사용하여 Missing1a.dll이라는 어셈블리로 컴파일할 수 있습니다.

  C#

  ```
  csc -t:library Missing1a.cs
  ```

  C#

  ```
  using System;

  public class InfoModule
  {
     private DateTime firstUse;
     private int ctr = 0;

     public InfoModule(DateTime dat)
     {
        firstUse = dat;
     }

     public int Increment()
     {
        return ++ctr;
     }

     public DateTime GetInitializationTime()
     {
        return firstUse;
  ```

```
    }
}
```

그런 다음 Missing1a.dll대한 참조를 포함하여 다음 예제를 Missing1.exe이라는 실행 파일로 컴파일할 수 있습니다.

C#

```
csc Missing1.cs /r:Missing1a.dll
```

그러나 Missing1a.dll 이름을 바꾸거나 이동하거나 삭제하고 예제를 실행하면 예외가 TypeInitializationException throw되고 예제에 표시된 출력이 표시됩니다. 예외 메시지에는 속성에 대한 정보가 포함됩니다 InnerException . 이 경우 내부 예외는 FileNotFoundException 런타임이 종속 어셈블리를 찾을 수 없기 때문에 throw되는 예외입니다.

C#

```
using System;

public class MissingEx1
{
    public static void Main()
    {
        Person p = new Person("John", "Doe");
        Console.WriteLine(p);
    }
}

public class Person
{
    static readonly InfoModule s_infoModule;

    readonly string _fName;
    readonly string _lName;

    static Person()
    {
        s_infoModule = new InfoModule(DateTime.UtcNow);
    }

    public Person(string fName, string lName)
    {
        _fName = fName;
        _lName = lName;
        s_infoModule.Increment();
    }

    public override string ToString()
    {
```

```
        return string.Format("{0} {1}", _fName, _lName);
    }
}
// The example displays the following output if missing1a.dll is renamed or
removed:
//    Unhandled Exception: System.TypeInitializationException:
//       The type initializer for 'Person' threw an exception. --->
//       System.IO.FileNotFoundException: Could not load file or assembly
//       'Missing1a, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null'
//       or one of its dependencies. The system cannot find the file
specified.
//       at Person..cctor()
//       --- End of inner exception stack trace ---
//       at Person..ctor(String fName, String lName)
//       at Example.Main()
```

> ⓘ **참고**
>
> 이 예제에서는 어셈블리를 로드할 수 없어서 TypeInitializationException 예외가 발생했습니다. 정적 생성자가 찾을 수 없는 구성 파일, XML 파일 또는 직렬화된 데이터가 포함된 파일과 같은 데이터 파일을 열려고 시도하는 경우에도 예외가 throw될 수 있습니다.

정규 표현식 일치 시간 제한 값

애플리케이션별 도메인 기준으로 정규식 패턴 일치 작업에 대한 기본 시간 제한 값을 설정할 수 있습니다. 시간 제한은 "REGEX_DEFAULT_MATCH_TIMEOUT" 속성의 TimeSpan 값을 메서드 내에서 지정하여 AppDomain.SetData 정의됩니다. 시간 간격은 0보다 크고 약 24일 미만인 유효한 TimeSpan 개체여야 합니다. 이러한 요구 사항이 충족되지 않으면 기본 시간 제한 값을 설정하려고 할 때 ArgumentOutOfRangeException가 throw되고, 이 값은 TypeInitializationException 예외로 래핑됩니다.

다음 예제는 "REGEX_DEFAULT_MATCH_TIMEOUT" 속성에 할당된 값이 유효하지 않을 때 던져지는 예외 TypeInitializationException를 보여줍니다. 예외를 제거하려면 "REGEX_DEFAULT_MATCH_TIMEOUT" 속성을 TimeSpan 0보다 크고 약 24일 미만의 값으로 설정합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class RegexEx1
{
    public static void Main()
```

```
    {
        AppDomain domain = AppDomain.CurrentDomain;
        // Set a timeout interval of -2 seconds.
        domain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT", TimeSpan.FromSeconds(-2));

        Regex rgx = new Regex("[aeiouy]");
        Console.WriteLine($"Regular expression pattern: {rgx.ToString()}");
        Console.WriteLine($"Timeout interval for this regex:
{rgx.MatchTimeout.TotalSeconds} seconds");
    }
}
// The example displays the following output:
//    Unhandled Exception: System.TypeInitializationException:
//       The type initializer for 'System.Text.RegularExpressions.Regex' threw an
exception. --->
//       System.ArgumentOutOfRangeException: Specified argument was out of the
range of valid values.
//       Parameter name: AppDomain data 'REGEX_DEFAULT_MATCH_TIMEOUT' contains an
invalid value or
//       object for specifying a default matching timeout for
System.Text.RegularExpressions.Regex.
//       at System.Text.RegularExpressions.Regex.InitDefaultMatchTimeout()
//       at System.Text.RegularExpressions.Regex..cctor()
//       --- End of inner exception stack trace ---
//       at System.Text.RegularExpressions.Regex..ctor(String pattern)
//       at Example.Main()
```

달력 및 문화 데이터

달력을 인스턴스화하려고 하지만 런타임에서 해당 달력에 해당하는 개체를 CultureInfo 인스턴스화할 수 없는 경우 예외가 TypeInitializationException 발생합니다. 이 예외는 다음 달력 클래스 생성자에 의해 throw될 수 있습니다.

- 클래스의 매개 변수가 없는 생성자입니다 JapaneseCalendar .
- 클래스의 매개 변수가 없는 생성자입니다 KoreanCalendar .
- 클래스의 매개 변수가 없는 생성자입니다 TaiwanCalendar .

이러한 문화권에 대한 문화권 데이터는 모든 시스템에서 사용할 수 있어야 하므로 이 예외가 발생하는 경우는 거의 없습니다.

.NET의 숫자

2025. 06. 17.

.NET은 다음과 같이 다양한 숫자 정수 및 부동 소수점 기본 형식을 제공합니다.

- System.Half- 반정밀도 부동 소수점 숫자를 나타냅니다.
- System.Decimal은 소수점 부동 소수점 숫자를 나타냅니다.
- System.Numerics.BigInteger이론적 상한 또는 하한이 없는 정수 계열 형식입니다.
- System.Numerics.Complex은 복소수를 나타냅니다.
- SIMD를 지원하는 데이터형의 System.Numerics 네임스페이스 내 집합입니다.

정수 형식

.NET은 다음 표에 나열된 부호 있는 8비트, 16비트, 32비트, 64비트 및 128비트 정수 형식을 모두 지원합니다.

부인 정수 형식

테이블 확장

유형	크기(바이트)	최소값	최대값
System.Int16	2	-32,768	32,767
System.Int32	4	-2,147,483,648	2,147,483,647
System.Int64	8 (여덟)	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
System.Int128	16	-170,141,183,460,469,231,731,687,303,715,884,105,728	170,141,183,460,469,231,731,687,303,715,884,105,727
System.SByte	1	-128	127
System.IntPtr (32비트 프로세스)	4	-2,147,483,648	2,147,483,647
System.IntPtr (64비트 프로세스)	8 (여덟)	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

부호 없는 정수 형식

테이블 확장

유형	크기(바이트)	최소값	최대값
System.Byte	1	0	255
System.UInt16	2	0	65,535

유형	크기(바이트)	최소값	최대값
System.UInt32	4	0	4,294,967,295
System.UInt64	8 (여덟)	0	18,446,744,073,709,551,615
System.UInt128	16	0	340,282,366,920,938,463,463,374,607,431,768,211,455
System.UIntPtr (32비트 프로세스)	4	0	4,294,967,295
System.UIntPtr (64비트 프로세스)	8 (여덟)	0	18,446,744,073,709,551,615

각 정수 형식은 표준 산술 연산자 집합을 지원합니다. 이 클래스는 System.Math 더 광범위한 수학 함수 집합에 대한 메서드를 제공합니다.

System.BitConverter 클래스를 사용하여 정수 값의 개별 비트로 작업할 수도 있습니다.

> ⓘ **참고**
>
> 부호 없는 정수 형식은 CLS 규격이 아닙니다. 자세한 내용은 **언어 독립성 및 언어 독립적 구성 요소를 참조하세요**.

BigInteger

구조체 System.Numerics.BigInteger 는 이론적으로 값에 상한 또는 하한이 없는 임의로 큰 정수를 나타내는 변경할 수 없는 형식입니다. 형식의 BigInteger 메서드는 다른 정수 계열 형식의 메서드와 밀접하게 유사합니다.

부동 소수점 형식

.NET에는 다음과 같은 부동 소수점 형식이 포함됩니다.

테이블 확장

유형	크기(바이트)	대략적 범위	원시적인?	비고
System.Half	2	±65504	아니오	.NET 5에 도입
System.Single	4	$\pm 3.4 \times 10^{38}$	예	
System.Double	8 (여덟)	$\pm 1.7 \times 10^{308}$	예	
System.Decimal	16	$\pm 7.9228 \times 10^{28}$	아니오	

Half, Single, 및 Double 형식은 수가 아님과 무한대를 나타내는 특수 값을 지원합니다. 예를 들어 형식은 Double 다음 값을 Double.NaNDouble.NegativeInfinityDouble.PositiveInfinity제공합니다. Double.IsNaN, Double.IsInfinity, Double.IsPositiveInfinity, 및 Double.IsNegativeInfinity 메서드를 사용하여 이러한 특수 값을 테스트합니다.

각 부동 소수점 형식은 표준 산술 연산자 집합을 지원합니다. 이 클래스는 System.Math 더 광범위한 수학 함수 집합에 대한 메서드를 제공합니다. .NET Core 2.0 이상에는 System.MathF 클래스가 포함되어 있으며, 이는 Single 형식의 인수를 위한 메서드를 제공합니다.

Double, Single, Half 값을 개별 비트 단위로 System.BitConverter 클래스를 사용하여 작업할 수도 있습니다. 구조체에는 소수 값의 개별 비트를 다루기 위한 System.Decimal 고유한 메서드와 몇 가지 추가 수학 연산을 수행하기 위한

Decimal.GetBits 및 Decimal(Int32[]) 고유 메서드 집합이 있습니다.

Double, Single, 및 Half 형식은 본질적으로 정확하지 않은 값(예: 두 별 사이의 거리)과 높은 정밀도와 작은 반올림 오류가 필요하지 않은 애플리케이션에서 사용되도록 설계되었습니다. System.Decimal 정밀도가 더 높고 반올림 오류를 최소화해야 하는 경우 형식을 사용합니다.

> ⓘ **참고**
>
> Decimal 형식은 반올림의 필요성을 없애지 않습니다. 대신 반올림으로 인한 오류를 최소화합니다.

콤플렉스

구조체는 System.Numerics.Complex 복소수, 즉 실수 부분과 허수 부분이 있는 숫자를 나타냅니다. 산술, 비교, 같음, 명시적 및 암시적 변환 연산자뿐만 아니라 수학, 대수 및 삼각 메서드의 표준 집합을 지원합니다.

SIMD 사용 형식

네임스페이스에는 System.Numerics .NET SIMD 사용 형식 집합이 포함됩니다. SIMD(단일 명령 다중 데이터) 작업은 하드웨어 수준에서 병렬 처리할 수 있습니다. 이는 수학, 과학 및 그래픽 앱에서 흔히 볼 수 있는 벡터화된 계산의 처리량을 증가합니다.

.NET SIMD 사용 형식에는 다음이 포함됩니다.

- Vector2, Vector3 및 Vector4 형식은 각각 2, 3, 4개의 Single 값을 가진 벡터를 나타냅니다.
- 3x2 행렬을 나타내고 Matrix3x24x4 행렬을 나타내는 두 개의 행렬 형식Matrix4x4입니다.
- Plane 3차원 공간의 평면을 나타내는 형식입니다.
- Quaternion 3차원 물리적 회전을 인코딩하는 데 사용되는 벡터를 나타내는 형식입니다.
- Vector<T> 지정된 숫자 형식의 벡터를 나타내고 SIMD 지원을 활용하는 광범위한 연산자 집합을 제공하는 형식입니다. 인스턴스 수는 Vector<T> 고정되지만 해당 값 Vector<T>.Count 은 코드가 실행되는 컴퓨터의 CPU에 따라 달라집니다.

  > ⓘ **참고**
  >
  > 형식은 Vector<T> .NET Core 및 .NET 5 이상에 포함되지만 .NET Framework에는 포함되지 않습니다. .NET Framework를 사용하는 경우 System.Numerics.Vectors NuGet 패키지를 설치하여 이 형식에 액세스합니다.

SIMD 사용 형식은 SIMD 사용이 아닌 하드웨어 또는 JIT 컴파일러와 함께 사용할 수 있는 방식으로 구현됩니다. SIMD 지침을 활용하려면 .NET Core 및 .NET Framework 4.6 이상 버전에 포함된 RyuJIT 컴파일러를 사용하는 런타임에서 64비트 앱을 실행해야 합니다. 64비트 프로세서를 대상으로 할 때 SIMD 지원을 추가합니다.

자세한 내용은 SIMD 가속 숫자 형식 사용을 참조하세요.

참고하십시오

- 표준 숫자 형식 문자열

- C의 부동 소수점 숫자 형식#

System.Boolean 구조체

아티클 • 2025. 03. 23.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Boolean 인스턴스는 `true` 또는 `false`두 값 중 하나를 가질 수 있습니다.

Boolean 구조는 다음 작업을 지원하는 메서드를 제공합니다.

- 불리언 값을 문자열로 변환: ToString
- 문자열을 파싱하여 부울 값으로 변환: Parse 및 TryParse
- 값 비교: CompareTo 및 Equals

이 문서에서는 이러한 작업 및 기타 사용량 세부 정보를 설명합니다.

부울 값 서식 지정

Boolean 문자열 표현은 `true` 값의 경우 "True"이고 `false` 값의 경우 "False"입니다. Boolean 값의 문자열 표현은 읽기 전용 TrueString 및 FalseString 필드에 의해 정의됩니다.

ToString 메서드를 사용하여 부울 값을 문자열로 변환합니다. 부울 구조에는 매개 변수가 없는 ToString() 메서드와 서식을 제어하는 매개 변수를 포함하는 ToString(IFormatProvider) 메서드의 두 가지 ToString 오버로드가 포함됩니다. 그러나 이 매개 변수는 무시되므로 두 오버로드는 동일한 문자열을 생성합니다. ToString(IFormatProvider) 메서드는 문화권 구분 서식을 지원하지 않습니다.

다음 예제에서는 ToString 메서드를 사용하여 서식을 지정하는 방법을 보여 줍니다. C# 및 VB 예제에서는 복합 서식 지정 기능을 사용하고 F# 예제에서는 문자열 보간사용합니다. 두 경우 모두 ToString 메서드가 암시적으로 호출됩니다.

C#

```
using System;

public class Example10
{
   public static void Main()
   {
      bool raining = false;
      bool busLate = true;

      Console.WriteLine($"It is raining: {raining}");
      Console.WriteLine($"The bus is late: {busLate}");
   }
```

```
}
// The example displays the following output:
//       It is raining: False
//       The bus is late: True
```

Boolean 구조에는 두 개의 값만 있을 수 있으므로 사용자 지정 서식을 쉽게 추가할 수 있습니다. 다른 문자열 리터럴이 "True" 및 "False"로 대체되는 간단한 사용자 지정 서식의 경우 C#의 조건부 연산자 또는 Visual Basic에서 If 연산자와 같이 언어에서 지원하는 조건부 평가 기능을 사용할 수 있습니다. 다음 예제에서는 이 기술을 사용하여 Boolean 값의 서식을 "True" 및 "False"가 아닌 "예" 및 "아니요"로 지정합니다.

C#

```
using System;

public class Example11
{
    public static void Main()
    {
        bool raining = false;
        bool busLate = true;

        Console.WriteLine($"It is raining: {(raining ? "Yes" : "No")}");
        Console.WriteLine($"The bus is late: {(busLate ? "Yes" : "No")}");
    }
}
// The example displays the following output:
//       It is raining: No
//       The bus is late: Yes
```

문화권 구분 서식을 포함하여 더 복잡한 사용자 지정 서식 지정 작업의 경우 String.Format(IFormatProvider, String, Object[]) 메서드를 호출하고 ICustomFormatter 구현을 제공할 수 있습니다. 다음 예제에서는 ICustomFormatter 및 IFormatProvider 인터페이스를 구현하여 영어(미국), 프랑스어(프랑스) 및 러시아어(러시아) 문화권에 문화권 구분 부울 문자열을 제공합니다.

C#

```
using System;
using System.Globalization;

public class Example4
{
   public static void Main()
   {
      String[] cultureNames = { "", "en-US", "fr-FR", "ru-RU" };
      foreach (var cultureName in cultureNames) {
         bool value = true;
         CultureInfo culture =
```

```
CultureInfo.CreateSpecificCulture(cultureName);
         BooleanFormatter formatter = new BooleanFormatter(culture);

         string result = string.Format(formatter, "Value for '{0}': {1}",
culture.Name, value);
         Console.WriteLine(result);
      }
   }
}

public class BooleanFormatter : ICustomFormatter, IFormatProvider
{
   private CultureInfo culture;

   public BooleanFormatter() : this(CultureInfo.CurrentCulture)
   { }

   public BooleanFormatter(CultureInfo culture)
   {
      this.culture = culture;
   }

   public Object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(string fmt, Object arg, IFormatProvider
formatProvider)
   {
      // Exit if another format provider is used.
      if (! formatProvider.Equals(this)) return null;

      // Exit if the type to be formatted is not a Boolean
      if (! (arg is Boolean)) return null;

      bool value = (bool) arg;
      switch (culture.Name) {
         case "en-US":
            return value.ToString();
         case "fr-FR":
            if (value)
               return "vrai";
            else
               return "faux";
         case "ru-RU":
            if (value)
               return "верно";
            else
               return "неверно";
         default:
            return value.ToString();
```

```
      }
   }
}
// The example displays the following output:
//       Value for '': True
//       Value for 'en-US': True
//       Value for 'fr-FR': vrai
//       Value for 'ru-RU': верно
```

선택적으로, 리소스 파일을 사용하여 특정 문화에 맞는 부울 문자열을 정의할 수 있습니다.

부울 값으로 변환 및 변환 해제

Boolean 구조체는 IConvertible 인터페이스를 구현합니다. 따라서 Convert 클래스를 사용하여 Boolean 값과 .NET의 다른 기본 형식 간에 변환을 수행하거나 Boolean 구조체의 명시적 구현을 호출할 수 있습니다. 그러나 Boolean과 다음 형식 간의 변환은 지원되지 않으므로 해당 변환 메서드가 InvalidCastException 예외를 발생시킵니다.

- Boolean Char(Convert.ToBoolean(Char) 및 Convert.ToChar(Boolean) 메서드) 간의 변환입니다.
- Boolean DateTime(Convert.ToBoolean(DateTime) 및 Convert.ToDateTime(Boolean) 메서드) 간의 변환입니다.

모든 정수 또는 부동 소수점 숫자를 부울 값으로 변환할 때, 0이 아닌 값은 `true`으로, 0 값은 `false`로 변환됩니다. 다음 예제에서는 Convert.ToBoolean 클래스의 선택한 오버로드를 호출하여 이를 보여 줍니다.

C#

```
using System;

public class Example2
{
   public static void Main()
   {
      Byte byteValue = 12;
      Console.WriteLine(Convert.ToBoolean(byteValue));
      Byte byteValue2 = 0;
      Console.WriteLine(Convert.ToBoolean(byteValue2));
      int intValue = -16345;
      Console.WriteLine(Convert.ToBoolean(intValue));
      long longValue = 945;
      Console.WriteLine(Convert.ToBoolean(longValue));
      SByte sbyteValue = -12;
      Console.WriteLine(Convert.ToBoolean(sbyteValue));
      double dblValue = 0;
```

```
      Console.WriteLine(Convert.ToBoolean(dblValue));
      float sngValue = .0001f;
      Console.WriteLine(Convert.ToBoolean(sngValue));
   }
}
// The example displays the following output:
//       True
//       False
//       True
//       True
//       True
//       False
//       True
```

Boolean에서 숫자 값으로 변환할 때, Convert 클래스의 변환 메소드는 `true`을 1로 변환하고 `false`를 0으로 변환합니다. 그러나 Visual Basic 변환 함수는 `true` 255(Byte 값으로 변환하는 경우) 또는 -1(다른 모든 숫자 변환의 경우)로 변환합니다. 다음 예제에서는 Convert 메서드를 사용하여 `true` 숫자 값으로 변환하고, Visual Basic 예제의 경우 Visual Basic 언어의 고유한 변환 연산자를 사용합니다.

C#

```
using System;

public class Example3
{
   public static void Main()
   {
      bool flag = true;

      byte byteValue;
      byteValue = Convert.ToByte(flag);
      Console.WriteLine($"{flag} -> {byteValue}");

      sbyte sbyteValue;
      sbyteValue = Convert.ToSByte(flag);
      Console.WriteLine($"{flag} -> {sbyteValue}");

      double dblValue;
      dblValue = Convert.ToDouble(flag);
      Console.WriteLine($"{flag} -> {dblValue}");

      int intValue;
      intValue = Convert.ToInt32(flag);
      Console.WriteLine($"{flag} -> {intValue}");
   }
}
// The example displays the following output:
//       True -> 1
//       True -> 1
```

```
//       True -> 1
//       True -> 1
```

Boolean 문자열 값으로의 변환은 서식 부울 값 섹션을 참조하세요. 문자열에서 Boolean 값으로 변환하려면 부울 값 구문 분석 섹션을 참조하세요.

부울 값 구문 분석

Boolean 구조에는 문자열을 부울 값으로 변환하는 Parse 및 TryParse두 개의 정적 구문 분석 메서드가 포함됩니다. 부울 값의 문자열 표현은 각각 "True" 및 "False"인 TrueString 및 FalseString 필드 값의 대/소문자를 구분하지 않는 값으로 정의됩니다. 즉, 성공적으로 구문 분석할 수 있는 문자열은 "True", "False", "true", "false" 또는 어떤 대소문자 조합의 동등한 문자열입니다. "0" 또는 "1"과 같은 숫자 문자열을 구문 분석할 수 없습니다. 문자열 비교를 수행할 때 선행 또는 후행 공백 문자는 고려되지 않습니다.

다음 예제에서는 Parse 및 TryParse 메서드를 사용하여 여러 문자열을 구문 분석합니다. 대/소문자를 구분하지 않는 "True" 및 "False"만 성공적으로 구문 분석할 수 있습니다.

C#

```
using System;

public class Example7
{
   public static void Main()
   {
      string[] values = { null, String.Empty, "True", "False",
                          "true", "false", "    true    ",
                           "TrUe", "fAlSe", "fa lse", "0",
                          "1", "-1", "string" };
      // Parse strings using the Boolean.Parse method.
      foreach (var value in values) {
         try {
            bool flag = Boolean.Parse(value);
            Console.WriteLine($"'{value}' --> {flag}");
         }
         catch (ArgumentException) {
            Console.WriteLine("Cannot parse a null string.");
         }
         catch (FormatException) {
            Console.WriteLine($"Cannot parse '{value}'.");
         }
      }
      Console.WriteLine();
      // Parse strings using the Boolean.TryParse method.
      foreach (var value in values) {
         bool flag = false;
         if (Boolean.TryParse(value, out flag))
            Console.WriteLine($"'{value}' --> {flag}");
```

```
         else
            Console.WriteLine($"Unable to parse '{value}'");
      }
   }
}
// The example displays the following output:
//       Cannot parse a null string.
//       Cannot parse ''.
//       'True' --> True
//       'False' --> False
//       'true' --> True
//       'false' --> False
//       '    true    ' --> True
//       'TrUe' --> True
//       'fAlSe' --> False
//       Cannot parse 'fa lse'.
//       Cannot parse '0'.
//       Cannot parse '1'.
//       Cannot parse '-1'.
//       Cannot parse 'string'.
//
//       Unable to parse ''
//       Unable to parse ''
//       'True' --> True
//       'False' --> False
//       'true' --> True
//       'false' --> False
//       '    true    ' --> True
//       'TrUe' --> True
//       'fAlSe' --> False
//       Cannot parse 'fa lse'.
//       Unable to parse '0'
//       Unable to parse '1'
//       Unable to parse '-1'
//       Unable to parse 'string'
```

Visual Basic에서 프로그래밍하는 경우 `CBool` 함수를 사용하여 숫자의 문자열 표현을 부울 값으로 변환할 수 있습니다. "0"은 `false`변환되고 0이 아닌 값의 문자열 표현은 `true` 변환됩니다. Visual Basic에서 프로그래밍하지 않는 경우, 숫자 문자열을 부울로 변환하기 전에 먼저 숫자로 변환해야 합니다. 다음 예제에서는 정수 배열을 부울 값으로 변환하여 이를 보여 줍니다.

C#

```
using System;

public class Example8
{
   public static void Main()
   {
      String[] values = { "09", "12.6", "0", "-13 " };
      foreach (var value in values) {
```

```
         bool success, result;
         int number;
         success = Int32.TryParse(value, out number);
         if (success) {
            // The method throws no exceptions.
            result = Convert.ToBoolean(number);
            Console.WriteLine($"Converted '{value}' to {result}");
         }
         else {
            Console.WriteLine($"Unable to convert '{value}'");
         }
      }
   }
}
// The example displays the following output:
//       Converted '09' to True
//       Unable to convert '12.6'
//       Converted '0' to False
//       Converted '-13 ' to True
```

부울 값 비교

부울 값은 `true` 또는 `false`때문에 인스턴스가 지정된 값보다 크거나 작거나 같은지 여부를 나타내는 CompareTo 메서드를 명시적으로 호출할 이유가 거의 없습니다. 일반적으로 두 부울 변수를 비교하려면 Equals 메서드를 호출하거나 언어의 같음 연산자를 사용합니다.

그러나 부울 변수를 리터럴 부울 값 `true` 또는 `false`비교하려는 경우 부울 값을 계산한 결과가 부울 값이므로 명시적 비교를 수행할 필요가 없습니다. 예를 들어 다음 두 식은 동일하지만 두 번째 식은 더 간결합니다. 그러나 두 기술 모두 비슷한 성능을 제공합니다.

C#

```
if (booleanValue == true) {
```

C#

```
if (booleanValue) {
```

부울 값을 이진 형태로 다루기

부울 값은 다음 예제와 같이 1 바이트 메모리를 차지합니다. C# 예제는 `/unsafe` 스위치를 사용하여 컴파일해야 합니다.

```
```

C#

```
using System;

public struct BoolStruct
{
   public bool flag1;
   public bool flag2;
   public bool flag3;
   public bool flag4;
   public bool flag5;
}

public class Example9
{
   public static void Main()
   {
      unsafe {
         BoolStruct b = new BoolStruct();
         bool* addr = (bool*) &b;
         Console.WriteLine($"Size of BoolStruct: {sizeof(BoolStruct)}");
         Console.WriteLine("Field offsets:");
         Console.WriteLine($"   flag1: {(bool*) &b.flag1 - addr}");
         Console.WriteLine($"   flag1: {(bool*) &b.flag2 - addr}");
         Console.WriteLine($"   flag1: {(bool*) &b.flag3 - addr}");
         Console.WriteLine($"   flag1: {(bool*) &b.flag4 - addr}");
         Console.WriteLine($"   flag1: {(bool*) &b.flag5 - addr}");
      }
   }
}
// The example displays the following output:
//       Size of BoolStruct: 5
//       Field offsets:
//          flag1: 0
//          flag1: 1
//          flag1: 2
//          flag1: 3
//          flag1: 4
```

바이트의 하위 비트는 해당 값을 나타내는 데 사용됩니다. 값이 1이면 `true`; 값이 0이면 `false`나타냅니다.

> **팁**
>
> System.Collections.Specialized.BitVector32 구조를 사용하여 부울 값 집합과 작업을 수행할 수 있습니다.

BitConverter.GetBytes(Boolean) 메서드를 호출하여 부울 값을 이진 표현으로 변환할 수 있습니다. 메서드는 단일 요소가 있는 바이트 배열을 반환합니다. 이진 표현에서 부울 값

을 복원하려면 BitConverter.ToBoolean(Byte[], Int32) 메서드를 호출할 수 있습니다.

다음 예제에서는 BitConverter.GetBytes 메서드를 호출하여 부울 값을 이진 표현으로 변환하고 값의 개별 비트를 표시한 다음 BitConverter.ToBoolean 메서드를 호출하여 이진 표현에서 값을 복원합니다.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        bool[] flags = { true, false };
        foreach (var flag in flags)
        {
            // Get binary representation of flag.
            Byte value = BitConverter.GetBytes(flag)[0];
            Console.WriteLine($"Original value: {flag}");
            Console.WriteLine($"Binary value:   {value}
({GetBinaryString(value)})");
            // Restore the flag from its binary representation.
            bool newFlag = BitConverter.ToBoolean(new Byte[] { value }, 0);
            Console.WriteLine($"Restored value: {flag}
{Environment.NewLine}");
        }
    }

    private static string GetBinaryString(Byte value)
    {
        string retVal = Convert.ToString(value, 2);
        return new string('0', 8 - retVal.Length) + retVal;
    }
}
// The example displays the following output:
//       Original value: True
//       Binary value:   1 (00000001)
//       Restored value: True
//
//       Original value: False
//       Binary value:   0 (00000000)
//       Restored value: False
```

부울 값을 사용하여 연산 수행

이 섹션에서는 앱에서 부울 값을 사용하는 방법을 보여 줍니다. 첫 번째 섹션에서는 플래그로 사용하는 것을 설명합니다. 두 번째는 산술 연산에 사용하는 방법을 보여 줍니다.

불리언 값을 플래그로 사용하기

부울 변수는 가장 일반적으로 플래그로 사용되어 일부 조건의 존재 여부 또는 부재를 알릴 수 있습니다. 예를 들어 String.Compare(String, String, Boolean) 메서드에서 `ignoreCase` 마지막 매개 변수는 두 문자열의 비교가 대/소문자를 구분하지 않는지(`ignoreCase` `true`) 또는 대/소문자 구분(`ignoreCase` `false`)인지를 나타내는 플래그입니다. 그런 다음 조건문에서 플래그 값을 평가할 수 있습니다.

다음 예제에서는 간단한 콘솔 앱을 사용하여 부울 변수를 플래그로 사용하는 방법을 보여 줍니다. 앱은 출력을 지정된 파일(`/f` 스위치)으로 리디렉션할 수 있도록 하고 출력을 지정된 파일과 콘솔(`/b` 스위치)으로 보낼 수 있도록 하는 명령줄 매개 변수를 허용합니다. 앱은 출력을 파일로 보낼지 여부를 나타내는 `isRedirected` 플래그와 출력을 콘솔로 보내야 함을 나타내는 `isBoth` 플래그를 정의합니다. F# 예제에서는 재귀 함수 사용하여 인수를 구문 분석합니다.

C#

```
using System;
using System.IO;
using System.Threading;

public class Example5
{
   public static void Main()
   {
      // Initialize flag variables.
      bool isRedirected = false;
      bool isBoth = false;
      String fileName = "";
      StreamWriter sw = null;

      // Get any command line arguments.
      String[] args = Environment.GetCommandLineArgs();
      // Handle any arguments.
      if (args.Length > 1) {
         for (int ctr = 1; ctr < args.Length; ctr++) {
            String arg = args[ctr];
            if (arg.StartsWith("/") || arg.StartsWith("-")) {
               switch (arg.Substring(1).ToLower())
               {
                  case "f":
                     isRedirected = true;
                     if (args.Length < ctr + 2) {
                        ShowSyntax("The /f switch must be followed by a
filename.");
                        return;
                     }
                     fileName = args[ctr + 1];
                     ctr++;
                     break;
```

```
                  case "b":
                     isBoth = true;
                     break;
                  default:
                     ShowSyntax(String.Format("The {0} switch is not
supported",
                                              args[ctr]));
                     return;
               }
            }
         }
      }

      // If isBoth is True, isRedirected must be True.
      if (isBoth &&  ! isRedirected) {
         ShowSyntax("The /f switch must be used if /b is used.");
         return;
      }

      // Handle output.
      if (isRedirected) {
         sw = new StreamWriter(fileName);
         if (!isBoth)
            Console.SetOut(sw);
      }
      String msg = String.Format("Application began at {0}", DateTime.Now);
      Console.WriteLine(msg);
      if (isBoth) sw.WriteLine(msg);
      Thread.Sleep(5000);
      msg = String.Format("Application ended normally at {0}",
DateTime.Now);
      Console.WriteLine(msg);
      if (isBoth) sw.WriteLine(msg);
      if (isRedirected) sw.Close();
   }

   private static void ShowSyntax(String errMsg)
   {
      Console.WriteLine(errMsg);
      Console.WriteLine("\nSyntax: Example [[/f <filename> [/b]]\n");
   }
}
```

부울 및 산술 연산

부울 값은 가끔 수학 계산을 유발하는 조건의 존재를 나타내는 데 사용됩니다. 예를 들어 `hasShippingCharge` 변수는 송장 금액에 배송 요금을 추가할지 여부를 나타내는 플래그로 사용될 수 있습니다.

`false` 값과의 연산은 결과에 영향을 미치지 않으므로, 부울(Boolean)을 수학적 연산에 사용할 정수 값으로 변환할 필요가 없습니다. 대신 조건부 논리를 사용할 수 있습니다.

다음 예제에서는 부분합, 배송 요금 및 선택적 서비스 요금으로 구성된 금액을 계산합니다. `hasServiceCharge` 변수는 서비스 요금이 적용되는지 여부를 결정합니다. 이 예제에서는 `hasServiceCharge` 숫자 값으로 변환하고 서비스 요금의 양을 곱하는 대신 조건부 논리를 사용하여 서비스 요금 금액을 적용할 수 있는 경우 추가합니다.

C#

```
using System;

public class Example6
{
   public static void Main()
   {
      bool[] hasServiceCharges = { true, false };
      Decimal subtotal = 120.62m;
      Decimal shippingCharge = 2.50m;
      Decimal serviceCharge = 5.00m;

      foreach (var hasServiceCharge in hasServiceCharges) {
         Decimal total = subtotal + shippingCharge +
                                 (hasServiceCharge ? serviceCharge : 0);
         Console.WriteLine($"hasServiceCharge = {hasServiceCharge}: The
total is {total:C2}.");
      }
   }
}
// The example displays output like the following:
//       hasServiceCharge = True: The total is $128.12.
//       hasServiceCharge = False: The total is $123.12.
```

부울 및 interop

기본 데이터 형식을 COM으로 마샬링하는 것은 일반적으로 간단하지만 Boolean 데이터 형식은 예외입니다. MarshalAsAttribute 특성을 적용하여 Boolean 형식을 다음 표현으로 마샬링할 수 있습니다.

테이블 확장

열거형 형식	관리되지 않는 형식
UnmanagedType.Bool	0이 아닌 값이 `true` 나타내고 0이 `false`나타내는 4바이트 정수 값입니다. 이는 구조체의 Boolean 필드와 플랫폼 호출에서 Boolean 매개 변수의 기본 형식입니다.
UnmanagedType.U1	1은 `true` 나타내고 0은 `false`나타내는 1 바이트 정수 값입니다.
UnmanagedType.VariantBool	2 바이트 정수 값입니다. 여기서 -1 `true` 나타내고 0은 `false`나타냅니다. COM interop 호출에서 Boolean 매개 변수의 기본 형식입

열거형 형식	관리되지 않는 형식
	니다.

System.Byte 구조체

아티클 • 2025. 03. 29.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Byte 0(Byte.MinValue 상수로 표시됨)에서 255(Byte.MaxValue 상수로 표현됨)까지의 값을 가진 부호 없는 정수를 나타내는 변경할 수 없는 값 형식입니다. .NET에는 -128 127까지의 값을 나타내는 부가적인 8비트 정수 값 형식(SByte)도 포함되어 있습니다.

바이트 값 인스턴스화

다음과 같은 여러 가지 방법으로 Byte 값을 인스턴스화할 수 있습니다.

- Byte 변수를 선언하고 Byte 데이터 형식의 범위 내에 있는 리터럴 정수 값을 할당할 수 있습니다. 다음 예제에서는 두 개의 Byte 변수를 선언하고 이러한 방식으로 값을 할당합니다.

  C#

  ```
  byte value1 = 64;
  byte value2 = 255;
  ```

- 바이트가 아닌 숫자 값을 바이트에 할당할 수 있습니다. 축소 변환이므로 C# 및 F#의 캐스트 연산자 또는 `Option Strict` 있는 경우 Visual Basic의 변환 메서드가 필요합니다. 비바이트 값이 소수 구성 요소를 포함하는 Single, Double또는 Decimal 값인 경우 해당 소수 부분의 처리는 변환을 수행하는 컴파일러에 따라 달라집니다. 다음 예제에서는 Byte 변수에 여러 숫자 값을 할당합니다.

  C#

  ```
  int int1 = 128;
  try
  {
      byte value1 = (byte)int1;
      Console.WriteLine(value1);
  }
  catch (OverflowException)
  {
      Console.WriteLine($"{int1} is out of range of a byte.");
  }

  double dbl2 = 3.997;
  try
  {
      byte value2 = (byte)dbl2;
  ```

```
    Console.WriteLine(value2);
}
catch (OverflowException)
{
    Console.WriteLine($"{dbl2} is out of range of a byte.");
}
// The example displays the following output:
//       128
//       3
```

- Convert 클래스의 메서드를 호출하여 지원되는 모든 형식을 Byte 값으로 변환할 수 있습니다. Byte IConvertible 인터페이스를 지원하므로 가능합니다. 다음 예제에서는 Int32 값 배열을 Byte 값으로 변환하는 방법을 보여 줍니다.

C#

```
int[] numbers = { Int32.MinValue, -1, 0, 121, 340, Int32.MaxValue };
byte result;
foreach (int number in numbers)
{
    try
    {
        result = Convert.ToByte(number);
        Console.WriteLine($"Converted the {number.GetType().Name} value
{number} to the {result.GetType().Name} value {result}.");
    }
    catch (OverflowException)
    {
        Console.WriteLine($"The {number.GetType().Name} value {number}
is outside the range of the Byte type.");
    }
}
// The example displays the following output:
//       The Int32 value -2147483648 is outside the range of the Byte
type.
//       The Int32 value -1 is outside the range of the Byte type.
//       Converted the Int32 value 0 to the Byte value 0.
//       Converted the Int32 value 121 to the Byte value 121.
//       The Int32 value 340 is outside the range of the Byte type.
//       The Int32 value 2147483647 is outside the range of the Byte
type.
```

- Parse 또는 TryParse 메서드를 호출하여 Byte 값의 문자열 표현을 Byte변환할 수 있습니다. 문자열은 10진수 또는 16진수를 포함할 수 있습니다. 다음 예제에서는 10진수 및 16진수 문자열을 모두 사용하여 구문 분석 작업을 보여 줍니다.

C#

```
string string1 = "244";
try
```

```
{
    byte byte1 = Byte.Parse(string1);
    Console.WriteLine(byte1);
}
catch (OverflowException)
{
    Console.WriteLine($"'{string1}' is out of range of a byte.");
}
catch (FormatException)
{
    Console.WriteLine($"'{string1}' is out of range of a byte.");
}

string string2 = "F9";
try
{
    byte byte2 = Byte.Parse(string2,

System.Globalization.NumberStyles.HexNumber);
    Console.WriteLine(byte2);
}
catch (OverflowException)
{
    Console.WriteLine($"'{string2}' is out of range of a byte.");
}
catch (FormatException)
{
    Console.WriteLine($"'{string2}' is out of range of a byte.");
}
// The example displays the following output:
//       244
//       249
```

바이트 값에 대한 작업 수행

Byte 형식은 더하기, 빼기, 나누기, 곱하기, 빼기, 부정 및 단항 부정과 같은 표준 수학 연산을 지원합니다. 다른 정수 계열 형식과 마찬가지로 Byte 형식은 비트 `AND`, `OR`, `XOR`, 왼쪽 시프트 및 오른쪽 시프트 연산자도 지원합니다.

표준 숫자 연산자를 사용하여 두 Byte 값을 비교하거나 CompareTo 또는 Equals 메서드를 호출할 수 있습니다.

또한 Math 클래스의 멤버를 호출하여 숫자의 절대값 가져오기, 정수 나누기에서 몫 및 나머지 계산, 두 정수의 최대값 또는 최소값 결정, 숫자 기호 가져오기 및 숫자 반올림 등 다양한 숫자 연산을 수행할 수 있습니다.

바이트를 문자열로 표현

Byte 형식은 표준 및 사용자 지정 숫자 형식 문자열을 완전히 지원합니다. (자세한 내용은 형식, 표준 숫자 서식 문자열 및 사용자 지정 숫자 서식 문자열 참조하세요.) 그러나 가장 일반적으로 바이트 값은 추가 서식 없이 1자리에서 3자리 값으로 표현되거나 두 자리 16진수 값으로 표시됩니다.

Byte 값을 선행 0이 없는 정수 문자열로 서식을 지정하려면 매개 변수가 없는 ToString() 메서드를 호출할 수 있습니다. "D" 형식 지정자를 사용하여 문자열 표현에 지정된 수의 선행 0을 포함할 수도 있습니다. "X" 형식 지정자를 사용하여 Byte 값을 16진수 문자열로 나타낼 수 있습니다. 다음 예제에서는 이러한 세 가지 방법으로 Byte 값 배열의 요소 서식을 지정합니다.

C#

```
byte[] numbers = { 0, 16, 104, 213 };
foreach (byte number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-3}  -->   ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write(number.ToString("D3") + "   ");
    // Display value with hexadecimal.
    Console.Write(number.ToString("X2") + "   ");
    // Display value with four hexadecimal digits.
    Console.WriteLine(number.ToString("X4"));
}
// The example displays the following output:
//       0    -->   000   00   0000
//       16   -->   016   10   0010
//       104  -->   104   68   0068
//       213  -->   213   D5   00D5
```

ToString(Byte, Int32) 메서드를 호출하고 밑을 메서드의 두 번째 매개 변수로 제공하여 Byte 값을 이진, 8진수, 10진수 또는 16진수 문자열로 서식을 지정할 수도 있습니다. 다음 예제에서는 바이트 값 배열의 이진, 8진수 및 16진수 표현을 표시하기 위해 이 메서드를 호출합니다.

C#

```
byte[] numbers = { 0, 16, 104, 213 };
Console.WriteLine("{0}   {1,8}   {2,5}   {3,5}",
                  "Value", "Binary", "Octal", "Hex");
foreach (byte number in numbers)
{
    Console.WriteLine("{0,5}   {1,8}   {2,5}   {3,5}",
                      number, Convert.ToString(number, 2),
                      Convert.ToString(number, 8),
                      Convert.ToString(number, 16));
}
// The example displays the following output:
```

```
//       Value     Binary   Octal     Hex
//           0          0       0       0
//          16      10000      20      10
//         104    1101000     150      68
//         213   11010101     325      d5
```

10진수가 아닌 바이트 값 사용

개별 바이트를 10진수 값으로 사용하는 것 외에도 바이트 값을 사용하여 비트 연산을 수행하거나 바이트 배열 또는 바이트 값의 이진 또는 16진수 표현으로 작업할 수 있습니다. 예를 들어 BitConverter.GetBytes 메서드의 오버로드는 각 기본 데이터 형식을 바이트 배열로 변환할 수 있으며 BigInteger.ToByteArray 메서드는 BigInteger 값을 바이트 배열로 변환합니다.

Byte 값은 부호 비트 없이 크기로만 8비트로 표시됩니다. 이는 Byte 값에 대해 비트 연산을 수행하거나 개별 비트로 작업할 때 유의해야 합니다. 소수점이 아닌 두 값에 대해 숫자, 부울 또는 비교 작업을 수행하려면 두 값 모두 동일한 표현을 사용해야 합니다.

두 Byte 값에 대해 작업을 수행하는 경우 값은 동일한 표현을 공유하므로 결과가 정확합니다. 다음 예제에서는 Byte 값의 가장 낮은 순서 비트를 마스크하여 짝수인지 확인합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { Convert.ToString(12, 16),
                          Convert.ToString(123, 16),
                          Convert.ToString(245, 16) };

      byte mask = 0xFE;
      foreach (string value in values) {
         Byte byteValue = Byte.Parse(value, NumberStyles.AllowHexSpecifier);
         Console.WriteLine($"{byteValue} And {mask} = {byteValue & mask}");
      }
   }
}
// The example displays the following output:
//       12 And 254 = 12
//       123 And 254 = 122
//       245 And 254 = 244
```

반면에 부호 없는 비트와 부호 있는 비트를 모두 사용하는 경우 비트 연산은 SByte 값이 양수 값에 대해 부호 및 크기 표현을 사용하고 음수 값에 대해 두 개의 보수 표현을 사용한다는 사실 때문에 복잡해집니다. 의미 있는 비트 연산을 수행하려면 값을 두 개의 동등한 표현으로 변환해야 하며 부호 비트에 대한 정보는 보존되어야 합니다. 다음 예제에서는 부호 있는 8비트 및 부호 없는 값 배열의 비트 2와 4를 마스킹합니다.

C#

```
using System;
using System.Collections.Generic;
using System.Globalization;

public struct ByteString
{
    public string Value;
    public int Sign;
}

public class Example1
{
    public static void Main()
    {
        ByteString[] values = CreateArray(-15, 123, 245);

        byte mask = 0x14;        // Mask all bits but 2 and 4.

        foreach (ByteString strValue in values)
        {
            byte byteValue = Byte.Parse(strValue.Value,
NumberStyles.AllowHexSpecifier);
            Console.WriteLine($"{strValue.Sign * byteValue}
({Convert.ToString(byteValue, 2)}) And {mask} ({Convert.ToString(mask, 2)})
= {(strValue.Sign & Math.Sign(mask)) * (byteValue & mask)}
({Convert.ToString(byteValue & mask, 2)})");
        }
    }

    private static ByteString[] CreateArray(params int[] values)
    {
        List<ByteString> byteStrings = new List<ByteString>();

        foreach (object value in values)
        {
            ByteString temp = new ByteString();
            int sign = Math.Sign((int)value);
            temp.Sign = sign;

            // Change two's complement to magnitude-only representation.
            temp.Value = Convert.ToString(((int)value) * sign, 16);

            byteStrings.Add(temp);
        }
        return byteStrings.ToArray();
```

```
    }
}
// The example displays the following output:
//       -15 (1111) And 20 (10100) = 4 (100)
//       123 (1111011) And 20 (10100) = 16 (10000)
//       245 (11110101) And 20 (10100) = 20 (10100)
```

System.Decimal 구조체

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

값 형식은 Decimal 양의 79,228,162,514,264,337,593,543,950,335에서 음의 79,228,162,514,264,337,593,543,950,335까지의 소수 숫자를 나타냅니다. 기본값은 `Decimal` 0입니다. 값 형식은 Decimal 많은 수의 정수 및 소수 자릿수가 필요하고 반올림 오류가 없는 재무 계산에 적합합니다. Decimal 형식은 반올림의 필요성을 없애지 않습니다. 대신 반올림으로 인한 오류를 최소화합니다. 예를 들어 다음 코드는 1이 아닌 0.9999999999999999999999999999의 결과를 생성합니다.

C#

```
decimal dividend = Decimal.One;
decimal divisor = 3;
// The following displays 0.99999999999999999999999999 to the console
Console.WriteLine(dividend/divisor * divisor);
```

나누기 및 곱셈의 결과가 메서드에 Round 전달되면 다음 코드와 같이 정밀도가 손실되지 않습니다.

C#

```
decimal dividend = Decimal.One;
decimal divisor = 3;
// The following displays 1.00 to the console
Console.WriteLine(Math.Round(dividend/divisor * divisor, 2));
```

10진수는 부호, 값의 각 숫자가 0에서 9 사이인 숫자 값, 숫자 값의 정수 부분과 소수 부분을 구분하는 부동 소수점 위치를 나타내는 배율 인수로 구성된 부동 소수점 값입니다.

값의 `Decimal` 이진 표현은 96비트 정수로 구성된 128비트 및 10진수 부분을 지정하는 데 사용되는 부호 및 배율 인수와 같은 항목을 나타내는 32비트 플래그 집합입니다. 따라서 값 Decimal의 이진 표현은 형식 ((-2^{96}에서 2^{96}) / $10^{(0 \sim 28)}$), 여기서 -(2^{96}-1)은 MinValue과 같고, 2^{96}-1은 MaxValue과 같습니다. Decimal 값의 이진 표현 및 예제에 대한 자세한 내용은 Decimal(Int32[]) 생성자와 GetBits 메서드를 참조하십시오.

배율 인수는 숫자 Decimal의 끝에 오는 0도 그대로 유지합니다. 후행 0은 산술 또는 비교 연산의 Decimal 숫자 값에 영향을 미치지 않습니다. 그러나 적절한 형식 문자열이 적용되는 경우 메서드에서 ToString 후행 0을 표시할 수 있습니다.

변환 고려 사항

이 형식은 Decimal 값을 SByte, Int16, Int32, Int64, Byte, UInt16, UInt32, 및 UInt64 값으로 변환하는 메서드를 제공합니다. 이러한 정수 계열 형식에서 Decimal 로의 변환은 정보를 손실하거나 예외를 발생시키지 않는 확장 변환입니다.

Decimal에서 정수형으로 변환하는 경우, Decimal 값을 0 방향으로 가장 가까운 정수 값으로 반올림하는 축소 변환이 됩니다. C#와 같은 일부 언어는 Decimal 값을 Char 값으로 변환하는 것도 지원합니다. 이러한 변환의 결과를 대상 형식으로 나타낼 수 없는 경우 예외가 OverflowException throw됩니다.

Decimal 형식은 또한 Decimal 값을 Single 및 Double 값으로 변환하는 메서드를 제공합니다. Decimal에서 Single 또는 Double로의 변환은 정밀도를 잃을 수 있는 축소 변환이지만, 변환된 값의 크기에 대한 정보는 손실되지 않습니다. 변환은 예외를 발생시키지 않습니다.

Single 또는 Double에서 Decimal로의 변환 결과를 OverflowException로 나타낼 수 없는 경우 Decimal 예외를 throw합니다.

10진수 값에 대한 작업 수행

이 형식은 Decimal 더하기, 빼기, 나누기, 곱하기 및 단항 부정과 같은 표준 수학 연산을 지원합니다. Decimal 메서드를 호출해서 GetBits 값의 이진 표현을 직접 다룰 수도 있습니다.

두 Decimal 값을 비교하려면 표준 숫자 비교 연산자를 사용하거나 또는 CompareTo 메서드를 Equals 호출할 수 있습니다.

클래스의 Math 멤버를 호출하여 숫자의 절대값 가져오기, 두 Decimal 값의 최대값 또는 최소값 결정, 숫자 기호 가져오기, 숫자 반올림 등 다양한 숫자 연산을 수행할 수도 있습니다.

예시

다음 코드 예제에서는 .의 Decimal사용을 보여 줍니다.

C#

```
/// <summary>
/// Keeping my fortune in Decimals to avoid the round-off errors.
/// </summary>
class PiggyBank {
    protected decimal MyFortune;

    public void AddPenny() {
        MyFortune = Decimal.Add(MyFortune, .01m);
    }
```

```
    public decimal Capacity {
        get {
            return Decimal.MaxValue;
        }
    }

    public decimal Dollars {
        get {
            return Decimal.Floor(MyFortune);
        }
    }

    public decimal Cents {
        get {
            return Decimal.Subtract(MyFortune, Decimal.Floor(MyFortune));
        }
    }

    public override string ToString() {
        return MyFortune.ToString("C")+" in piggy bank";
    }
}
```

System.Double 구조체

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

값 형식은 Double 음수 1.79769313486232e308에서 양수 1.79769313486232e308까지, 양수 또는 음수 0, PositiveInfinity, NegativeInfinity, 그리고 숫자가 아님(NaN)을 포함한 배정밀도 64비트 숫자를 나타냅니다. 매우 크거나(예: 행성이나 은하 사이의 거리) 또는 매우 작은 값(예: 킬로그램 단위의 물질의 분자 질량)을 나타내기 위한 것이며 종종 부정확합니다(예: 지구에서 다른 태양계까지의 거리). Double 형식은 이진 부동 소수점 산술 연산에 대한 IEC 60559:1989(IEEE 754) 표준을 준수합니다.

부동 소수점 표현 및 정밀도

데이터 형식은 Double 다음 표와 같이 배정밀도 부동 소수점 값을 64비트 이진 형식으로 저장합니다.

⛶ **테이블 확장**

부분	비트
가수(Significand) 또는 맨티사(mantissa)	0-51
지수	52-62
기호(0 = 양수, 1 = 음수)	63

소수 자릿수가 일부 분수 값(예: 1/3 또는 Math.PI)을 정확하게 나타낼 수 없는 것처럼 이진 분수는 일부 소수 값을 나타낼 수 없습니다. 예를 들어 .1로 정확하게 10진수로 표현되는 1/10은 .001100110011 이진 분수로 표현되며 패턴 "0011"은 무한대로 반복됩니다. 이 경우 부동 소수점 값은 나타내는 숫자의 부정확한 표현을 제공합니다. 원래 부동 소수점 값에 대해 추가 수학 연산을 수행하면 정밀도가 부족한 경우가 많습니다. 예를 들어 .1을 10으로 곱하고 .1을 .1에서 .1로 9번 추가한 결과를 비교하면 8개의 작업이 더 포함되었기 때문에 정확도가 낮은 결과가 생성되었음을 알 수 있습니다. .NET 10 이전에는 두 개의 Double 값을 "R" 표준 숫자 형식 문자열을 사용하여 표시할 때, Double 형식이 지원하는 최대 17자리의 ~전체~ 자릿수 정확도가 드러나면서 이러한 차이가 분명해집니다.

C#

```
using System;

public class Example13
{
    public static void Main()
    {
        Double value = .1;
        Double result1 = value * 10;
        Double result2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine($".1 * 10:           {result1:R}");
        Console.WriteLine($".1 Added 10 times: {result2:R}");
    }
}
// The example displays the following output:
//       .1 * 10:           1
//       .1 Added 10 times: 0.99999999999999989
```

일부 숫자는 소수점 이진 값으로 정확하게 나타낼 수 없으므로 부동 소수점 숫자는 실제 숫자와 근사치일 수 있습니다.

또한 모든 부동 소수점 숫자에는 제한된 수의 유효 자릿수가 있으며, 이는 부동 소수점 값이 실제 숫자와 근사치를 얼마나 정확하게 나타내는지도 결정합니다. Double 값의 전체 자릿수는 15자리이지만 내부적으로는 최대 17자리 자릿수가 유지됩니다. 즉, 일부 부동 소수점 연산에는 부동 소수점 값을 변경할 정밀도가 부족할 수 있습니다. 다음 예제에서 이에 대해 설명합니다. 매우 큰 부동 소수점 값을 정의한 다음, Double.Epsilon과 천조의 곱을 그것에 추가합니다. 그러나 제품이 너무 작아서 원래 부동 소수점 값을 수정할 수 없습니다. 가장 낮은 유효 자릿수는 천 번째인 반면 제품에서 가장 중요한 숫자는 10^{-309}입니다.

C#

```
using System;

public class Example14
{
    public static void Main()
    {
        Double value = 123456789012.34567;
        Double additional = Double.Epsilon * 1e15;
        Console.WriteLine($"{value} + {additional} = {value + additional}");
    }
}
// The example displays the following output:
//    123456789012.346 + 4.94065645841247E-309 = 123456789012.346
```

부동 소수점 숫자의 제한된 정밀도에는 다음과 같은 몇 가지 결과가 있습니다.

- 특정 정밀도에서는 동일하게 보이는 두 부동소수점 숫자가 가장 낮은 자리 숫자가 다르기 때문에 서로 같지 않을 수 있습니다. 다음 예제에서는 일련의 숫자가 함께 추가되고 해당 합계가 예상 합계와 비교됩니다.

  C#

```
using System;

public class Example10
{
    public static void Main()
    {
        Double[] values = { 10.0, 2.88, 2.88, 2.88, 9.0 };
        Double result = 27.64;
        Double total = 0;
        foreach (var value in values)
            total += value;

        if (total.Equals(result))
            Console.WriteLine("The sum of the values equals the total.");
        else
            Console.WriteLine($"The sum of the values ({total}) does not equal
the total ({result}).");
    }
}
// The example displays the following output:
//       The sum of the values (36.64) does not equal the total (36.64).
//
// If the index items in the Console.WriteLine statement are changed to {0:R},
// the example displays the following output:
//       The sum of the values (27.639999999999997) does not equal the total
(27.64).
```

  두 값은 더하기 작업 중에 정밀도가 손실되어 같지 않습니다. 이 경우 비교를 수행하기 전에 Math.Round(Double, Int32) 메서드를 호출하여 Double 값을 원하는 전체 자릿수로 반올림하여 문제를 해결할 수 있습니다.

- 부동 소수점 숫자를 사용하는 수학 또는 비교 연산은 이진 부동 소수점 숫자가 10진수와 같지 않을 수 있으므로 소수점 숫자를 사용하는 경우 동일한 결과를 생성하지 못할 수 있습니다. 이전 예제에서는 .1을 10으로 곱하고 .1을 곱한 결과를 표시하여 이를 설명했습니다.

  소수점 값을 가진 숫자 연산의 정확도가 중요한 경우 Decimal 형식을 사용하고 Double 형식은 사용하지 않는 것이 좋습니다. 정수 값이 형식 범위를 초과하는 숫자 연산의 Int128UInt128 정확도가 중요한 경우 형식을 BigInteger 사용합니다.

- 부동 소수점 숫자가 포함된 경우 값이 왕복 않을 수 있습니다. 연산이 원래의 부동 소수점 숫자를 다른 형태로 변환하고, 역연산이 그 변환된 형태를 다시 부동 소수점 숫자로 변환하며, 최종 부동 소수점 숫자가 원래의 부동 소수점 숫자와 같다면, 이러한 과정을 왕복 과정이라고 합니다. 변환에서 하나 이상의 유효 자릿수가 손실되거나 변경되어 왕복이 실패할 수 있습니다.

  다음 예제에서는 세 Double 값이 문자열로 변환되고 파일에 저장됩니다. .NET Framework에서 이 예제를 실행하면 값이 동일한 것처럼 보이지만 복원된 값은 원래 값과 같지 않습니다. 이후 .NET에서 값이 정확하게 왕복되도록 이 문제를 해결했습니다.

  C#

```
StreamWriter sw = new(@"./Doubles.dat");
double[] values = [2.2 / 1.01, 1.0 / 3, Math.PI];
for (int ctr = 0; ctr < values.Length; ctr++)
{
    sw.Write(values[ctr].ToString());
    if (ctr != values.Length - 1)
        sw.Write("|");
}
sw.Close();

double[] restoredValues = new double[values.Length];
StreamReader sr = new(@"./Doubles.dat");
string temp = sr.ReadToEnd();
string[] tempStrings = temp.Split('|');
for (int ctr = 0; ctr < tempStrings.Length; ctr++)
    restoredValues[ctr] = double.Parse(tempStrings[ctr]);

for (int ctr = 0; ctr < values.Length; ctr++)
Console.WriteLine($"{values[ctr]} {(values[ctr].Equals(restoredValues[ctr]) ?
"=" : "<>")} {restoredValues[ctr]}");

// For .NET Framework only, the example displays the following output:
//       2.17821782178218 <> 2.17821782178218
//       0.333333333333333 <> 0.333333333333333
//       3.14159265358979 <> 3.14159265358979
```

  .NET Framework를 대상으로 하는 경우 "G17" 표준 숫자 형식 문자열을 사용하여 Double 값의 전체 자릿수를 유지함으로써 성공적으로 라운드트립할 수 있습니다.

- Single 값은 Double 값보다 정밀도가 낮습니다. Single 값이 표면상 동등한 Double 값으로 변환되더라도, 정밀도의 차이로 인해 Double 값과 동일하지 않은 경우가 많습니다. 다음 예제에서는 동일한 나누기 작업의 결과가 a Double 및 Single 값에 할당됩니다. Single 값이 Double캐스팅된 후 두 값을 비교하면 같지 않음이 표시됩니다.

  C#

```
using System;

public class Example9
{
    public static void Main()
    {
        Double value1 = 1 / 3.0;
        Single sValue2 = 1 / 3.0f;
        Double value2 = (Double)sValue2;
        Console.WriteLine($"{value1:R} = {value2:R}:
{value1.Equals(value2)}");
    }
}
// The example displays the following output:
//        0.3333333333333331 = 0.3333333432674408: False
```

이 문제를 방지하려면 Double를 데이터 형식 Single 대신 사용하거나, 두 값의 정확도가 같아지도록 Round 메서드를 사용하십시오.

또한 Double 값을 사용한 산술 및 할당 연산의 결과는 Double 유형의 정밀도 손실로 인해 플랫폼에 따라 다소 차이가 있을 수 있습니다. 예를 들어 리터럴 Double 값을 할당한 결과는 32비트 및 64비트 버전의 .NET 다를 수 있습니다. 다음 예제에서는 리터럴 값 -4.423306042444772E-305와 값이 -4.423306042444772E-305인 변수가 변수에 Double 할당된 경우의 이 차이를 보여 줍니다. 이 경우 Parse(String) 메서드의 결과는 정밀도 손실이 없습니다.

C#

```
double value = -4.42330604244772E-305;

double fromLiteral = -4.42330604244772E-305;
double fromVariable = value;
double fromParse = Double.Parse("-4.42330604244772E-305");

Console.WriteLine("Double value from literal: {0,29:R}", fromLiteral);
Console.WriteLine("Double value from variable: {0,28:R}", fromVariable);
Console.WriteLine("Double value from Parse method: {0,24:R}", fromParse);
// On 32-bit versions of the .NET Framework, the output is:
//    Double value from literal:        -4.42330604244772E-305
//    Double value from variable:       -4.42330604244772E-305
//    Double value from Parse method:   -4.42330604244772E-305
//
// On other versions of the .NET Framework, the output is:
//    Double value from literal:      -4.4233060424477198E-305
//    Double value from variable:     -4.4233060424477198E-305
//    Double value from Parse method:   -4.42330604244772E-305
```

동등성 테스트

같게 간주하려면 두 Double 값이 동일한 값을 나타내야 합니다. 그러나 값 간의 정밀도 차이 또는 하나 또는 두 값 모두에 의한 정밀도 손실로 인해 동일할 것으로 예상되는 부동 소수점 값은 가장 낮은 유효 자릿수의 차이로 인해 같지 않은 것으로 판명되는 경우가 많습니다. 따라서 두 값이 같은지 여부를 확인하기 위해 Equals 메서드를 호출하거나 두 CompareTo 값 간의 관계를 확인하기 위해 Double 메서드를 호출하면 예기치 않은 결과가 발생하는 경우가 많습니다. 이는 다음 예제에서 분명하게 알 수 있습니다. 첫 번째 값은 15자리의 정밀도를 가지고 두 번째 값은 17자리이므로 동일한 두 Double 값이 같지 않은 것으로 판명됩니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      double value1 = .333333333333333;
      double value2 = 1.0/3;
      Console.WriteLine($"{value1} = {value2}: {value1.Equals(value2)}");
   }
}
// The example displays the following output:
//       0.333333333333333 = 0.3333333333333331: False
```

정밀도 손실이 비교 결과에 영향을 미칠 가능성이 있는 경우, Equals 또는 CompareTo 메서드를 호출하는 대신 다음 대안 중 하나를 채택할 수 있습니다.

- Math.Round 메서드를 호출하여 두 값의 정밀도가 같은지 확인합니다. 다음 예제에서는 두 개의 소수 값이 동일하도록 이 방법을 사용하도록 이전 예제를 수정합니다.

  C#

  ```
  double value1 = .333333333333333;
  double value2 = 1.0 / 3;
  int precision = 7;
  value1 = Math.Round(value1, precision);
  value2 = Math.Round(value2, precision);
  Console.WriteLine($"{value1} = {value2}: {value1.Equals(value2)}");

  // The example displays the following output:
  //       0.3333333 = 0.3333333: True
  ```

  정밀도 문제는 여전히 중간점 값의 반올림에 적용됩니다. 자세한 내용은 Math.Round(Double, Int32, MidpointRounding) 메서드를 참조하세요.

- 정확한 같음이 아닌 대략적인 같음을 테스트합니다. 이렇게 하려면 두 값이 다를 수 있지만 여전히 같을 수 있는 절대 크기를 정의하거나 더 작은 값이 더 큰 값과 다를 수 있는 상대 크기를 정의해야 합니다.

  > ⚠ **Warning**
  >
  > Double.Epsilon 같음을 테스트할 때 두 Double 값 사이의 거리를 절대 측정값으로 사용하는 경우가 있습니다. 그러나 Double.Epsilon는 값이 0인 Double에 더하거나 뺄 수 있는 가능한 가장 작은 값을 측정합니다. 대부분의 양수 및 음수 Double 값의 경우 Double.Epsilon 값이 너무 작아 검색할 수 없습니다. 따라서 0인 값을 제외하고 같음 테스트에는 사용하지 않는 것이 좋습니다.

  다음 예제에서는 후자의 방법을 사용하여 두 값 간의 상대적 차이를 테스트하는 `IsApproximatelyEqual` 메서드를 정의합니다. 이 메서드는 `Math.Max(value1, value2)`으로 나누어 두 값 중 절대값이 더 큰 값을 기준으로 상대적으로 비교하여, 결과가 올바른 크기의 순서에 놓이도록 합니다. 값 중 하나가 0이고 다른 하나가 음수인 경우 `Math.Max`가 0을 반환하면, 메서드는 `Math.Min(value1, value2)`로 대체되어 0이 아닌 값을 제수로 사용합니다. 또한 `IsApproximatelyEqual` 메서드 및 Equals(Double) 메서드에 대한 호출의 결과와 대조됩니다.

  C#

```
using System;

public class Example3
{
    public static void Main()
    {
        double one1 = .1 * 10;
        double one2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            one2 += .1;

        Console.WriteLine($"{one1} = {one2}: {one1.Equals(one2)}");
        Console.WriteLine($"{one1} is approximately equal to {one2}:
{IsApproximatelyEqual(one1, one2, .000000001)}");
    }

    static bool IsApproximatelyEqual(double value1, double value2, double
epsilon)
    {
        // If they are equal anyway, just return True.
        if (value1.Equals(value2))
            return true;
```

```
        // Handle NaN, Infinity.
        if (Double.IsInfinity(value1) | Double.IsNaN(value1))
            return value1.Equals(value2);
        else if (Double.IsInfinity(value2) | Double.IsNaN(value2))
            return value1.Equals(value2);

        // Handle zero to avoid division by zero
        double divisor = Math.Max(value1, value2);
        if (divisor.Equals(0))
            divisor = Math.Min(value1, value2);

        return Math.Abs((value1 - value2) / divisor) <= epsilon;
    }
}

// The example displays the following output:
//       1 = 0.99999999999999989: False
//       1 is approximately equal to 0.99999999999999989: True
```

부동 소수점 값 및 예외

정수 계열 형식의 연산은 0으로 나눌 때 DivideByZeroException 예외를, 오버플로 시 OverflowException 예외를 검사된 컨텍스트에서 throw하는 것과 달리, 부동 소수점 값과 관련된 연산은 예외를 throw하지 않습니다. 대신 예외적인 상황에서 부동 소수점 연산의 결과는 0, 양수 무한대, 음의 무한대 또는 숫자(NaN)가 아닙니다.

- 부동 소수점 연산의 결과가 대상 형식에 비해 너무 작으면 결과는 0입니다. 이 문제는 다음 예제와 같이 두 개의 매우 작은 숫자를 곱할 때 발생할 수 있습니다.

C#

```
using System;

public class Example6
{
    public static void Main()
    {
        Double value1 = 1.1632875981534209e-225;
        Double value2 = 9.1642346778e-175;
        Double result = value1 * value2;
        Console.WriteLine($"{value1} * {value2} = {result}");
        Console.WriteLine($"{result} = 0: {result.Equals(0.0)}");
    }
}
// The example displays the following output:
//       1.16328759815342E-225 * 9.1642346778E-175 = 0
//       0 = 0: True
```

- 부동 소수점 연산 결과의 크기가 대상 형식의 범위를 초과하는 경우, 결과의 부호에 따라 연산의 결과는 PositiveInfinity 또는 NegativeInfinity이 됩니다. 어떤 작업이 Double.MaxValue에서 오버플로하면 그 결과는 PositiveInfinity이고, Double.MinValue에서 오버플로하는 작업의 결과는 NegativeInfinity입니다. 이는 다음 예제에서 볼 수 있습니다.

  C#

```
using System;

public class Example7
{
    public static void Main()
    {
        Double value1 = 4.565e153;
        Double value2 = 6.9375e172;
        Double result = value1 * value2;
        Console.WriteLine($"PositiveInfinity:
{Double.IsPositiveInfinity(result)}");
        Console.WriteLine($"NegativeInfinity:
{Double.IsNegativeInfinity(result)}{Environment.NewLine}");

        value1 = -value1;
        result = value1 * value2;
        Console.WriteLine($"PositiveInfinity:
{Double.IsPositiveInfinity(result)}");
        Console.WriteLine($"NegativeInfinity:
{Double.IsNegativeInfinity(result)}");
    }
}

// The example displays the following output:
//       PositiveInfinity: True
//       NegativeInfinity: False
//
//       PositiveInfinity: False
//       NegativeInfinity: True
```

  PositiveInfinity 또한 양의 배수를 0으로 나누기에서 비롯된 결과이며, NegativeInfinity은 음의 배수를 0으로 나누기에서 비롯된 결과입니다.

- 부동 소수점 연산이 유효하지 않으면 그 연산의 결과는 NaN입니다. 예를 들어, 다음 연산의 결과는 NaN입니다.

  - 배당금이 0인 0으로 나누기. 다른 경우에 0으로 나누면 PositiveInfinity 또는 NegativeInfinity이 발생할 수 있음을 유의하십시오.

  - 잘못된 입력이 있는 모든 부동 소수점 작업. 예를 들어 음수 Math.Sqrt 값으로 메서드를 호출하면 1보다 크거나 음수 NaN 보다 작은 값으로 메서드를 호출하는 것과 마찬가지

로 반환Math.Acos됩니다.

- 값이 Double.NaN인수가 있는 모든 연산입니다.

형식 변환

Double 구조체는 명시적 또는 암시적 변환 연산자를 정의하지 않습니다. 대신 변환은 컴파일러에 의해 구현됩니다.

기본 숫자 형식의 값을 a Double 로 변환하는 것은 확대 변환이므로 컴파일러가 명시적으로 요구하지 않는 한 명시적 캐스트 연산자 또는 변환 메서드 호출이 필요하지 않습니다. 예를 들어 C# 컴파일러는 Decimal에서 Double 변환하는 데 캐스팅 연산자가 필요하지만 Visual Basic 컴파일러는 변환하지 않습니다. 다음 예제에서는 다른 기본 숫자 형식의 최소값 또는 최대값을 로 Double변환합니다.

C#

```
dynamic[] values = { Byte.MinValue, Byte.MaxValue, Decimal.MinValue,
                     Decimal.MaxValue, Int16.MinValue, Int16.MaxValue,
                     Int32.MinValue, Int32.MaxValue, Int64.MinValue,
                     Int64.MaxValue, SByte.MinValue, SByte.MaxValue,
                     Single.MinValue, Single.MaxValue, UInt16.MinValue,
                     UInt16.MaxValue, UInt32.MinValue, UInt32.MaxValue,
                     UInt64.MinValue, UInt64.MaxValue };
double dblValue;
foreach (dynamic value in values)
{
    if (value.GetType() == typeof(decimal))
        dblValue = (double)value;
    else
        dblValue = value;
    Console.WriteLine($"{value} ({value.GetType().Name}) --> " +
        $"{dblValue:R} ({dblValue.GetType().Name})");
}

// The example displays the following output:
//    0 (Byte) --> 0 (Double)
//    255 (Byte) --> 255 (Double)
//    -79228162514264337593543950335 (Decimal) --> -7.9228162514264338E+28 (Double)
//    79228162514264337593543950335 (Decimal) --> 7.9228162514264338E+28 (Double)
//    -32768 (Int16) --> -32768 (Double)
//    32767 (Int16) --> 32767 (Double)
//    -2147483648 (Int32) --> -2147483648 (Double)
//    2147483647 (Int32) --> 2147483647 (Double)
//    -9223372036854775808 (Int64) --> -9.2233720368547758E+18 (Double)
//    9223372036854775807 (Int64) --> 9.2233720368547758E+18 (Double)
//    -128 (SByte) --> -128 (Double)
//    127 (SByte) --> 127 (Double)
//    -3.402823E+38 (Single) --> -3.4028234663852886E+38 (Double)
```

```
//    3.402823E+38 (Single) --> 3.4028234663852886E+38 (Double)
//    0 (UInt16) --> 0 (Double)
//    65535 (UInt16) --> 65535 (Double)
//    0 (UInt32) --> 0 (Double)
//    4294967295 (UInt32) --> 4294967295 (Double)
//    0 (UInt64) --> 0 (Double)
//    18446744073709551615 (UInt64) --> 1.8446744073709552E+19 (Double)
```

또한 Single 값 Single.NaN, Single.PositiveInfinity및 Single.NegativeInfinity은 각각 Double.NaN, Double.PositiveInfinity및 Double.NegativeInfinity으로 변환됩니다.

일부 숫자 형식의 값을 Double 값으로 변환하면 정밀도가 떨어질 수 있습니다. 예시에서 설명하듯이 Decimal, Int64, 및 UInt64 값을 Double 값으로 변환할 때 정밀도 손실이 발생할 수 있습니다.

Double 값을 다른 기본 숫자 데이터 형식의 값으로 변환하려면 축소 변환이며 캐스트 연산자(C#), 변환 메서드(Visual Basic) 또는 Convert 메서드 호출이 필요합니다. 대상 형식의 `MinValue` 및 `MaxValue` 속성으로 정의된 대상 데이터 형식의 범위를 벗어난 값은 다음 표와 같이 동작합니다.

테이블 확장

대상 형식	결과
모든 정수 계열 형식	확인된 컨텍스트에서 변환이 발생하는 경우 OverflowException 예외입니다. 확인되지 않은 컨텍스트(C#의 기본값)에서 변환이 발생하면 변환 작업이 성공하지만 값이 오버플로됩니다.
Decimal	예외입니다 OverflowException .
Single	Single.NegativeInfinity 음수 값의 경우 Single.PositiveInfinity 양수 값에 대해.

또한 Double.NaN, Double.PositiveInfinity및 Double.NegativeInfinity는 확인된 컨텍스트에서 정수로 변환하기 위한 OverflowException을 던지지만, 이러한 값은 확인되지 않은 컨텍스트에서 정수로 변환될 때 오버플로가 발생합니다. Decimal변환을 할 때는 항상 OverflowException를 던집니다. Single변환의 경우 각각 Single.NaN, Single.PositiveInfinity및 Single.NegativeInfinity변환합니다.

정밀도 손실은 Double 값을 다른 숫자 형식으로 변환할 때 발생할 수 있습니다. 예제의 출력에서와 같이 정수 계열 형식으로 변환하는 경우 Double 값이 Visual Basic 반올림되거나 잘리면(C#에

서와 같이) 소수 구성 요소가 손실됩니다. 두 값 Decimal과 Single로 변환할 때, Double 값이 대상 데이터 형식에 정확하게 표현되지 않을 수 있습니다.

다음 예제에서는 여러 Double 값을 다른 여러 숫자 형식으로 변환합니다. 변환은 기본적으로 Visual Basic에서 발생하며, C#에서는 checked 키워드로 인해, F#에서는 Checked 모듈로 인해 발생합니다. 예제의 출력은 확인된 선택되지 않은 컨텍스트 모두에서 변환 결과를 보여 줍니다. Visual Basic에서 `/removeintchecks+` 컴파일러 스위치를 사용하여 컴파일하고, C#에서 `checked` 문을 주석 처리하고, F#에서 `open Checked` 문을 주석 처리하여 확인되지 않은 컨텍스트에서 변환을 수행할 수 있습니다.

C#

```
using System;

public class Example5
{
    public static void Main()
    {
        Double[] values = { Double.MinValue, -67890.1234, -12345.6789,
                            12345.6789, 67890.1234, Double.MaxValue,
                            Double.NaN, Double.PositiveInfinity,
                            Double.NegativeInfinity };
        checked
        {
            foreach (var value in values)
            {
                try
                {
                    Int64 lValue = (long)value;
                    Console.WriteLine($"{value} ({value.GetType().Name}) -->
{lValue} (0x{lValue:X16}) ({lValue.GetType().Name})");
                }
                catch (OverflowException)
                {
                    Console.WriteLine($"Unable to convert {value} to Int64.");
                }
                try
                {
                    UInt64 ulValue = (ulong)value;
                    Console.WriteLine($"{value} ({value.GetType().Name}) -->
{ulValue} (0x{ulValue:X16}) ({ulValue.GetType().Name})");
                }
                catch (OverflowException)
                {
                    Console.WriteLine($"Unable to convert {value} to UInt64.");
                }
                try
                {
                    Decimal dValue = (decimal)value;
                    Console.WriteLine($"{value} ({value.GetType().Name}) -->
{dValue} ({dValue.GetType().Name})");
```

```
                }
                catch (OverflowException)
                {
                    Console.WriteLine($"Unable to convert {value} to Decimal.");
                }
                try
                {
                    Single sValue = (float)value;
                    Console.WriteLine($"{value} ({value.GetType().Name}) -->
{sValue} ({sValue.GetType().Name})");
                }
                catch (OverflowException)
                {
                    Console.WriteLine($"Unable to convert {value} to Single.");
                }
                Console.WriteLine();
            }
        }
    }
}
// The example displays the following output for conversions performed
// in a checked context:
//       Unable to convert -1.79769313486232E+308 to Int64.
//       Unable to convert -1.79769313486232E+308 to UInt64.
//       Unable to convert -1.79769313486232E+308 to Decimal.
//       -1.79769313486232E+308 (Double) --> -Infinity (Single)
//
//       -67890.1234 (Double) --> -67890 (0xFFFFFFFFFFFEF6CE) (Int64)
//       Unable to convert -67890.1234 to UInt64.
//       -67890.1234 (Double) --> -67890.1234 (Decimal)
//       -67890.1234 (Double) --> -67890.13 (Single)
//
//       -12345.6789 (Double) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//       Unable to convert -12345.6789 to UInt64.
//       -12345.6789 (Double) --> -12345.6789 (Decimal)
//       -12345.6789 (Double) --> -12345.68 (Single)
//
//       12345.6789 (Double) --> 12345 (0x0000000000003039) (Int64)
//       12345.6789 (Double) --> 12345 (0x0000000000003039) (UInt64)
//       12345.6789 (Double) --> 12345.6789 (Decimal)
//       12345.6789 (Double) --> 12345.68 (Single)
//
//       67890.1234 (Double) --> 67890 (0x0000000000010932) (Int64)
//       67890.1234 (Double) --> 67890 (0x0000000000010932) (UInt64)
//       67890.1234 (Double) --> 67890.1234 (Decimal)
//       67890.1234 (Double) --> 67890.13 (Single)
//
//       Unable to convert 1.79769313486232E+308 to Int64.
//       Unable to convert 1.79769313486232E+308 to UInt64.
//       Unable to convert 1.79769313486232E+308 to Decimal.
//       1.79769313486232E+308 (Double) --> Infinity (Single)
//
//       Unable to convert NaN to Int64.
//       Unable to convert NaN to UInt64.
//       Unable to convert NaN to Decimal.
```

```
//       NaN (Double) --> NaN (Single)
//
//       Unable to convert Infinity to Int64.
//       Unable to convert Infinity to UInt64.
//       Unable to convert Infinity to Decimal.
//       Infinity (Double) --> Infinity (Single)
//
//       Unable to convert -Infinity to Int64.
//       Unable to convert -Infinity to UInt64.
//       Unable to convert -Infinity to Decimal.
//       -Infinity (Double) --> -Infinity (Single)
// The example displays the following output for conversions performed
// in an unchecked context:
//       -1.79769313486232E+308 (Double) --> -9223372036854775808
(0x8000000000000000) (Int64)
//       -1.79769313486232E+308 (Double) --> 9223372036854775808
(0x8000000000000000) (UInt64)
//       Unable to convert -1.79769313486232E+308 to Decimal.
//       -1.79769313486232E+308 (Double) --> -Infinity (Single)
//
//       -67890.1234 (Double) --> -67890 (0xFFFFFFFFFFFEF6CE) (Int64)
//       -67890.1234 (Double) --> 18446744073709483726 (0xFFFFFFFFFFFEF6CE)
(UInt64)
//       -67890.1234 (Double) --> -67890.1234 (Decimal)
//       -67890.1234 (Double) --> -67890.13 (Single)
//
//       -12345.6789 (Double) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//       -12345.6789 (Double) --> 18446744073709539271 (0xFFFFFFFFFFFFCFC7)
(UInt64)
//       -12345.6789 (Double) --> -12345.6789 (Decimal)
//       -12345.6789 (Double) --> -12345.68 (Single)
//
//       12345.6789 (Double) --> 12345 (0x0000000000003039) (Int64)
//       12345.6789 (Double) --> 12345 (0x0000000000003039) (UInt64)
//       12345.6789 (Double) --> 12345.6789 (Decimal)
//       12345.6789 (Double) --> 12345.68 (Single)
//
//       67890.1234 (Double) --> 67890 (0x0000000000010932) (Int64)
//       67890.1234 (Double) --> 67890 (0x0000000000010932) (UInt64)
//       67890.1234 (Double) --> 67890.1234 (Decimal)
//       67890.1234 (Double) --> 67890.13 (Single)
//
//       1.79769313486232E+308 (Double) --> -9223372036854775808
(0x8000000000000000) (Int64)
//       1.79769313486232E+308 (Double) --> 0 (0x0000000000000000) (UInt64)
//       Unable to convert 1.79769313486232E+308 to Decimal.
//       1.79769313486232E+308 (Double) --> Infinity (Single)
//
//       NaN (Double) --> -9223372036854775808 (0x8000000000000000) (Int64)
//       NaN (Double) --> 0 (0x0000000000000000) (UInt64)
//       Unable to convert NaN to Decimal.
//       NaN (Double) --> NaN (Single)
//
//       Infinity (Double) --> -9223372036854775808 (0x8000000000000000) (Int64)
//       Infinity (Double) --> 0 (0x0000000000000000) (UInt64)
```

```
//       Unable to convert Infinity to Decimal.
//       Infinity (Double) --> Infinity (Single)
//
//       -Infinity (Double) --> -9223372036854775808 (0x8000000000000000) (Int64)
//       -Infinity (Double) --> 9223372036854775808 (0x8000000000000000) (UInt64)
//       Unable to convert -Infinity to Decimal.
//       -Infinity (Double) --> -Infinity (Single)
```

숫자 형식의 변환에 대한 자세한 내용은 .NET 및 Type 변환 테이블의 형식 변환을 참조하세요.

부동 소수점 기능

구조체 및 관련 형식은 Double 다음 영역에서 작업을 수행하는 메서드를 제공합니다.

- 값의 비교입니다. Equals 메서드를 호출하여 두 Double 값이 같은지 또는 CompareTo 메서드를 호출하여 두 값 간의 관계를 확인할 수 있습니다.

  Double 구조체는 전체 비교 연산자 집합도 지원합니다. 예를 들어 같음 또는 같지 않음을 테스트하거나 한 값이 다른 값보다 크거나 같은지 확인할 수 있습니다. 피연산자 중 하나가 숫자 형식이 아닌 Double인 경우, 비교를 수행하기 전에 Double로 변환됩니다.

  > ⚠ **Warning**
  >
  > 정밀도의 차이로 인해 같을 것으로 예상되는 두 **Double** 값이 같지 않은 것으로 판명되어 비교 결과에 영향을 줄 수 있습니다. 두 **Double** 값을 비교하는 방법에 대한 자세한 내용은 '동일성 테스트' 섹션을 참조하세요.

  IsNaN, IsInfinity, IsPositiveInfinity및 IsNegativeInfinity 메서드를 호출하여 이러한 특수 값을 테스트할 수도 있습니다.

- **수학 연산**. 더하기, 빼기, 곱하기 및 나누기와 같은 일반적인 산술 연산은 메서드가 아닌 언어 컴파일러 및 CIL(공용 중간 언어) 명령에 의해 Double 구현됩니다. 수학 연산의 피연산자 중 하나가 Double가 아닌 다른 숫자 형식인 경우, 연산을 수행하기 전에 Double로 변환됩니다. 작업의 결과도 Double 값입니다.

  다른 수학 연산은 `static` 클래스에서 `Shared`(Visual Basic System.Math) 메서드를 호출하여 수행할 수 있습니다. 여기에는 산술(예: Math.Abs, Math.Sign 및 Math.Sqrt), 기하(예: Math.Cos 및 Math.Sin), 그리고 미적분(예: Math.Log)에 일반적으로 사용되는 추가적인 메서드가 포함됩니다.

  Double 값의 개별 비트를 조작할 수도 있습니다. 이 메서드는 BitConverter.DoubleToInt64Bits 값의 비트 패턴을 64비트 정수로 유지합니다 Double .

BitConverter.GetBytes(Double) 메서드는 바이트 배열에서 비트 패턴을 반환합니다.

- **반올림**. 반올림은 부동 소수점 표현과 정밀도 문제로 인해 발생하는 값 간의 차이를 줄이는 기술로 자주 사용됩니다. Double 메서드를 호출하여 Math.Round 값을 반올림할 수 있습니다.

- **서식**. 값을 문자열로 변환하려면 Double 메서드를 호출하거나 복합 서식 지정 기능을 사용하세요. 서식 문자열이 부동 소수점 값의 문자열 표현을 제어하는 방법에 대한 자세한 내용은 표준 숫자 형식 문자열 및 사용자 지정 숫자 형식 문자열을 참조하세요.

- **문자열**구문 분석. 당신은 Double 또는 Parse 메서드를 호출하여 부동 소수점 값의 문자열 표현을 TryParse 값으로 변환할 수 있습니다. 구문 분석 작업이 실패하면 Parse 메서드는 예외를 throw하는 반면 TryParse 메서드는 `false`반환합니다.

- **형식 변환**. Double 구조체는 두 표준 .NET 데이터 형식 간의 변환을 지원하는 IConvertible 인터페이스에 대한 명시적 인터페이스 구현을 제공합니다. 언어 컴파일러에서는 다른 모든 표준 숫자 형식의 값을 Double 값으로 암시적으로 변환하는 것도 지원합니다. 표준 숫자 형식의 값을 a Double 로 변환하는 것은 확대 변환이며 캐스팅 연산자 또는 변환 메서드의 사용자가 필요하지 않습니다.

  그러나 Int64와 Single 값의 변환에는 정밀도 손실이 발생할 수 있습니다. 다음 표에서는 이러한 각 형식의 정밀도 차이를 보여줍니다.

테이블 확장

유형	최대 정밀도	내부 정밀도
Double	15	17
Int64	19자리 십진수	19자리 십진수
Single	10진수 7자리	10진수 9자리

정밀도 문제는 Single 값으로 변환되는 Double 값에 가장 자주 영향을 줍니다. 다음 예제에서는 값 중 하나가 단정밀도 부동 소수점 값 Double이므로 동일한 나누기 연산에서 생성된 두 값은 같지 않습니다.

C#

```
using System;

public class Example13
{
    public static void Main()
```

```
    {
        Double value = .1;
        Double result1 = value * 10;
        Double result2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine($".1 * 10:           {result1:R}");
        Console.WriteLine($".1 Added 10 times: {result2:R}");
    }
}
// The example displays the following output:
//       .1 * 10:           1
//       .1 Added 10 times: 0.99999999999999989
```

예시

다음 코드 예제에서는 다음을 Double사용하는 방법을 보여 줍니다.

C#

```
// The Temperature class stores the temperature as a Double
// and delegates most of the functionality to the Double
// implementation.
public class Temperature : IComparable, IFormattable
{
    // IComparable.CompareTo implementation.
    public int CompareTo(object obj) {
        if (obj == null) return 1;

        Temperature temp = obj as Temperature;
        if (obj != null)
            return m_value.CompareTo(temp.m_value);
        else
            throw new ArgumentException("object is not a Temperature");
    }

    // IFormattable.ToString implementation.
    public string ToString(string format, IFormatProvider provider) {
        if( format != null ) {
            if( format.Equals("F") ) {
                return String.Format("{0}'F", this.Value.ToString());
            }
            if( format.Equals("C") ) {
                return String.Format("{0}'C", this.Celsius.ToString());
            }
        }

        return m_value.ToString(format, provider);
    }

    // Parses the temperature from a string in the form
```

```
    // [ws][sign]digits['F|'C][ws]
    public static Temperature Parse(string s, NumberStyles styles, IFormatProvider
provider) {
        Temperature temp = new Temperature();

        if( s.TrimEnd(null).EndsWith("'F") ) {
            temp.Value = Double.Parse( s.Remove(s.LastIndexOf('\''), 2), styles,
provider);
        }
        else if( s.TrimEnd(null).EndsWith("'C") ) {
            temp.Celsius = Double.Parse( s.Remove(s.LastIndexOf('\''), 2), styles,
provider);
        }
        else {
            temp.Value = Double.Parse(s, styles, provider);
        }

        return temp;
    }

    // The value holder
    protected double m_value;

    public double Value {
        get {
            return m_value;
        }
        set {
            m_value = value;
        }
    }

    public double Celsius {
        get {
            return (m_value-32.0)/1.8;
        }
        set {
            m_value = 1.8*value+32.0;
        }
    }
}
```

Last updated on 2026. 04. 07.

System.Double.CompareTo 메서드

아티클 • 2025. 03. 23.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

CompareTo(Double) 메서드

값이 동일해야 동등하다고 간주됩니다. 특히 부동 소수점 값이 여러 수학 연산에 의존하는 경우 정밀도를 상실하는 것이 일반적이며, 가장 적은 유효 자릿수를 제외하고 값이 거의 동일해야 합니다. 이 때문에 때때로 CompareTo 메서드의 반환 값은 놀라운 것처럼 보일 수 있습니다. 예를 들어, 특정 값을 곱한 다음 같은 값으로 나누기를 하면 원래 값이 됩니다. 그러나 다음 예제에서는 계산된 값이 원래 값보다 큰 것으로 나타났습니다. "R" 표준 숫자 형식 문자열 사용하여 두 값의 모든 유효 자릿수를 표시하면 계산된 값이 가장 낮은 유효 자릿수의 원래 값과 다르다는 것을 나타냅니다. 이러한 비교를 처리하는 방법에 대한 자세한 내용은 Equals(Double) 메서드의 설명 섹션을 참조하세요.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        double value1 = 6.185;
        double value2 = value1 * .1 / .1;
        Console.WriteLine($"Comparing {value1} and {value2}:
{value1.CompareTo(value2)}{Environment.NewLine}");
        Console.WriteLine($"Comparing {value1:R} and {value2:R}:
{value1.CompareTo(value2)}");
    }
}
// The example displays the following output:
//       Comparing 6.185 and 6.185: -1
//
//       Comparing 6.185 and 6.1850000000000005: -1
```

이 메서드는 System.IComparable<T> 인터페이스를 구현하고 `value` 매개 변수를 개체로 변환할 필요가 없으므로 Double.CompareTo 메서드보다 약간 더 잘 수행됩니다.

값이 NaN인 개체는 값이 NaN인 다른 개체와 (자신과도) 같지 않지만, IComparable<T> 인터페이스는 `A.CompareTo(A)`이 0을 반환해야 한다고 요구합니다.

CompareTo(Object) 메서드

`value` 매개 변수는 `null` 또는 `Double`; 인스턴스여야 합니다. 그렇지 않으면 예외가 throw됩니다. 값에 관계없이 Double인스턴스는 `null`보다 큰 것으로 간주됩니다.

값이 같다고 간주되려면 동일해야 합니다. 특히 부동 소수점 값이 여러 수학 연산에 의존하는 경우 정밀도를 상실하는 것이 일반적이며, 가장 적은 유효 자릿수를 제외하고 값이 거의 동일해야 합니다. 이 때문에 때때로 CompareTo 메서드의 반환 값은 놀라운 것처럼 보일 수 있습니다. 예를 들어, 특정 값으로 곱한 후 동일한 값으로 나누면 원래 값이 됩니다. 그러나 다음 예제에서는 계산된 값이 원래 값보다 큰 것으로 나타났습니다. "R" 표준 숫자 형식 문자열 사용하여 두 값의 모든 유효 자릿수를 표시하면 계산된 값이 가장 낮은 유효 자릿수의 원래 값과 다르다는 것을 나타냅니다. 이러한 비교를 처리하는 방법에 대한 자세한 내용은 Equals(Double) 메서드의 설명 섹션을 참조하세요.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        double value1 = 6.185;
        object value2 = value1 * .1 / .1;
        Console.WriteLine($"Comparing {value1} and {value2}:
{value1.CompareTo(value2)}{Environment.NewLine}");
        Console.WriteLine($"Comparing {value1:R} and {value2:R}:
{value1.CompareTo(value2)}");
    }
}
// The example displays the following output:
//       Comparing 6.185 and 6.185: -1
//
//       Comparing 6.185 and 6.1850000000000005: -1
```

이 메서드는 IComparable 인터페이스를 지원하도록 구현됩니다. 참고로, NaN는 다른 NaN와 같지 않으며, 심지어 자신과도 같지 않지만, IComparable 인터페이스는 `A.CompareTo(A)`이 0을 반환하도록 요구합니다.

확대 변환

프로그래밍 언어에 따라 매개 변수 형식이 인스턴스 형식보다 적은 비트(더 좁은)가 있는 CompareTo 메서드를 코딩할 수 있습니다. 일부 프로그래밍 언어는 인스턴스만큼 많은 비트가 있는 형식으로 매개 변수를 나타내는 암시적 확대 변환을 수행하기 때문에 가능합니다.

예를 들어 인스턴스 형식이 Double 매개 변수 형식이 Int32가정해 보겠습니다. Microsoft C# 컴파일러는 매개 변수 값을 Double 개체로 나타내는 명령을 생성한 다음 인스턴스의 값과 매개 변수의 확장된 표현을 비교하는 Double.CompareTo(Double) 메서드를 생성합니다.

프로그래밍 언어 설명서를 참조하여 컴파일러가 숫자 형식의 암시적 확대 변환을 수행하는지 확인합니다. 자세한 내용은 형식 변환 테이블 항목을 참조하세요.

비교의 정밀도

문서화된 전체 자릿수를 초과하는 부동 소수점 숫자의 정밀도는 .NET의 구현 및 버전과 관련이 있습니다. 따라서 숫자의 내부 표현의 정밀도가 변경될 수 있기 때문에 .NET 버전 간에 두 개의 특정 숫자를 비교하면 결과가 달라질 수 있습니다.

System.Double.Equals 메서드

메서드는 Double.Equals(Double) 인터페이스를 구현하고, 매개 변수를 개체로 변환할 필요가 없기 때문에 System.IEquatable<T>가 Double.Equals(Object)보다 약간 더 잘 수행됩니다.

확대 변환

프로그래밍 언어에 따라 인스턴스 형식보다 매개 변수 형식의 비트 수가 적은(더 좁은) 메서드를 Equals 코딩할 수 있습니다. 일부 프로그래밍 언어는 인스턴스만큼 많은 비트가 있는 형식으로 매개 변수를 나타내는 암시적 확대 변환을 수행하기 때문에 가능합니다.

예를 들어 인스턴스 형식이 Double 고 매개 변수 형식이 .입니다 Int32. Microsoft C# 컴파일러는 매개 변수의 값을 개체로 Double 나타내는 지침을 생성한 다음 인스턴스의 값과 매개 변수의 확장된 표현을 비교하는 메서드를 생성 Double.Equals(Double) 합니다.

프로그래밍 언어 설명서를 참조하여 컴파일러가 숫자 형식의 암시적 확대 변환을 수행하는지 확인합니다. 자세한 내용은 형식 변환 테이블 항목을 참조 하세요 .

비교의 정밀도

Equals 메서드는 주의해서 사용해야 합니다. 이는 두 값이 명목상 동일해 보여도 정밀도가 달라서 동일하지 않을 수 있기 때문입니다. 다음 예제에서는 Double 값 .333333과 1을 3으로 나누어 반환된 Double 값이 같지 않음을 보고합니다.

C#

```
// Initialize two doubles with apparently identical values
double double1 = .33333;
double double2 = (double) 1/3;
// Compare them for equality
Console.WriteLine(double1.Equals(double2));    // displays false
```

같음을 비교하는 대신, 한 가지 기술에는 두 값 간의 허용되는 상대 차이 여백(예: 값 중 하나의 .001%)을 정의하는 작업이 포함됩니다. 두 값 간의 차이의 절대값이 해당 여백보다 작거나 같으면 정밀도 차이로 인해 차이가 발생할 수 있으므로 값이 같을 가능성이 높습니다. 다음 예제에서는 이 기술을 사용하여 이전 코드 예제에서 같지 않은 것으로 확인된 두 Double 값인 .33333과 1/3을 비교합니다. 이 경우 값은 같습니다.

C#

```
// Initialize two doubles with apparently identical values
double double1 = .333333;
double double2 = (double) 1/3;
// Define the tolerance for variation in their values
```

```
double difference = Math.Abs(double1 * .00001);

// Compare the values
// The output to the console indicates that the two values are equal
if (Math.Abs(double1 - double2) <= difference)
   Console.WriteLine("double1 and double2 are equal.");
else
   Console.WriteLine("double1 and double2 are unequal.");
```

> ⓘ **참고 항목**
>
> Epsilon 범위가 0에 가까운 양수 값의 최소 식을 정의하므로 비슷한 두 값 사이의 차이 여백은 0보다 Epsilon커야 합니다. 일반적으로 Epsilon보다 여러 배 더 큽니다. 이 때문에 를 사용하여 Epsilon 값을 비교할 때는 같음 비교에 사용하지 않기를 권장합니다.

두 번째 기술에는 두 부동 소수점 숫자의 차이를 일부 절대값과 비교하는 작업이 포함됩니다. 차이가 절대값보다 작거나 같으면 숫자는 같습니다. 값이 크면 숫자가 같지 않습니다. 한 가지 대안은 임의로 절대값을 선택하는 것입니다. 그러나 허용되는 차이의 여백은 값의 Double 크기에 따라 달라지므로 문제가 됩니다. 두 번째 대안은 부동 소수점 형식의 디자인 기능을 활용합니다. 두 부동 소수점 값의 정수 표현 간의 차이는 부동 소수점 값을 구분하는 가능한 부동 소수점 값의 수를 나타냅니다. 예를 들어, 0.0과 Epsilon의 차이는 1입니다. 이는 값이 0인 Epsilon에서 작업할 때 Double이 가장 작은 표현 가능한 값이기 때문입니다. 다음 예제에서는 이 기술을 사용하여 이전 코드 예제와 같지 않은 것으로 확인된 메서드의 두 Double 값인 .33333과 Equals(Double) 1/3을 비교합니다. 이 예제에서는 메서드를 BitConverter.DoubleToInt64Bits 사용하여 배정밀도 부동 소수점 값을 정수 표현으로 변환합니다. 이 예제에서는 정수 표현 사이에 가능한 부동 소수점 값이 없는 경우 값을 같게 선언합니다.

C#

```
public static void Main()
{
    // Initialize the values.
    double value1 = .1 * 10;
    double value2 = 0;
    for (int ctr = 0; ctr < 10; ctr++)
        value2 += .1;

    Console.WriteLine($"{value1:R} = {value2:R}: " +
        $"{HasMinimalDifference(value1, value2, 1)}");
}

public static bool HasMinimalDifference(
    double value1,
    double value2,
    int allowableDifference
    )
{
```

```
    // Convert the double values to long values.
    long lValue1 = BitConverter.DoubleToInt64Bits(value1);
    long lValue2 = BitConverter.DoubleToInt64Bits(value2);

    // If the signs are different, return false except for +0 and -0.
    if ((lValue1 >> 63) != (lValue2 >> 63))
    {
        if (value1 == value2)
            return true;

        return false;
    }

    // Calculate the number of possible
    // floating-point values in the difference.
    long diff = Math.Abs(lValue1 - lValue2);

    if (diff <= allowableDifference)
        return true;

    return false;
}
// The example displays the following output:
//
//        1 = 0.99999999999999989: True
```

> ⓘ **참고 항목**
>
> 일부 값의 경우 정수 표현 사이에 가능한 부동 소수점 값이 *있는* 경우에도 동일하게 간주할 수 있습니다. 예를 들어, `0.39` 및 `1.69 - 1.3`이라는 double 값을 `0.3899999999999999`로 계산된다고 고려합니다. little-endian 컴퓨터에서 이러한 값의 정수 표현은 각각 `4600697235336603894` 및 `4600697235336603892`입니다. 정수 값 간의 차이는 입니다. 이는 와 사이에 가능한 부동 소수점 값이 있을 수 있음을 의미합니다.

버전 차이점

문서화된 전체 자릿수를 초과하는 부동 소수점 숫자의 정밀도는 .NET의 구현 및 버전과 관련이 있습니다. .NET의 버전에 따라 숫자의 내부 표현의 정밀도가 변경될 수 있으므로, 특정 두 숫자의 비교 결과가 달라질 수 있습니다.

NaN

두 Double.NaN 값을 메서드 Equals를 호출하여 동일한지 테스트하면, 메서드가 `true`를 반환합니다. 그러나 두 Double.NaN 값이 *같음 연산자*를 사용하여 같은지 테스트하면, 연산자가 `false`

를 반환합니다. 값 Double 이 숫자(NaN)가 아닌지 확인하려는 경우 메서드를 호출 IsNaN 하는 것이 대안입니다.

Last updated on 2026. 02. 12.

System.Double.Epsilon 속성

아티클 • 2025. 03. 29.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Epsilon 속성 값은 Double 인스턴스의 값이 0일 경우 숫자 연산 또는 비교에서 중요한 가장 작은 양의 Double 값을 반영합니다. 예를 들어 다음 코드에서는 0과 Epsilon 같지 않은 값으로 간주되는 반면, Epsilon 값의 0과 절반은 같은 것으로 간주됩니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      double[] values = { 0, Double.Epsilon, Double.Epsilon * .5 };

      for (int ctr = 0; ctr <= values.Length - 2; ctr++)
      {
         for (int ctr2 = ctr + 1; ctr2 <= values.Length - 1; ctr2++)
         {
            Console.WriteLine($"{values[ctr]:r} = {values[ctr2]:r}:
{values[ctr].Equals(values[ctr2])}");
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       0 = 4.94065645841247E-324: False
//       0 = 0: True
//
//       4.94065645841247E-324 = 0: False
```

보다 정확하게 말하면 부동 소수점 형식은 부호, 52비트 매니사 또는 significand 및 11비트 지수로 구성됩니다. 다음 예제에서 알 수 있듯이 0에는 지수가 -1022 및 매니티사는 0입니다. Epsilon은 지수가 -1022이고 유효숫자가 1입니다. 즉, Epsilon는 0보다 큰 가장 작은 양의 Double 값이며, Double의 지수가 -1022인 경우 가능한 가장 작은 값과 가장 작은 증분을 나타냅니다.

C#

```
using System;

public class Example1
{
```

```
    public static void Main()
    {
        double[] values = { 0.0, Double.Epsilon };
        foreach (var value in values)
        {
            Console.WriteLine(GetComponentParts(value));
            Console.WriteLine();
        }
    }

    private static string GetComponentParts(double value)
    {
        string result = String.Format("{0:R}: ", value);
        int indent = result.Length;

        // Convert the double to an 8-byte array.
        byte[] bytes = BitConverter.GetBytes(value);
        // Get the sign bit (byte 7, bit 7).
        result += String.Format("Sign: {0}\n",
                                (bytes[7] & 0x80) == 0x80 ? "1 (-)" : "0
(+)");

        // Get the exponent (byte 6 bits 4-7 to byte 7, bits 0-6)
        int exponent = (bytes[7] & 0x07F) << 4;
        exponent = exponent | ((bytes[6] & 0xF0) >> 4);
        int adjustment = exponent != 0 ? 1023 : 1022;
        result += String.Format("{0}Exponent: 0x{1:X4} ({1})\n", new
String(' ', indent), exponent - adjustment);

        // Get the significand (bits 0-51)
        long significand = ((bytes[6] & 0x0F) << 48);
        significand = significand | ((long)bytes[5] << 40);
        significand = significand | ((long)bytes[4] << 32);
        significand = significand | ((long)bytes[3] << 24);
        significand = significand | ((long)bytes[2] << 16);
        significand = significand | ((long)bytes[1] << 8);
        significand = significand | bytes[0];
        result += String.Format("{0}Mantissa: 0x{1:X13}\n", new String(' ',
indent), significand);

        return result;
    }
}
//       // The example displays the following output:
//       0: Sign: 0 (+)
//          Exponent: 0xFFFFFC02 (-1022)
//          Mantissa: 0x0000000000000
//
//
//       4.94065645841247E-324: Sign: 0 (+)
//                              Exponent: 0xFFFFFC02 (-1022)
//                              Mantissa: 0x0000000000001
```

그러나 Epsilon 속성은 Double 형식의 보편적인 정밀도 측정 수단이 아닙니다. 값이 0이거나 지수가 -1022인 Double 인스턴스에만 적용됩니다.

> ⓘ **참고**
>
> Epsilon 속성의 값은 부동 소수점 산술 연산의 반올림으로 인한 상대 오차의 상한을 나타내는 컴퓨터 엡실론과 동일하지 않습니다.

이 상수의 값은 4.94065645841247e-324입니다.

두 개의 분명히 동등한 부동 소수점 숫자는 가장 낮은 유효 자릿수의 차이로 인해 같지 않을 수 있습니다. 예를 들어, C# 식(`(double)1/3 == (double)0.33333`)은 왼쪽의 나누기 연산이 최대 정밀도를 가지지만, 오른쪽의 상수는 정확도가 지정된 자릿수로 제한되어 있기 때문에 동일하게 비교되지 않습니다. 두 부동 소수점 숫자를 같게 간주할 수 있는지 여부를 결정하는 사용자 지정 알고리즘을 만드는 경우 Epsilon 상수의 값을 기반으로 알고리즘을 기반으로 두 값이 같은 것으로 간주될 수 있는 절대 차이 여백을 설정하는 것은 권장되지 않습니다. (일반적으로 차이의 여백은 Epsilon.보다 여러 배 더 큽니다.) 두 개의 배정밀도 부동 소수점 값을 비교하는 방법에 대한 자세한 내용은 Double 및 Equals(Double)참조하세요.

플랫폼 참고 사항

ARM 시스템에서는 Epsilon 상수의 값이 너무 작아 감지할 수 없으므로 0과 같습니다. 대신 2.2250738585072014E-308과 같은 대체 엡실론 값을 정의할 수 있습니다.

System.Int32 구조체

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Int32는 음수 -2,147,483,648(Int32.MinValue 상수로 표시됨)부터 양수 2,147,483,647(Int32.MaxValue 상수로 표시됨)까지의 값을 가지는 부호 있는 정수를 나타내는 불변의 값 형식입니다. .NET에는 0에서 4,294,967,295까지의 값을 나타내는 부호 없는 32비트 정수 값 형식 UInt32도 포함됩니다.

Int32 값 인스턴스화

다음과 같은 Int32 여러 가지 방법으로 값을 인스턴스화할 수 있습니다.

- 변수를 Int32 선언하고 데이터 형식 범위 내에 있는 리터럴 정수 값을 할당할 Int32 수 있습니다. 다음 예제에서는 두 개의 Int32 변수를 선언하고 이러한 방식으로 값을 할당합니다.

  ```
  C#

  int number1 = 64301;
  int number2 = 25548612;
  ```

- 사용자는 범위가 Int32 형식의 하위 집합인 정수 형식 값을 할당할 수 있습니다. 이는 C#의 캐스트 연산자나 Visual Basic의 변환 메서드가 필요하지 않지만 F#에 캐스트 연산자가 필요한 확대 변환입니다.

  ```
  C#

  sbyte value1 = 124;
  short value2 = 1618;

  int number1 = value1;
  int number2 = value2;
  ```

- 범위가 Int32 형식의 범위를 초과하는 숫자 형식 값을 할당할 수 있습니다. 축소 변환이므로 C# 또는 F#의 캐스트 연산자와 Visual Basic의 변환 메서드(있는 경우 `Option Strict`)가 필요합니다. 숫자 값이 SingleDouble소수 구성 요소를 포함하는 값이거나 Decimal 소수 부분의 처리는 변환을 수행하는 컴파일러에 따라 달라집니다. 다음 예제에서는 축소 변환을 수행하여 여러 숫자 값을 변수에 Int32 할당합니다.

C#

```
long lNumber = 163245617;
try {
   int number1 = (int) lNumber;
   Console.WriteLine(number1);
}
catch (OverflowException) {
   Console.WriteLine($"{lNumber} is out of range of an Int32.");
}

double dbl2 = 35901.997;
try {
   int number2 = (int) dbl2;
   Console.WriteLine(number2);
}
catch (OverflowException) {
   Console.WriteLine($"{dbl2} is out of range of an Int32.");
}

BigInteger bigNumber = 132451;
try {
   int number3 = (int) bigNumber;
   Console.WriteLine(number3);
}
catch (OverflowException) {
   Console.WriteLine($"{bigNumber} is out of range of an Int32.");
}
// The example displays the following output:
//       163245617
//       35902
//       132451
```

- 클래스의 Convert 메서드를 호출하여 지원되는 모든 형식을 값으로 Int32 변환할 수 있습니다. Int32 IConvertible 인터페이스를 지원하므로 가능합니다. 다음 예제에서는 Decimal 값 배열을 Int32 값으로 변환하는 방법을 보여 줍니다.

C#

```
decimal[] values= { Decimal.MinValue, -1034.23m, -12m, 0m, 147m,
                    199.55m, 9214.16m, Decimal.MaxValue };
int result;

foreach (decimal value in values)
{
   try {
      result = Convert.ToInt32(value);
      Console.WriteLine($"Converted the {value.GetType().Name} value
'{value}' to the {result.GetType().Name} value {result}.");
   }
   catch (OverflowException) {
      Console.WriteLine($"{value} is outside the range of the Int32
```

```
type.");
   }
}
// The example displays the following output:
//    -79228162514264337593543950335 is outside the range of the Int32
type.
//    Converted the Decimal value '-1034.23' to the Int32 value -1034.
//    Converted the Decimal value '-12' to the Int32 value -12.
//    Converted the Decimal value '0' to the Int32 value 0.
//    Converted the Decimal value '147' to the Int32 value 147.
//    Converted the Decimal value '199.55' to the Int32 value 200.
//    Converted the Decimal value '9214.16' to the Int32 value 9214.
//    79228162514264337593543950335 is outside the range of the Int32
type.
```

- Parse 또는 TryParse 메서드를 호출하여 Int32 값의 문자열 표현을 Int32으로 변환할 수 있습니다. 문자열은 10진수 또는 16진수를 포함할 수 있습니다. 다음 예제에서는 10진수 및 16진수 문자열을 모두 사용하여 구문 분석 작업을 보여 줍니다.

C#

```
string string1 = "244681";
try {
   int number1 = Int32.Parse(string1);
   Console.WriteLine(number1);
}
catch (OverflowException) {
   Console.WriteLine($"'{string1}' is out of range of a 32-bit
integer.");
}
catch (FormatException) {
   Console.WriteLine($"The format of '{string1}' is invalid.");
}

string string2 = "F9A3C";
try {
   int number2 = Int32.Parse(string2,

System.Globalization.NumberStyles.HexNumber);
   Console.WriteLine(number2);
}
catch (OverflowException) {
   Console.WriteLine($"'{string2}' is out of range of a 32-bit
integer.");
}
catch (FormatException) {
   Console.WriteLine($"The format of '{string2}' is invalid.");
}
// The example displays the following output:
//       244681
//       1022524
```

Int32 값에 대한 작업 수행

이 형식은 Int32 더하기, 빼기, 나누기, 곱하기, 부정 및 단항 부정과 같은 표준 수학 연산을 지원합니다. 다른 정수 계열 형식과 마찬가지로 Int32 형식은 비트 `AND`, `OR`, `XOR`, 왼쪽 시프트 및 오른쪽 시프트 연산자도 지원합니다.

표준 숫자 연산자를 사용하여 두 Int32 값을 비교하거나 CompareTo 또는 Equals 메서드를 호출할 수 있습니다.

또한 Math 클래스의 멤버를 호출하여 숫자의 절대값 가져오기, 정수 나누기에서 몫 및 나머지 계산, 두 정수의 최대값 또는 최소값 결정, 숫자 기호 가져오기 및 숫자 반올림 등 다양한 숫자 연산을 수행할 수 있습니다.

Int32를 문자열로 표현

Int32 형식은 표준 및 사용자 지정 숫자 형식 문자열을 완전히 지원합니다. (자세한 내용은 서식 형식, 표준 숫자 형식 문자열 및 사용자 지정 숫자 형식 문자열을 참조하세요.)

값을 선행 0이 없는 정수 문자열로 서식 Int32 을 지정하려면 매개 변수가 없는 ToString() 메서드를 호출할 수 있습니다. "D" 형식 지정자를 사용하여 문자열 표현에 지정된 수의 선행 0을 포함할 수도 있습니다. "N" 형식 지정자를 사용하여 그룹 구분 기호를 포함하고 숫자의 문자열 표현에 표시할 소수 자릿수를 지정할 수 있습니다. "X" 형식 지정자를 사용하여 값을 16진수 문자열로 나타낼 Int32 수 있습니다. 다음 예제에서는 이러한 네 가지 방법으로 값 배열의 Int32 요소 형식을 지정합니다.

C#

```
int[] numbers = { -1403, 0, 169, 1483104 };
foreach (int number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-8}  -->   ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write("{0,11:D3}", number);
    // Display value with 1 decimal digit.
    Console.Write("{0,13:N1}", number);
    // Display value as hexadecimal.
    Console.Write("{0,12:X2}", number);
    // Display value with eight hexadecimal digits.
    Console.WriteLine("{0,14:X8}", number);
}
// The example displays the following output:
//    -1403     -->         -1403     -1,403.0    FFFFFA85      FFFFFA85
//    0         -->           000          0.0          00      00000000
//    169       -->           169        169.0          A9      000000A9
//    1483104   -->       1483104  1,483,104.0      16A160      0016A160
```

Int32 값을 이진, 8진수, 10진수 또는 16진수 문자열로 서식을 지정하기 위해 메서드를 호출하여 ToString(Int32, Int32) 메서드의 두 번째 매개 변수로 기본 값을 제공합니다. 다음 예제에서는 이 메서드를 호출하여 정수 값 배열의 이진, 8진수 및 16진수 표현을 표시합니다.

C#

```
int[] numbers = { -146, 11043, 2781913 };
Console.WriteLine("{0,8}   {1,32}   {2,11}   {3,10}",
                  "Value", "Binary", "Octal", "Hex");
foreach (int number in numbers)
{
    Console.WriteLine("{0,8}   {1,32}   {2,11}   {3,10}",
                      number, Convert.ToString(number, 2),
                      Convert.ToString(number, 8),
                      Convert.ToString(number, 16));
}
// The example displays the following output:
//       Value                                Binary         Octal           Hex
//        -146   11111111111111111111111101101110   37777777556     ffffff6e
//       11043                       10101100100011         25443         2b23
//     2781913             1010100111001011011001      12471331       2a72d9
```

10진수가 아닌 32비트 정수 값 사용

개별 정수를 10진수 값으로 사용하는 것 외에도 정수 값으로 비트 연산을 수행하거나 정수 값의 이진 또는 16진수 표현으로 작업할 수 있습니다. Int32 값은 31비트로 표시되고 30초 비트는 부호 비트로 사용됩니다. 양수 값은 부호 및 진도 표현을 사용하여 표현됩니다. 음수 값은 2의 보수로 표현됩니다. 이는 Int32 값에 대해 비트 연산을 수행하거나 개별 비트로 작업할 때 유의해야 합니다. 소수점이 아닌 두 값에 대해 숫자, 부울 또는 비교 작업을 수행하려면 두 값 모두 동일한 표현을 사용해야 합니다.

System.Int64 구조체

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Int64 는 음수 9,223,372,036,854,775,808(상수로 표시 Int64.MinValue 됨)부터 양수 9,223,372,036,854,775,807까지의 값으로 부호 있는 정수(상수로 Int64.MaxValue 표시됨)를 나타내는 변경할 수 없는 값 형식입니다. .NET에는 0에서 18,446,744,073,709,551,615까지의 값을 나타내는 부호 없는 64비트 정수 값 형식 UInt64도 포함됩니다.

Int64 값 인스턴스화

다음과 같은 Int64 여러 가지 방법으로 값을 인스턴스화할 수 있습니다.

- 변수를 Int64 선언하고 데이터 형식 범위 내에 있는 리터럴 정수 값을 할당할 Int64 수 있습니다. 다음 예제에서는 두 개의 Int64 변수를 선언하고 이러한 방식으로 값을 할당합니다.

  C#

  ```
  long number1 = -64301728;
  long number2 = 255486129307;
  ```

- 범위가 Int64 형식의 하위 집합인 정수형 값을 할당할 수 있습니다. C#의 캐스트 연산자 또는 Visual Basic의 변환 메서드가 필요하지 않은 확대 변환입니다. F#에서는 오직 Int32 형식만 자동으로 확장될 수 있습니다.

  C#

  ```
  sbyte value1 = 124;
  short value2 = 1618;
  int value3 = Int32.MaxValue;

  long number1 = value1;
  long number2 = value2;
  long number3 = value3;
  ```

- 범위가 Int64 형식의 범위를 초과하는 숫자 형식 값을 할당할 수 있습니다. 축소 변환이므로 C# 또는 F#의 캐스트 연산자와 Visual Basic의 변환 메서드(있는 경우 `Option Strict`)가 필요합니다. 숫자 값이 SingleDouble소수 구성 요소를 포함하는 값이거나 Decimal 소수 부분의 처리는 변환을 수행하는 컴파일러에 따라 달라집니다. 다음 예제에서는 축소 변환을 수행하여 여러 숫자 값을 변수에 Int64 할당합니다.

  C#

```
ulong ulNumber = 163245617943825;
try {
   long number1 = (long) ulNumber;
   Console.WriteLine(number1);
}
catch (OverflowException) {
   Console.WriteLine($"{ulNumber} is out of range of an Int64.");
}

double dbl2 = 35901.997;
try {
   long number2 = (long) dbl2;
   Console.WriteLine(number2);
}
catch (OverflowException) {
   Console.WriteLine($"{dbl2} is out of range of an Int64.");
}

BigInteger bigNumber = (BigInteger) 1.63201978555e30;
try {
   long number3 = (long) bigNumber;
   Console.WriteLine(number3);
}
catch (OverflowException) {
   Console.WriteLine($"{bigNumber} is out of range of an Int64.");
}
// The example displays the following output:
//    163245617943825
//    35902
//    1,632,019,785,549,999,969,612,091,883,520 is out of range of an Int64.
```

- 클래스의 Convert 메서드를 호출하여 지원되는 모든 형식을 값으로 Int64 변환할 수 있습니다. Int64 IConvertible 인터페이스를 지원하므로 가능합니다. 다음 예제에서는 Decimal 값 배열을 Int64 값으로 변환하는 방법을 보여 줍니다.

C#

```
decimal[] values= { Decimal.MinValue, -1034.23m, -12m, 0m, 147m,
                    199.55m, 9214.16m, Decimal.MaxValue };
long result;

foreach (decimal value in values)
{
   try {
      result = Convert.ToInt64(value);
      Console.WriteLine($"Converted the {value.GetType().Name} value
'{value}' to the {result.GetType().Name} value {result}.");
   }
   catch (OverflowException) {
      Console.WriteLine($"{value} is outside the range of the Int64 type.");
   }
}
```

```
// The example displays the following output:
//    -79228162514264337593543950335 is outside the range of the Int64 type.
//    Converted the Decimal value '-1034.23' to the Int64 value -1034.
//    Converted the Decimal value '-12' to the Int64 value -12.
//    Converted the Decimal value '0' to the Int64 value 0.
//    Converted the Decimal value '147' to the Int64 value 147.
//    Converted the Decimal value '199.55' to the Int64 value 200.
//    Converted the Decimal value '9214.16' to the Int64 value 9214.
//    79228162514264337593543950335 is outside the range of the Int64 type.
```

- Parse 또는 TryParse 메서드를 호출하여 Int64 값의 문자열 표현을 Int64으로 변환할 수 있습니다. 문자열은 10진수 또는 16진수를 포함할 수 있습니다. 다음 예제에서는 10진수 및 16진수 문자열을 모두 사용하여 구문 분석 작업을 보여 줍니다.

C#

```
string string1 = "244681903147";
try {
   long number1 = Int64.Parse(string1);
   Console.WriteLine(number1);
}
catch (OverflowException) {
   Console.WriteLine($"'{string1}' is out of range of a 64-bit integer.");
}
catch (FormatException) {
   Console.WriteLine($"The format of '{string1}' is invalid.");
}

string string2 = "F9A3CFF0A";
try {
   long number2 = Int64.Parse(string2,
                              System.Globalization.NumberStyles.HexNumber);
   Console.WriteLine(number2);
}
catch (OverflowException) {
   Console.WriteLine($"'{string2}' is out of range of a 64-bit integer.");
}
catch (FormatException) {
   Console.WriteLine($"The format of '{string2}' is invalid.");
}
// The example displays the following output:
//    244681903147
//    67012198154
```

Int64 값에 대한 작업 수행

이 형식은 Int64 더하기, 빼기, 나누기, 곱하기, 부정 및 단항 부정과 같은 표준 수학 연산을 지원합니다. 다른 정수 계열 형식과 마찬가지로 Int64 형식은 비트 `AND`, `OR`, `XOR`, 왼쪽 시프트 및 오른쪽 시프트 연산자도 지원합니다.

표준 숫자 연산자를 사용하여 두 Int64 값을 비교하거나 CompareTo 또는 Equals 메서드를 호출할 수 있습니다.

또한 클래스의 Math 멤버를 호출하여 숫자의 절대값 가져오기, 정수 나누기에서 몫 및 나머지 계산, 두 개의 긴 정수의 최대값 또는 최소값 결정, 숫자 기호 가져오기, 숫자 반올림 등 다양한 숫자 연산을 수행할 수 있습니다.

Int64를 문자열로 표현

Int64 형식은 표준 및 사용자 지정 숫자 형식 문자열을 완전히 지원합니다. (자세한 내용은 서식 형식, 표준 숫자 형식 문자열 및 사용자 지정 숫자 형식 문자열을 참조하세요.)

값을 선행 0이 없는 정수 문자열로 서식 Int64 을 지정하려면 매개 변수가 없는 ToString() 메서드를 호출할 수 있습니다. "D" 형식 지정자를 사용하여 문자열 표현에 지정된 수의 선행 0을 포함할 수도 있습니다. "N" 형식 지정자를 사용하여 그룹 구분 기호를 포함하고 숫자의 문자열 표현에 표시할 소수 자릿수를 지정할 수 있습니다. "X" 형식 지정자를 사용하여 값을 16진수 문자열로 나타낼 Int64 수 있습니다. 다음 예제에서는 이러한 네 가지 방법으로 값 배열의 Int64 요소 형식을 지정합니다.

C#

```
long[] numbers = { -1403, 0, 169, 1483104 };
foreach (var number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-8}  -->   ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write("{0,8:D3}", number);
    // Display value with 1 decimal digit.
    Console.Write("{0,13:N1}", number);
    // Display value as hexadecimal.
    Console.Write("{0,18:X2}", number);
    // Display value with eight hexadecimal digits.
    Console.WriteLine("{0,18:X8}", number);
}
// The example displays the following output:
//    -1403     -->      -1403     -1,403.0  FFFFFFFFFFFFFA85  FFFFFFFFFFFFFA85
//    0         -->        000          0.0                00          00000000
//    169       -->        169        169.0                A9          000000A9
//    1483104   -->    1483104  1,483,104.0            16A160          0016A160
```

Int64 값을 이진, 8진수, 10진수 또는 16진수 문자열로 서식을 지정하기 위해 메서드를 호출하여 ToString(Int64, Int32) 메서드의 두 번째 매개 변수로 기본 값을 제공합니다. 다음 예제에서는 이 메서드를 호출하여 정수 값 배열의 이진, 8진수 및 16진수 표현을 표시합니다.

C#

```
long[] numbers = { -146, 11043, 2781913 };
foreach (var number in numbers)
{
    Console.WriteLine($"{number} (Base 10):");
    Console.WriteLine($"   Binary:  {Convert.ToString(number, 2)}");
    Console.WriteLine($"   Octal:   {Convert.ToString(number, 8)}");
    Console.WriteLine($"   Hex:     {Convert.ToString(number, 16)}
{Environment.NewLine}");
}
// The example displays the following output:
//    -146 (Base 10):
//       Binary:  1111111111111111111111111111111111111111111111111111111101101110
//       Octal:   1777777777777777777556
//       Hex:     ffffffffffffff6e
//
//    11043 (Base 10):
//       Binary:  10101100100011
//       Octal:   25443
//       Hex:     2b23
//
//    2781913 (Base 10):
//       Binary:  1010100111001011011001
//       Octal:   12471331
//       Hex:     2a72d9
```

10진수가 아닌 32비트 정수 값 사용

개별 long 정수를 10진수 값으로 사용하는 것 외에도, 긴 정수 값에 대해 비트 연산을 수행하거나 긴 정수 값을 이진수 또는 16진수로 변환하여 작업할 수 있습니다. Int64 값은 63비트로 표시되며, 64비트가 부호 비트로 사용됩니다. 양수 값은 부호 및 진도 표현을 사용하여 표현됩니다. 음수 값은 2의 보수로 표현됩니다. 이는 Int64 값에 대해 비트 연산을 수행하거나 개별 비트로 작업할 때 유의해야 합니다. 소수점이 아닌 두 값에 대해 숫자, 부울 또는 비교 작업을 수행하려면 두 값 모두 동일한 표현을 사용해야 합니다.

System.Numerics.BigInteger 구조체

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

형식은 BigInteger 이론적으로 값에 상한이나 하한이 없는 임의로 큰 정수를 나타내는 변경할 수 없는 형식입니다. 형식 BigInteger의 멤버는 다른 정수 계열 형식(Byte, Int16, Int32, Int64, SByte, UInt16, UInt32, 및 UInt64 형식)의 멤버와 밀접하게 유사합니다. 이 형식은 .NET의 다른 정수 계열 형식과 다르며, 범위는 해당 `MinValue` 형식과 `MaxValue` 속성으로 표시됩니다.

> **참고**
>
> BigInteger 형식은 변경할 수 없고(**변경 가능성** 참조) 상한이나 하한 OutOfMemoryException 이 없으므로 값이 너무 커지는 BigInteger 모든 작업에 대해 throw될 수 있습니다.

BigInteger 개체 인스턴스화

다음과 같은 여러 가지 방법으로 개체를 BigInteger 인스턴스화할 수 있습니다.

- 키워드를 `new` 사용하고 모든 정수 또는 부동 소수점 값을 생성자에 매개 변수 BigInteger 로 제공할 수 있습니다. 부동 소수점 값은 BigInteger에 할당되기 전에 잘립니다. 다음 예에서는 `new` 키워드를 사용하여 BigInteger 값을 인스턴스화하는 방법을 보여 줍니다.

  C#

  ```
  BigInteger bigIntFromDouble = new BigInteger(179032.6541);
  Console.WriteLine(bigIntFromDouble);
  BigInteger bigIntFromInt64 = new BigInteger(934157136952);
  Console.WriteLine(bigIntFromInt64);
  // The example displays the following output:
  //   179032
  //   934157136952
  ```

- 변수를 BigInteger 선언하고 해당 값이 정수 형식인 경우 숫자 형식과 마찬가지로 값을 할당할 수 있습니다. 다음 예제에서는 Int64을(를) 할당하여 BigInteger 값을 만듭니다.

  C#

  ```
  long longValue = 6315489358112;
  BigInteger assignedFromLong = longValue;
  Console.WriteLine(assignedFromLong);
  ```

```
// The example displays the following output:
//   6315489358112
```

- 값을 캐스팅하거나 먼저 변환하는 경우 개체에 BigInteger 소수점 또는 부동 소수점 값을 할당할 수 있습니다. 다음 예제에서는 C#에서 명시적으로 캐스팅하거나 Visual Basic에서 변환하여 Double와 Decimal 값을 BigInteger로 만듭니다.

C#

```
BigInteger assignedFromDouble = (BigInteger) 179032.6541;
Console.WriteLine(assignedFromDouble);
BigInteger assignedFromDecimal = (BigInteger) 64312.65m;
Console.WriteLine(assignedFromDecimal);
// The example displays the following output:
//   179032
//   64312
```

이러한 메서드를 사용하면 값이 기존 숫자 형식 중 하나의 범위에만 있는 개체를 인스턴스화 BigInteger 할 수 있습니다. 다음 세 가지 방법 중 하나로 값이 기존 숫자 형식의 범위를 초과할 수 있는 개체를 인스턴스화 BigInteger 할 수 있습니다.

- 키워드를 `new` 사용하고 모든 크기의 바이트 배열을 BigInteger.BigInteger 생성자에 제공할 수 있습니다. 다음은 그 예입니다.

C#

```
byte[] byteArray = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
BigInteger newBigInt = new BigInteger(byteArray);
Console.WriteLine($"The value of newBigInt is {newBigInt} (or
0x{newBigInt:x}).");
// The example displays the following output:
//   The value of newBigInt is 4759477275222530853130 (or
0x102030405060708090a).
```

- Parse 메서드 또는 TryParse 메서드를 호출하여 숫자의 문자열 표현을 BigInteger로 변환할 수 있습니다. 다음은 그 예입니다.

C#

```
string positiveString = "91389681247993671255432112000000";
string negativeString = "-90315837410896312071002088037140000";
BigInteger posBigInt = 0;
BigInteger negBigInt = 0;

try {
   posBigInt = BigInteger.Parse(positiveString);
   Console.WriteLine(posBigInt);
}
```

```
catch (FormatException)
{
   Console.WriteLine($"Unable to convert the string '{positiveString}' to a
BigInteger value.");
}

if (BigInteger.TryParse(negativeString, out negBigInt))
  Console.WriteLine(negBigInt);
else
   Console.WriteLine($"Unable to convert the string '{negativeString}' to a
BigInteger value.");

// The example displays the following output:
//   9.13896812479936712554321l2E+31
//   -9.031583741089631207100208803 7E+34
```

- 숫자 식에 대해 일부 작업을 수행하고 계산된 BigInteger 결과를 반환하는 BigInteger 메서드를 (`Shared` Visual Basic에서) 호출 `static`할 수 있습니다. 다음 예제에서는 UInt64.MaxValue을 세제곱하여 그 결과를 BigInteger에 할당합니다.

```
C#

BigInteger number = BigInteger.Pow(UInt64.MaxValue, 3);
Console.WriteLine(number);
// The example displays the following output:
//    627710173538668076281494232244485102576757185438985853337 5
```

BigInteger의 초기화되지 않은 값은 Zero입니다.

BigInteger 값에 대한 작업 수행

다른 정수 계열 형식을 BigInteger 사용하는 것처럼 인스턴스를 사용할 수 있습니다. BigInteger 표준 숫자 연산자를 오버로드하여 더하기, 빼기, 나누기, 곱하기 및 단항 부정과 같은 기본 수학 연산을 수행할 수 있습니다. 표준 숫자 연산자를 사용하여 두 BigInteger 값을 서로 비교할 수도 있습니다. 다른 정수 유형과 마찬가지로 BigInteger는 비트 연산자`And` 및 왼쪽 이동`Or` 및 오른쪽 이동`XOr` 연산자를 지원합니다. 사용자 지정 연산자를 지원하지 않는 언어의 BigInteger 경우 이 구조는 수학 연산을 수행하기 위한 동일한 메서드도 제공합니다. 여기에는 Add, Divide, Multiply, Negate, Subtract 및 다른 여러 가지 항목이 포함됩니다.

구조체의 많은 멤버는 BigInteger 다른 정수 계열 형식의 멤버에 직접 해당합니다. 또한 BigInteger 다음과 같은 멤버를 추가합니다.

- 값 Sign는 BigInteger 값의 부호를 나타내는 값을 반환합니다.
- Abs는 BigInteger 값의 절대값을 반환합니다.

- DivRem는 나눗셈의 몫과 나머지를 모두 반환합니다.
- GreatestCommonDivisor- 두 BigInteger 값의 가장 큰 공통 수수를 반환합니다.

이러한 추가 멤버의 대부분은 기본 숫자 형식을 사용할 수 있는 기능을 제공하는 클래스의 Math 멤버에 해당합니다.

가변성

다음 예제에서는 BigInteger 객체를 생성하고 그 값에 1을 더합니다.

C#

```
BigInteger number = BigInteger.Multiply(Int64.MaxValue, 3);
number++;
Console.WriteLine(number);
```

이 예제에서는 기존 개체의 값을 수정하는 것처럼 보이지만 그렇지 않습니다. BigInteger 개체는 변경할 수 없습니다. 즉, 내부적으로 공용 언어 런타임은 실제로 새 BigInteger 개체를 만들고 이전 값보다 큰 값을 할당합니다. 그런 다음 이 새 개체가 호출자에게 반환됩니다.

> ⓘ **참고**
>
> .NET의 다른 숫자 형식도 변경할 수 없습니다. 그러나 형식에 **BigInteger** 상한 또는 하한이 없으므로 해당 값이 매우 커지고 성능에 측정 가능한 영향을 줄 수 있습니다.

이 프로세스는 호출자에게 투명하게 보이지만, 성능 페널티가 발생합니다. 경우에 따라 특히 매우 큰 BigInteger 값에 대한 루프에서 반복 작업이 수행되는 경우 성능 저하가 클 수 있습니다. 예를 들어 다음 예제에서는 작업이 최대 백만 번 반복적으로 수행되고 BigInteger 작업이 성공할 때마다 값이 하나씩 증가합니다.

C#

```
BigInteger number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (!SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    number++;
}
```

이러한 경우 변수에 대한 모든 중간 할당을 수행하여 성능을 향상시킬 수 있습니다 Int32 . 그런 다음 루프가 종료될 때 변수의 BigInteger 최종 값을 개체에 할당할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
BigInteger number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
int actualRepetitions = 0;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (!SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    actualRepetitions++;
}
number += actualRepetitions;
```

바이트 배열 및 16진수 문자열

값을 바이트 배열로 변환 BigInteger 하거나 바이트 배열을 값으로 변환하는 BigInteger 경우 바이트 순서를 고려해야 합니다. 이 BigInteger 구조체는 바이트 배열에 있는 개별 바이트가 little-endian 순서로 나열되어야 합니다(즉, 값의 하위 바이트가 상위 바이트보다 앞서야 합니다). BigInteger 값을 왕복하려면 ToByteArray 메서드를 호출하고 결과 바이트 배열을 BigInteger(Byte[]) 생성자에 전달할 수 있습니다. 다음 예제를 참고하세요.

C#

```
BigInteger number = BigInteger.Pow(Int64.MaxValue, 2);
Console.WriteLine(number);

// Write the BigInteger value to a byte array.
byte[] bytes = number.ToByteArray();

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0:X2} ", byteValue);
Console.WriteLine();

// Restore the BigInteger value from a Byte array.
BigInteger newNumber = new BigInteger(bytes);
Console.WriteLine(newNumber);
// The example displays the following output:
//    8.5070591730234615847396907784E+37
//    0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0x3F
```

```
//
//    8.5070591730234615847396907784E+37
```

다른 정수 계열 형식의 값을 나타내는 바이트 배열에서 값을 인스턴스화 BigInteger 하려면 정수 값을 메서드에 BitConverter.GetBytes 전달한 다음 결과 바이트 배열을 BigInteger(Byte[]) 생성자에 전달할 수 있습니다. 다음 예제에서는 Int16 값을 나타내는 바이트 배열에서 BigInteger 값을 인스턴스화합니다.

C#

```
short originalValue = 30000;
Console.WriteLine(originalValue);

// Convert the Int16 value to a byte array.
byte[] bytes = BitConverter.GetBytes(originalValue);

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0} ", byteValue.ToString("X2"));
Console.WriteLine();

// Pass byte array to the BigInteger constructor.
BigInteger number = new BigInteger(bytes);
Console.WriteLine(number);
// The example displays the following output:
//       30000
//       0x30 0x75
//       30000
```

구조는 BigInteger 음수 값을 2의 보수 표현으로 저장한다고 가정합니다. 구조체는 BigInteger 고정 길이 BigInteger(Byte[]) 가 없는 숫자 값을 나타내므로 생성자는 항상 배열의 마지막 바이트 중 가장 중요한 비트를 부호 비트로 해석합니다. 생성자가 음수 값의 두 보완 표현을 양수 값의 부호 및 크기 표현과 혼동하지 않도록 BigInteger(Byte[]) 하려면 바이트 배열에서 마지막 바이트의 가장 중요한 비트가 일반적으로 설정되는 양수 값에는 값이 0인 추가 바이트가 포함되어야 합니다. 예를 들어, 0xC0 0xBD 0xF0 0xFF는 -1,000,000 또는 4,293,967,296의 little-endian 16진수 표현입니다. 이 배열의 마지막 바이트의 최상위 비트가 켜져 있어, 생성자는 바이트 배열의 값을 -1,000,000으로 해석합니다. BigInteger(Byte[]) 값이 양수인 값을 인스턴스화 BigInteger 하려면 요소가 0xC0 0xBD 0xF0 0xFF 0x00 바이트 배열을 생성자에 전달해야 합니다. 다음 예제에서는 이를 보여 줍니다.

C#

```
int negativeNumber = -1000000;
uint positiveNumber = 4293967296;

byte[] negativeBytes = BitConverter.GetBytes(negativeNumber);
BigInteger negativeBigInt = new BigInteger(negativeBytes);
Console.WriteLine(negativeBigInt.ToString("N0"));
```

```
byte[] tempPosBytes = BitConverter.GetBytes(positiveNumber);
byte[] positiveBytes = new byte[tempPosBytes.Length + 1];
Array.Copy(tempPosBytes, positiveBytes, tempPosBytes.Length);
BigInteger positiveBigInt = new BigInteger(positiveBytes);
Console.WriteLine(positiveBigInt.ToString("N0"));
// The example displays the following output:
//    -1,000,000
//    4,293,967,296
```

양수 값을 사용하여 ToByteArray 메서드가 생성한 바이트 배열에는 이 추가 0-값 바이트가 포함됩니다. 따라서 구조체는 BigInteger 다음 예제와 같이 바이트 배열에 할당한 다음 바이트 배열에서 복원하여 성공적으로 왕복 값을 반환할 수 있습니다.

C#

```
BigInteger positiveValue = 15777216;
BigInteger negativeValue = -1000000;

Console.WriteLine("Positive value: " + positiveValue.ToString("N0"));
byte[] bytes = positiveValue.ToByteArray();

foreach (byte byteValue in bytes)
    Console.Write("{0:X2} ", byteValue);
Console.WriteLine();
positiveValue = new BigInteger(bytes);
Console.WriteLine("Restored positive value: " + positiveValue.ToString("N0"));

Console.WriteLine();

Console.WriteLine("Negative value: " + negativeValue.ToString("N0"));
bytes = negativeValue.ToByteArray();
foreach (byte byteValue in bytes)
    Console.Write("{0:X2} ", byteValue);
Console.WriteLine();
negativeValue = new BigInteger(bytes);
Console.WriteLine("Restored negative value: " + negativeValue.ToString("N0"));
// The example displays the following output:
//       Positive value: 15,777,216
//       C0 BD F0 00
//       Restored positive value: 15,777,216
//
//       Negative value: -1,000,000
//       C0 BD F0
//       Restored negative value: -1,000,000
```

그러나 개발자가 동적으로 만들거나 부호 없는 정수를 바이트 배열(예 BitConverter.GetBytes(UInt16)BitConverter.GetBytes(UInt32): , 및BitConverter.GetBytes(UInt64))으로 변환하는 메서드에 의해 반환되는 바이트 배열에 이 0 값 바이트를 추가해야 할 수 있습니다.

16진수 문자열 BigInteger.Parse(String, NumberStyles) 을 구문 분석할 때 및 BigInteger.Parse(String, NumberStyles, IFormatProvider) 메서드는 문자열에서 첫 번째 바이트의 가장 중요한 비트가 설정되거나 문자열의 첫 번째 16진수 숫자가 바이트 값의 하위 4비트를 나타내는 경우 값이 2의 보수 표현을 사용하여 표현된다고 가정합니다. 예를 들어 "FF01" 및 "F01"은 모두 10진수 값 -255를 나타냅니다. 음수 값과 긍정을 구분하려면 양수 값에 선행 0이 포함되어야 합니다. 메서드의 ToString 관련 오버로드는 "X" 형식 문자열을 전달할 때 양수 값에 대해 반환된 16진수 문자열에 선행 0을 추가합니다. 이렇게 하면 다음 예제에서 ToString 및 Parse 메서드를 사용하여 BigInteger 값을 왕복 처리할 수 있습니다.

C#

```
BigInteger negativeNumber = -1000000;
BigInteger positiveNumber = 15777216;

string negativeHex = negativeNumber.ToString("X");
string positiveHex = positiveNumber.ToString("X");

BigInteger negativeNumber2, positiveNumber2;
negativeNumber2 = BigInteger.Parse(negativeHex,
                                   NumberStyles.HexNumber);
positiveNumber2 = BigInteger.Parse(positiveHex,
                                   NumberStyles.HexNumber);

Console.WriteLine($"Converted {negativeNumber:N0} to {negativeHex} back to
{negativeNumber2:N0}.");
Console.WriteLine($"Converted {positiveNumber:N0} to {positiveHex} back to
{positiveNumber2:N0}.");
// The example displays the following output:
//       Converted -1,000,000 to F0BDC0 back to -1,000,000.
//       Converted 15,777,216 to 0F0BDC0 back to 15,777,216.
```

그러나 다른 정수 계열 형식의 메서드 또는 매개 변수를 포함하는 `toBase` 메서드의 ToString 오버로드를 호출 `ToString` 하여 만든 16진수 문자열은 16진수 문자열이 파생된 값 또는 원본 데이터 형식의 부호를 나타내지 않습니다. 이러한 문자열에서 값을 성공적으로 인스턴스화 BigInteger 하려면 몇 가지 추가 논리가 필요합니다. 다음 예제에서는 하나의 가능한 구현을 제공합니다.

C#

```
using System;
using System.Globalization;
using System.Numerics;

public struct HexValue
{
    public int Sign;
    public string Value;
}
```

```
public class ByteHexExample2
{
    public static void Main()
    {
        uint positiveNumber = 4039543321;
        int negativeNumber = -255423975;

        // Convert the numbers to hex strings.
        HexValue hexValue1, hexValue2;
        hexValue1.Value = positiveNumber.ToString("X");
        hexValue1.Sign = Math.Sign(positiveNumber);

        hexValue2.Value = Convert.ToString(negativeNumber, 16);
        hexValue2.Sign = Math.Sign(negativeNumber);

        // Round-trip the hexadecimal values to BigInteger values.
        string hexString;
        BigInteger positiveBigInt, negativeBigInt;

        hexString = (hexValue1.Sign == 1 ? "0" : "") + hexValue1.Value;
        positiveBigInt = BigInteger.Parse(hexString, NumberStyles.HexNumber);
        Console.WriteLine($"Converted {positiveNumber} to {hexValue1.Value} and
back to {positiveBigInt}.");

        hexString = (hexValue2.Sign == 1 ? "0" : "") + hexValue2.Value;
        negativeBigInt = BigInteger.Parse(hexString, NumberStyles.HexNumber);
        Console.WriteLine($"Converted {negativeNumber} to {hexValue2.Value} and
back to {negativeBigInt}.");
    }
}
// The example displays the following output:
//       Converted 4039543321 to F0C68A19 and back to 4039543321.
//       Converted -255423975 to f0c68a19 and back to -255423975.
```

System.Numerics.Complex 구조체

2025. 05. 15.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

복소수는 실수 부분과 허수 부분으로 구성된 숫자입니다. 복소수 z는 일반적으로 형식 `z = x + yi` 으로 작성됩니다. 여기서 *x*와 *y*는 실수이고, *i*는 i 2 = -1 속성을 가진 허수 단위입니다. 복소수의 실제 부분은 x로 표시되고, 복소수의 허수 부분은 y로 표시됩니다.

이 형식은 Complex 복소수를 인스턴스화하고 조작할 때 카티전 좌표계(실제, 가상)를 사용합니다. 복소수는 복소수 평면이라고 하는 2차원 좌표계의 한 지점으로 나타낼 수 있습니다. 복소수의 실제 부분은 x축(가로 축)에 배치되고 가상 부분은 y축(세로 축)에 배치됩니다.

복소수 평면의 모든 지점은 극좌표계를 사용하여 절대값에 따라 표현할 수도 있습니다. 극좌표에서 점의 특징은 두 가지입니다.

- 점의 크기는 원점(즉, 0,0 또는 x축과 y축이 교차하는 점)에서의 거리입니다.
- 그것의 위상은 실제 축과 원점에서 점까지 그리는 선 사이의 각도입니다.

복소수 인스턴스화

다음 방법 중 하나로 복소수에 값을 할당할 수 있습니다.

- 생성자에 두 Double 값을 전달합니다. 첫 번째 값은 복소수의 실제 부분을 나타내고 두 번째 값은 허수 부분을 나타냅니다. 이러한 값은 2차원 카티전 좌표계의 복소수 위치를 나타냅니다.

- 정적 `Shared` 메서드를 호출하여 극좌표로부터 복소수를 만듭니다 (Complex.FromPolarCoordinates는 Visual Basic에서 사용).

- 개체에 Byte, , SByte, Int16, UInt16Int32, UInt32, Int64, UInt64Single또는 Double 값을 할당합니다Complex. 값은 복소수의 실제 부분이 되고 허수 부분은 0과 같습니다.

- C#에서는 Decimal 또는 BigInteger 값을 Complex 개체로 캐스팅하거나, Visual Basic에서는 변환합니다. 값은 복소수의 실제 부분이 되고 허수 부분은 0과 같습니다.

- 메서드 또는 연산자가 반환하는 복소수를 개체에 할당합니다 Complex . 예를 들어 두 Complex.Add 복소수의 합계인 복소수를 반환하고 Complex.Addition 연산자는 두 개의 복소수를 추가하고 결과를 반환하는 정적 메서드입니다.

다음 예제에서는 복소수에 값을 할당하는 이러한 다섯 가지 방법을 각각 보여 줍니다.

```
C#
```

```
using System;
using System.Numerics;

public class CreateEx
{
    public static void Run()
    {
        // Create a complex number by calling its class constructor.
        Complex c1 = new Complex(12, 6);
        Console.WriteLine(c1);

        // Assign a Double to a complex number.
        Complex c2 = 3.14;
        Console.WriteLine(c2);

        // Cast a Decimal to a complex number.
        Complex c3 = (Complex)12.3m;
        Console.WriteLine(c3);

        // Assign the return value of a method to a Complex variable.
        Complex c4 = Complex.Pow(Complex.One, -1);
        Console.WriteLine(c4);

        // Assign the value returned by an operator to a Complex variable.
        Complex c5 = Complex.One + Complex.One;
        Console.WriteLine(c5);

        // Instantiate a complex number from its polar coordinates.
        Complex c6 = Complex.FromPolarCoordinates(10, .524);
        Console.WriteLine(c6);
    }
}
// The example displays the following output:
//       (12, 6)
//       (3.14, 0)
//       (12.3, 0)
//       (1, 0)
//       (2, 0)
//       (8.65824721882145, 5.00347430269914)
```

복소수를 사용하는 작업

.NET의 Complex 구조에는 다음 기능을 제공하는 멤버가 포함됩니다.

- 두 복소수를 비교하여 같은지 여부를 확인하는 메서드입니다.
- 복소수에 대한 산술 연산을 수행하는 연산자입니다. Complex 연산자를 사용하면 복소수를 사용하여 더하기, 빼기, 곱하기, 나누기 및 단항 부정을 수행할 수 있습니다.
- 복소수에 대해 다른 숫자 연산을 수행하는 메서드입니다. 네 가지 기본 산술 연산 외에도 복소수를 지정된 전력으로 올리고, 복소수의 제곱근을 찾고, 복소수의 절대값을 가져올 수

있습니다.
- 복소수에 대해 삼각 연산을 수행하는 메서드입니다. 예를 들어 복소수로 표시되는 각도의 탄젠트를 계산할 수 있습니다.

Real 및 Imaginary 속성이 읽기 전용이므로, 기존 Complex 개체의 값을 수정할 수 없습니다. 반환 값이 형식Complex인 경우 숫자에 대한 Complex 작업을 수행하는 모든 메서드는 새 Complex 숫자를 반환합니다.

정밀도 및 복소수

복소수의 실제 부분과 가상 부분은 두 개의 배정밀도 부동 소수점 값으로 표시됩니다. 즉 Complex , 배정밀도 부동 소수점 값과 같은 값은 숫자 연산의 결과로 정밀도를 잃을 수 있습니다. 즉, 두 값 간의 차이가 정밀도 손실로 인해 발생하더라도 두 Complex 값의 같음에 대한 엄격한 비교가 실패할 수 있습니다. 자세한 내용은 Double를 참조하세요.

예를 들어, 숫자의 로그에 대해 거듭제곱 연산을 수행하면 원래 숫자가 반환됩니다. 그러나 일부 경우에는 부동 소수점 값의 정확성 손실로 인해 두 값 사이에 약간의 차이가 발생할 수 있습니다. 이는 다음 예제가 보여줍니다.

C#

```
Complex value = new Complex(Double.MinValue / 2, Double.MinValue / 2);
Complex value2 = Complex.Exp(Complex.Log(value));
Console.WriteLine($"{value} \n{value2} \nEqual: {value == value2}");
// The example displays the following output:
//    (-8.98846567431158E+307, -8.98846567431158E+307)
//    (-8.98846567431161E+307, -8.98846567431161E+307)
//    Equal: False
```

마찬가지로 숫자의 제곱근을 계산하는 다음 예제는 .NET의 Complex 32비트 및 IA64 버전에서 약간 다른 결과를 생성합니다.

C#

```
Complex minusOne = new Complex(-1, 0);
Console.WriteLine(Complex.Sqrt(minusOne));
// The example displays the following output:
//    (6.12303176911189E-17, 1) on 32-bit systems.
//    (6.12323399573677E-17,1) on IA64 systems.
```

무한대 및 NaN

복소수의 실제 부분과 허수 부분은 값으로 Double 표시됩니다. 복소수의 실제 또는 허수 부분은 Double.MinValue에서 Double.MaxValue까지뿐만 아니라, 그 값은 Double.PositiveInfinity,

Double.NegativeInfinity, 또는 Double.NaN가 될 수 있습니다. Double.PositiveInfinity, Double.NegativeInfinity및 Double.NaN 모든 산술 또는 삼각 연산에서 전파됩니다.

다음 예제에서 Zero로 나누면 실수 부분과 허수 부분이 둘 다 Double.NaN인 복소수가 생성됩니다. 따라서 이 값을 곱하면 실제 부분과 허수 부분이 있는 복소수도 생성됩니다 Double.NaN. 마찬가지로 Double 형식의 범위를 초과하는 곱셈을 수행하면 실수 부분이 Double.NaN이고 허수 부분이 Double.PositiveInfinity인 복소수가 생성됩니다. 이후에 이 복소수로 나누기를 수행하면 실제 부분과 허수 부분이 Double.NaNDouble.PositiveInfinity있는 복소수를 반환합니다.

C#

```
using System;
using System.Numerics;

public class NaNEx
{
    public static void Run()
    {
        Complex c1 = new Complex(Double.MaxValue / 2, Double.MaxValue / 2);

        Complex c2 = c1 / Complex.Zero;
        Console.WriteLine(c2.ToString());
        c2 = c2 * new Complex(1.5, 1.5);
        Console.WriteLine(c2.ToString());
        Console.WriteLine();

        Complex c3 = c1 * new Complex(2.5, 3.5);
        Console.WriteLine(c3.ToString());
        c3 = c3 + new Complex(Double.MinValue / 2, Double.MaxValue / 2);
        Console.WriteLine(c3);
    }
}
// The example displays the following output:
//       (NaN, NaN)
//       (NaN, NaN)
//       (NaN, Infinity)
//       (NaN, Infinity)
```

유효하지 않은 경우거나 데이터 유형의 범위를 초과하는 복소수의 Double 수학 연산은 예외를 발생시키지 않습니다. 대신에 Double.PositiveInfinity, Double.NegativeInfinity, 또는 Double.NaN를 다음 조건에서 반환합니다.

- 양수를 0으로 나누면 Double.PositiveInfinity를 반환합니다.
- 데이터 형식의 Double 상한을 오버플로하는 모든 작업은 반환됩니다 Double.PositiveInfinity.
- 음수를 0으로 나누면 반환됩니다 Double.NegativeInfinity.
- 데이터 형식의 하한을 오버플로하는 Double 모든 작업은 반환됩니다 Double.NegativeInfinity.

- 0을 0으로 나누면 반환됩니다 Double.NaN.
- 피연산자의 값이 Double.PositiveInfinity, Double.NegativeInfinity, 또는 Double.NaN인 경우 수행되는 모든 작업은 특정 작업에 따라 Double.PositiveInfinity, Double.NegativeInfinity, 또는 Double.NaN를 반환합니다.

이는 메서드에서 수행하는 중간 계산에 적용됩니다. 예를 들어 곱하기 `new Complex(9e308, 9e308) and new Complex(2.5, 3.5)` 는 수식(ac - bd) + (ad + bc)i를 사용합니다. 곱하기에서 발생하는 실제 구성 요소의 계산은 9e308 *2.5 - 9e308* 3.5 식을 계산합니다. 이 식의 각 중간 곱셈은 Double.PositiveInfinity를 반환하며, Double.PositiveInfinity에서 Double.PositiveInfinity를 빼려는 시도는 Double.NaN를 반환합니다.

복소수 서식 지정

기본적으로 복소수의 문자열 표현은 `<`*실수*`;`*허수*`>` 형식을 취합니다. 여기서 *실수*와 *허수*는 복소수를 이루는 실수 및 허수 구성 요소 값의 문자열 표현입니다. 메서드의 ToString 일부 오버로드를 사용하면 이러한 Double 값의 문자열 표현을 사용자 지정하여 특정 문화권의 서식 규칙을 반영하거나 표준 또는 사용자 지정 숫자 형식 문자열로 정의된 특정 형식으로 표시할 수 있습니다. (자세한 내용은 를 참조하세요 .표준 숫자 형식 문자열 및사용자 지정 숫자 형식 문자열입니다.)

복소수의 문자열 표현을 표현하는 보다 일반적인 방법 중 하나는 복소수 `a` 의 실제 구성 요소이며 `b` 복소수의 허수 구성 요소인 형식`a + bi`을 사용합니다. 전기 공학에서 복소수는 가장 일반적으로 .로 `a + bj`표현됩니다. 이러한 두 양식 중 하나에서 복소수의 문자열 표현을 반환할 수 있습니다. 이렇게 하려면 ICustomFormatter 및 IFormatProvider 인터페이스를 구현하여 사용자 지정 형식 공급자를 정의한 다음, String.Format(IFormatProvider, String, Object[]) 메서드를 호출합니다.

다음 예제에서는 `ComplexFormatter` 클래스를 정의하여 복소수를 문자열로 표현하는 방법을 보여 줍니다. 이 문자열은 `a + bi` 또는 `a + bj` 형식으로 나타낼 수 있습니다.

C#

```
using System;
using System.Numerics;

public class ComplexFormatter : IFormatProvider, ICustomFormatter
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(ICustomFormatter))
            return this;
        else
            return null;
    }
```

```
    public string Format(string format, object arg,
                         IFormatProvider provider)
    {
        if (arg is Complex c1)
        {
            // Check if the format string has a precision specifier.
            int precision;
            string fmtString = string.Empty;
            if (format.Length > 1)
            {
                try
                {
                    precision = int.Parse(format.Substring(1));
                }
                catch (FormatException)
                {
                    precision = 0;
                }
                fmtString = "N" + precision.ToString();
            }
            if (format.Substring(0, 1).Equals("I",
StringComparison.OrdinalIgnoreCase))
                return c1.Real.ToString(fmtString) + " + " +
c1.Imaginary.ToString(fmtString) + "i";
            else if (format.Substring(0, 1).Equals("J",
StringComparison.OrdinalIgnoreCase))
                return c1.Real.ToString(fmtString) + " + " +
c1.Imaginary.ToString(fmtString) + "j";
            else
                return c1.ToString(format, provider);
        }
        else
        {
            if (arg is IFormattable formattable)
                return formattable.ToString(format, provider);
            else if (arg != null)
                return arg.ToString();
            else
                return string.Empty;
        }
    }
}
```

다음 예제에서는 이 사용자 지정 포맷터를 사용하여 복소수의 문자열 표현을 표시합니다.

C#

```
public class CustomFormatEx
{
    public static void Run()
    {
        Complex c1 = new(12.1, 15.4);
        Console.WriteLine($"Formatting with ToString:         {c1}");
```

```
        Console.WriteLine($"Formatting with ToString(format): {c1:N2}");
        Console.WriteLine($"Custom formatting with I0:\t" +
            $"  {string.Format(new ComplexFormatter(), "{0:I0}", c1)}");
        Console.WriteLine($"Custom formatting with J3:\t" +
            $"  {string.Format(new ComplexFormatter(), "{0:J3}", c1)}");
    }
}

// The example displays the following output:
//    Formatting with ToString():       <12.1; 15.4>
//    Formatting with ToString(format): <12.10; 15.40>
//    Custom formatting with I0:        12 + 15i
//    Custom formatting with J3:        12.100 + 15.400j
```

System.Single 구조체

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Single 값 형식은 음수 3.402823e38에서 양수 3.402823e38까지의 값과 양수 또는 음수 0, PositiveInfinity, NegativeInfinity및 숫자(NaN)가 아닌 단일 정밀도 32비트 숫자를 나타냅니다. 이는 매우 크거나(예: 행성이나 은하 사이의 거리) 또는 매우 작거나(예: 킬로그램 단위의 물질의 분자 질량) 종종 부정확한 값(예: 지구에서 다른 태양계까지의 거리)을 나타내기 위한 것입니다. Single 형식은 이진 부동 소수점 산술 연산에 대한 IEC 60559:1989(IEEE 754) 표준을 준수합니다.

System.Single 이 형식의 인스턴스를 비교하고, 인스턴스 값을 해당 문자열 표현으로 변환하고, 숫자의 문자열 표현을 이 형식의 인스턴스로 변환하는 메서드를 제공합니다. 서식 사양 코드가 값 형식의 문자열 표현을 제어하는 방법에 대한 자세한 내용은 형식, 표준 숫자 서식 문자열및 사용자 지정 숫자 서식 문자열 참조하세요.

부동 소수점 표현 및 정밀도

Single 데이터 형식은 다음 표와 같이 단정밀도 부동 소수점 값을 32비트 이진 형식으로 저장합니다.

테이블 확장

부분	비트
가수(Significand) 또는 맨티사(mantissa)	0-22
지수	23-30
기호(0 = 양수, 1 = 음수)	31

소수 자릿수가 일부 분수 값(예: 1/3 또는 Math.PI)을 정확하게 나타낼 수 없는 것처럼 이진 분수는 일부 소수 값을 나타낼 수 없습니다. 예를 들어 .2로 정확하게 10진수로 표현되는 2/10은 .0011111001001100 이진 분수로 표현되며 패턴 "1100"은 무한대로 반복됩니다. 이 경우 부동 소수점 값은 나타내는 숫자의 부정확한 표현을 제공합니다. 원래 부동 소수점 값에 대해 추가 수학 연산을 수행하면 정밀도가 부족한 경우가 많습니다. 예를 들어 .3을 10으로 곱하고 .3을 .3에서 .3으로 9번 추가하는 결과를 비교하면 곱하기보다 8개의 연산이 더 많이 포함되므로 정확도가 낮은 결과가 생성됩니다. 이 차이는 두 Single 값을 "R" 표준 숫자 형식 문자열을 사용하여 표시할 때만 명백합니다. 이 형식은 필요한 경우 Single 형식이 지원하는 9자리의 정밀도를 모두 표시합니다.

C#

```
using System;
```

```
public class Example12
{
    public static void Main()
    {
        Single value = .2f;
        Single result1 = value * 10f;
        Single result2 = 0f;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine($".2 * 10:           {result1:R}");
        Console.WriteLine($".2 Added 10 times: {result2:R}");
    }
}
// The example displays the following output:
//       .2 * 10:           2
//       .2 Added 10 times: 2.0000002
```

일부 숫자는 소수점 이진 값으로 정확하게 나타낼 수 없으므로 부동 소수점 숫자는 실제 숫자와 근사치일 수 있습니다.

모든 부동 소수점 숫자에는 제한된 수의 유효 자릿수가 있으며, 이는 부동 소수점 값이 실제 숫자와 얼마나 정확하게 일치하는지 결정합니다. Single 값의 전체 자릿수는 최대 7자리이지만 내부적으로는 최대 9자리 자릿수가 유지됩니다. 즉, 일부 부동 소수점 연산에는 부동 소수점 값을 변경할 정밀도가 부족할 수 있습니다. 다음 예제에서는 큰 단정밀도 부동 소수점 값을 정의한 다음 Single.Epsilon와 1조의 곱을 더합니다. 그러나 제품이 너무 작아서 원래 부동 소수점 값을 수정할 수 없습니다. 가장 낮은 유효 자릿수는 천 번째인 반면, 제품에서 가장 중요한 숫자는 10^{-30}.

C#

```
using System;

public class Example13
{
    public static void Main()
    {
        Single value = 123.456f;
        Single additional = Single.Epsilon * 1e15f;
        Console.WriteLine($"{value} + {additional} = {value + additional}");
    }
}

// The example displays the following output:
//    123.456 + 1.401298E-30 = 123.456
```

부동 소수점 숫자의 제한된 정밀도에는 다음과 같은 몇 가지 결과가 있습니다.

- 정밀도와 관계없이 동일하게 보이는 두 개의 부동 소수점 숫자도, 가장 덜 중요한 자릿수가 다르기 때문에 동일하게 비교되지 않을 수 있습니다. 다음 예제에서는 일련의 숫자가 함께

추가되고 해당 합계가 예상 합계와 비교됩니다. 메서드를 호출하면 `Equals` 값이 같지 않음을 나타냅니다.

C#

```
using System;

public class PrecisionList3Example
{
    public static void Main()
    {
        Single[] values = { 10.01f, 2.88f, 2.88f, 2.88f, 9.0f };
        Single result = 27.65f;
        Single total = 0f;
        foreach (var value in values)
            total += value;

        if (total.Equals(result))
            Console.WriteLine("The sum of the values equals the total.");
        else
            Console.WriteLine($"The sum of the values ({total}) does not equal
the total ({result}).");
    }
}

// The example displays the following output on .NET:
//      The sum of the values (27.650002) does not equal the total (27.65).
// The example displays the following output on .NET Framework:
//      The sum of the values (27.65) does not equal the total (27.65).
```

두 값은 더하기 작업 중에 정밀도가 손실되어 같지 않습니다. 이 경우 비교를 수행하기 전에 Math.Round(Double, Int32) 메서드를 호출하여 Single 값을 원하는 전체 자릿수로 반올림하여 문제를 해결할 수 있습니다.

- 부동 소수점 숫자를 사용하는 수학 또는 비교 연산은 이진 부동 소수점 숫자가 10진수와 같지 않을 수 있으므로 소수점 숫자를 사용하는 경우 동일한 결과를 생성하지 못할 수 있습니다. 이전 예제에서는 .3을 10으로 곱하고 .3을 .3에 9번 추가하는 결과를 표시하여 이를 설명했습니다.

  소수 자릿수 값이 있는 숫자 연산의 정확도가 중요한 경우 Decimal 형식 대신 Single 형식을 사용합니다. 정수 계열 값이 Int64 또는 UInt64 형식 범위를 벗어나는 숫자 연산의 정확도가 중요한 경우 BigInteger 형식을 사용합니다.

- 부동 소수점 숫자가 포함된 경우 값이 왕복 않을 수 있습니다. 연산이 원래의 부동 소수점 숫자를 다른 형태로 변환하고, 역연산이 그 변환된 형태를 다시 부동 소수점 숫자로 변환하며, 최종 부동 소수점 숫자가 원래의 부동 소수점 숫자와 같다면, 이러한 과정을 왕복 과정이라고 합니다. 변환 시 하나 이상의 유효 자릿수가 손실되거나 변경되어 왕복이 실패할 수 있습니다.

다음 예제에서는 세 Single 값이 문자열로 변환되고 파일에 저장됩니다. .NET Framework에서 이 예제를 실행하면 값이 동일한 것처럼 보이지만 복원된 값은 원래 값과 같지 않습니다. (이 문제는 이후 .NET에서 처리되었으며, 여기서 값은 올바르게 왕복합니다.)

C#

```
StreamWriter sw = new(@"./Singles.dat");
float[] values = { 3.2f / 1.11f, 1.0f / 3f, (float)Math.PI };
for (int ctr = 0; ctr < values.Length; ctr++)
{
    sw.Write(values[ctr].ToString());
    if (ctr != values.Length - 1)
        sw.Write("|");
}
sw.Close();

float[] restoredValues = new float[values.Length];
StreamReader sr = new(@"./Singles.dat");
string temp = sr.ReadToEnd();
string[] tempStrings = temp.Split('|');
for (int ctr = 0; ctr < tempStrings.Length; ctr++)
    restoredValues[ctr] = float.Parse(tempStrings[ctr]);

for (int ctr = 0; ctr < values.Length; ctr++)
    Console.WriteLine($"{values[ctr]} {(values[ctr].Equals(restoredValues[ctr])
? "=" : "<>")} {restoredValues[ctr]}");

// The example displays the following output on .NET Framework:
//       2.882883 <> 2.882883
//       0.3333333 <> 0.3333333
//       3.141593 <> 3.141593
```

.NET Framework를 대상으로 하는 경우 "G9" 표준 숫자 형식 문자열을 사용하여 Single 값의 전체 정밀도를 보존함으로써 값을 성공적으로 되돌릴 수 있습니다.

- Single 값은 Double 값보다 정밀도가 낮습니다. 변환 후 겉보기에 동등해 보이는 Single로 전환된 Double 값은 종종 정밀도의 차이로 인해 Double 값과 같지 않을 수 있습니다. 다음 예제에서는 동일한 나누기 작업의 결과가 Double 값과 Single 값에 할당됩니다. Single 값이 Double캐스팅된 후 두 값을 비교하면 같지 않음이 표시됩니다.

C#

```
using System;

public class Example9
{
    public static void Main()
    {
        Double value1 = 1 / 3.0;
        Single sValue2 = 1 / 3.0f;
        Double value2 = (Double)sValue2;
```

```
        Console.WriteLine($"{value1:R} = {value2:R}: {value1.Equals(value2)}");
    }
}
// The example displays the following output:
//       0.33333333333333331 = 0.3333333432674408: False
```

이 문제를 방지하려면 Double 데이터 형식 대신 Single 데이터 형식을 사용하거나 두 값의 전체 자릿수가 같도록 Round 메서드를 사용합니다.

동등성 검사

같게 간주하려면 두 Single 값이 동일한 값을 나타내야 합니다. 그러나 값 간의 정밀도 차이 또는 하나 또는 두 값 모두에 의한 정밀도 손실로 인해 동일할 것으로 예상되는 부동 소수점 값은 가장 낮은 유효 자릿수의 차이로 인해 같지 않은 것으로 판명되는 경우가 많습니다. 따라서 두 값이 같은지 여부를 확인하기 위해 Equals 메서드를 호출하거나 두 CompareTo 값 간의 관계를 확인하기 위해 Single 메서드를 호출하면 예기치 않은 결과가 발생하는 경우가 많습니다. 첫 번째 값의 전체 자릿수는 7자리이고 두 번째 값은 9이므로 다음 예제에서는 두 개의 분명히 같은 Single 값이 같지 않은 것으로 판명됩니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      float value1 = .3333333f;
      float value2 = 1.0f/3;
      Console.WriteLine($"{value1:R} = {value2:R}: {value1.Equals(value2)}");
   }
}
// The example displays the following output:
//       0.3333333 = 0.333333343: False
```

서로 다른 코드 경로를 따르고 여러 가지 방법으로 조작되는 계산 값은 종종 같지 않은 것으로 판명됩니다. 다음 예제에서는 한 Single 값이 제곱되고 제곱근이 계산되어 원래 값을 복원합니다. 두 번째 Single에 3.51을 곱하고 제곱한 후에, 결과의 제곱근을 3.51로 나누어 원래 값을 복원합니다. 두 값이 동일한 것처럼 보이지만 Equals(Single) 메서드를 호출하면 값이 같지 않음을 나타냅니다.

C#

```
float value1 = 10.201438f;
value1 = (float)Math.Sqrt((float)Math.Pow(value1, 2));
float value2 = (float)Math.Pow((float)value1 * 3.51f, 2);
value2 = ((float)Math.Sqrt(value2)) / 3.51f;
```

```
Console.WriteLine($"{value1} = {value2}: {value1.Equals(value2)}");

// The example displays the following output on .NET:
//       10.201438 = 10.201439: False
// The example displays the following output on .NET Framework:
//       10.20144 = 10.20144: False
```

정밀도 손실이 비교 결과에 영향을 줄 가능성이 있는 경우 Equals 또는 CompareTo 메서드를 호출하는 대신 다음 기술을 사용할 수 있습니다.

- Math.Round 메서드를 호출하여 두 값의 정밀도가 같은지 확인합니다. 다음 예제에서는 두 개의 소수 값이 동일하도록 이 방법을 사용하도록 이전 예제를 수정합니다.

  C#

  ```
  float value1 = .3333333f;
  float value2 = 1.0f / 3;
  int precision = 7;
  value1 = (float)Math.Round(value1, precision);
  value2 = (float)Math.Round(value2, precision);
  Console.WriteLine($"{value1:R} = {value2:R}: {value1.Equals(value2)}");

  // The example displays the following output:
  //        0.3333333 = 0.3333333: True
  ```

  정밀도 문제는 여전히 중간점 값의 반올림에 적용됩니다. 자세한 내용은 Math.Round(Double, Int32, MidpointRounding) 메서드를 참조하세요.

- 같음을 테스트하는 대신 대략적인 같음을 테스트합니다. 이 기술을 사용하려면 두 값이 다를 수 있지만 여전히 같을 수 있는 절대 크기를 정의하거나 더 작은 값이 더 큰 값과 다를 수 있는 상대 크기를 정의해야 합니다.

  > **⚠ Warning**
  >
  > Single.Epsilon 같음을 테스트할 때 두 Single 값 사이의 거리를 절대 측정값으로 사용하는 경우가 있습니다. 그러나 Single.Epsilon는 값이 0인 Single에 더하거나 뺄 수 있는 가능한 가장 작은 값을 측정합니다. 대부분의 양수 및 음수 Single 값의 경우 Single.Epsilon 값이 너무 작아 검색할 수 없습니다. 따라서 0인 값을 제외하고 같음 테스트에는 사용하지 않는 것이 좋습니다.

  다음 예제에서는 후자의 방법을 사용하여 두 값 간의 상대적 차이를 테스트하는 `IsApproximatelyEqual` 메서드를 정의합니다. 또한 `IsApproximatelyEqual` 메서드 및 Equals(Single) 메서드에 대한 호출의 결과와 대조됩니다.

  C#

```
public static void Main()
{
    float one1 = .1f * 10;
    float one2 = 0f;
    for (int ctr = 1; ctr <= 10; ctr++)
        one2 += .1f;

    Console.WriteLine($"{one1:R} = {one2:R}: {one1.Equals(one2)}");
    Console.WriteLine($"{one1:R} is approximately equal to {one2:R}: " +
        $"{IsApproximatelyEqual(one1, one2, .000001f)}");

    float negativeOne1 = -1 * one1;
    float negativeOne2 = -1 * one2;

    Console.WriteLine($"{negativeOne1:R} = {negativeOne2:R}:
{negativeOne1.Equals(negativeOne2)}");
    Console.WriteLine($"{negativeOne1:R} is approximately equal to
{negativeOne2:R}: " +
        $"{IsApproximatelyEqual(negativeOne1, negativeOne2, .000001f)}");
}

static bool IsApproximatelyEqual(float value1, float value2, float epsilon)
{
    // If they are equal anyway, just return True.
    if (value1.Equals(value2))
        return true;

    // Handle NaN, Infinity.
    if (Double.IsInfinity(value1) | Double.IsNaN(value1))
        return value1.Equals(value2);
    else if (Double.IsInfinity(value2) | Double.IsNaN(value2))
        return value1.Equals(value2);

    // Handle zero to avoid division by zero.
    double divisor = Math.Max(value1, value2);
    if (divisor.Equals(0))
        divisor = Math.Min(value1, value2);

    return Math.Abs((value1 - value2) / divisor) <= epsilon;
}

// The example displays the following output on .NET:
//       1 = 1.0000001: False
//       1 is approximately equal to 1.0000001: True
//       -1 = -1.0000001: False
//       -1 is approximately equal to -1.0000001: True
```

부동 소수점 값 및 예외

부동 소수점 값이 있는 작업은 정수 형식이 있는 작업과 달리 예외를 throw하지 않으며, 0으로 나누기 또는 오버플로와 같은 잘못된 작업의 경우 예외를 throw합니다. 대신 이러한 상황에서

부동 소수점 연산의 결과는 0, 양수 무한대, 음수 무한대 또는 숫자(NaN)가 아닙니다.

- 부동 소수점 연산의 결과가 대상 형식에 비해 너무 작으면 결과는 0입니다. 이 문제는 다음 예제와 같이 두 개의 매우 작은 부동 소수점 숫자를 곱할 때 발생할 수 있습니다.

  C#

  ```
  float value1 = 1.163287e-36f;
  float value2 = 9.164234e-25f;
  float result = value1 * value2;
  Console.WriteLine($"{value1} * {value2} = {result}");
  Console.WriteLine($"{result} = 0: {result.Equals(0.0f)}");

  // The example displays the following output:
  //       1.163287E-36 * 9.164234E-25 = 0
  //       0 = 0: True
  ```

- 부동 소수점 연산 결과의 크기가 대상 형식의 범위를 초과하는 경우, 결과의 부호에 따라 연산의 결과는 PositiveInfinity 또는 NegativeInfinity이 됩니다. 어떤 작업이 Single.MaxValue에서 오버플로하면 그 결과는 PositiveInfinity이고, Single.MinValue에서 오버플로하는 작업의 결과는 NegativeInfinity입니다. 이는 다음 예제에서 볼 수 있습니다.

  C#

  ```
  float value1 = 3.065e35f;
  float value2 = 6.9375e32f;
  float result = value1 * value2;
  Console.WriteLine($"PositiveInfinity: {Single.IsPositiveInfinity(result)}");
  Console.WriteLine($"NegativeInfinity: {Single.IsNegativeInfinity(result)}");
  Console.WriteLine();

  value1 = -value1;
  result = value1 * value2;
  Console.WriteLine($"PositiveInfinity: {Single.IsPositiveInfinity(result)}");
  Console.WriteLine($"NegativeInfinity: {Single.IsNegativeInfinity(result)}");

  // The example displays the following output:
  //       PositiveInfinity: True
  //       NegativeInfinity: False
  //
  //       PositiveInfinity: False
  //       NegativeInfinity: True
  ```

  PositiveInfinity 또한 양의 배수를 0으로 나누기에서 비롯된 결과이며, NegativeInfinity은 음의 배수를 0으로 나누기에서 비롯된 결과입니다.

- 부동 소수점 연산이 유효하지 않으면 그 연산의 결과는 NaN입니다. 예를 들어, 다음 연산의 결과는 NaN입니다.

- 배당금이 0인 0으로 나누기. 다른 경우에 0으로 나누면 PositiveInfinity 또는 NegativeInfinity이 발생할 수 있음을 유의하십시오.
- 잘못된 입력이 있는 모든 부동 소수점 연산. 예를 들어 음수 값의 제곱근을 찾으려고 시도하면 NaN반환됩니다.
- 값이 Single.NaN인수가 있는 모든 연산입니다.

타입 변환

Single 구조체는 명시적 또는 암시적 변환 연산자를 정의하지 않습니다. 대신 변환은 컴파일러에 의해 구현됩니다.

다음 표에서는 다른 기본 숫자 형식의 값을 값으로 변환할 수 있는 Single 방법을 나열합니다. 또한 변환이 확대 또는 축소되는지 여부와 결과 Single 정밀도가 원래 값보다 작을 수 있는지 여부를 나타냅니다.

테이블 확장

변환하기	확대/축소	가능한 정밀도 손실
Byte	확대	아니오
Decimal	확대 C#에는 캐스트 연산자가 필요합니다.	예. Decimal 10진수 29자리의 정밀도를 지원합니다. Single 9를 지원합니다.
Double	축소; 범위를 벗어난 값은 Double.NegativeInfinity 또는 Double.PositiveInfinity변환됩니다.	예. Double 17진수의 정밀도를 지원합니다. Single 9를 지원합니다.
Int16	확대	아니오
Int32	확대	예. Int32 10진수의 정밀도를 지원합니다. Single 9를 지원합니다.
Int64	확대	예. Int64 19진수의 정밀도를 지원합니다. Single 9를 지원합니다.
SByte	확대	아니오
UInt16	확대	아니오
UInt32	확대	예. UInt32 10진수의 정밀도를 지원합니다. Single 9를 지원합니다.
UInt64	확대	예. Int64 20진수의 정밀도를 지원합니다. Single 9를 지원합니다.

다음 예제에서는 다른 기본 숫자 형식의 최소값 또는 최대값을 Single 값으로 변환합니다.

C#

```
using System;

public class Example4
{
    public static void Main()
    {
        dynamic[] values = { Byte.MinValue, Byte.MaxValue, Decimal.MinValue,
                             Decimal.MaxValue, Double.MinValue, Double.MaxValue,
                             Int16.MinValue, Int16.MaxValue, Int32.MinValue,
                             Int32.MaxValue, Int64.MinValue, Int64.MaxValue,
                             SByte.MinValue, SByte.MaxValue, UInt16.MinValue,
                             UInt16.MaxValue, UInt32.MinValue, UInt32.MaxValue,
                             UInt64.MinValue, UInt64.MaxValue };
        float sngValue;
        foreach (var value in values)
        {
            if (value.GetType() == typeof(Decimal) ||
                value.GetType() == typeof(Double))
                sngValue = (float)value;
            else
                sngValue = value;
            Console.WriteLine($"{value} ({value.GetType().Name}) --> {sngValue:R}
({sngValue.GetType().Name})");
        }
    }
}
// The example displays the following output:
//       0 (Byte) --> 0 (Single)
//       255 (Byte) --> 255 (Single)
//       -79228162514264337593543950335 (Decimal) --> -7.92281625E+28 (Single)
//       79228162514264337593543950335 (Decimal) --> 7.92281625E+28 (Single)
//       -1.79769313486232E+308 (Double) --> -Infinity (Single)
//       1.79769313486232E+308 (Double) --> Infinity (Single)
//       -32768 (Int16) --> -32768 (Single)
//       32767 (Int16) --> 32767 (Single)
//       -2147483648 (Int32) --> -2.14748365E+09 (Single)
//       2147483647 (Int32) --> 2.14748365E+09 (Single)
//       -9223372036854775808 (Int64) --> -9.223372E+18 (Single)
//       9223372036854775807 (Int64) --> 9.223372E+18 (Single)
//       -128 (SByte) --> -128 (Single)
//       127 (SByte) --> 127 (Single)
//       0 (UInt16) --> 0 (Single)
//       65535 (UInt16) --> 65535 (Single)
//       0 (UInt32) --> 0 (Single)
//       4294967295 (UInt32) --> 4.2949673E+09 (Single)
//       0 (UInt64) --> 0 (Single)
//       18446744073709551615 (UInt64) --> 1.84467441E+19 (Single)
```

또한 Double 값 Double.NaN, Double.PositiveInfinity및 Double.NegativeInfinity은 각각 Single.NaN, Single.PositiveInfinity및 Single.NegativeInfinity으로 변환됩니다.

일부 숫자 형식의 값을 Single 값으로 변환하면 정밀도가 떨어질 수 있습니다. 예제에서 알 수 있듯이 Decimal, Double, Int32, Int64, UInt32및 UInt64 값을 Single 값으로 변환할 때 정밀도 손실이 발생할 수 있습니다.

Single 값을 Double 변환하는 것은 확대 변환입니다. 변환은 Double 형식에 Single 값에 대한 정확한 표현이 없는 경우 정밀도가 손실될 수 있습니다.

Single 값을 Double 이외의 기본 숫자 데이터 형식 값으로 변환하려면 축소 변환이며 캐스트 연산자(C#) 또는 변환 메서드(Visual Basic)가 필요합니다. 대상 형식의 `MinValue` 및 `MaxValue` 속성으로 정의된 대상 데이터 형식의 범위를 벗어난 값은 다음 표와 같이 동작합니다.

테이블 확장

대상 유형	결과
모든 정수 계열 형식	확인된 컨텍스트에서 변환이 발생하는 경우 OverflowException 예외입니다. 확인되지 않은 컨텍스트(C#의 기본값)에서 변환이 발생하면 변환 작업이 성공하지만 값이 오버플로됩니다.
Decimal	예외입니다 OverflowException .

또한 Single.NaN, Single.PositiveInfinity및 Single.NegativeInfinity는 확인된 컨텍스트에서 정수로 변환하기 위한 OverflowException을 던지지만, 이러한 값은 확인되지 않은 컨텍스트에서 정수로 변환될 때 오버플로가 발생합니다. Decimal변환을 할 때는 항상 OverflowException를 던집니다. Double변환의 경우 각각 Double.NaN, Double.PositiveInfinity및 Double.NegativeInfinity변환합니다.

숫자 변환 중 Single 값의 정밀도 손실이 발생할 수 있음을 유의하십시오. 예제의 출력과 같이 비계열 Single 값을 변환하는 경우 Single 값이 반올림되거나(Visual Basic에서와 같이) 잘리면(C# 및 F#에서와 같이) 소수 구성 요소가 손실됩니다. Decimal 값으로 변환하는 경우 Single 값에 대상 데이터 형식의 정확한 표현이 없을 수 있습니다.

다음 예제에서는 여러 Single 값을 다른 여러 숫자 형식으로 변환합니다. 변환은 Visual Basic(기본값), C#(확인된 키워드로 인해) 및 F#(`open Checked` 문으로 인해)의 확인된 컨텍스트에서 발생합니다. 예제의 출력은 확인된 선택되지 않은 컨텍스트 모두에서 변환 결과를 보여 줍니다. `/removeintchecks+` 컴파일러 스위치를 사용하여 컴파일하고, C#에서 `checked` 문을 주석으로 처리하고, F#에서 `open Checked` 문을 주석으로 처리하여 Visual Basic에서 확인되지 않은 컨텍스트에서 변환을 수행할 수 있습니다.

C#

```
float[] values = { Single.MinValue, -67890.1234f, -12345.6789f,
                   12345.6789f, 67890.1234f, Single.MaxValue,
                   Single.NaN, Single.PositiveInfinity,
```

```
                  Single.NegativeInfinity };
checked
{
    foreach (var value in values)
    {
        try
        {
            Int64 lValue = (long)value;
            Console.WriteLine($"{value} ({value.GetType().Name}) --> {lValue}
(0x{lValue:X16}) ({lValue.GetType().Name})");
        }
        catch (OverflowException)
        {
            Console.WriteLine($"Unable to convert {value} to Int64.");
        }
        try
        {
            UInt64 ulValue = (ulong)value;
            Console.WriteLine($"{value} ({value.GetType().Name}) --> {ulValue}
(0x{ulValue:X16}) ({ulValue.GetType().Name})");
        }
        catch (OverflowException)
        {
            Console.WriteLine($"Unable to convert {value} to UInt64.");
        }
        try
        {
            Decimal dValue = (decimal)value;
            Console.WriteLine($"{value} ({value.GetType().Name}) --> {dValue}
({dValue.GetType().Name})");
        }
        catch (OverflowException)
        {
            Console.WriteLine($"Unable to convert {value} to Decimal.");
        }

        Double dblValue = value;
        Console.WriteLine($"{value} ({value.GetType().Name}) --> {dblValue}
({dblValue.GetType().Name})");
        Console.WriteLine();
    }
}

// The example displays the following output for conversions performed
// in a checked context:
//       Unable to convert -3.402823E+38 to Int64.
//       Unable to convert -3.402823E+38 to UInt64.
//       Unable to convert -3.402823E+38 to Decimal.
//       -3.402823E+38 (Single) --> -3.40282346638529E+38 (Double)
//
//       -67890.13 (Single) --> -67890 (0xFFFFFFFFFFFEF6CE) (Int64)
//       Unable to convert -67890.13 to UInt64.
//       -67890.13 (Single) --> -67890.12 (Decimal)
//       -67890.13 (Single) --> -67890.125 (Double)
//
```

```
//       -12345.68 (Single) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//       Unable to convert -12345.68 to UInt64.
//       -12345.68 (Single) --> -12345.68 (Decimal)
//       -12345.68 (Single) --> -12345.6787109375 (Double)
//
//       12345.68 (Single) --> 12345 (0x0000000000003039) (Int64)
//       12345.68 (Single) --> 12345 (0x0000000000003039) (UInt64)
//       12345.68 (Single) --> 12345.68 (Decimal)
//       12345.68 (Single) --> 12345.6787109375 (Double)
//
//       67890.13 (Single) --> 67890 (0x0000000000010932) (Int64)
//       67890.13 (Single) --> 67890 (0x0000000000010932) (UInt64)
//       67890.13 (Single) --> 67890.12 (Decimal)
//       67890.13 (Single) --> 67890.125 (Double)
//
//       Unable to convert 3.402823E+38 to Int64.
//       Unable to convert 3.402823E+38 to UInt64.
//       Unable to convert 3.402823E+38 to Decimal.
//       3.402823E+38 (Single) --> 3.40282346638529E+38 (Double)
//
//       Unable to convert NaN to Int64.
//       Unable to convert NaN to UInt64.
//       Unable to convert NaN to Decimal.
//       NaN (Single) --> NaN (Double)
//
//       Unable to convert ∞ to Int64.
//       Unable to convert ∞ to UInt64.
//       Unable to convert ∞ to Decimal.
//       ∞ (Single) --> ∞ (Double)
//
//       Unable to convert -∞ to Int64.
//       Unable to convert -∞ to UInt64.
//       Unable to convert -∞ to Decimal.
//       -∞ (Single) --> -∞ (Double)
```

숫자 형식의 변환에 대한 자세한 내용은 .NET 형식 변환 및 형식 변환 테이블참조하세요.

부동 소수점 기능

Single 구조 및 관련 형식은 다음과 같은 작업 범주를 수행하는 메서드를 제공합니다.

- 값의 비교입니다. Equals 메서드를 호출하여 두 Single 값이 같은지 또는 CompareTo 메서드를 호출하여 두 값 간의 관계를 확인할 수 있습니다.

  Single 구조체는 전체 비교 연산자 집합도 지원합니다. 예를 들어 같음 또는 같지 않음을 테스트하거나 한 값이 다른 값보다 크거나 같은지 확인할 수 있습니다. 피연산자 중 하나가 Double경우 비교를 수행하기 전에 Single 값이 Double 변환됩니다. 피연산자 중 하나가 정수 형식인 경우 비교를 수행하기 전에 Single 변환됩니다. 이러한 변환은 확대되지만 정밀도 손실이 발생할 수 있습니다.

> ⚠ **Warning**
>
> 정밀도의 차이로 인해 같을 것으로 예상되는 두 개의 **Single** 값이 같지 않은 것으로 판명되어 비교 결과에 영향을 줄 수 있습니다. 두 값을 비교하는 것에 대한 더 많은 정보를 보려면 **Single** 섹션을 참조하세요.

IsNaN, IsInfinity, IsPositiveInfinity및 IsNegativeInfinity 메서드를 호출하여 이러한 특수 값을 테스트할 수도 있습니다.

- **수학 연산**. 더하기, 빼기, 곱하기 및 나누기와 같은 일반적인 산술 연산은 Single 메서드가 아닌 언어 컴파일러 및 CIL(공용 중간 언어) 명령에 의해 구현됩니다. 수학 연산의 다른 피연산자는 Double경우 연산을 수행하기 전에 SingleDouble 변환되고 연산 결과도 Double 값입니다. 다른 피연산자는 정수 계열 형식인 경우 작업을 수행하기 전에 Single 변환되며 작업의 결과도 Single 값입니다.

  `static` 클래스에서 `Shared`(Visual Basic에서는System.Math) 메서드를 호출하여 다른 수학 연산을 수행할 수 있습니다. 여기에는 산술(예: Math.Abs, Math.Sign및 Math.Sqrt), 기하 도형(예: Math.Cos 및 Math.Sin) 및 미적분(예: Math.Log)에 일반적으로 사용되는 추가 메서드가 포함됩니다. 모든 경우에 Single 값은 Double변환됩니다.

  Single 값의 개별 비트를 조작할 수도 있습니다. BitConverter.GetBytes(Single) 메서드는 바이트 배열에서 비트 패턴을 반환합니다. 해당 바이트 배열을 BitConverter.ToInt32 메서드에 전달하면 Single 값의 비트 패턴을 32비트 정수로 유지할 수도 있습니다.

- **반올림**. 반올림은 부동 소수점 표현과 정밀도 문제로 인해 발생하는 값 간의 차이를 줄이는 기술로 자주 사용됩니다. Single 메서드를 호출하여 Math.Round 값을 반올림할 수 있습니다. 그러나 메서드가 호출되기 전에 Single 값이 Double로 변환되며, 이 변환은 정밀도가 손실될 수 있습니다.

- **서식**. Single 메서드를 호출하거나 ToString 기능을 사용하여 값을 문자열 표현으로 변환할 수 있습니다. 서식 문자열이 부동 소수점 값의 문자열 표현을 제어하는 방법에 대한 자세한 내용은 표준 숫자 형식 문자열 및 사용자 지정 숫자 형식 문자열을 참조하세요.

- **문자열**구문 분석. Single 또는 Parse 메서드를 호출하여 부동 소수점 값의 문자열 표현을 TryParse 값으로 변환할 수 있습니다. 구문 분석 작업이 실패하면 Parse 메서드는 예외를 throw하는 반면 TryParse 메서드는 `false`반환합니다.

- **형식 변환**. Single 구조는 두 표준 .NET 데이터 형식 간의 변환을 지원하는 IConvertible 인터페이스에 대한 명시적 인터페이스 구현을 제공합니다. 언어 컴파일러는 DoubleSingle 값으로 변환하는 것을 제외하고 다른 모든 표준 숫자 형식에 대한 값의 암시적 변환도 지원합니다. Double 이외의 표준 숫자 형식의 값을 Single 변환하는 것은 확대 변환이며 캐스팅 연산자 또는 변환 메서드를 사용할 필요가 없습니다.

그러나 32비트 및 64비트 정수 값의 변환에는 정밀도 손실이 포함될 수 있습니다. 다음 표는 32비트, 64비트 및 Double 형식의 정밀도 차이를 나열합니다.

테이블 확장

유형	최대 정밀도(10진수)	내부 정밀도 (소수 자릿수)
Double	15	17
Int32 및 UInt32	10	10
Int64 및 UInt64	19	19
Single	7	9

정밀도 문제는 Single 값으로 변환되는 Double 값에 가장 자주 영향을 줍니다. 다음 예제에서는 값 중 하나가 Double변환되는 단정밀도 부동 소수점 값이므로 동일한 나누기 연산에서 생성된 두 값은 같지 않습니다.

C#

```
Double value1 = 1 / 3.0;
Single sValue2 = 1 / 3.0f;
Double value2 = (Double)sValue2;
Console.WriteLine($"{value1:R} = {value2:R}: {value1.Equals(value2)}");

// The example displays the following output on .NET:
//       0.3333333333333333 = 0.3333333432674408: False
```

Last updated on 2026. 01. 23.

System.Single.CompareTo 메서드들

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

값이 동일해야 동등하다고 간주됩니다. 특히 부동 소수점 값이 여러 수학적 연산에 의존하는 경우 정밀도를 상실하는 것이 일반적이며, 가장 적은 유효 자릿수를 제외하고 값이 거의 동일해야 합니다. 이 때문에 CompareTo 메서드의 반환 값은 때때로 놀라운 것처럼 보일 수 있습니다. 예를 들어 특정 값을 곱한 다음 같은 값을 나누면 원래 값이 생성되지만 다음 예제에서는 계산된 값이 원래 값보다 큰 것으로 나타났습니다. "R" 표준 숫자 형식 문자열 사용하여 두 값의 모든 유효 자릿수를 표시하면 계산된 값이 가장 낮은 유효 자릿수의 원래 값과 다르다는 것을 나타냅니다. 이러한 비교를 처리하는 방법에 대한 자세한 내용은 Equals(Single) 메서드의 설명 섹션을 참조하세요.

비록 값이 NaN인 개체가 값이 NaN인 다른 개체와 같지 않으며 심지어 자기 자신과도 같지 않더라도, IComparable<T> 인터페이스는 `A.CompareTo(A)`이 0을 반환해야 한다고 요구합니다.

CompareTo(System.Object)은 특정 개체와 현재 개체를 비교하는 메서드입니다.

`value` 매개 변수는 `null` 또는 Single; 인스턴스여야 합니다. 그렇지 않으면 예외가 throw 됩니다. 값에 관계없이 Single인스턴스는 `null`보다 큰 것으로 간주됩니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
       float value1 = 16.5457f;
       float operand = 3.8899982f;
       object value2 = value1 * operand / operand;
       Console.WriteLine($"Comparing {value1} and {value2}:
{value1.CompareTo(value2)}");
       Console.WriteLine();
       Console.WriteLine($"Comparing {value1:R} and {value2:R}:
{value1.CompareTo(value2)}");
   }
}
// The example displays the following output:
//       Comparing 16.5457 and 16.5457: -1
```

```
//
//       Comparing 16.5457 and 16.545702: -1
```

이 메서드는 IComparable 인터페이스를 지원하도록 구현됩니다.

CompareTo(System.Single)

이 메서드는 System.IComparable<T> 인터페이스를 구현하고 `value` 매개 변수를 개체로 변환할 필요가 없으므로 Single.CompareTo(Object) 오버로드보다 약간 더 잘 수행됩니다.

C#

```
using System;

public class Example2
{
   public static void Main()
   {
       float value1 = 16.5457f;
       float operand = 3.8899982f;
       float value2 = value1 * operand / operand;
       Console.WriteLine($"Comparing {value1} and {value2}:
{value1.CompareTo(value2)}");
       Console.WriteLine();
       Console.WriteLine($"Comparing {value1:R} and {value2:R}:
{value1.CompareTo(value2)}");
   }
}
// The example displays the following output:
//       Comparing 16.5457 and 16.5457: -1
//
//       Comparing 16.5457 and 16.545702: -1
```

확대 변환

프로그래밍 언어에 따라 매개 변수 형식이 인스턴스 형식보다 적은 비트(더 좁은)가 있는 CompareTo 메서드를 코딩할 수 있습니다. 일부 프로그래밍 언어는 인스턴스만큼 많은 비트가 있는 형식으로 매개 변수를 나타내는 암시적 확대 변환을 수행하기 때문에 가능합니다.

예를 들어 인스턴스 형식이 Single 매개 변수 형식이 Int32가정해 보겠습니다. Microsoft C# 컴파일러는 매개 변수 값을 Single 개체로 나타내는 명령을 생성한 다음 인스턴스의 값과 매개 변수의 확장된 표현을 비교하는 Single.CompareTo(Single) 메서드를 생성합니다.

프로그래밍 언어 설명서를 참조하여 컴파일러가 숫자 형식의 암시적 확대 변환을 수행하는지 확인합니다. 자세한 내용은 형식 변환 테이블 항목을 참조하세요.

비교의 정밀도

문서화된 전체 자릿수를 초과하는 부동 소수점 숫자의 정밀도는 .NET의 구현 및 버전과 관련이 있습니다. 따라서 숫자의 내부 표현의 정밀도가 변경될 수 있기 때문에 .NET 버전 간에 두 개의 특정 숫자를 비교하면 결과가 달라질 수 있습니다.

System.Single.Epsilon 속성

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Epsilon 속성 값은 Single 인스턴스의 값이 0일 경우 숫자 연산 또는 비교에서 중요한 가장 작은 양의 Single 값을 반영합니다. 예를 들어 다음 코드에서는 0과 Epsilon 같지 않은 값으로 간주되는 반면, Epsilon 값의 0과 절반은 같은 것으로 간주됩니다.

C#

```
using System;

public class Example1
{
   public static void Main()
   {
      float[] values = { 0f, Single.Epsilon, Single.Epsilon * .5f };

      for (int ctr = 0; ctr <= values.Length - 2; ctr++)
      {
         for (int ctr2 = ctr + 1; ctr2 <= values.Length - 1; ctr2++)
         {
            Console.WriteLine($"{values[ctr]:r} = {values[ctr2]:r}:
{values[ctr].Equals(values[ctr2])}");
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       0 = 1.401298E-45: False
//       0 = 0: True
//
//       1.401298E-45 = 0: False
```

보다 정확하게 말하자면, 단정밀도 부동 소수점 형식은 부호, 23비트 매니티사 또는 유의성 및 8비트 지수로 구성됩니다. 다음 예제에서 알 수 있듯이 0에는 지수가 -126 및 매니티사는 0입니다. Epsilon 에는 -126 지수와 1의 가수가 있습니다. 즉, Single.Epsilon 0보다 크고 가능한 가장 작은 값과 지수가 -126인 값에 대해 Single 가능한 가장 작은 증분을 나타내는 가장 작은 양의 Single 값입니다.

C#

```
using System;

public class Example2
{
   public static void Main()
```

```
   {
      float[] values = { 0.0f, Single.Epsilon };
      foreach (var value in values) {
         Console.WriteLine(GetComponentParts(value));
         Console.WriteLine();
      }
   }

   private static string GetComponentParts(float value)
   {
      string result = String.Format("{0:R}: ", value);
      int indent = result.Length;

      // Convert the single to a 4-byte array.
      byte[] bytes = BitConverter.GetBytes(value);
      int formattedSingle = BitConverter.ToInt32(bytes, 0);

      // Get the sign bit (byte 3, bit 7).
      result += String.Format("Sign: {0}\n",
                              (formattedSingle >> 31) != 0 ? "1 (-)" : "0 (+)");

      // Get the exponent (byte 2 bit 7 to byte 3, bits 6)
      int exponent =  (formattedSingle >> 23) & 0x000000FF;
      int adjustment = (exponent != 0) ? 127 : 126;
      result += String.Format("{0}Exponent: 0x{1:X4} ({1})\n", new String(' ',
indent), exponent - adjustment);

      // Get the significand (bits 0-22)
      long significand = exponent != 0 ?
                         ((formattedSingle & 0x007FFFFF) | 0x800000) :
                         (formattedSingle & 0x007FFFFF);
      result += String.Format("{0}Mantissa: 0x{1:X13}\n", new String(' ', indent),
significand);
      return result;
   }
}
//       // The example displays the following output:
//       0: Sign: 0 (+)
//          Exponent: 0xFFFFFF82 (-126)
//          Mantissa: 0x0000000000000
//
//
//       1.401298E-45: Sign: 0 (+)
//                     Exponent: 0xFFFFFF82 (-126)
//                     Mantissa: 0x0000000000001
```

그러나 Epsilon 속성은 형식의 Single 전체 자릿수에 대한 일반적인 측정값이 아니며 값이 0인 인스턴스에 Single 만 적용됩니다.

> ⓘ **참고**

Epsilon 속성의 값은 부동 소수점 산술 연산의 반올림으로 인한 상대 오차의 상한을 나타내는 컴퓨터 엡실론과 동일하지 않습니다.

이 상수의 값은 1.4e-45입니다.

두 개의 분명히 동등한 부동 소수점 숫자는 가장 낮은 유효 자릿수의 차이로 인해 같지 않을 수 있습니다. 예를 들어, C# 식(`(float)1/3 == (float)0.33333`)은 왼쪽의 나누기 연산이 최대 정밀도를 가지지만, 오른쪽의 상수는 정확도가 지정된 자릿수로 제한되어 있기 때문에 동일하게 비교되지 않습니다. 두 부동 소수점 숫자를 같음으로 간주할 수 있는지 여부를 결정하는 사용자 지정 알고리즘을 만드는 경우 상수보다 Epsilon 큰 값을 사용하여 두 값이 같은 것으로 간주될 수 있는 절대 차이 여백을 설정해야 합니다. (일반적으로 차이의 여백은 Epsilon보다 여러 배 더 큽니다.)

플랫폼 참고 사항

ARM 시스템에서는 Epsilon 상수의 값이 너무 작아 감지할 수 없으므로 0과 같습니다. 대신 1.175494351E-38과 같은 대체 엡실론 값을 정의할 수 있습니다.

System.Single.Equals 메서드

아티클 • 2025. 03. 26.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Single.Equals(Single) 메서드는 System.IEquatable<T> 인터페이스를 구현하고 `obj` 매개 변수를 개체로 변환할 필요가 없으므로 Single.Equals(Object) 보다 약간 더 잘 수행합니다.

확대 변환

프로그래밍 언어에 따라 매개 변수 형식에 인스턴스 형식보다 적은 비트(더 좁은)가 있는 Equals 메서드를 코딩할 수 있습니다. 일부 프로그래밍 언어는 인스턴스만큼 많은 비트가 있는 형식으로 매개 변수를 나타내는 암시적 확대 변환을 수행하기 때문에 가능합니다.

예를 들어 인스턴스 형식이 Single 매개 변수 형식이 Int32가정해 보겠습니다. Microsoft C# 컴파일러는 매개 변수 값을 Single 개체로 나타내는 명령을 생성한 다음 인스턴스 값과 매개 변수의 확장된 표현을 비교하는 Single.Equals(Single) 메서드를 생성합니다.

프로그래밍 언어 설명서를 참조하여 컴파일러가 숫자 형식의 암시적 확대 변환을 수행하는지 확인합니다. 자세한 내용은 형식 변환 테이블참조하세요.

비교의 정밀도

Equals 메서드는 사용 시 주의가 필요합니다. 겉보기에는 동일해 보이는 두 값이 서로 다를 수 있기 때문입니다. 이 차이는 두 값 간의 정밀도의 차이에 의해 발생할 수 있습니다. 다음 예제에서는 Single 값 .3333과 1을 3으로 나누어 반환된 Single 같지 않은 것으로 보고합니다.

C#

```
// Initialize two floats with apparently identical values
float float1 = .33333f;
float float2 = 1/3;
// Compare them for equality
Console.WriteLine(float1.Equals(float2));    // displays false
```

같음 비교와 관련된 문제를 방지하는 한 가지 비교 기술에는 두 값 간의 허용 가능한 차이 여백(예: 값 중 하나의 .01%)을 정의하는 작업이 포함됩니다. 두 값 간의 차이의 절대값이 해당 여백보다 작거나 같으면 이 차이는 정밀도 차이의 결과일 수 있으므로 값이 같을 가

능성이 높습니다. 다음 예제에서는 이 기술을 사용하여 이전 코드 예제에서 같지 않은 것으로 확인된 두 Single 값인 .33333과 1/3을 비교합니다.

C#

```
// Initialize two floats with apparently identical values
float float1 = .33333f;
float float2 = (float) 1/3;
// Define the tolerance for variation in their values
float difference = Math.Abs(float1 * .0001f);

// Compare the values
// The output to the console indicates that the two values are equal
if (Math.Abs(float1 - float2) <= difference)
   Console.WriteLine("float1 and float2 are equal.");
else
   Console.WriteLine("float1 and float2 are unequal.");
```

이 경우 값은 같습니다.

> ⓘ **참고**
>
> Epsilon 범위가 0에 가까운 양수 값의 최소 식을 정의하므로 차이의 여백은 Epsilon 보다 커야 합니다. 일반적으로 Epsilon보다 여러 배 더 큽 수 있습니다. 따라서 같음을 위해 Double 값을 비교할 때 Epsilon 사용하지 않는 것이 좋습니다.

같음 비교와 관련된 문제를 방지하는 두 번째 기술은 두 부동 소수점 숫자 간의 차이를 절대값과 비교하는 것입니다. 차이가 절대값보다 작거나 같으면 숫자는 같습니다. 값이 크면 숫자가 같지 않습니다. 이 작업을 수행하는 한 가지 방법은 임의로 절대값을 선택하는 것입니다. 그러나 허용되는 차이의 여백은 Single 값의 크기에 따라 달라지므로 문제가 됩니다. 두 번째 방법은 부동 소수점 형식의 설계상의 기능을 활용합니다. 두 부동 소수점 값의 정수 표현에서 가수 요소 간의 차이는 두 값을 분리하는 가능한 부동 소수점 값의 수를 나타냅니다. 예를 들어 값이 0인 Single 사용할 때 Epsilon 가장 작은 표현 가능한 값이므로 0.0과 Epsilon 간의 차이는 1입니다. 다음 예제에서는 이 기술을 사용하여 이전 코드 예제와 Equals(Single) 메서드가 같지 않은 것으로 확인된 두 Double 값인 .33333과 1/3을 비교합니다. 이 예제에서는 BitConverter.GetBytes 및 BitConverter.ToInt32 메서드를 사용하여 단정밀도 부동 소수점 값을 정수 표현으로 변환합니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
```

```
      float value1 = .1f * 10f;
      float value2 = 0f;
      for (int ctr = 0; ctr < 10; ctr++)
         value2 += .1f;

      Console.WriteLine($"{value1:R} = {value2:R}:
{HasMinimalDifference(value1, value2, 1)}");
   }

   public static bool HasMinimalDifference(float value1, float value2, int
units)
   {
      byte[] bytes = BitConverter.GetBytes(value1);
      int iValue1 = BitConverter.ToInt32(bytes, 0);

      bytes = BitConverter.GetBytes(value2);
      int iValue2 = BitConverter.ToInt32(bytes, 0);

      // If the signs are different, return false except for +0 and -0.
      if ((iValue1 >> 31) != (iValue2 >> 31))
      {
         if (value1 == value2)
            return true;

         return false;
      }

      int diff = Math.Abs(iValue1 - iValue2);

      if (diff <= units)
         return true;

      return false;
   }
}
// The example displays the following output:
//        1 = 1.00000012: True
```

문서화된 전체 자릿수를 초과하는 부동 소수점 숫자의 정밀도는 .NET의 구현 및 버전과 관련이 있습니다. 따라서 두 숫자를 비교하면 숫자의 내부 표현 정밀도가 변경될 수 있으므로 .NET 버전에 따라 다른 결과가 생성될 수 있습니다.

날짜, 시간 및 표준 시간대

아티클 • 2024. 12. 14.

.NET은 날짜, 시간 및 표준 시간대를 나타내는 형식을 제공합니다. 이 문서에서는 이러한 형식에 대해 설명합니다.

.NET은 기본 DateTime 구조 외에도 표준 시간대 작업을 지원하는 다음 클래스를 제공합니다.

- TimeZoneInfo

  이 클래스를 사용하여 시스템에 미리 정의된 모든 표준 시간대로 작업하고, 새 표준 시간대를 만들고, 날짜와 시간을 한 표준 시간대에서 다른 표준 시간대로 쉽게 변환할 수 있습니다. 새 개발의 경우 TimeZoneInfo 클래스 대신 TimeZone 클래스를 사용합니다.

- DateTimeOffset

  UTC의 오프셋(또는 차이)이 알려진 날짜 및 시간을 사용하려면 이 구조를 사용합니다. DateTimeOffset 구조체는 날짜 및 시간 값을 UTC의 해당 시간 오프셋과 결합합니다. UTC와의 관계 때문에 개별 날짜 및 시간 값은 단일 시점을 명확하게 식별합니다. 이렇게 하면 DateTimeOffset 값이 DateTime 값보다 한 컴퓨터에서 다른 컴퓨터로 이식할 수 있습니다.

시간 작업을 지원하는 클래스는 다음과 같습니다.

- TimeSpan

  경과된 시간 또는 두 날짜 간의 차이와 같은 시간 간격을 나타내려면 이 구조를 사용합니다.

- TimeOnly

  날짜가 없는 시간을 나타내려면 이 구조를 사용합니다. 시간은 특정이 아닌 일의 시간, 분 및 초를 나타냅니다. `TimeOnly`은(는) `00:00:00.0000000`에서 `23:59:59.9999999`까지의 범위를 가집니다. 이 형식은 해당 형식을 사용하여 시간을 나타낼 때 코드의 `DateTime` 및 `TimeSpan` 형식을 바꾸는 데 사용할 수 있습니다. 자세한 내용은 DateOnly 및 TimeOnly 구조사용하는 방법을 참조하세요.

  > **ⓘ 중요**
  >
  > TimeOnly .NET Framework에 사용할 수 없습니다.

- TimeProvider

  시간 추상화 기능을 제공하는 기본 클래스입니다. 현재 시간을 확인하는 일반적인 방법은 `DateTime.UtcNow` 또는 `DateTimeOffset.UtcNow`사용하는 것입니다. 그러나 이러한 형식은 "지금"으로 간주되는 항목에 대한 제어를 제공하지 않습니다. 왜 그것을 통제하고 싶습니까? 테스트 용이성. 예를 들어 이벤트 1일 전에 미리 알림을 제공하는 이벤트 추적 애플리케이션을 작성하는 것이 좋습니다. 앱의 논리는 매시간 이벤트 시간을 확인하고 이벤트 24시간 전에 사용자에게 경고하는 것입니다. 앱에 대한 테스트를 작성할 때, 이 논리를 테스트하기 위해 자체적으로 `DateTimeOffset.UtcNow`을 래핑하는 형식을 제공해야 하지만 이제는 .NET에서 이 추상화 클래스를 제공합니다.

  자세한 내용은 What is TimeProvider를 참조하세요.

  `TimeProvider` 형식은 .NET에 포함됩니다.

  .NET Framework 및 .NET Standard의 경우 `TimeProvider`은 **Microsoft.Bcl.TimeProvider** NuGet 패키지 에 의해 제공됩니다.

날짜 작업을 지원하는 다음 클래스는 다음과 같습니다.

- DateOnly

  날짜만 나타내는 값으로 작업할 때 이 구조를 사용합니다. 날짜는 하루의 시작부터 끝까지의 전체 일을 나타냅니다. `DateOnly`는 `0001-01-01`에서 `9999-12-31`까지의 범위가 있습니다. 또한 이 형식은 특정 시간 없이 월, 일 및 연도 조합을 나타냅니다. 이전에 코드에서 `DateTime` 형식을 사용하여 시간을 무시한 날짜를 나타내는 경우 해당 위치에서 이 형식을 사용합니다. 자세한 내용은 DateOnly 및 TimeOnly 구조사용하는 방법을 참조하세요.

  > **ⓘ 중요**
  >
  > DateOnly .NET Framework에 사용할 수 없습니다.

다음 섹션에서는 표준 시간대를 사용하고 날짜와 시간을 표준 시간대에서 다른 표준 시간대로 변환할 수 있는 표준 시간대 인식 애플리케이션을 만드는 데 필요한 정보를 제공합니다.

이 섹션에서는

표준 시간대 개요
표준 시간대 인식 애플리케이션을 만드는 데 관련된 용어, 개념 및 문제에 대해 설명합니

다.

DateTime, DateTimeOffset, TimeSpan 및 TimeZoneInfo 중에서 선택
날짜 및 시간 데이터로 작업할 때 DateTime, DateTimeOffset및 TimeZoneInfo 형식을 사용하는 경우에 대해 설명합니다.

로컬 시스템에서 정의된 표준 시간대 찾기
로컬 시스템에 있는 표준 시간대를 열거하는 방법을 설명합니다.

사용법 안내: 컴퓨터에 존재하는 표준 시간대 열거
컴퓨터 레지스트리에 정의된 표준 시간대를 열거하고 사용자가 목록에서 미리 정의된 표준 시간대를 선택할 수 있도록 하는 예제를 제공합니다.

방법: 미리 정의된 UTC 및 현지 표준 시간대 개체에 액세스
협정 세계시 및 현지 표준 시간대에 액세스하는 방법을 설명합니다.

TimeZoneInfo 개체 인스턴스화하는 방법
로컬 시스템 레지스트리에서 TimeZoneInfo 개체를 인스턴스화하는 방법을 설명합니다.

DateTimeOffset 개체 인스턴스화
DateTimeOffset 개체를 인스턴스화하는 방법과 DateTime 값을 DateTimeOffset 값으로 변환할 수 있는 방법에 대해 설명합니다.

방법: 조정 규칙 없이 표준 시간대 만들기
일광 절약 시간제로의 전환을 지원하지 않는 사용자 지정 표준 시간대를 만드는 방법을 설명합니다.

방법: 조정 규칙을 사용하여 표준 시간대 만들기
사용자 정의 시간대를 생성하여 일광 절약 시간제로의 전환과 그로부터의 전환을 지원하는 방법을 설명합니다.

시간대 저장 및 복원
표준 시간대 데이터의 직렬화 및 역직렬화에 대한 TimeZoneInfo 지원에 대해 설명하고 이러한 기능을 사용할 수 있는 몇 가지 시나리오를 보여 줍니다.

방법: 포함된 리소스에 표준 시간대 저장하는 방법
사용자 지정 표준 시간대를 만들고 해당 정보를 리소스 파일에 저장하는 방법을 설명합니다.

방법: 포함된 리소스에서 표준 시간대를 복원하는 방법
포함된 리소스 파일에 저장된 사용자 지정 표준 시간대를 인스턴스화하는 방법을 설명합니다.

날짜 및 시간 사용하여 산술 연산 수행
DateTime 및 DateTimeOffset 값 추가, 빼기 및 비교와 관련된 문제에 대해 설명합니다.

방법: 날짜 및 시간 계산에서 표준 시간대를 사용하는 방법
표준 시간대의 조정 규칙을 반영하는 날짜 및 시간 산술 연산을 수행하는 방법에 대해 설명합니다.

DateTime과 DateTimeOffset 간의 변환
DateTime 값과 DateTimeOffset 값 간에 변환하는 방법을 설명합니다.

시간대 사이 시간 변환
시간을 한 시간대에서 다른 시간대로 변환하는 방법을 설명합니다.

모호한 시간을 해결하는 방법
표준 시간대로 매핑하여 모호한 시간을 해결하는 방법을 설명합니다.

방법: 사용자가 모호한 시간을 해결하도록 허용하기
사용자가 모호한 현지 시간과 협정 세계시 사이의 매핑을 결정할 수 있도록 하는 방법을 설명합니다.

참조

System.TimeZoneInfo

DateTime, DateOnly, DateTimeOffset, TimeSpan, TimeOnly 및 TimeZoneInfo 중에서 선택

아티클 • 2024. 12. 15.

.NET 애플리케이션은 여러 가지 방법으로 날짜 및 시간 정보를 사용할 수 있습니다. 날짜 및 시간 정보의 일반적인 용도는 다음과 같습니다.

- 날짜만 반영하기 때문에 시간 정보가 중요하지 않습니다.
- 날짜 정보가 중요하지 않도록 시간만 반영합니다.
- 특정 시간과 장소에 연결되지 않은 추상적인 날짜 및 시간을 반영합니다(예: 국제 체인의 대부분의 매장은 평일 오전 9시에 영업).
- .NET 외부의 원본에서 날짜 및 시간 정보를 검색하려면 일반적으로 날짜 및 시간 정보가 간단한 데이터 형식으로 저장됩니다.
- 단일 시점을 고유하고 명확하게 식별합니다. 일부 애플리케이션에서는 호스트 시스템에서만 날짜와 시간이 모호해야 합니다. 다른 앱에서는 시스템 간에 모호하지 않도록 하는 것이 필요합니다. 즉, 한 시스템에서 직렬화된 날짜가 다른 시스템에서 의미 있게 역직렬화되어 세계 어디서나 사용될 수 있습니다.
- 여러 관련 시간(예: 요청자의 현지 시간 및 웹 요청에 대한 서버의 수신 시간)을 유지합니다.
- 날짜 및 시간 연산을 수행하여, 하나의 시점을 명확하고 고유하게 식별할 수 있는 결과를 얻을 수 있습니다.

.NET에는 날짜 및 시간을 사용하는 애플리케이션을 빌드하는 데 사용할 수 있는 DateTime, DateOnly, DateTimeOffset, TimeSpan, TimeOnly및 TimeZoneInfo 형식이 포함됩니다.

> **참고**
>
> 이 문서에서는 해당 기능이 거의 전적으로 TimeZone 클래스에 통합되어 있으므로 TimeZoneInfo 설명하지 않습니다. 가능하면 TimeZoneInfo 클래스 대신 TimeZone 클래스를 사용합니다.

DateTimeOffset 구조체

DateTimeOffset 구조체는 날짜 및 시간 값을 UTC와 얼마나 다른지 나타내는 오프셋과 함께 나타냅니다. 따라서 값은 항상 단일 시점을 명확하게 식별합니다.

DateTimeOffset 형식에는 표준 시간대 인식과 함께 DateTime 형식의 모든 기능이 포함됩니다. 이렇게 하면 다음과 같은 애플리케이션에 적합합니다.

- 단일 시점을 고유하고 명확하게 식별합니다. DateTimeOffset 형식을 사용하여 "now"의 의미를 명확하게 정의하고, 트랜잭션 시간을 기록하고, 시스템 또는 애플리케이션 이벤트의 시간을 기록하고, 파일 생성 및 수정 시간을 기록할 수 있습니다.
- 일반 날짜 및 시간 산술 연산을 수행합니다.
- 이러한 시간이 두 개의 별도 값 또는 구조체의 두 멤버로 저장되어 있는 한, 관련된 여러 시간을 보존합니다.

> ⓘ **참고**
>
> DateTimeOffset 값에 대한 이러한 사용은 DateTime 값보다 훨씬 일반적입니다. 따라서 DateTimeOffset 애플리케이션 개발의 기본 날짜 및 시간 유형으로 간주합니다.

DateTimeOffset 값은 특정 표준 시간대에 연결되지 않지만 다양한 표준 시간대에서 비롯할 수 있습니다. 다음 예제에서는 여러 DateTimeOffset 값(현지 태평양 표준시 포함)이 속할 수 있는 표준 시간대를 나열합니다.

C#

```
using System;
using System.Collections.ObjectModel;

public class TimeOffsets
{
   public static void Main()
   {
      DateTime thisDate = new DateTime(2007, 3, 10, 0, 0, 0);
      DateTime dstDate = new DateTime(2007, 6, 10, 0, 0, 0);
      DateTimeOffset thisTime;

      thisTime = new DateTimeOffset(dstDate, new TimeSpan(-7, 0, 0));
      ShowPossibleTimeZones(thisTime);

      thisTime = new DateTimeOffset(thisDate, new TimeSpan(-6, 0, 0));
      ShowPossibleTimeZones(thisTime);

      thisTime = new DateTimeOffset(thisDate, new TimeSpan(+1, 0, 0));
      ShowPossibleTimeZones(thisTime);
   }

   private static void ShowPossibleTimeZones(DateTimeOffset offsetTime)
   {
      TimeSpan offset = offsetTime.Offset;
      ReadOnlyCollection<TimeZoneInfo> timeZones;

      Console.WriteLine("{0} could belong to the following time zones:",
```

```
                        offsetTime.ToString());
      // Get all time zones defined on local system
      timeZones = TimeZoneInfo.GetSystemTimeZones();
      // Iterate time zones
      foreach (TimeZoneInfo timeZone in timeZones)
      {
         // Compare offset with offset for that date in that time zone
         if (timeZone.GetUtcOffset(offsetTime.DateTime).Equals(offset))
            Console.WriteLine("   {0}", timeZone.DisplayName);
      }
      Console.WriteLine();
   }
}
// This example displays the following output to the console:
//       6/10/2007 12:00:00 AM -07:00 could belong to the following time
zones:
//          (GMT-07:00) Arizona
//          (GMT-08:00) Pacific Time (US & Canada)
//          (GMT-08:00) Tijuana, Baja California
//
//       3/10/2007 12:00:00 AM -06:00 could belong to the following time
zones:
//          (GMT-06:00) Central America
//          (GMT-06:00) Central Time (US & Canada)
//          (GMT-06:00) Guadalajara, Mexico City, Monterrey - New
//          (GMT-06:00) Guadalajara, Mexico City, Monterrey - Old
//          (GMT-06:00) Saskatchewan
//
//       3/10/2007 12:00:00 AM +01:00 could belong to the following time
zones:
//          (GMT+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna
//          (GMT+01:00) Belgrade, Bratislava, Budapest, Ljubljana, Prague
//          (GMT+01:00) Brussels, Copenhagen, Madrid, Paris
//          (GMT+01:00) Sarajevo, Skopje, Warsaw, Zagreb
//          (GMT+01:00) West Central Africa
```

출력은 이 예제의 각 날짜 및 시간 값이 세 개 이상의 서로 다른 표준 시간대에 속할 수 있음을 보여 줍니다. 2007년 6월 10일의 DateTimeOffset 값은 날짜 및 시간 값이 서머타임을 나타내는 경우 그 UTC 오프셋이 원래 시간대의 기본 UTC 오프셋이나 표시 이름에 나와 있는 UTC 오프셋과 반드시 일치하지 않을 수 있음을 보여 줍니다. 단일 DateTimeOffset 값은 표준 시간대와 긴밀하게 결합되지 않으므로 일광 절약 시간제로의 표준 시간대 전환을 반영할 수 없습니다. 날짜 및 시간 산술 연산을 사용하여 DateTimeOffset 값을 조작할 때 문제가 될 수 있습니다. 시간대의 조정 규칙을 고려하여 날짜 및 시간 산술 연산을 수행하는 방법에 대한 논의는 날짜 및 시간에 대한 산술 연산 수행을 참조하세요.

DateTime 구조체

DateTime 값은 특정 날짜 및 시간을 정의합니다. 여기에는 해당 날짜와 시간이 속한 표준 시간대에 대한 제한된 정보를 제공하는 Kind 속성이 포함되어 있습니다. DateTimeKind 속성에서 반환된 Kind 값은 DateTime 값이 현지 시간(DateTimeKind.Local), UTC(협정 세계시)(DateTimeKind.Utc) 또는 지정되지 않은 시간(DateTimeKind.Unspecified)을 나타내는지 여부를 나타냅니다.

DateTime 구조는 다음 특성 중 하나 이상을 가진 애플리케이션에 적합합니다.

- 추상 날짜 및 시간을 사용합니다.
- 시간대 정보가 없는 날짜 및 시간을 처리합니다.
- UTC 날짜 및 시간만 사용합니다.
- 날짜와 시간 계산을 수행하지만, 일반적인 결과에 중점을 둡니다. 예를 들어 특정 날짜 및 시간에 6개월을 추가하는 추가 작업에서는 일광 절약 시간제에 맞게 결과를 조정할지 여부는 중요하지 않은 경우가 많습니다.

특정 DateTime 값이 UTC를 나타내지 않는 한 해당 날짜 및 시간 값은 종종 모호하거나 이식성이 제한됩니다. 예를 들어 DateTime 값이 현지 시간을 나타내는 경우 해당 현지 표준 시간대 내에서 이식 가능합니다(즉, 동일한 표준 시간대의 다른 시스템에서 값이 역직렬화되는 경우 해당 값은 여전히 단일 시점을 명확하게 식별함). 현지 표준 시간대 외부에서 해당 DateTime 값은 여러 해석을 가질 수 있습니다. 값의 Kind 속성이 DateTimeKind.Unspecified인 경우, 이식성이 더욱 떨어집니다. 이제 동일한 표준 시간대 내에서도 모호할 수 있으며, 심지어 처음 직렬화된 동일한 시스템에서도 모호할 수 있습니다. DateTime 값이 UTC를 나타내는 경우에만 해당 값은 값이 사용되는 시스템 또는 표준 시간대에 관계없이 단일 시점을 명확하게 식별합니다.

> **ⓘ 중요**
>
> DateTime 데이터를 저장하거나 공유할 때 UTC를 사용하고 DateTime 값의 Kind 속성을 DateTimeKind.Utc설정합니다.

DateOnly 구조체

DateOnly 구조체는 시간 없이 특정 날짜를 나타냅니다. 시간 구성 요소가 없으므로 하루의 시작부터 종료까지의 날짜를 나타냅니다. 이 구조는 생년월일, 기념일, 휴일 또는 비즈니스 관련 날짜와 같은 특정 날짜를 저장하는 데 적합합니다.

시간 구성 요소를 무시하고 `DateTime`를 사용할 수도 있지만, `DateOnly`대신 `DateTime`을 사용하는 데에는 몇 가지 이점이 있습니다.

- `DateTime` 구조체가 표준 시간대로 오프셋된 경우 이전 또는 다음 날로 롤아웃될 수 있습니다. `DateOnly` 표준 시간대로 오프셋할 수 없으며 항상 설정된 날짜를 나타냅

니다.

- `DateTime` 구조체 직렬화에는 데이터의 의도를 모호하게 할 수 있는 시간 구성 요소가 포함됩니다. 또한 `DateOnly` 적은 데이터를 직렬화합니다.
- 코드가 SQL Server와 같은 데이터베이스와 상호 작용하는 경우 전체 날짜는 일반적으로 시간을 포함하지 않는 `date` 데이터 형식으로 저장됩니다. `DateOnly` 데이터베이스 형식과 더 잘 일치합니다.

`DateOnly`대한 자세한 내용은 DateOnly 및 TimeOnly 구조체사용하는 방법을 참조하세요.

> **ⓘ 중요**
>
> `DateOnly` .NET Framework에 사용할 수 없습니다.

TimeSpan 구조체

TimeSpan 구조체는 시간 간격을 나타냅니다. 일반적인 두 가지 용도는 다음과 같습니다.

- 두 날짜와 시간 값 사이의 시간 간격을 반영합니다. 예를 들어 한 DateTime 값을 다른 값에서 빼면 TimeSpan 값이 반환됩니다.
- 경과된 시간 측정 예를 들어 Stopwatch.Elapsed 속성은 경과된 시간을 측정하기 시작하는 TimeSpan 메서드 중 하나에 대한 호출 이후 경과된 시간 간격을 반영하는 Stopwatch 값을 반환합니다.

TimeSpan 값은 해당 값이 특정 날짜에 대한 참조 없이 시간을 반영하는 경우 DateTime 값의 대체로 사용할 수도 있습니다. 이 사용은 날짜에 대한 참조 없이 시간을 나타내는 DateTime.TimeOfDay 값을 반환하는 DateTimeOffset.TimeOfDay 및 TimeSpan 속성과 비슷합니다. 예를 들어 TimeSpan 구조체를 사용하여 매장의 일별 영업 또는 폐점 시간을 반영하거나 일반 이벤트가 발생하는 시간을 나타내는 데 사용할 수 있습니다.

다음 예제에서는 저장소 열기 및 닫는 시간에 대한 `StoreInfo` 개체와 저장소의 표준 시간대를 나타내는 TimeSpan 개체를 포함하는 TimeZoneInfo 구조를 정의합니다. 또한 구조에는 `IsOpenNow` 및 `IsOpenAt`두 가지 메서드가 포함되어 있으며, 이는 로컬 표준 시간대에 있는 것으로 간주되는 사용자가 지정한 시간에 저장소가 열려 있는지 여부를 나타냅니다.

C#

```
using System;

public struct StoreInfo
{
   public String store;
   public TimeZoneInfo tz;
   public TimeSpan open;
```

```
   public TimeSpan close;

   public bool IsOpenNow()
   {
      return IsOpenAt(DateTime.Now.TimeOfDay);
   }

   public bool IsOpenAt(TimeSpan time)
   {
      TimeZoneInfo local = TimeZoneInfo.Local;
      TimeSpan offset = TimeZoneInfo.Local.BaseUtcOffset;

      // Is the store in the same time zone?
      if (tz.Equals(local)) {
         return time >= open & time <= close;
      }
      else {
         TimeSpan delta = TimeSpan.Zero;
         TimeSpan storeDelta = TimeSpan.Zero;

         // Is it daylight saving time in either time zone?
         if (local.IsDaylightSavingTime(DateTime.Now.Date + time))
            delta = local.GetAdjustmentRules()
[local.GetAdjustmentRules().Length - 1].DaylightDelta;

         if
(tz.IsDaylightSavingTime(TimeZoneInfo.ConvertTime(DateTime.Now.Date + time,
local, tz)))
            storeDelta = tz.GetAdjustmentRules()
[tz.GetAdjustmentRules().Length - 1].DaylightDelta;

         TimeSpan comparisonTime = time + (offset -
tz.BaseUtcOffset).Negate() + (delta - storeDelta).Negate();
         return comparisonTime >= open && comparisonTime <= close;
      }
   }
}
```

`StoreInfo` 구조체는 다음과 같은 클라이언트 코드에서 사용할 수 있습니다.

C#

```
public class Example
{
   public static void Main()
   {
      // Instantiate a StoreInfo object.
      var store103 = new StoreInfo();
      store103.store = "Store #103";
      store103.tz = TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard
Time");
      // Store opens at 8:00.
      store103.open = new TimeSpan(8, 0, 0);
```

```
      // Store closes at 9:30.
      store103.close = new TimeSpan(21, 30, 0);

      Console.WriteLine("Store is open now at {0}: {1}",
                        DateTime.Now.TimeOfDay, store103.IsOpenNow());
      TimeSpan[] times = { new TimeSpan(8, 0, 0), new TimeSpan(21, 0, 0),
                           new TimeSpan(4, 59, 0), new TimeSpan(18, 31, 0)
};
      foreach (var time in times)
         Console.WriteLine("Store is open at {0}: {1}",
                           time, store103.IsOpenAt(time));
   }
}
// The example displays the following output:
//       Store is open now at 15:29:01.6129911: True
//       Store is open at 08:00:00: True
//       Store is open at 21:00:00: True
//       Store is open at 04:59:00: False
//       Store is open at 18:31:00: True
```

TimeOnly 구조체

TimeOnly 구조는 매일 알람 시계 또는 매일 점심을 먹는 시간과 같은 하루 중 시간 값을 나타냅니다. `TimeOnly`은 특정 시간인 **00:00:00.0000000 - 23:59:59.9999999**범위로 제한됩니다.

`TimeOnly` 형식이 도입되기 전에 프로그래머는 일반적으로 DateTime 형식 또는 TimeSpan 형식을 사용하여 특정 시간을 나타냅니다. 그러나 이러한 구조를 사용하여 날짜가 없는 시간을 시뮬레이션하면 몇 가지 문제가 발생할 수 있습니다. 이러한 문제는 `TimeOnly`가 해결합니다.

- `TimeSpan` 스톱워치로 측정된 시간과 같은 경과된 시간을 나타냅니다. 상한 범위는 29,000년 이상이며 해당 값은 시간이 지나면 뒤로 이동함을 나타내기 위해 음수일 수 있습니다. `TimeSpan`이 음수일 경우, 하루 중 특정 시간을 나타내지 않습니다.
- `TimeSpan` 하루 중 시간으로 사용되는 경우 24시간 외의 값으로 조작할 수 있는 위험이 있습니다. `TimeOnly` 이러한 위험이 없습니다. 예를 들어 직원의 근무 교대 근무가 18:00에 시작되고 8시간 동안 지속되는 경우 `TimeOnly` 구조에 8시간을 추가하면 2:00으로 롤오버됩니다.
- 하루 중 시간에 `DateTime` 사용하려면 임의의 날짜를 시간과 연결한 다음 나중에 무시해야 합니다. `DateTime.MinValue`(0001-01-01)를 날짜로 선택하는 것이 일반적이지만 `DateTime` 값에서 시간을 빼면 `OutOfRange` 예외가 발생할 수 있습니다. `TimeOnly`는 시간이 24시간 동안 앞뒤로 조정될 수 있기 때문에 이 문제가 없습니다.
- `DateTime` 구조체 직렬화에는 데이터의 의도를 모호하게 할 수 있는 날짜 구성 요소가 포함됩니다. 또한 `TimeOnly` 적은 데이터를 직렬화합니다.

`TimeOnly`대한 자세한 내용은 DateOnly 및 TimeOnly 구조체사용하는 방법을 참조하세요.

> **ⓘ 중요**
>
> `TimeOnly` .NET Framework에 사용할 수 없습니다.

TimeZoneInfo 클래스

TimeZoneInfo 클래스는 지구의 표준 시간대를 나타내며, 한 표준 시간대의 날짜와 시간을 다른 표준 시간대의 해당 날짜와 시간으로 변환할 수 있습니다. TimeZoneInfo 클래스를 사용하면 날짜 및 시간 값을 사용하여 단일 시점을 명확하게 식별할 수 있습니다. TimeZoneInfo 클래스도 확장할 수 있습니다. Windows 시스템에 대해 제공되고 레지스트리에 정의된 표준 시간대 정보에 따라 달라지지만 사용자 지정 표준 시간대 만들기를 지원합니다. 표준 시간대 정보의 직렬화 및 역직렬화도 지원합니다.

경우에 따라 TimeZoneInfo 클래스를 최대한 활용하려면 추가 개발 작업이 필요할 수 있습니다. 날짜 및 시간 값이 속한 표준 시간대와 긴밀하게 결합되지 않은 경우 추가 작업이 필요합니다. 애플리케이션에서 날짜 및 시간을 연결된 표준 시간대와 연결하는 메커니즘을 제공하지 않는 한 특정 날짜 및 시간 값이 해당 표준 시간대에서 분리되기 쉽습니다. 이 정보를 연결하는 한 가지 방법은 날짜 및 시간 값과 관련 표준 시간대 개체를 모두 포함하는 클래스 또는 구조를 정의하는 것입니다.

.NET에서 표준 시간대 지원을 활용하려면 해당 날짜 및 시간 개체가 인스턴스화될 때 날짜 및 시간 값이 속하는 표준 시간대를 알고 있어야 합니다. 표준 시간대는 특히 웹 또는 네트워크 앱에서 잘 알려지지 않는 경우가 많습니다.

참고

- 날짜, 시간 및 표준 시간대

달력 작업

아티클 • 2025. 03. 25.

날짜 및 시간 값은 특정 시간을 나타내지만 문자열 표현은 문화권을 구분하며 특정 문화권별 날짜 및 시간 값을 표시하는 데 사용되는 규칙과 해당 문화권에서 사용하는 달력에 따라 달라집니다. 이 항목에서는 .NET의 일정에 대한 지원을 살펴보고 날짜 값으로 작업할 때 일정 클래스를 사용하는 것에 대해 설명합니다.

.NET의 일정

.NET의 모든 달력은 기본 달력 구현을 제공하는 System.Globalization.Calendar 클래스에서 파생됩니다. Calendar 클래스에서 상속되는 클래스 중 하나는 모든 lunisolar 달력의 기본 클래스인 EastAsianLunisolarCalendar 클래스입니다. .NET에는 다음과 같은 일정 구현이 포함됩니다.

- ChineseLunisolarCalendar- 중국어 음력 달력을 나타냅니다.
- GregorianCalendar- 그레고리오력을 나타냅니다. 이 달력은 System.Globalization.GregorianCalendarTypes 열거형으로 정의된 하위 형식(예: 아랍어 및 중동 프랑스어)으로 나뉩니다. GregorianCalendar.CalendarType 속성은 그레고리오력의 하위 형식을 지정합니다.
- 히브리어 달력을 나타내는 HebrewCalendar.
- HijriCalendar- Hijri 달력을 나타냅니다.
- 일본어 달력을 나타내는 JapaneseCalendar.
- JapaneseLunisolarCalendar- 일본어 음력 달력을 나타냅니다.
- JulianCalendar- 줄리안 달력을 나타냅니다.
- KoreanCalendar- 한국어 달력을 나타냅니다.
- KoreanLunisolarCalendar- 한국어 루니솔라 달력을 나타냅니다.
- PersianCalendar- 페르시아 달력을 나타냅니다.
- TaiwanCalendar- 대만 달력을 나타냅니다.
- TaiwanLunisolarCalendar- 대만 루니솔라 달력을 나타냅니다.
- ThaiBuddhistCalendar태국 불교 달력을 나타냅니다.

- UmAlQuraCalendar- Um Al Qura 달력을 나타냅니다.

달력은 다음 두 가지 방법 중 하나로 사용할 수 있습니다.

- 특정 문화권에서 사용하는 달력입니다. 각 CultureInfo 개체에는 개체가 현재 사용하고 있는 달력인 현재 달력이 있습니다. 모든 날짜 및 시간 값의 문자열 표현은 현재 문화권과 현재 달력을 자동으로 반영합니다. 일반적으로 현재 달력은 문화권의 기본 달력입니다. CultureInfo 개체에는 선택적으로 문화권에서 사용할 수 있는 추가 일정이 포함된 달력이 있습니다.

- 특정 문화권과 독립적인 독립 실행형 달력입니다. 이 경우 Calendar 메서드를 사용하여 날짜를 달력을 반영하는 값으로 표현합니다.

ChineseLunisolarCalendar, JapaneseLunisolarCalendar, JulianCalendar, KoreanLunisolarCalendar, PersianCalendar및 TaiwanLunisolarCalendar 6개의 일정 클래스는 독립 실행형 달력으로만 사용할 수 있습니다. 문화권에서는 기본 달력이나 선택적 달력으로 사용되지 않습니다.

캘린더와 문화

각 문화권에는 CultureInfo.Calendar 속성으로 정의된 기본 달력이 있습니다. CultureInfo.OptionalCalendars 속성은 해당 문화권의 기본 달력을 포함하여 특정 문화권에서 지원하는 모든 달력을 지정하는 Calendar 개체의 배열을 반환합니다.

다음 예제에서는 CultureInfo.Calendar 및 CultureInfo.OptionalCalendars 속성을 보여 줍니다. 태국어(태국) 및 일본어(일본) 문화권에 대한 `CultureInfo` 개체를 만들고 기본 및 선택적 달력을 표시합니다. 두 경우 모두 문화권의 기본 달력도 CultureInfo.OptionalCalendars 컬렉션에 포함됩니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      // Create a CultureInfo for Thai in Thailand.
      CultureInfo th = CultureInfo.CreateSpecificCulture("th-TH");
      DisplayCalendars(th);

      // Create a CultureInfo for Japanese in Japan.
      CultureInfo ja = CultureInfo.CreateSpecificCulture("ja-JP");
      DisplayCalendars(ja);
   }
```

```
   static void DisplayCalendars(CultureInfo ci)
   {
      Console.WriteLine($"Calendars for the {ci.Name} culture:");

      // Get the culture's default calendar.
      Calendar defaultCalendar = ci.Calendar;
      Console.Write("   Default Calendar: {0}",
GetCalendarName(defaultCalendar));

      if (defaultCalendar is GregorianCalendar)
         Console.WriteLine(" ({0})",
                           ((GregorianCalendar)
defaultCalendar).CalendarType);
      else
         Console.WriteLine();

      // Get the culture's optional calendars.
      Console.WriteLine("   Optional Calendars:");
      foreach (var optionalCalendar in ci.OptionalCalendars) {
         Console.Write("{0,6}{1}", "", GetCalendarName(optionalCalendar));
         if (optionalCalendar is GregorianCalendar)
            Console.Write(" ({0})",
                          ((GregorianCalendar)
optionalCalendar).CalendarType);

         Console.WriteLine();
      }
      Console.WriteLine();
   }

   static string GetCalendarName(Calendar cal)
   {
      return cal.ToString().Replace("System.Globalization.", "");
   }
}
// The example displays the following output:
//       Calendars for the th-TH culture:
//          Default Calendar: ThaiBuddhistCalendar
//          Optional Calendars:
//             ThaiBuddhistCalendar
//             GregorianCalendar (Localized)
//
//       Calendars for the ja-JP culture:
//          Default Calendar: GregorianCalendar (Localized)
//          Optional Calendars:
//             GregorianCalendar (Localized)
//             JapaneseCalendar
//             GregorianCalendar (USEnglish)
```

현재 특정 CultureInfo 개체에서 사용 중인 달력은 문화권의 DateTimeFormatInfo.Calendar 속성에 의해 정의됩니다. 문화권의 DateTimeFormatInfo 개체는 CultureInfo.DateTimeFormat 속성에 의해 반환됩니다. 문화권이 만들어지면 기본값은 CultureInfo.Calendar 속성의 값과 동일합니다. 그러나 문화권의 현재 달력을

CultureInfo.OptionalCalendars 속성에서 반환된 배열에 포함된 일정으로 변경할 수 있습니다. 현재 달력을 CultureInfo.OptionalCalendars 속성 값에 포함되지 않은 달력으로 설정하려고 하면 ArgumentException이 발생됩니다.

다음 예제에서는 아랍어(사우디아라비아) 문화권에서 사용하는 달력을 변경합니다. 먼저 DateTime 값을 인스턴스화하고 현재 문화권(이 경우 영어(미국))과 현재 문화권의 달력(이 경우 그레고리력)을 사용하여 표시합니다. 다음으로 현재 문화권을 아랍어(사우디아라비아)로 변경하고 기본 Um Al-Qura 달력을 사용하여 날짜를 표시합니다. 그런 다음 `CalendarExists` 메서드를 호출하여 히즈리 달력이 아랍어(사우디아라비아) 문화권에서 지원되는지 여부를 확인합니다. 일정이 지원되므로 현재 달력을 Hijri로 변경하고 날짜를 다시 표시합니다. 각 경우에 날짜는 현재 문화권의 현재 달력을 사용하여 표시됩니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      DateTime date1 = new DateTime(2011, 6, 20);

      DisplayCurrentInfo();
      // Display the date using the current culture and calendar.
      Console.WriteLine(date1.ToString("d"));
      Console.WriteLine();

      CultureInfo arSA = CultureInfo.CreateSpecificCulture("ar-SA");

      // Change the current culture to Arabic (Saudi Arabia).
      Thread.CurrentThread.CurrentCulture = arSA;
      // Display date and information about the current culture.
      DisplayCurrentInfo();
      Console.WriteLine(date1.ToString("d"));
      Console.WriteLine();

      // Change the calendar to Hijri.
      Calendar hijri = new HijriCalendar();
      if (CalendarExists(arSA, hijri)) {
         arSA.DateTimeFormat.Calendar = hijri;
         // Display date and information about the current culture.
         DisplayCurrentInfo();
         Console.WriteLine(date1.ToString("d"));
      }
   }

   private static void DisplayCurrentInfo()
   {
      Console.WriteLine($"Current Culture:
```

```
{CultureInfo.CurrentCulture.Name}");
      Console.WriteLine($"Current Calendar:
{DateTimeFormatInfo.CurrentInfo.Calendar}");
   }

   private static bool CalendarExists(CultureInfo culture, Calendar cal)
   {
      foreach (Calendar optionalCalendar in culture.OptionalCalendars)
         if (cal.ToString().Equals(optionalCalendar.ToString()))
            return true;

      return false;
   }
}
// The example displays the following output:
//    Current Culture: en-US
//    Current Calendar: System.Globalization.GregorianCalendar
//    6/20/2011
//
//    Current Culture: ar-SA
//    Current Calendar: System.Globalization.UmAlQuraCalendar
//    18/07/32
//
//    Current Culture: ar-SA
//    Current Calendar: System.Globalization.HijriCalendar
//    19/07/32
```

날짜 및 일정

Calendar 형식의 매개 변수를 포함하고 날짜의 요소(즉, 월, 일 및 연도)가 지정된 달력의 값을 반영하도록 허용하는 생성자를 제외하고 DateTime 값과 DateTimeOffset 값은 항상 양력에 기반합니다. 예를 들어 DateTime.Year 속성은 그레고리오력에서 연도를 반환하고 DateTime.Day 속성은 그레고리오력에서 해당 월의 일을 반환합니다.

> **ⓘ 중요**
>
> 날짜 값과 해당 문자열 표현 사이에는 차이가 있음을 기억해야 합니다. 전자는 그레고리오력을 기반으로 합니다. 후자는 특정 문화권의 현재 달력을 기반으로 합니다.

다음 예제에서는 DateTime 속성과 해당 Calendar 메서드 간의 이러한 차이점을 보여 줍니다. 이 예제에서 현재 문화권은 아랍어(이집트)이고 현재 달력은 Um Al Qura입니다. DateTime 값은 2011년 7월 15일로 설정됩니다. 이러한 동일한 값은 고정 문화권의 규칙을 사용할 때 DateTime.ToString(String, IFormatProvider) 메서드에서 반환되기 때문에 이는 그레고리오 날짜로 해석됩니다. 현재 문화권의 규칙을 사용하여 형식이 지정된 날짜의 문자열 표현은 Um Al Qura 달력의 동일한 날짜인 14/08/32입니다. 다음으로, `DateTime` 및 `Calendar` 멤버는 DateTime 값의 일, 월 및 연도를 반환하는 데 사용됩니다. 각 경우에

서 DateTime 멤버가 반환하는 값은 양력의 값을 반영하는 반면, UmAlQuraCalendar 멤버가 반환하는 값은 Uum al-Qura 달력의 값을 반영합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Make Arabic (Egypt) the current culture
      // and Umm al-Qura calendar the current calendar.
      CultureInfo arEG = CultureInfo.CreateSpecificCulture("ar-EG");
      Calendar cal = new UmAlQuraCalendar();
      arEG.DateTimeFormat.Calendar = cal;
      Thread.CurrentThread.CurrentCulture = arEG;

      // Display information on current culture and calendar.
      DisplayCurrentInfo();

      // Instantiate a date object.
      DateTime date1 = new DateTime(2011, 7, 15);

      // Display the string representation of the date.
      Console.WriteLine($"Date: {date1:d}");
      Console.WriteLine("Date in the Invariant Culture: {0}",
                        date1.ToString("d", CultureInfo.InvariantCulture));
      Console.WriteLine();

      // Compare DateTime properties and Calendar methods.
      Console.WriteLine($"DateTime.Month property: {date1.Month}");
      Console.WriteLine("UmAlQura.GetMonth: {0}",
                        cal.GetMonth(date1));
      Console.WriteLine();

      Console.WriteLine($"DateTime.Day property: {date1.Day}");
      Console.WriteLine("UmAlQura.GetDayOfMonth: {0}",
                        cal.GetDayOfMonth(date1));
      Console.WriteLine();

      Console.WriteLine($"DateTime.Year property: {date1.Year:D4}");
      Console.WriteLine("UmAlQura.GetYear: {0}",
                        cal.GetYear(date1));
      Console.WriteLine();
   }

   private static void DisplayCurrentInfo()
   {
      Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.Name}");
      Console.WriteLine($"Current Calendar:
```

```
{DateTimeFormatInfo.CurrentInfo.Calendar}");
   }
}
// The example displays the following output:
//    Current Culture: ar-EG
//    Current Calendar: System.Globalization.UmAlQuraCalendar
//    Date: 14/08/32
//    Date in the Invariant Culture: 07/15/2011
//
//    DateTime.Month property: 7
//    UmAlQura.GetMonth: 8
//
//    DateTime.Day property: 15
//    UmAlQura.GetDayOfMonth: 14
//
//    DateTime.Year property: 2011
//    UmAlQura.GetYear: 1432
```

달력을 기반으로 날짜 인스턴스화

DateTime 및 DateTimeOffset 값은 그레고리오력을 기반으로 하기 때문에 다른 달력의 날짜, 월 또는 연도 값을 사용하려면 Calendar 형식의 매개 변수를 포함하는 오버로드된 생성자를 호출하여 날짜 값을 인스턴스화해야 합니다. 특정 일정의 Calendar.ToDateTime 메서드 오버로드 중 하나를 호출하여 특정 달력의 값에 따라 DateTime 개체를 인스턴스화할 수도 있습니다.

다음 예제에서는 HebrewCalendar 개체를 DateTime 생성자에 전달하여 하나의 DateTime 값을 인스턴스화하고 HebrewCalendar.ToDateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Int32) 메서드를 호출하여 두 번째 DateTime 값을 인스턴스화합니다. 두 값은 히브리어 달력에서 동일한 값으로 만들어지므로 DateTime.Equals 메서드를 호출하면 두 DateTime 값이 같음이 표시됩니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      HebrewCalendar hc = new HebrewCalendar();

      DateTime date1 = new DateTime(5771, 6, 1, hc);
      DateTime date2 = hc.ToDateTime(5771, 6, 1, 0, 0, 0, 0);

      Console.WriteLine("{0:d} (Gregorian) = {1:d2}/{2:d2}/{3:d4} ({4}):
{5}",
                        date1,
```

```
                        hc.GetMonth(date2),
                        hc.GetDayOfMonth(date2),
                        hc.GetYear(date2),
                        GetCalendarName(hc),
                        date1.Equals(date2));
   }

   private static string GetCalendarName(Calendar cal)
   {
      return cal.ToString().Replace("System.Globalization.", "").
                            Replace("Calendar", "");
   }
}
// The example displays the following output:
//    2/5/2011 (Gregorian) = 06/01/5771 (Hebrew): True
```

현재 달력의 날짜 표시

날짜 및 시간 서식 지정 메서드는 날짜를 문자열로 변환할 때 항상 현재 달력을 사용합니다. 즉, 연도, 월 및 월의 날짜의 문자열 표현은 현재 달력을 반영하며 반드시 그레고리오력을 반영하지는 않습니다.

다음 예제에서는 현재 달력이 날짜의 문자열 표현에 미치는 영향을 보여 줍니다. 현재 문화권을 중국어(번체, 대만)로 변경하고 날짜 값을 인스턴스화합니다. 그런 다음 현재 달력과 날짜를 표시하고, 현재 달력을 TaiwanCalendar변경하고, 현재 달력과 날짜를 다시 한 번 표시합니다. 날짜가 처음 표시되면 그레고리오력에서 날짜로 표시됩니다. 두 번째로 표시되면 대만 달력에 날짜로 표시됩니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Change the current culture to zh-TW.
      CultureInfo zhTW = CultureInfo.CreateSpecificCulture("zh-TW");
      Thread.CurrentThread.CurrentCulture = zhTW;
      // Define a date.
      DateTime date1 = new DateTime(2011, 1, 16);

      // Display the date using the default (Gregorian) calendar.
      Console.WriteLine($"Current calendar:
{zhTW.DateTimeFormat.Calendar}");
      Console.WriteLine(date1.ToString("d"));

      // Change the current calendar and display the date.
```

```
      zhTW.DateTimeFormat.Calendar = new TaiwanCalendar();
      Console.WriteLine($"Current calendar:
{zhTW.DateTimeFormat.Calendar}");
      Console.WriteLine(date1.ToString("d"));
   }
}
// The example displays the following output:
//    Current calendar: System.Globalization.GregorianCalendar
//    2011/1/16
//    Current calendar: System.Globalization.TaiwanCalendar
//    100/1/16
```

현재가 아닌 달력의 날짜 표시

특정 문화권의 현재 달력이 아닌 달력을 사용하여 날짜를 나타내려면 해당 Calendar 개체의 메서드를 호출해야 합니다. 예를 들어 Calendar.GetYear, Calendar.GetMonth및 Calendar.GetDayOfMonth 메서드는 연도, 월 및 일을 특정 달력을 반영하는 값으로 변환합니다.

> **⚠ 경고**
>
> 일부 일정은 문화권의 선택적 달력이 아니므로 이러한 일정의 날짜를 나타내려면 항상 일정 메서드를 호출해야 합니다. 이는 EastAsianLunisolarCalendar, JulianCalendar및 PersianCalendar 클래스에서 파생되는 모든 일정에 해당합니다.

다음 예제에서는 JulianCalendar 개체를 사용하여 Julian 달력에서 1905년 1월 9일 날짜를 인스턴스화합니다. 이 날짜는 기본(그레고리오력) 달력을 사용하여 표시되면 1905년 1월 22일로 표시됩니다. 개별 JulianCalendar 메서드를 호출하면 날짜가 Julian 달력에 표시될 수 있습니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      JulianCalendar julian = new JulianCalendar();
      DateTime date1 = new DateTime(1905, 1, 9, julian);

      Console.WriteLine("Date ({0}): {1:d}",
                        CultureInfo.CurrentCulture.Calendar,
                        date1);
      Console.WriteLine("Date in Julian calendar: {0:d2}/{1:d2}/{2:d4}",
                        julian.GetMonth(date1),
```

```
                        julian.GetDayOfMonth(date1),
                        julian.GetYear(date1));
   }
}
// The example displays the following output:
//    Date (System.Globalization.GregorianCalendar): 1/22/1905
//    Date in Julian calendar: 01/09/1905
```

일정 및 날짜 범위

일정에서 지원하는 가장 빠른 날짜는 해당 달력의 Calendar.MinSupportedDateTime 속성으로 표시됩니다. GregorianCalendar 클래스의 경우 해당 날짜는 0001년 1월 1일 C.E. .NET의 다른 일정 대부분은 이후 날짜를 지원합니다. 일정의 지원되는 가장 빠른 날짜 앞에 있는 날짜 및 시간 값을 사용하려고 하면 ArgumentOutOfRangeException 예외가 발생합니다.

그러나 한 가지 중요한 예외가 있습니다. DateTime 개체와 DateTimeOffset 개체의 기본값(초기화되지 않은) 값은 GregorianCalendar.MinSupportedDateTime 값과 같습니다. 0001년 1월 1일 C.E.를 지원하지 않는 달력에서 이 날짜의 서식을 지정하려고 하고 서식 지정자를 제공하지 않는 경우 서식 지정 메서드는 "G"(일반 날짜/시간 패턴) 형식 지정자 대신 "s"(정렬 가능한 날짜/시간 패턴) 형식 지정자를 사용합니다. 따라서 서식 지정 작업은 ArgumentOutOfRangeException 예외를 throw하지 않습니다. 대신 지원되지 않는 날짜를 반환합니다. 다음은 현재 문화권이 일본 달력을 사용하여 일본어(일본)로 설정된 경우 DateTime.MinValue 값을 표시하고, Um Al Qura 달력을 사용하여 아랍어(이집트)로 표시하는 다음 예제에 설명되어 있습니다. 또한 현재 문화권을 영어(미국)로 설정하고 이러한 각 CultureInfo 개체를 사용하여 DateTime.ToString(IFormatProvider) 메서드를 호출합니다. 각 경우에 정렬 가능한 날짜/시간 패턴을 사용하여 날짜가 표시됩니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      DateTime dat = DateTime.MinValue;

      // Change the current culture to ja-JP with the Japanese Calendar.
      CultureInfo jaJP = CultureInfo.CreateSpecificCulture("ja-JP");
      jaJP.DateTimeFormat.Calendar = new JapaneseCalendar();
      Thread.CurrentThread.CurrentCulture = jaJP;
      Console.WriteLine("Earliest supported date by {1} calendar: {0:d}",
                        jaJP.DateTimeFormat.Calendar.MinSupportedDateTime,
                        GetCalendarName(jaJP));
```

```
      // Attempt to display the date.
      Console.WriteLine(dat.ToString());
      Console.WriteLine();

      // Change the current culture to ar-EG with the Um Al Qura calendar.
      CultureInfo arEG = CultureInfo.CreateSpecificCulture("ar-EG");
      arEG.DateTimeFormat.Calendar = new UmAlQuraCalendar();
      Thread.CurrentThread.CurrentCulture = arEG;
      Console.WriteLine("Earliest supported date by {1} calendar: {0:d}",
                        arEG.DateTimeFormat.Calendar.MinSupportedDateTime,
                        GetCalendarName(arEG));
      // Attempt to display the date.
      Console.WriteLine(dat.ToString());
      Console.WriteLine();

      // Change the current culture to en-US.
      Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
      Console.WriteLine(dat.ToString(jaJP));
      Console.WriteLine(dat.ToString(arEG));
      Console.WriteLine(dat.ToString("d"));
   }

   private static string GetCalendarName(CultureInfo culture)
   {
      Calendar cal = culture.DateTimeFormat.Calendar;
      return cal.GetType().Name.Replace("System.Globalization.",
"").Replace("Calendar", "");
   }
}
// The example displays the following output:
//       Earliest supported date by Japanese calendar: 明治 1/9/8
//       0001-01-01T00:00:00
//
//       Earliest supported date by UmAlQura calendar: 01/01/18
//       0001-01-01T00:00:00
//
//       0001-01-01T00:00:00
//       0001-01-01T00:00:00
//       1/1/0001
```

연대 작업

달력은 일반적으로 날짜를 연대로 나눕니다. 그러나 .NET의 Calendar 클래스는 달력에 정의된 모든 연대를 지원하지 않으며 대부분의 Calendar 클래스는 단일 연대만 지원합니다. JapaneseCalendar 및 JapaneseLunisolarCalendar 클래스만 여러 연대를 지원합니다.

> ⓘ **중요**

JapaneseCalendar JapaneseLunisolarCalendar새로운 시대인 레이와 시대는 2019년 5월 1일에 시작됩니다. 이 변경 내용은 이러한 달력을 사용하는 모든 애플리케이션에 영향을 줍니다. 자세한 내용은 다음 아티클을 참조하세요.

- .NET 일본 달력에서 새 시대를 처리합니다. 이 기능은 여러 연대가 있는 달력을 지원하기 위해 .NET에 추가된 기능을 문서화하고 다중 시대 달력을 처리할 때 사용할 모범 사례를 설명합니다.
- 일본의 시대 변화에 대비하여 애플리케이션을 준비합니다.Windows에서 애플리케이션을 테스트하여 시대 변화에 대비할 수 있도록 정보를 제공합니다.
- 새로운 일본어 달력 시대와 관련된 개별 Windows 버전에 대한 .NET Framework 업데이트를 나열하는 .NET Framework 대한 새로운 일본어 시대 업데이트 요약, 다중 시대 지원을 위한 새로운 .NET Framework 기능에 대해 설명하며 애플리케이션 테스트 시 찾을 항목이 포함되어 있습니다.

대부분의 달력의 시대는 매우 긴 기간을 나타냅니다. 예를 들어 양력에서는 현재 시대가 2천년 이상 지속됩니다. JapaneseCalendar 및 JapaneseLunisolarCalendar여러 연대를 지원하는 두 달력의 경우는 그렇지 않습니다. 시대는 황제의 통치 기간에 해당합니다. 여러 연대에 대한 지원, 특히 현재 시대의 상한을 알 수없는 경우, 특별한 도전을 제기한다.

연대 및 연대 이름

.NET에서 특정 달력 구현에서 지원하는 연대를 나타내는 정수는 Calendar.Eras 배열에 역순으로 저장됩니다. 현재 시대(최신 시간 범위가 있는 시대)는 인덱스 0이며, 여러 연대를 지원하는 Calendar 클래스의 경우 각 연속 인덱스는 이전 시대를 반영합니다. 정적 Calendar.CurrentEra 속성은 Calendar.Eras 배열에서 현재 연대의 인덱스를 정의합니다. 값이 항상 0인 상수입니다. 개별 Calendar 클래스에는 현재 시대의 값을 반환하는 정적 필드도 포함됩니다. 다음 표에 나열되어 있습니다.

테이블 확장

캘린더 클래스	현재 연대 필드
ChineseLunisolarCalendar	ChineseEra
GregorianCalendar	ADEra
HebrewCalendar	HebrewEra
HijriCalendar	HijriEra
JapaneseLunisolarCalendar	JapaneseEra
JulianCalendar	JulianEra

캘린더 클래스	현재 연대 필드
KoreanCalendar	KoreanEra
KoreanLunisolarCalendar	GregorianEra
PersianCalendar	PersianEra
ThaiBuddhistCalendar	ThaiBuddhistEra
UmAlQuraCalendar	UmAlQuraEra

특정 연대 번호에 해당하는 이름은 DateTimeFormatInfo.GetEraName 또는 DateTimeFormatInfo.GetAbbreviatedEraName 메서드에 연대 번호를 전달하여 검색할 수 있습니다. 다음 예제에서는 이러한 메서드를 호출하여 GregorianCalendar 클래스의 연대 지원에 대한 정보를 검색합니다. 현재 시대 2년차의 1월 1일에 해당하는 양력 달력 날짜와 지원되는 각 일본 달력 시대의 2년차 1월 1일에 해당하는 양력 날짜가 표시됩니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      int year = 2;
      int month = 1;
      int day = 1;
      Calendar cal = new JapaneseCalendar();

      Console.WriteLine("\nDate instantiated without an era:");
      DateTime date1 = new DateTime(year, month, day, 0, 0, 0, 0, cal);
      Console.WriteLine("{0}/{1}/{2} in Japanese Calendar -> {3:d} in
Gregorian",
                        cal.GetMonth(date1), cal.GetDayOfMonth(date1),
                        cal.GetYear(date1), date1);

      Console.WriteLine("\nDates instantiated with eras:");
      foreach (int era in cal.Eras) {
         DateTime date2 = cal.ToDateTime(year, month, day, 0, 0, 0, 0, era);
         Console.WriteLine("{0}/{1}/{2} era {3} in Japanese Calendar ->
{4:d} in Gregorian",
                           cal.GetMonth(date2), cal.GetDayOfMonth(date2),
                           cal.GetYear(date2), cal.GetEra(date2), date2);
      }
   }
}
```

또한 "g" 사용자 지정 날짜 및 시간 형식 문자열에는 날짜 및 시간의 문자열 표현에 달력의 연대 이름이 포함됩니다. 자세한 내용은 사용자 지정 날짜 및 시간 형식 문자열을 참조하세요.

연대를 사용하여 날짜 인스턴스화

여러 연대를 지원하는 두 Calendar 클래스의 경우 특정 연도, 월 및 일로 구성된 날짜가 모호할 수 있습니다. 예를 들어, JapaneseCalendar이 지원하는 모든 시대에는 연도가 '1'인 해가 있습니다. 일반적으로 연대를 지정하지 않으면 날짜와 시간 및 달력 메서드 모두 값이 현재 연대에 속한다고 가정합니다. 이는 Calendar형식의 매개 변수를 포함하는 DateTime 및 DateTimeOffset 생성자뿐만 아니라 JapaneseCalendar.ToDateTime 및 JapaneseLunisolarCalendar.ToDateTime 메서드도 마찬가지입니다. 다음 예제에서는 지정되지 않은 연대의 2년 1월 1일을 나타내는 날짜를 인스턴스화합니다. 레이와 시대가 현재 시대인 경우 예제를 실행하면 날짜는 레이와 시대의 2년차로 해석됩니다. 令 Era는 DateTime.ToString(String, IFormatProvider) 메서드에서 반환된 문자열의 연도보다 앞이며 그레고리오력에서 2020년 1월 1일에 해당합니다. (레이와 시대는 그레고리오력의 2019년에 시작됩니다.)

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        var date = new DateTime(2, 1, 1, japaneseCal);
        Console.WriteLine($"Gregorian calendar date: {date:d}");
        Console.WriteLine($"Japanese calendar date: {date.ToString("d",
jaJp)}");
    }
}
```

그러나 시대가 바뀌면 이 코드의 의도가 모호해집니다. 날짜는 현재 시대의 2 년을 나타내기위한 것입니까, 아니면 헤이세이 시대의 2 년을 대표하기위한 것입니까? 이러한 모호성을 방지하는 방법에는 두 가지가 있습니다.

- 기본 GregorianCalendar 클래스를 사용하여 날짜 및 시간 값을 인스턴스화합니다. 그런 다음, 다음 예제와 같이 일본어 달력 또는 일본어 Lunisolar 달력을 사용하여 날짜의 문자열 표현을 사용할 수 있습니다.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        var date = new DateTime(1905, 2, 12);
        Console.WriteLine($"Gregorian calendar date: {date:d}");

        // Call the ToString(IFormatProvider) method.
        Console.WriteLine($"Japanese calendar date: {date.ToString("d",
jaJp)}");

        // Use a FormattableString object.
        FormattableString fmt = $"{date:d}";
        Console.WriteLine($"Japanese calendar date:
{fmt.ToString(jaJp)}");

        // Use the JapaneseCalendar object.
        Console.WriteLine($"Japanese calendar date:
{jaJp.DateTimeFormat.GetEraName(japaneseCal.GetEra(date))}" +
                          $"
{japaneseCal.GetYear(date)}/{japaneseCal.GetMonth(date)}/{japaneseCal.G
etDayOfMonth(date)}");

        // Use the current culture.
        CultureInfo.CurrentCulture = jaJp;
        Console.WriteLine($"Japanese calendar date: {date:d}");
    }
}
// The example displays the following output:
//    Gregorian calendar date: 2/12/1905
//    Japanese calendar date: 明治38/2/12
//    Japanese calendar date: 明治38/2/12
//    Japanese calendar date: 明治38/2/12
//    Japanese calendar date: 明治38/2/12
```

- 연대를 명시적으로 지정하는 날짜 및 시간 메서드를 호출합니다. 여기에는 다음 메서드가 포함됩니다.
  - JapaneseCalendar 또는 JapaneseLunisolarCalendar 클래스의 ToDateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Int32) 메서드입니다.
  - 현재 문화권이 Japanese-Japan("ja-JP")이며 그 문화권의 달력이 JapaneseCalendar인 경우, 선택적으로 DateTimeStyles 인수가 포함되는 문자열

을 구문 분석하기 위해 Parse, TryParse, ParseExact또는 TryParseExact와 같은 DateTime 또는 DateTimeOffset 구문 분석 메서드입니다. 구문 분석할 문자열에는 연대가 포함되어야 합니다.

- `provider` 매개 변수를 IFormatProvider형식으로 포함하는 DateTime 또는 DateTimeOffset 구문 분석 메서드입니다. `provider`은 JapaneseCalendar인 현재 달력을 가진 Japanese-Japan("ja-JP") 문화권을 나타내는 CultureInfo 개체이거나, Calendar 속성이 JapaneseCalendar인 DateTimeFormatInfo 개체여야 합니다. 구문 분석할 문자열에는 연대가 포함되어야 합니다.

다음 예제에서는 이러한 세 가지 방법을 사용하여 1868년 9월 8일에 시작되어 1912년 7월 29일에 종료된 메이지 시대의 날짜와 시간을 인스턴스화합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        // We can get the era index by calling
DateTimeFormatInfo.GetEraName.
        int eraIndex = 0;

        for (int ctr = 0; ctr <
jaJp.DateTimeFormat.Calendar.Eras.Length; ctr++)
           if (jaJp.DateTimeFormat.GetEraName(ctr) == "明治")
              eraIndex = ctr;
        var date1 = japaneseCal.ToDateTime(23, 9, 8, 0, 0, 0, 0,
eraIndex);
        Console.WriteLine($"{date1.ToString("d", jaJp)} (Gregorian
{date1:d})");

        try {
            var date2 = DateTime.Parse("明治23/9/8", jaJp);
            Console.WriteLine($"{date2.ToString("d", jaJp)} (Gregorian
{date2:d})");
        }
        catch (FormatException)
        {
            Console.WriteLine("The parsing operation failed.");
        }

        try {
            var date3 = DateTime.ParseExact("明治23/9/8", "gyy/M/d",
```

```
jaJp);
            Console.WriteLine($"{date3.ToString("d", jaJp)} (Gregorian
{date3:d})");
        }
        catch (FormatException)
        {
            Console.WriteLine("The parsing operation failed.");
        }
    }
}
// The example displays the following output:
//    明治23/9/8 (Gregorian 9/8/1890)
//    明治23/9/8 (Gregorian 9/8/1890)
//    明治23/9/8 (Gregorian 9/8/1890)
```

> **팁**
>
> 여러 연대를 지원하는 달력을 사용하는 경우 *항상 그레고리오 날짜를 사용하여 날짜를 인스턴스화하거나 해당 달력에 따라 날짜와 시간을 인스턴스화할 때 연대를 지정할* 있습니다.

ToDateTime(Int32, Int32, Int32, Int32, Int32, Int32, Int32) 메서드에 연대를 지정할 때 달력의 Eras 속성에 연대의 인덱스도 제공합니다. 그러나 연대가 변경될 수 있는 달력의 경우 이러한 인덱스는 상수 값이 아닙니다. 현재 시대는 인덱스 0이고 가장 오래된 연대는 인덱스 `Eras.Length - 1`. 달력에 새 연대가 추가되면 이전 연대의 인덱스가 하나씩 증가합니다. 다음과 같이 적절한 연대 인덱스를 제공할 수 있습니다.

- 현재 시대의 날짜의 경우 항상 달력의 CurrentEra 속성을 사용합니다.
- 지정된 연대의 날짜에 대해 DateTimeFormatInfo.GetEraName 메서드를 사용하여 지정된 연대 이름에 해당하는 인덱스를 검색합니다. JapaneseCalendar는 ja-JP 문화권을 나타내는 CultureInfo 개체의 현재 달력이어야 합니다. (이 기술은 JapaneseCalendar동일한 연대를 지원하므로 JapaneseLunisolarCalendar에도 작동합니다. 이전 예제에서는 이 방법을 보여 줍니다.

달력, 연대 및 날짜 범위: 완화된 범위 확인

개별 달력에서 지원되는 날짜 범위와 마찬가지로 JapaneseCalendar 및 JapaneseLunisolarCalendar 클래스의 연대도 지원됩니다. 이전에는 .NET에서 엄격한 연대 범위 검사를 사용하여 연대별 날짜가 해당 연대 범위 내에 있는지 확인했습니다. 즉, 날짜가 지정된 연대 범위를 벗어나면 메서드는 ArgumentOutOfRangeExceptionthrow합니다. 현재 .NET은 기본적으로 완화된 범위 검사를 사용합니다. .NET의 모든 버전에 대한 업데이트에서는 완화된 연대 범위 검사를 도입했습니다. 지정된 연대 범위를 벗어난 연대별 날짜를 다음 연대로 인스턴스화하려는 시도에서는 예외가 발생하지 않습니다.

다음 예제에서는 1926년 12월 25일에 시작되어 1989년 1월 7일에 종료된 쇼와 시대의 65번째 해에 날짜를 인스턴스화하려고 시도합니다. 이 날짜는 1990년 1월 9일에 해당하며, 이 날짜는 JapaneseCalendar쇼와 시대의 범위를 벗어났습니다. 예제의 출력에서 알 수 있듯이 예제에 표시된 날짜는 헤이세이 시대 2년차인 1990년 1월 9일입니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      var jaJp = new CultureInfo("ja-JP");
      var cal = new JapaneseCalendar();
      jaJp.DateTimeFormat.Calendar = cal;
      string showaEra = "昭和";

      var dt = cal.ToDateTime(65, 1, 9, 15, 0, 0, 0, GetEraIndex(showaEra));
      FormattableString fmt = $"{dt:d}";

      Console.WriteLine($"Japanese calendar date: {fmt.ToString(jaJp)}");
      Console.WriteLine($"Gregorian calendar date: {fmt}");

      int GetEraIndex(string eraName)
      {
         foreach (var ctr in cal.Eras)
            if (jaJp.DateTimeFormat.GetEraName(ctr) == eraName)
               return ctr;

         return 0;
      }
   }
}
// The example displays the following output:
//   Japanese calendar date: 平成2/1/9
//   Gregorian calendar date: 1/9/1990
```

완화된 범위 검사가 바람직하지 않은 경우 애플리케이션이 실행 중인 .NET 버전에 따라 여러 가지 방법으로 엄격한 범위 검사를 복원할 수 있습니다.

- .NET Core :*.netcore.runtime.json* 구성 파일에 다음을 추가합니다.

  JSON

  ```
  "runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.EnforceJapaneseEraYearRanges": true
  ```

```
    }
}
```

- .NET Framework 4.6 이상 :*app.config* 파일에서 다음 AppContext 스위치를 설정합니다.

```
XML

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <AppContextSwitchOverrides
value="Switch.System.Globalization.EnforceJapaneseEraYearRanges=true"
/>
  </runtime>
</configuration>
```

- .NET Framework 4.5.2 이전 버전 : 다음 레지스트리 값을 설정합니다.

테이블 확장

	가치
키	**HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext**
항목	Switch.System.Globalization.일본 연호 연도 범위 강제 적용
유형	REG_SZ
값	맞다

엄격한 범위 검사를 사용하도록 설정한 상태에서 이전 예제에서는 ArgumentOutOfRangeException throw하고 다음 출력을 표시합니다.

```
콘솔

Unhandled Exception: System.ArgumentOutOfRangeException: Valid values are
between 1 and 64, inclusive.
Parameter name: year
   at System.Globalization.GregorianCalendarHelper.GetYearOffset(Int32 year,
Int32 era, Boolean throwOnError)
   at System.Globalization.GregorianCalendarHelper.ToDateTime(Int32 year,
Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32
millisecond, Int32 era)
   at Example.Main()
```

여러 연대가 있는 달력의 날짜 표시

Calendar 개체가 연대를 지원하고 CultureInfo 개체의 현재 달력인 경우 해당 연대는 전체 날짜 및 시간, 긴 날짜 및 짧은 날짜 패턴에 대한 날짜 및 시간 값의 문자열 표현에 포함됩니다. 다음은 현재 문화권이 일본(일본)이고 현재 달력이 일본 달력인 경우 이러한 날짜 패턴을 표시하는 예제입니다.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example
{
   public static void Main()
   {
      StreamWriter sw = new StreamWriter(@".\eras.txt");
      DateTime dt = new DateTime(2012, 5, 1);

      CultureInfo culture = CultureInfo.CreateSpecificCulture("ja-JP");
      DateTimeFormatInfo dtfi = culture.DateTimeFormat;
      dtfi.Calendar = new JapaneseCalendar();
      Thread.CurrentThread.CurrentCulture = culture;

      sw.WriteLine("\n{0,-43} {1}", "Full Date and Time Pattern:",
dtfi.FullDateTimePattern);
      sw.WriteLine(dt.ToString("F"));
      sw.WriteLine();

      sw.WriteLine("\n{0,-43} {1}", "Long Date Pattern:",
dtfi.LongDatePattern);
      sw.WriteLine(dt.ToString("D"));

      sw.WriteLine("\n{0,-43} {1}", "Short Date Pattern:",
dtfi.ShortDatePattern);
      sw.WriteLine(dt.ToString("d"));
      sw.Close();
    }
}
// The example writes the following output to a file:
//    Full Date and Time Pattern:                gg y'年'M'月'd'日' H:mm:ss
//    平成 24年5月1日 0:00:00
//
//    Long Date Pattern:                         gg y'年'M'月'd'日'
//    平成 24年5月1日
//
//    Short Date Pattern:                        gg y/M/d
//    平成 24/5/1
```

⚠ 경고

JapaneseCalendar 클래스는 .NET에서 여러 연대의 날짜를 지원하며 CultureInfo 개체의 현재 달력이 될 수 있는 유일한 달력 클래스입니다. 특히, 이는 일본(일본어) 문화권을 나타내는 CultureInfo 개체를 위해 사용됩니다.

모든 달력의 경우 "g" 사용자 지정 형식 지정자에 결과 문자열의 연대가 포함됩니다. 다음 예제에서는 "MM-dd-yyyy g" 사용자 지정 형식 문자열을 사용하여 현재 달력이 그레고리오력인 경우 결과 문자열에 연대를 포함합니다.

C#

```
   DateTime dat = new DateTime(2012, 5, 1);
   Console.WriteLine($"{dat:MM-dd-yyyy g}");
// The example displays the following output:
//     05-01-2012 A.D.
```

날짜의 문자열 표현이 현재 달력이 아닌 달력에서 표현되는 경우 Calendar 클래스에는 날짜와 날짜가 속한 연대를 명확하게 나타내기 위해 Calendar.GetYear, Calendar.GetMonth 및 Calendar.GetDayOfMonth 메서드와 함께 사용할 수 있는 Calendar.GetEra 메서드가 포함됩니다. 다음 예제에서는 JapaneseLunisolarCalendar 클래스를 사용하여 그림을 제공합니다. 그러나 결과 문자열에서 연대의 정수 대신 의미 있는 이름이나 약어를 포함하려면 DateTimeFormatInfo 개체를 인스턴스화하고 JapaneseCalendar을 현재 달력으로 설정해야 합니다. (JapaneseLunisolarCalendar 달력은 문화권의 현재 달력일 수 없지만, 이 경우 두 달력은 동일한 연대를 공유합니다.)

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime date1 = new DateTime(2011, 8, 28);
      Calendar cal = new JapaneseLunisolarCalendar();

      Console.WriteLine("{0} {1:d4}/{2:d2}/{3:d2}",
                        cal.GetEra(date1),
                        cal.GetYear(date1),
                        cal.GetMonth(date1),
                        cal.GetDayOfMonth(date1));

      // Display eras
      CultureInfo culture = CultureInfo.CreateSpecificCulture("ja-JP");
      DateTimeFormatInfo dtfi = culture.DateTimeFormat;
      dtfi.Calendar = new JapaneseCalendar();

      Console.WriteLine("{0} {1:d4}/{2:d2}/{3:d2}",
```

```
                        dtfi.GetAbbreviatedEraName(cal.GetEra(date1)),
                        cal.GetYear(date1),
                        cal.GetMonth(date1),
                        cal.GetDayOfMonth(date1));
   }
}
// The example displays the following output:
//       4 0023/07/29
//       平 0023/07/29
```

일본 달력에서 시대의 첫 해는 Gannen (元年)이라고합니다. 예를 들어 헤이세이 1 대신 헤이세이 시대의 첫 해를 헤이세이 가넨으로 묘사할 수 있습니다. .NET은 JapaneseCalendar 클래스를 사용하여 Japanese-Japan("ja-JP") 문화권을 나타내는 CultureInfo 개체와 함께 사용될 때 다음 표준 또는 사용자 지정 날짜 및 시간 형식 문자열로 서식이 지정된 날짜 및 시간에 대한 서식 지정 작업에서 이 규칙을 채택합니다.

- "D" 표준 날짜 및 시간 형식 문자열로 표시된 긴 날짜 패턴.
- "F" 표준 날짜 및 시간 형식 문자열로 표시된 전체 날짜 긴 시간 패턴.
- "f" 표준 날짜 및 시간 형식 문자열로 표시된 전체 날짜 짧은 시간 패턴.
- "Y" 또는 "y" 표준 날짜 및 시간 형식 문자열로 표시된 연도/월 패턴.
- "ggy'年'" 또는 "ggy年"사용자 지정 날짜 및 시간 형식 문자열.

예를 들어, 다음 예제는 JapaneseCalendar형식으로 헤이세이 시대 첫 해의 날짜를 표시합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
         var enUs = new CultureInfo("en-US");
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;
        string heiseiEra = "平成";

        var date = japaneseCal.ToDateTime(1, 8, 18, 0, 0, 0, 0,
GetEraIndex(heiseiEra));
        FormattableString fmt = $"{date:D}";
        Console.WriteLine($"Japanese calendar date: {fmt.ToString(jaJp)}
(Gregorian: {fmt.ToString(enUs)})");

        int GetEraIndex(string eraName)
        {
           foreach (var ctr in japaneseCal.Eras)
              if (jaJp.DateTimeFormat.GetEraName(ctr) == eraName)
```

```
                return ctr;

            return 0;
        }
    }
}
// The example displays the following output:
//    Japanese calendar date: 平成元年8月18日 (Gregorian: Friday, August 18,
1989)
```

서식 지정 작업에서 이 동작이 바람직하지 않은 경우 .NET 버전에 따라 다음을 수행하여 항상 연대의 첫 해를 "Gannen"이 아닌 "1"로 나타내는 이전 동작을 복원할 수 있습니다.

- .NET Core :*.netcore.runtime.json* 구성 파일에 다음을 추가합니다.

  JSON

  ```
  "runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.FormatJapaneseFirstYearAsANumber":
  true
    }
  }
  ```

- .NET Framework 4.6 이상 :*app.config* 파일에서 다음 AppContext 스위치를 설정합니다.

  XML

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <runtime>
      <AppContextSwitchOverrides
  value="Switch.System.Globalization.FormatJapaneseFirstYearAsANumber=tru
  e" />
    </runtime>
  </configuration>
  ```

- .NET Framework 4.5.2 이전 버전 : 다음 레지스트리 값을 설정합니다.

  테이블 확장

	가치
키	**HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext**
항목	스위치.시스템.세계화.일본첫해를숫자로형식화합니다.

	가치
유형	REG_SZ
값	맞다

서식 지정 작업에서 gannen 지원을 사용하지 않도록 설정하면 이전 예제에서는 다음 출력을 표시합니다.

콘솔

```
Japanese calendar date: 平成1年8月18日 (Gregorian: Friday, August 18, 1989)
```

또한 날짜 및 시간 구문 분석 작업이 "1" 또는 Gannen으로 표시된 연도를 포함하는 문자열을 지원하게 되도록 .NET도 업데이트되었습니다. 이 작업을 수행할 필요는 없지만 이전 동작을 복원하여 "1"만 시대의 첫 해로 인식할 수 있습니다. .NET 버전에 따라 다음과 같이 이 작업을 수행할 수 있습니다.

- .NET Core :*.netcore.runtime.json* 구성 파일에 다음을 추가합니다.

  JSON

  ```
  "runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.EnforceLegacyJapaneseDateParsing":
  true
    }
  }
  ```

- .NET Framework 4.6 이상 :*app.config* 파일에서 다음 AppContext 스위치를 설정합니다.

  XML

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <runtime>
      <AppContextSwitchOverrides
  value="Switch.System.Globalization.EnforceLegacyJapaneseDateParsing=tru
  e" />
    </runtime>
  </configuration>
  ```

- .NET Framework 4.5.2 이전 버전 : 다음 레지스트리 값을 설정합니다.

테이블 확장

	가치
키	**HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext**
항목	Switch.System.Globalization.EnforceLegacyJapaneseDateParsing
유형	REG_SZ
값	맞다

참고하십시오

- 방법: 양력 이외의 달력에 날짜 표시
- Calendar 클래스

DateOnly 및 TimeOnly 구조를 사용하는 방법

DateOnly 및 TimeOnly 구조체는 .NET 6에서 도입되었으며 각각 특정 날짜 또는 시간을 나타냅니다. .NET 6 이전에는 항상 .NET Framework에서 개발자는 DateTime 형식(또는 다른 대안)을 사용하여 다음 중 하나를 나타냅니다.

- 전체 날짜 및 시간입니다.
- 시간을 무시하는 날짜입니다.
- 날짜를 무시하는 시간입니다.

`DateOnly` 및 `TimeOnly` `DateTime` 형식의 특정 부분을 나타내는 형식입니다.

> ⓘ **Important**
>
> DateOnly 및 TimeOnly 형식은 .NET Framework에서 사용할 수 없습니다.

DateOnly 구조체

DateOnly 구조체는 시간 없이 특정 날짜를 나타냅니다. 시간 구성 요소가 없으므로 하루의 시작부터 종료까지의 날짜를 나타냅니다. 이 구조는 생년월일, 기념일 또는 비즈니스 관련 날짜와 같은 특정 날짜를 저장하는 데 적합합니다.

시간 구성 요소를 무시하고 `DateTime`을 사용할 수도 있지만, `DateOnly`보다 `DateTime`을 사용할 때의 몇 가지 이점이 있습니다.

- `DateTime` 구조체가 시간대에 따라 조정된 경우, 이전 또는 다음 날로 넘어갈 수 있습니다. `DateOnly` 표준 시간대로 오프셋할 수 없으며 항상 설정된 날짜를 나타냅니다.
- `DateTime` 구조체 직렬화에는 데이터의 의도를 모호하게 할 수 있는 시간 구성 요소가 포함됩니다. 또한 `DateOnly` 적은 데이터를 직렬화합니다.
- 코드가 SQL Server와 같은 데이터베이스와 상호 작용하는 경우 전체 날짜는 일반적으로 시간을 포함하지 않는 `date` 데이터 형식으로 저장됩니다. `DateOnly` 데이터베이스 형식과 더 잘 일치합니다.

`DateOnly`는 `DateTime`과 마찬가지로 0001-01-01부터 9999-12-31까지의 범위를 가집니다. `DateOnly` 생성자에서 특정 달력을 지정할 수 있습니다. 그러나 `DateOnly` 개체는 생성에 사용된 달력에 관계없이 항상 proleptic Gregorian 달력의 날짜를 나타냅니다. 예를 들어 히브리어 달력에서 날짜를 작성할 수 있지만 날짜는 그레고리오력으로 변환됩니다.

C#

```
var hebrewCalendar = new System.Globalization.HebrewCalendar();
var theDate = new DateOnly(5776, 2, 8, hebrewCalendar); // 8 Cheshvan 5776

Console.WriteLine(theDate);

/* This example produces the following output:
 *
 * 10/21/2015
*/
```

DateOnly 예제

다음 예제를 사용하여 `DateOnly` 대해 알아봅니다.

- DateTime을 DateOnly로 변환하기
- 일, 월, 연도 추가 또는 빼기
- DateOnly 구문 분석 및 서식
- DateOnly 비교

DateTime을 DateOnly로 변환

다음 코드에 설명된 대로 DateOnly.FromDateTime 정적 메서드를 사용하여 `DateOnly` 형식에서 `DateTime` 형식을 만듭니다.

C#

```
var today = DateOnly.FromDateTime(DateTime.Now);
Console.WriteLine($"Today is {today}");

/* This example produces output similar to the following:
 *
 * Today is 12/28/2022
*/
```

일, 월, 연도 추가 또는 빼기

DateOnly 구조를 조정하는 데 사용되는 세 가지 메서드는 AddDays, AddMonths 및 AddYears. 각 메서드는 정수 매개 변수를 사용하고 해당 측정값에 따라 날짜를 늘립니다. 음수가 제공되면 그 값만큼 날짜가 줄어듭니다. 메서드는 구조가 변경할 수 없기 때문에 `DateOnly` 의 새로운 인스턴스를 반환합니다.

C#

```
var theDate = new DateOnly(2015, 10, 21);

var nextDay = theDate.AddDays(1);
var previousDay = theDate.AddDays(-1);
var decadeLater = theDate.AddYears(10);
var lastMonth = theDate.AddMonths(-1);

Console.WriteLine($"Date: {theDate}");
Console.WriteLine($" Next day: {nextDay}");
Console.WriteLine($" Previous day: {previousDay}");
Console.WriteLine($" Decade later: {decadeLater}");
Console.WriteLine($" Last month: {lastMonth}");

/* This example produces the following output:
 *
 * Date: 10/21/2015
 *  Next day: 10/22/2015
 *  Previous day: 10/20/2015
 *  Decade later: 10/21/2025
 *  Last month: 9/21/2015
*/
```

DateOnly 구문 분석 및 서식 지정

DateOnly는 문자열에서 구문 분석할 수 있으며, DateTime 구조도 마찬가지입니다. 모든 표준 .NET 날짜 기반 구문 분석 토큰은 `DateOnly`과 함께 작동합니다. `DateOnly` 형식을 문자열로 변환할 때 표준 .NET 날짜 기반 서식 지정 패턴도 사용할 수 있습니다. 문자열 서식 지정에 대한 자세한 내용은 표준 날짜 및 시간 서식 문자열참조하세요.

C#

```
var theDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture);  // Custom format
var theDate2 = DateOnly.Parse("October 21, 2015", CultureInfo.InvariantCulture);

Console.WriteLine(theDate.ToString("m", CultureInfo.InvariantCulture));     // Month
day pattern
Console.WriteLine(theDate2.ToString("o", CultureInfo.InvariantCulture));    // ISO
8601 format
Console.WriteLine(theDate2.ToLongDateString());

/* This example produces the following output:
 *
 * October 21
 * 2015-10-21
 * Wednesday, October 21, 2015
*/
```

날짜 전용 비교

DateOnly 다른 인스턴스와 비교할 수 있습니다. 예를 들어 날짜가 다른 날짜 이전 또는 이후인지 또는 오늘 날짜가 특정 날짜와 일치하는지 확인할 수 있습니다.

C#

```
var theDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture);  // Custom format
var theDate2 = DateOnly.Parse("October 21, 2015", CultureInfo.InvariantCulture);
var dateLater = theDate.AddMonths(6);
var dateBefore = theDate.AddDays(-10);

Console.WriteLine($"Consider {theDate}...");
Console.WriteLine($" Is '{nameof(theDate2)}' equal? {theDate == theDate2}");
Console.WriteLine($" Is {dateLater} after? {dateLater > theDate} ");
Console.WriteLine($" Is {dateLater} before? {dateLater < theDate} ");
Console.WriteLine($" Is {dateBefore} after? {dateBefore > theDate} ");
Console.WriteLine($" Is {dateBefore} before? {dateBefore < theDate} ");

/* This example produces the following output:
 *
 * Consider 10/21/2015
 *  Is 'theDate2' equal? True
 *  Is 4/21/2016 after? True
 *  Is 4/21/2016 before? False
 *  Is 10/11/2015 after? False
 *  Is 10/11/2015 before? True
*/
```

TimeOnly 구조체

TimeOnly 구조는 매일 알람 시계 또는 매일 점심을 먹는 시간과 같은 하루 중 시간 값을 나타냅니다. `TimeOnly`은 특정 시간인 **00:00:00.0000000 - 23:59:59.9999999**범위로 제한됩니다.

`TimeOnly` 형식이 도입되기 전에 프로그래머는 일반적으로 DateTime 형식 또는 TimeSpan 형식을 사용하여 특정 시간을 나타냅니다. 그러나 이러한 구조를 사용하여 날짜 없이 시간을 시뮬레이션할 때 몇 가지 문제가 발생할 수 있는데 `TimeOnly`가 그 문제를 해결합니다.

- `TimeSpan` 스톱워치로 측정된 시간과 같은 경과된 시간을 나타냅니다. 상한 범위는 29,000년 이상이며 해당 값은 시간이 지나면 뒤로 이동함을 나타내기 위해 음수일 수 있습니다. 음수 `TimeSpan`는 특정한 시간대를 나타내지 않습니다.

- `TimeSpan` 하루 중 시간으로 사용되는 경우 24시간 외의 값으로 조작할 수 있는 위험이 있습니다. `TimeOnly` 이러한 위험이 없습니다. 예를 들어 직원의 근무 교대 근무가 18:00에 시작되고 8시간 동안 지속되는 경우 `TimeOnly` 구조에 8시간을 추가하면 2:00으로 롤오버됩니다.

- 하루의 특정 시간에 `DateTime`를 사용하려면 임의의 날짜를 그 시간과 연결한 다음 나중에 무시해야 합니다. `DateTime.MinValue`(0001-01-01)를 날짜로 선택하는 것이 일반적이지만 `DateTime` 값에서 시간을 빼면 `OutOfRange` 예외가 발생할 수 있습니다. `TimeOnly`는 시간이 24시간 내에서 앞뒤로 움직이기 때문에 이 문제가 발생하지 않습니다.
- `DateTime` 구조체 직렬화에는 데이터의 의도를 모호하게 할 수 있는 날짜 구성 요소가 포함됩니다. 또한 `TimeOnly` 적은 데이터를 직렬화합니다.

TimeOnly 사례

다음 예제를 사용하여 `TimeOnly`대해 알아봅니다.

- DateTime을 TimeOnly로 변환하기
- 시간 더하기 또는 빼기
- TimeOnly 구문 분석 및 서식
- TimeSpan 및 DateTime과 작업하기
- 산술 연산자와 TimeOnly 비교하기

DateTime을 TimeOnly로 변환

다음 코드에 설명된 대로 TimeOnly.FromDateTime 정적 메서드를 사용하여 `TimeOnly` 형식에서 `DateTime` 형식을 만듭니다.

C#

```
var now = TimeOnly.FromDateTime(DateTime.Now);
Console.WriteLine($"It is {now} right now");

/* This example produces output similar to the following:
 *
 * It is 2:01 PM right now
*/
```

시간 추가 또는 빼기

TimeOnly 구조를 조정하는 데 사용되는 세 가지 메서드는 AddHours, AddMinutes및 Add. `AddHours` 및 `AddMinutes` 모두 정수 매개 변수를 사용하고 그에 따라 값을 조정합니다. 음수 값을 사용하여 빼고 양수 값을 사용하여 추가할 수 있습니다. 메서드는 구조가 불변이므로 `TimeOnly`의 새 인스턴스를 반환합니다. `Add` 메서드는 TimeSpan 매개 변수를 사용하고 `TimeOnly` 값에서 값을 추가하거나 뺍니다.

`TimeOnly` 24시간 기간만 나타내므로 이러한 세 가지 메서드에 제공된 값을 추가할 때 적절하게 앞으로 또는 뒤로 롤오버됩니다. 예를 들어 `01:30:00` 값을 사용하여 오전 1시 30분으로 표시한 뒤, 그 시점에 -4 시간을 더하면 오후 9시 30분인 `21:30:00`로 되돌아갑니다. 롤오버된 일 수를 캡처하는 `AddHours`, `AddMinutes`및 `Add` 대한 메서드 오버로드가 있습니다.

C#

```
var theTime = new TimeOnly(7, 23, 11);

var hourLater = theTime.AddHours(1);
var minutesBefore = theTime.AddMinutes(-12);
var secondsAfter = theTime.Add(TimeSpan.FromSeconds(10));
var daysLater = theTime.Add(new TimeSpan(hours: 21, minutes: 200, seconds: 83), out
int wrappedDays);
var daysBehind = theTime.AddHours(-222, out int wrappedDaysFromHours);

Console.WriteLine($"Time: {theTime}");
Console.WriteLine($" Hours later: {hourLater}");
Console.WriteLine($" Minutes before: {minutesBefore}");
Console.WriteLine($" Seconds after: {secondsAfter}");
Console.WriteLine($" {daysLater} is the time, which is {wrappedDays} days later");
Console.WriteLine($" {daysBehind} is the time, which is {wrappedDaysFromHours} days
prior");

/* This example produces the following output:
 *
 * Time: 7:23 AM
 *  Hours later: 8:23 AM
 *  Minutes before: 7:11 AM
 *  Seconds after: 7:23 AM
 *  7:44 AM is the time, which is 1 days later
 *  1:23 AM is the time, which is -9 days prior
*/
```

"'TimeOnly' 구문 분석 및 서식 지정"

TimeOnly는 문자열에서 구문 분석할 수 있으며, DateTime 구조도 마찬가지입니다. 모든 표준 .NET 시간 기반 구문 분석 토큰은 `TimeOnly`와 함께 작동합니다. `TimeOnly` 형식을 문자열로 변환할 때 표준 .NET 날짜 기반 서식 지정 패턴도 사용할 수 있습니다. 문자열 서식 지정에 대한 자세한 내용은 표준 날짜 및 시간 서식 문자열참조하세요.

C#

```
var theTime = TimeOnly.ParseExact("5:00 pm", "h:mm tt",
CultureInfo.InvariantCulture);  // Custom format
var theTime2 = TimeOnly.Parse("17:30:25", CultureInfo.InvariantCulture);

Console.WriteLine(theTime.ToString("o", CultureInfo.InvariantCulture));     //
Round-trip pattern.
```

```
Console.WriteLine(theTime2.ToString("t", CultureInfo.InvariantCulture));    // Long
time format
Console.WriteLine(theTime2.ToLongTimeString());

/* This example produces the following output:
 *
 * 17:00:00.0000000
 * 17:30
 * 5:30:25 PM
*/
```

DateOnly 및 TimeOnly 형식 직렬화

.NET 7 이상에서는 `System.Text.Json` DateOnly 및 TimeOnly 형식의 직렬화 및 역직렬화를 지원합니다. 다음 개체를 고려합니다.

C#

```
sealed file record Appointment(
    Guid Id,
    string Description,
    DateOnly Date,
    TimeOnly StartTime,
    TimeOnly EndTime);
```

다음 예제에서는 `Appointment` 개체를 직렬화하고 결과 JSON을 표시한 다음 다시 `Appointment` 형식의 새 인스턴스로 역직렬화합니다. 마지막으로 원래 인스턴스와 새로 역직렬화된 인스턴스를 같음으로 비교하고 결과는 콘솔에 기록됩니다.

C#

```
Appointment originalAppointment = new(
    Id: Guid.NewGuid(),
    Description: "Take dog to veterinarian.",
    Date: new DateOnly(2002, 1, 13),
    StartTime: new TimeOnly(5,15),
    EndTime: new TimeOnly(5, 45));
string serialized = JsonSerializer.Serialize(originalAppointment);

Console.WriteLine($"Resulting JSON: {serialized}");

Appointment deserializedAppointment =
    JsonSerializer.Deserialize<Appointment>(serialized)!;

bool valuesAreTheSame = originalAppointment == deserializedAppointment;
Console.WriteLine($"""
    Original record has the same values as the deserialized record:
{valuesAreTheSame}
    """);
```

앞의 코드에서 다음을 수행합니다.

- `Appointment` 개체가 인스턴스화되고 `appointment` 변수에 할당됩니다.
- `appointment` 인스턴스는 JsonSerializer.Serialize사용하여 JSON으로 직렬화됩니다.
- 결과 JSON은 콘솔에 기록됩니다.
- JSON은 `Appointment`사용하여 JsonSerializer.Deserialize 형식의 새 인스턴스로 다시 역직렬화됩니다.
- 원래 인스턴스와 새로 역직렬화된 인스턴스가 동일한지 비교됩니다.
- 비교 결과는 콘솔에 기록됩니다.

자세한 내용은 .NETJSON을 직렬화하고 역직렬화하는 방법을 참조하세요.

TimeSpan 및 DateTime과 함께 작업하기

TimeOnly은(는) TimeSpan로부터 생성되고 TimeSpan으로 변환될 수 있습니다. 또한 `TimeOnly`와 DateTime을 사용하여 `TimeOnly` 인스턴스를 생성하거나, 날짜가 제공되는 경우 `DateTime` 인스턴스를 생성할 수 있습니다.

다음 예제에서는 `TimeOnly`에서 `TimeSpan` 개체를 만든 다음, 그 개체를 다시 변환합니다.

C#

```
// TimeSpan must in the range of 00:00:00.0000000 to 23:59:59.9999999
var theTime = TimeOnly.FromTimeSpan(new TimeSpan(23, 59, 59));
var theTimeSpan = theTime.ToTimeSpan();

Console.WriteLine($"Variable '{nameof(theTime)}' is {theTime}");
Console.WriteLine($"Variable '{nameof(theTimeSpan)}' is {theTimeSpan}");

/* This example produces the following output:
 *
 * Variable 'theTime' is 11:59 PM
 * Variable 'theTimeSpan' is 23:59:59
*/
```

다음 예제에서는 임의의 날짜가 선택된 `DateTime` 개체에서 `TimeOnly` 만듭니다.

C#

```
var theTime = new TimeOnly(11, 25, 46);   // 11:25 AM and 46 seconds
var theDate = new DateOnly(2015, 10, 21); // October 21, 2015
var theDateTime = theDate.ToDateTime(theTime);
var reverseTime = TimeOnly.FromDateTime(theDateTime);

Console.WriteLine($"Date only is {theDate}");
Console.WriteLine($"Time only is {theTime}");
Console.WriteLine();
```

```
Console.WriteLine($"Combined to a DateTime type, the value is {theDateTime}");
Console.WriteLine($"Converted back from DateTime, the time is {reverseTime}");

/* This example produces the following output:
 *
 * Date only is 10/21/2015
 * Time only is 11:25 AM
 *
 * Combined to a DateTime type, the value is 10/21/2015 11:25:46 AM
 * Converted back from DateTime, the time is 11:25 AM
*/
```

산술 연산자와 TimeOnly 비교

두 TimeOnly 인스턴스를 서로 비교할 수 있으며 IsBetween 메서드를 사용하여 시간이 두 다른 시점 사이에 있는지 확인할 수 있습니다. `TimeOnly`에 더하기 또는 빼기 연산자를 사용하면, 시간의 기간을 나타내는 TimeSpan이 반환됩니다.

C#

```
var start = new TimeOnly(10, 12, 01); // 10:12:01 AM
var end = new TimeOnly(14, 00, 53); // 02:00:53 PM

var outside = start.AddMinutes(-3);
var inside = start.AddMinutes(120);

Console.WriteLine($"Time starts at {start} and ends at {end}");
Console.WriteLine($" Is {outside} between the start and end?
{outside.IsBetween(start, end)}");
Console.WriteLine($" Is {inside} between the start and end? {inside.IsBetween(start,
end)}");
Console.WriteLine($" Is {start} less than {end}? {start < end}");
Console.WriteLine($" Is {start} greater than {end}? {start > end}");
Console.WriteLine($" Does {start} equal {end}? {start == end}");
Console.WriteLine($" The time between {start} and {end} is {end - start}");

/* This example produces the following output:
 *
 * Time starts at 10:12 AM and ends at 2:00 PM
 *  Is 10:09 AM between the start and end? False
 *  Is 12:12 PM between the start and end? True
 *  Is 10:12 AM less than 2:00 PM? True
 *  Is 10:12 AM greater than 2:00 PM? False
 *  Does 10:12 AM equal 2:00 PM? False
 *  The time between 10:12 AM and 2:00 PM is 03:48:52
*/
```

Last updated on 2026. 01. 16.

날짜 및 시간을 사용하여 산술 연산 수행

아티클 • 2025. 03. 23.

DateTime 구조와 DateTimeOffset 구조는 값에 대한 산술 연산을 수행하는 멤버를 제공하지만 산술 연산의 결과는 매우 다릅니다. 이 문서에서는 이러한 차이점을 검토하고 날짜 및 시간 데이터의 표준 시간대 인식 수준과 관련하며 날짜 및 시간 데이터를 사용하여 완전히 표준 시간대 인식 작업을 수행하는 방법을 설명합니다.

DateTime 값과 비교 및 산술 연산

DateTime.Kind 속성을 사용하면 날짜 및 시간에 DateTimeKind 값을 할당하여 현지 시간, UTC(협정 세계시) 또는 지정되지 않은 표준 시간대의 시간을 나타내는지 여부를 나타낼 수 있습니다. 그러나 이 제한된 표준 시간대 정보는 DateTimeKind 값에 대한 날짜 및 시간 산술 연산을 비교하거나 수행할 때 무시됩니다. 다음 예제에서는 현재 현지 시간을 현재 UTC 시간과 비교하여 표준 시간대 정보가 무시되는 방법을 보여 줍니다.

C#

```
using System;

public enum TimeComparison2
{
    EarlierThan = -1,
    TheSameAs = 0,
    LaterThan = 1
}

public class DateManipulation
{
    public static void Main()
    {
        DateTime localTime = DateTime.Now;
        DateTime utcTime = DateTime.UtcNow;

        Console.WriteLine("Difference between {0} and {1} time: {2}:{3}
hours",
                          localTime.Kind,
                          utcTime.Kind,
                          (localTime - utcTime).Hours,
                          (localTime - utcTime).Minutes);
        Console.WriteLine("The {0} time is {1} the {2} time.",
                          localTime.Kind,
                          Enum.GetName(typeof(TimeComparison2),
localTime.CompareTo(utcTime)),
                          utcTime.Kind);
    }
}
```

```
// If run in the U.S. Pacific Standard Time zone, the example displays
// the following output to the console:
//    Difference between Local and Utc time: -7:0 hours
//    The Local time is EarlierThan the Utc time.
```

CompareTo(DateTime) 메서드는 현지 시간이 UTC 시간보다 이전(또는 보다 작음)임을 보고하고 빼기 연산은 미국 태평양 표준시 영역에 있는 시스템의 UTC와 현지 시간 간의 차이가 7시간임을 나타냅니다. 그러나 이 두 값은 단일 시점의 다른 표현을 제공하기 때문에 이 경우 시간 간격이 UTC에서 현지 표준 시간대의 오프셋에 완전히 기인한다는 것이 분명합니다.

더 일반적으로, DateTime.Kind 속성은 Kind 비교 및 산술 메서드에서 반환된 결과에 영향을 주지 않지만(두 개의 동일한 시간점 비교를 예로 들 수 있습니다), 이러한 결과의 해석에는 영향을 미칠 수 있습니다. 다음은 그 예입니다.

- DateTime.Kind 속성이 모두 같은 두 날짜 및 시간 값에 대해 수행된 산술 연산의 결과는 두 값 사이의 실제 시간 간격을 반영하는 DateTimeKind. 마찬가지로 이러한 두 날짜 및 시간 값을 비교하면 시간 간의 관계가 정확하게 반영됩니다.

- 두 날짜 및 시간 값의 DateTime.Kind 속성이 모두 DateTimeKind과 같거나, 다른 DateTime.Kind 속성 값을 가진 두 날짜 및 시간 값에 대해 수행된 산술 또는 비교 연산의 결과는 두 값의 클록 시간 차이를 반영합니다.

- 현지 날짜 및 시간 값에 대한 산술 또는 비교 작업은 특정 값이 모호하거나 잘못된지 여부를 고려하지 않으며, 현지 표준 시간대의 일광 절약 시간제 전환으로 인한 조정 규칙의 영향을 고려하지 않습니다.

- UTC와 현지 시간의 차이를 비교하거나 계산하는 모든 작업에는 결과에서 UTC의 현지 표준 시간대 오프셋과 동일한 시간 간격이 포함됩니다.

- 지정되지 않은 시간과 UTC 또는 현지 시간의 차이를 비교하거나 계산하는 모든 작업은 간단한 클록 시간을 반영합니다. 표준 시간대 차이는 고려되지 않으며 결과는 표준 시간대 조정 규칙의 적용을 반영하지 않습니다.

- 지정되지 않은 두 시간의 차이를 비교하거나 계산하는 모든 작업에는 서로 다른 두 표준 시간대의 시간 차이를 반영하는 알 수 없는 간격이 포함될 수 있습니다.

표준 시간대 차이가 날짜 및 시간 계산에 영향을 미치지 않는 많은 시나리오가 있습니다(이러한 시나리오 중 일부에 대한 설명은 DateTime, DateTimeOffset, TimeSpan 및 TimeZoneInfo중에서 선택 참조)하거나 날짜 및 시간 데이터의 컨텍스트에서 비교 또는 산술 연산의 의미를 정의합니다.

DateTimeOffset 값을 사용하여 비교 및 산술 연산

DateTimeOffset 값에는 날짜 및 시간뿐만 아니라 UTC를 기준으로 해당 날짜와 시간을 명확하게 정의하는 오프셋도 포함됩니다. 이 오프셋을 사용하면 DateTime 값과 다르게 같음을 정의할 수 있습니다. DateTime 값은 날짜 및 시간 값이 같으면 같지만 DateTimeOffset 값은 모두 같은 시점을 참조하는 경우 같습니다. 비교 및 두 날짜와 시간 사이의 간격을 결정하는 대부분의 산술 연산에서 사용되는 경우 DateTimeOffset 값이 더 정확하고 해석이 덜 필요합니다. 다음 예제는 로컬 및 UTC DateTimeOffset 값을 비교한 이전 예제와 동일한 DateTimeOffset 동작의 차이를 보여 줍니다.

C#

```
using System;

public enum TimeComparison
{
    EarlierThan = -1,
    TheSameAs = 0,
    LaterThan = 1
}

public class DateTimeOffsetManipulation
{
    public static void Main()
    {
        DateTimeOffset localTime = DateTimeOffset.Now;
        DateTimeOffset utcTime = DateTimeOffset.UtcNow;

        Console.WriteLine("Difference between local time and UTC: {0}:{1:D2}
hours",
                          (localTime - utcTime).Hours,
                          (localTime - utcTime).Minutes);
        Console.WriteLine("The local time is {0} UTC.",
                          Enum.GetName(typeof(TimeComparison),
localTime.CompareTo(utcTime)));
    }
}
// Regardless of the local time zone, the example displays
// the following output to the console:
//    Difference between local time and UTC: 0:00 hours.
//    The local time is TheSameAs UTC.
```

이 예제에서 CompareTo 메서드는 현재 현지 시간과 현재 UTC 시간이 같으며 CompareTo(DateTimeOffset) 값을 빼면 두 시간 사이의 차이가 TimeSpan.Zero나타냅니다.

날짜 및 시간 산술 연산에서 DateTimeOffset 값을 사용하는 주요 제한 사항은 DateTimeOffset 값에 표준 시간대 인식이 있지만 완전히 표준 시간대를 인식하지는 않는다는 것입니다. DateTimeOffset 값의 오프셋은 DateTimeOffset 변수에 값이 처음 할당될 때 UTC에서 표준 시간대의 오프셋을 반영하지만 이후 표준 시간대에서 연결이 해제됩니다. 더 이상 식별 가능한 시간과 직접 연결되지 않으므로 날짜 및 시간 간격의 추가 및 빼기는 표준 시간대의 조정 규칙을 고려하지 않습니다.

이를 설명하기 위해 미국 중부 표준 시간대의 일광 절약 시간제로의 전환은 2008년 3월 9일 오전 2:00에 발생합니다. 이를 염두에 두고 2008년 3월 9일 오전 1시 30분에 중부 표준 시간대에 2시간 반 간격을 추가하면 2008년 3월 9일 오전 5:00의 날짜와 시간이 생성됩니다. 그러나 다음 예제와 같이 추가 결과는 2008년 3월 9일 오전 4:00입니다. 이 작업의 결과는 올바른 시점을 나타내지만 관심 있는 표준 시간대의 시간(즉, 예상 표준 시간대 오프셋이 없음)은 아닙니다.

C#

```
using System;

public class IntervalArithmetic
{
    public static void Main()
    {
        DateTime generalTime = new DateTime(2008, 3, 9, 1, 30, 0);
        const string tzName = "Central Standard Time";
        TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

        // Instantiate DateTimeOffset value to have correct CST offset
        try
        {
            DateTimeOffset centralTime1 = new DateTimeOffset(generalTime,

TimeZoneInfo.FindSystemTimeZoneById(tzName).GetUtcOffset(generalTime));

            // Add two and a half hours
            DateTimeOffset centralTime2 =
centralTime1.Add(twoAndAHalfHours);
            // Display result
            Console.WriteLine("{0} + {1} hours = {2}", centralTime1,

twoAndAHalfHours.ToString(),
                                                       centralTime2);
        }
        catch (TimeZoneNotFoundException)
        {
            Console.WriteLine("Unable to retrieve Central Standard Time zone
information.");
        }
    }
}
// The example displays the following output to the console:
```

```
//    3/9/2008 1:30:00 AM -06:00 + 02:30:00 hours = 3/9/2008 4:00:00 AM
-06:00
```

표준 시간대의 시간을 사용하는 산술 연산

TimeZoneInfo 클래스에는 시간을 표준 시간대에서 다른 표준 시간대로 변환할 때 조정을 자동으로 적용하는 변환 메서드가 포함되어 있습니다. 이러한 변환 방법은 다음과 같습니다.

- 두 표준 시간대 간의 시간을 변환하는 ConvertTime 및 ConvertTimeBySystemTimeZoneId 메서드입니다.
- 특정 표준 시간대의 시간을 UTC로 변환하거나 UTC를 특정 표준 시간대의 시간으로 변환하는 ConvertTimeFromUtc 및 ConvertTimeToUtc 메서드입니다.

자세한 내용은 표준 시간대간의 시간 변환을 참조하세요.

TimeZoneInfo 클래스는 날짜 및 시간 산술 연산을 수행할 때 조정 규칙을 자동으로 적용하는 메서드를 제공하지 않습니다. 그러나 표준 시간대의 시간을 UTC로 변환하고 산술 연산을 수행한 다음 UTC에서 표준 시간대의 시간으로 다시 변환하여 조정 규칙을 적용할 수 있습니다. 자세한 내용은 날짜 및 시간 계산에서 시간대를 사용하는 방법에 관한 내용입니다.

예를 들어 다음 코드는 2008년 3월 9일 오전 2:00에 2시간 반을 추가한 이전 코드와 유사합니다. 그러나 날짜 및 시간 산술 연산을 수행하기 전에 중앙 표준 시간을 UTC로 변환한 다음 결과를 UTC에서 다시 중앙 표준 시간으로 변환하기 때문에 결과 시간은 일광 절약 시간으로의 중앙 표준 시간대 전환을 반영합니다.

C#

```
using System;

public class TimeZoneAwareArithmetic
{
    public static void Main()
    {
        const string tzName = "Central Standard Time";

        DateTime generalTime = new DateTime(2008, 3, 9, 1, 30, 0);
        TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
        TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

        // Instantiate DateTimeOffset value to have correct CST offset
        try
        {
            DateTimeOffset centralTime1 = new DateTimeOffset(generalTime,
                                                cst.GetUtcOffset(generalTime));
```

```
            // Add two and a half hours
            DateTimeOffset utcTime = centralTime1.ToUniversalTime();
            utcTime += twoAndAHalfHours;

            DateTimeOffset centralTime2 = TimeZoneInfo.ConvertTime(utcTime,
cst);
            // Display result
            Console.WriteLine("{0} + {1} hours = {2}", centralTime1,

twoAndAHalfHours.ToString(),
                                                       centralTime2);
        }
        catch (TimeZoneNotFoundException)
        {
            Console.WriteLine("Unable to retrieve Central Standard Time zone
information.");
        }
    }
}
// The example displays the following output to the console:
//    3/9/2008 1:30:00 AM -06:00 + 02:30:00 hours = 3/9/2008 5:00:00 AM
-05:00
```

참고하십시오

- 날짜, 시간 및 표준 시간대
- 방법: 날짜 및 시간 계산에서 표준 시간대를 사용하는 방법

TimeProvider란?

System.TimeProvider 특정 시점을 DateTimeOffset 형식으로 제공하는 시간의 추상화입니다. `TimeProvider`사용하여 코드가 테스트 가능하고 예측 가능한지 확인합니다. `TimeProvider` 는 다음 프레임워크에서 사용할 수 있습니다.

테이블 확장

프레임워크	비고
.NET 8 이상	프레임워크에 포함됩니다.
.NET 5 - .NET 7	NuGet 패키지에Microsoft.Bcl.TimeProvider 제공됩니다.
.NET Framework 4.6.2 이상	NuGet 패키지에Microsoft.Bcl.TimeProvider 제공됩니다.
.NET Standard 2.0	NuGet 패키지에Microsoft.Bcl.TimeProvider 제공됩니다.

TimeProvider 클래스는 다음 기능을 정의합니다.

- TimeProvider.GetUtcNow() 및 TimeProvider.GetLocalNow()통해 날짜 및 시간에 대한 액세스를 제공합니다.
- TimeProvider.GetTimestamp()있는 빈도가 높은 타임스탬프입니다.
- TimeProvider.GetElapsedTime사용하여 두 타임스탬프 사이의 시간을 측정합니다.
- TimeProvider.CreateTimer(TimerCallback, Object, TimeSpan, TimeSpan)있는 고해상도 타이머.
- TimeProvider.LocalTimeZone사용하여 현재 표준 시간대를 가져옵니다.

기본 구현

.NET은 다음과 같은 특성을 사용하여 TimeProvider 속성을 통해 TimeProvider.System 구현합니다.

- 날짜 및 시간은 DateTimeOffset.UtcNow 및 TimeZoneInfo.Local를 사용하여 계산됩니다.
- 타임스탬프는 System.Diagnostics.Stopwatch에 의해 제공됩니다.
- 타이머는 내부 클래스를 통해 구현되고 System.Threading.ITimer로 노출됩니다.

다음 예제에서는 현재 날짜 및 시간을 가져오는 데 사용하는 TimeProvider 방법을 보여줍니다.

C#

```
Console.WriteLine($"Local: {TimeProvider.System.GetLocalNow()}");
Console.WriteLine($"Utc:   {TimeProvider.System.GetUtcNow()}");

/* This example produces output similar to the following:
```

```
 *
 * Local: 12/5/2024 10:41:14 AM -08:00
 * Utc:   12/5/2024 6:41:14 PM +00:00
*/
```

다음 예제에서는 다음을 사용하여 TimeProvider.GetTimestamp()경과된 시간을 캡처하는 방법을 보여줍니다.

C#

```
long stampStart = TimeProvider.System.GetTimestamp();
Console.WriteLine($"Starting timestamp: {stampStart}");

long stampEnd = TimeProvider.System.GetTimestamp();
Console.WriteLine($"Ending timestamp:   {stampEnd}");

Console.WriteLine($"Elapsed time: {TimeProvider.System.GetElapsedTime(stampStart,
stampEnd)}");
Console.WriteLine($"Nanoseconds: {TimeProvider.System.GetElapsedTime(stampStart,
stampEnd).TotalNanoseconds}");

/* This example produces output similar to the following:
 *
 * Starting timestamp: 55185546133
 * Ending timestamp:   55185549929
 * Elapsed time: 00:00:00.0003796
 * Nanoseconds: 379600
*/
```

FakeTimeProvider 구현

Microsoft.Extensions.TimeProvider.Testing NuGet 패키지 단위 테스트를 위해 설계된 제어 가능한 `TimeProvider` 구현을 제공합니다.

다음 목록에서는 FakeTimeProvider 클래스의 일부 기능을 설명합니다.

- 특정 날짜 및 시간을 설정합니다.
- 날짜와 시간이 읽힐 때마다 이를 지정된 양만큼 자동으로 진행합니다.
- 날짜 및 시간을 수동으로 설정하세요.

사용자 지정 구현

FakeTimeProvider는 시간에 따라 예측 가능성이 필요한 대부분의 시나리오를 다루지만 사용자 고유의 구현을 제공할 수 있습니다. TimeProvider에서 파생된 새 클래스를 만들고, 시간을 어떻게 제공할지 제어할 수 있도록 멤버를 재정의하세요. 예를 들어 다음 클래스는 달 착륙 날짜인 단일 날짜만 제공합니다.

C#

```
public class MoonLandingTimeProviderPST: TimeProvider
{
    // July 20, 1969, at 20:17:40 UTC
    private readonly DateTimeOffset _specificDateTime = new(1969, 7, 20, 20, 17, 40,
TimeZoneInfo.Utc.BaseUtcOffset);

    public override DateTimeOffset GetUtcNow() => _specificDateTime;

    public override TimeZoneInfo LocalTimeZone =>
TimeZoneInfo.FindSystemTimeZoneById("PST");
}
```

이 클래스를 사용하는 코드가 `MoonLandingTimeProviderPST.GetUtcNow`호출하면 UTC에서 달 착륙 날짜가 반환됩니다. `MoonLandingTimeProviderPST.GetLocalNow`이(가) 호출되면, 기본 클래스는 `MoonLandingTimeProviderPST.LocalTimeZone`을(를) `GetUtcNow`에 적용하고 PST 시간대의 달 착륙 날짜 및 시간을 반환합니다.

시간 제어의 유용성을 보여 주려면 다음 예제를 참조하세요. 매일 앱이 처음 열릴 때 사용자에게 인사말을 보내는 일정 앱을 작성한다고 가정해 보겠습니다. 앱은 현재 날에 달 착륙 기념일과 같은 이벤트와 관련된 이벤트가 있을 때 특별한 인사말을 말합니다.

C#

```
public static class CalendarHelper
{
    static readonly DateTimeOffset MoonLandingDateTime = new(1969, 7, 20, 20, 17,
40, TimeZoneInfo.Utc.BaseUtcOffset);

    public static void SendGreeting(TimeProvider currentTime, string name)
    {
        DateTimeOffset localTime = currentTime.GetLocalNow();

        Console.WriteLine($"Good morning, {name}!");
        Console.WriteLine($"The date is {localTime.Date:d} and the day is
{localTime.Date.DayOfWeek}.");

        if (localTime.Date.Month == MoonLandingDateTime.Date.Month
            && localTime.Date.Day == MoonLandingDateTime.Date.Day)
        {
            Console.WriteLine("Did you know that on this day in 1969 humans landed
on the Moon?");
        }

        Console.WriteLine($"I hope you enjoy your day!");
    }
}
```

DateTime대신 DateTimeOffset 또는 TimeProvider 사용하여 이전 코드를 작성하여 현재 날짜와 시간을 가져오는 것이 좋습니다. 그러나 단위 테스트를 사용하면 직접 DateTime 또는 DateTimeOffset 해결하기가 어렵습니다. 달 착륙 날짜와 달에 테스트를 실행하거나 코드를 더 작지만 테스트 가능한 단위로 추상화해야 합니다.

앱의 정상적인 작업은 `TimeProvider.System` 사용하여 현재 날짜 및 시간을 검색합니다.

C#

```
CalendarHelper.SendGreeting(TimeProvider.System, "Eric Solomon");

/* This example produces output similar to the following:
 *
 * Good morning, Eric Solomon!
 * The date is 12/5/2024 and the day is Thursday.
 * I hope you enjoy your day!
*/
```

그리고 단위 테스트를 작성하여 달 착륙 기념일 테스트와 같은 특정 시나리오를 테스트할 수 있습니다.

C#

```
CalendarHelper.SendGreeting(new MoonLandingTimeProviderPST(), "Eric Solomon");

/* This example produces output similar to the following:
 *
 * Good morning, Eric Solomon!
 * The date is 7/20/1969 and the day is Sunday.
 * Did you know that on this day in 1969 humans landed on the Moon?
 * I hope you enjoy your day!
*/
```

.NET과 함께 사용

.NET 8부터 런타임 라이브러리는 클래스를 TimeProvider 제공합니다. .NET Standard 2.0을 대상으로 하는 이전 버전의 .NET 또는 라이브러리는 NuGet 패키지를 참조 Microsoft.Bcl.TimeProvider 해야 합니다.

비동기 프로그래밍과 관련된 다음 메서드는 `TimeProvider`와 함께 작업합니다.

- CancellationTokenSource(TimeSpan, TimeProvider)
- Task.Delay(TimeSpan, TimeProvider)
- Task.Delay(TimeSpan, TimeProvider, CancellationToken)
- Task.WaitAsync(TimeSpan, TimeProvider)
- Task.WaitAsync(TimeSpan, TimeProvider, CancellationToken)

.NET Framework와 함께 사용

NuGet 패키지는Microsoft.Bcl.TimeProvider .를 구현합니다.TimeProvider

비동기 프로그래밍 시나리오에서 `TimeProvider` 작업에 대한 지원은 다음 확장 메서드를 통해 추가되었습니다.

- TimeProviderTaskExtensions.CreateCancellationTokenSource(TimeProvider, TimeSpan)
- TimeProviderTaskExtensions.Delay(TimeProvider, TimeSpan, CancellationToken)
- TimeProviderTaskExtensions.WaitAsync(Task, TimeSpan, TimeProvider, CancellationToken)
- TimeProviderTaskExtensions.WaitAsync<TResult>(Task<TResult>, TimeSpan, TimeProvider, CancellationToken)

Last updated on 2026. 01. 22.

System.Text.Json의 DateTime 및 DateTimeOffset 지원

`System.Text.Json` 라이브러리는 ISO 8601-1:2019 확장 프로필에 따라 DateTime 및 DateTimeOffset 값을 구문 분석하고 씁니다. 변환기는 JsonSerializer을(를) 사용하여 직렬화 및 역직렬화에 대한 맞춤 지원을 제공합니다. Utf8JsonReader 및 Utf8JsonWriter을(를) 사용하여 사용자 지정 지원을 구현할 수도 있습니다.

ISO 8601-1:2019 형식 지원

JsonSerializer, Utf8JsonReader, Utf8JsonWriter 및 JsonElement 유형은 ISO 8601-1:2019 형식의 확장 프로필에 따라 DateTime 및 DateTimeOffset 텍스트 표현을 구문 분석하고 작성합니다. 예를 들어 `2019-07-26T16:59:57-05:00`과 같습니다.

DateTime 및 DateTimeOffset 데이터는 JsonSerializer사용하여 serialize 할 수 있습니다.

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        Product p = new Product();
        p.Name = "Banana";
        p.ExpiryDate = new DateTime(2019, 7, 26);

        string json = JsonSerializer.Serialize(p);
        Console.WriteLine(json);
    }
}

// The example displays the following output:
// {"Name":"Banana","ExpiryDate":"2019-07-26T00:00:00"}
```

DateTime 및 DateTimeOffsetJsonSerializer을(를) 사용하여 역직렬화할 수도 있습니다.

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        string json = @"{""Name"":""Banana"",""ExpiryDate"":""2019-07-
26T00:00:00""}";
        Product p = JsonSerializer.Deserialize<Product>(json)!;
        Console.WriteLine(p.Name);
        Console.WriteLine(p.ExpiryDate);
    }
}

// The example displays output similar to the following:
// Banana
// 7/26/2019 12:00:00 AM
```

기본 옵션을 사용하면 입력 DateTime 및 DateTimeOffset 텍스트 표현이 확장된 ISO 8601-1:2019 프로필을 따라야 합니다. 프로필을 JsonSerializer 준수하지 않는 표현을 역직렬화하려고 하면 JsonException를 던집니다.

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        string json = @"{""Name"":""Banana"",""ExpiryDate"":""26/07/2019""}";
        try
        {
            Product _ = JsonSerializer.Deserialize<Product>(json)!;
        }
        catch (JsonException e)
        {
            Console.WriteLine(e.Message);
        }
    }
```

```
}

// The example displays the following output:
// The JSON value could not be converted to System.DateTime. Path: $.ExpiryDate |
LineNumber: 0 | BytePositionInLine: 42.
```

JsonDocument 에서는 JSON 페이로드의 콘텐츠(예: 표현) DateTimeDateTimeOffset 에 대한 구조적 액세스를 제공합니다. 다음 예제에서는 온도 컬렉션에서 월요일의 평균 온도를 계산하는 방법을 보여 줍니다.

C#

```
using System.Text.Json;

public class Example
{
    private static double ComputeAverageTemperatures(string json)
    {
        JsonDocumentOptions options = new JsonDocumentOptions
        {
            AllowTrailingCommas = true
        };

        using (JsonDocument document = JsonDocument.Parse(json, options))
        {
            int sumOfAllTemperatures = 0;
            int count = 0;

            foreach (JsonElement element in document.RootElement.EnumerateArray())
            {
                DateTimeOffset date =
element.GetProperty("date").GetDateTimeOffset();

                if (date.DayOfWeek == DayOfWeek.Monday)
                {
                    int temp = element.GetProperty("temp").GetInt32();
                    sumOfAllTemperatures += temp;
                    count++;
                }
            }

            double averageTemp = (double)sumOfAllTemperatures / count;
            return averageTemp;
        }
    }

    public static void Main(string[] args)
    {
        string json =
                @"[" +
                    @"{" +
                        @"""date"": ""2013-01-07T00:00:00Z""," +
                        @"""temp"": 23," +
```

```
                    @"}," +
                    @"{" +
                        @"""date"": ""2013-01-08T00:00:00Z""," +
                        @"""temp"": 28," +
                    @"}," +
                    @"{" +
                        @"""date"": ""2013-01-14T00:00:00Z""," +
                        @"""temp"": 8," +
                    @"}," +
                @"]";

        Console.WriteLine(ComputeAverageTemperatures(json));
    }
}

// The example displays the following output:
// 15.5
```

비준수 DateTime 표현으로 페이로드가 주어졌을 때 평균 온도를 계산하려고 하면, JsonDocument에서 FormatException를 발생시킵니다.

C#

```
using System.Text.Json;

public class Example
{
    private static double ComputeAverageTemperatures(string json)
    {
        JsonDocumentOptions options = new JsonDocumentOptions
        {
            AllowTrailingCommas = true
        };

        using (JsonDocument document = JsonDocument.Parse(json, options))
        {
            int sumOfAllTemperatures = 0;
            int count = 0;

            foreach (JsonElement element in document.RootElement.EnumerateArray())
            {
                DateTimeOffset date =
element.GetProperty("date").GetDateTimeOffset();

                if (date.DayOfWeek == DayOfWeek.Monday)
                {
                    int temp = element.GetProperty("temp").GetInt32();
                    sumOfAllTemperatures += temp;
                    count++;
                }
            }

            double averageTemp = (double)sumOfAllTemperatures / count;
```

```
            return averageTemp;
        }
    }

    public static void Main(string[] args)
    {
        // Computing the average temperatures will fail because the DateTimeOffset
        // values in the payload do not conform to the extended ISO 8601-1:2019
profile.
        string json =
                @"[" +
                    @"{" +
                        @"""date"": ""2013/01/07 00:00:00Z""," +
                        @"""temp"": 23," +
                    @"}," +
                    @"{" +
                        @"""date"": ""2013/01/08 00:00:00Z""," +
                        @"""temp"": 28," +
                    @"}," +
                    @"{" +
                        @"""date"": ""2013/01/14 00:00:00Z""," +
                        @"""temp"": 8," +
                    @"}," +
                @"]";

        Console.WriteLine(ComputeAverageTemperatures(json));
    }
}

// The example displays the following output:
// Unhandled exception.System.FormatException: One of the identified items was in
an invalid format.
//    at System.Text.Json.JsonElement.GetDateTimeOffset()
```

하위 수준 Utf8JsonWriterDateTime 및 DateTimeOffset 데이터를 씁니다.

C#

```
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        JsonWriterOptions options = new JsonWriterOptions
        {
            Indented = true
        };

        using (MemoryStream stream = new MemoryStream())
        {
            using (Utf8JsonWriter writer = new Utf8JsonWriter(stream, options))
```

```
            {
                writer.WriteStartObject();
                writer.WriteString("date", DateTimeOffset.UtcNow);
                writer.WriteNumber("temp", 42);
                writer.WriteEndObject();
            }

            string json = Encoding.UTF8.GetString(stream.ToArray());
            Console.WriteLine(json);
        }
    }
}

// The example output similar to the following:
// {
//     "date": "2019-07-26T00:00:00+00:00",
//     "temp": 42
// }
```

Utf8JsonReader는 DateTime 및 DateTimeOffset 데이터를 구문 분석합니다.

C#

```
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""2019-07-26T00:00:00""");

        Utf8JsonReader json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                Console.WriteLine(json.TryGetDateTime(out DateTime datetime));
                Console.WriteLine(datetime);
                Console.WriteLine(json.GetDateTime());
            }
        }
    }
}

// The example displays output similar to the following:
// True
// 7/26/2019 12:00:00 AM
// 7/26/2019 12:00:00 AM
```

호환되지 않는 형식 Utf8JsonReader을 읽으려고 하면 다음과 같은 오류가 발생합니다: FormatException.

C#

```
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""2019/07/26 00:00:00""");

        Utf8JsonReader json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                Console.WriteLine(json.TryGetDateTime(out DateTime datetime));
                Console.WriteLine(datetime);

                DateTime _ = json.GetDateTime();
            }
        }
    }
}

// The example displays the following output:
// False
// 1/1/0001 12:00:00 AM
// Unhandled exception. System.FormatException: The JSON value is not in a
supported DateTime format.
//     at System.Text.Json.Utf8JsonReader.GetDateTime()
```

DateOnly 및 TimeOnly 속성 직렬화

.NET 7부터 `System.Text.Json`는 DateOnly 및 TimeOnly 형식의 직렬화 및 역직렬화를 지원합니다. 다음 개체를 고려해 보세요.

C#

```
sealed file record Appointment(
    Guid Id,
    string Description,
    DateOnly Date,
    TimeOnly StartTime,
    TimeOnly EndTime);
```

다음 예제에서는 `Appointment` 개체를 직렬화하고 결과 JSON을 표시한 다음, `Appointment` 형식의 새 인스턴스로 다시 역직렬화합니다. 마지막으로 원래 인스턴스와 새로 역직렬화된 인스턴스가 같은지 비교된 후 결과가 콘솔에 기록됩니다.

C#

```
Appointment originalAppointment = new(
    Id: Guid.NewGuid(),
    Description: "Take dog to veterinarian.",
    Date: new DateOnly(2002, 1, 13),
    StartTime: new TimeOnly(5,15),
    EndTime: new TimeOnly(5, 45));
string serialized = JsonSerializer.Serialize(originalAppointment);

Console.WriteLine($"Resulting JSON: {serialized}");

Appointment deserializedAppointment =
    JsonSerializer.Deserialize<Appointment>(serialized)!;

bool valuesAreTheSame = originalAppointment == deserializedAppointment;
Console.WriteLine($"""
    Original record has the same values as the deserialized record:
{valuesAreTheSame}
    """);
```

위의 코드에서

- `Appointment` 개체가 인스턴스화되고 `appointment` 변수에 할당됩니다.
- `appointment` 인스턴스가 JsonSerializer.Serialize를 사용하여 JSON으로 직렬화됩니다.
- 결과 JSON이 콘솔에 기록됩니다.
- JSON은 `Appointment`를 사용하여 JsonSerializer.Deserialize 형식의 새 인스턴스로 다시 역직렬화됩니다.
- 원래 인스턴스와 새로 역직렬화된 인스턴스가 같은지 비교됩니다.
- 비교 결과는 콘솔에 기록됩니다.

DateTime 및 DateTimeOffset에 대한 사용자 지정 지원

JsonSerializer을(를) 사용하는 경우

직렬 변환기가 사용자 지정 구문 분석 또는 서식 지정을 수행하도록 하려면 사용자 지정 변환기를 구현할 수 있습니다. 다음 섹션에서는 몇 가지 예를 보여 줍니다.

- DateTime(Offset).Parse 및 DateTime(Offset).ToString
- Utf8Parser 및 Utf8Formatter
- 대체 방법으로 DateTime(오프셋).Parse를 사용합니다
- Unix epoch 날짜 형식 사용

DateTime(오프셋). 구문 분석 및 DateTime(오프셋). ToString

입력 DateTime 또는 DateTimeOffset 텍스트 표현의 형식을 확인할 수 없는 경우 변환기 읽기 논리에서 `DateTime(Offset).Parse` 메서드를 사용할 수 있습니다. 이 메서드를 사용하면 확장된 ISO 8601-1:2019 프로필을 준수하지 않는 ISO 8601 형식과 ISO 8601이 아닌 문자열을 포함하여 다양한 DateTime 및 DateTimeOffset 텍스트 형식을 구문 분석할 수 있는 .NET의 광범위한 지원을 활용할 수 있습니다. 이 방법은 serializer의 네이티브 구현을 사용하는 것보다 성능이 떨어집니다.

직렬화의 경우 변환기 쓰기 논리에서 `DateTime(Offset).ToString` 메서드를 사용할 수 있습니다. 이 메서드를 사용하면 DateTime과 DateTimeOffset을 사용하여 및 값을 작성할 수 있습니다. 또한 이 방법은 serializer의 네이티브 구현을 사용하는 것보다 성능이 떨어집니다.

C#

```
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

namespace DateTimeConverterExamples;

public class DateTimeConverterUsingDateTimeParse : JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert,
JsonSerializerOptions options)
    {
        Debug.Assert(typeToConvert == typeof(DateTime));
        return DateTime.Parse(reader.GetString() ?? string.Empty);
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
    {
        writer.WriteStringValue(value.ToString());
    }
}

class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""04-10-2008 6:30
AM""");
    }

    private static void FormatDateTimeWithDefaultOptions()
    {
        Console.WriteLine(JsonSerializer.Serialize(DateTime.Parse("04-10-2008 6:30
AM -4")));
    }
```

```
    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new DateTimeConverterUsingDateTimeParse());

        string testDateTimeStr = "04-10-2008 6:30 AM";
        string testDateTimeJson = @""""" + testDateTimeStr + @""""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        string resultDateTimeJson =
JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr), options);
        Console.WriteLine(Regex.Unescape(resultDateTimeJson));
    }

    static void Main(string[] args)
    {
        // Parsing non-compliant format as DateTime fails by default.
        try
        {
            ParseDateTimeWithDefaultOptions();
        }
        catch (JsonException e)
        {
            Console.WriteLine(e.Message);
        }

        // Formatting with default options prints according to extended ISO 8601
profile.
        FormatDateTimeWithDefaultOptions();

        // Using converters gives you control over the serializers parsing and
formatting.
        ProcessDateTimeWithCustomConverter();
    }
}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime. Path: $ | LineNumber:
0 | BytePositionInLine: 20.
// "2008-04-10T06:30:00-04:00"
// 4/10/2008 6:30:00 AM
// "4/10/2008 6:30:00 AM"
```

ⓘ 참고 항목

JsonConverter<T>구현하고 `T`이(가) DateTime인 경우, `typeToConvert` 매개 변수는 항상 `typeof(DateTime)`입니다. 이 매개 변수는 다형 사례를 처리하고 성능이 좋은 방식으로 `typeof(T)`을(를) 얻기 위해 제네릭을 사용할 때 유용합니다.

Utf8Parser 및 Utf8Formatter

입력 DateTime 또는 DateTimeOffset 텍스트 표현이 "R", "I", "O" 또는 "G" 표준 날짜 및 시간 형식 문자열 중 하나를 준수하거나 이러한 형식 중 하나에 따라 작성하려는 경우 변환기 논리에서 빠른 UTF-8 기반 구문 분석 및 서식 지정 메서드를 사용할 수 있습니다. 이 방법은 사용 `DateTime(Offset).Parse` 및 `DateTime(Offset).ToString`보다 훨씬 빠릅니다.

다음 예제에서는 DateTime에 따라 값을 직렬화하고 역직렬화하는 사용자 지정 변환기를 보여 줍니다.

C#

```
using System.Buffers;
using System.Buffers.Text;
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace DateTimeConverterExamples;

// This converter reads and writes DateTime values according to the "R" standard
format specifier:
// https://learn.microsoft.com/dotnet/standard/base-types/standard-date-and-time-
format-strings#the-rfc1123-r-r-format-specifier.
public class DateTimeConverterForCustomStandardFormatR : JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert,
JsonSerializerOptions options)
    {
        Debug.Assert(typeToConvert == typeof(DateTime));

        if (Utf8Parser.TryParse(reader.ValueSpan, out DateTime value, out _, 'R'))
        {
            return value;
        }

        throw new FormatException();
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
    {
        // The "R" standard format will always be 29 bytes.
        Span<byte> utf8Date = new byte[29];
```

```
        bool result = Utf8Formatter.TryFormat(value, utf8Date, out _, new
StandardFormat('R'));
        Debug.Assert(result);

        writer.WriteStringValue(utf8Date);
    }
}

class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""Thu, 25 Jul 2019
13:36:07 GMT""");
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new DateTimeConverterForCustomStandardFormatR());

        string testDateTimeStr = "Thu, 25 Jul 2019 13:36:07 GMT";
        string testDateTimeJson = @""""" + testDateTimeStr + @""""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        Console.WriteLine(JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr),
options));
    }

    static void Main(string[] args)
    {
        // Parsing non-compliant format as DateTime fails by default.
        try
        {
            ParseDateTimeWithDefaultOptions();
        }
        catch (JsonException e)
        {
            Console.WriteLine(e.Message);
        }

        // Using converters gives you control over the serializers parsing and
formatting.
        ProcessDateTimeWithCustomConverter();
    }
}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime.Path: $ | LineNumber: 0
| BytePositionInLine: 31.
// 7/25/2019 1:36:07 PM
// "Thu, 25 Jul 2019 09:36:07 GMT"
```

ⓘ **참고 항목**

"R" 표준 형식은 항상 29자 길이입니다.

"l"(소문자 "L") 형식은 다른 **표준 날짜 및 시간 형식 문자열**과 함께 문서화되지 않는 이유는 `Utf8Parser` 및 `Utf8Formatter` 유형에서만 지원되기 때문입니다. 형식은 소문자 RFC 1123("R" 형식의 소문자 버전)입니다. 예: "thu, 25 jul 2019 06:36:07 gmt".

DateTime(Offset).Parse를 대체 수단으로 사용하십시오.

일반적으로 입력 DateTime 또는 DateTimeOffset 데이터가 확장된 ISO 8601-1:2019 프로필을 따를 것으로 예상되는 경우, serializer의 네이티브 구문 분석 로직을 사용할 수 있습니다. 대체 메커니즘을 구현할 수도 있습니다. 다음 예제에서는 DateTime을(를) 사용하여 TryGetDateTime(DateTime) 텍스트 표현을 구문 분석하지 못한 후 변환기가 Parse(String)을(를) 사용하여 데이터를 구문 분석하는 방법을 보여 줍니다.

C#

```
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

namespace DateTimeConverterExamples;

public class DateTimeConverterUsingDateTimeParseAsFallback :
JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert,
JsonSerializerOptions options)
    {
        Debug.Assert(typeToConvert == typeof(DateTime));

        if (!reader.TryGetDateTime(out DateTime value))
        {
            value = DateTime.Parse(reader.GetString()!);
        }

        return value;
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
    {
        writer.WriteStringValue(value.ToString("dd/MM/yyyy"));
    }
}
```

```
class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""2019-07-16
16:45:27.4937872+00:00""");
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new
DateTimeConverterUsingDateTimeParseAsFallback());

        string testDateTimeStr = "2019-07-16 16:45:27.4937872+00:00";
        string testDateTimeJson = @""" + testDateTimeStr + @""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        string resultDateTimeJson =
JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr), options);
        Console.WriteLine(Regex.Unescape(resultDateTimeJson));
    }

    static void Main(string[] args)
    {
        // Parsing non-compliant format as DateTime fails by default.
        try
        {
            ParseDateTimeWithDefaultOptions();
        }
        catch (JsonException e)
        {
            Console.WriteLine(e.Message);
        }

        // Using converters gives you control over the serializers parsing and
formatting.
        ProcessDateTimeWithCustomConverter();
    }
}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime.Path: $ | LineNumber: 0
| BytePositionInLine: 35.
// 7/16/2019 4:45:27 PM
// "16/07/2019"
```

Unix epoch 날짜 형식 사용

다음 변환기는 표준 시간대 오프셋(`/Date(1590863400000-0700)/` 또는 `/Date(1590863400000)/` 같은 값)을 사용하거나 사용하지 않고 Unix epoch 형식을 처리합니다.

C#

```
sealed class UnixEpochDateTimeOffsetConverter :
System.Text.Json.Serialization.JsonConverter<DateTimeOffset>
{
    static readonly DateTimeOffset s_epoch = new(1970, 1, 1, 0, 0, 0,
TimeSpan.Zero);
    static readonly Regex s_regex = new(
        "^/Date\\(([+-]*\\d+)([+-])(\\d{2})(\\d{2})\\)/$",
        RegexOptions.CultureInvariant);

    public override DateTimeOffset Read(
        ref Utf8JsonReader reader,
        Type typeToConvert,
        JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
                !match.Success
                || !long.TryParse(match.Groups[1].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out long unixTime)
                || !int.TryParse(match.Groups[3].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out int hours)
                || !int.TryParse(match.Groups[4].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out int minutes))
        {
            throw new System.Text.Json.JsonException();
        }

        int sign = match.Groups[2].Value[0] == '+' ? 1 : -1;
        TimeSpan utcOffset = new(hours * sign, minutes * sign, 0);

        return s_epoch.AddMilliseconds(unixTime).ToOffset(utcOffset);
    }

    public override void Write(
        Utf8JsonWriter writer,
        DateTimeOffset value,
        JsonSerializerOptions options)
    {
        long unixTime = value.ToUnixTimeMilliseconds();

        TimeSpan utcOffset = value.Offset;

        string formatted = string.Create(
            CultureInfo.InvariantCulture,
            $"/Date({unixTime}{(utcOffset >= TimeSpan.Zero ? "+" : "-")}
{utcOffset:hhmm})/");
```

```
        writer.WriteStringValue(formatted);
    }
}
```

C#

```
sealed class UnixEpochDateTimeConverter :
System.Text.Json.Serialization.JsonConverter<DateTime>
{
    static readonly DateTime s_epoch = new(1970, 1, 1, 0, 0, 0);
    static readonly Regex s_regex = new(
        "^/Date\\(([+-]*\\d+)\\)/$",
        RegexOptions.CultureInvariant);

    public override DateTime Read(
        ref Utf8JsonReader reader,
        Type typeToConvert,
        JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out long unixTime))
        {
            throw new System.Text.Json.JsonException();
        }

        return s_epoch.AddMilliseconds(unixTime);
    }

    public override void Write(
        Utf8JsonWriter writer,
        DateTime value,
        JsonSerializerOptions options)
    {
        long unixTime = (value - s_epoch).Ticks / TimeSpan.TicksPerMillisecond;

        string formatted = string.Create(CultureInfo.InvariantCulture,
$"/Date({unixTime})/");
        writer.WriteStringValue(formatted);
    }
}
```

Utf8JsonWriter을(를) 사용하는 경우

DateTime을(를) 사용하여 사용자 지정 DateTimeOffset 또는 Utf8JsonWriter 텍스트 표현을 작성하려면 사용자 지정 표현을 String, `ReadOnlySpan<Byte>`, `ReadOnlySpan<Char>` 또는

JsonEncodedText 서식을 지정한 다음 해당 Utf8JsonWriter.WriteStringValue 또는 Utf8JsonWriter.WriteString 메서드에 전달할 수 있습니다.

다음 예제에서는 DateTime을(를) 사용하여 사용자 지정 ToString(String, IFormatProvider) 형식을 만든 다음 WriteStringValue(String) 메서드로 작성하는 방법을 보여줍니다.

C#

```
using System.Globalization;
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        var options = new JsonWriterOptions
        {
            Indented = true
        };

        using (var stream = new MemoryStream())
        {
            using (var writer = new Utf8JsonWriter(stream, options))
            {
                string dateStr = DateTime.UtcNow.ToString("F",
CultureInfo.InvariantCulture);

                writer.WriteStartObject();
                writer.WriteString("date", dateStr);
                writer.WriteNumber("temp", 42);
                writer.WriteEndObject();
            }

            string json = Encoding.UTF8.GetString(stream.ToArray());
            Console.WriteLine(json);
        }
    }
}

// The example displays output similar to the following:
// {
//     "date": "Tuesday, 27 August 2019 19:21:44",
//     "temp": 42
// }
```

Utf8JsonReader을(를) 사용하는 경우

사용자 지정 DateTime 또는 DateTimeOffset 텍스트 표현을 Utf8JsonReader과 함께 읽으려면, String 메서드를 사용하여 현재 JSON 토큰을 GetString()로 가져온 다음, 사용자 지정 논리를 활

용하여 값을 구문 분석할 수 있습니다.

다음 예제에서는 DateTimeOffset 메서드를 사용하여 사용자 지정 GetString() 텍스트 표현을 검색한 다음 ParseExact(String, String, IFormatProvider)을(를) 사용하여 구문 분석하는 방법을 보여 줍니다.

C#

```
using System.Globalization;
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""Friday, 26 July 2019
00:00:00""");

        var json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                string value = json.GetString();
                DateTimeOffset dto = DateTimeOffset.ParseExact(value, "F",
CultureInfo.InvariantCulture);
                Console.WriteLine(dto);
            }
        }
    }
}

// The example displays output similar to the following:
// 7/26/2019 12:00:00 AM -04:00
```

System.Text.Json의 ISO 8601-1:2019 확장 프로필

날짜 및 시간 구성 요소

System.Text.Json에서 구현된 확장 ISO 8601-1:2019 프로필은 날짜 및 시간 표현에 대해 다음 구성 요소를 정의합니다. 이러한 구성 요소는 DateTime 및 DateTimeOffset 표현을 구문 분석하고 서식을 지정할 때 지원되는 다양한 수준의 세분성을 정의하는 데 사용됩니다.

테이블 확장

구성 요소	서식	설명
연도	"yyyy"	0001-9999
월	"MM"	01-12
요일	"dd"	월/연도 기준 01-28, 01-29, 01-30, 01-31
한 시간	"HH"	00-23
분	"mm"	00-59
둘째	"ss"	00-59
두 번째 분수	"FFFFFFF"	최소 한 자리, 최대 16자리 숫자
시간 오프셋	"K"	"Z" 또는 "('+'/'-')HH':'mm"입니다.
파트 타임	"HH':'mm':'ss[FFFFFFF]"	UTC 오프셋 정보가 없는 시간입니다.
전체 날짜	"yyyy'-'MM'-'dd"	달력 날짜.
풀 타임	"'부분 시간'K"	UTC 시간 또는 현지 시간과 UTC 간의 시간 오프셋을 포함한 현지 시간입니다.
날짜 및 시간	"'전체 날짜''T''전체 시간'"	달력 날짜 및 시간(예: 2019-07-26T16:59:57-05:00).

구문 분석 지원

구문 분석을 위해 다음과 같은 세분성 수준이 정의됩니다.

1. '전체 날짜'
   a. "yyyy'-'MM'-'dd"
2. "'전체 날짜''T''시'':''분'"
   a. "yyyy'-'MM'-'dd'T'HH':'mm"
3. "'전체 날짜''T''부분 시간'"
   a. "yyyy'-'MM'-'dd'T'HH':'mm':'ss"(정렬 가능("s") 형식 지정자)
   b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.' FFFFFFF"
4. "'전체 날짜''T''시간 시'':''분''시간 오프셋'"
   a. "yyyy'-'MM'-'dd'T'HH':'mmZ"
   b. "yyyy'-'MM'-'dd'T'HH':'mm('+'/'-')HH':'mm"
5. 날짜와 시간

a. yyyy'-'MM'-'dd'T'HH':'mm':'ssZ
b. yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'FFFFFFFZ
c. "yyyy'-'MM'-'dd'T'HH':'mm':'ss('+'/'-')HH':'mm"
d. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.' FFFFFFF('+'/'-')HH':'mm"

이 수준의 세분성은 날짜 및 시간 정보를 교환하는 데 사용되는 ISO 8601의 널리 채택된 프로필인 RFC 3339를 준수합니다. 그러나 `System.Text.Json` 구현에는 몇 가지 제한 사항이 있습니다.

- RFC 3339는 소수 자릿수 초 숫자의 최대 수를 지정하지 않지만 소수 자릿수 초 섹션이 있는 경우 적어도 한 자리가 마침표 뒤에 오도록 지정합니다. `System.Text.Json` 구현은 최대 16자리(다른 프로그래밍 언어 및 프레임워크와의 interop 지원)를 허용하지만 처음 7자리만 구문 분석합니다. JsonException 및 `DateTime` 인스턴스를 읽을 때 소수 자릿수가 16자리보다 많은 경우 `DateTimeOffset`이(가) throw됩니다.
- RFC 3339는 "T" 및 "Z" 문자를 각각 "t" 또는 "z"로 허용하지만 애플리케이션은 대문자 변형으로만 지원을 제한할 수 있습니다. `System.Text.Json`을(를) 구현하려면 "T" 및 "Z"여야 합니다. JsonException 및 `DateTime` 인스턴스를 읽을 때 입력 페이로드에 "t" 또는 "z"가 포함된 경우 `DateTimeOffset`이(가) throw됩니다.
- RFC 3339는 날짜 및 시간 섹션을 "T"로 구분하도록 지정하지만 애플리케이션은 대신 공백(" ")으로 구분할 수 있습니다. `System.Text.Json` 날짜 및 시간 섹션을 "T"로 구분해야 합니다. 입력 페이로드에 JsonException 및 `DateTime` 인스턴스를 읽을 때 공백(" ")이 포함된 경우 `DateTimeOffset`이(가) throw됩니다.

초 동안 소수 자릿수가 있는 경우 적어도 한 자리가 있어야 합니다. `2019-07-26T00:00:00.`은(는) 허용되지 않습니다. 최대 16개의 소수 자릿수가 허용되지만, 시스템에서는 처음 7개만 구문 분석됩니다. 그 이상의 모든 것은 0으로 간주됩니다. 예를 들어 `2019-07-26T00:00:00.1234567890`이(가) `2019-07-26T00:00:00.1234567`인 것처럼 구문 분석됩니다. 이 접근법은 이 해상도로 제한되는 DateTime 구현과의 호환성을 유지합니다.

윤초는 지원되지 않습니다.

서식 지정 지원

다음 세분성 수준은 서식 지정에 대해 정의됩니다.

1. "'전체 날짜''T''부분 시간'"

   a. "yyyy'-'MM'-'dd'T'HH':'mm':'ss"(정렬 가능("s") 형식 지정자)

      소수 초와 오프셋 정보 없이 DateTime를 형식화하는 데 사용됩니다.

   b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.' FFFFFFF"

DateTime를 소수 자릿수 초로 서식화하되, 오프셋 정보는 포함하지 않는 데 사용됩니다.

2. 날짜와 시간

   a. yyyy'-'MM'-'dd'T'HH':'mm':'ssZ

      소수 자릿수 초 없이 UTC 오프셋을 사용하여 DateTime 서식을 지정하는 데 사용됩니다.

   b. yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'FFFFFFFZ

      소수 자릿수 초 및 UTC 오프셋을 사용하여 DateTime 서식을 지정하는 데 사용됩니다.

   c. "yyyy'-'MM'-'dd'T'HH':'mm':'ss('+'/'-')HH':'mm"

      소수 자릿수 초가 아닌 로컬 오프셋을 사용하여 DateTime 또는 DateTimeOffset 서식을 지정하는 데 사용됩니다.

   d. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.' FFFFFFF('+'/'-')HH':'mm"

      소수 자릿수 초 및 로컬 오프셋을 사용하여 DateTime 또는 DateTimeOffset 서식을 지정하는 데 사용됩니다.

   이 세분성 수준은 RFC 3339 를 준수합니다.

왕복 형식으로 표현된 DateTime 또는 DateTimeOffset 인스턴스에 소수 자릿수 초에 후행 0이 있는 경우, JsonSerializer 및 Utf8JsonWriter는 후행 0을 생략한 상태로 인스턴스를 형식화합니다. 예를 들어 DateTime의 왕복 형식 표현이 인 `2019-04-24T14:50:17.1010000Z` 인스턴스는 `2019-04-24T14:50:17.101Z` 및 JsonSerializer에 의해 Utf8JsonWriter으로 형식이 지정됩니다.

왕복 형식이 DateTime 또는 DateTimeOffset 인스턴스의 표현에서 소수 초 부분이 모두 0인 경우, JsonSerializer 및 Utf8JsonWriter는 소수 초 없이 인스턴스를 형식화합니다. 예를 들어 DateTime의 왕복 형식 표현이 인 `2019-04-24T14:50:17.0000000+02:00` 인스턴스는 `2019-04-24T14:50:17+02:00` 및 JsonSerializer에 의해 Utf8JsonWriter으로 형식이 지정됩니다.

초 자릿수의 0을 절단하면 왕복 과정에서 정보 보존을 위해 필요한 최소한의 출력이 생성됩니다.

최대 7개의 소수 자릿수 초 숫자가 기록됩니다. 이 최대값은 이 해상도로 제한되는 DateTime 구현과 일치합니다.

Last updated on 2026. 02. 24.

표준 시간대 개요

2025. 06. 17.

이 클래스는 TimeZoneInfo 표준 시간대 인식 애플리케이션의 생성을 간소화합니다. 이 클래스는 TimeZone 현지 표준 시간대 및 UTC(협정 세계시) 작업을 지원합니다. 클래스는 TimeZoneInfo 이러한 영역과 레지스트리에 미리 정의된 정보에 대한 모든 표준 시간대를 모두 지원합니다. 시스템에 정보가 없는 사용자 지정 표준 시간대를 정의하는 데 사용할 TimeZoneInfo 수도 있습니다.

표준 시간대 필수 요소

표준 시간대는 동일한 시간이 사용되는 지리적 지역입니다. 일반적으로 항상 그렇지는 않지만 인접한 표준 시간대는 1시간 떨어져 있습니다. 세계 표준 시간대의 시간은 UTC(협정 세계시)의 오프셋으로 표현될 수 있습니다.

전 세계의 많은 표준 시간대는 일광 절약 시간을 지원합니다. 일광 절약 시간제는 봄이나 초여름에 시간을 1시간 앞당기고 늦은 여름이나 가을에 정상(또는 표준) 시간으로 돌아가서 일광 시간을 최대화하려고 합니다. 표준시에서 이러한 변경 내용을 조정 규칙이라고 합니다.

특정 표준 시간대의 일광 절약 시간제 전환은 고정 또는 부동 조정 규칙에 의해 정의될 수 있습니다. 고정 조정 규칙은 일광 절약 시간제로의 전환이 매년 발생하는 특정 날짜를 설정합니다. 예를 들어 매년 10월 25일에 발생하는 일광 절약 시간에서 표준 시간으로의 전환은 고정 조정 규칙을 따릅니다. 훨씬 더 일반적인 부동 조정 규칙은 일광 절약 시간제로 전환하기 위해 특정 월의 특정 요일을 설정합니다. 예를 들어 3월 3일 일요일에 발생하는 표준 시간에서 일광 절약 시간으로의 전환은 부동 조정 규칙을 따릅니다.

조정 규칙을 지원하는 표준 시간대의 경우 일광 절약 시간제로 전환하면 잘못된 시간과 모호한 시간이라는 두 종류의 비정상적인 시간이 생성됩니다. 잘못된 시간은 표준 시간에서 일광 절약 시간으로 전환하여 생성된 존재하지 않는 시간입니다. 예를 들어 이 전환이 오전 2시에 특정 일에 발생하고 시간이 오전 3:00으로 변경되는 경우 오전 2시에서 오전 2시 59분 59분 사이의 각 시간 간격은 유효하지 않습니다. 모호한 시간은 하나의 표준 시간대에서 서로 다른 두 번에 대응할 수 있는 시간입니다. 일광 절약 시간제에서 표준 시간으로 전환하여 생성됩니다. 예를 들어 이 전환이 오전 2시에 특정 일에 발생하고 시간이 오전 1:00으로 변경되는 경우 오전 1시에서 오전 1시 59분 59분 사이의 각 시간 간격은 표준 시간 또는 일광 절약 시간으로 해석될 수 있습니다.

표준 시간대 용어

다음 표에서는 표준 시간대를 사용하고 표준 시간대 인식 애플리케이션을 개발할 때 일반적으로 사용되는 용어를 정의합니다.

테이블 확장

기간	정의
조정 규칙	표준 시간에서 일광 절약 시간 및 일광 절약 시간에서 표준 시간으로의 전환이 발생하는 시기를 정의하는 규칙입니다. 각 조정 규칙에는 규칙이 적용된 시점(예: 조정 규칙이 1986년 1월 1일부터 2006년 12월 31일까지), 델타(조정 규칙 적용의 결과로 표준 시간이 변경되는 시간) 및 조정 기간 동안 전환이 발생하는 특정 날짜 및 시간에 대한 정보를 정의하는 시작 및 종료 날짜가 있습니다. 전환은 고정 규칙 또는 부동 규칙을 따를 수 있습니다.
모호한 시간	단일 표준 시간대에서 서로 다른 두 시간에 매핑할 수 있는 시간입니다. 표준 시간대의 일광 절약 시간에서 표준 시간으로 전환하는 동안과 같이 시계 시간이 시간을 거슬러 조정될 때 발생합니다. 예를 들어 이 전환이 오전 2시에 특정 일에 발생하고 시간이 오전 1:00으로 변경되는 경우 오전 1시에서 오전 1시 59분 59분 사이의 각 시간 간격은 표준 시간 또는 일광 절약 시간으로 해석될 수 있습니다.
고정 규칙	일광 절약 시간제로 전환할 특정 날짜를 설정하는 조정 규칙입니다. 예를 들어 매년 10월 25일에 발생하는 일광 절약 시간에서 표준 시간으로의 전환은 고정 조정 규칙을 따릅니다.
유동 규칙	일광 절약 시간제로 전환할 특정 월의 특정 요일을 설정하는 조정 규칙입니다. 예를 들어 3월 3일 일요일에 발생하는 표준 시간에서 일광 절약 시간으로의 전환은 부동 조정 규칙을 따릅니다.
잘못된 형식	표준 시간에서 일광 절약 시간으로의 전환의 아티팩트인 존재하지 않는 시간입니다. 표준 시간대 표준시에서 일광 절약 시간으로 전환하는 동안과 같이 시계 시간이 정방향으로 조정될 때 발생합니다. 예를 들어 이 전환이 오전 2시에 특정 일에 발생하고 시간이 오전 3:00으로 변경되는 경우 오전 2시에서 오전 2시 59분 59분 사이의 각 시간 간격은 유효하지 않습니다.
전환 시간	특정 시간대에서 서머타임에서 표준 시간으로 또는 그 반대로의 변경과 같은 특정 시간 변경에 대한 정보입니다.

표준 시간대 및 TimeZoneInfo 클래스

.NET에서 TimeZoneInfo 객체는 시간대를 나타냅니다. TimeZoneInfo 클래스에는 GetAdjustmentRules 메서드가 있으며, 이 메서드는 TimeZoneInfo.AdjustmentRule 개체 배열을 반환합니다. 이 배열의 각 요소는 특정 기간 동안 일광 절약 시간제로의 전환에 대한 정보를 제공합니다. (일광 절약 시간을 지원하지 않는 표준 시간대의 경우 메서드는 빈 배열을 반환합니다.) 각 TimeZoneInfo.AdjustmentRule 개체에는 DaylightTransitionStartDaylightTransitionEnd 일광 절약 시간제와 전환의 특정 날짜 및 시간을 정의하는 속성이 있습니다. 이 속성은 IsFixedDateRule 해당 전환이 고정되어 있는지 또는 부동인지를 나타냅니다.

.NET은 Windows 운영 체제에서 제공하고 레지스트리에 저장된 표준 시간대 정보를 사용합니다. 지구의 표준 시간대 수로 인해 모든 기존 표준 시간대가 레지스트리에 표시되지 않습니다. 또한 레지스트리는 동적 구조이므로 미리 정의된 표준 시간대를 추가하거나 제거할 수 있습니다. 마지막으로 레지스트리에 기록 표준 시간대 데이터가 반드시 포함되어 있지는 않습니다. 예를 들어 Windows XP에서 레지스트리에는 단일 표준 시간대 조정 집합에 대한 데이터만 포함됩니다. Windows Vista는 동적 표준 시간대 데이터를 지원합니다. 즉, 단일 표준 시간대에는 특정 연도 간격에 적용되는 여러 조정 규칙이 있을 수 있습니다. 그러나 Windows Vista 레지스트리에 정의되고 일광 절약 시간을 지원하는 대부분의 표준 시간대에는 미리 정의된 조정 규칙이 하나 또는 두 개뿐입니다.

레지스트리에 대한 클래스의 TimeZoneInfo 의존성은 표준 시간대 인식 애플리케이션이 특정 표준 시간대가 레지스트리에 정의되어 있는지 확신할 수 없음을 의미합니다. 따라서 특정 표준 시간대(현지 표준 시간대 또는 UTC를 나타내는 표준 시간대 제외)를 인스턴스화하려는 시도는 예외 처리를 사용해야 합니다. 또한 필요한 TimeZoneInfo 개체를 레지스트리에서 인스턴스화할 수 없는 경우 애플리케이션을 계속하도록 하는 몇 가지 방법을 제공해야 합니다.

필요한 표준 시간대 TimeZoneInfo 의 부재를 처리하기 위해 클래스에는 레지스트리에서 찾을 수 없는 사용자 지정 표준 시간대를 만드는 데 사용할 수 있는 메서드가 포함 CreateCustomTimeZone 됩니다. 사용자 지정 표준 시간대를 만드는 방법에 대한 자세한 내용은 방법: 조정 규칙 없이 표준 시간대 만들기 및 방법: 조정 규칙을 사용하여 표준 시간대 만들기를 참조하세요. 또한 이 메서드를 사용하여 ToSerializedString 새로 만든 표준 시간대를 문자열로 변환하고 데이터 저장소(예: 데이터베이스, 텍스트 파일, 레지스트리 또는 애플리케이션 리소스)에 저장할 수 있습니다. 그런 다음 메서드를 FromSerializedString 사용하여 이 문자열을 개체로 다시 변환할 수 있습니다 TimeZoneInfo . 자세한 내용은 방법: 포함된 리소스에 표준 시간대 저장 및 방법: 포함된 리소스에서 표준 시간대 복원을 참조하세요.

각 표준 시간대는 UTC의 기본 오프셋과 기존 조정 규칙을 반영하는 UTC의 오프셋으로 특징지어지므로 한 표준 시간대의 시간을 다른 표준 시간대의 시간으로 쉽게 변환할 수 있습니다. 이 목적을 위해 객체에는 여러 변환 메서드가 포함되어 있으며, TimeZoneInfo을(를) 포함합니다.

- ConvertTimeFromUtc는 UTC를 지정된 표준 시간대의 시간으로 변환합니다.
- ConvertTimeToUtc- 지정된 표준 시간대의 시간을 UTC로 변환합니다.
- ConvertTime- 지정된 표준 시간대의 시간을 지정된 다른 표준 시간대의 시간으로 변환합니다.
- ConvertTimeBySystemTimeZoneId- 지정된 표준 시간대의 시간을 지정된 다른 표준 시간대의 TimeZoneInfo 시간으로 변환하는 매개 변수로 표준 시간대 식별자(개체 대신)를 사용합니다.

표준 시간대 간 시간 변환에 대한 자세한 내용은 표준 시간대 간 시간 변환을 참조하세요.

참고하십시오

- 날짜, 시간 및 표준 시간대

방법: 날짜 및 시간 산술 연산에서 표준 시간대 사용

아티클 • 2024. 06. 08.

일반적으로 DateTime 또는 DateTimeOffset 값을 사용하여 날짜 및 시간 산술 연산을 수행하는 경우 결과는 표준 시간대 조정 규칙을 반영하지 않습니다. 날짜 및 시간 값의 표준 시간대를 명확하게 식별할 수 있는 경우에도 마찬가지입니다(예: Kind 속성이 Local로 설정된 경우). 이 항목에서는 특정 표준 시간대에 속하는 날짜 및 시간 값에 대한 산술 연산 작업을 수행하는 방법을 보여 줍니다. 산술 연산 작업의 결과는 표준 시간대의 조정 규칙을 반영합니다.

날짜 및 시간 산술 연산에 조정 규칙을 적용하려면

1. 날짜 및 시간 값을 속해 있는 표준 시간대와 밀접하게 연결하는 몇 가지 메서드를 구현합니다. 예를 들어, 날짜 및 시간 값 및 해당 표준 시간대를 모두 포함하는 구조체를 선언합니다. 다음 예에서는 이 접근 방식을 사용하여 DateTime 값을 해당 표준 시간대와 연결합니다.

   C#

   ```
   // Define a structure for DateTime values for internal use only
   internal struct TimeWithTimeZone
   {
      TimeZoneInfo TimeZone;
      DateTime Time;
   }
   ```

2. ConvertTimeToUtc 메서드 또는 ConvertTime 메서드를 호출하여 시간을 UTC(협정 세계시)로 변환합니다.

3. UTC 시간에 대한 산술 연산을 수행합니다.

4. TimeZoneInfo.ConvertTime(DateTime, TimeZoneInfo) 메서드를 호출하여 시간을 UTC에서 원래 시간의 관련 표준 시간대로 변환합니다.

예시

다음 예제에서는 중부 표준시 2008년 3월 9일 오전 1시 30분에 2시간 30분을 추가합니다. 30분 뒤인 2008년 3월 9일 오전 2시에 표준 시간대가 일광 절약 시간으로 전환됩니다. 이 예제에서는 이전 섹션의 4단계를 수행하므로 결과 시간이 2008년 3월 9일 오전 5시로 올바르게 보고됩니다.

C#

```
using System;

public struct TimeZoneTime
{
   public TimeZoneInfo TimeZone;
   public DateTime Time;

   public TimeZoneTime(TimeZoneInfo tz, DateTime time)
   {
      if (tz == null)
         throw new ArgumentNullException("The time zone cannot be a null
reference.");

      this.TimeZone = tz;
      this.Time = time;
   }

   public TimeZoneTime AddTime(TimeSpan interval)
   {
      // Convert time to UTC
      DateTime utcTime = TimeZoneInfo.ConvertTimeToUtc(this.Time,
this.TimeZone);
      // Add time interval to time
      utcTime = utcTime.Add(interval);
      // Convert time back to time in time zone
      return new TimeZoneTime(this.TimeZone,
TimeZoneInfo.ConvertTime(utcTime,
                               TimeZoneInfo.Utc, this.TimeZone));
   }
}

public class TimeArithmetic
{
   public const string tzName = "Central Standard Time";

   public static void Main()
   {
      try
      {
         TimeZoneTime cstTime1, cstTime2;

         TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
         DateTime time1 = new DateTime(2008, 3, 9, 1, 30, 0);
         TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

         cstTime1 = new TimeZoneTime(cst, time1);
         cstTime2 = cstTime1.AddTime(twoAndAHalfHours);
         Console.WriteLine("{0} + {1} hours = {2}", cstTime1.Time,

twoAndAHalfHours.ToString(),
                                                    cstTime2.Time);
      }
```

```
      catch
      {
         Console.WriteLine("Unable to find {0}.", tzName);
      }
   }
}
```

DateTime 및 DateTimeOffset 값은 속해 있을 수 있는 표준 시간대에서 분리됩니다. 자동으로 표준 시간대의 조정 규칙을 적용하는 방식으로 날짜 및 시간 산술 연산을 수행하려면 모든 날짜 및 시간 값이 속해 있는 표준 시간대를 즉시 식별할 수 있어야 합니다. 즉, 날짜 및 시간과 관련된 표준 시간대는 밀접하게 결합되어야 합니다. 여러 가지 방법을 통해 다음을 포함하는 작업을 수행할 수 있습니다.

- 애플리케이션에서 사용되는 모든 시간이 특정 표준 시간대에 속한다고 가정합니다. 일부 경우에는 적합하지만 이 방식은 제한된 유연성과 이식성을 제공합니다.
- 모두 형식의 필드로 포함하여 관련된 표준 시간대와 날짜 및 시간을 밀접하게 결합하는 형식을 정의합니다. 이 방법은 코드 예제에 사용되며 여기서 두 개의 구성원 필드에 있는 날짜 및 시간과 표준 시간대를 저장하는 구조체를 정의합니다.

이 예에서는 표준 시간대 조정 규칙이 결과에 적용되도록 DateTime 값에 대한 산술 연산을 수행하는 방법을 보여 줍니다. 그러나 DateTimeOffset 값도 마찬가지로 쉽게 사용할 수 있습니다. 다음 예에서는 원래 예의 코드가 DateTime 값 대신 DateTimeOffset을 사용하도록 조정하는 방법을 보여 줍니다.

C#

```
using System;

public struct TimeZoneTime
{
   public TimeZoneInfo TimeZone;
   public DateTimeOffset Time;

   public TimeZoneTime(TimeZoneInfo tz, DateTimeOffset time)
   {
      if (tz == null)
         throw new ArgumentNullException("The time zone cannot be a null
reference.");

      this.TimeZone = tz;
      this.Time = time;
   }

   public TimeZoneTime AddTime(TimeSpan interval)
   {
      // Convert time to UTC
      DateTimeOffset utcTime = TimeZoneInfo.ConvertTime(this.Time,
TimeZoneInfo.Utc);
```

```
      // Add time interval to time
      utcTime = utcTime.Add(interval);
      // Convert time back to time in time zone
      return new TimeZoneTime(this.TimeZone,
TimeZoneInfo.ConvertTime(utcTime, this.TimeZone));
   }
}

public class TimeArithmetic
{
   public const string tzName = "Central Standard Time";

   public static void Main()
   {
      try
      {
         TimeZoneTime cstTime1, cstTime2;

         TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
         DateTime time1 = new DateTime(2008, 3, 9, 1, 30, 0);
         TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

         cstTime1 = new TimeZoneTime(cst,
                        new DateTimeOffset(time1, cst.GetUtcOffset(time1)));
         cstTime2 = cstTime1.AddTime(twoAndAHalfHours);
         Console.WriteLine("{0} + {1} hours = {2}", cstTime1.Time,

twoAndAHalfHours.ToString(),
                                                    cstTime2.Time);
      }
      catch
      {
         Console.WriteLine("Unable to find {0}.", tzName);
      }
   }
}
```

이 추가 작업이 DateTimeOffset 값을 UTC로 변환하지 않고 간단히 수행하는 경우 결과는 올바른 시점을 반영하지만 해당 오프셋은 해당 시간에 지정된 표준 시간대의 시점을 반영하지 않습니다.

참고 항목

- 날짜, 시간 및 표준 시간대
- 날짜 및 시간에 대한 산술 연산 수행

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

DateTime과 DateTimeOffset 간 변환

구조체가 DateTimeOffset 구조체보다 DateTime 더 높은 수준의 표준 시간대 인식을 제공하지만 매개 DateTime 변수는 메서드 호출에서 더 일반적으로 사용됩니다. 이 접근 방식 때문에 DateTimeOffset 값을 DateTime 값으로 변환하는 능력과 DateTime 값을 DateTimeOffset 값으로 변환하는 능력이 중요합니다. 이 문서에서는 가능한 한 많은 표준 시간대 정보를 유지하는 방식으로 이러한 변환을 수행하는 방법을 보여 줍니다.

> ⓘ **참고 항목**
>
> 두 가지 DateTime 및 DateTimeOffset 형식은 표준 시간대의 시간을 나타낼 때 몇 가지 제한 사항이 있습니다. 해당 Kind 속성을 DateTime 사용하면 UTC(협정 세계시)와 시스템의 현지 표준 시간대만 반영할 수 있습니다. DateTimeOffset 는 UTC의 시간 오프셋을 반영하지만 해당 오프셋이 속한 실제 표준 시간대를 반영하지는 않습니다. 표준 시간대에 대한 시간 값 및 지원에 대한 자세한 내용은 **DateTime, DateTimeOffset, TimeSpan 및 TimeZoneInfo 중에서 선택** 항목을 참조하세요.

DateTime에서 DateTimeOffset으로 변환

구조체는 DateTimeOffsetDateTime에서 DateTimeOffset로의 변환을 수행하기 위한 두 가지 동등한 방법을 제공하며, 이는 대부분의 변환에 유용합니다.

- 생성자 DateTimeOffset는 DateTimeOffset 값을 기반으로 새로운 DateTime 객체를 만듭니다.
- 값 DateTime을 개체에 DateTimeOffset 할당할 수 있는 암시적 변환 연산자입니다.

UTC 및 로컬 DateTime 값 Offset 의 경우 결과 DateTimeOffset 값의 속성은 UTC 또는 현지 표준 시간대 오프셋을 정확하게 반영합니다. 예를 들어 다음 코드는 UTC 시간을 해당 DateTimeOffset 값으로 변환합니다.

C#

```
DateTime utcTime1 = new DateTime(2008, 6, 19, 7, 0, 0);
utcTime1 = DateTime.SpecifyKind(utcTime1, DateTimeKind.Utc);
DateTimeOffset utcTime2 = utcTime1;
Console.WriteLine($"Converted {utcTime1} {utcTime1.Kind} to a DateTimeOffset value
of {utcTime2}");
// This example displays the following output to the console:
//    Converted 6/19/2008 7:00:00 AM Utc to a DateTimeOffset value of 6/19/2008
7:00:00 AM +00:00
```

이 경우 변수의 `utcTime2` 오프셋은 00:00입니다. 마찬가지로 다음 코드는 현지 시간을 해당 DateTimeOffset 값으로 변환합니다. 변환은 미국 태평양 표준 시간대에서 실행됩니다.

C#

```
DateTime localTime1 = new DateTime(2008, 6, 19, 7, 0, 0);
localTime1 = DateTime.SpecifyKind(localTime1, DateTimeKind.Local);
DateTimeOffset localTime2 = localTime1;
Console.WriteLine($"Converted {localTime1} {localTime1.Kind} to a DateTimeOffset
value of {localTime2}");
// This example displays the following output to the console:
//    Converted 6/19/2008 7:00:00 AM Local to a DateTimeOffset value of 6/19/2008
7:00:00 AM -07:00
```

속성이 DateTime인 Kind 값의 경우, 이 두 변환 메서드는 현지 표준 시간대의 오프셋을 가진 DateTimeKind.Unspecified 값을 생성합니다. 변환은 미국 태평양 표준 시간대에서 실행되는 다음 예제에 나와 있습니다.

C#

```
DateTime time1 = new DateTime(2008, 6, 19, 7, 0, 0);  // Kind is
DateTimeKind.Unspecified
DateTimeOffset time2 = time1;
Console.WriteLine($"Converted {time1} {time1.Kind} to a DateTimeOffset value of
{time2}");
// This example displays the following output to the console:
//    Converted 6/19/2008 7:00:00 AM Unspecified to a DateTimeOffset value of
6/19/2008 7:00:00 AM -07:00
```

값이 DateTime 현지 표준 시간대 또는 UTC 이외의 날짜와 시간을 반영하는 경우 오버로드된 DateTimeOffset 생성자를 호출하여 해당 표준 시간대 정보를 값으로 변환 DateTimeOffset 하고 해당 표준 시간대 정보를 유지할 수 있습니다. 예를 들어 다음 예제에서는 중앙 표준시를 DateTimeOffset 반영하는 개체를 인스턴스화합니다.

C#

```
DateTime time1 = new DateTime(2008, 6, 19, 7, 0, 0);     // Kind is
DateTimeKind.Unspecified
try
{
   DateTimeOffset time2 = new DateTimeOffset(time1,
                  TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(time1));
   Console.WriteLine($"Converted {time1} {time1.Kind} to a DateTime value of
{time2}");
}
// Handle exception if time zone is not defined in registry
catch (TimeZoneNotFoundException)
{
```

```
   Console.WriteLine("Unable to identify target time zone for conversion.");
}
// This example displays the following output to the console:
//    Converted 6/19/2008 7:00:00 AM Unspecified to a DateTime value of 6/19/2008
7:00:00 AM -05:00
```

이 생성자 오버로드에 대한 두 번째 매개 변수는 UTC의 시간 오프셋을 나타내는 개체입니다 TimeSpan . 해당 시간대의 메서드 TimeZoneInfo.GetUtcOffset(DateTime)를 호출하여 시간을 검색합니다. 메서드의 단일 매개 변수는 변환할 날짜 및 시간을 나타내는 값입니다 DateTime . 표준 시간대에서 일광 절약 시간을 지원하는 경우 이 매개 변수를 사용하면 메서드가 특정 날짜 및 시간에 대한 적절한 오프셋을 확인할 수 있습니다.

DateTimeOffset에서 DateTime으로 변환

이 DateTime 속성은 변환을 수행하는 DateTimeOffsetDateTime 데 가장 일반적으로 사용됩니다. 그러나 속성이 DateTime인 Kind 값을 반환하는데, 이는 다음 예제에서 볼 수 있듯이 Unspecified입니다.

C#

```
DateTime baseTime = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset sourceTime;
DateTime targetTime;

// Convert UTC to DateTime value
sourceTime = new DateTimeOffset(baseTime, TimeSpan.Zero);
targetTime = sourceTime.DateTime;
Console.WriteLine($"{sourceTime} converts to {targetTime} {targetTime.Kind}");

// Convert local time to DateTime value
sourceTime = new DateTimeOffset(baseTime,
                                TimeZoneInfo.Local.GetUtcOffset(baseTime));
targetTime = sourceTime.DateTime;
Console.WriteLine($"{sourceTime} converts to {targetTime} {targetTime.Kind}");

// Convert Central Standard Time to a DateTime value
try
{
   TimeSpan offset = TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(baseTime);
   sourceTime = new DateTimeOffset(baseTime, offset);
   targetTime = sourceTime.DateTime;
   Console.WriteLine($"{sourceTime} converts to {targetTime} {targetTime.Kind}");
}
catch (TimeZoneNotFoundException)
{
   Console.WriteLine("Unable to create DateTimeOffset based on U.S. Central
Standard Time.");
}
```

```
// This example displays the following output to the console:
//    6/19/2008 7:00:00 AM +00:00 converts to 6/19/2008 7:00:00 AM Unspecified
//    6/19/2008 7:00:00 AM -07:00 converts to 6/19/2008 7:00:00 AM Unspecified
//    6/19/2008 7:00:00 AM -05:00 converts to 6/19/2008 7:00:00 AM Unspecified
```

앞의 예제는 DateTimeOffset 속성이 사용될 때 변환 중에 DateTime 값과 UTC의 관계에 대한 정보가 손실됨을 보여 줍니다. 이 동작은 DateTimeOffset에 해당하는 값이 UTC 시간 또는 시스템의 현지 시간에 대응되기 때문에, 구조체가 DateTime 속성에서 오직 이 두 표준 시간대만 반영합니다.

값을 DateTimeOffset에서 DateTime 값으로 변환할 때 가능한 한 많은 표준 시간대 정보를 유지하려면 DateTimeOffset.UtcDateTime 및 DateTimeOffset.LocalDateTime 속성을 사용할 수 있습니다.

UTC 시간 변환

변환된 DateTime 값이 UTC 시간임을 나타내기 위해 DateTimeOffset.UtcDateTime 속성의 값을 검색하세요. 다음과 같은 두 가지 방법으로 속성과 DateTime 다릅니다.

- 속성이 DateTime인 Kind 값을 반환합니다 Utc.
- 속성 값이 Offset 같지 TimeSpan.Zero않으면 시간을 UTC로 변환합니다.

> ⓘ **참고 항목**
>
> 애플리케이션에서 변환된 DateTime 값이 단일 시점을 명확하게 식별해야 하는 경우, DateTimeOffset.UtcDateTime 속성을 사용하여 모든 DateTimeOffset를 DateTime로 변환하는 것을 고려하십시오.

다음 코드에서는 UtcDateTime 속성을 사용하여 오프셋이 DateTimeOffset인 TimeSpan.Zero 값을 DateTime 값으로 변환합니다.

C#

```
DateTimeOffset utcTime1 = new DateTimeOffset(2008, 6, 19, 7, 0, 0, TimeSpan.Zero);
DateTime utcTime2 = utcTime1.UtcDateTime;
Console.WriteLine($"{utcTime1} converted to {utcTime2} {utcTime2.Kind}");
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM +00:00 converted to 6/19/2008 7:00:00 AM Utc
```

다음 코드에서는 UtcDateTime 속성을 사용하여 DateTimeOffset 값에 대해 표준 시간대 변환과 형식 변환을 모두 수행합니다.

C#

```
DateTimeOffset originalTime = new DateTimeOffset(2008, 6, 19, 7, 0, 0, new
TimeSpan(5, 0, 0));
DateTime utcTime = originalTime.UtcDateTime;
Console.WriteLine($"{originalTime} converted to {utcTime} {utcTime.Kind}");
// The example displays the following output to the console:
//       6/19/2008 7:00:00 AM +05:00 converted to 6/19/2008 2:00:00 AM Utc
```

현지 시간 변환

DateTimeOffset 값이 현지 시간을 나타낸다는 것을 나타내기 위해 DateTime 속성에서 반환된 DateTimeOffset.DateTime 값을 `static`(Visual Basic 경우 `Shared`) SpecifyKind 메서드에 전달할 수 있습니다. 메서드는 전달된 날짜와 시간을 첫 번째 매개 변수로 반환하지만 두 번째 매개 변수로 지정된 값으로 속성을 설정합니다 Kind . 다음 코드는 SpecifyKind 메서드를 사용하여 오프셋이 로컬 시간대의 오프셋에 해당하는 DateTimeOffset 값을 변환합니다.

C#

```
DateTime sourceDate = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset utcTime1 = new DateTimeOffset(sourceDate,
                          TimeZoneInfo.Local.GetUtcOffset(sourceDate));
DateTime utcTime2 = utcTime1.DateTime;
if (utcTime1.Offset.Equals(TimeZoneInfo.Local.GetUtcOffset(utcTime1.DateTime)))
   utcTime2 = DateTime.SpecifyKind(utcTime2, DateTimeKind.Local);

Console.WriteLine($"{utcTime1} converted to {utcTime2} {utcTime2.Kind}");
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

이 속성을 사용하여 DateTimeOffset.LocalDateTime 값을 로컬 DateTimeOffset 값으로 DateTime 변환할 수도 있습니다. 반환된 Kind 값의 DateTime 속성은 Local입니다. 다음 코드는 오프셋이 현지 시간대의 오프셋에 해당하는 DateTimeOffset.LocalDateTime 값을 변환할 때 DateTimeOffset 속성을 사용합니다.

C#

```
DateTime sourceDate = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset localTime1 = new DateTimeOffset(sourceDate,
                          TimeZoneInfo.Local.GetUtcOffset(sourceDate));
DateTime localTime2 = localTime1.LocalDateTime;

Console.WriteLine($"{localTime1} converted to {localTime2} {localTime2.Kind}");
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

속성을 사용하여 DateTime 값을 검색 DateTimeOffset.LocalDateTime 할 때 속성의 `get` 접근자가 먼저 값을 UTC로 변환 DateTimeOffset 한 다음 메서드를 호출 ToLocalTime 하여 로컬 시간으

로 변환합니다. 이 동작은 형식 변환을 수행하는 동시에 표준 시간대 변환을 수행하기 위해 속성에서 DateTimeOffset.LocalDateTime 값을 검색할 수 있음을 의미합니다. 또한 변환을 수행할 때 현지 표준 시간대의 조정 규칙이 적용됨을 의미합니다. 다음 코드에서는 형식 및 표준 시간대 변환을 모두 수행 하려면 속성을 사용 DateTimeOffset.LocalDateTime 하는 방법을 보여 줍니다. 샘플 출력은 태평양 표준 시간대(미국 및 캐나다)로 설정된 머신에 대한 것입니다. 11월 날짜는 태평양 표준시(UTC-8)이며, 6월 날짜는 일광 절약 시간(UTC-7)입니다.

C#

```
DateTimeOffset originalDate;
DateTime localDate;

// Convert time originating in a different time zone
originalDate = new DateTimeOffset(2008, 6, 18, 7, 0, 0,
                                  new TimeSpan(-5, 0, 0));
localDate = originalDate.LocalDateTime;
Console.WriteLine($"{originalDate} converted to {localDate} {localDate.Kind}");
// Convert time originating in a different time zone
// so local time zone's adjustment rules are applied
originalDate = new DateTimeOffset(2007, 11, 4, 4, 0, 0,
                                  new TimeSpan(-5, 0, 0));
localDate = originalDate.LocalDateTime;
Console.WriteLine($"{originalDate} converted to {localDate} {localDate.Kind}");
// The example displays the following output to the console,
// when you run it on a machine that is set to Pacific Time (US & Canada):
//       6/18/2008 7:00:00 AM -05:00 converted to 6/18/2008 5:00:00 AM Local
//       11/4/2007 4:00:00 AM -05:00 converted to 11/4/2007 1:00:00 AM Local
```

범용 변환 방법

다음 예제에서는 `ConvertFromDateTimeOffset`이라는 이름의 메서드를 정의하며, 이는 DateTimeOffset 값을 DateTime 값으로 변환합니다. 오프셋에 따라 값이 UTC 시간, 현지 시간 또는 다른 시간인지를 DateTimeOffset 결정하고 그에 따라 반환된 날짜 및 시간 값의 Kind 속성을 정의합니다.

C#

```
static DateTime ConvertFromDateTimeOffset(DateTimeOffset dateTime)
{
   if (dateTime.Offset.Equals(TimeSpan.Zero))
      return dateTime.UtcDateTime;
   else if
(dateTime.Offset.Equals(TimeZoneInfo.Local.GetUtcOffset(dateTime.DateTime)))
      return DateTime.SpecifyKind(dateTime.DateTime, DateTimeKind.Local);
   else
      return dateTime.DateTime;
}
```

다음 예제에서는 메서드를 `ConvertFromDateTimeOffset` 호출하여 UTC 시간, 현지 시간 및 미국 중부 표준 시간대의 시간을 나타내는 값을 변환 DateTimeOffset 합니다.

C#

```
DateTime timeComponent = new DateTime(2008, 6, 19, 7, 0, 0);
DateTime returnedDate;

// Convert UTC time
DateTimeOffset utcTime = new DateTimeOffset(timeComponent, TimeSpan.Zero);
returnedDate = ConvertFromDateTimeOffset(utcTime);
Console.WriteLine($"{utcTime} converted to {returnedDate} {returnedDate.Kind}");

// Convert local time
DateTimeOffset localTime = new DateTimeOffset(timeComponent,
                           TimeZoneInfo.Local.GetUtcOffset(timeComponent));
returnedDate = ConvertFromDateTimeOffset(localTime);
Console.WriteLine($"{localTime} converted to {returnedDate} {returnedDate.Kind}");

// Convert Central Standard Time
DateTimeOffset cstTime = new DateTimeOffset(timeComponent,
               TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(timeComponent));
returnedDate = ConvertFromDateTimeOffset(cstTime);
Console.WriteLine($"{cstTime} converted to {returnedDate} {returnedDate.Kind}");
// The example displays the following output to the console:
//    6/19/2008 7:00:00 AM +00:00 converted to 6/19/2008 7:00:00 AM Utc
//    6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
//    6/19/2008 7:00:00 AM -05:00 converted to 6/19/2008 7:00:00 AM Unspecified
```

ⓘ 참고 항목

이 코드는 애플리케이션과 해당 날짜 및 시간 값의 원본에 따라 다음 두 가지 가정이 항상 유효하지 않을 수 있습니다.

- 오프셋이 UTC를 나타내는 날짜 및 시간 값이라고 TimeSpan.Zero 가정합니다. 실제로 UTC는 특정 표준 시간대의 시간이 아니라 세계 표준 시간대의 시간이 표준화된 시간과 관련된 시간입니다. 표준 시간대의 오프셋 Zero을 가질 수도 있습니다.
- 오프셋이 현지 표준 시간대와 같은 날짜 및 시간이 현지 표준 시간대를 나타낸다고 가정합니다. 날짜 및 시간 값은 원래 표준 시간대에서 연결되지 않으므로 그렇지 않을 수 있습니다. 날짜와 시간이 동일한 오프셋을 사용하여 다른 표준 시간대에서 시작되었을 수 있습니다.

참고하십시오

- 날짜, 시간 및 표준 시간대

Last updated on 2026. 03. 31.

표준 시간대 간 시간 변환

아티클 • 2025. 04. 03.

날짜 및 시간을 사용하는 모든 애플리케이션에서 표준 시간대 간의 차이를 처리하는 것이 점점 더 중요해지고 있습니다. 애플리케이션은 더 이상 모든 시간을 DateTime 구조에서 제공되는 현지 시간으로 표현할 수 있다고 가정할 수 없습니다. 예를 들어 미국 동부 지역의 현재 시간을 표시하는 웹 페이지는 동아시아의 고객에 대한 신뢰성이 부족합니다. 이 문서에서는 시간대 간 시간 변환과 시간대 인식이 제한된 DateTimeOffset 값을 변환하는 방법을 설명합니다.

협정 세계시로 변환

UTC(협정 세계시)는 정밀도가 높은 원자성 시간 표준입니다. 전 세계의 표준 시간대는 UTC의 양수 또는 음수 오프셋으로 표현됩니다. 따라서 UTC는 시간대에 구애받지 않거나 시간대에 중립적인 시간을 제공합니다. 컴퓨터에서 날짜 및 시간의 이식성이 중요한 경우 UTC를 사용하는 것이 좋습니다. 날짜 및 시간을 사용하는 세부 정보 및 기타 모범 사례는 .NET Framework에서 DateTime을 사용하는 코딩 모범 사례를 참조하세요. 개별 표준 시간대를 UTC로 변환하면 시간을 쉽게 비교할 수 있습니다.

> **참고**
>
> 단일 시점을 DateTimeOffset 명확하게 나타내도록 구조체를 직렬화할 수도 있습니다. DateTimeOffset 개체는 UTC의 오프셋과 함께 날짜 및 시간 값을 저장하므로 항상 UTC와 관련하여 특정 시점을 나타냅니다.

시간을 UTC로 변환하는 가장 쉬운 방법은 (`Shared`Visual Basic에서) TimeZoneInfo.ConvertTimeToUtc(DateTime) 메서드를 `static` 호출하는 것입니다. 메서드에서 수행하는 정확한 변환은 다음 표와 같이 매개 변수 속성의 Kind 값 `dateTime` 에 따라 달라집니다.

테이블 확장

`DateTime.Kind`	변환
`DateTimeKind.Local`	현지 시간을 UTC로 변환합니다.
`DateTimeKind.Unspecified`	매개 변수가 `dateTime` 현지 시간이라고 가정하고 현지 시간을 UTC로 변환합니다.
`DateTimeKind.Utc`	변경되지 않은 매개 변수를 `dateTime` 반환합니다.

다음 코드는 현재 현지 시간을 UTC로 변환하고 결과를 콘솔에 표시합니다.

C#

```
DateTime dateNow = DateTime.Now;
Console.WriteLine($"The date and time are
{TimeZoneInfo.ConvertTimeToUtc(dateNow)} UTC.");
```

날짜 및 시간 값이 현지 시간 또는 UTC를 나타내지 않으면 메서드가 ToUniversalTime 잘못된 결과를 반환할 가능성이 높습니다. 그러나 이 메서드를 TimeZoneInfo.ConvertTimeToUtc 사용하여 지정된 표준 시간대에서 날짜와 시간을 변환할 수 있습니다. 대상 표준 시간대를 나타내는 개체를 검색하는 TimeZoneInfo 방법은 로컬 시스템에 정의된 표준 시간대 찾기를 참조하세요. 다음 코드에서는 이 메서드를 TimeZoneInfo.ConvertTimeToUtc 사용하여 동부 표준시를 UTC로 변환합니다.

C#

```
DateTime easternTime = new DateTime(2007, 01, 02, 12, 16, 00);
string easternZoneId = "Eastern Standard Time";
try
{
    TimeZoneInfo easternZone =
TimeZoneInfo.FindSystemTimeZoneById(easternZoneId);
    Console.WriteLine($"The date and time are
{TimeZoneInfo.ConvertTimeToUtc(easternTime, easternZone)} UTC.");
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine($"Unable to find the {easternZoneId} zone in the
registry.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine($"Registry data on the {easternZoneId} zone has been
corrupted.");
}
```

DateTime 메서드는 DateTime 개체의 Kind 속성과 표준 시간대가 일치하지 않으면 ArgumentException을(를) 던집니다. Kind 속성이 DateTimeKind.Local이지만 TimeZoneInfo 개체가 현지 표준 시간대를 나타내지 않거나, Kind 속성이 DateTimeKind.Utc인데 TimeZoneInfo 개체가 TimeZoneInfo.Utc가 아닌 경우 불일치가 발생합니다.

모든 이러한 메서드는 DateTime 값을 매개변수로 받아 DateTime 값을 반환합니다. 값의 경우, DateTimeOffset 구조체에는 현재 인스턴스의 날짜와 시간을 UTC로 변환하는 ToUniversalTime 인스턴스 메서드가 있습니다. 다음 예제에서는 메서드를 ToUniversalTime 호출하여 현지 시간을 여러 번 UTC로 변환합니다.

C#

```
DateTimeOffset localTime, otherTime, universalTime;

// Define local time in local time zone
localTime = new DateTimeOffset(new DateTime(2007, 6, 15, 12, 0, 0));
Console.WriteLine($"Local time: {localTime}");
Console.WriteLine();

// Convert local time to offset 0 and assign to otherTime
otherTime = localTime.ToOffset(TimeSpan.Zero);
Console.WriteLine($"Other time: {otherTime}");
Console.WriteLine($"{localTime} = {otherTime}:
{localTime.Equals(otherTime)}");
Console.WriteLine($"{localTime} exactly equals {otherTime}:
{localTime.EqualsExact(otherTime)}");
Console.WriteLine();

// Convert other time to UTC
universalTime = localTime.ToUniversalTime();
Console.WriteLine($"Universal time: {universalTime}");
Console.WriteLine($"{otherTime} = {universalTime}:
{universalTime.Equals(otherTime)}");
Console.WriteLine($"{otherTime} exactly equals {universalTime}:
{universalTime.EqualsExact(otherTime)}");
Console.WriteLine();

// The example produces the following output to the console:
//    Local time: 6/15/2007 12:00:00 PM -07:00
//
//    Other time: 6/15/2007 7:00:00 PM +00:00
//    6/15/2007 12:00:00 PM -07:00 = 6/15/2007 7:00:00 PM +00:00: True
//    6/15/2007 12:00:00 PM -07:00 exactly equals 6/15/2007 7:00:00 PM
+00:00: False
//
//    Universal time: 6/15/2007 7:00:00 PM +00:00
//    6/15/2007 7:00:00 PM +00:00 = 6/15/2007 7:00:00 PM +00:00: True
//    6/15/2007 7:00:00 PM +00:00 exactly equals 6/15/2007 7:00:00 PM
+00:00: True
```

UTC를 지정된 표준 시간대로 변환

UTC를 현지 시간으로 변환하려면 다음 UTC를 현지 시간으로 변환 섹션을 참조하세요. UTC를 지정한 표준 시간대의 시간으로 변환하려면 메서드를 호출합니다 ConvertTimeFromUtc . 이 메서드는 다음 두 개의 매개 변수를 사용합니다.

- 변환할 UTC입니다. 이 값의 Kind 속성은 `Unspecified` 또는 `Utc`로 설정되어야 합니다.
- UTC를 변환할 대상 시간대입니다.

다음 코드는 UTC를 중앙 표준시로 변환합니다.

C#

```
DateTime timeUtc = DateTime.UtcNow;
try
{
    TimeZoneInfo cstZone = TimeZoneInfo.FindSystemTimeZoneById("Central
Standard Time");
    DateTime cstTime = TimeZoneInfo.ConvertTimeFromUtc(timeUtc, cstZone);
    Console.WriteLine("The date and time are {0} {1}.",
                      cstTime,
                      cstZone.IsDaylightSavingTime(cstTime) ?
                              cstZone.DaylightName : cstZone.StandardName);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The registry does not define the Central Standard
Time zone.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("Registry data on the Central Standard Time zone has
been corrupted.");
}
```

UTC를 현지 시간으로 변환

UTC를 현지 시간으로 변환하려면 변환할 시간을 가진 개체의 DateTime 메서드를 호출 ToLocalTime 합니다. 메서드의 정확한 동작은 다음 표와 같이 개체 Kind 의 속성 값에 따라 달라집니다.

테이블 확장

`DateTime.Kind`	변환
`DateTimeKind.Local`	DateTime 변경되지 않은 값을 반환합니다.
`DateTimeKind.Unspecified`	값이 DateTime UTC이고 UTC를 현지 시간으로 변환한다고 가정합니다.
`DateTimeKind.Utc`	값을 현지 시간으로 변환합니다 DateTime .

참고

TimeZone.ToLocalTime 메서드는 `DateTime.ToLocalTime` 메서드와 동일하게 동작합니다. 변환하려면 날짜 및 시간 값인 단일 매개 변수가 필요합니다.

(`Shared`Visual Basic에서) TimeZoneInfo.ConvertTime 메서드를 사용하여 지정된 표준 시간대의 `static` 시간을 현지 시간으로 변환할 수도 있습니다. 이 기술은 다음 섹션에서 설명합니다.

두 표준 시간대 간 변환

클래스의 다음 두 가지 `static`(`Shared`는 Visual Basic에서) 메서드를 사용하여 어떤 두 표준 시간대 간에도 변환할 수 있습니다.

- ConvertTime

  이 메서드의 매개 변수는 변환할 날짜 및 시간 값, `TimeZoneInfo` 날짜 및 시간 값의 표준 시간대를 나타내는 개체 및 날짜 및 `TimeZoneInfo` 시간 값을 변환할 표준 시간대를 나타내는 개체입니다.

- ConvertTimeBySystemTimeZoneId

  이 메서드의 매개 변수는 변환할 날짜 및 시간 값, 날짜 및 시간 값의 표준 시간대 식별자 및 날짜 및 시간 값을 변환할 표준 시간대의 식별자입니다.

두 메서드 Kind 모두 변환할 날짜 및 시간 값의 속성과 TimeZoneInfo 해당 표준 시간대를 나타내는 개체 또는 표준 시간대 식별자가 서로 일치해야 합니다. 그렇지 않으면 ArgumentException가 던져집니다. 예를 들어 날짜 및 시간 값의 `Kind` 속성이 `DateTimeKind.Local`인 경우, 메서드에 매개 변수로 전달된 `TimeZoneInfo` 객체가 `TimeZoneInfo.Local`와 같지 않으면 예외가 발생합니다. 메서드에 매개 변수로 전달된 식별자가 `TimeZoneInfo.Local.Id`와 같지 않으면 예외가 throw됩니다.

다음 예제에서는 이 메서드를 ConvertTime 사용하여 하와이 표준시에서 현지 시간으로 변환합니다.

C#

```
DateTime hwTime = new DateTime(2007, 02, 01, 08, 00, 00);
try
{
    TimeZoneInfo hwZone = TimeZoneInfo.FindSystemTimeZoneById("Hawaiian
Standard Time");
    Console.WriteLine("{0} {1} is {2} local time.",
            hwTime,
            hwZone.IsDaylightSavingTime(hwTime) ? hwZone.DaylightName :
hwZone.StandardName,
```

```
            TimeZoneInfo.ConvertTime(hwTime, hwZone, TimeZoneInfo.Local));
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The registry does not define the Hawaiian Standard
Time zone.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("Registry data on the Hawaiian Standard Time zone has
been corrupted.");
}
```

DateTimeOffset 값 변환

개체가 DateTimeOffset 인스턴스화될 때 해당 표준 시간대에서 연결이 해제되므로 개체가 나타내는 날짜 및 시간 값은 완전히 표준 시간대를 인식하지 못합니다. 그러나 대부분의 경우 애플리케이션은 특정 표준 시간대가 아닌 UTC에서 두 개의 서로 다른 오프셋을 기반으로 날짜와 시간을 변환하기만 하면 됩니다. 이 변환을 수행하려면 현재 인스턴스의 ToOffset 메서드를 호출할 수 있습니다. 메서드의 단일 매개 변수는 메서드가 반환할 새 날짜 및 시간 값의 오프셋입니다.

예를 들어 웹 페이지에 대한 사용자 요청의 날짜 및 시간이 알려져 있고 MM/dd/yyyy hh:mm:ss zzzz 형식의 문자열로 serialize되는 경우 다음 메서드는 `ReturnTimeOnServer` 이 날짜 및 시간 값을 웹 서버의 날짜 및 시간으로 변환합니다.

C#

```
public DateTimeOffset ReturnTimeOnServer(string clientString)
{
   string format = @"M/d/yyyy H:m:s zzz";
   TimeSpan serverOffset =
TimeZoneInfo.Local.GetUtcOffset(DateTimeOffset.Now);

   try
   {
      DateTimeOffset clientTime = DateTimeOffset.ParseExact(clientString,
format, CultureInfo.InvariantCulture);
      DateTimeOffset serverTime = clientTime.ToOffset(serverOffset);
      return serverTime;
   }
   catch (FormatException)
   {
      return DateTimeOffset.MinValue;
   }
}
```

메서드가 UTC보다 5시간 이전 표준 시간대의 날짜 및 시간을 나타내는 "9/1/2007 5:32:07 -05:00" 문자열을 전달하는 경우 미국 태평양 표준 시간대에 있는 서버에 대해 "9/1/2007 오전 3:32:07"을 반환합니다.

또한 TimeZoneInfo 클래스에는 TimeZoneInfo 값을 사용하여 시간대 변환을 수행하는 TimeZoneInfo.ConvertTime(DateTimeOffset, TimeZoneInfo) 메서드의 오버로드도 포함됩니다. 메서드의 매개 변수는 DateTimeOffset 값과 시간대를 참조하여 시간을 변환하는 데 사용됩니다. 메서드 호출은 값을 반환합니다 DateTimeOffset . 예를 들어 이전 예제의 `ReturnTimeOnServer` 메서드를 다음과 같이 다시 작성하여 메서드를 호출 ConvertTime(DateTimeOffset, TimeZoneInfo) 할 수 있습니다.

C#

```
public DateTimeOffset ReturnTimeOnServer(string clientString)
{
   string format = @"M/d/yyyy H:m:s zzz";

   try
   {
      DateTimeOffset clientTime = DateTimeOffset.ParseExact(clientString,
format,
                                  CultureInfo.InvariantCulture);
      DateTimeOffset serverTime = TimeZoneInfo.ConvertTime(clientTime,
                                  TimeZoneInfo.Local);
      return serverTime;
   }
   catch (FormatException)
   {
      return DateTimeOffset.MinValue;
   }
}
```

참고하십시오

- TimeZoneInfo
- 날짜, 시간 및 표준 시간대
- 로컬 시스템에서 정의된 표준 시간대 찾기

방법: 모호한 시간 확인

모호한 시간은 하나 이상의 UTC(협정 세계시)를 매핑하는 시간입니다. 표준 시간대의 일광 절약 시간에서 표준 시간으로 전환하는 것과 같이 클록 시간을 뒤로 조정하는 경우 발생합니다. 모호한 시간을 처리하는 경우 다음 중 하나를 수행할 수 있습니다.

- 시간을 UTC로 매핑하는 방법에 대해 가정합니다. 예를 들어 모호한 시간은 항상 표준 시간대의 표준 시간으로 표시된다고 가정할 수 있습니다.
- 모호한 시간이 사용자로부터 입력된 데이터 항목인 경우 사용자가 모호성을 해결하도록 맡기면 됩니다.

이 토픽에서는 모호한 시간이 표준 시간대의 표준 시간을 나타낸다고 가정하여 확인하는 방법을 보여 줍니다.

모호한 시간을 표준 시간대의 표준 시간으로 매핑하려면

1. IsAmbiguousTime메서드를 호출하여 시간이 모호한지 여부를 확인합니다.
2. 시간이 모호한 경우, 표준 시간대의 BaseUtcOffset 속성이 반환한 TimeSpan 개체에서 시간을 뺍니다.
3. `static`(Visual Basic .NET의 `Shared`) SpecifyKind 메서드를 호출하여 UTC 날짜 및 시간 값의 Kind 속성을 DateTimeKind.Utc(으)로 설정합니다.

예시

다음 예제에서는 모호한 시간이 현지 표준 시간대의 표준 시간을 나타낸다고 가정하여 UTC로 변환하는 방법을 설명합니다.

C#

```
private static DateTime ResolveAmbiguousTime(DateTime ambiguousTime)
{
    // Time is not ambiguous
    if (!TimeZoneInfo.Local.IsAmbiguousTime(ambiguousTime))
    {
        return ambiguousTime;
    }
    // Time is ambiguous
    else
    {
        DateTime utcTime = DateTime.SpecifyKind(ambiguousTime -
TimeZoneInfo.Local.BaseUtcOffset,
                                                DateTimeKind.Utc);
        Console.WriteLine("{0} local time corresponds to {1} {2}.",
```

```
                            ambiguousTime, utcTime, utcTime.Kind.ToString());
        return utcTime;
    }
}
```

예제에서는 전달된 `ResolveAmbiguousTime` 값이 모호한지 여부를 결정하는 DateTime(이)라는 메서드로 구성됩니다. 값이 모호한 경우 메서드는 해당 UTC 시간을 나타내는 DateTime 값을 반환합니다. 메서드는 현지 시간에서 현지 표준 시간대의 BaseUtcOffset 속성 값을 빼서 이 변환을 처리합니다.

일반적으로 모호한 시간에서 가능한 UTC 오프셋을 포함하는 GetAmbiguousTimeOffsets 개체의 배열을 검색하는 TimeSpan 메서드를 호출하여 모호한 시간을 처리합니다. 그러나 이 예제에서는 모호한 시간이 표준 시간대의 표준 시간으로 항상 매핑되어야 한다고 임의로 가정합니다. BaseUtcOffset 속성은 UTC와 표준 시간대의 표준 시간 간에 오프셋을 반환합니다.

이 예제에서는 TimeZoneInfo.Local 속성을 통해 현지 표준 시간대를 알아보았습니다. 로컬 표준 시간대는 개체 변수에 할당되지 않습니다. TimeZoneInfo.ClearCachedData 메서드를 호출하면 현지 표준 시간대가 할당된 개체가 무효화되므로 이 방법을 사용하는 것이 좋습니다.

참고하기

- 날짜, 시간 및 표준 시간대
- 방법: 사용자의 모호한 시간 확인 작업 허용

Last updated on 2026. 03. 31.

방법: 사용자에게 모호한 시간 확인 작업 허용

아티클 • 2024. 03. 13.

모호한 시간은 하나 이상의 UTC(협정 세계시)를 매핑하는 시간입니다. 표준 시간대의 일광 절약 시간에서 표준 시간으로 전환하는 것과 같이 클록 시간을 뒤로 조정하는 경우 발생합니다. 모호한 시간을 처리하는 경우 다음 중 하나를 수행할 수 있습니다.

- 모호한 시간이 사용자로부터 입력된 데이터 항목인 경우 사용자가 모호성을 해결하도록 맡기면 됩니다.
- 시간을 UTC로 매핑하는 방법에 대해 가정합니다. 예를 들어 있는 모호한 시간을 항상 표준 시간대의 표준 시간으로 표시한다고 가정할 수 있습니다.

이 토픽에서는 사용자가 모호한 시간을 확인하도록 허용하는 방법을 보여 줍니다.

사용자가 모호한 시간을 확인하도록 허용하려면

1. 사용자가 입력한 시간과 날짜를 가져옵니다.
2. IsAmbiguousTime 메서드를 호출하여 시간이 모호한지 여부를 확인합니다.
3. 시간이 모호한 경우 GetAmbiguousTimeOffsets 메서드를 호출하여 TimeSpan 개체 배열을 검색합니다. 배열의 각 요소에는 모호한 시간이 매핑할 수 있는 UTC 오프셋이 포함됩니다.
4. 사용자가 원하는 오프셋을 선택하도록 합니다.
5. 현지 시간에서 사용자가 선택한 오프셋을 빼서 UTC 날짜 및 시간을 가져옵니다.
6. `static`(Visual Basic .NET의 `Shared`) SpecifyKind 메서드를 호출하여 UTC 날짜 및 시간 값의 Kind 속성을 DateTimeKind.Utc(으)로 설정합니다.

예시

다음 예제에서는 날짜와 시간을 입력하라는 메시지가 표시되며 모호한 경우 모호한 시간이 매핑하는 UTC 시간을 사용자가 선택할 수 있습니다.

C#

```
private void GetUserDateInput()
{
    // Get date and time from user
```

```
    DateTime inputDate = GetUserDateTime();
    DateTime utcDate;

    // Exit if date has no significant value
    if (inputDate == DateTime.MinValue) return;

    if (TimeZoneInfo.Local.IsAmbiguousTime(inputDate))
    {
        Console.WriteLine("The date you've entered is ambiguous.");
        Console.WriteLine("Please select the correct offset from Universal
Coordinated Time:");
        TimeSpan[] offsets =
TimeZoneInfo.Local.GetAmbiguousTimeOffsets(inputDate);
        for (int ctr = 0; ctr < offsets.Length; ctr++)
        {
            Console.WriteLine($"{ctr}.) {offsets[ctr].Hours} hours,
{offsets[ctr].Minutes} minutes");
        }
        Console.Write("> ");
        int selection = Convert.ToInt32(Console.ReadLine());

        // Convert local time to UTC, and set Kind property to
DateTimeKind.Utc
        utcDate = DateTime.SpecifyKind(inputDate - offsets[selection],
DateTimeKind.Utc);

        Console.WriteLine($"{inputDate} local time corresponds to {utcDate}
{utcDate.Kind.ToString()}.");
    }
    else
    {
        utcDate = inputDate.ToUniversalTime();
        Console.WriteLine($"{inputDate} local time corresponds to {utcDate}
{utcDate.Kind.ToString()}.");
    }
}

private static DateTime GetUserDateTime()
{
    // Flag to exit loop if date is valid.
    bool exitFlag = false;
    string? dateString;
    DateTime inputDate = DateTime.MinValue;

    Console.Write("Enter a local date and time: ");
    while (!exitFlag)
    {
        dateString = Console.ReadLine();
        if (dateString?.ToUpper() == "E")
            exitFlag = true;

        if (DateTime.TryParse(dateString, out inputDate))
            exitFlag = true;
        else
            Console.Write("Enter a valid date and time, or enter 'e' to
```

```
exit: ");
    }

    return inputDate;
}
```

VB

```
Private Sub GetUserDateInput()
    ' Get date and time from user
    Dim inputDate As Date = GetUserDateTime()
    Dim utcDate As Date

    ' Exit if date has no significant value
    If inputDate = Date.MinValue Then Exit Sub

    If TimeZoneInfo.Local.IsAmbiguousTime(inputDate) Then
        Console.WriteLine("The date you've entered is ambiguous.")
        Console.WriteLine("Please select the correct offset from Universal
Coordinated Time:")
        Dim offsets() As TimeSpan =
TimeZoneInfo.Local.GetAmbiguousTimeOffsets(inputDate)
        For ctr As Integer = 0 to offsets.Length - 1
            Dim zoneDescription As String
            If offsets(ctr).Equals(TimeZoneInfo.Local.BaseUtcOffset) Then
                zoneDescription = TimeZoneInfo.Local.StandardName
            Else
                zoneDescription = TimeZoneInfo.Local.DaylightName
            End If
            Console.WriteLine("{0}.) {1} hours, {2} minutes ({3})", _
                              ctr, offsets(ctr).Hours, offsets(ctr).Minutes,
zoneDescription)
        Next
        Console.Write("> ")
        Dim selection As Integer = CInt(Console.ReadLine())

        ' Convert local time to UTC, and set Kind property to
DateTimeKind.Utc
        utcDate = Date.SpecifyKind(inputDate - offsets(selection),
DateTimeKind.Utc)

        Console.WriteLine("{0} local time corresponds to {1} {2}.",
inputDate, utcDate, utcDate.Kind.ToString())
    Else
        utcDate = inputDate.ToUniversalTime()
        Console.WriteLine("{0} local time corresponds to {1} {2}.",
inputDate, utcDate, utcDate.Kind.ToString())
    End If
End Sub

Private Function GetUserDateTime() As Date
    Dim exitFlag As Boolean = False            ' flag to exit loop if date
is valid
```

```
    Dim dateString As String
    Dim inputDate As Date = Date.MinValue

    Console.Write("Enter a local date and time: ")
    Do While Not exitFlag
        dateString = Console.ReadLine()
        If dateString.ToUpper = "E" Then exitFlag = True
        If Date.TryParse(dateString, inputDate) Then
            exitFlag = true
        Else
            Console.Write("Enter a valid date and time, or enter 'e' to
exit: ")
        End If
    Loop

    Return inputDate
End Function
```

예제 코드의 코어에서는 TimeSpan 개체의 배열을 사용하여 UTC에서 모호한 시간의 가능한 오프셋을 나타냅니다. 그러나 이러한 오프셋은 사용자에게 유용하지 않을 가능성이 높습니다. 오프셋의 의미를 분명히 하기 위해 코드는 오프셋이 현지 표준 시간대의 표준 시간 또는 일광 절약 시간을 나타내는지를 적어둡니다. 코드는 BaseUtcOffset 속성의 값을 가진 오프셋을 비교하여 표준 시간 및 월광 절약 시간을 결정합니다. 이 속성은 UTC와 표준 시간대의 표준 시간 사이의 차이를 나타냅니다.

이 예제에서는 TimeZoneInfo.Local 속성을 통해 현지 표준 시간대를 알아보았습니다. 로컬 표준 시간대는 개체 변수에 할당되지 않습니다. TimeZoneInfo.ClearCachedData 메서드를 호출하면 현지 표준 시간대가 할당된 개체가 무효화되므로 이 방법을 사용하는 것이 좋습니다.

코드 컴파일

이 예제에는 다음 사항이 필요합니다.

- `using` 문(C# 코드에 필요)을 사용하여 System 네임스페이스를 가져옵니다.

참고 항목

- 날짜, 시간 및 표준 시간대
- 방법: 모호한 시간 확인

동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

DateTimeOffset 개체 인스턴스화

이 구조는 DateTimeOffset 새 DateTimeOffset 값을 만드는 여러 가지 방법을 제공합니다. 대부분의 메서드는 UTC(협정 세계시)에서 날짜 및 시간 값의 오프셋을 지정할 수 있는 향상된 기능을 사용하여 새 DateTime 값을 인스턴스화하는 데 사용할 수 있는 메서드에 직접 해당합니다. 특히 다음과 같은 방법으로 값을 인스턴스화 DateTimeOffset 할 수 있습니다.

- 날짜 및 시간 리터럴을 사용합니다.
- DateTimeOffset 생성자를 호출합니다.
- 값을 DateTimeOffset 값으로 암시적으로 변환합니다.
- 날짜와 시간의 문자열을 분석하여 해석합니다.

이 항목에서는 새 DateTimeOffset 값을 인스턴스화하는 이러한 방법을 보여 주는 더 자세한 세부 정보 및 코드 예제를 제공합니다.

날짜 및 시간 리터럴

이를 지원하는 언어의 경우 값을 인스턴스화하는 DateTime 가장 일반적인 방법 중 하나는 날짜와 시간을 하드 코딩된 리터럴 값으로 제공하는 것입니다. 예를 들어 다음 Visual Basic 코드는 2008년 5월 1일 오전 8:06:32에 값이 있는 개체를 만듭니다 DateTime .

VB

```
Dim literalDate1 As Date = #05/01/2008 8:06:32 AM#
Console.WriteLine(literalDate1.ToString())
' Displays:
'              5/1/2008 8:06:32 AM
```

DateTimeOffset 리터럴을 지원하는 DateTime 언어를 사용할 때 날짜 및 시간 리터럴을 사용하여 값을 초기화할 수도 있습니다. 예를 들어 다음 Visual Basic 코드는 개체를 DateTimeOffset 만듭니다.

VB

```
Dim literalDate As DateTimeOffset = #05/01/2008 8:06:32 AM#
Console.WriteLine(literalDate.ToString())
' Displays:
'              5/1/2008 8:06:32 AM -07:00
```

콘솔 출력에서 보이는 것처럼, 이렇게 생성된 DateTimeOffset 값에는 현지 시간대의 오프셋이 할당됩니다. 즉 DateTimeOffset , 문자 리터럴을 사용하여 할당된 값은 코드가 다른 컴퓨터에서 실행되는 경우 단일 시간을 식별하지 않습니다.

DateTimeOffset 생성자

이 형식은 DateTimeOffset 6개의 생성자를 정의합니다. 그 중 네 개는 생성자 DateTime에 직접 해당하며, TimeSpan 유형의 추가 매개변수를 사용하여 UTC에서 날짜 및 시간의 오프셋을 정의합니다. 이를 통해 개별 날짜 및 시간 구성 요소의 값을 기반으로 값을 정의 DateTimeOffset 할 수 있습니다. 예를 들어 다음 코드는 이러한 네 개의 생성자를 사용하여 2008년 5월 1일 8:06:32 +01:00의 동일한 값으로 개체를 인스턴스화 DateTimeOffset 합니다.

C#

```
DateTimeOffset dateAndTime;

// Instantiate date and time using years, months, days,
// hours, minutes, and seconds
dateAndTime = new DateTimeOffset(2008, 5, 1, 8, 6, 32,
                                 new TimeSpan(1, 0, 0));
Console.WriteLine(dateAndTime);
// Instantiate date and time using years, months, days,
// hours, minutes, seconds, and milliseconds
dateAndTime = new DateTimeOffset(2008, 5, 1, 8, 6, 32, 545,
                                 new TimeSpan(1, 0, 0));
Console.WriteLine($"{dateAndTime.ToString("G")} {dateAndTime.ToString("zzz")}");

// Instantiate date and time using Persian calendar with years,
// months, days, hours, minutes, seconds, and milliseconds
dateAndTime = new DateTimeOffset(1387, 2, 12, 8, 6, 32, 545,
                                 new PersianCalendar(),
                                 new TimeSpan(1, 0, 0));
// Note that the console output displays the date in the Gregorian
// calendar, not the Persian calendar.
Console.WriteLine($"{dateAndTime.ToString("G")} {dateAndTime.ToString("zzz")}");

// Instantiate date and time using number of ticks
// 05/01/2008 8:06:32 AM is 633,452,259,920,000,000 ticks
dateAndTime = new DateTimeOffset(633452259920000000, new TimeSpan(1, 0, 0));
Console.WriteLine(dateAndTime);
// The example displays the following output to the console:
//       5/1/2008 8:06:32 AM +01:00
//       5/1/2008 8:06:32 AM +01:00
//       5/1/2008 8:06:32 AM +01:00
//       5/1/2008 8:06:32 AM +01:00
```

DateTimeOffset 개체를 생성자 인수 중 하나로 사용하여 인스턴스화된 PersianCalendar 개체의 값이 콘솔에 표시될 때, 그것은 페르시아 달력이 아닌 그레고리력의 날짜로 표현됩니다. 페르시아 달력을 사용하여 날짜를 출력하려면 항목의 예제를 PersianCalendar 참조하세요.

다른 두 생성자는 DateTimeOffset 값에서 DateTime 개체를 만듭니다. 이 중 첫 번째는 DateTime 값을 DateTimeOffset 값으로 변환하는 단일 매개 변수입니다. 결과 DateTimeOffset 값의 오프셋은 생성자의 단일 매개 변수 속성에 Kind 따라 달라집니다. 값이 DateTimeKind.Utc면 오프셋이

같게 TimeSpan.Zero설정됩니다. 그렇지 않으면 해당 오프셋이 현지 표준 시간대의 오프셋과 동일하게 설정됩니다. 다음 예제에서는 이 생성자를 사용하여 UTC 및 현지 표준 시간대를 나타내는 개체를 인스턴스화하는 DateTimeOffset 방법을 보여 줍니다.

C#

```
// Declare date; Kind property is DateTimeKind.Unspecified
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
DateTimeOffset targetTime;

// Instantiate a DateTimeOffset value from a UTC time
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = new DateTimeOffset(utcTime);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the offset is TimeSpan.Zero.

// Instantiate a DateTimeOffset value from a UTC time with a zero offset
targetTime = new DateTimeOffset(utcTime, TimeSpan.Zero);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the call to the constructor succeeds

// Instantiate a DateTimeOffset value from a UTC time with a negative offset
try
{
   targetTime = new DateTimeOffset(utcTime, new TimeSpan(-2, 0, 0));
   Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
   Console.WriteLine($"Attempt to create DateTimeOffset value from {targetTime}
failed.");
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM +00:00 failed.

// Instantiate a DateTimeOffset value from a local time
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
targetTime = new DateTimeOffset(localTime);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
// Because the Kind property is DateTimeKind.Local,
// the offset is that of the local time zone.

// Instantiate a DateTimeOffset value from an unspecified time
targetTime = new DateTimeOffset(sourceDate);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
```

```
// Because the Kind property is DateTimeKind.Unspecified,
// the offset is that of the local time zone.
```

> ⓘ **참고 항목**
>
> 단일 DateTimeOffset 매개 변수를 사용하는 DateTime 생성자의 오버로드를 호출하는 것은 DateTime 값을 DateTimeOffset 값으로 암시적으로 변환하는 것과 같습니다.

두 번째 생성자는 DateTimeOffset 값에서 DateTime 개체를 만듭니다. 이 생성자에는 두 개의 매개 변수가 있으며, 변환할 DateTime 값과 UTC로부터의 날짜 및 시간 오프셋을 나타내는 TimeSpan 값을 포함합니다. 이 오프셋 값은 Kind 생성자의 첫 번째 매개변수의 속성에 해당해야 하며, 그렇지 않으면 ArgumentException 예외가 발생합니다. 첫 번째 Kind 매개 변수의 속성이 DateTimeKind.Utc면 두 번째 매개 변수의 값은 이어야 TimeSpan.Zero합니다. Kind 첫 번째 매개 변수의 속성이DateTimeKind.Local면 두 번째 매개 변수의 값은 로컬 시스템 표준 시간대의 오프셋이어야 합니다. 첫 번째 Kind 매개 변수의 속성이 DateTimeKind.Unspecified인 경우, 오프셋은 어떤 값이든 유효할 수 있습니다. 이 생성자는 DateTime를 DateTimeOffset 값으로 변환하기 위해 호출됩니다. 다음 코드에서 이를 보여줍니다.

C#

```
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
DateTimeOffset targetTime;

// Instantiate a DateTimeOffset value from a UTC time with a zero offset.
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = new DateTimeOffset(utcTime, TimeSpan.Zero);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the call to the constructor succeeds

// Instantiate a DateTimeOffset value from a UTC time with a non-zero offset.
try
{
   targetTime = new DateTimeOffset(utcTime, new TimeSpan(-2, 0, 0));
   Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
   Console.WriteLine($"Attempt to create DateTimeOffset value from {utcTime}
failed.");
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM failed.

// Instantiate a DateTimeOffset value from a local time with
// the offset of the local time zone
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
```

```
targetTime = new DateTimeOffset(localTime,
                                TimeZoneInfo.Local.GetUtcOffset(localTime));
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
// Because the Kind property is DateTimeKind.Local and the offset matches
// that of the local time zone, the call to the constructor succeeds.

// Instantiate a DateTimeOffset value from a local time with a zero offset.
try
{
   targetTime = new DateTimeOffset(localTime, TimeSpan.Zero);
   Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
   Console.WriteLine($"Attempt to create DateTimeOffset value from {localTime}
failed.");
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM failed.

// Instantiate a DateTimeOffset value with an arbitrary time zone.
string timeZoneName = "Central Standard Time";
TimeSpan offset = TimeZoneInfo.FindSystemTimeZoneById(timeZoneName).
                          GetUtcOffset(sourceDate);
targetTime = new DateTimeOffset(sourceDate, offset);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -05:00
```

암시적 형식 변환

형식 DateTimeOffset은 값에서 DateTime 값으로의 DateTimeOffset 형식 변환을 지원합니다. 암시적 형식 변환은 명시적 캐스트(C#) 또는 변환(Visual Basic의 경우)이 필요하지 않고 정보를 잃지 않는 형식으로의 변환입니다. 다음과 같은 코드를 사용할 수 있습니다.

C#

```
DateTimeOffset targetTime;

// The Kind property of sourceDate is DateTimeKind.Unspecified
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
targetTime = sourceDate;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00

// define a UTC time (Kind property is DateTimeKind.Utc)
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = utcTime;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
```

```
// Define a local time (Kind property is DateTimeKind.Local)
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
targetTime = localTime;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
```

결과 DateTimeOffset 값의 오프셋은 속성 값에 DateTime.Kind 따라 달라집니다. 값이 DateTimeKind.Utc면 오프셋이 같게 TimeSpan.Zero설정됩니다. 값 중 하나 DateTimeKind.Local 이거나 DateTimeKind.Unspecified오프셋이 현지 표준 시간대의 값과 같게 설정됩니다.

날짜 및 시간의 문자열 표현 파싱

DateTimeOffset 형식은 날짜 및 시간의 문자열 표현을 DateTimeOffset 값으로 변환할 수 있는 네 가지 메서드를 지원합니다.

- Parse- 날짜 및 시간의 문자열 표현을 값으로 DateTimeOffset 변환하려고 시도하고 변환에 실패할 경우 예외를 throw합니다.
- TryParse- 날짜 및 시간의 문자열 표현을 값으로 DateTimeOffset 변환하려고 시도하고 변환에 실패하면 반환합니다 `false` .
- ParseExact- 지정된 형식의 날짜 및 시간의 문자열 표현을 값으로 DateTimeOffset 변환하려고 합니다. 변환이 실패하면 메서드가 예외를 throw합니다.
- TryParseExact- 지정된 형식의 날짜 및 시간의 문자열 표현을 값으로 DateTimeOffset 변환하려고 합니다. 변환이 실패하면 메서드가 반환 `false` 됩니다.

다음 예제에서는 값을 인스턴스화하기 위해 이러한 네 개의 문자열 변환 메서드 각각에 대한 호출을 DateTimeOffset 보여 줍니다.

C#

```
string timeString;
DateTimeOffset targetTime;

timeString = "05/01/2008 8:30 AM +01:00";
try
{
   targetTime = DateTimeOffset.Parse(timeString);
   Console.WriteLine(targetTime);
}
catch (FormatException)
{
   Console.WriteLine($"Unable to parse {timeString}.");
}

timeString = "05/01/2008 8:30 AM";
if (DateTimeOffset.TryParse(timeString, out targetTime))
```

```
   Console.WriteLine(targetTime);
else
   Console.WriteLine($"Unable to parse {timeString}.");

timeString = "Thursday, 01 May 2008 08:30";
try
{
   targetTime = DateTimeOffset.ParseExact(timeString, "f",
                CultureInfo.InvariantCulture);
   Console.WriteLine(targetTime);
}
catch (FormatException)
{
   Console.WriteLine($"Unable to parse {timeString}.");
}

timeString = "Thursday, 01 May 2008 08:30 +02:00";
string formatString;
formatString = CultureInfo.InvariantCulture.DateTimeFormat.LongDatePattern +
               " " +
               CultureInfo.InvariantCulture.DateTimeFormat.ShortTimePattern +
               " zzz";
if (DateTimeOffset.TryParseExact(timeString,
                                 formatString,
                                 CultureInfo.InvariantCulture,
                                 DateTimeStyles.AllowLeadingWhite,
                                 out targetTime))
   Console.WriteLine(targetTime);
else
   Console.WriteLine($"Unable to parse {timeString}.");
// The example displays the following output to the console:
//    5/1/2008 8:30:00 AM +01:00
//    5/1/2008 8:30:00 AM -07:00
//    5/1/2008 8:30:00 AM -07:00
//    5/1/2008 8:30:00 AM +02:00
```

참고하십시오

- 날짜, 시간 및 표준 시간대

Last updated on 2026. 03. 31.

방법: 조정 규칙 없이 표준 시간대 만들기

아티클 • 2025. 04. 01.

애플리케이션에 필요한 정확한 표준 시간대 정보는 여러 가지 이유로 특정 시스템에 없을 수 있습니다.

- 표준 시간대는 로컬 시스템의 레지스트리에 정의된 적이 없습니다.
- 표준 시간대에 대한 데이터가 레지스트리에서 수정되거나 제거되었습니다.
- 표준 시간대는 존재하지만 특정 역사적 기간의 표준 시간대 조정에 대한 정확한 정보는 없습니다.

이러한 경우 CreateCustomTimeZone 메서드를 호출하여 애플리케이션에 필요한 표준 시간대를 정의할 수 있습니다. 이 메서드의 오버로드를 사용하여 조정 규칙이 있거나 없는 표준 시간대를 만들 수 있습니다. 표준 시간대에서 일광 절약 시간을 지원하는 경우 고정 또는 부동 조정 규칙을 사용하여 조정을 정의할 수 있습니다. (이러한 용어의 정의는 표준 시간대 개요"표준 시간대 용어" 섹션을 참조하세요.)

> **ⓘ 중요**
>
> CreateCustomTimeZone 메서드를 호출하여 만든 사용자 지정 표준 시간대는 레지스트리에 추가되지 않습니다. 대신 CreateCustomTimeZone 메서드 호출에서 반환된 개체 참조를 통해서만 액세스할 수 있습니다.

이 항목에서는 조정 규칙 없이 표준 시간대를 만드는 방법을 보여줍니다. 일광 절약 시간 조정 규칙이 적용되는 표준 시간대를 만들려면 방법: 표준 시간대 만들기 조정 규칙을 참조하세요.

조정 규칙 없이 표준 시간대를 만들려면

1. 표준 시간대의 표시 이름을 정의합니다.

   표시 이름은 표준 시간대의 UTC(협정 세계시)의 오프셋을 괄호로 묶은 다음 표준 시간대, 표준 시간대의 하나 이상의 도시 또는 표준 시간대의 하나 이상의 국가 또는 지역을 식별하는 문자열을 따르는 표준 형식을 따릅니다.

2. 표준 시간대 표준시 이름을 정의합니다. 일반적으로 이 문자열은 표준 시간대의 식별자로도 사용됩니다.

3. 표준 시간대의 표준 이름과 다른 식별자를 사용하려면 표준 시간대 식별자를 정의합니다.

4. UTC에서 표준 시간대의 오프셋을 정의하는 TimeSpan 개체를 인스턴스화합니다. UTC보다 늦은 시간이 있는 표준 시간대에는 양수 오프셋이 있습니다. UTC보다 이전 시간이 있는 표준시에는 음수 오프셋이 있습니다.

5. TimeZoneInfo.CreateCustomTimeZone(String, TimeSpan, String, String) 메서드를 호출하여 새 표준 시간대를 인스턴스화합니다.

예시

다음 예제에서는 조정 규칙이 없는 남극 Mawson에 대한 사용자 지정 표준 시간대를 정의합니다.

```
C#
```

```
string displayName = "(GMT+06:00) Antarctica/Mawson Time";
string standardName = "Mawson Time";
TimeSpan offset = new(06, 00, 00);
TimeZoneInfo mawson = TimeZoneInfo.CreateCustomTimeZone(standardName,
offset, displayName, standardName);
Console.WriteLine($"The current time is
{TimeZoneInfo.ConvertTime(DateTime.Now, TimeZoneInfo.Local, mawson)}
{mawson.StandardName}");
```

DisplayName 속성에 할당된 문자열은 표준 형식을 따릅니다. 이 형식은 UTC에서 표준 시간대의 오프셋 뒤에 표준 시간대에 대한 친숙한 설명이 뒤에 옵니다.

코드 컴파일

이 예제에는 다음 사항이 필요합니다.

- 다음 네임스페이스를 가져올 수 있습니다.

  ```
  C#
  ```

  ```
  using System.Collections.Generic;
  ```

참고하십시오

- 날짜, 시간 및 표준 시간대
- 표준 시간대 개요
- 방법: 조정 규칙을 사용하여 표준 시간대 만들기

방법: 조정 규칙을 사용하여 표준 시간대 만들기

아티클 • 2025. 04. 09.

애플리케이션에 필요한 정확한 표준 시간대 정보는 여러 가지 이유로 특정 시스템에 없을 수 있습니다.

- 표준 시간대는 로컬 시스템의 레지스트리에 정의된 적이 없습니다.
- 표준 시간대에 대한 데이터가 레지스트리에서 수정되거나 제거되었습니다.
- 표준 시간대에는 특정 기록 기간의 표준 시간대 조정에 대한 정확한 정보가 없습니다.

이러한 경우 CreateCustomTimeZone 메서드를 호출하여 애플리케이션에 필요한 표준 시간대를 정의할 수 있습니다. 이 메서드의 오버로드를 사용하여 조정 규칙이 있거나 없는 표준 시간대를 만들 수 있습니다. 표준 시간대에서 일광 절약 시간을 지원하는 경우 고정 또는 부동 조정 규칙을 사용하여 조정을 정의할 수 있습니다. (이러한 용어의 정의는 표준 시간대 개요"표준 시간대 용어" 섹션을 참조하세요.)

> **ⓘ 중요**
>
> CreateCustomTimeZone 메서드를 호출하여 만든 사용자 지정 표준 시간대는 레지스트리에 추가되지 않습니다. 대신 CreateCustomTimeZone 메서드 호출에서 반환된 개체 참조를 통해서만 액세스할 수 있습니다.

이 항목에서는 조정 규칙을 사용하여 표준 시간대를 만드는 방법을 보여줍니다. 일광 절약 시간 조정 규칙을 지원하지 않는 표준 시간대를 만들려면 방법: 조정 규칙 없이 표준 시간대 만들기를 참조하세요.

부동 조정 규칙을 사용하여 표준 시간대를 만들려면

1. 각 조정(즉, 특정 시간 간격 동안 표준 시간으로 전환할 때마다)에 대해 다음을 수행합니다.

   a. 표준 시간대 조정을 위한 시작 전환 시간을 정의합니다.

      메서드를 TimeZoneInfo.TransitionTime.CreateFloatingDateRule 호출하고 전환 시간을 정의하는 값, 전환 월을 정의하는 정수 값, 전환이 발생하는 주를 정의하는 정수 값 및 DayOfWeek 전환이 발생하는 요일을 정의하는 값을 전달 DateTime 해야 합니다. 이 메서드 호출은 개체를 TimeZoneInfo.TransitionTime 인스턴스화합니다.

   b. 표준 시간대 조정에 대한 종료 전환 시간을 정의합니다. 이렇게 하려면 메서드에 대한 또 다른 호출이 TimeZoneInfo.TransitionTime.CreateFloatingDateRule 필요합니다. 이 메

서드 호출은 두 번째 TimeZoneInfo.TransitionTime 개체를 인스턴스화합니다.

c. 메서드를 CreateAdjustmentRule 호출하여 조정의 유효 시작 및 종료 날짜, TimeSpan 전환 시간을 정의하는 개체 및 일광 절약 시간제로 전환이 발생하는 시기를 정의하는 두 TimeZoneInfo.TransitionTime 개체를 전달합니다. 이 메서드 호출은 개체를 TimeZoneInfo.AdjustmentRule 인스턴스화합니다.

d. 개체 배열 TimeZoneInfo.AdjustmentRule 에 TimeZoneInfo.AdjustmentRule 개체를 할당합니다.

2. 표준 시간대의 표시 이름을 정의합니다. 표시 이름은 표준 시간대의 UTC(협정 세계시)의 오프셋을 괄호로 묶은 다음 표준 시간대, 표준 시간대의 하나 이상의 도시 또는 표준 시간대의 하나 이상의 국가 또는 지역을 식별하는 문자열을 따르는 표준 형식을 따릅니다.

3. 표준 시간대 표준시 이름을 정의합니다. 일반적으로 이 문자열은 표준 시간대의 식별자로도 사용됩니다.

4. 표준 시간대의 일광 시간 이름을 정의합니다.

5. 표준 시간대의 표준 이름과 다른 식별자를 사용하려면 표준 시간대 식별자를 정의합니다.

6. UTC에서 표준 시간대의 오프셋을 정의하는 TimeSpan 개체를 인스턴스화합니다. UTC보다 늦은 시간이 있는 표준 시간대에는 양수 오프셋이 있습니다. UTC보다 이전 시간이 있는 표준시에는 음수 오프셋이 있습니다.

7. TimeZoneInfo.CreateCustomTimeZone(String, TimeSpan, String, String, String, TimeZoneInfo+AdjustmentRule[]) 메서드를 호출하여 새 표준 시간대를 인스턴스화합니다.

예시

다음 예제에서는 1918년부터 현재까지의 다양한 시간 간격에 대한 조정 규칙을 포함하는 미국의 중앙 표준 시간대를 정의합니다.

C#

```
TimeZoneInfo cst;
// Declare necessary TimeZoneInfo.AdjustmentRule objects for time zone
TimeSpan delta = new(1, 0, 0);
TimeZoneInfo.AdjustmentRule adjustment;
List<TimeZoneInfo.AdjustmentRule> adjustmentList = [];
// Declare transition time variables to hold transition time information
TimeZoneInfo.TransitionTime transitionRuleStart, transitionRuleEnd;

// Define new Central Standard Time zone 6 hours earlier than UTC
// Define rule 1 (for 1918-1919)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 03, 05, DayOfWeek.Sunday);
```

```
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 10, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1918,
1, 1), new DateTime(1919, 12, 31), delta,
                                                              transitionRuleStart,
transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 2 (for 1942)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 02, 09);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1942,
1, 1), new DateTime(1942, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 3 (for 1945)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 23, 0, 0), 08, 14);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 09, 30);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1945,
1, 1), new DateTime(1945, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define end rule (for 1967-2006)
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 10, 5, DayOfWeek.Sunday);
// Define rule 4 (for 1967-73)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1967,
1, 1), new DateTime(1973, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 5 (for 1974 only)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 01, 06);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1974,
1, 1), new DateTime(1974, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 6 (for 1975 only)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 02, 23);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1975,
1, 1), new DateTime(1975, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 7 (1976-1986)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 05, DayOfWeek.Sunday);
```

```
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1976,
1, 1), new DateTime(1986, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 8 (1987-2006)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 01, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1987,
1, 1), new DateTime(2006, 12, 31),
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 9 (2007- )
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 03, 02, DayOfWeek.Sunday);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 11, 01, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(2007,
1, 1), DateTime.MaxValue.Date,
                                                              delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);

// Convert list of adjustment rules to an array
TimeZoneInfo.AdjustmentRule[] adjustments = new
TimeZoneInfo.AdjustmentRule[adjustmentList.Count];
adjustmentList.CopyTo(adjustments);

cst = TimeZoneInfo.CreateCustomTimeZone("Central Standard Time", new TimeSpan(-6,
0, 0),
      "(GMT-06:00) Central Time (US Only)", "Central Standard Time",
      "Central Daylight Time", adjustments);
```

이 예제에서 만든 표준 시간대에는 여러 조정 규칙이 있습니다. 조정 규칙의 유효 시작 및 종료 날짜가 다른 조정 규칙의 날짜와 겹치지 않도록 주의해야 합니다. 겹치는 경우 InvalidTimeZoneException 예외가 발생합니다.

부동 조정 규칙의 경우 값 5는 특정 월의 CreateFloatingDateRule 마지막 주에 전환이 발생함을 나타내기 위해 메서드의 매개 변수에 전달 `week` 됩니다.

메서드 호출에 TimeZoneInfo.CreateCustomTimeZone(String, TimeSpan, String, String, String, TimeZoneInfo+AdjustmentRule[]) 사용할 개체 배열 TimeZoneInfo.AdjustmentRule 을 만들 때 코드는 표준 시간대에 대해 만들 조정 횟수에 필요한 크기로 배열을 초기화할 수 있습니다. 대신 이 코드 예제에서는 메서드를 Add 호출하여 개체의 TimeZoneInfo.AdjustmentRule 제네릭 List<T> 컬렉션에 각 조정 규칙을 추가합니다. 그런 다음 코드는 메서드를 CopyTo 호출하여 이 컬렉션의 멤버를 배열에 복사합니다.

또한 이 예제에서는 이 메서드를 CreateFixedDateRule 사용하여 고정 날짜 조정을 정의합니다. 이는 전환 매개 변수의 CreateFloatingDateRule 시간, 월 및 일만 필요하다는 점을 제외하고 메서

드를 호출하는 것과 유사합니다.

예제는 다음과 같은 코드를 사용하여 테스트할 수 있습니다.

```
C#
```

```
TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard Time");

DateTime pastDate1 = new(1942, 2, 11);
Console.WriteLine($"Is {pastDate1} daylight saving time:
{cst.IsDaylightSavingTime(pastDate1)}");

DateTime pastDate2 = new(1967, 10, 29, 1, 30, 00);
Console.WriteLine($"Is {pastDate2} ambiguous: {cst.IsAmbiguousTime(pastDate2)}");

DateTime pastDate3 = new(1974, 1, 7, 2, 59, 00);
Console.WriteLine("{0} {1} is {2} {3}",
    pastDate3,
    est.IsDaylightSavingTime(pastDate3) ? est.DaylightName : est.StandardName,
    TimeZoneInfo.ConvertTime(pastDate3, est, cst),
    cst.IsDaylightSavingTime(TimeZoneInfo.ConvertTime(pastDate3, est, cst)) ?
cst.DaylightName : cst.StandardName);

// This code produces the following output to the console:
//
//    Is 2/11/1942 12:00:00 AM daylight saving time: True
//    Is 10/29/1967 1:30:00 AM ambiguous: True
//    1/7/1974 2:59:00 AM Eastern Standard Time is 1/7/1974 2:59:00 AM Central
Daylight Time
```

코드 컴파일

이 예제에는 다음 사항이 필요합니다.

- 다음 네임스페이스를 가져올 수 있습니다.

  ```
  C#
  ```

  ```
  using System.Collections.Generic;
  ```

참고하십시오

- 날짜, 시간 및 표준 시간대
- 표준 시간대 개요
- 방법: 조정 규칙 없이 표준 시간대 만들기

로컬 시스템에 정의된 표준 시간대 찾기

클래스는 TimeZoneInfo 공용 생성자를 노출하지 않습니다. 따라서 키워드를 `new` 사용하여 새 TimeZoneInfo 개체를 만들 수 없습니다. 대신 TimeZoneInfo 레지스트리에서 미리 정의된 표준 시간대에 대한 정보를 검색하거나 사용자 지정 표준 시간대를 만들어 개체를 인스턴스화합니다. 이 항목에서는 레지스트리에 저장된 데이터에서 표준 시간대를 인스턴스화하는 것에 대해 설명합니다. 또한 `static` 클래스의 `shared` (TimeZoneInfoVisual Basic에서) 속성은 UTC(협정 세계시) 및 현지 표준 시간대에 대한 액세스를 제공합니다.

> **참고 항목**
>
> 레지스트리에 정의되지 않은 표준 시간대의 경우 메서드의 오버로드를 호출하여 사용자 지정 표준 시간대를 CreateCustomTimeZone 만들 수 있습니다. 사용자 지정 표준 시간대 만들기는 **방법: 조정 규칙 없이 표준 시간대 만들기** 및 **방법: 조정 규칙 항목이 있는 표준 시간대 만들기** 에 대해 설명합니다. 또한 FromSerializedString 메서드를 사용하여 TimeZoneInfo 객체를 직렬화된 문자열에서 복원하여 인스턴스화할 수 있습니다. 개체 직렬화 및 역직렬화 TimeZoneInfo 는 **방법: 포함된 리소스에 표준 시간대 저장** 및 **방법: 포함된 리소스 항목에서 표준 시간대 복원** 에 대해 설명합니다.

개별 표준 시간대에 액세스

이 클래스는 TimeZoneInfo UTC 시간과 현지 표준 시간대를 나타내는 두 개의 미리 정의된 표준 시간대 개체를 제공합니다. Utc 및 Local 속성에서 각각 사용할 수 있습니다. UTC 또는 현지 표준 시간대에 액세스하는 방법에 대한 지침은 방법: 미리 정의된 UTC 및 현지 표준 시간대 개체에 액세스하는 방법을 참조하세요.

레지스트리에 정의된 모든 표준 시간대를 TimeZoneInfo 나타내는 개체를 인스턴스화할 수도 있습니다. 특정 표준 시간대 개체를 인스턴스화하는 방법에 대한 지침은 방법: TimeZoneInfo 개체 인스턴스화를 참조하세요.

표준 시간대 식별자

표준 시간대 식별자는 표준 시간대를 고유하게 식별하는 키 필드입니다. 대부분의 키는 비교적 짧지만 표준 시간대 식별자는 비교적 깁니다. 대부분의 경우 해당 값은 표준 시간대의 표준시 이름을 제공하는 데 사용되는 속성에 해당 TimeZoneInfo.StandardName 합니다. 그러나 예외도 있습니다. 유효한 식별자를 제공하는 가장 좋은 방법은 시스템에서 사용할 수 있는 표준 시간대를 열거하고 관련 식별자를 적어 두는 것입니다.

참고하십시오

- 날짜, 시간 및 표준 시간대
- 방법: 미리 정의된 UTC 및 현지 표준 시간대 개체에 액세스
- TimeZoneInfo 개체 인스턴스화하는 방법
- 시간대 사이 시간 변환

Last updated on 2026. 03. 31.

방법: 컴퓨터에 있는 표준 시간대 열거

지정된 표준 시간대를 성공적으로 사용하려면 해당 표준 시간대에 대한 정보를 시스템에서 사용할 수 있어야 합니다. Windows XP 및 Windows Vista 운영 체제는 이 정보를 레지스트리에 저장합니다. 그러나 전 세계에 존재하는 총 표준 시간대 수는 크지만 레지스트리에는 하위 집합에 대한 정보만 포함됩니다. 또한 레지스트리 자체는 내용이 의도적이고 실수로 변경될 수 있는 동적 구조입니다. 따라서 애플리케이션은 특정 표준 시간대가 정의되고 시스템에서 사용 가능하다고 항상 가정할 수 없습니다. 표준 시간대 정보 애플리케이션을 사용하는 많은 애플리케이션의 첫 번째 단계는 필요한 표준 시간대를 로컬 시스템에서 사용할 수 있는지 여부를 확인하거나 사용자에게 선택할 표준 시간대 목록을 제공하는 것입니다. 이렇게 하려면 애플리케이션이 로컬 시스템에 정의된 표준 시간대를 열거해야 합니다.

> **참고 항목**
>
> 애플리케이션이 로컬 시스템에 정의되지 않을 수 있는 특정 표준 시간대의 존재에 의존하는 경우 애플리케이션은 표준 시간대에 대한 정보를 직렬화하고 역직렬화하여 해당 존재를 보장할 수 있습니다. 그런 다음 애플리케이션 사용자가 선택할 수 있도록 표준 시간대를 목록 컨트롤에 추가할 수 있습니다. 자세한 내용은 **방법: 포함된 리소스에 표준 시간대 저장** 및 **방법: 포함된 리소스에서 표준 시간대 복원을** 참조하세요.

로컬 시스템에 있는 표준 시간대를 열거하려면

1. TimeZoneInfo.GetSystemTimeZones 메서드를 호출합니다. 메서드는 개체의 ReadOnlyCollection<T> 제네릭 TimeZoneInfo 컬렉션을 반환합니다. 컬렉션의 항목은 해당 DisplayName 속성별로 정렬됩니다. 다음은 그 예입니다.

   C#

   ```
   ReadOnlyCollection<TimeZoneInfo> tzCollection;
   tzCollection = TimeZoneInfo.GetSystemTimeZones();
   ```

2. 루프(C#) 또는 TimeZoneInfo...를 사용하여 `foreach` 컬렉션의 개별 `For Each` 개체를 열거합니다. `Next` 루프(Visual Basic에서 사용하여) 각 개체에 필요한 모든 처리를 수행합니다. 예를 들어, 다음 코드는 1단계에서 반환된 ReadOnlyCollection<T> 개체의 TimeZoneInfo 컬렉션을 열거하고 콘솔에서 각 시간대의 표시 이름을 나열합니다.

   C#

   ```
   foreach (TimeZoneInfo timeZone in tzCollection)
       Console.WriteLine($"   {timeZone.Id}: {timeZone.DisplayName}");
   ```

사용자에게 로컬 시스템에 있는 표준 시간대 목록을 표시하려면

1. TimeZoneInfo.GetSystemTimeZones 메서드를 호출합니다. 메서드는 개체의 ReadOnlyCollection<T> 제네릭 TimeZoneInfo 컬렉션을 반환합니다.
2. 1 `DataSource` 단계에서 반환된 컬렉션을 Windows 양식 또는 ASP.NET 목록 컨트롤의 속성에 할당합니다.
3. TimeZoneInfo 사용자가 선택한 개체를 검색합니다.

이 예제에서는 Windows 애플리케이션에 대한 그림을 제공합니다.

예시

이 예제에서는 목록 상자의 시스템에 정의된 표준 시간대를 표시하는 Windows 애플리케이션을 시작합니다. 그런 다음 사용자가 선택한 표준 시간대 개체의 DisplayName 속성 값이 들어 있는 대화 상자를 표시하는 예제입니다.

C#

```
private void Form1_Load(object sender, EventArgs e)
{
    ReadOnlyCollection<TimeZoneInfo> tzCollection;
    tzCollection = TimeZoneInfo.GetSystemTimeZones();
    _timeZoneList.DataSource = tzCollection;
}

private void OkButton_Click(object sender, EventArgs e)
{
    TimeZoneInfo? selectedTimeZone = (TimeZoneInfo?)_timeZoneList.SelectedItem;
    MessageBox.Show($"You selected the {selectedTimeZone?.ToString()} time zone.");
}
```

대부분의 목록 컨트롤(예: System.Windows.Forms.ListBox 또는 System.Web.UI.WebControls.BulletedList 컨트롤)을 사용하면, 컬렉션이 IEnumerable 인터페이스를 구현하는 경우 해당 개체 변수 컬렉션을 `DataSource` 속성에 지정할 수 있습니다. (제네릭 ReadOnlyCollection<T> 클래스는 이 작업을 수행합니다.) 컬렉션에 개별 개체를 표시하기 위해 컨트롤은 해당 개체의 `ToString` 메서드를 호출하여 개체를 나타내는 데 사용되는 문자열을 추출합니다. TimeZoneInfo 개체의 경우, `ToString` 메서드는 DisplayName 속성 값인 TimeZoneInfo 개체의 표시 이름을 반환합니다.

참고 항목

목록 컨트롤은 개체의 `ToString` 메서드를 호출하므로 개체 컬렉션을 TimeZoneInfo 컨트롤에 할당하고 컨트롤에 각 개체에 대한 의미 있는 이름을 표시하고 사용자가 선택한 개체를 검색 TimeZoneInfo 할 수 있습니다. 이렇게 하면 컬렉션의 각 개체에 대한 문자열을 추출하고, 다시 컨트롤의 `DataSource` 속성에 할당된 컬렉션에 문자열을 할당하고, 사용자가 선택한 문자열을 검색한 다음, 이 문자열을 사용하여 설명하는 개체를 추출할 필요가 없습니다.

코드 컴파일

이 예제에는 다음 사항이 필요합니다.

- 다음 네임스페이스를 가져올 수 있습니다.

  System (C# 코드에서)

  System.Collections.ObjectModel

참고하십시오

- 날짜, 시간 및 표준 시간대
- 방법: 포함된 리소스에 표준 시간대 저장하는 방법
- 방법: 포함된 리소스에서 표준 시간대를 복원하는 방법

Last updated on 2026. 03. 31.

방법: 미리 정의된 UTC 및 현지 표준 시간대 개체에 액세스

TimeZoneInfo 클래스에서는 미리 정의된 시간대 개체에 코드 액세스를 제공하는 두 가지 속성(Utc 및 Local)을 제공합니다. 이 항목에서는 이러한 속성들이 반환하는 TimeZoneInfo 개체에 액세스하는 방법에 설명합니다.

UTC TimeZoneInfo 개체에 접근하려면

1. `static`(Visual Basic의 경우 `Shared`) TimeZoneInfo.Utc 속성을 사용하여 협정 세계시에 액세스합니다.

2. 속성에서 반환하는 TimeZoneInfo 개체를 개체 변수에 할당하는 대신 TimeZoneInfo.Utc 속성을 통해 협정 세계시에 계속 액세스합니다.

현지 시간대에 액세스하려면

1. `static`(Visual Basic의 경우 `Shared`) TimeZoneInfo.Local 속성을 사용하여 로컬 시스템 표준 시간대에 액세스합니다.

2. 속성에서 반환하는 TimeZoneInfo 개체를 개체 변수에 할당하는 대신 TimeZoneInfo.Local 속성을 통해 로컬 표준 시간대에 계속 액세스합니다.

예시

다음 코드에서는 TimeZoneInfo.Local 및 TimeZoneInfo.Utc 속성을 사용하여 시간을 미국 및 캐나다 동부 표준 시간대에서 변환하고 해당 표준 시간대 이름을 콘솔에 표시합니다.

C#

```
// Create Eastern Standard Time value and TimeZoneInfo object
DateTime estTime = new DateTime(2007, 1, 1, 00, 00, 00);
string timeZoneName = "Eastern Standard Time";
try
{
    TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById(timeZoneName);

    // Convert EST to local time
    DateTime localTime = TimeZoneInfo.ConvertTime(estTime, est,
TimeZoneInfo.Local);
    Console.WriteLine("At {0} {1}, the local time is {2} {3}.",
            estTime,
            est,
            localTime,
```

```
            TimeZoneInfo.Local.IsDaylightSavingTime(localTime) ?
                      TimeZoneInfo.Local.DaylightName :
                      TimeZoneInfo.Local.StandardName);

    // Convert EST to UTC
    DateTime utcTime = TimeZoneInfo.ConvertTime(estTime, est, TimeZoneInfo.Utc);
    Console.WriteLine("At {0} {1}, the time is {2} {3}.",
            estTime,
            est,
            utcTime,
            TimeZoneInfo.Utc.StandardName);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The {timeZoneName} zone cannot be found in the registry.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("The registry contains invalid data for the {timeZoneName}
zone.");
}

// The example produces the following output to the console:
//    At 1/1/2007 12:00:00 AM (UTC-05:00) Eastern Time (US & Canada), the local
time is 1/1/2007 12:00:00 AM Eastern Standard Time.
//    At 1/1/2007 12:00:00 AM (UTC-05:00) Eastern Time (US & Canada), the time is
1/1/2007 5:00:00 AM UTC.
```

로컬 표준 시간대를 TimeZoneInfo.Local 개체 변수에 할당하는 대신 항상 TimeZoneInfo 속성을 통해 로컬 표준 시간대에 액세스해야 합니다. 마찬가지로 UTC 영역을 TimeZoneInfo.Utc 개체 변수에 할당하는 대신 항상 TimeZoneInfo 속성을 통해 협정 세계시에 액세스해야 합니다. 이렇게 하면 TimeZoneInfo 메서드를 호출하여 TimeZoneInfo.ClearCachedData 개체 변수가 무효화되는 것을 방지할 수 있습니다.

참고하기

- 날짜, 시간 및 표준 시간대
- 로컬 시스템에 정의된 표준 시간대 찾기
- 방법: TimeZoneInfo 개체 인스턴스화

Last updated on 2026. 03. 31.

방법: TimeZoneInfo 개체 가져오기

개체를 가져오는 TimeZoneInfo 가장 일반적인 방법은 레지스트리에서 개체에 대한 정보를 검색하는 것입니다. 개체를 가져오려면 `static` (Visual Basic의 경우 `Shared`) 메서드를 호출하여 레지스트리를 찾습니다. 메서드에서 throw된 예외를 처리하며, 특히 레지스트리에 표준 시간대가 정의되지 않은 경우 발생하는 TimeZoneNotFoundException 예외를 다룹니다.

> ⓘ **참고 항목**
>
> .NET 8 TimeZoneInfo.FindSystemTimeZoneById 부터 새 개체를 인스턴스화하는 대신 캐시된 TimeZoneInfo 개체를 반환합니다. 자세한 내용은 FindSystemTimeZoneById가 새 개체를 반환하지 않음을 참조하세요.

예시

다음 코드는 동부 표준 시간대를 나타내는 개체를 검색 TimeZoneInfo 하고 현지 시간에 해당하는 동부 표준시를 표시합니다.

C#

```
DateTime timeNow = DateTime.Now;
try
{
    TimeZoneInfo easternZone = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
    DateTime easternTimeNow = TimeZoneInfo.ConvertTime(
        timeNow,
        TimeZoneInfo.Local,
        easternZone
        );
    Console.WriteLine("{0} {1} corresponds to {2} {3}.",
                      timeNow,
                      TimeZoneInfo.Local.IsDaylightSavingTime(timeNow) ?
                                TimeZoneInfo.Local.DaylightName :
                                TimeZoneInfo.Local.StandardName,
                      easternTimeNow,
                      easternZone.IsDaylightSavingTime(easternTimeNow) ?
                                  easternZone.DaylightName :
                                  easternZone.StandardName);
}
// Handle exception
//
// As an alternative to simply displaying an error message, an alternate Eastern
// Standard Time TimeZoneInfo object could be instantiated here either by restoring
// it from a serialized string or by providing the necessary data to the
// CreateCustomTimeZone method.
catch (TimeZoneNotFoundException)
```

```
{
    Console.WriteLine("The Eastern Standard Time Zone cannot be found on the local
system.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("The Eastern Standard Time Zone contains invalid or missing
data.");
}
catch (SecurityException)
{
    Console.WriteLine("The application lacks permission to read time zone
information from the registry.");
}
catch (OutOfMemoryException)
{
    Console.WriteLine("Not enough memory is available to load information on the
Eastern Standard Time zone.");
}
// If we weren't passing FindSystemTimeZoneById a literal string, we also
// would handle an ArgumentNullException.
```

TimeZoneInfo.FindSystemTimeZoneById 메서드의 단일 매개 변수는 검색할 표준 시간대의 식별자이며 이는 개체의 TimeZoneInfo.Id 속성에 해당합니다. 표준 시간대 식별자는 표준 시간대를 고유하게 식별하는 키 필드입니다. 대부분의 키는 비교적 짧지만 표준 시간대 식별자는 비교적 깁니다. 대부분의 경우, 값은 TimeZoneInfo 오브젝트의 StandardName 속성에 해당하며, 이는 표준 시간대의 표준시 이름을 제공하는 데 사용됩니다. 그러나 예외도 있습니다. 유효한 식별자를 제공하는 가장 좋은 방법은 시스템에서 사용할 수 있는 표준 시간대를 열거하고 해당 식별자에 있는 표준 시간대의 식별자를 적어 두는 것입니다. 자세한 내용은 방법: 컴퓨터에 있는 표준 시간대 열거를 참조하세요. 로컬 시스템 문서에 정의된 표준 시간대 찾기에는 선택한 표준 시간대 식별자 목록도 포함되어 있습니다.

표준 시간대를 찾은 경우 메서드는 해당 개체를 반환합니다 TimeZoneInfo . 표준 시간대를 찾지 못하면 메서드가 TimeZoneNotFoundException 예외를 발생시킵니다. 표준 시간대를 찾았지만 데이터가 손상되었거나 불완전한 경우, 메서드는 InvalidTimeZoneException를 던집니다.

애플리케이션이 있어야 하는 표준 시간대를 사용하는 경우 먼저 메서드를 FindSystemTimeZoneById 호출하여 레지스트리에서 표준 시간대 정보를 검색해야 합니다. 메서드 호출이 실패하면 예외 처리기가 표준 시간대의 새 인스턴스를 만들거나 직렬화된 TimeZoneInfo 개체를 역직렬화하여 다시 만들어야 합니다. 포함된 리소스에서 표준 시간대를 복원하는 방법의 예제를 참조하세요.

참고하십시오

- 날짜, 시간 및 표준 시간대
- 로컬 시스템에서 정의된 표준 시간대 찾기

- 방법: 미리 정의된 UTC 및 현지 표준 시간대 개체에 액세스

Last updated on 2026. 03. 31.

표준 시간대 저장 및 복원

아티클 • 2024. 03. 13.

TimeZoneInfo 클래스는 레지스트리를 사용하여 미리 정의된 표준 시간대 데이터를 검색합니다. 그러나 레지스트리는 동적 구조입니다. 또한 레지스트리에 포함된 표준 시간대 정보는 운영 체제에서 주로 현재 연도의 시간 조정 및 변환을 처리하는 데 사용됩니다. 이는 정확한 표준 시간대 데이터를 사용하는 애플리케이션에 다음과 같은 두 가지 주요한 영향을 미칩니다.

- 애플리케이션에 필요한 표준 시간대는 레지스트리에 정의되지 않았거나 레지스트리에서 이름이 변경되거나 제거되었을 수 있습니다.
- 레지스트리에 정의된 표준 시간대에는 과거 표준 시간대 변환에 필요한 특정 조정 규칙에 대한 정보가 부족할 수 있습니다.

TimeZoneInfo 클래스는 표준 시간대 데이터의 serialization(저장) 및 deserialization(복원)에 대한 지원을 통해 이러한 제한 사항을 해결합니다.

표준 시간대 serialization 및 deserialization

표준 시간대 데이터 직렬화 및 역직렬화를 통한 표준 시간대 저장 및 복원에는 다음 두 가지 메서드 호출만 포함됩니다.

- 해당 개체의 ToSerializedString 메서드를 호출하여 TimeZoneInfo 개체를 직렬화할 수 있습니다. 메서드는 매개 변수를 사용하지 않고 표준 시간대 정보가 포함된 문자열을 반환합니다.
- `static`(Visual Basic의 `Shared`) TimeZoneInfo.FromSerializedString 메서드에 해당 문자열을 전달하여 직렬화된 문자열에서 TimeZoneInfo 개체를 역직렬화할 수 있습니다.

serialization 및 deserialization 시나리오

나중에 사용할 수 있도록 TimeZoneInfo 개체를 문자열에 저장(또는 직렬화)하고 복원(또는 역직렬화)하는 기능은 TimeZoneInfo 클래스의 유틸리티와 유연성을 모두 향상합니다. 이 섹션에서는 serialization 및 deserialization이 가장 유용한 몇 가지 상황을 살펴봅니다.

애플리케이션에서 표준 시간대 데이터 직렬화 및 역직렬화

필요한 경우 문자열에서 직렬화된 표준 시간대를 복원할 수 있습니다. 레지스트리에서 검색된 표준 시간대가 특정 날짜 범위 내에서 날짜와 시간을 올바르게 변환할 수 없는 경우

애플리케이션에서 이 작업을 수행할 수 있습니다. 예를 들어 Windows XP 레지스트리의 표준 시간대 데이터는 단일 조정 규칙을 지원하지만 Windows Vista 레지스트리에 정의된 표준 시간대는 일반적으로 두 가지 조정 규칙에 대한 정보를 제공합니다. 이는 과거 시간 변환이 정확하지 않을 수 있음을 의미합니다. 표준 시간대 데이터의 serialization 및 deserialization는 이 제한을 처리할 수 있습니다.

다음 예제에서는 미국의 일광 절약 시간을 도입하기 전에 1883년에서 1917년 사이의 미국 동부 표준시를 나타내는 사용자 지정 TimeZoneInfo 클래스를 조정 규칙 없이 정의합니다. 사용자 지정 표준 시간대는 전역 범위가 있는 변수로 직렬화됩니다. 표준 시간대 변환 메서드 `ConvertUtcTime`(은)는 변환할 UTC(협정 세계시) 시간을 전달합니다. 날짜 및 시간이 1917년 또는 그 이전에 발생하는 경우 사용자 지정 동부 표준 시간대는 직렬화된 문자열에서 복원되고 레지스트리에서 검색된 표준 시간대를 대체합니다.

C#

```
using System;

public class TimeZoneSerialization
{
   static string serializedEst;

   public static void Main()
   {
      // Retrieve Eastern Standard Time zone from registry
      try
      {
         TimeZoneSerialization tzs = new TimeZoneSerialization();
         TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
         // Create custom Eastern Time Zone for historical (pre-1918)
conversions
         CreateTimeZone();
         // Call conversion function with one current and one pre-1918 date
and time
         Console.WriteLine(ConvertUtcTime(DateTime.UtcNow, est));
         Console.WriteLine(ConvertUtcTime(new DateTime(1900, 11, 15, 9, 32,
00, DateTimeKind.Utc), est));
      }
      catch (TimeZoneNotFoundException)
      {
         Console.WriteLine("The Eastern Standard Time zone is not in the
registry.");
      }
      catch (InvalidTimeZoneException)
      {
         Console.WriteLine("Data on the Eastern Standard Time Zone in the
registry is corrupted.");
      }
   }

   private static void CreateTimeZone()
```

```
   {
      // Create a simple Eastern Standard time zone
      // without adjustment rules for 1883-1918
      TimeZoneInfo earlyEstZone = TimeZoneInfo.CreateCustomTimeZone("Eastern
Standard Time",
                                          new TimeSpan(-5, 0, 0),
                                          " (GMT-05:00) Eastern Time (United
States)",
                                          "Eastern Standard Time");
      serializedEst = earlyEstZone.ToSerializedString();
   }

   private static DateTime ConvertUtcTime(DateTime utcDate, TimeZoneInfo tz)
   {
      // Use time zone object from registry
      if (utcDate.Year > 1917)
      {
         return TimeZoneInfo.ConvertTimeFromUtc(utcDate, tz);
      }
      // Handle dates before introduction of DST
      else
      {
         // Restore serialized time zone object
         tz = TimeZoneInfo.FromSerializedString(serializedEst);
         return TimeZoneInfo.ConvertTimeFromUtc(utcDate, tz);
      }
   }
}
```

표준 시간대 예외 처리

레지스트리는 동적 구조이므로 해당 콘텐츠는 실수로 또는 의도적으로 수정될 수 있습니다. 즉, 레지스트리에 정의되어야 하고 애플리케이션을 성공적으로 실행하는 데 필요한 표준 시간대가 없을 수 있습니다. 표준 시간대 serialization 및 deserialization을 지원하지 않으면 애플리케이션을 종료하여 결과 TimeZoneNotFoundException(을)를 처리할 수밖에 없습니다. 그러나 표준 시간대 serialization 및 deserialization을 사용하면 직렬화된 문자열에서 필요한 표준 시간대를 복원하여 예기치 않은 TimeZoneNotFoundException 작업을 처리할 수 있으며 애플리케이션은 계속 실행됩니다.

다음 예제에서는 사용자 지정 중앙 표준 시간대를 만들고 직렬화합니다. 그런 다음 레지스트리에서 중앙 표준 시간대를 검색하려고 시도합니다. 검색 작업이 TimeZoneNotFoundException 또는 InvalidTimeZoneException를 throw하는 경우 예외 처리기는 표준 시간대를 역직렬화합니다.

C#

```
using System;
using System.Collections.Generic;
```

```
public class TimeZoneApplication
{
   // Define collection of custom time zones
   private Dictionary<string, string> customTimeZones = new
Dictionary<string, string>();
   private TimeZoneInfo cst;

   public TimeZoneApplication()
   {
      // Create custom Central Standard Time
      //
      // Declare necessary TimeZoneInfo.AdjustmentRule objects for time zone
      TimeZoneInfo customTimeZone;
      TimeSpan delta = new TimeSpan(1, 0, 0);
      TimeZoneInfo.AdjustmentRule adjustment;
      List<TimeZoneInfo.AdjustmentRule> adjustmentList = new
List<TimeZoneInfo.AdjustmentRule>();
      // Declare transition time variables to hold transition time
information
      TimeZoneInfo.TransitionTime transitionRuleStart, transitionRuleEnd;

      // Define end rule (for 1976-2006)
      transitionRuleEnd =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 10, 5, DayOfWeek.Sunday);
      // Define rule (1976-1986)
      transitionRuleStart =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 04, 05, DayOfWeek.Sunday);
      adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1976, 1, 1), new DateTime(1986, 12, 31), delta,
transitionRuleStart, transitionRuleEnd);
      adjustmentList.Add(adjustment);
      // Define rule (1987-2006)
      transitionRuleStart =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 04, 01, DayOfWeek.Sunday);
      adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1987, 1, 1), new DateTime(2006, 12, 31), delta,
transitionRuleStart, transitionRuleEnd);
      adjustmentList.Add(adjustment);
      // Define rule (2007- )
      transitionRuleStart =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 03, 02, DayOfWeek.Sunday);
      transitionRuleEnd =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 11, 01, DayOfWeek.Sunday);
      adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(2007, 01, 01), DateTime.MaxValue.Date, delta, transitionRuleStart,
transitionRuleEnd);
      adjustmentList.Add(adjustment);

      // Create custom U.S. Central Standard Time zone
      customTimeZone = TimeZoneInfo.CreateCustomTimeZone("Central Standard
```

```
Time",
                        new TimeSpan(-6, 0, 0),
                        "(GMT-06:00) Central Time (US Only)", "Central
Standard Time",
                        "Central Daylight Time", adjustmentList.ToArray());
      // Add time zone to collection
      customTimeZones.Add(customTimeZone.Id,
customTimeZone.ToSerializedString());

      // Create any other required time zones
   }

   public static void Main()
   {
      TimeZoneApplication tza = new TimeZoneApplication();
      tza.AppEntryPoint();
   }

   private void AppEntryPoint()
   {
      try
      {
         cst = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
      }
      catch (TimeZoneNotFoundException)
      {
         if (customTimeZones.ContainsKey("Central Standard Time"))
            HandleTimeZoneException("Central Standard Time");
      }
      catch (InvalidTimeZoneException)
      {
         if (customTimeZones.ContainsKey("Central Standard Time"))
            HandleTimeZoneException("Central Standard Time");
      }
      if (cst == null)
      {
         Console.WriteLine("Unable to load Central Standard Time zone.");
         return;
      }
      DateTime currentTime = DateTime.Now;
      Console.WriteLine("The current {0} time is {1}.",
                        TimeZoneInfo.Local.IsDaylightSavingTime(currentTime)
?
                            TimeZoneInfo.Local.StandardName :
                            TimeZoneInfo.Local.DaylightName,
                        currentTime.ToString("f"));
      Console.WriteLine("The current {0} time is {1}.",
                        cst.IsDaylightSavingTime(currentTime) ?
                            cst.StandardName :
                            cst.DaylightName,
                        TimeZoneInfo.ConvertTime(currentTime,
TimeZoneInfo.Local, cst).ToString("f"));
   }

   private void HandleTimeZoneException(string timeZoneName)
```

```
    {
        string tzString = customTimeZones[timeZoneName];
        cst = TimeZoneInfo.FromSerializedString(tzString);
    }
}
```

직렬화된 문자열 저장 및 필요 시 복원

이전 예제에서는 표준 시간대 정보를 문자열 변수에 저장하고 필요할 때 복원했습니다. 그러나 직렬화된 표준 시간대 정보를 포함하는 문자열 자체는 외부 파일, 애플리케이션에 포함된 리소스 파일 또는 레지스트리와 같은 일부 스토리지 매체에 저장할 수 있습니다. (사용자 지정 표준 시간대에 대한 정보는 레지스트리에 있는 시스템의 표준 시간대 키와 별도로 저장되어야 합니다.)

이러한 방식으로 직렬화된 표준 시간대 문자열을 저장하면 표준 시간대 만들기 루틴이 애플리케이션 자체와 분리됩니다. 예를 들어 표준 시간대 만들기 루틴은 애플리케이션에서 사용할 수 있는 기록 표준 시간대 정보를 포함하는 데이터 파일을 실행하고 만들 수 있습니다. 그런 다음, 애플리케이션과 함께 데이터 파일을 설치할 수 있으며, 열 수 있으며 애플리케이션에 필요할 때 하나 이상의 표준 시간대를 역직렬화할 수 있습니다.

포함된 리소스를 사용하여 직렬화된 표준 시간대 데이터를 저장하는 예제는 방법: 포함된 리소스에 표준 시간대 저장 및 방법: 포함된 리소스에서 표준 시간대 복원을 참조하세요.

참고 항목

- 날짜, 시간 및 표준 시간대

방법: 포함된 리소스에 표준 시간대 저장

아티클 • 2025. 04. 01.

표준 시간대 인식 애플리케이션에는 종종 특정 표준 시간대가 있어야 합니다. 그러나 개별 TimeZoneInfo 개체의 가용성은 로컬 시스템의 레지스트리에 저장된 정보에 따라 달라지므로 일반적으로 사용 가능한 표준 시간대도 없을 수 있습니다. 또한 CreateCustomTimeZone 메서드를 사용하여 인스턴스화된 사용자 지정 표준 시간대에 대한 정보는 레지스트리의 다른 표준 시간대 정보와 함께 저장되지 않습니다. 필요할 때 이러한 표준 시간대를 사용할 수 있도록 하려면 표준 시간대를 직렬화하여 저장하고 나중에 역직렬화하여 복원할 수 있습니다.

일반적으로 TimeZoneInfo 개체 직렬화는 표준 시간대 인식 애플리케이션과는 별도로 발생합니다. 직렬화된 TimeZoneInfo 개체를 저장하는 데 사용되는 데이터 저장소에 따라 표준 시간대 데이터는 설치 또는 설치 루틴(예: 데이터가 레지스트리의 애플리케이션 키에 저장되는 경우)의 일부로 직렬화되거나 최종 애플리케이션이 컴파일되기 전에 실행되는 유틸리티 루틴의 일부로 직렬화될 수 있습니다(예: 직렬화된 데이터가 .NET XML 리소스(.resx) 파일에 저장되면 입니다.

애플리케이션으로 컴파일되는 리소스 파일 외에도 표준 시간대 정보에 다른 여러 데이터 저장소를 사용할 수 있습니다. 여기에는 다음이 포함되었습니다.

- 레지스트리입니다. 애플리케이션은 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Time Zones하위 키를 사용하는 대신 자체 애플리케이션 키의 하위 키를 사용하여 사용자 지정 표준 시간대 데이터를 저장해야 합니다.
- 구성 파일.
- 다른 시스템 파일.

표준 시간대를 .resx 파일로 직렬화하여 저장하려면

1. 기존 표준 시간대를 검색하거나 새 표준 시간대를 만듭니다.

   기존 표준 시간대를 검색하려면 방법: 미리 정의된 UTC 및 현지 표준 시간대 개체 액세스 및 방법: TimeZoneInfo 개체인스턴스화를 참조하세요.

   새 표준 시간대를 만들려면 CreateCustomTimeZone 메서드의 오버로드 중 하나를 호출합니다. 자세한 내용은 방법: 조정 규칙 없이 표준 시간대 만들기 및 방법: 조정 규칙표준 시간대 만들기를 참조하세요.

2. ToSerializedString 메서드를 호출하여 표준 시간대의 데이터가 포함된 문자열을 만듭니다.

3. 이름 및 선택적으로 .resx 파일의 경로를 StreamWriter 클래스 생성자에 제공하여 StreamWriter 개체를 인스턴스화합니다.

4. StreamWriter 개체를 ResXResourceWriter 클래스 생성자에 전달하여 ResXResourceWriter 개체를 인스턴스화합니다.

5. 표준 시간대의 직렬화된 문자열을 ResXResourceWriter.AddResource 메서드에 전달합니다.

6. ResXResourceWriter.Generate 메서드를 호출합니다.

7. ResXResourceWriter.Close 메서드를 호출합니다.

8. Close 메서드를 호출하여 StreamWriter 개체를 닫습니다.

9. 생성된 .resx 파일을 애플리케이션의 Visual Studio 프로젝트에 추가합니다.

10. Visual Studio의 **속성** 창을 사용하여 .resx 파일의 **빌드 작업** 속성이 **포함된 리소스**로 설정되었는지 확인하세요.

예시

다음은 Central Standard Time을 나타내는 TimeZoneInfo 개체와 남극의 Palmer Station을 나타내는 TimeZoneInfo 개체를 SerializedTimeZones.resx라는 .NET XML 리소스 파일로 직렬화하는 예제입니다. 중앙 표준시 일반적으로 레지스트리에 정의; 파머 역, 남극은 사용자 지정 표준 시간대입니다.

C#

```
TimeZoneSerialization()
{
   TextWriter writeStream;
   Dictionary<string, string> resources = new Dictionary<string, string>();
   // Determine if .resx file exists
   if (File.Exists(resxName))
   {
      // Open reader
      TextReader readStream = new StreamReader(resxName);
      ResXResourceReader resReader = new ResXResourceReader(readStream);
      foreach (DictionaryEntry item in resReader)
      {
         if (! (((string) item.Key) == "CentralStandardTime" ||
                ((string) item.Key) == "PalmerStandardTime" ))
            resources.Add((string)item.Key, (string) item.Value);
      }
```

```
      readStream.Close();
      // Delete file, since write method creates duplicate xml headers
      File.Delete(resxName);
   }

   // Open stream to write to .resx file
   try
   {
      writeStream = new StreamWriter(resxName, true);
   }
   catch (FileNotFoundException e)
   {
      // Handle failure to find file
      Console.WriteLine($"{e.GetType().Name}: The file {resxName} could not
be found.");
      return;
   }

   // Get resource writer
   ResXResourceWriter resWriter = new ResXResourceWriter(writeStream);

   // Add resources from existing file
   foreach (KeyValuePair<string, string> item in resources)
   {
      resWriter.AddResource(item.Key, item.Value);
   }

   // Serialize Central Standard Time
   try
   {
      TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById("Central
Standard Time");
      resWriter.AddResource(cst.Id.Replace(" ", string.Empty),
cst.ToSerializedString());
   }
   catch (TimeZoneNotFoundException)
   {
      Console.WriteLine("The Central Standard Time zone could not be
found.");
   }

   // Create time zone for Palmer, Antarctica
   //
   // Define transition times to/from DST
   TimeZoneInfo.TransitionTime startTransition =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 4,
0, 0),

10, 2, DayOfWeek.Sunday);
   TimeZoneInfo.TransitionTime endTransition =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 3,
0, 0),

3, 2, DayOfWeek.Sunday);
   // Define adjustment rule
```

```
   TimeSpan delta = new TimeSpan(1, 0, 0);
   TimeZoneInfo.AdjustmentRule adjustment =
TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1999, 10, 1),
                                        DateTime.MaxValue.Date, delta,
startTransition, endTransition);
   // Create array for adjustment rules
   TimeZoneInfo.AdjustmentRule[] adjustments = {adjustment};
   // Define other custom time zone arguments
   string DisplayName = "(GMT-04:00) Antarctica/Palmer Time";
   string standardName = "Palmer Standard Time";
   string daylightName = "Palmer Daylight Time";
   TimeSpan offset = new TimeSpan(-4, 0, 0);
   TimeZoneInfo palmer = TimeZoneInfo.CreateCustomTimeZone(standardName,
offset, DisplayName, standardName, daylightName, adjustments);
   resWriter.AddResource(palmer.Id.Replace(" ", String.Empty),
palmer.ToSerializedString());

   // Save changes to .resx file
   resWriter.Generate();
   resWriter.Close();
   writeStream.Close();
}
```

이 예제에서는 컴파일 시간에 리소스 파일에서 사용할 수 있도록 TimeZoneInfo 개체를 직렬화합니다.

ResXResourceWriter.Generate 메서드는 .NET XML 리소스 파일에 전체 헤더 정보를 추가하므로 기존 파일에 리소스를 추가하는 데 사용할 수 없습니다. 예제에서는 SerializedTimeZones.resx 파일이 존재하는 경우 두 개의 직렬화된 표준 시간대 이외의 모든 리소스를 제네릭 Dictionary<TKey,TValue> 개체에 저장합니다. 그러면 기존 파일이 삭제되고 기존 리소스가 새 SerializedTimeZones.resx 파일에 추가됩니다. 직렬화된 표준 시간대 데이터도 이 파일에 추가됩니다.

리소스의 키(또는 **이름**) 필드에는 포함된 공백이 포함되어서는 안 됩니다. Replace(String, String) 메서드는 리소스 파일에 할당되기 전에 표준 시간대 식별자의 모든 포함된 공간을 제거하기 위해 호출됩니다.

코드 컴파일

이 예제에는 다음 사항이 필요합니다.

- System.Windows.Forms.dll 및 System.Core.dll 대한 참조를 프로젝트에 추가할 수 있습니다.
- 다음 네임스페이스를 가져올 수 있습니다.

  C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;
```

참고하십시오

- 날짜, 시간 및 표준 시간대
- 표준 시간대 개요
- 방법: 포함된 리소스에서 표준 시간대를 복원하는 방법

방법: 포함 리소스에서 표준 시간대 복원

아티클 • 2024. 03. 12.

이 토픽에서는 리소스 파일에 저장된 표준 시간대를 복원하는 방법을 설명합니다. 표준 시간대 저장에 대한 자세한 내용 및 지침은 방법: 포함된 리소스에 표준 시간대 저장을 참조하세요.

포함된 리소스에서 TimeZoneInfo 개체를 역직렬화하려면 다음과 같이 수행합니다.

1. 검색할 표준 시간대가 사용자 지정 표준 시간대가 아닌 경우 FindSystemTimeZoneById 메서드를 사용하여 인스턴스화해 봅니다.

2. 포함된 리소스 파일의 정규화된 이름과 리소스 파일이 포함된 어셈블리에 대한 참조를 전달하여 ResourceManager 개체를 인스턴스화합니다.

   포함된 리소스 파일의 정규화된 이름을 확인할 수 없는 경우 Ildasm.exe(IL 디스어셈블러)를 사용하여 어셈블리의 매니페스트를 검사합니다. `.mresource` 항목은 리소스를 식별합니다. 예제에서 리소스의 정규화된 이름은 `SerializeTimeZoneData.SerializedTimeZones`입니다.

   리소스 파일이 표준 시간대 인스턴스화 코드를 포함하는 동일한 어셈블리에 포함된 경우 `static`(Visual Basic에서 `Shared`)GetExecutingAssembly 메서드를 호출하여 해당 파일에 대한 참조를 검색할 수 있습니다.

3. FindSystemTimeZoneById메서드 호출이 실패하거나 사용자 지정 표준 시간대를 인스턴스화할 경우 ResourceManager.GetString 메서드를 호출하여 직렬화된 표준 시간대가 포함된 문자열을 검색합니다.

4. FromSerializedString 메서드를 호출하여 표준 시간대 데이터를 역직렬화합니다.

예시

다음 예제에서는 포함된 .NET XML 리소스 파일에 저장된 TimeZoneInfo 개체를 역직렬화합니다.

C#

```
private void DeserializeTimeZones()
{
   TimeZoneInfo cst, palmer;
   string timeZoneString;
```

```
   ResourceManager resMgr = new
ResourceManager("SerializeTimeZoneData.SerializedTimeZones",
this.GetType().Assembly);

   // Attempt to retrieve time zone from system
   try
   {
      cst = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
   }
   catch (TimeZoneNotFoundException)
   {
      // Time zone not in system; retrieve from resource
      timeZoneString = resMgr.GetString("CentralStandardTime");
      if (! String.IsNullOrEmpty(timeZoneString))
      {
         cst = TimeZoneInfo.FromSerializedString(timeZoneString);
      }
      else
      {
         MessageBox.Show("Unable to create Central Standard Time Zone.
Application must exit.", "Application Error");
         return;
      }
   }
   // Retrieve custom time zone
   try
   {
      timeZoneString = resMgr.GetString("PalmerStandardTime");
      palmer = TimeZoneInfo.FromSerializedString(timeZoneString);
   }
   catch (MissingManifestResourceException)
   {
      MessageBox.Show("Unable to retrieve the Palmer Standard Time Zone from
the resource file. Application must exit.");
      return;
   }
}
```

이 코드는 애플리케이션에 필요한 TimeZoneInfo 개체가 있는지 확인하기 위한 예외 처리를 보여 줍니다. 먼저 FindSystemTimeZoneById 메서드를 사용하여 레지스트리에서 개체를 검색하여 TimeZoneInfo 개체를 인스턴스화하려고 합니다. 표준 시간대를 인스턴스화할 수 없는 경우 코드는 포함된 리소스 파일에서 검색합니다.

사용자 지정 표준 시간대(CreateCustomTimeZone 메서드를 사용하여 인스턴스화된 표준 시간대)에 대한 데이터는 레지스트리에 저장되지 않으므로 코드는 남극 팔머의 표준 시간대를 인스턴스화하기 위해 FindSystemTimeZoneById를 호출하지 않습니다. 대신 즉시 포함된 리소스 파일을 검색하여 FromSerializedString 메서드를 호출하기 전에 표준 시간대의 데이터가 포함된 문자열을 검색합니다.

코드 컴파일

이 예제에는 다음 사항이 필요합니다.

- System.Windows.Forms.dll 및 System.Core.dll 대한 참조를 프로젝트에 추가할 수 있습니다.
- 다음 네임스페이스를 가져올 수 있습니다.

```
C#

using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;
```

참고 항목

- 날짜, 시간 및 표준 시간대
- 표준 시간대 개요
- 방법: 포함 리소스에 표준 시간대 저장

System.DateTime 구조체

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> **ⓘ 중요**
>
> 일본 달력의 시대는 황제의 통치를 기반으로하므로 변경 될 것으로 예상된다. 예를 들어, 2019년 5월 1일은 레이와 시대의 JapaneseCalendar 시작을 표시했다 JapaneseLunisolarCalendar. 이러한 시대의 변화는 이러한 달력을 사용하는 모든 애플리케이션에 영향을 줍니다. 자세한 내용과 애플리케이션이 영향을 받는지 여부를 확인하려면 **.NET의 일본 달력에서 새 시대 처리를** 참조하세요. Windows 시스템에서 애플리케이션을 테스트하여 시대 변화에 대한 준비 상태를 확인하는 방법에 대한 자세한 내용은 **일본 시대 변경을 위한 애플리케이션 준비를 참조하세요**. 여러 연대가 있는 달력을 지원하는 .NET의 기능과 여러 연대를 지원하는 달력으로 작업할 때 모범 사례는 **연대 작업(Working with era)**을 참조하세요.

개요

값 형식은 그레고리력에서 서기 0001년 1월 1일 자정(00:00:00)부터 서기 9999년 12월 31일 오후 11시 59분 59초(C.E.)까지의 날짜 및 시간을 나타냅니다.

시간 값은 틱이라고 하는 100나노초 단위로 측정됩니다. 특정 날짜는 GregorianCalendar 달력의 서기 0001년 1월 1일 자정 12시 0분(C.E.) 이후의 틱 수입니다. 이 숫자는 윤초에 추가될 틱을 제외합니다. 예를 들어 31241376000000000L 틱 값은 금요일, 1월 1일, 0100 12:00:00 자정을 나타냅니다. DateTime 값은 항상 명시적 또는 기본 달력의 컨텍스트에서 표현됩니다.

> **ⓘ 참고**
>
> 변환할 틱 값을 사용하고 있으며, 이를 분이나 초와 같은 다른 시간 단위로 변환하려면, 변환 수행을 위해 TimeSpan.TicksPerDay, TimeSpan.TicksPerHour, TimeSpan.TicksPerMinute, TimeSpan.TicksPerSecond, 또는 TimeSpan.TicksPerMillisecond 상수를 사용해야 합니다. 예를 들어 지정된 틱 Second 수로 표시되는 시간(초)을 값의 DateTime 구성 요소에 추가하려면 식을 `dateValue.Second + nTicks/Timespan.TicksPerSecond` 사용할 수 있습니다.

Visual Basic, F# 또는 C#에서 이 문서의 전체 예제 집합에 대한 원본을 볼 수 있습니다.

> ⓘ 참고
>
> 특정 표준 시간대의 DateTime 날짜 및 시간 값을 사용하기 위한 구조체의 대안은 구조체 DateTimeOffset 입니다. 구조체는 날짜 및 시간 정보를 DateTimeOffset의 비공개 DateTime 필드에 저장하고, 그 날짜와 시간에서 UTC와의 차이를 분 단위로 표시하는 값을 비공개 Int16 필드에 저장합니다. 이렇게 하면 DateTimeOffset 값이 특정 표준 시간대의 시간을 반영할 수 있는 반면 DateTime 값은 UTC 및 현지 표준 시간대의 시간만 명확하게 반영할 수 있습니다. 날짜 및 시간 값을 다룰 때 DateTime 구조를 사용할지 DateTimeOffset 구조를 사용할지에 대한 토론 내용은 DateTime, DateTimeOffset, TimeSpan 및 TimeZoneInfo 중에서 선택 항목을 참조하세요.

예제 코드에 대한 빠른 링크

> ⓘ 참고
>
> 이 문서의 일부 C# 예제는 Try.NET 인라인 코드 실행기 및 플레이그라운드에서 실행됩니다. **실행** 단추를 선택하여 대화형 창에서 예제를 실행합니다. 코드를 실행하면 코드를 수정하고 **실행** 다시 선택하여 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나 컴파일에 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.
>
> Try.NET 인라인 코드 러너 및 플레이그라운드의 **현지 표준 시간대** 협정 세계시(UTC)입니다. 이는 DateTime, DateTimeOffset및 TimeZoneInfo 형식 및 해당 멤버를 보여 주는 예제의 동작 및 출력에 영향을 줄 수 있습니다.

이 문서에는 `DateTime` 형식을 사용하는 몇 가지 예제가 포함되어 있습니다.

초기화 예제

- 생성자 호출
- 암시적 매개 변수 없는 생성자 호출
- 반환 값에서 할당
- 날짜 및 시간을 나타내는 문자열 구문 분석
- 날짜 및 시간을 초기화하는 Visual Basic 구문

개체 `DateTime`를 문자열로 서식하는 예제

- 기본 날짜 시간 형식 사용
- 특정 문화권을 사용하여 날짜 및 시간 서식 지정

- 표준 또는 사용자 지정 형식 문자열을 사용하여 날짜 시간 서식 지정
- 형식 문자열과 특정 문화권 모두 지정
- 웹 서비스에 ISO 8601 표준을 사용하여 날짜 시간 형식 지정

문자열을 `DateTime` 개체로 구문 분석하는 예제

- Parse 또는 TryParse을 사용하여 문자열을 날짜 및 시간으로 변환
- ParseExact 또는 TryParseExact을 사용하여 알려진 형식의 문자열을 변환합니다
- ISO 8601 문자열 표현에서 날짜 및 시간으로 변환

`DateTime` 해결 방법 예제

- 날짜 및 시간 값의 해결 방법 살펴보기
- 허용 오차 내에서의 동등성 비교

문화 및 달력 사례

- 문화권별 달력을 사용하여 날짜 및 시간 값 표시
- 문화권별 달력에 따라 문자열 구문 분석
- 특정 문화권의 달력에서 날짜 및 시간 초기화
- 특정 문화권의 달력을 사용하여 날짜 및 시간 속성 액세스
- 문화권별 달력을 사용하여 올해의 주 검색

지속성 예제

- 현지 표준 시간대에서 날짜 및 시간 값을 문자열로 유지
- 날짜 및 시간 값을 문화권 및 시간 고정 형식의 문자열로 유지
- 날짜 및 시간 값을 정수로 유지
- XmlSerializer를 사용하여 날짜 및 시간 값 저장

DateTime 개체 초기화

여러 가지 방법으로 새 `DateTime` 값에 초기 값을 할당할 수 있습니다.

- 값에 대한 인수를 지정하거나 암시적 매개 변수 없는 생성자를 사용하는 생성자를 호출합니다.
- `DateTime` 속성 또는 메서드의 반환 값에 할당
- 문자열 표현에서 `DateTime` 값을 구문 분석합니다.
- Visual Basic 특화 언어 기능을 사용하여 `DateTime`을(를) 인스턴스화하기.

다음 코드 조각은 각각에 대한 예제를 보여 줍니다.

생성자 호출

날짜 및 시간 값의 DateTime 요소(예: 연도, 월, 일 또는 틱 수)를 지정하는 생성자의 오버로드를 호출합니다. 다음 코드는 연도, 월, 일, 시간, 분 및 초를 지정하는 생성자를 사용하여 DateTime 특정 날짜를 만듭니다.

```
C#
```

```
var date1 = new DateTime(2008, 5, 1, 8, 30, 52);
Console.WriteLine(date1);
```

`DateTime`를 기본값으로 초기화하려면 `DateTime` 구조체의 암시적 매개 변수 없는 생성자를 호출합니다. (값 형식의 암시적 매개 변수 없는 생성자에 대한 자세한 내용은 값 형식을 참조 하세요.) 일부 컴파일러에서는 값을 명시적으로 할당하지 않고도 값 선언 DateTime 을 지원합니다. 명시적 초기화 없이 값을 만들면 기본값도 생성됩니다. 다음 예제에서는 C# 및 Visual Basic의 DateTime 암시적 매개 변수 없는 생성자와 Visual Basic에서 할당이 없는 선언을 DateTime 보여 줍니다.

```
C#
```

```
var dat1 = new DateTime();
// The following method call displays 1/1/0001 12:00:00 AM.
Console.WriteLine(dat1.ToString(System.Globalization.CultureInfo.InvariantCu
lture));
// The following method call displays True.
Console.WriteLine(dat1.Equals(DateTime.MinValue));
```

계산 값 할당

속성 또는 메서드에서 반환된 DateTime 날짜 및 시간 값을 개체에 할당할 수 있습니다. 다음 예제에서는 현재 날짜 및 시간, 현재 UTC(협정 세계시) 날짜 및 시간 및 현재 날짜를 세 개의 새 DateTime 변수에 할당합니다.

```
C#
```

```
DateTime date1 = DateTime.Now;
DateTime date2 = DateTime.UtcNow;
DateTime date3 = DateTime.Today;
```

DateTime을 나타내는 문자열 구문 분석

, ParseExact, TryParse및 TryParseExact 메서드는 Parse모두 문자열을 해당하는 날짜 및 시간 값으로 변환합니다. 다음 예제에서는 Parse 및 ParseExact 메서드를 사용하여 문자열을 구문 분석하고 DateTime 값으로 변환합니다. 두 번째 형식은 문자열 형식의 날짜 및 시간을 나타내는 ISO 8601 표준에서 지원하는 양식을 사용합니다. 이 표준 표현은 종종 웹 서비스에서 날짜 정보를 전송하는 데 사용됩니다.

C#

```
var dateString = "5/1/2008 8:30:52 AM";
DateTime date1 = DateTime.Parse(dateString,

System.Globalization.CultureInfo.InvariantCulture);
var iso8601String = "20080501T08:30:52Z";
DateTime dateISO8602 = DateTime.ParseExact(iso8601String,
"yyyyMMddTHH:mm:ssZ",

System.Globalization.CultureInfo.InvariantCulture);
```

및 TryParseExact 메서드는 TryParse 문자열이 값의 DateTime 유효한 표현인지 여부를 나타내며, 값인 경우 변환을 수행합니다.

Visual Basic에 대한 언어별 구문

다음 Visual Basic 문은 새 DateTime 값을 초기화합니다.

VB

```
Dim date1 As Date = #5/1/2008 8:30:52AM#
```

DateTime 값 및 해당 문자열 표현

내부적으로 모든 DateTime 값은 0001년 1월 1일 자정 12:00:00 이후 경과된 틱 수(100나노초 간격 수)로 표시됩니다. 실제 DateTime 값은 해당 값이 표시될 때 나타나는 방식과 독립적입니다. 값의 DateTime 모양은 값을 문자열 표현으로 변환하는 서식 지정 작업의 결과입니다.

날짜 및 시간 값의 모양은 문화권, 국제 표준, 응용 프로그램 요구 사항 및 개인 기본 설정에 따라 달라집니다. 이 구조는 DateTime 오버로드를 통해 날짜 및 시간 값의 ToString서식을 유연하게 지정할 수 있습니다. 기본 DateTime.ToString() 메서드는 현재 문화권의 짧은 날짜 및 긴 시간 패턴을 사용하여 날짜 및 시간 값의 문자열 표현을 반환합니다. 다음 예제에서는 기본 DateTime.ToString() 메서드를 사용합니다. 현재 문화권의 짧은 날짜 및 긴 시간 패턴을 사용하여 날짜와 시간을 표시합니다. en-US 문화권은 예제가 실행된 컴퓨터의 현재 문화권입니다.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString());
// For en-US culture, displays 3/1/2008 7:00:00 AM
```

서버가 클라이언트와 다른 문화권에 있을 수 있는 웹 시나리오를 지원하려면 특정 문화권의 날짜 서식을 지정해야 할 수 있습니다. 메서드DateTime.ToString(IFormatProvider)를 사용하여 특정 문화권에서 짧은 날짜와 긴 시간을 표현하도록 문화권을 지정합니다. 다음 예제에서는 메서드를 DateTime.ToString(IFormatProvider) 사용 하 여 fr-FR 문화권에 대 한 짧은 날짜 및 긴 시간 패턴을 사용 하 여 날짜와 시간을 표시 합니다.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString(System.Globalization.CultureInfo.CreateSpec
ificCulture("fr-FR")));
// Displays 01/03/2008 07:00:00
```

다른 애플리케이션에는 날짜의 다른 문자열 표현이 필요할 수 있습니다. 이 메서드는 DateTime.ToString(String) 현재 문화권의 서식 규칙을 사용하여 표준 또는 사용자 지정 형식 지정자가 정의한 문자열 표현을 반환합니다. 다음 예제에서는 메서드를 DateTime.ToString(String) 사용하여 예제가 실행된 컴퓨터의 현재 문화권인 en-US 문화권에 대한 전체 날짜 및 시간 패턴을 표시합니다.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString("F"));
// Displays Saturday, March 01, 2008 7:00:00 AM
```

마지막으로 메서드를 사용하여 DateTime.ToString(String, IFormatProvider) 문화권과 형식을 모두 지정할 수 있습니다. 다음 예제에서는 메서드를 DateTime.ToString(String, IFormatProvider) 사용하여 fr-FR 문화권의 전체 날짜 및 시간 패턴을 표시합니다.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString("F", new
System.Globalization.CultureInfo("fr-FR")));
// Displays samedi 1 mars 2008 07:00:00
```

오버로드는 DateTime.ToString(String) 사용자 지정 형식 문자열과 함께 사용하여 다른 형식을 지정할 수도 있습니다. 다음 예제에서는 웹 서비스에 자주 사용되는 ISO 8601 표

준 형식을 사용하여 문자열의 서식을 지정하는 방법을 보여 줍니다. Iso 8601 형식에는 해당 표준 형식 문자열이 없습니다.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0, DateTimeKind.Utc);
Console.WriteLine(date1.ToString("yyyy-MM-ddTHH:mm:sszzz",
System.Globalization.CultureInfo.InvariantCulture));
// Displays 2008-03-01T07:00:00+00:00
```

값 서식 지정 DateTime 에 대한 자세한 내용은 표준 날짜 및 시간 서식 문자열 및사용자 지정 날짜 및 시간 형식 문자열을 참조하세요.

문자열에서 날짜 및 시간 값을 분석

구문 분석에서는 날짜 및 시간의 문자열 표현을 값으로 DateTime 변환합니다. 일반적으로 날짜 및 시간 문자열은 애플리케이션에서 두 가지 용도로 사용됩니다.

- 날짜와 시간은 다양한 형식을 사용하며 현재 문화권 또는 특정 문화권의 규칙을 반영합니다. 예를 들어 애플리케이션을 사용하면 현재 문화권이 en-US 사용자가 날짜 값을 "2013년 12월 15일" 또는 "2013년 12월 15일"로 입력할 수 있습니다. 현재 문화권이 en-gb 사용자가 날짜 값을 "2013년 15월 12일" 또는 "2013년 12월 15일"로 입력할 수 있습니다.

- 날짜 및 시간은 미리 정의된 형식으로 표시됩니다. 예를 들어 애플리케이션은 앱을 실행하는 문화권과 독립적으로 날짜를 "20130103"로 직렬화합니다. 애플리케이션은 현재 문화권의 짧은 날짜 형식으로 날짜를 입력해야 할 수 있습니다.

Parse 또는 TryParse 메서드를 사용하여 문화권에서 사용하는 공통 날짜 및 시간 형식 중 하나로부터 DateTime 값을 문자열로 변환합니다. 다음 예제에서는 다양한 문화권별 형식의 날짜 문자열을 DateTime 값으로 변환하기 위해 TryParse을 사용하는 방법을 보여 줍니다. 현재 문화권을 영어(영국)로 변경하고 메서드를 GetDateTimeFormats() 호출하여 날짜 및 시간 문자열의 배열을 생성합니다. 그런 다음 배열의 각 요소를 메서드에 TryParse 전달합니다. 예제의 출력은 구문 분석 메서드가 각 문화권별 날짜 및 시간 문자열을 성공적으로 변환할 수 있었다는 것을 보여줍니다.

```
C#
```

```
System.Threading.Thread.CurrentThread.CurrentCulture =
    System.Globalization.CultureInfo.CreateSpecificCulture("en-GB");

var date1 = new DateTime(2013, 6, 1, 12, 32, 30);
var badFormats = new List<String>();

Console.WriteLine($"{"Date String",-37} {"Date",-19}\n");
```

```
foreach (var dateString in date1.GetDateTimeFormats())
{
    DateTime parsedDate;
    if (DateTime.TryParse(dateString, out parsedDate))
        Console.WriteLine($"{dateString,-37}
{DateTime.Parse(dateString),-19}");
    else
        badFormats.Add(dateString);
}

// Display strings that could not be parsed.
if (badFormats.Count > 0)
{
    Console.WriteLine("\nStrings that could not be parsed: ");
    foreach (var badFormat in badFormats)
        Console.WriteLine($"   {badFormat}");
}
// Press "Run" to see the output.
```

ParseExact 및 TryParseExact 메서드를 사용하여 특정 형식과 일치해야 하는 문자열을 DateTime 값으로 변환합니다. 하나 이상의 날짜 및 시간 형식 문자열을 구문 분석 메서드의 매개 변수로 지정합니다. 다음 예제에서는 메서드를 TryParseExact(String, String[], IFormatProvider, DateTimeStyles, DateTime) 사용하여 "yyyyMMdd" 형식 또는 "HHmmss" 형식이어야 하는 문자열을 값으로 DateTime 변환합니다.

C#

```
string[] formats = { "yyyyMMdd", "HHmmss" };
string[] dateStrings = { "20130816", "20131608", "  20130816   ",
                   "115216", "521116", "  115216  " };
DateTime parsedDate;

foreach (var dateString in dateStrings)
{
    if (DateTime.TryParseExact(dateString, formats, null,

System.Globalization.DateTimeStyles.AllowWhiteSpaces |

System.Globalization.DateTimeStyles.AdjustToUniversal,
                               out parsedDate))
        Console.WriteLine($"{dateString} --> {parsedDate:g}");
    else
        Console.WriteLine($"Cannot convert {dateString}");
}
// The example displays the following output:
//       20130816 --> 8/16/2013 12:00 AM
//       Cannot convert 20131608
//         20130816    --> 8/16/2013 12:00 AM
//       115216 --> 4/22/2013 11:52 AM
//       Cannot convert 521116
//         115216   --> 4/22/2013 11:52 AM
```

한 가지 일반적인 용도 ParseExact 는 일반적으로 ISO 8601 표준 형식으로 웹 서비스에서 문자열 표현을 변환하는 것입니다. 다음 코드는 사용할 올바른 형식 문자열을 보여줍니다.

```
C#
```

```
var iso8601String = "20080501T08:30:52Z";
DateTime dateISO8602 = DateTime.ParseExact(iso8601String,
"yyyyMMddTHH:mm:ssZ",
    System.Globalization.CultureInfo.InvariantCulture);
Console.WriteLine($"{iso8601String} --> {dateISO8602:g}");
```

문자열을 구문 분석할 수 없는 경우, Parse 및 ParseExact 메서드는 예외를 throw합니다. TryParse 및 TryParseExact 메서드는 변환이 성공했는지 실패했는지를 나타내는 Boolean 값을 반환합니다. 성능이 중요한 시나리오에서는 TryParse 메서드나 TryParseExact 메서드를 사용해야 합니다. 날짜 및 시간 문자열에 대한 구문 분석 작업은 실패율이 높고 예외 처리 비용이 많이 듭니다. 문자열이 사용자가 입력하거나 알 수 없는 소스에서 오는 경우 이러한 메서드를 사용합니다.

날짜 및 시간 값 구문 분석에 대한 자세한 내용은 날짜 및 시간 문자열 구문 분석을 참조하세요.

날짜 시간 값

형식의 시간 값에 DateTime 대한 설명은 UTC(협정 세계시) 표준을 사용하여 표현되는 경우가 많습니다. 조정된 유니버설 타임은 그리니치 표준시(GMT)의 국제적으로 인정받는 이름입니다. 조정된 유니버설 타임은 경도 0도, UTC 원점으로 측정된 시간입니다. 일광 절약 시간제는 UTC에 적용되지 않습니다.

현지 시간은 특정 표준 시간대를 기준으로 합니다. 표준 시간대는 시간대 차이와 연결됩니다. 표준 시간대 오프셋은 UTC 원본 지점에서 몇 시간 단위로 측정된 표준 시간대의 변위입니다. 또한 현지 시간은 시간 간격 조정을 추가하거나 빼는 일광 절약 시간의 영향을 선택적으로 받습니다. 현지 시간은 UTC에 표준 시간대 오프셋을 추가하고 필요한 경우 일광 절약 시간을 조정하여 계산됩니다. UTC 원점의 표준 시간대 오프셋은 0입니다.

UTC 시간은 계산, 비교 및 날짜 및 시간을 파일에 저장하는 데 적합합니다. 로컬 시간은 데스크톱 애플리케이션의 사용자 인터페이스에 표시에 적합합니다. 표준 시간대 인식 애플리케이션(예: 많은 웹 애플리케이션)도 여러 다른 표준 시간대로 작업해야 합니다.

개체의 KindDateTime 속성이 DateTimeKind.Unspecified면 표시되는 시간이 현지 시간, UTC 시간 또는 다른 표준 시간대의 시간인지 지정되지 않습니다.

날짜 시간 해상도

> ⓘ 참고
>
> 경과된 시간을 측정하기 위해 값에 대한 DateTime 날짜 및 시간 산술 연산을 수행하는 대신 클래스를 Stopwatch 사용할 수 있습니다.

이 속성은 Ticks 날짜 및 시간 값을 1000만 초 단위로 표현합니다. 이 속성은 Millisecond 날짜 및 시간 값에서 1/10000초를 반환합니다. 경과된 시간을 측정하기 위해 속성에 DateTime.Now 대한 반복 호출을 사용하는 것은 시스템 클록에 따라 달라집니다. Windows 7 및 Windows 8 시스템의 시스템 클록의 해상도는 약 15밀리초입니다. 이 해상도는 100밀리초 미만의 작은 시간 간격에 영향을 줍니다.

다음 예제에서는 시스템 클록의 해상도에 대한 현재 날짜 및 시간 값의 의존성을 보여 줍니다. 이 예제에서 외부 루프는 20회 반복되고 내부 루프는 외부 루프를 지연시키는 역할을 합니다. 외부 루프 카운터의 값이 10이면 메서드를 호출하면 Thread.Sleep 5밀리초 지연이 발생합니다. 다음 예제에서는 Thread.Sleep를 호출한 후에만 `DateTime.Now.Milliseconds` 속성이 반환하는 밀리초 수를 보여 줍니다.

C#

```
string output = "";
for (int ctr = 0; ctr <= 20; ctr++)
{
    output += String.Format($"{DateTime.Now.Millisecond}\n");
    // Introduce a delay loop.
    for (int delay = 0; delay <= 1000; delay++)
    { }

    if (ctr == 10)
    {
        output += "Thread.Sleep called...\n";
        System.Threading.Thread.Sleep(5);
    }
}
Console.WriteLine(output);
// Press "Run" to see the output.
```

날짜 및 시간 작업

DateTime 구조를 사용한 계산은 Add나 Subtract와 같은 구조체의 값을 수정하지 않습니다. 대신 계산은 해당 값이 계산의 결과인 새 DateTime 구조를 반환합니다.

표준 시간대 간(예: UTC와 현지 시간 사이 또는 표준 시간대와 다른 표준 시간대 간) 간의 변환 작업은 일광 절약 시간을 고려하지만 산술 및 비교 작업은 고려하지 않습니다.

DateTime 구조체 자체는 한 표준 시간대에서 다른 표준 시간대로 변환하기 위한 제한된 지원을 제공합니다. 이 메서드를 ToLocalTime 사용하여 UTC를 현지 시간으로 변환하거나 이 메서드를 ToUniversalTime 사용하여 현지 시간에서 UTC로 변환할 수 있습니다. 그러나 TimeZoneInfo 클래스에서 전체 표준 시간대 변환 메서드를 사용할 수 있습니다. 이러한 방법을 사용하여 전 세계 표준 시간대의 시간을 다른 표준 시간대의 시간으로 변환합니다.

개체의 계산 및 비교 DateTime 는 개체가 동일한 표준 시간대의 시간을 나타내는 경우에만 의미가 있습니다. TimeZoneInfo 개체를 사용하여 DateTime 값의 표준 시간대를 나타낼 수 있지만, 두 항목은 느슨하게 결합되어 있습니다. DateTime 개체에는 해당 날짜 및 시간 값의 표준 시간대를 나타내는 개체를 반환하는 속성이 없습니다. 이 속성은 Kind UTC, 현지 시간을 나타내는지 `DateTime` 또는 지정되지 않은지를 나타냅니다. 표준 시간대 인식 애플리케이션에서는 일부 외부 메커니즘을 사용하여 개체가 만들어진 표준 시간대를 DateTime 결정해야 합니다. 값과 TimeZoneInfo 값의 표준 시간대를 DateTime 나타내는 개체를 모두 래핑하는 구조를 사용할 수 있습니다DateTime. 계산에 UTC를 사용하고 값과 DateTime 비교하는 방법에 대한 자세한 내용은 날짜 및 시간을 사용하여 산술 연산 수행을 참조하세요.

각 DateTime 멤버는 음력 달력을 암시적으로 사용하여 작업을 수행합니다. 예외는 달력을 암시적으로 지정하는 메서드입니다. 여기에는 달력을 지정하는 생성자와 다음과 같이 System.Globalization.DateTimeFormatInfo파생된 매개 변수가 있는 IFormatProvider메서드가 포함됩니다.

형식의 DateTime 멤버별 작업은 윤년 및 한 달의 일 수와 같은 세부 정보를 고려합니다.

DateTime 값 및 일정

.NET 클래스 라이브러리에는 다양한 달력 클래스가 포함되어 있으며, 모두 클래스에서 Calendar 파생됩니다. 그들은 다음과 같습니다.

- 클래스입니다 ChineseLunisolarCalendar .
- 클래스입니다 EastAsianLunisolarCalendar .
- 클래스입니다 GregorianCalendar .
- 클래스입니다 HebrewCalendar .
- 클래스입니다 HijriCalendar .
- 클래스입니다 JapaneseCalendar .
- 클래스입니다 JapaneseLunisolarCalendar .
- 클래스입니다 JulianCalendar .
- 클래스입니다 KoreanCalendar .

- 클래스입니다 KoreanLunisolarCalendar .
- 클래스입니다 PersianCalendar .
- 클래스입니다 TaiwanCalendar .
- 클래스입니다 TaiwanLunisolarCalendar .
- 클래스입니다 ThaiBuddhistCalendar .
- 클래스입니다 UmAlQuraCalendar .

> **ⓘ 중요**
>
> 일본 달력의 시대는 황제의 통치를 기반으로하므로 변경 될 것으로 예상된다. 예를 들어, 2019년 5월 1일은 JapaneseCalendar와 JapaneseLunisolarCalendar에서 레이와 시대의 시작의 표시가 되었다. 이러한 시대의 변화는 이러한 달력을 사용하는 모든 애플리케이션에 영향을 줍니다. 자세한 내용과 애플리케이션이 영향을 받는지 여부를 확인하려면 **.NET의 일본 달력에서 새 시대 처리를** 참조하세요. Windows 시스템에서 애플리케이션을 테스트하여 시대 변화에 대한 준비 상태를 확인하는 방법에 대한 자세한 내용은 **일본 시대 변경을 위한 애플리케이션 준비를 참조하세요**. 여러 연대가 있는 달력을 지원하는 .NET의 기능과 여러 연대를 지원하는 달력으로 작업할 때 모범 사례는 **연대 작업(Working with era)**을 참조하세요.

각 문화권은 읽기 전용 CultureInfo.Calendar 속성으로 정의된 기본 달력을 사용합니다. 각 문화권은 읽기 전용 CultureInfo.OptionalCalendars 속성으로 정의된 하나 이상의 일정을 지원할 수 있습니다. 현재 특정 CultureInfo 개체에서 사용되는 달력은 해당 DateTimeFormatInfo.Calendar 속성에 의해 정의됩니다. 배열에 있는 달력 CultureInfo.OptionalCalendars 중 하나여야 합니다.

문화권의 현재 달력은 해당 문화권에 대한 모든 서식 지정 작업에 사용됩니다. 예를 들어 태국 불교 문화의 기본 달력은 클래스가 나타내는 ThaiBuddhistCalendar 태국 불교 시대 달력입니다. CultureInfo 태국 불교 문화를 나타내는 개체가 날짜 및 시간 서식 작업에 사용되는 경우 태국 불교 시대 달력은 기본적으로 사용됩니다. 다음 예제와 같이 문화권의 DateTimeFormatInfo.Calendar 속성이 변경된 경우에만 그레고리오력이 사용됩니다.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var value = new DateTime(2016, 5, 28);

Console.WriteLine(value.ToString(thTH));

thTH.DateTimeFormat.Calendar = new System.Globalization.GregorianCalendar();
Console.WriteLine(value.ToString(thTH));
// The example displays the following output:
//       28/5/2559 0:00:00
//       28/5/2016 0:00:00
```

다음 예제와 같이 문화권의 현재 달력은 해당 문화권에 대한 모든 구문 분석 작업에도 사용됩니다.

```
C#
```

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var value = DateTime.Parse("28/05/2559", thTH);
Console.WriteLine(value.ToString(thTH));

thTH.DateTimeFormat.Calendar = new System.Globalization.GregorianCalendar();
Console.WriteLine(value.ToString(thTH));
// The example displays the following output:
//       28/5/2559 0:00:00
//       28/5/2016 0:00:00
```

특정 달력의 연, 월, 일 요소를 사용하여 DateTime 값을 인스턴스화하려면 `calendar` 매개 변수가 포함된 DateTime 생성자를 호출하고, 해당 달력을 나타내는 Calendar 객체를 전달해야 합니다. 다음 예제에서는 달력의 날짜 및 시간 요소를 ThaiBuddhistCalendar 사용합니다.

```
C#
```

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var dat = new DateTime(2559, 5, 28, thTH.DateTimeFormat.Calendar);
Console.WriteLine($"Thai Buddhist era date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Gregorian date:   {dat:d}");
// The example displays the following output:
//       Thai Buddhist Era Date:  28/5/2559
//       Gregorian Date:     28/05/2016
```

DateTime 매개 변수를 포함하지 `calendar` 않는 생성자는 날짜 및 시간 요소가 그레고리오력의 단위로 표현된다고 가정합니다.

다른 DateTime 모든 속성과 메서드는 그레고리오력을 사용합니다. 예를 들어 속성은 DateTime.Year 그레고리오력에서 연도를 반환하고 DateTime.IsLeapYear(Int32) 메서드는 매개 변수가 `year` 그레고리오력의 1년이라고 가정합니다. 양력 달력을 사용하는 각 DateTime 멤버에는 특정 달력을 사용하는 클래스의 Calendar 해당 멤버가 있습니다. 예를 들어 메서드는 Calendar.GetYear 특정 달력에서 연도를 반환하고 메서드는 Calendar.IsLeapYear 매개 변수를 특정 달력의 `year` 연도 번호로 해석합니다. 다음 예제에서는 DateTime 및 ThaiBuddhistCalendar 클래스의 멤버를 모두 사용합니다.

```
C#
```

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var cal = thTH.DateTimeFormat.Calendar;
var dat = new DateTime(2559, 5, 28, cal);
```

```
Console.WriteLine("Using the Thai Buddhist Era calendar:");
Console.WriteLine($"Date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Year: {cal.GetYear(dat)}");
Console.WriteLine($"Leap year: {cal.IsLeapYear(cal.GetYear(dat))}\n");

Console.WriteLine("Using the Gregorian calendar:");
Console.WriteLine($"Date: {dat:d}");
Console.WriteLine($"Year: {dat.Year}");
Console.WriteLine($"Leap year: {DateTime.IsLeapYear(dat.Year)}");
// The example displays the following output:
//       Using the Thai Buddhist Era calendar
//       Date :   28/5/2559
//       Year: 2559
//       Leap year :   True
//
//       Using the Gregorian calendar
//       Date :   28/05/2016
//       Year: 2016
//       Leap year :   True
```

DateTime 구조체에는 그레고리오력에서 요일을 반환하는 DayOfWeek 속성이 포함됩니다. 연도의 주 번호를 검색할 수 있는 멤버는 포함되지 않습니다. 연도의 주를 검색하려면 개별 일정의 Calendar.GetWeekOfYear 메서드를 호출합니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var thCalendar = thTH.DateTimeFormat.Calendar;
var dat = new DateTime(1395, 8, 18, thCalendar);
Console.WriteLine("Using the Thai Buddhist Era calendar:");
Console.WriteLine($"Date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Day of Week: {thCalendar.GetDayOfWeek(dat)}");
Console.WriteLine($"Week of year: {thCalendar.GetWeekOfYear(dat,
System.Globalization.CalendarWeekRule.FirstDay, DayOfWeek.Sunday)}\n");

var greg = new System.Globalization.GregorianCalendar();
Console.WriteLine("Using the Gregorian calendar:");
Console.WriteLine($"Date: {dat:d}");
Console.WriteLine($"Day of Week: {dat.DayOfWeek}");
Console.WriteLine($"Week of year: {greg.GetWeekOfYear(dat,
System.Globalization.CalendarWeekRule.FirstDay,DayOfWeek.Sunday)}");
// The example displays the following output:
//       Using the Thai Buddhist Era calendar
//       Date :  18/8/1395
//       Day of Week: Sunday
//       Week of year: 34
//
//       Using the Gregorian calendar
//       Date :  18/08/0852
```

```
//       Day of Week: Sunday
//       Week of year: 34
```

날짜 및 일정에 대한 자세한 내용은 일정 작업을 참조하세요.

DateTime 값 유지

다음과 같은 방법으로 DateTime 값을 저장할 수 있습니다.

- 문자열로 변환 하고 문자열을 유지합니다.
- 64비트 정수 값 (속성 값 Ticks)으로 변환하고 정수는 유지합니다.
- DateTime 값을 serialize합니다.

DateTime 값을 복원하는 루틴은 어떤 기술을 선택하든 데이터를 손실하거나 예외를 발생시키지 않도록 해야 합니다. DateTime 값은 왕복이어야 합니다. 즉, 원래 값과 복원된 값은 동일해야 합니다. 또한 원래 DateTime 값이 단일 시간의 순간을 나타내는 경우 복원될 때 동일한 순간을 식별해야 합니다.

값을 문자열로 유지

문자열로 유지되는 값을 성공적으로 복원 DateTime 하려면 다음 규칙을 따릅니다.

- 문자열을 유지할 때와 같이 문자열을 복원할 때 문화권별 서식 지정에 대해 동일한 가정을 합니다. 현재의 문화권과 다른 시스템에서 문자열을 복원할 수 있도록 하려면, 고정 문화권의 규칙을 사용하여 오버로드 ToString를 호출해 문자열을 저장하세요. Parse(String, IFormatProvider, DateTimeStyles) 오버로드 또는 TryParse(String, IFormatProvider, DateTimeStyles, DateTime) 오버로드를 호출하여 고정 문화권의 규칙에 따라 문자열을 복원합니다. 현재 문화권의 관례를 따르는 ToString(), Parse(String), 또는 TryParse(String, DateTime) 오버로드는 사용하지 마세요.

- 날짜가 단일 시간을 나타내는 경우 다른 표준 시간대에서도 복원할 때 동일한 시간을 나타내는지 확인합니다. 값을 저장하기 전에 DateTime 또는 DateTimeOffset을 사용하여 값을 UTC(협정 세계시)로 변환하십시오.

값을 문자열로 유지할 DateTime 때 발생하는 가장 일반적인 오류는 기본 또는 현재 문화권의 서식 지정 규칙에 의존하는 것입니다. 문자열을 저장하고 복원할 때 현재 문화권이 다른 경우 문제가 발생합니다. 다음 예제에서는 이러한 문제를 보여 줍니다. 현재 문화권의 서식 규칙을 사용하여 5개의 날짜를 저장합니다. 이 경우 영어(미국)입니다. 다른 문화권의 서식 규칙을 사용하여 날짜를 복원합니다. 이 경우 영어(영국)입니다. 두 문화권의 서식 규칙이 다르기 때문에 두 날짜를 복원할 수 없으며 나머지 세 날짜는 잘못 해석됩니다. 또한 원래 날짜 및 시간 값이 시간의 단일 순간을 나타내는 경우 표준 시간대 정보가 손실되므로 복원된 시간이 잘못되었습니다.

C#

```
public static void PersistAsLocalStrings()
{
    SaveLocalDatesAsString();
    RestoreLocalDatesFromString();
}

private static void SaveLocalDatesAsString()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                         new DateTime(2014, 7, 10, 23, 49, 0),
                         new DateTime(2015, 1, 10, 1, 16, 0),
                         new DateTime(2014, 12, 20, 21, 45, 0),
                         new DateTime(2014, 6, 2, 15, 14, 0) };
    string? output = null;

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        output += dates[ctr].ToString() + (ctr != dates.Length - 1 ? "|" :
"");
    }
    var sw = new StreamWriter(filenameTxt);
    sw.Write(output);
    sw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreLocalDatesFromString()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    StreamReader sr = new StreamReader(filenameTxt);
    string[] inputValues = sr.ReadToEnd().Split(new char[] { '|' },

StringSplitOptions.RemoveEmptyEntries);
    sr.Close();
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    foreach (var inputValue in inputValues)
    {
        DateTime dateValue;
        if (DateTime.TryParse(inputValue, out dateValue))
        {
            Console.WriteLine($"'{inputValue}' --> {dateValue:f}");
        }
```

```
            else
            {
                Console.WriteLine($"Cannot parse '{inputValue}'");
            }
        }
        Console.WriteLine("Restored dates...");
    }
    // When saved on an en-US system, the example displays the following output:
    //       Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
    //       The dates on an en-US system:
    //       Saturday, June 14, 2014 6:32 AM
    //       Thursday, July 10, 2014 11:49 PM
    //       Saturday, January 10, 2015 1:16 AM
    //       Saturday, December 20, 2014 9:45 PM
    //       Monday, June 02, 2014 3:14 PM
    //       Saved dates...
    //
    // When restored on an en-GB system, the example displays the following
    output:
    //       Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
    //       The dates on an en-GB system:
    //       Cannot parse //6/14/2014 6:32:00 AM//
    //       //7/10/2014 11:49:00 PM// --> 07 October 2014 23:49
    //       //1/10/2015 1:16:00 AM// --> 01 October 2015 01:16
    //       Cannot parse //12/20/2014 9:45:00 PM//
    //       //6/2/2014 3:14:00 PM// --> 06 February 2014 15:14
    //       Restored dates...
```

왕복 DateTime 값을 성공적으로 처리하려면 다음 단계를 따라야 합니다.

1. 값이 시간의 단일 순간을 나타내는 경우 메서드를 호출 ToUniversalTime 하여 현지 시간에서 UTC로 변환합니다.
2. ToString(String, IFormatProvider) 또는 String.Format(IFormatProvider, String, Object[]) 오버로드를 호출하여 날짜를 문자열 표현으로 변환합니다. CultureInfo.InvariantCulture을(를) `provider` 인수로 지정하여 불변 문화권의 서식 규칙을 사용합니다. "O" 또는 "R" 표준 형식 문자열을 사용하여 값이 왕복되도록 지정합니다.

데이터 손실 없이 지속형 DateTime 값을 복원하려면 다음 단계를 수행합니다.

1. ParseExact 오버로드 또는 TryParseExact 오버로드를 호출하여 데이터를 구문 분석합니다. CultureInfo.InvariantCulture를 `provider` 인수로 지정하고, 변환 중 `format` 인수에 사용한 것과 동일한 표준 형식 문자열을 사용하세요. 인수에 DateTimeStyles.RoundtripKind`styles` 값을 포함합니다.
2. 값이 시간 단위 DateTime 의 단일 순간을 나타내는 경우 메서드를 호출 ToLocalTime 하여 구문 분석된 날짜를 UTC에서 현지 시간으로 변환합니다.

다음 예제에서는 고정 문화권 및 "O" 표준 형식 문자열을 사용하여 저장 및 복원된 값이 DateTime 원본 및 대상 시스템의 시스템, 문화권 또는 표준 시간대에 관계없이 동일한 시간을 나타내도록 합니다.

C#

```
public static void PersistAsInvariantStrings()
{
    SaveDatesAsInvariantStrings();
    RestoreDatesAsInvariantStrings();
}

private static void SaveDatesAsInvariantStrings()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                         new DateTime(2014, 7, 10, 23, 49, 0),
                         new DateTime(2015, 1, 10, 1, 16, 0),
                         new DateTime(2014, 12, 20, 21, 45, 0),
                         new DateTime(2014, 6, 2, 15, 14, 0) };
    string? output = null;

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        output += dates[ctr].ToUniversalTime().ToString("O",
CultureInfo.InvariantCulture)
                  + (ctr != dates.Length - 1 ? "|" : "");
    }
    var sw = new StreamWriter(filenameTxt);
    sw.Write(output);
    sw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreDatesAsInvariantStrings()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    StreamReader sr = new StreamReader(filenameTxt);
    string[] inputValues = sr.ReadToEnd().Split(new char[] { '|' },

StringSplitOptions.RemoveEmptyEntries);
    sr.Close();
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    foreach (var inputValue in inputValues)
```

```
    {
        DateTime dateValue;
        if (DateTime.TryParseExact(inputValue, "O",
CultureInfo.InvariantCulture,
                                   DateTimeStyles.RoundtripKind, out dateValue))
        {
            Console.WriteLine($"'{inputValue}' -->
{dateValue.ToLocalTime():f}");
        }
        else
        {
            Console.WriteLine($"Cannot parse '{inputValue}'");
        }
    }
    Console.WriteLine("Restored dates...");
}
// When saved on an en-US system, the example displays the following output:
//       Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//       The dates on an en-US system:
//       Saturday, June 14, 2014 6:32 AM
//       Thursday, July 10, 2014 11:49 PM
//       Saturday, January 10, 2015 1:16 AM
//       Saturday, December 20, 2014 9:45 PM
//       Monday, June 02, 2014 3:14 PM
//       Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//       Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//       The dates on an en-GB system:
//       '2014-06-14T13:32:00.0000000Z' --> 14 June 2014 14:32
//       '2014-07-11T06:49:00.0000000Z' --> 11 July 2014 07:49
//       '2015-01-10T09:16:00.0000000Z' --> 10 January 2015 09:16
//       '2014-12-21T05:45:00.0000000Z' --> 21 December 2014 05:45
//       '2014-06-02T22:14:00.0000000Z' --> 02 June 2014 23:14
//       Restored dates...
```

값을 정수로 유지

날짜와 시간을 틱의 수를 나타내는 Int64 값으로 저장할 수 있습니다. 이 경우 값이 유지되고 복원되는 시스템의 DateTime 문화권을 고려할 필요가 없습니다.

값을 정수로 유지 DateTime 하려면 다음을 수행합니다.

1. 값이 시간 단위 DateTime 의 단일 순간을 나타내는 경우 메서드를 호출 ToUniversalTime 하여 UTC로 변환합니다.
2. 해당 속성의 Ticks 값이 표시하는 틱 수를 검색합니다.

정수로 DateTime 유지된 값을 복원하려면 다음을 수행합니다.

1. Int64 값을 생성자에 전달하여 새 DateTime 개체를 DateTime(Int64)로 인스턴스화합니다.
2. 값이 DateTime 한 순간을 나타내는 경우 메서드를 호출 ToLocalTime 하여 UTC에서 현지 시간으로 변환합니다.

다음 예제에서는 값 배열을 미국 태평양 표준 시간대의 DateTime 시스템에 정수로 유지합니다. UTC 영역의 시스템에서 복원합니다. 정수를 포함한 파일에는 Int64의 값 개수가 바로 뒤에 오는 총 개수임을 나타내는 Int32 값이 포함되어 있습니다.

C#

```
public static void PersistAsIntegers()
{
    SaveDatesAsInts();
    RestoreDatesAsInts();
}

private static void SaveDatesAsInts()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                         new DateTime(2014, 7, 10, 23, 49, 0),
                         new DateTime(2015, 1, 10, 1, 16, 0),
                         new DateTime(2014, 12, 20, 21, 45, 0),
                         new DateTime(2014, 6, 2, 15, 14, 0) };

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    var ticks = new long[dates.Length];
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        ticks[ctr] = dates[ctr].ToUniversalTime().Ticks;
    }
    var fs = new FileStream(filenameInts, FileMode.Create);
    var bw = new BinaryWriter(fs);
    bw.Write(ticks.Length);
    foreach (var tick in ticks)
        bw.Write(tick);

    bw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreDatesAsInts()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
```

```
    FileStream fs = new FileStream(filenameInts, FileMode.Open);
    BinaryReader br = new BinaryReader(fs);
    int items;
    DateTime[] dates;

    try
    {
        items = br.ReadInt32();
        dates = new DateTime[items];

        for (int ctr = 0; ctr < items; ctr++)
        {
            long ticks = br.ReadInt64();
            dates[ctr] = new DateTime(ticks).ToLocalTime();
        }
    }
    catch (EndOfStreamException)
    {
        Console.WriteLine("File corruption detected. Unable to restore
data...");
        return;
    }
    catch (IOException)
    {
        Console.WriteLine("Unspecified I/O error. Unable to restore
data...");
        return;
    }
    // Thrown during array initialization.
    catch (OutOfMemoryException)
    {
        Console.WriteLine("File corruption detected. Unable to restore
data...");
        return;
    }
    finally
    {
        br.Close();
    }

    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    foreach (var value in dates)
        Console.WriteLine(value.ToString("f"));

    Console.WriteLine("Restored dates...");
}
// When saved on an en-US system, the example displays the following output:
//       Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//       The dates on an en-US system:
//       Saturday, June 14, 2014 6:32 AM
//       Thursday, July 10, 2014 11:49 PM
//       Saturday, January 10, 2015 1:16 AM
//       Saturday, December 20, 2014 9:45 PM
//       Monday, June 02, 2014 3:14 PM
```

```
//       Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//       Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//       The dates on an en-GB system:
//       14 June 2014 14:32
//       11 July 2014 07:49
//       10 January 2015 09:16
//       21 December 2014 05:45
//       02 June 2014 23:14
//       Restored dates...
```

DateTime 값 직렬화

직렬화를 통해 스트림이나 파일에 DateTime 값을 저장할 수 있으며, 그런 다음 역직렬화를 통해 이 값을 복원할 수 있습니다. DateTime 데이터는 지정된 개체 형식으로 직렬화됩니다. 개체는 역직렬화될 때 복원됩니다. 포맷터 또는 직렬화기(예: JsonSerializer 또는 XmlSerializer)는 직렬화 및 디시리얼라이제이션 프로세스를 처리합니다. serialization 및 .NET에서 지원하는 serialization 유형에 대한 자세한 내용은 Serialization을 참조하세요.

다음 예제에서는 클래스를 XmlSerializer 사용하여 값을 직렬화하고 역직렬화합니다 DateTime . 값은 21세기의 모든 윤년 일을 나타냅니다. 현재 문화권이 영어(영국)인 시스템에서 예제가 실행되는 경우 출력은 결과를 나타냅니다. 개체 자체를 역직렬화 DateTime 했으므로 코드는 날짜 및 시간 형식의 문화권 차이를 처리할 필요가 없습니다.

C#

```
public static void PersistAsXML()
{
    // Serialize the data.
    var leapYears = new List<DateTime>();
    for (int year = 2000; year <= 2100; year += 4)
    {
        if (DateTime.IsLeapYear(year))
            leapYears.Add(new DateTime(year, 2, 29));
    }
    DateTime[] dateArray = leapYears.ToArray();

    var serializer = new XmlSerializer(dateArray.GetType());
    TextWriter sw = new StreamWriter(filenameXml);

    try
    {
        serializer.Serialize(sw, dateArray);
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine(e.InnerException?.Message);
```

```
    }
    finally
    {
        if (sw != null) sw.Close();
    }

    // Deserialize the data.
    DateTime[]? deserializedDates;
    using (var fs = new FileStream(filenameXml, FileMode.Open))
    {
        deserializedDates = (DateTime[]?)serializer.Deserialize(fs);
    }

    // Display the dates.
    Console.WriteLine($"Leap year days from 2000-2100 on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    int nItems = 0;
    if (deserializedDates is not null)
    {
        foreach (var dat in deserializedDates)
        {
            Console.Write($"   {dat:d}     ");
            nItems++;
            if (nItems % 5 == 0)
                Console.WriteLine();
        }
    }
}
// The example displays the following output:
//    Leap year days from 2000-2100 on an en-GB system:
//       29/02/2000       29/02/2004       29/02/2008       29/02/2012
29/02/2016
//       29/02/2020       29/02/2024       29/02/2028       29/02/2032
29/02/2036
//       29/02/2040       29/02/2044       29/02/2048       29/02/2052
29/02/2056
//       29/02/2060       29/02/2064       29/02/2068       29/02/2072
29/02/2076
//       29/02/2080       29/02/2084       29/02/2088       29/02/2092
29/02/2096
```

이전 예제에는 시간 정보가 포함되지 않습니다. 값이 DateTime 특정 시간을 나타내고 현지 시간으로 표현되는 경우 메서드를 호출 ToUniversalTime 하여 직렬화하기 전에 현지 시간에서 UTC로 변환합니다. 역직렬화한 후 메서드를 호출 ToLocalTime 하여 UTC에서 현지 시간으로 변환합니다.

DateTime과 TimeSpan 비교

DateTime 값 형식과 TimeSpan 값 형식은 DateTime가 시간의 순간을 나타내는 반면, TimeSpan는 시간 간격을 나타낸다는 점에서 다릅니다. 한 인스턴스를 DateTime 다른 인

스턴스에서 빼서 둘 사이의 시간 간격을 TimeSpan 나타내는 개체를 가져올 수 있습니다. 또는 현재 DateTime 에 긍정 TimeSpan 을 추가하여 미래 날짜를 나타내는 값을 가져올 DateTime 수 있습니다.

개체에서 DateTime 시간 간격을 추가하거나 뺄 수 있습니다. 시간 간격은 음수 또는 양수일 수 있으며 틱, 초 또는 개체와 같은 단위로 TimeSpan 표현할 수 있습니다.

허용 범위 내의 같음 비교

값에 대한 DateTime 같음 비교는 정확합니다. 같게 간주하려면 두 값을 같은 수의 틱으로 표현해야 합니다. 이러한 정밀도는 종종 많은 애플리케이션에서 불필요하거나 정확하지 않습니다. 종종 DateTime 개체가 **대략적으로 같은지** 테스트하려고 합니다.

다음 예제에서는 거의 동등한 DateTime 값을 비교하는 방법을 보여 줍니다. 같음으로 선언할 때 약간의 차이 여백을 허용합니다.

C#

```
public static bool RoughlyEquals(DateTime time, DateTime timeWithWindow, int
windowInSeconds, int frequencyInSeconds)
{
    long delta = (long)((TimeSpan)(timeWithWindow - time)).TotalSeconds %
frequencyInSeconds;
    delta = delta > windowInSeconds ? frequencyInSeconds - delta : delta;
    return Math.Abs(delta) < windowInSeconds;
}

public static void TestRoughlyEquals()
{
    int window = 10;
    int freq = 60 * 60 * 2; // 2 hours;

    DateTime d1 = DateTime.Now;

    DateTime d2 = d1.AddSeconds(2 * window);
    DateTime d3 = d1.AddSeconds(-2 * window);
    DateTime d4 = d1.AddSeconds(window / 2);
    DateTime d5 = d1.AddSeconds(-window / 2);

    DateTime d6 = (d1.AddHours(2)).AddSeconds(2 * window);
    DateTime d7 = (d1.AddHours(2)).AddSeconds(-2 * window);
    DateTime d8 = (d1.AddHours(2)).AddSeconds(window / 2);
    DateTime d9 = (d1.AddHours(2)).AddSeconds(-window / 2);

    Console.WriteLine($"d1 ({d1}) ~= d1 ({d1}): {RoughlyEquals(d1, d1,
window, freq)}");
    Console.WriteLine($"d1 ({d1}) ~= d2 ({d2}): {RoughlyEquals(d1, d2,
window, freq)}");
    Console.WriteLine($"d1 ({d1}) ~= d3 ({d3}): {RoughlyEquals(d1, d3,
window, freq)}");
```

```
    Console.WriteLine($"d1 ({d1}) ~= d4 ({d4}): {RoughlyEquals(d1, d4,
window, freq)}");
    Console.WriteLine($"d1 ({d1}) ~= d5 ({d5}): {RoughlyEquals(d1, d5,
window, freq)}");

    Console.WriteLine($"d1 ({d1}) ~= d6 ({d6}): {RoughlyEquals(d1, d6,
window, freq)}");
    Console.WriteLine($"d1 ({d1}) ~= d7 ({d7}): {RoughlyEquals(d1, d7,
window, freq)}");
    Console.WriteLine($"d1 ({d1}) ~= d8 ({d8}): {RoughlyEquals(d1, d8,
window, freq)}");
    Console.WriteLine($"d1 ({d1}) ~= d9 ({d9}): {RoughlyEquals(d1, d9,
window, freq)}");
}
// The example displays output similar to the following:
//    d1 (1/28/2010 9:01:26 PM) ~= d1 (1/28/2010 9:01:26 PM): True
//    d1 (1/28/2010 9:01:26 PM) ~= d2 (1/28/2010 9:01:46 PM): False
//    d1 (1/28/2010 9:01:26 PM) ~= d3 (1/28/2010 9:01:06 PM): False
//    d1 (1/28/2010 9:01:26 PM) ~= d4 (1/28/2010 9:01:31 PM): True
//    d1 (1/28/2010 9:01:26 PM) ~= d5 (1/28/2010 9:01:21 PM): True
//    d1 (1/28/2010 9:01:26 PM) ~= d6 (1/28/2010 11:01:46 PM): False
//    d1 (1/28/2010 9:01:26 PM) ~= d7 (1/28/2010 11:01:06 PM): False
//    d1 (1/28/2010 9:01:26 PM) ~= d8 (1/28/2010 11:01:31 PM): True
//    d1 (1/28/2010 9:01:26 PM) ~= d9 (1/28/2010 11:01:21 PM): True
```

COM interop 고려 사항

DateTime COM 애플리케이션으로 전송된 다음 관리되는 애플리케이션으로 다시 전송되는 값을 왕복이라고 합니다. 그러나 DateTime 시간만 지정하는 값은 예상한 대로 왕복하지 않습니다.

오후 3시와 같은 시간만 왕복하는 경우 최종 날짜와 시간은 1899년 12월 30일 오후 3시(0001년 1월 1일 오후 3:00)가 아닌 오후 3시입니다. NET 및 COM은 시간만 지정된 경우 기본 날짜를 가정합니다. 그러나 COM 시스템은 1899년 12월 30일 C.E.의 기본 날짜를 가정하고. .NET은 0001년 1월 1일의 기본 날짜를 가정합니다.

.NET에서 COM으로 시간만 전달되면 시간을 COM에서 사용하는 형식으로 변환하는 특수 처리가 수행됩니다. COM에서 .NET으로 시간만 전달되면 1899년 12월 30일 또는 그 이전에 합법적인 날짜와 시간이 손상되기 때문에 특별한 처리가 수행되지 않습니다. 날짜가 COM에서 왕복을 시작하는 경우 .NET 및 COM은 날짜를 유지합니다.

.NET 및 COM의 동작은 애플리케이션이 시간만을 지정하는 DateTime을 보내고 다시 받아올 때, 최종 DateTime 객체의 잘못된 날짜를 수정하거나 무시해야 함을 의미합니다.

System.DateTime.ToBinary 및 FromBinary 메서드

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

메서드를 ToBinary 사용하여 현재 DateTime 개체의 값을 이진 값으로 변환합니다. 이후에는 이진 값과 메서드를 FromBinary 사용하여 원래 DateTime 개체를 다시 만듭니다.

> ⓘ **Important**
>
> 경우에 따라 DateTime 메서드에서 반환된 FromBinary 값이 DateTime 메서드에 제공된 원래 ToBinary 값과 동일하지 않을 수 있습니다. 자세한 내용은 다음 섹션인 "현지 시간 조정"을 참조하세요.

구조체는 지정된 시간 값이 현지 시간, UTC(협정 세계시)를 기반으로 하는지 여부를 나타내는 개인 DateTime 필드와, 날짜와 시간을 나타내는 100나노초 틱 수를 포함하는 프라이빗 Kind 필드로 구성됩니다.

현지 시간 조정

현지 시간은 현지 표준 시간대로 조정된 협정 세계시로, DateTime 속성이 Kind인 Local 구조체로 나타납니다. 메서드에서 생성 DateTime 되는 이진 표현에서 로컬 ToBinary 값을 복원하는 경우 메서드 FromBinary 는 원래 값과 같지 않도록 다시 만든 값을 조정할 수 있습니다. 이 문제는 다음과 같은 조건에서 발생할 수 있습니다.

- 로컬 DateTime 개체가 ToBinary 메서드에 의해 한 표준 시간대에서 직렬화되었다가, FromBinary 메서드에 의해 다른 표준 시간대에서 역직렬화되면, 결과 DateTime 개체가 나타내는 현지 시간이 자동으로 두 번째 표준 시간대로 조정됩니다.

  예를 들어 오후 3시(현지 시간)를 나타내는 DateTime 개체를 고려하세요. 미국 태평양 표준 시간대에서 실행되는 애플리케이션은 메서드 ToBinary을(를) 사용하여 해당 개체를 DateTime 이진 값으로 변환합니다. 미국 동부 표준 시간대에서 실행되는 또 다른 애플리케이션은 이 메서드를 FromBinary 사용하여 이진 값을 새 DateTime 개체로 변환합니다. 새 DateTime 개체의 값은 오후 6시로, 원래 오후 3시 값과 동일한 시점을 나타내지만 동부 표준 시간대의 현지 시간으로 조정됩니다.

- 로컬 DateTime 값의 이진 표현이 호출되는 FromBinary 시스템의 현지 표준 시간대에서 잘못된 시간을 나타내는 경우 시간이 유효하게 조정됩니다.

예를 들어 표준시에서 일광 절약 시간제로의 전환은 2010년 3월 14일 오전 2:00에 미국의 태평양 표준 시간대에서 1시간이 지나면 오전 3:00까지 발생합니다. 이 시간 간격은 잘못된 시간, 즉 이 표준 시간대에 존재하지 않는 시간 간격입니다. 다음 예제에서는 이 범위 내에 있는 시간이 메서드에 의해 ToBinary 이진 값으로 변환된 다음 메서드에 의해 FromBinary 복원될 때 원래 값이 유효한 시간이 되도록 조정되는 것을 보여줍니다. 예제와 같이 특정 날짜 및 시간 값을 메서드에 전달 TimeZoneInfo.IsInvalidTime 하여 수정할 수 있는지 여부를 확인할 수 있습니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime localDate = new DateTime(2010, 3, 14, 2, 30, 0,
DateTimeKind.Local);
      long binLocal = localDate.ToBinary();
      if (TimeZoneInfo.Local.IsInvalidTime(localDate))
         Console.WriteLine($"{localDate} is an invalid time in the
{TimeZoneInfo.Local.StandardName} zone.");

      DateTime localDate2 = DateTime.FromBinary(binLocal);
      Console.WriteLine($"{localDate} = {localDate2}:
{localDate.Equals(localDate2)}");
   }
}
// The example displays the following output:
//    3/14/2010 2:30:00 AM is an invalid time in the Pacific Standard Time
zone.
//    3/14/2010 2:30:00 AM = 3/14/2010 3:30:00 AM: False
```

Last updated on 2026. 02. 12.

System.DateTime.TryParse 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 메서드는 DateTime.TryParse(String, IFormatProvider, DateTimeStyles, DateTime) 날짜, 시간 및 표준 시간대 정보를 포함할 수 있는 문자열을 구문 분석합니다. 메서드는 변환이 실패할 경우 예외를 발생시키지 않는다는 점에서 DateTime.Parse(String, IFormatProvider, DateTimeStyles) 메서드와 비슷하지만, DateTime.TryParse(String, DateTime) 메서드와 차이가 있습니다.

이 메서드는 인식할 수 없는 데이터를 무시하고 입력 문자열(`s`)을 완전히 구문 분석하려고 시도합니다. 시간이 포함되지만 날짜가 없는 경우 `s` 메서드는 기본적으로 현재 날짜를 대체하거나 플래그를 `styles` 포함하는 경우 NoCurrentDateDefault 대체합니다`DateTime.Date.MinValue`. 날짜가 포함되어 있지만 시간이 없는 경우 `s` 12:00 자정이 기본 시간으로 사용됩니다. 날짜가 있지만 해당 연도 구성 요소가 두 자리 숫자로만 구성된 경우 속성 값 `provider` 에 따라 매개 변수의 현재 달력에서 Calendar.TwoDigitYearMax 1년으로 변환됩니다. 선행, 내부 또는 후행 공백 문자는 `s`에서 무시됩니다. 날짜와 시간은 선행 및 후행 NUMBER SIGN 문자 쌍('#', U+0023)으로 괄호로 묶을 수 있으며 하나 이상의 NULL 문자(U+0000)로 후행할 수 있습니다.

날짜 및 시간 요소에 대한 특정 유효한 형식과 날짜 및 시간에 사용되는 이름과 기호는 매개 변수에 의해 `provider` 정의되며 다음 중 하나로 정의될 수 있습니다.

- CultureInfo 개체는 `s` 매개 변수에서 사용되는 서식의 문화권을 나타냅니다. DateTimeFormatInfo 개체는 CultureInfo.DateTimeFormat 속성에서 반환되며 `s`에서 사용되는 서식을 정의합니다.
- DateTimeFormatInfo 객체는 `s`에서 사용되는 서식을 정의합니다.
- 사용자 지정 IFormatProvider 구현입니다. 해당 IFormatProvider.GetFormat 메서드는 DateTimeFormatInfo 개체를 반환하며, 이는 `s`에서 사용되는 서식을 정의합니다.

`provider`가 `null`인 경우, 현재 문화권이 적용됩니다.

`s`가 현재 달력에서 윤년의 윤일을 문자열로 표현한 경우, 메서드가 `s`을 성공적으로 구문 분석합니다. 현재 달력`s`에서 윤년이 아닌 연도의 윤일 문자열 표현인 경우 `provider` 구문 분석 작업이 실패하고 메서드가 `false`를 반환합니다.

매개 변수는 `styles` 구문 분석된 문자열의 정확한 해석과 구문 분석 작업에서 처리해야 하는 방법을 정의합니다. 다음 표에 설명된 대로 열거형의 DateTimeStyles 하나 이상의 멤버일 수 있습니다.

테이블 확장

DateTimeStyles 멤버	설명
AdjustToUniversal	`s` 구문 분석하고 필요한 경우 UTC로 변환합니다. `s`에 표준 시간대 오프셋이 포함되어 있거나 `s`에 표준 시간대 정보가 없지만 `styles`가 DateTimeStyles.AssumeLocal 플래그를 포함하고 있는 경우, 이 메서드는 문자열을 구문 분석하고 ToUniversalTime을 호출하여 반환된 DateTime 값을 UTC로 변환하며, Kind 속성을 DateTimeKind.Utc로 설정합니다. `s`이 UTC를 나타내거나 `s`에 표준 시간대 정보가 없고 `styles`에 DateTimeStyles.AssumeUniversal 플래그가 포함되어 있는 경우, 메서드는 문자열을 구문 분석하고 반환된 DateTime 값에 대해 표준 시간대 변환을 수행하지 않으며 Kind 속성을 DateTimeKind.Utc로 설정합니다. 다른 모든 경우에서는 플래그가 적용되지 않습니다.
AllowInnerWhite	유효하지만 이 값은 무시됩니다. 날짜 및 시간 요소 `s`의 내부 공백이 허용됩니다.
AllowLeadingWhite	유효하지만 이 값은 무시됩니다. `s`의 날짜 및 시간 요소에는 선행 공백이 허용됩니다.
AllowTrailingWhite	유효하지만 이 값은 무시됩니다. 날짜 및 시간 요소인 `s`에서 후행 공백이 허용됩니다.
AllowWhiteSpaces	`s`에 선행, 내부 및 후행 공백이 포함될 수 있음을 명시합니다. 이 옵션은 기본 동작입니다. 와 같은 DateTimeStyles보다 제한적인 DateTimeStyles.None 열거형 값을 제공하여 재정의할 수 없습니다.
AssumeLocal	표준 시간대 정보가 없는 경우 `s` 현지 시간을 나타내는 것으로 간주되도록 지정합니다. DateTimeStyles.AdjustToUniversal 플래그가 존재하지 않는 경우 반환된 Kind 값의 DateTime 속성은 DateTimeKind.Local으로 설정됩니다.
AssumeUniversal	표준 시간대 정보가 없는 경우 `s` UTC를 나타내는 것으로 간주되도록 지정합니다. 플래그가 DateTimeStyles.AdjustToUniversal 없는 한 메서드는 반환 DateTime 된 값을 UTC에서 현지 시간으로 변환하고 해당 Kind 속성을 DateTimeKind.Local.로 설정합니다.
None	유효하지만 이 값은 무시됩니다.
RoundtripKind	표준 시간대 정보를 포함하는 문자열의 경우, 날짜 및 시간 문자열을 DateTime 값으로, Kind 속성이 DateTimeKind.Local로 설정된 상태로 변환하지 않도록 시도합니다. 일반적으로 이러한 문자열은 "o", "r" 또는 "u" 표준 형식 지정자를 사용하여 메서드를 호출 DateTime.ToString(String) 하여 생성됩니다.

`s`에 표준 시간대 정보가 포함되지 않은 경우, DateTime.TryParse(String, IFormatProvider, DateTimeStyles, DateTime) 메서드는 플래그 DateTime가 그렇지 않음을 나타내지 않는 한 Kind 속성이 DateTimeKind.Unspecified인 `styles` 값을 반환합니다. 표준 시간대 또는 표준 시간대 오프셋 정보가 포함된 경우 `s` 메서드는 DateTime.TryParse(String, IFormatProvider, DateTimeStyles, DateTime) 필요한 시간 변환을 수행하고 다음 중 하나를 반환합니다.

- DateTime은 날짜와 시간이 현지 시간을 반영하고 Kind 속성이 DateTimeKind.Local인 값입니다.

- 또는 `styles`에 AdjustToUniversal 플래그가 포함된 경우, 날짜와 시간이 UTC 기준으로 한 DateTime 값이며 Kind 속성이 DateTimeKind.Utc입니다.

이 동작은 DateTimeStyles.RoundtripKind 플래그를 사용하여 재정의할 수 있습니다.

사용자 지정 문화권 구문 분석

사용자 지정 문화권에 대해 생성된 날짜 및 시간 문자열을 구문 분석하는 경우 TryParseExact 메서드를 사용하여 TryParse 메서드 대신 구문 분석 작업이 성공할 가능성을 높입니다. 사용자 지정 문화권 날짜 및 시간 문자열은 복잡하고 구문 분석하기 어려울 수 있습니다. 이 메서드는 TryParse 여러 암시적 구문 분석 패턴으로 문자열을 구문 분석하려고 시도하며 모두 실패할 수 있습니다. 반면, 메서드를 TryParseExact 사용하려면 성공할 가능성이 있는 하나 이상의 정확한 구문 분석 패턴을 명시적으로 지정해야 합니다.

사용자 지정 문화권에 대한 자세한 내용은 클래스를 참조하세요 System.Globalization.CultureAndRegionInfoBuilder .

System.TimeSpan 구조체

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

시간 객체 TimeSpan 는 양수 또는 음수 형태로 일, 시간, 분, 초 및 초의 부분 단위로 측정되는 시간 간격(지속 시간 또는 경과된 시간)을 나타냅니다. 이 구조체는 TimeSpan 특정 날짜와 관련이 없는 경우에만 하루 중 시간을 나타내는 데 사용할 수 있습니다. 그렇지 않으면 구조체 DateTime 를 DateTimeOffset 대신 사용해야 합니다. (구조를 사용하여 TimeSpan 하루 중 시간을 반영하는 방법에 대한 자세한 내용은 DateTime, DateTimeOffset, TimeSpan 및 TimeZoneInfo 중에서 선택 항목을 참조하세요.)

> ⓘ **참고**
>
> 값은 **TimeSpan** 시간 간격을 나타내며 특정 일 수, 시간, 분, 초 및 밀리초로 표현할 수 있습니다. 특정 시작점이나 끝점에 대한 참조 없이 일반적인 간격을 나타내므로 연도 및 월로 표현할 수 없으며 둘 다 일 수가 가변적입니다. **DateTime** 값은 특정 시간대를 참조하지 않는 날짜 및 시간을 나타내는 것과, **DateTimeOffset** 값은 특정 시간을 나타내는 것과 다릅니다.

구조에서 기간을 측정하는 데 사용하는 가장 큰 시간 TimeSpan 단위는 일입니다. 시간 간격은 월 및 연도와 같이 더 큰 시간 단위의 일 수가 다르기 때문에 일관성을 위해 일 단위로 측정됩니다.

개체의 TimeSpan 값은 표시된 시간 간격과 같은 틱 수입니다. 틱은 100나노초 또는 1초의 1,000만 분의 1과 같습니다. 개체의 값의 범위는 TimeSpan부터 TimeSpan.MinValue까지입니다.

TimeSpan 값 인스턴스화

다음과 같은 TimeSpan 여러 가지 방법으로 값을 인스턴스화할 수 있습니다.

- 암시적 매개 변수 없는 생성자를 호출합니다. 다음 예제와 같이 값이 TimeSpan.Zero있는 개체를 만듭니다.

  ```
  C#
  ```

  ```
  TimeSpan interval = new TimeSpan();
  Console.WriteLine(interval.Equals(TimeSpan.Zero));    // Displays "True".
  ```

- 명시적 생성자 중 하나를 호출합니다. 다음 예제에서는 지정된 시간, 분 및 초 수로 값을 초기화 TimeSpan 합니다.

C#

```
TimeSpan interval = new TimeSpan(2, 14, 18);
Console.WriteLine(interval.ToString());

// Displays "02:14:18".
```

- 메서드를 호출하거나 TimeSpan 값을 반환하는 작업을 수행하여. 예를 들어 다음 예제와 같이 두 날짜 값과 시간 값 사이의 간격을 나타내는 값을 인스턴스화 TimeSpan 할 수 있습니다.

C#

```
DateTime departure = new DateTime(2010, 6, 12, 18, 32, 0);
DateTime arrival = new DateTime(2010, 6, 13, 22, 47, 0);
TimeSpan travelTime = arrival - departure;
Console.WriteLine($"{arrival} - {departure} = {travelTime}");

// The example displays the following output:
//       6/13/2010 10:47:00 PM - 6/12/2010 6:32:00 PM = 1.04:15:00
```

다음 예제와 TimeSpan 같이 이러한 방식으로 개체를 0 시간 값으로 초기화할 수도 있습니다.

C#

```
Random rnd = new Random();

TimeSpan timeSpent = TimeSpan.Zero;

timeSpent += GetTimeBeforeLunch();
timeSpent += GetTimeAfterLunch();

Console.WriteLine($"Total time: {timeSpent}");

TimeSpan GetTimeBeforeLunch()
{
    return new TimeSpan(rnd.Next(3, 6), 0, 0);
}

TimeSpan GetTimeAfterLunch()
{
    return new TimeSpan(rnd.Next(3, 6), 0, 0);
}

// The example displays output like the following:
//        Total time: 08:00:00
```

TimeSpan값은 산술 연산자와 DateTime, DateTimeOffset, TimeSpan 구조체의 메서드에 의해 반환됩니다.

- TimeSpan 값의 문자열 표현을 구문 분석하여. Parse 및 TryParse 메서드를 사용하여 시간 간격이 포함된 문자열을 TimeSpan 값으로 변환할 수 있습니다. 다음 예제에서는 메서드를 Parse 사용하여 문자열 배열을 값으로 TimeSpan 변환합니다.

```
C#
```

```
string[] values = { "12", "31.", "5.8:32:16", "12:12:15.95", ".12"};
foreach (string value in values)
{
   try {
      TimeSpan ts = TimeSpan.Parse(value);
      Console.WriteLine($"'{value}' --> {ts}");
   }
   catch (FormatException) {
      Console.WriteLine($"Unable to parse '{value}'");
   }
   catch (OverflowException) {
      Console.WriteLine($"'{value}' is outside the range of a TimeSpan.");
   }
}

// The example displays the following output:
//       '12' --> 12.00:00:00
//       Unable to parse '31.'
//       '5.8:32:16' --> 5.08:32:16
//       '12:12:15.95' --> 12:12:15.9500000
//       Unable to parse '.12'
```

또한, TimeSpan 또는 ParseExact 메서드를 호출하여 구문 분석하고 값으로 변환할 입력 문자열의 정확한 형식을 정의할 수 있습니다.

TimeSpan 값에 대한 작업 수행

Addition 및 Subtraction 연산자를 사용하거나 Add 및 Subtract 메서드를 호출하여 시간을 추가하거나 뺄 수 있습니다. 및 메서드를 호출CompareCompareTo하여 두 시간 기간을 비교할 수도 있습니다Equals. 이 TimeSpan 구조에는 시간 간격을 양수 값과 음수 값으로 변환하는 Duration 및 Negate 메서드가 포함됩니다.

값의 범위는 TimeSpan, MinValue 입니다.

TimeSpan 값 서식 지정

TimeSpan 값은 [-]*d.hh*:*mm*:*ss.ff* 형식으로 나타낼 수 있습니다. 여기서 선택적인 빼기 기호는 음수 시간 간격을 나타냅니다. *d* 구성 요소는 일, *hh*는 24시간제로 측정한 시간, *mm*는 분, *ss*는 초, *ff*는 초의 분수를 나타냅니다. 즉, 시간 간격은 하루 중 시간이 없는 양수 또는 음수 일 수 또는 하루 중 시간이 있는 일 수 또는 하루 중 시간만 있는 일 수로 구성됩니다.

.NET Framework 4부터는 TimeSpan 구조체가 ToString 메서드 오버로드를 통해 값을 문자열 표현으로 변환할 때, 문화권별 형식을 지원합니다 TimeSpan. 기본 TimeSpan.ToString() 메서드는 이전 버전의 .NET Framework에서 반환 값과 동일한 고정 형식을 사용하여 시간 간격을 반환합니다. TimeSpan.ToString(String) 오버로드를 사용하면 시간 간격의 문자열 표현을 정의하는 형식 문자열을 지정할 수 있습니다. TimeSpan.ToString(String, IFormatProvider) 오버로드를 사용하면 형식 문자열 및 해당 형식 규칙이 시간 간격의 문자열 표현을 만드는 데 사용되는 문화권을 지정할 수 있습니다. TimeSpan 는 표준 및 사용자 지정 형식 문자열을 모두 지원합니다. (자세한 내용은 표준 TimeSpan 형식 문자열 및사용자 지정 TimeSpan 형식 문자열을 참조하세요.) 그러나 표준 형식 문자열만 문화권을 구분합니다.

레거시 TimeSpan 서식 복원

경우에 따라 .NET Framework 3.5 및 이전 버전에서 값의 서식을 성공적으로 지정 TimeSpan 하는 코드가 .NET Framework 4에서 실패합니다. 이는 TimeSpan_LegacyFormatMode< 요소 메서드를> 호출하여 형식 문자열을 사용하여 값의 형식을 TimeSpan 지정하는 코드에서 가장 일반적입니다. 다음 예제는 .NET Framework 3.5 및 이전 버전에서는 TimeSpan 값의 서식을 성공적으로 지정하지만, .NET Framework 4 이상 버전에서는 예외를 발생시킵니다. 주의할 점은, .NET Framework 3.5 및 이전 버전에서는 지원되지 않는 형식 지정자를 사용하여 TimeSpan 값을 서식화하려고 시도하지만 무시된다는 것입니다.

C#

```
ShowFormattingCode();
// Output from .NET Framework 3.5 and earlier versions:
//       12:30:45
// Output from .NET Framework 4:
//       Invalid Format

Console.WriteLine("---");

ShowParsingCode();
// Output:
//       000000006 --> 6.00:00:00

void ShowFormattingCode()
{
    TimeSpan interval = new TimeSpan(12, 30, 45);
    string output;
    try
    {
        output = String.Format("{0:r}", interval);
```

```
    }
    catch (FormatException)
    {
        output = "Invalid Format";
    }
    Console.WriteLine(output);
}

void ShowParsingCode()
{
    string value = "000000006";
    try
    {
        TimeSpan interval = TimeSpan.Parse(value);
        Console.WriteLine($"{value} --> {interval}");
    }
    catch (FormatException)
    {
        Console.WriteLine($"{value}: Bad Format");
    }
    catch (OverflowException)
    {
        Console.WriteLine($"{value}: Overflow");
    }
}
```

코드를 수정할 수 없는 경우 다음 방법 중 하나로 값의 TimeSpan 레거시 서식을 복원할 수 있습니다.

- TimeSpan_LegacyFormatMode 요소가 포함된 구성 파일을 만듭니다<.> 이 요소의 `enabled` 특성을 `true`로 설정하면 애플리케이션별로 레거시 TimeSpan 서식이 복원됩니다.

- 애플리케이션 도메인을 만들 때 "NetFx40_TimeSpanLegacyFormatMode" 호환성 스위치를 설정합니다. 이렇게 하면 애플리케이션 도메인별로 레거시 TimeSpan 서식을 지정할 수 있습니다. 다음 예제에서는 레거시 서식을 사용하는 애플리케이션 도메인을 TimeSpan 만듭니다.

  C#

```
using System;

public class Example2
{
    public static void Main()
    {
        AppDomainSetup appSetup = new AppDomainSetup();
        appSetup.SetCompatibilitySwitches(new string[] {
"NetFx40_TimeSpanLegacyFormatMode" });
        AppDomain legacyDomain = AppDomain.CreateDomain("legacyDomain",
                                                        null, appSetup);
        legacyDomain.ExecuteAssembly("ShowTimeSpan.exe");
```

```
    }
}
```

다음 코드가 새 애플리케이션 도메인에서 실행되면 레거시 TimeSpan 서식 지정 동작으로 되돌아갑니다.

C#

```
using System;

public class Example3
{
   public static void Main()
   {
      TimeSpan interval = DateTime.Now - DateTime.Now.Date;
      string msg = String.Format("Elapsed Time Today: {0:d} hours.",
                                 interval);
      Console.WriteLine(msg);
   }
}
// The example displays the following output:
//       Elapsed Time Today: 01:40:52.2524662 hours.
```

System.TimeSpan.Parse 메서드

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

메서드에 대한 Parse 입력 문자열에는 다음과 같은 형식의 시간 간격 사양이 포함됩니다.

```
[ws][-]{ d | [d.]hh:mm[:ss[.ff]] }[ws]
```

대괄호(`[` 및 `]`)의 요소는 선택 사항입니다. 중괄호(`{` 및 `}`)로 묶이고 세로 막대(|)로 구분된 대체 목록에서 하나를 선택해야 합니다. 다음 표에서는 각 요소에 대해 설명합니다.

테이블 확장

요소	설명
ws	선택적 공백
-	음수임을 나타내는 선택적인 빼기 기호 TimeSpan입니다.
d	0에서 10675199 사이의 일 수입니다.
.	일과 시간을 구분하는 문화권 구분 기호입니다. 고정 형식은 마침표(".") 문자를 사용합니다.
hh	시간(0에서 23까지)입니다.
:	문화권에 민감한 시간 구분 기호입니다. 고정 형식은 콜론(":") 문자를 사용합니다.
mm	분(0에서 59까지)
ss	0에서 59까지의 선택적 초입니다.
.	초와 초의 분수를 구분하는 문화적 차이에 민감한 기호입니다. 고정 형식은 마침표(".") 문자를 사용합니다.
ff	1~7자리의 소수 자릿수로 구성된 선택적인 소수 초입니다.

입력 문자열이 일 값만 아닌 경우 시간 및 분 구성 요소를 포함해야 합니다. 다른 구성 요소는 선택 사항입니다. 이 값이 있는 경우 각 시간 구성 요소의 값은 지정된 범위 내에 있어야 합니다. 예를 들어 시간 구성 요소인 *hh*의 값은 0에서 23 사이여야 합니다. 이 때문에 메서드에 "23:00:00"을 전달하면 Parse 23시간의 시간 간격이 반환됩니다. 반면에 "24:00:00"을 전달하면 24일의 시간 간격이 반환됩니다. "24"는 시간 구성 요소 범위를 벗어나므로 일 구성 요소로 해석됩니다.

입력 문자열의 구성 요소는 TimeSpan.MinValue보다 크거나 같고 TimeSpan.MaxValue보다 작거나 같은 시간 간격을 집합적으로 지정해야 합니다.

이 메서드는 Parse(String) 현재 문화권에 대한 각 문화권별 형식을 사용하여 입력 문자열을 구문 분석하려고 합니다.

호출자에 대한 참고 사항

구문 분석할 문자열의 시간 간격 구성 요소에 7자리 이상의 숫자가 포함된 경우 .NET Framework 3.5 및 이전 버전의 구문 분석 작업은 .NET Framework 4 이상 버전의 구문 분석 작업과 다르게 동작할 수 있습니다. 경우에 따라 .NET Framework 3.5 및 이전 버전에서 성공한 파싱 작업이 .NET Framework 4 이상에서는 실패하고 OverflowException를 던질 수 있습니다. 다른 경우에는 .NET Framework 3.5 및 이전 버전에서 throw FormatException 하는 구문 분석 작업이 실패하고 .NET Framework 4 이상에서 throw OverflowException 할 수 있습니다. 다음 예제에서는 두 시나리오를 모두 보여 줍니다.

C#

```
string[] values = { "000000006", "12.12:12:12.12345678" };
foreach (string value in values)
{
   try {
      TimeSpan interval = TimeSpan.Parse(value);
      Console.WriteLine($"{value} --> {interval}");
   }
   catch (FormatException) {
      Console.WriteLine($"{value}: Bad Format");
   }
   catch (OverflowException) {
      Console.WriteLine($"{value}: Overflow");
   }
}

// Output from .NET Framework 3.5 and earlier versions:
//       000000006 --> 6.00:00:00
//       12.12:12:12.12345678: Bad Format
// Output from .NET Framework 4 and later versions or .NET Core:
//       000000006: Overflow
//       12.12:12:12.12345678: Overflow
```

System.TimeSpan.TryParse 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

TryParse(System.String,System.TimeSpan@) 메서드

TimeSpan.TryParse(String, TimeSpan) 메서드는 TimeSpan.Parse(String) 메서드와 비슷하지만, 변환이 실패할 경우 예외를 발생시키지 않습니다.

매개 변수에는 `s` 다음과 같은 형식의 시간 간격 사양이 포함됩니다.

```
[ws][-]{ d | d.hh:mm[:ss[.ff]] | hh:mm[:ss[.ff]] }[ws]
```

대괄호([및])의 요소는 선택 사항입니다. 중괄호({ 및 })로 묶고 세로 막대(|)로 구분된 대체 항목 목록에서 하나를 선택해야 합니다. 다음 표에서는 각 요소에 대해 설명합니다.

테이블 확장

요소	설명
ws	선택적 공백
-	음수임을 나타내는 선택적인 빼기 기호 TimeSpan입니다.
d	0에서 10675199 사이의 일 수입니다.
.	일과 시간을 구분하는 문화권 구분 기호입니다. 고정 형식은 마침표(".") 문자를 사용합니다.
hh	시간(0에서 23까지)입니다.
:	문화권에 민감한 시간 구분 기호입니다. 고정 형식은 콜론(":") 문자를 사용합니다.
밀리미터	분(0에서 59까지)
ss	0에서 59까지의 선택적 초입니다.
.	초와 초의 분수를 구분하는 문화적 차이에 민감한 기호입니다. 고정 형식은 마침표(".") 문자를 사용합니다.
Ff 로	1~7자리의 소수 자릿수로 구성된 선택적인 소수 초입니다.

구성 요소의 시간 간격은 `s` 이상이고 TimeSpan.MinValue 이하로 집합적으로 지정되어야 합니다.

이 메서드는 Parse(String) 현재 문화권 `s` 에 대한 각 문화권별 형식을 사용하여 구문 분석을 시도합니다.

TryParse(String, IFormatProvider, TimeSpan) 메서드

TryParse(String, IFormatProvider, TimeSpan) 메서드는 변환이 실패할 경우 예외를 throw하지 않는다는 점을 제외하고 Parse(String, IFormatProvider) 메서드와 유사합니다.

매개 변수에는 `input` 다음과 같은 형식의 시간 간격 사양이 포함됩니다.

```
[ws][-]{ d | d.hh:mm[:ss[.ff]] | hh:mm[:ss[.ff]] }[ws]
```

대괄호([및])의 요소는 선택 사항입니다. 중괄호({ 및 })로 묶고 세로 막대(|)로 구분된 대체 항목 목록에서 하나를 선택해야 합니다. 다음 표에서는 각 요소에 대해 설명합니다.

테이블 확장

요소	설명
ws	선택적 공백
-	음수임을 나타내는 선택적인 빼기 기호 TimeSpan입니다.
d	0에서 10675199 사이의 일 수입니다.
.	일과 시간을 구분하는 문화권 구분 기호입니다. 고정 형식은 마침표(".") 문자를 사용합니다.
hh	시간(0에서 23까지)입니다.
:	문화권에 민감한 시간 구분 기호입니다. 고정 형식은 콜론(":") 문자를 사용합니다.
밀리미터	분(0에서 59까지)
ss	0에서 59까지의 선택적 초입니다.
.	초와 초의 분수를 구분하는 문화적 차이에 민감한 기호입니다. 고정 형식은 마침표(".") 문자를 사용합니다.
Ff 로	1~7자리의 소수 자릿수로 구성된 선택적인 소수 초입니다.

구성 요소는 `input` 보다 크거나 같고 TimeSpan.MinValue 및 TimeSpan.MaxValue보다 작거나 같은 시간 간격을 집합적으로 지정해야 합니다.

메서드 TryParse(String, IFormatProvider, TimeSpan)는 `input` 로 지정된 문화권에 대해 각 문화권별 형식을 사용하여 `formatProvider` 의 구문 분석을 시도합니다.

`formatProvider` 매개 변수는 반환된 IFormatProvider 문자열의 형식에 대한 문화권별 정보를 제공하는 구현입니다. 매개 변수는 `formatProvider` 다음 중 어느 것일 수 있습니다.

- CultureInfo 반환된 문자열에 적용되는 서식 규칙을 가진 문화권을 나타내는 개체입니다. DateTimeFormatInfo 속성에서 반환된 CultureInfo.DateTimeFormat 개체는 반환된 문자열의 서식을 정의합니다.
- DateTimeFormatInfo 반환된 문자열의 서식을 정의하는 개체입니다.
- 인터페이스를 구현하는 사용자 지정 개체입니다 IFormatProvider . 해당 메서드는 IFormatProvider.GetFormatDateTimeFormatInfo 서식 정보를 제공하는 개체를 반환합니다.

`formatProvider`이 `null`이면, 현재 문화권과 연결된 DateTimeFormatInfo 개체가 사용됩니다.

호출자에 대한 참고 사항

구문 분석할 문자열의 시간 간격 구성 요소에 7자리 이상의 숫자가 포함된 경우 .NET Framework 3.5 및 이전 버전에서 성공하고 반환 `true` 된 구문 분석 작업이 실패하고 .NET Framework 4 이상 버전에서 반환 `false` 될 수 있습니다. 다음 예제에서는 이 시나리오를 보여 줍니다.

C#

```
string value = "000000006";
TimeSpan interval;
if (TimeSpan.TryParse(value, out interval))
   Console.WriteLine($"{value} --> {interval}");
else
   Console.WriteLine($"Unable to parse '{value}'");

// Output from .NET Framework 3.5 and earlier versions:
//       000000006 --> 6.00:00:00
// Output from .NET Framework 4:
//       Unable to parse //000000006//
```

특성을 사용하여 메타데이터 확장

2025. 06. 17.

공용 언어 런타임에서는 형식, 필드, 메서드 및 속성과 같은 프로그래밍 요소에 주석을 달기 위해 특성이라는 키워드 방식의 설명적 선언을 추가할 수 있습니다. 런타임에 대한 코드를 컴파일하면 CIL(공용 중간 언어)로 변환되고 컴파일러에서 생성된 메타데이터와 함께 PE(이식 가능한 실행 파일) 파일 내에 배치됩니다. 특성을 사용하면 런타임 리플렉션 서비스를 사용하여 추출할 수 있는 메타데이터에 추가 설명 정보를 배치할 수 있습니다. 특수 클래스가 System.Attribute에서 파생될 때, 해당 인스턴스를 선언하면 컴파일러가 특성을 만듭니다.

.NET은 다양한 이유로 특성을 사용하고 다양한 문제를 해결합니다. 특성은 데이터를 직렬화하고, 보안을 적용하는 데 사용되는 특성을 지정하고, 코드를 디버그하기 쉽게 유지하도록 JIT(Just-In-Time) 컴파일러에서 최적화를 제한하는 방법을 설명합니다. 또한 특성은 파일의 이름이나 코드 작성자를 기록하거나 양식 개발 중에 컨트롤 및 멤버의 표시 여부를 제어할 수 있습니다.

관련 문서

테이블 확장

제목	설명
특성 적용	코드 요소에 특성을 적용하는 방법을 설명합니다.
사용자 지정 특성 작성	사용자 지정 특성 클래스를 디자인하는 방법을 설명합니다.
속성에 저장된 정보 검색	실행 컨텍스트에 로드되는 코드에 대한 사용자 지정 특성을 검색하는 방법을 설명합니다.
메타데이터 및 Self-Describing 구성 요소	메타데이터의 개요를 제공하고 .NET PE(이식 가능한 실행 파일) 파일에서 구현되는 방법을 설명합니다.
방법: Reflection-Only 컨텍스트에 어셈블리 로드	리플렉션 전용 컨텍스트에서 사용자 지정 특성 정보를 검색하는 방법을 설명합니다.

참고 문헌

- System.Attribute

특성 적용

다음 프로세스를 사용하여 코드 요소에 특성을 적용합니다.

1. 새 특성을 정의하거나 기존 .NET 특성을 사용합니다.
2. 요소 바로 앞에 배치하여 코드 요소에 특성을 적용합니다.

   각 언어에는 고유한 특성 구문이 있습니다. C++ 및 C#에서 특성은 대괄호로 둘러싸여 있으며 줄 바꿈을 포함할 수 있는 공백으로 요소와 구분됩니다. Visual Basic에서 속성은 꺾쇠 괄호로 둘러싸여 있으며, 동일한 논리적인 줄에 있어야 합니다. 줄 바꿈이 필요한 경우, 줄 연속 문자를 사용할 수 있습니다.
3. 특성에 대한 위치 매개 변수 및 명명된 매개 변수를 지정합니다.

   위치 매개 변수는 필수이며 명명된 매개 변수 앞에 와야 합니다. 특성 생성자 중 하나의 매개 변수에 해당합니다. *명명된* 매개 변수는 선택 사항이며 특성의 읽기/쓰기 속성에 해당합니다. C++와 C#에서는 각 선택적 매개변수에 대해 `name=value`를 지정합니다. 여기서 `name`은 속성의 이름입니다. Visual Basic에서 `name:=value`을(를) 지정하십시오.

코드를 컴파일할 때 특성이 메타데이터로 내보내지고 런타임 리플렉션 서비스를 통해 공용 언어 런타임 및 사용자 지정 도구 또는 애플리케이션에서 사용할 수 있습니다.

규칙에 따라 모든 특성 이름은 "특성"으로 끝납니다. 그러나 Visual Basic 및 C#과 같이 런타임을 대상으로 하는 여러 언어에서는 특성의 전체 이름을 지정할 필요가 없습니다. 예를 들어 초기화 System.ObsoleteAttribute하려는 경우 **사용되지** 않는 것으로만 참조하면 됩니다.

유효한 특성 인수

특성에 인수를 전달하는 경우 다음 종류의 식 중 하나를 사용합니다.

- 상수 식(리터럴, `const`/`Const` 값 및 열거형 값).
- 형식 식(C#의 경우 `typeof`, Visual Basic `GetType`).
- 컴파일 시간에 문자열 상수가 생성되는 이름 식(C#의 경우 `nameof`, Visual Basic `NameOf`)입니다.
- 이전 식만 요소 값으로 사용하는 특성 매개 변수 형식의 배열 생성 식입니다.

다음 형식은 특성 매개 변수 형식으로 유효합니다.

- 단순 형식(C# 키워드/Visual Basic 키워드/.NET 런타임 형식):

테이블 확장

C#	Visual Basic	.NET 런타임 유형
`bool`	`Boolean`	Boolean
`byte`	`Byte`	Byte
`char`	`Char`	Char
`double`	`Double`	Double
`float`	`Single`	Single
`int`	`Integer`	Int32
`long`	`Long`	Int64
`short`	`Short`	Int16
`string`	`String`	String

- `object` (C#에서 값이 유효한 특성 인수 형식 또는 1차원 배열 중 하나인 경우).
- Type;
- 사용 사이트에서 액세스할 수 있는 열거형 형식입니다.
- 이전 형식의 1차원 배열입니다.

> **참고 항목**
>
> 리터럴 상수를 지원하더라도 형식 `sbyte`, `ushort`, `uint`, `ulong` `decimal`, 및 `nint` `nuint` 유효한 특성 매개 변수 형식이 아닙니다.

다음 예제에서는 유효한 특성 인수를 보여 줍니다.

C#

```
[MyAttr(true)]                          // bool literal
[MyAttr(42)]                            // int literal
[MyAttr("hello")]                       // string literal
[MyAttr(MyEnum.Value)]                  // enum value
[MyAttr(typeof(string))]                // typeof expression
[MyAttr(nameof(MyClass))]               // nameof expression (string constant)
[MyAttr(new int[] { 1, 2, 3 })]         // array of constants
[MyAttr(new string[] { "a", "b" })]     // array of strings
```

다음 예제에서는 컴파일러 오류를 일으키는 인수를 보여 줍니다.

C#

```
string value = "test";
[MyAttr(value)]        // Error CS0182: not a constant expression
[MyAttr(GetValue())]   // Error CS0182: method calls aren't allowed
```

메서드에 특성 적용

다음 코드 예제는 System.ObsoleteAttribute를 사용하여 코드를 사용되지 않는 것으로 표시하는 방법을 보여 줍니다. 문자열 `"Will be removed in next version"` 이 특성에 전달됩니다. 이 특성은 특성이 설명하는 코드가 호출될 때 전달된 문자열을 표시하는 컴파일러 경고를 발생합니다.

C#

```
public class Example
{
    // Specify attributes between square brackets in C#.
    // This attribute is applied only to the Add method.
    [Obsolete("Will be removed in next version.")]
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}

class Test
{
    public static void Main()
    {
        // This generates a compile-time warning.
        int i = Example.Add(2, 2);
    }
}
```

어셈블리 수준에서 특성 적용

어셈블리 수준에서 특성을 적용하려면 `assembly`(Visual Basic에서는 `Assembly`) 키워드를 사용합니다. 다음 코드는 어셈블리 수준에서 적용된 것을 보여줍니다 AssemblyTitleAttribute .

C#

```
using System.Reflection;
[assembly:AssemblyTitle("My Assembly")]
```

이 특성을 적용하면 문자열 `"My Assembly"` 이 파일의 메타데이터 부분에 있는 어셈블리 매니페스트에 배치됩니다. IL 디스어셈블러(Ildasm.exe)를 사용하거나 특성을 검색하는 사용자 지정 프

로그램을 만들어 특성을 볼 수 있습니다.

참고하십시오

- 특성
- 속성에 저장된 정보 검색
- 개념
- 특성(C#)
- 속성 개요 (Visual Basic)

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

사용자 지정 특성 작성

사용자 지정 특성을 디자인하기 위해 많은 새로운 개념을 배울 필요가 없습니다. 개체 지향 프로그래밍에 익숙하고 클래스를 디자인하는 방법을 알고 있다면 필요한 대부분의 지식이 이미 있습니다. 사용자 지정 특성은 System.Attribute 클래스로부터 직접 또는 간접적으로 파생된 전통적인 클래스입니다. 기존 클래스와 마찬가지로 사용자 지정 특성에는 데이터를 저장하고 검색하는 메서드가 포함됩니다.

사용자 지정 특성 클래스를 올바르게 디자인하는 기본 단계는 다음과 같습니다.

- AttributeUsageAttribute를 적용하기
- 특성 클래스 선언
- 생성자 선언
- 속성 선언

이 섹션에서는 이러한 각 단계를 설명하고 사용자 지정 특성 예제로 마무리합니다.

AttributeUsageAttribute를 적용하기

사용자 지정 특성 선언은 System.AttributeUsageAttribute 특성 클래스의 주요 특성 중 일부를 정의하는 특성으로 시작합니다. 예를 들어 특성을 다른 클래스에서 상속할 수 있는지 또는 특성을 적용할 수 있는 요소를 지정할 수 있습니다. 다음 코드 조각은 AttributeUsageAttribute을 사용하는 방법을 보여 줍니다.

C#

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

AttributeUsageAttribute 사용자 지정 특성을 만드는 데 중요한 세 가지 멤버인 AttributeTargets, Inherited 및 AllowMultiple이 있습니다.

AttributeTargets 멤버

앞의 예제 AttributeTargets.All 에서는 이 특성을 모든 프로그램 요소에 적용할 수 있음을 나타내는 지정됩니다. 또는 특성을 클래스에만 적용할 수 있음을 나타내거나 AttributeTargets.Class메서드에만 특성을 적용할 수 있음을 나타내는 것을 지정할 AttributeTargets.Method수 있습니다. 모든 프로그램 요소는 이러한 방식으로 사용자 지정 특성에 의해 설명을 표시할 수 있습니다.

여러 AttributeTargets 값을 전달할 수도 있습니다. 다음 코드 조각은 사용자 지정 특성을 모든 클래스 또는 메서드에 적용할 수 있도록 지정합니다.

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
```

상속된 프로퍼티

이 속성은 AttributeUsageAttribute.Inherited 특성이 적용되는 클래스에서 파생된 클래스에서 특성을 상속할 수 있는지 여부를 나타냅니다. 이 속성은 (기본값) 또는 `true` 플래그를 사용합니다 `false` . 다음 예제에서 `MyAttribute`는 기본 Inherited 값으로 `true`을, `YourAttribute`는 Inherited 값으로 `false`을 가집니다.

C#

```
// This defaults to Inherited = true.
public class MyAttribute : Attribute
{
    //...
}

[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
    //...
}
```

그런 다음 두 특성이 기본 클래스 `MyClass`의 메서드에 적용됩니다.

C#

```
public class MyClass
{
    [MyAttribute]
    [YourAttribute]
    public virtual void MyMethod()
    {
        //...
    }
}
```

마지막으로 클래스는 기본 클래스 `YourClass` `MyClass`에서 상속됩니다. 메서드 `MyMethod`은 `MyAttribute`을 표시하지만 `YourAttribute`은 표시하지 않습니다.

C#

```
public class YourClass : MyClass
{
    // MyMethod will have MyAttribute but not YourAttribute.
```

```
    public override void MyMethod()
    {
        //...
    }
}
```

AllowMultiple 속성

속성은 AttributeUsageAttribute.AllowMultiple 특성의 여러 인스턴스가 요소에 존재할 수 있는지 여부를 나타냅니다. 로 `true` 설정하면 여러 인스턴스가 허용됩니다. (기본값)으로 `false` 설정하면 하나의 인스턴스만 허용됩니다.

다음 예제에서, `MyAttribute` 은 기본 AllowMultiple 값이 `false` 인 반면, `YourAttribute` 의 값은 `true` 입니다.

C#

```
//This defaults to AllowMultiple = false.
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class YourAttribute : Attribute
{
}
```

이러한 특성의 여러 인스턴스가 적용 `MyAttribute` 되면 컴파일러 오류가 발생합니다. 다음 코드 예제에서는 `YourAttribute` 의 올바른 사용과 `MyAttribute` 의 잘못된 사용을 보여줍니다.

C#

```
public class MyClass
{
    // This produces an error.
    // Duplicates are not allowed.
    [MyAttribute]
    [MyAttribute]
    public void MyMethod()
    {
        //...
    }

    // This is valid.
    [YourAttribute]
    [YourAttribute]
    public void YourMethod()
    {
```

```
        //...
    }
}
```

AllowMultiple 속성과 Inherited 속성이 모두 `true`로 설정되는 경우, 다른 클래스에서 상속된 클래스가 해당 특성을 상속받고 동일한 자식 클래스에 동일한 특성의 또 다른 인스턴스를 적용할 수 있습니다. AllowMultiple이(가) `false`으로 설정되면, 부모 클래스의 모든 속성 값은 자식 클래스에서 동일한 속성의 새로운 인스턴스로 덮어쓰게 됩니다.

특성 클래스 선언

적용한 AttributeUsageAttribute후 특성의 세부 사항을 정의하기 시작합니다. 특성 클래스의 선언은 다음 코드에서 설명한 대로 기존 클래스의 선언과 유사합니다.

C#

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
    // . . .
}
```

이 특성 정의는 다음 사항을 보여 줍니다.

- 특성 클래스는 공용 클래스로 선언해야 합니다.
- 규칙에 따라 특성 클래스의 **이름은 특성이라는** 단어로 끝납니다. 필수는 아니지만 가독성을 위해 이 규칙을 사용하는 것이 좋습니다. 특성이 적용되면 특성이라는 단어를 포함하는 것은 선택 사항입니다.
- 모든 특성 클래스는 System.Attribute 클래스로부터 직접 또는 간접적으로 상속해야 합니다.
- Microsoft Visual Basic 모든 사용자 지정 특성 클래스에는 System.AttributeUsageAttribute 특성이 있어야 합니다.

생성자 선언

기존 클래스와 마찬가지로 특성은 생성자를 사용하여 초기화됩니다. 다음 코드 조각은 일반적인 특성 생성자를 보여 줍니다. 이 공용 생성자는 매개 변수를 사용하고 멤버 변수를 해당 값과 동일하게 설정합니다.

C#

```
public MyAttribute(bool myvalue)
{
    this.myvalue = myvalue;
}
```

다양한 값 조합을 수용하도록 생성자를 오버로드할 수 있습니다. 사용자 지정 특성 클래스에 대한 속성 도 정의하는 경우 특성을 초기화할 때 명명된 매개 변수와 위치 매개 변수의 조합을 사용할 수 있습니다. 일반적으로 모든 필수 매개 변수를 위치로 정의하고 모든 선택적 매개 변수를 명명된 것으로 정의합니다. 이 경우 필수 매개 변수 없이는 특성을 초기화할 수 없습니다. 다른 모든 매개 변수는 선택적 요소입니다.

> ⓘ **참고 항목**
>
> Visual Basic 특성 클래스의 생성자는 `ParamArray` 인수를 사용하면 안 됩니다.

런타임이 메타데이터에서 직접 특성 값을 읽을 수 있어야 하므로 특성의 생성자 매개 변수 및 공용 속성은 제한된 형식 집합으로 제한됩니다. 유효한 특성 매개 변수 형식은 다음과 같습니다.

- 단순 형식(C# 키워드/Visual Basic 키워드/.NET 런타임 형식):

테이블 확장

C#	Visual Basic	.NET 런타임 유형
`bool`	`Boolean`	Boolean
`byte`	`Byte`	Byte
`char`	`Char`	Char
`double`	`Double`	Double
`float`	`Single`	Single
`int`	`Integer`	Int32
`long`	`Long`	Int64
`short`	`Short`	Int16
`string`	`String`	String

- Type;
- 특성 사용 사이트에서 액세스할 수 있는 열거형 형식입니다.

- C# `object` 에서는 값이 유효한 특성 인수 형식 또는 1차원 배열 중 하나인 경우
- 이전 형식의 1차원 배열입니다.

이 목록 외부의 형식을 허용하는 생성자를 정의하면 특성이 성공적으로 컴파일되지만 적용하려고 할 때 컴파일러 오류가 발생합니다. 특성을 적용할 때 허용되는 식에 대한 자세한 내용은 특성 적용을 참조하세요.

> ⓘ **참고 항목**
>
> 리터럴 상수를 지원하더라도 형식 `sbyte`, `ushort`, `uint`, `ulong` `decimal`, 및 `nint` `nuint` 유효한 특성 매개 변수 형식이 아닙니다.

다음 코드 예제에서는 선택적 매개 변수 및 필수 매개 변수를 사용 하 여 이전 생성자를 사용 하는 특성을 적용할 수 있는 방법을 보여 있습니다. 특성에 하나의 필수 부울 값과 하나의 선택적 문자열 속성이 있다고 가정합니다.

C#

```
// One required (positional) and one optional (named) parameter are applied.
[MyAttribute(false, OptionalParameter = "optional data")]
public class SomeClass
{
    //...
}
// One required (positional) parameter is applied.
[MyAttribute(false)]
public class SomeOtherClass
{
    //...
}
```

속성 선언

명명된 매개 변수를 정의하거나 특성에 저장된 값을 쉽게 반환할 수 있는 방법을 제공하려면 속성을 선언합니다. 반환될 데이터 형식에 대한 설명과 함께 특성 속성을 공용 엔터티로 선언해야 합니다. 속성 값을 보유할 변수를 정의하고 해당 변수를 `get` 및 `set` 메서드와 연결합니다. 다음 코드 예제에서는 특성에서 속성을 구현하는 방법을 보여 줍니다.

C#

```
public bool MyProperty
{
    get {return this.myvalue;}
```

```
    set {this.myvalue = value;}
}
```

사용자 지정 특성 예제

이 섹션에서는 이전 정보를 통합하고 코드 섹션의 작성자 정보를 문서화하는 특성을 디자인하는 방법을 보여 줍니다. 이 예제의 특성은 프로그래머의 이름과 수준 및 코드를 검토했는지 여부를 저장합니다. 세 개의 프라이빗 변수를 사용하여 저장할 실제 값을 저장합니다. 각 변수는 값을 가져오고 설정하는 public 속성으로 표시됩니다. 마지막으로 생성자는 다음 두 개의 필수 매개 변수로 정의됩니다.

C#

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : Attribute
{
    // Private fields.
    private string name;
    private string level;
    private bool reviewed;

    // This constructor defines two required parameters: name and level.

    public DeveloperAttribute(string name, string level)
    {
        this.name = name;
        this.level = level;
        this.reviewed = false;
    }

    // Define Name property.
    // This is a read-only attribute.

    public virtual string Name
    {
        get {return name;}
    }

    // Define Level property.
    // This is a read-only attribute.

    public virtual string Level
    {
        get {return level;}
    }

    // Define Reviewed property.
    // This is a read/write attribute.

    public virtual bool Reviewed
```

```
    {
        get {return reviewed;}
        set {reviewed = value;}
    }
}
```

다음 방법 중 하나로 전체 이름을 `DeveloperAttribute` 사용하거나 약어 이름을 `Developer` 사용하여 이 특성을 적용할 수 있습니다.

C#

```
[Developer("Joan Smith", "1")]

-or-

[Developer("Joan Smith", "1", Reviewed = true)]
```

첫 번째 예제에서는 필요한 명명된 매개 변수만 사용하여 적용된 특성을 보여줍니다. 두 번째 예제에서는 필수 매개 변수와 선택적 매개 변수를 모두 사용하여 적용된 특성을 보여줍니다.

참고하십시오

- System.Attribute
- System.AttributeUsageAttribute
- 특성
- 특성 매개 변수 형식

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

특성에 저장된 정보 검색

아티클 • 2025. 02. 08.

사용자 지정 특성을 검색하는 것은 간단한 프로세스입니다. 먼저 검색할 특성의 인스턴스를 선언합니다. 그런 다음 Attribute.GetCustomAttribute 메서드를 사용하여 검색하려는 특성 값으로 새 특성을 초기화합니다. 새 특성이 초기화되면 해당 속성을 사용하여 값을 가져올 수 있습니다.

> **ⓘ 중요**
>
> 이 문서에서는 실행 컨텍스트에 로드된 코드의 특성을 검색하는 방법을 설명합니다. 리플렉션 전용 컨텍스트에 로드된 코드의 속성을 검색하려면, **방법: Reflection-Only 컨텍스트**어셈블리 로드에 설명된 대로 CustomAttributeData 클래스를 사용해야 합니다.

이 섹션에서는 특성을 검색하는 다음과 같은 방법을 설명합니다.

- 속성의 단일 인스턴스 검색
- 동일한 범위 적용된 특성의 여러 인스턴스 검색
- 다양한 범위에 적용된 특성의 여러 인스턴스 검색

특성의 단일 인스턴스 검색

다음 예제에서는 이전 섹션에서 설명한 `DeveloperAttribute`가 클래스 수준에서 `MainApp` 클래스에 적용됩니다. `GetAttribute` 메서드는 `GetCustomAttribute` 사용하여 클래스 수준에서 `DeveloperAttribute` 저장된 값을 검색한 후 콘솔에 표시합니다.

C#

```
using System;
using System.Reflection;
using CustomCodeAttributes;

[Developer("Joan Smith", "42", Reviewed = true)]
class MainApp
{
    public static void Main()
    {
        // Call function to get and display the attribute.
        GetAttribute(typeof(MainApp));
    }
```

```
    public static void GetAttribute(Type t)
    {
        // Get instance of the attribute.
        DeveloperAttribute MyAttribute =
            (DeveloperAttribute) Attribute.GetCustomAttribute(t, typeof
(DeveloperAttribute));

        if (MyAttribute == null)
        {
            Console.WriteLine("The attribute was not found.");
        }
        else
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." ,
MyAttribute.Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." ,
MyAttribute.Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}." ,
MyAttribute.Reviewed);
        }
    }
}
```

이전 프로그램을 실행하면 다음 텍스트가 표시됩니다.

콘솔

```
The Name Attribute is: Joan Smith.
The Level Attribute is: 42.
The Reviewed Attribute is: True.
```

특성을 찾을 수 없으면 `GetCustomAttribute` 메서드는 `MyAttribute` null 값으로 초기화합니다. 이 예제에서는 이러한 인스턴스에 대한 `MyAttribute` 확인하고 특성을 찾을 수 없는 경우 사용자에게 알릴 수 있습니다. 클래스 범위에서 `DeveloperAttribute` 찾을 수 없으면 콘솔에 다음 메시지가 표시됩니다.

콘솔

```
The attribute was not found.
```

앞의 예제에서는 특성 정의가 현재 네임스페이스에 있다고 가정합니다. 특성 정의가 현재 네임스페이스에 없는 경우 상주하는 네임스페이스를 가져와야 합니다.

동일한 범위에 적용된 특성의 여러 인스턴스 검색

앞의 예제에서는 검사할 클래스와 찾을 특정 특성이 GetCustomAttribute 메서드에 전달됩니다. 이 코드는 특성의 인스턴스가 클래스 수준에서만 적용되는 경우 잘 작동합니다. 그러나 특성의 여러 인스턴스가 동일한 클래스 수준에서 적용되는 경우 `GetCustomAttribute` 메서드는 모든 정보를 검색하지 않습니다. 동일한 특성의 여러 인스턴스가 동일한 범위에 적용되는 경우 Attribute.GetCustomAttributes 메서드를 사용하여 특성의 모든 인스턴스를 배열에 배치할 수 있습니다. 예를 들어 `DeveloperAttribute` 두 인스턴스가 동일한 클래스의 클래스 수준에 적용되는 경우 두 특성에 있는 정보를 표시하도록 `GetAttribute` 메서드를 수정할 수 있습니다. 동일한 수준에서 여러 특성을 적용해야 합니다. AttributeUsageAttribute 클래스에서 `AllowMultiple` 속성을 `true`으로 설정하여 특성을 정의해야 합니다.

다음 코드 예제에서는 `GetCustomAttributes` 메서드를 사용하여 지정된 클래스의 모든 `DeveloperAttribute` 인스턴스를 참조하는 배열을 만드는 방법을 보여 줍니다. 그런 다음 코드는 모든 특성의 값을 콘솔에 출력합니다.

C#

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute[] MyAttributes =
        (DeveloperAttribute[]) Attribute.GetCustomAttributes(t, typeof
(DeveloperAttribute));

    if (MyAttributes.Length == 0)
    {
        Console.WriteLine("The attribute was not found.");
    }
    else
    {
        for (int i = 0 ; i < MyAttributes.Length ; i++)
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." ,
MyAttributes[i].Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." ,
MyAttributes[i].Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}.",
MyAttributes[i].Reviewed);
        }
    }
}
```

특성을 찾을 수 없는 경우 이 코드는 사용자에게 경고합니다. 그렇지 않으면 두 `DeveloperAttribute` 인스턴스에 포함된 정보가 표시됩니다.

다른 범위에 적용된 특성의 여러 인스턴스 검색

GetCustomAttributes 및 GetCustomAttribute 메서드는 전체 클래스를 검색하지 않고 해당 클래스에 있는 특성의 모든 인스턴스를 반환합니다. 대신 지정된 메서드 또는 멤버를 한 번에 하나만 검색합니다. 모든 멤버에 동일한 특성이 적용된 클래스가 있고 그 멤버에 적용된 모든 특성의 값을 검색하려는 경우, 모든 메서드 또는 멤버를 개별적으로 `GetCustomAttributes` 및 `GetCustomAttribute`에 제공해야 합니다.

다음 코드 예제에서는 클래스를 매개 변수로 사용하고 클래스 수준 및 해당 클래스의 모든 개별 메서드에서 `DeveloperAttribute`(이전에 정의됨)를 검색합니다.

C#

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute att;

    // Get the class-level attributes.

    // Put the instance of the attribute on the class level in the att
object.
    att = (DeveloperAttribute) Attribute.GetCustomAttribute (t, typeof
(DeveloperAttribute));

    if (att == null)
    {
        Console.WriteLine("No attribute in class {0}.\n", t.ToString());
    }
    else
    {
        Console.WriteLine("The Name Attribute on the class level is: {0}.",
att.Name);
        Console.WriteLine("The Level Attribute on the class level is: {0}.",
att.Level);
        Console.WriteLine("The Reviewed Attribute on the class level is:
{0}.\n", att.Reviewed);
    }

    // Get the method-level attributes.

    // Get all methods in this class, and put them
    // in an array of System.Reflection.MemberInfo objects.
    MemberInfo[] MyMemberInfo = t.GetMethods();

    // Loop through all methods in this class that are in the
    // MyMemberInfo array.
    for (int i = 0; i < MyMemberInfo.Length; i++)
```

```
    {
        att = (DeveloperAttribute)
Attribute.GetCustomAttribute(MyMemberInfo[i], typeof (DeveloperAttribute));
        if (att == null)
        {
            Console.WriteLine("No attribute in member function {0}.\n" ,
MyMemberInfo[i].ToString());
        }
        else
        {
            Console.WriteLine("The Name Attribute for the {0} member is:
{1}.",
                MyMemberInfo[i].ToString(), att.Name);
            Console.WriteLine("The Level Attribute for the {0} member is:
{1}.",
                MyMemberInfo[i].ToString(), att.Level);
            Console.WriteLine("The Reviewed Attribute for the {0} member is:
{1}.\n",
                MyMemberInfo[i].ToString(), att.Reviewed);
        }
    }
}
```

메서드 수준 또는 클래스 수준에서 `DeveloperAttribute` 인스턴스가 없는 경우 `GetAttribute` 메서드는 사용자에게 특성을 찾을 수 없음을 알리고 특성이 포함되지 않은 메서드 또는 클래스의 이름을 표시합니다. 특성이 발견되면 콘솔에 `Name`, `Level` 및 `Reviewed` 필드가 표시됩니다.

Type 클래스의 멤버를 사용하여 전달된 클래스의 개별 메서드와 멤버를 가져올 수 있습니다. 이 예제에서는 먼저 `Type` 개체를 쿼리하여 클래스 수준에 대한 특성 정보를 가져옵니다. 다음으로, Type.GetMethods 사용하여 모든 메서드의 인스턴스를 System.Reflection.MemberInfo 개체 배열에 배치하여 메서드 수준에 대한 특성 정보를 검색합니다. Type.GetProperties 메서드를 사용하여 속성 수준에서 특성을 확인하거나 Type.GetConstructors 생성자 수준에서 특성을 확인할 수도 있습니다.

클래스 멤버에서 특성 검색

클래스 수준에서 특성을 검색하는 것 외에도 메서드, 속성 및 필드와 같은 개별 멤버에도 특성을 적용할 수 있습니다. `GetCustomAttribute` 및 `GetCustomAttributes` 메서드를 사용하여 이러한 특성을 검색할 수 있습니다.

본보기

다음 예제에서는 메서드에 적용된 특성을 검색하는 방법을 보여 줍니다.

```
C#
```

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
    public string Description { get; }
    public MyAttribute(string description) { Description = description; }
}

public class MyClass
{
    [MyAttribute("This is a sample method.")]
    public void MyMethod() { }
}

class AttributeRetrieval
{
    public static void Main()
    {
        // Create an instance of MyClass
        MyClass myClass = new MyClass();

        // Retrieve the method information for MyMethod
        MethodInfo methodInfo = typeof(MyClass).GetMethod("MyMethod");
        MyAttribute attribute =
(MyAttribute)Attribute.GetCustomAttribute(methodInfo, typeof(MyAttribute));

        if (attribute != null)
        {
            // Print the description of the method attribute
            Console.WriteLine("Method Attribute: {0}",
attribute.Description);
        }
        else
        {
            Console.WriteLine("Attribute not found.");
        }
    }
}
```

참고 항목

- System.Type
- Attribute.GetCustomAttribute
- Attribute.GetCustomAttributes
- 특성

메모리 관련 및 스팬 유형

2025. 06. 17.

.NET에는 임의 메모리의 연속적이고 강력한 형식의 영역을 나타내는 여러 가지 상호 연결된 형식이 포함되어 있습니다. 이러한 형식은 *메모리를 복사하거나 관리되는 힙에* 필요한 것보다 더 많은 할당을 방지하는 알고리즘을 만들 수 있도록 설계되었습니다. `Slice`, `AsSpan()`, 컬렉션 표현식 또는 생성자를 통해 이들을 생성하는 것은 기본 버퍼를 복제하는 것이 아니라, 래핑된 메모리의 "뷰"를 나타내는 관련 참조 및 오프셋만 업데이트합니다. 고성능 코드에서 범위는 문자열을 불필요하게 할당하지 않도록 하는 데 자주 사용됩니다.

형식은 다음과 같습니다.

- System.Span<T>- 인접한 메모리 영역에 액세스하는 데 사용되는 형식입니다. 인스턴스는 Span<T> 형식 `T`배열, stackalloc로 할당된 버퍼 또는 관리되지 않는 메모리에 대한 포인터로 백업할 수 있습니다. 스택에 할당해야 하므로 여러 가지 제한 사항이 있습니다. 예를 들어 클래스의 필드는 형식 Span<T>일 수 없으며 비동기 작업에서 사용할 수도 없습니다.

- System.ReadOnlySpan<T>- 변경할 수 없는 구조 버전입니다 Span<T> . 인스턴스는 String에 의해 지원될 수도 있습니다.

- System.Memory<T>는 인접한 메모리 영역에 대한 래퍼입니다. 인스턴스는 Memory<T> 형식의 배열이나 메모리 관리자에 의해 지원될 수 있습니다. 관리되는 힙 Memory<T> 에 저장할 수 있으므로 제한 사항이 Span<T>없습니다.

- System.ReadOnlyMemory<T>- 변경할 수 없는 구조 버전입니다 Memory<T> . 인스턴스는 또한 String에 의해 지원될 수 있습니다.

- System.Buffers.MemoryPool<T>는 메모리 풀에서 강력한 형식의 메모리 블록을 소유자에게 할당합니다. IMemoryOwner<T> 인스턴스는 MemoryPool<T>.Rent을(를) 호출하여 풀에서 임대할 수 있으며, MemoryPool<T>.Dispose()을(를) 호출하여 풀로 다시 반환할 수 있습니다.

- System.Buffers.IMemoryOwner<T>메모리 블록의 소유자를 나타내고 수명 관리를 제어하는 입니다.

- MemoryManager<T>는 Memory<T>가 안전 핸들과 같은 추가 형식으로 백업될 수 있도록 Memory<T>의 구현을 대체하는 데 사용할 수 있는 추상 기본 클래스입니다. MemoryManager<T> 는 고급 시나리오를 위한 것입니다.

- ArraySegment<T>- 특정 인덱스에서 시작하는 배열 요소의 특정 수에 대한 래퍼입니다.

- System.MemoryExtensions문자열, 배열 및 배열 세그먼트를 블록으로 변환하기 위한 확장 메서드의 컬렉션입니다 Memory<T> .

자세한 내용은 네임스페이스를 System.Buffers 참조하세요.

메모리 및 범위 작업

메모리 관련 및 범위 관련 형식은 일반적으로 처리 파이프라인에 데이터를 저장하는 데 사용되므로, 사용 시 모범 사례 집합 및 관련 형식을 따르는 것이 중요합니다Span<T>Memory<T>. 이러한 모범 사례는 Memory<T 및 Span>T<> 사용 지침에 설명되어 있습니다.

참고하십시오

- System.Memory<T>
- System.ReadOnlyMemory<T>
- System.Span<T>
- System.ReadOnlySpan<T>
- System.Buffers

메모리<T> 및 Span<T> 사용 지침

아티클 • 2025. 04. 21.

.NET에는 임의 연속 메모리 영역을 나타내는 여러 형식이 포함되어 있습니다. Span<T> 관리되거나 ReadOnlySpan<T> 관리되지 않는 메모리에 대한 참조를 래핑하는 경량 메모리 버퍼입니다. 이러한 형식은 스택에만 저장할 수 있으므로 비동기 메서드 호출과 같은 시나리오에는 적합하지 않습니다. 이 문제를 해결하기 위해 .NET 2.1에는 , , 및 MemoryPool<T>를 비롯한 Memory<T>몇 가지 추가 형식이 IMemoryOwner<T>추가ReadOnlyMemory<T>되었습니다. 마찬가지로 Span<T>관련 Memory<T> 형식은 관리되는 메모리와 관리되지 않는 메모리 모두에서 지원될 수 있습니다. 달리 Span<T>관리되는 Memory<T> 힙에 저장할 수 있습니다.

Memory<T> 둘 다 Span<T> 파이프라인에서 사용할 수 있는 구조화된 데이터의 버퍼에 대한 래퍼입니다. 즉, 일부 또는 모든 데이터를 파이프라인의 구성 요소에 효율적으로 전달할 수 있도록 설계되어 이를 처리하고 필요에 따라 버퍼를 수정할 수 있습니다. 여러 구성 요소 또는 여러 스레드에서 관련 형식에 액세스할 수 있으므로 Memory<T> 강력한 코드를 생성하려면 몇 가지 표준 사용 지침을 따르는 것이 중요합니다.

소유자, 소비자 및 수명 관리

버퍼는 API 간에 전달될 수 있으며 때로는 여러 스레드에서 액세스할 수 있으므로 버퍼의 수명이 어떻게 관리되는지 알고 있어야 합니다. 세 가지 핵심 개념이 있습니다.

- **소유권**. 버퍼 인스턴스의 소유자는 더 이상 사용되지 않을 때 버퍼를 삭제하는 것을 포함하여 수명 관리를 담당합니다. 모든 버퍼에는 단일 소유자가 있습니다. 일반적으로 소유자는 버퍼를 만들거나 팩터리에서 버퍼를 받은 구성 요소입니다. 소유권을 이전할 수도 있습니다. **Component-A** 는 버퍼의 제어를 **Component-B**로 포기할 수 있으며, 이때 **Component-A** 는 더 이상 버퍼를 사용하지 않을 수 있으며, **Component-B** 는 더 이상 사용되지 않을 때 버퍼를 삭제합니다.

- **소비**량. 버퍼 인스턴스의 소비자는 버퍼 인스턴스에서 읽고 쓸 수 있도록 버퍼 인스턴스를 사용할 수 있습니다. 일부 외부 동기화 메커니즘이 제공되지 않는 한 버퍼는 한 번에 하나의 소비자를 가질 수 있습니다. 버퍼의 활성 소비자가 반드시 버퍼의 소유자가 아닌 것은 아닙니다.

- **임대**. 임대는 특정 구성 요소가 버퍼의 소비자가 될 수 있는 시간입니다.

다음 의사코드 예제에서는 이러한 세 가지 개념을 설명합니다. `Buffer`는 의사 코드에서 Memory<T> 또는 Span<T> 버퍼 중 Char 형식의 버퍼를 나타냅니다. 메서드는 `Main` 버퍼를 인스턴스화하고, 메서드를 호출 `WriteInt32ToBuffer` 하여 정수의 문자열 표현을 버퍼에 쓴 다음, 메서드를 호출 `DisplayBufferToConsole` 하여 버퍼의 값을 표시합니다.

```
C#
```

```
using System;

class Program
{
    // Write 'value' as a human-readable string to the output buffer.
    void WriteInt32ToBuffer(int value, Buffer buffer);

    // Display the contents of the buffer to the console.
    void DisplayBufferToConsole(Buffer buffer);

    // Application code
    static void Main()
    {
        var buffer = CreateBuffer();
        try
        {
            int value = Int32.Parse(Console.ReadLine());
            WriteInt32ToBuffer(value, buffer);
            DisplayBufferToConsole(buffer);
        }
        finally
        {
            buffer.Destroy();
        }
    }
}
```

메서드는 `Main` 버퍼를 만들고 소유자도 마찬가지입니다. 따라서 `Main` 버퍼가 더 이상 사용되지 않을 때 버퍼를 삭제해야 합니다. 의사 코드는 버퍼에서 메서드 `Destroy`를 호출함으로써 이를 설명합니다. (Memory<T>와 Span<T> 모두 실제로 `Destroy` 메서드를 가지고 있지 않습니다. 실제 코드 예제는 이 문서의 후반부에서 보게 될 것입니다.)

버퍼에는 두 명의 소비자, `WriteInt32ToBuffer`와 `DisplayBufferToConsole`가 있습니다. 한 번에 하나의 소비자(첫 번째 `WriteInt32ToBuffer`, 다음 `DisplayBufferToConsole`)만 있으며 두 소비자 모두 버퍼를 소유하지 않습니다. 또한 이 컨텍스트의 "소비자"는 버퍼의 읽기 전용 보기를 의미하지는 않습니다. 소비자는 버퍼의 읽기/쓰기 뷰가 지정된 경우와 마찬가지로 `WriteInt32ToBuffer` 버퍼의 콘텐츠를 수정할 수 있습니다.

메서드는 메서드 호출이 시작되는 시점부터 메서드가 반환하기까지 버퍼를 사용할 수 있는 임대를 갖고 있습니다. 마찬가지로, `DisplayBufferToConsole`가 실행되는 동안 버퍼에 임대가 있으며, 메서드가 해제될 때 그 임대는 해제됩니다. (임대 관리를 위한 API는 없습니다. "임대"는 개념적 문제입니다.)

메모리<T> 및 소유자/소비자 모델

소유자, 소비자 및 수명 관리 섹션에서 설명한 것처럼 버퍼에는 항상 소유자가 있습니다. .NET은 다음 두 가지 소유권 모델을 지원합니다.

- 단일 소유권을 지원하는 모델입니다. 버퍼에는 전체 수명 동안 단일 소유자가 있습니다.
- 소유권 이전을 지원하는 모델입니다. 버퍼의 소유권은 원래 소유자(작성자)에서 다른 구성 요소로 이전할 수 있으며, 그러면 버퍼의 수명 관리를 담당하게 됩니다. 해당 소유자는 소유권을 다른 구성 요소로 이전할 수 있습니다.

인터페이스를 System.Buffers.IMemoryOwner<T> 사용하여 버퍼의 소유권을 명시적으로 관리합니다. IMemoryOwner<T> 는 두 소유권 모델을 모두 지원합니다. 참조가 있는 IMemoryOwner<T> 구성 요소는 버퍼를 소유합니다. 다음 예제에서는 인스턴스를 IMemoryOwner<T> 사용하여 버퍼의 소유권을 반영합니다 Memory<T> .

C#

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

        Console.Write("Enter a number: ");
        try
        {
            string? s = Console.ReadLine();

            if (s is null)
                return;

            var value = Int32.Parse(s);

            var memory = owner.Memory;

            WriteInt32ToBuffer(value, memory);

            DisplayBufferToConsole(owner.Memory.Slice(0,
value.ToString().Length));
        }
        catch (FormatException)
        {
            Console.WriteLine("You did not enter a valid number.");
        }
        catch (OverflowException)
        {
            Console.WriteLine($"You entered a number less than {Int32.MinValue:N0}
or greater than {Int32.MaxValue:N0}.");
        }
        finally
```

```
        {
            owner?.Dispose();
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Span;
        for (int ctr = 0; ctr < strValue.Length; ctr++)
            span[ctr] = strValue[ctr];
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

다음 명령문을 사용하여 이 예제를 작성할 수도 있습니다.using

C#

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        using (IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent())
        {
            Console.Write("Enter a number: ");
            try
            {
                string? s = Console.ReadLine();

                if (s is null)
                    return;

                var value = Int32.Parse(s);

                var memory = owner.Memory;
                WriteInt32ToBuffer(value, memory);
                DisplayBufferToConsole(memory.Slice(0, value.ToString().Length));
            }
            catch (FormatException)
            {
                Console.WriteLine("You did not enter a valid number.");
            }
            catch (OverflowException)
            {
                Console.WriteLine($"You entered a number less than
{Int32.MinValue:N0} or greater than {Int32.MaxValue:N0}.");
```

```
            }
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Slice(0, strValue.Length).Span;
        strValue.AsSpan().CopyTo(span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

이 코드에서는 다음을 수행합니다.

- 메서드는 `Main` 인스턴스에 대한 참조를 IMemoryOwner<T> 보유하므로 `Main` 메서드는 버퍼의 소유자입니다.
- `WriteInt32ToBuffer` 및 `DisplayBufferToConsole` 메서드는 Memory<T>를 공용 API로 허용합니다. 따라서 버퍼의 소비자입니다. 이러한 메서드는 버퍼를 한 번에 하나씩 소모합니다.

`WriteInt32ToBuffer` 메서드는 버퍼에 값을 쓰기 위한 것이지만, `DisplayBufferToConsole` 메서드는 그 목적이 아닙니다. 이를 반영하기 위해 형식 ReadOnlyMemory<T>의 인수를 수락했을 수 있습니다. 자세한 ReadOnlyMemory<T>내용은 규칙 #2: 버퍼가 읽기 전용이어야 하는 경우 ReadOnlySpan<T> 또는 ReadOnlyMemory<T> 를 사용합니다.

"소유자 없는" 메모리<T> 인스턴스

Memory<T> 인스턴스를 IMemoryOwner<T> 사용하지 않고 만들 수 있습니다. 이 경우 버퍼의 소유권은 명시적이 아닌 암시적이며 단일 소유자 모델만 지원됩니다. 다음을 통해 이 작업을 수행할 수 있습니다.

- Memory<T> 생성자 중 하나를 직접 호출하여 `T[]`을 전달하는, 다음 예제와 같은 방식으로 수행합니다.
- String.AsMemory 확장 메서드를 호출하여 인스턴스를 생성합니다`ReadOnlyMemory<char>`.

C#

```
using System;

class Example
{
    static void Main()
    {
        Memory<char> memory = new char[64];
```

```
        Console.Write("Enter a number: ");
        string? s = Console.ReadLine();

        if (s is null)
            return;

        var value = Int32.Parse(s);

        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory);
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();
        strValue.AsSpan().CopyTo(buffer.Slice(0, strValue.Length).Span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

처음에 인스턴스를 Memory<T> 만드는 메서드는 버퍼의 암시적 소유자입니다. 이전을 용이하게 하는 인스턴스가 없으므로 소유권을 다른 구성 요소로 이전할 수 없습니다 IMemoryOwner<T> . (또는 런타임의 가비지 수집기가 버퍼를 소유하고 모든 메서드가 버퍼만 사용한다고 상상할 수도 있습니다.)

사용 지침

메모리 블록은 소유되지만 여러 구성 요소에 전달되기 위한 것이며, 그 중 일부는 특정 메모리 블록에서 동시에 작동할 수 있으므로 두 구성 요소 모두를 Memory<T>Span<T>사용하기 위한 지침을 설정하는 것이 중요합니다. 구성 요소가 다음을 수행할 수 있으므로 지침이 필요합니다.

- 소유자가 메모리 블록을 해제한 후 메모리 블록에 대한 참조를 유지합니다.
- 다른 구성 요소가 버퍼에서 작동하는 동시에 버퍼 자체에서 작업하여 데이터를 손상시킵니다.

스택 할당 특성은 Span<T> 성능을 최적화하고 메모리 블록에서 작동하는 데 Span<T>가 선호되는 형식이 되게 하지만, 또한 그로 인해 Span<T>에 몇 가지 주요 제한 사항이 적용됩니다. 언제 Span<T>를 사용하고, 언제 Memory<T>를 사용해야 하는지 아는 것이 중요합니다.

다음은 Memory<T> 및 관련 유형을 성공적으로 사용하기 위한 권장 사항입니다. Memory<T> 및 Span<T>에 적용되는 지침은 달리 명시되지 않는 한 ReadOnlyMemory<T> 및 ReadOnlySpan<T>에도 적용됩니다.

- 규칙 #1: 동기 API에서는 가능하면 매개 변수로 Memory<T> 대신 Span<T>를 사용합니다.

- 규칙 #2: 버퍼를 읽기 전용으로 사용 ReadOnlySpan<T> 하거나 ReadOnlyMemory<T> 사용해야 하는 경우
- 규칙 #3: 메서드가 Memory<T> 수락하고 반환하는 경우 메서드가 반환 void된 후 인스턴스를 Memory<T> 사용하면 안 됩니다.
- 규칙 #4: 메서드가 작업을 수락 Memory<T> 하고 반환하는 경우 태스크가 Memory<T> 터미널 상태로 전환된 후 인스턴스를 사용하면 안 됩니다.
- 규칙 #5: 생성자가 매개 변수로 허용하는 Memory<T> 경우 생성된 개체의 인스턴스 메서드는 인스턴스의 Memory<T> 소비자로 간주됩니다.
- 규칙 #6: 형식에 settable Memory<T>-typed 속성(또는 해당하는 인스턴스 메서드)이 있는 경우 해당 개체의 인스턴스 메서드는 인스턴스의 Memory<T> 소비자로 간주됩니다.
- 규칙 #7: 참조가 IMemoryOwner<T> 있는 경우 특정 시점에 참조를 삭제하거나 소유권을 이전해야 합니다(둘 다 아님).
- 규칙 #8: API 화면에 매개 변수가 IMemoryOwner<T> 있는 경우 해당 인스턴스의 소유권을 수락합니다.
- 규칙 #9: 동기 P/Invoke 메서드를 래핑하는 경우 API가 매개 변수로 수락 Span<T> 해야 합니다.
- 규칙 #10: 비동기 p/invoke 메서드를 래핑하는 경우 API가 매개 변수로 수락 Memory<T> 해야 합니다.

규칙 #1: 동기 API의 경우 가능하면 메모리<T 대신 Span<T>>를 매개 변수로 사용합니다.

Span<T>는 Memory<T>보다 더 다양하며 더 많은 연속 메모리 버퍼를 나타낼 수 있습니다. Span<T> 은 .보다 Memory<T>더 나은 성능을 제공합니다. 마지막으로 Memory<T>.Span 속성을 사용하여 Memory<T> 인스턴스를 Span<T>으로 변환할 수 있습니다. 하지만 Span<T>를 Memory<T>로 변환하는 것은 불가능합니다. 따라서 만약 호출자에게 Memory<T> 인스턴스가 있다면, 어쨌든 Span<T> 매개 변수를 사용하여 메서드를 호출할 수 있습니다.

형식 Memory<T> 대신 형식 Span<T> 의 매개 변수를 사용하면 올바른 사용 방법 구현을 작성할 수도 있습니다. 메서드의 임대를 넘어 버퍼에 액세스하려고 시도하지 않는지 확인하기 위해 컴파일 시간 검사를 자동으로 받습니다(나중에 자세히 참조).

경우에 따라 완전히 동기적인 경우에도 Span<T> 매개 변수 대신 Memory<T> 매개 변수를 사용해야 합니다. 아마도 사용자가 사용하는 API는 인수만 Memory<T> 허용합니다. 이것은 괜찮지만 동기적으로 사용할 Memory<T> 때 관련된 장단점에 유의하십시오.

규칙 #2: 버퍼가 읽기 전용이어야 하는 경우 ReadOnlySpan<T> 또는 ReadOnlyMemory<T> 사용

이전 예제에서 메서드는 `DisplayBufferToConsole` 버퍼에서만 읽습니다. 버퍼의 내용은 수정하지 않습니다. 메서드 시그니처는 다음으로 변경해야 합니다.

C#

```
void DisplayBufferToConsole(ReadOnlyMemory<char> buffer);
```

실제로 이 규칙과 규칙 #1을 결합하면 다음과 같이 메서드 서명을 더 잘 수행하고 다시 작성할 수 있습니다.

C#

```
void DisplayBufferToConsole(ReadOnlySpan<char> buffer);
```

이제 `DisplayBufferToConsole` 메서드는 거의 모든 버퍼 유형에서 작동합니다. 예를 들어, `T[]`, stackalloc로 할당된 스토리지 등입니다. 직접 String를 보낼 수 있습니다! 자세한 내용은 GitHub 문제 dotnet/docs #25551 을 참조하세요.

규칙 #3: 메서드가 Memory<T> 를 수락하고 반환하는 경우 메서드가 반환 `void`된 후 메모리<T> 인스턴스를 사용하면 안 됩니다.

이는 앞에서 언급한 "임대" 개념과 관련이 있습니다. 인스턴스에 대한 Memory<T> void 반환 메서드의 임대는 메서드가 입력될 때 시작되고 메서드가 종료될 때 종료됩니다. 콘솔의 입력을 기반으로 루프에서 호출 `Log` 하는 다음 예제를 고려해 보세요.

C#

```
// <Snippet1>
using System;
using System.Buffers;

public class Example
{
    // implementation provided by third party
    static extern void Log(ReadOnlyMemory<char> message);

    // user code
    public static void Main()
    {
        using (var owner = MemoryPool<char>.Shared.Rent())
        {
            var memory = owner.Memory;
            var span = memory.Span;
            while (true)
            {
                string? s = Console.ReadLine();

                if (s is null)
```

```
                    return;

                int value = Int32.Parse(s);
                if (value < 0)
                    return;

                int numCharsWritten = ToBuffer(value, span);
                Log(memory.Slice(0, numCharsWritten));
            }
        }
    }

    private static int ToBuffer(int value, Span<char> span)
    {
        string strValue = value.ToString();
        int length = strValue.Length;
        strValue.AsSpan().CopyTo(span.Slice(0, length));
        return length;
    }
}
// </Snippet1>

// Possible implementation of Log:
    // private static void Log(ReadOnlyMemory<char> message)
    // {
    //     Console.WriteLine(message);
    // }
```

완전히 동기 메서드인 경우 `Log` 지정된 시간에 메모리 인스턴스의 활성 소비자는 하나뿐이므로 이 코드는 예상대로 작동합니다. 그렇지만 `Log`에 이 구현이 있다고 상상해 보세요.

C#

```
// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread while performing
IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
    });
}
```

이 구현 `Log` 에서는 원래 메서드가 반환된 후에도 백그라운드에서 인스턴스를 Memory<T> 계속 사용하려고 하기 때문에 임대를 위반합니다. 이 메서드는 `Main` 버퍼에서 읽으려고 시도하는 동안 `Log` 버퍼를 변경하여 데이터가 손상될 수 있습니다.

이 문제를 해결하는 몇 가지 방법은 다음과 같습니다.

- `Log` 메서드는 `void` 대신 Task을 반환할 수 있으며, 다음 구현에서는 `Log` 메서드가 그렇게 합니다.

```
C#

// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while
performing IO.
    return Task.Run(() => {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}
```

- `Log` 대신 다음과 같이 구현할 수 있습니다.

```
C#

// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    string defensiveCopy = message.ToString();
    // Run in the background so that we don't block the main thread while
performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

규칙 #4: 메서드가 메모리<T> 를 수락하고 작업을 반환하는 경우 태스크가 터미널 상태로 전환된 후 메모리<T> 인스턴스를 사용하면 안 됩니다.

규칙 #3의 비동기 변형일 뿐입니다. 이전 예제의 메서드를 `Log` 다음과 같이 작성하여 이 규칙을 준수할 수 있습니다.

```
C#

// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
```

```
    // Run in the background so that we don't block the main thread while
performing IO.
    return Task.Run(() => {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}
```

여기서 "터미널 상태"는 태스크가 완료됨, 오류 또는 취소된 상태로 전환됨을 의미합니다. 즉, "터미널 상태"는 "대기를 throw하거나 실행을 계속할 수 있는 모든 것"을 의미합니다.

이 지침은 반환하는 Task메서드, Task<TResult>ValueTask<TResult>또는 유사한 형식에 적용됩니다.

규칙 #5: 생성자가 Memory<T> 를 매개 변수로 허용하는 경우 생성된 개체의 인스턴스 메서드는 Memory<T> 인스턴스의 소비자로 간주됩니다.

다음 예제를 고려하세요.

C#

```
class OddValueExtractor
{
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
      Console.WriteLine(value);
    }
}
```

`OddValueExtractor` 여기서 생성자는 생성자 매개 변수로 허용 `ReadOnlyMemory<int>` 하므로 생성자 자체는 인스턴스의 소비자이며 반환된 값의 `ReadOnlyMemory<int>` 모든 인스턴스 메서드도 원래 `ReadOnlyMemory<int>` 인스턴스의 소비자입니다. 즉 `TryReadNextOddValue` , 인스턴스가 메서드에 `ReadOnlyMemory<int>` 직접 전달되지 않더라도 인스턴스를 `TryReadNextOddValue` 사용합니다.

규칙 #6: 형식에 Settable Memory<T> 형식 속성(또는 해당하는 인스턴스 메서드)이 있는 경우 해당 개체의 인스턴스 메서드는 Memory<T> 인스턴스의 소비자로 간주됩니다.

이것은 실제로 규칙 #5의 변형일 뿐입니다. 이 규칙은 속성 setter 또는 해당 메서드가 입력을 캡처하고 유지하는 것으로 간주되므로 동일한 개체의 인스턴스 메서드가 캡처된 상태를 활용할 수 있기 때문입니다.

다음 예제에서는 이 규칙을 트리거합니다.

C#

```
class Person
{
    // Settable property.
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

규칙 #7: IMemoryOwner<T> 참조가 있는 경우 어느 시점에서 삭제하거나 소유권을 이전해야 합니다(둘 다 아님).

Memory<T> 인스턴스가 관리되는 메모리 또는 관리되지 않는 메모리에서 지원될 수 있으므로, Memory<T> 인스턴스에서 수행된 작업이 완료되면 소유자는 IMemoryOwner<T>에 대해 `Dispose`을 호출해야 합니다. 또는 소유자가 인스턴스의 IMemoryOwner<T> 소유권을 다른 구성 요소로 이전할 수 있습니다. 이때 획득 구성 요소는 적절한 시간에 호출 `Dispose` 을 담당하게 됩니다(나중에 자세히 설명).

인스턴스에서 `Dispose` 메서드를 IMemoryOwner<T> 호출하지 않으면 관리되지 않는 메모리 누수 또는 기타 성능 저하가 발생할 수 있습니다.

이 규칙은 다음과 같은 MemoryPool<T>.Rent팩터리 메서드를 호출하는 코드에도 적용됩니다. 호출자는 반환 IMemoryOwner<T> 된 소유자가 되며 완료되면 인스턴스를 삭제해야 합니다.

규칙 #8: API 화면에 IMemoryOwner<T> 매개 변수가 있는 경우 해당 인스턴스의 소유권을 수락합니다.

이 형식의 인스턴스를 수락하면 구성 요소가 이 인스턴스의 소유권을 취하려 한다는 신호가 표시됩니다. 규칙 #7에 따라 구성 요소가 적절한 폐기를 담당하게 됩니다.

인스턴스의 IMemoryOwner<T> 소유권을 다른 구성 요소로 전송하는 모든 구성 요소는 메서드 호출이 완료된 후 해당 인스턴스를 더 이상 사용하지 않아야 합니다.

> ⓘ **중요**
>
> 생성자가 매개 변수로 수락 IMemoryOwner<T> 하는 경우 해당 형식이 구현IDisposable되어야 하며 Dispose 메서드가 개체를 IMemoryOwner<T> 호출 `Dispose` 해야 합니다.

규칙 #9: 동기 p/invoke 메서드를 래핑하는 경우 API는 Span<T> 를 매개 변수로 수락해야 합니다.

규칙 #1 Span<T> 에 따르면 일반적으로 동기 API에 사용할 올바른 형식입니다. 다음 예제와 같이 키워드를 fixed 통해 인스턴스를 고정 Span<T> 할 수 있습니다.

C#

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

이전 예제 `pbData` 에서는 입력 범위가 비어 있는 경우 null일 수 있습니다. 내보낸 메서드에서 `pbData`이(가) null이 아닌 것이 절대적으로 필요한 경우, `cbData`이(가) 0이라도 메서드를 다음과 같이 구현할 수 있습니다.

C#

```
public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
```

```
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy,
data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

규칙 #10: 비동기 p/invoke 메서드를 래핑하는 경우 API는 Memory<T> 를 매개 변수로 수락해야 합니다.

비동기 작업에서는 fixed 키워드를 사용할 수 없으므로, 인스턴스가 나타내는 연속 메모리의 종류에 관계없이 Memory<T>.Pin 메서드를 사용하여 Memory<T> 인스턴스를 고정합니다. 다음 예제에서는 이 API를 사용하여 비동기 p/invoke 호출을 수행하는 방법을 보여줍니다.

C#

```
using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);

[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int cbData,
IntPtr pState, IntPtr lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr = GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState
    {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc(state);

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;

    // make the call
    int result;
    try
    {
        result = ExportedAsyncMethod((byte*)memoryHandle.Pointer, data.Length,
pState, _callbackPtr);
    }
    catch
```

```
    {
        ((GCHandle)pState).Free(); // cleanup since callback won't be invoked
        memoryHandle.Dispose();
        throw;
    }

    if (result != PENDING)
    {
        // Operation completed synchronously; invoke callback manually
        // for result processing and cleanup.
        MyCompletedCallbackImplementation(pState, result);
    }

    return tcs.Task;
}

private static void MyCompletedCallbackImplementation(IntPtr state, int result)
{
    GCHandle handle = (GCHandle)state;
    var actualState = (MyCompletedCallbackState)(handle.Target);
    handle.Free();
    actualState.MemoryHandle.Dispose();

    /* error checking result goes here */

    if (error)
    {
        actualState.Tcs.SetException(...);
    }
    else
    {
        actualState.Tcs.SetResult(result);
    }
}

private static IntPtr GetCompletionCallbackPointer()
{
    OnCompletedCallback callback = MyCompletedCallbackImplementation;
    GCHandle.Alloc(callback); // keep alive for lifetime of application
    return Marshal.GetFunctionPointerForDelegate(callback);
}

private class MyCompletedCallbackState
{
    public TaskCompletionSource<int> Tcs;
    public MemoryHandle MemoryHandle;
}
```

참고하십시오

- System.Memory<T>
- System.Buffers.IMemoryOwner<T>

- System.Span<T>

SIMD 가속 숫자 형식 사용

SIMD(단일 명령, 여러 데이터)는 단일 명령을 사용하여 여러 데이터 조각에 대한 작업을 병렬로 수행하기 위한 하드웨어 지원을 제공합니다. .NET에는 네임스페이스 아래에 SIMD 가속 형식 집합이 System.Numerics 있습니다. SIMD 작업은 하드웨어 수준에서 병렬 처리할 수 있습니다. 이는 수학, 과학 및 그래픽 앱에서 흔히 볼 수 있는 벡터화된 계산의 처리량을 증가합니다.

.NET SIMD 가속 형식

.NET SIMD 가속 형식에는 다음 형식이 포함됩니다.

- Vector2, Vector3 및 Vector4 형식은 각각 2, 3, 4개의 Single 값을 가진 벡터를 나타냅니다.
- 3x2 행렬을 나타내는 Matrix3x2와 값 Matrix4x4의 4x4 행렬을 나타내는 Single라는 두 가지 행렬 유형.
- Plane 값을 사용하여 Single 3차원 공간의 평면을 나타내는 형식입니다.
- Quaternion 값을 사용하여 Single 3차원 물리적 회전을 인코딩하는 데 사용되는 벡터를 나타내는 형식입니다.
- Vector<T> 지정된 숫자 형식의 벡터를 나타내고 SIMD 지원을 활용하는 광범위한 연산자 집합을 제공하는 형식입니다. 인스턴스 수는 Vector<T> 애플리케이션의 수명 동안 고정되지만 해당 값 Vector<T>.Count 은 코드를 실행하는 컴퓨터의 CPU에 따라 달라집니다.

  > **참고 항목**
  >
  > 형식은 Vector<T> .NET Framework에 포함되지 않습니다. 이 형식에 액세스하려면 System.Numerics.Vectors NuGet 패키지를 설치해야 합니다.

SIMD 가속 형식은 SIMD 가속이 아닌 하드웨어 또는 JIT 컴파일러와 함께 사용할 수 있는 방식으로 구현됩니다. 런타임에 SIMD 가속을 사용할 수 있는지 여부를 확인하려면 Vector.IsHardwareAccelerated을(를) 사용하십시오. 해당 속성이 반환 `true`되는 경우 적어도 일부 API는 하드웨어 가속 SIMD 작업을 사용합니다. `false`가 반환되는 경우, API가 하드웨어 가속화를 사용하지 않습니다.

SIMD를 사용하는 방법

사용자 지정 SIMD 알고리즘을 실행하기 전에 호스트 컴퓨터가 SIMD를 지원하는지 확인하려면 Vector.IsHardwareAccelerated를 사용하세요. 이 방법은 Boolean을 반환합니다. 이는 특정 형식

에 대해 SIMD 가속이 사용하도록 설정되어 있음을 보장하지는 않지만 일부 형식에서 지원한다는 표시기입니다.

간단한 벡터

.NET에서 가장 기본 SIMD 가속 형식은 Vector2 2, Vector3 3 및 4 Vector4 값이 있는 벡터를 나타내는 형식 및 Single 형식입니다. 아래 예제에서는 두 개의 벡터를 추가하는 데 사용합니다 Vector2 .

C#

```
var v1 = new Vector2(0.1f, 0.2f);
var v2 = new Vector2(1.1f, 2.2f);
var vResult = v1 + v2;
```

.NET 벡터를 사용하여 벡터 `Dot product` `Transform` `Clamp` 등의 다른 수학 속성을 계산할 수도 있습니다.

C#

```
var v1 = new Vector2(0.1f, 0.2f);
var v2 = new Vector2(1.1f, 2.2f);
var vResult1 = Vector2.Dot(v1, v2);
var vResult2 = Vector2.Distance(v1, v2);
var vResult3 = Vector2.Clamp(v1, Vector2.Zero, Vector2.One);
```

매트릭스

Matrix3x2 3x2 행렬을 나타내고 Matrix4x4 4x4 행렬을 나타내는 입니다. 행렬 관련 계산에 사용할 수 있습니다. 아래 예제에서는 SIMD를 사용하여 행렬을 해당 특파원 트랜스포즈 행렬에 곱하는 방법을 보여 줍니다.

C#

```
var m1 = new Matrix4x4(
            1.1f, 1.2f, 1.3f, 1.4f,
            2.1f, 2.2f, 3.3f, 4.4f,
            3.1f, 3.2f, 3.3f, 3.4f,
            4.1f, 4.2f, 4.3f, 4.4f);

var m2 = Matrix4x4.Transpose(m1);
var mResult = Matrix4x4.Multiply(m1, m2);
```

벡터<T>

이 Vector<T> 기능은 더 긴 벡터를 사용하는 기능을 제공합니다. 인스턴스 수는 Vector<T> 고정되어 있지만 해당 값 Vector<T>.Count 은 코드를 실행하는 컴퓨터의 CPU에 따라 달라집니다.

다음 예제에서는 을 사용하여 Vector<T>두 배열의 요소 단위 합계를 계산하는 방법을 보여 줍니다.

C#

```
double[] Sum(double[] left, double[] right)
{
    if (left is null)
    {
        throw new ArgumentNullException(nameof(left));
    }

    if (right is null)
    {
        throw new ArgumentNullException(nameof(right));
    }

    if (left.Length != right.Length)
    {
        throw new ArgumentException($"{nameof(left)} and {nameof(right)} are not
the same length");
    }

    int length = left.Length;
    double[] result = new double[length];

    // Get the number of elements that can't be processed in the vector
    // NOTE: Vector<T>.Count is a JIT time constant and will get optimized
accordingly
    int remaining = length % Vector<double>.Count;

    for (int i = 0; i < length - remaining; i += Vector<double>.Count)
    {
        var v1 = new Vector<double>(left, i);
        var v2 = new Vector<double>(right, i);
        (v1 + v2).CopyTo(result, i);
    }

    for (int i = length - remaining; i < length; i++)
    {
        result[i] = left[i] + right[i];
    }

    return result;
}
```

비고

SIMD는 하나의 병목 상태를 제거하고 다음(예: 메모리 처리량)을 노출할 가능성이 높습니다. 일반적으로 SIMD 사용의 성능 이점은 특정 시나리오에 따라 다르며, 경우에 따라 SIMD가 아닌 단순 코드보다 성능이 더 나빠질 수도 있습니다.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 31.

값 튜플

2025. 06. 17.

값 튜플은 값의 특정 수와 시퀀스를 포함하는 데이터 구조체입니다. .NET은 다음과 같은 기본 제공 값 튜플 형식을 제공합니다.

- 구조체는 ValueTuple<T1> 하나의 요소가 있는 값 튜플을 나타냅니다.
- 구조체는 ValueTuple<T1,T2> 두 요소가 있는 값 튜플을 나타냅니다.-
- 구조체는 ValueTuple<T1,T2,T3> 세 개의 요소가 있는 값 튜플을 나타냅니다.
- 구조체는 ValueTuple<T1,T2,T3,T4> 4개의 요소가 있는 값 튜플을 나타냅니다.
- 구조체는 ValueTuple<T1,T2,T3,T4,T5> 5개의 요소가 있는 값 튜플을 나타냅니다.
- 구조체는 ValueTuple<T1,T2,T3,T4,T5,T6> 6개의 요소가 있는 값 튜플을 나타냅니다.
- 구조체는 ValueTuple<T1,T2,T3,T4,T5,T6,T7> 7개의 요소가 있는 값 튜플을 나타냅니다.
- 구조체는 ValueTuple<T1,T2,T3,T4,T5,T6,T7,TRest> 8개 이상의 요소가 있는 값 튜플을 나타냅니다.

값 튜플 형식은 다음과 같이 Tuple<T1,T2>튜플 형식과 다릅니다.

- 클래스(참조 형식)가 아닌 구조체(값 형식)입니다.
- Item1와 Item2 같은 멤버는 속성이 아닌 필드입니다.
- 해당 필드는 읽기 전용이 아니라 변경할 수 있습니다.

값 튜플 형식은 C#의 튜플과 F#의 구조체 튜플을 지원하는 런타임 구현을 제공합니다. 언어 구문을 사용하여 ValueTuple<T1,T2> 인스턴스를 생성하는 것 외에도 Create 팩터리 메서드를 호출할 수 있습니다.

참고하십시오

- 튜플 형식(C# 참조)

런타임 라이브러리 개요

.NET 런타임에는 런타임 라이브러리, 프레임워크 라이브러리 또는 BCL(기본 클래스 라이브러리)으로 알려진 광범위한 표준 클래스 라이브러리 집합이 있습니다. 또한 NuGet 패키지에 제공되는 런타임 라이브러리에 대한 확장도 있습니다.

이러한 라이브러리를 통해 많은 일반 및 앱별 형식, 알고리즘 및 유틸리티 기능을 구현할 수 있습니다.

런타임 라이브러리.

런타임 라이브러리는 기본 형식 및 유틸리티 기능을 제공하며 다른 모든 .NET 클래스 라이브러리의 기반입니다. 예를 들어 System.String 문자열 작업을 위한 API를 제공하는 클래스가 있습니다. 또 다른 예는 serialization 라이브러리입니다.

런타임 라이브러리에 대한 확장

일부 라이브러리는 런타임 공유 프레임워크의 일부가 아닌 NuGet 패키지에 제공됩니다. 이러한 라이브러리는 종종 .NET Framework와 같은 다운레벨 .NET 버전을 대상으로 하는 앱에서도 사용할 수 있습니다.

다음 표에서는 패키지 제공 라이브러리의 몇 가지 예를 보여 줍니다.

테이블 확장

NuGet 패키지	개념 콘텐츠
Microsoft.Extensions.AI	AI
Microsoft.Extensions.Configuration	설정
Microsoft.Extensions.DependencyInjection	종속성 주입
Microsoft.Extensions.FileSystemGlobbing	파일 글로빙
Microsoft.Extensions.Hosting	제네릭 호스트
Microsoft.Extensions.Http	HTTP(HTTP)
Microsoft.Extensions.Localization	지역화
Microsoft.Extensions.Logging	로깅

참고하십시오

- .NET 소개
- .NET SDK 또는 런타임 설치
- 사용할 설치된 .NET SDK 또는 런타임 버전을 선택합니다.
- 프레임워크 종속 앱 게시

Last updated on 2025. 11. 05.

미리 보기 API

아티클 • 2025. 01. 29.

.NET 플랫폼은 호환성을 중요하게 생각합니다. 따라서 라이브러리 에코시스템은 특히 API와 관련하여 호환성이 손상되는 변경을 방지하는 경향이 있습니다.

그럼에도 불구하고 API를 디자인할 때는 사용자로부터 피드백을 수집하고 필요한 경우 해당 피드백에 따라 API를 변경할 수 있어야 합니다. 놀라움을 방지하려면 사용하는 API가 안정적으로 간주되고 어떤 API가 아직 활성 개발 중이며 변경될 수 있는지 이해해야 합니다.

API가 미리 보기 형식임을 표현할 수 있는 여러 가지 방법이 있습니다.

- 전체 구성 요소는 노출되는 경우 미리 보기로 간주됩니다.
  - .NET 런타임의 미리 보기 릴리스에서
  - 시험판 NuGet 패키지에서
- 그렇지 않으면 안정적인 구성 요소는 다음 특성을 사용하여 특정 API를 미리 보기로 표시합니다.
  - RequiresPreviewFeaturesAttribute
  - ExperimentalAttribute

이 문서에서는 각 옵션의 작동 방식과 라이브러리 개발자의 경우 이러한 옵션 중에서 선택하는 방법을 설명합니다.

.NET 런타임 미리 보기

라이브 라이선스가 있는 릴리스 후보(RC)를 제외하고 .NET 런타임 및 SDK의 미리 보기 버전은 지원되지 않습니다.

따라서 .NET 미리 보기의 일부로 추가된 모든 API는 미리 보기가 받는 피드백에 따라 변경될 수 있는 것으로 간주됩니다. .NET 런타임 미리 보기를 사용하려면 프로젝트에서 최신 프레임워크 버전을 명시적으로 대상으로 지정해야 합니다. 이렇게 하면 변경될 수 있는 API를 사용하는 데 동의를 암시적으로 표현합니다.

시험판 NuGet 패키지

NuGet 패키지는 안정 버전이거나 시험판 수 있습니다. 시험판 패키지는 해당 버전에 시험판 접미사가 있는 것으로 표시됩니다. 예를 들어, `System.Text.Json 9.0.0-preview.2.24128.5`은/는 `preview.2.24128.5`의 시험판 접미사가 있습니다.

시험판 패키지는 일반적으로 얼리어답터로부터 피드백을 수집하는 수단으로 사용됩니다. 일반적으로 작성자가 지원하지 않습니다.

CLI 또는 UI를 통해 패키지를 설치할 때 시험판 버전을 설치할지 여부를 명시적으로 표시해야 합니다. 이렇게 하면 변경될 수 있는 API를 사용하는 데 동의를 암시적으로 표현합니다.

`RequiresPreviewFeaturesAttribute`

RequiresPreviewFeaturesAttribute 특성은 런타임, C# 컴파일러 및 라이브러리를 포함하여 스택 전체에서 미리 보기 동작이 필요한 API에 사용됩니다. 이 특성으로 표시된 API를 사용하는 경우 프로젝트 파일에 속성 `<EnablePreviewFeatures>true</EnablePreviewFeatures>`포함되어 있지 않으면 빌드 오류가 발생합니다. 해당 속성을 `true`으로 설정하면 `<LangVersion>Preview</LangVersion>`도 설정되어 미리 보기 언어 기능을 사용할 수 있습니다.

예를 들어 .NET 6에서 *제네릭 수학* 라이브러리는 당시 미리 보기 상태였던 정적 인터페이스 멤버가 필요했기 때문에 RequiresPreviewFeaturesAttribute 표시되었습니다.

`ExperimentalAttribute`

.NET 8은 런타임 또는 언어 미리 보기 기능이 필요하지 않고 지정된 API가 아직 안정적이지 않음을 나타내는 ExperimentalAttribute추가되었습니다.

실험적 API에 대해 빌드할 때 컴파일러는 오류를 생성합니다. 실험적으로 표시된 각 기능에는 별도의 진단 ID가 있습니다. 실험적인 API 사용에 동의하려면 특정 진단을 억제하세요. 진단을 표시하지 않는 방법을 통해 수행할 수 있지만 프로젝트의 `<NoWarn>` 속성에 진단을 추가하는 것이 좋습니다.

각 실험적 기능에는 별도의 ID가 있으므로 하나의 실험적 기능 사용에 동의해도 다른 기능을 사용하는 것에 동의하지 않습니다.

자세한 내용은 실험적 기능 및 일반 특성대한 C# 가이드의 문서를 참조하세요.

라이브러리 개발자를 위한 지침

라이브러리 개발자는 일반적으로 API가 사전 공개 상태에 있다는 것을 다음 두 가지 방법 중 하나로 표현해야 합니다.

- 패키지의 *시험판* 버전에 도입된 새 API의 경우 아무 작업도 수행할 필요가 없습니다. 패키지는 이미 미리 보기 품질을 표현합니다.

- 일부 미리 보기 품질 API가 포함된 안정적인 패키지를 제공하려면 `[Experimental]` 사용하여 해당 API를 표시해야 합니다. 사용자 고유의 진단 ID 사용하고 해당 기능과 관련이 있는지 확인합니다. 독립적인 기능이 여러 개인 경우 여러 ID를 사용하는 것이 좋습니다.

`[RequiresPreviewFeatures]` 특성은 .NET 플랫폼 자체의 구성 요소에만 사용됩니다. 심지어 런타임 및 언어 미리 보기 기능이 필요한 API에만 사용됩니다. 미리 보기에 있는 API인 경우 .NET 플랫폼은 `[Experimental]` 특성을 사용합니다.

이 규칙의 예외는 안정적인 라이브러리를 빌드하고 런타임 또는 언어 미리 보기 동작에 따라 달라지는 특정 기능을 노출하려는 경우입니다. 이 경우 해당 기능의 진입점에 `[RequiresPreviewFeatures]` 사용해야 합니다. 그러나 이러한 API의 사용자도 모든 런타임, 라이브러리 및 언어 미리 보기 동작에 노출되는 미리 보기 기능을 켜야 한다는 점을 고려해야 합니다.

개요: .NET에서 숫자, 날짜, 열거형 및 기타 형식의 형식을 지정하는 방법

서식 지정은 클래스나 구조체의 인스턴스 또는 열거형 값을 문자열 표현으로 변환하는 프로세스입니다. 목적은 결과 문자열을 사용자에게 표시하거나 나중에 역직렬화하여 원래 데이터 형식을 복원하는 것입니다. 이 문서에서는 .NET이 제공하는 서식 지정 메커니즘을 소개합니다.

> **참고**
>
> 구문 분석은 형식 지정과 반대 과정으로 진행됩니다. 구문 분석 작업은 해당 문자열 표현에서 데이터 형식의 인스턴스를 만듭니다. 자세한 내용은 **문자열 구문 분석**을 참조하세요. 직렬화 및 역직렬화에 대한 자세한 내용은 **.NET의 직렬화**를 참조하세요.

서식 지정에 대한 기본 메커니즘은 메서드의 Object.ToString 기본 구현이며, 이 메서드는 이 문서의 뒷부분에 있는 ToString 메서드를 사용하는 기본 서식 섹션에서 설명합니다. 그러나 .NET에서는 기본 형식 지정 지원을 수정하고 확장할 수 있는 여러 가지 방법을 제공합니다. 여기에는 다음이 포함됩니다.

- Object.ToString 메서드를 재정의하여 개체의 값에 대한 사용자 지정 문자열 표현을 정의합니다. 자세한 내용은 이 문서의 뒷부 분에 있는 ToString 메서드 재정의 섹션을 참조하세요.
- 개체의 값에 대한 문자열 표현에서 여러 형식을 사용할 수 있도록 형식 지정자를 정의합니다. 예를 들어, 다음 문의 "X" 형식 지정자는 정수를 16진수 값의 문자열 표현으로 변환합니다.

  C#

  ```
  int integerValue = 60312;
  Console.WriteLine(integerValue.ToString("X"));   // Displays EB98.
  ```

  형식 지정자에 대한 자세한 내용은 ToString 메서드 및 형식 문자열 단원을 참조하세요.

- 서식 공급자를 사용하여 특정 문화권의 서식 지정 규칙을 구현합니다. 예를 들어, 다음 문은 en-US 문화권의 형식 지정 규칙을 사용하여 통화 값을 표시합니다.

  C#

  ```
  double cost = 1632.54;
  Console.WriteLine(cost.ToString("C",
                    new System.Globalization.CultureInfo("en-US")));
  // The example displays the following output:
  //       $1,632.54
  ```

형식 공급자를 사용하여 형식을 지정하는 방법에 대한 자세한 내용은 형식 공급자 섹션을 참조하세요.

- IFormattable 인터페이스를 구현하여 Convert 클래스를 사용하여 문자열 변환과 복합 형식 지정을 지원합니다. 자세한 내용은 IFormattable 인터페이스 단원을 참조하세요.
- 복합 형식 지정을 사용하여 값의 문자열 표현을 더 큰 문자열에 포함합니다. 자세한 내용은 복합 형식 지정 단원을 참조하세요.
- 문자열 보간을 사용하면 보다 읽기 쉬운 구문으로 더 큰 문자열에 값의 문자열 표현을 포함할 수 있습니다. 자세한 내용은 문자열 보간을 참조하세요.
- ICustomFormatter 와 IFormatProvider 를 구현하여 완벽한 사용자 지정 형식 지정 솔루션을 제공합니다. 자세한 내용은 ICustomFormatter를 사용한 사용자 지정 형식 지정 단원을 참조하세요.

다음 단원에서는 개체를 문자열 표현으로 변환하는 데 대해 이러한 메서드를 검토합니다.

ToString 메서드를 사용한 기본 서식 지정

System.Object 에서 파생되는 모든 형식은 기본적으로 형식의 이름을 반환하는 매개 변수가 없는 `ToString` 메서드를 상속합니다. 다음 예제에서는 기본 `ToString` 메서드를 보여 줍니다. 이 예제에서는 구현이 없는 `Automobile` 이라는 클래스를 정의합니다. 클래스가 인스턴스화되고 `ToString` 메서드가 호출되면 해당 형식 이름이 표시됩니다. 이 메서드는 `ToString` 예제에서 명시적으로 호출되지 않습니다. Console.WriteLine(Object) 메서드는 인수로 전달되는 개체의 `ToString` 메서드를 암시적으로 호출합니다.

C#

```
using System;

public class Automobile
{
   // No implementation. All members are inherited from Object.
}

public class Example9
{
   public static void Main()
   {
      Automobile firstAuto = new Automobile();
      Console.WriteLine(firstAuto);
   }
}
// The example displays the following output:
//       Automobile
```

> ⚠ **경고**
>
> Windows 8.1부터 Windows 런타임에는 기본 형식 지원을 제공하는 단일 메서드인 IStringable이 포함된 인터페이스가 포함됩니다. 체계적으로 관리되는 형식은 `IStringable` 인터페이스를 구현하지 않도록 권장합니다. 자세한 내용은 **Windows 런타임 및 IStringable 인터페이스를 참조하세요**.

인터페이스를 제외한 모든 형식이 Object에서 파생되기 때문에 이 함수는 사용자 지정 클래스 또는 구조체에 자동으로 제공됩니다. 그러나 기본 `ToString` 메서드에서 제공하는 기능은 제한되어 있기 때문에 형식을 정의하더라도 해당 형식의 인스턴스에 대한 정보를 제공할 수 없습니다. 이 개체에 대한 정보를 제공하는 개체의 문자열 표현을 제공하려면 `ToString` 메서드를 재정의해야 합니다.

> ⓘ **참고**
>
> 구조체는 ValueType에서 상속되며, 이 형식은 다시 Object에서 파생됩니다. ValueType 이 Object.ToString을 재정의해도 해당 구현은 동일합니다.

ToString 메서드 재정의

형식의 이름을 표시하는 것은 사용이 제한되는 경우가 많으며 형식의 소비자가 한 인스턴스를 다른 인스턴스와 구분하는 것을 허용하지 않습니다. 그러나 `ToString` 메서드를 재정의하여 개체 값을 더 유용하게 표현할 수 있습니다. 다음 예제에서는 `Temperature` 개체를 정의하고 `ToString` 메서드를 재정의하여 온도를 섭씨 단위로 표시합니다.

C#

```
public class Temperature
{
    private decimal temp;

    public Temperature(decimal temperature)
    {
        this.temp = temperature;
    }

    public override string ToString()
    {
        return this.temp.ToString("N1") + "°C";
    }
}

public class Example12
```

```
{
    public static void Main()
    {
        Temperature currentTemperature = new Temperature(23.6m);
        Console.WriteLine($"The current temperature is {currentTemperature}");
    }
}
// The example displays the following output:
//       The current temperature is 23.6°C.
```

.NET에서 각 기본 값 형식의 `ToString` 메서드는 개체의 이름 대신 개체의 값을 표시하도록 재정의되었습니다. 다음 표에는 각 기본 형식의 오버라이드가 나와 있습니다. 재정의된 대부분의 메서드는 다른 오버로드의 `ToString` 메서드를 호출하여 형식의 일반 형식을 지정하는 "G" 형식 지정자와 현재 문화권을 나타내는 IFormatProvider 개체를 전달합니다.

테이블 확장

유형	ToString 재정의
Boolean	Boolean.TrueString 또는 Boolean.FalseString을 반환합니다.
Byte	`Byte.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Byte 값의 형식을 지정합니다.
Char	문자를 문자열로 반환합니다.
DateTime	`DateTime.ToString("G", DatetimeFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 날짜 및 시간 값의 형식을 지정합니다.
Decimal	`Decimal.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Decimal 값의 형식을 지정합니다.
Double	`Double.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Double 값의 형식을 지정합니다.
Int16	`Int16.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Int16 값의 형식을 지정합니다.
Int32	`Int32.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Int32 값의 형식을 지정합니다.
Int64	`Int64.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Int64 값의 형식을 지정합니다.
SByte	`SByte.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 SByte 값의 형식을 지정합니다.
Single	`Single.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 Single 값의 형식을 지정합니다.

유형	ToString 재정의
UInt16	`UInt16.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 UInt16 값의 형식을 지정합니다.
UInt32	`UInt32.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 UInt32 값의 형식을 지정합니다.
UInt64	`UInt64.ToString("G", NumberFormatInfo.CurrentInfo)` 을 호출하여 현재 문화권에 대한 UInt64 값의 형식을 지정합니다.

ToString 메서드 및 서식 문자열

개체에 문자열 표현이 하나만 있는 경우에는 기본 `ToString` 메서드를 사용하거나 `ToString` 을 재정의합니다. 하지만 개체의 값에 여러 표현이 있는 경우가 자주 있습니다. 예를 들어 온도는 화씨, 섭씨 또는 캘빈으로 표시할 수 있습니다. 마찬가지로 정수 값 10도 10, 10.0, 1.0e01, $10.00 등과 같은 여러 가지 방식으로 표현될 수 있습니다.

하나의 값에 여러 문자열 표현을 허용하기 위해 .NET에서는 형식 문자열을 사용합니다. 형식 문자열은 하나 이상의 미리 정의된 형식 지정자가 들어 있는 문자열이며, 이러한 형식 지정자는 `ToString` 메서드가 해당 출력의 형식을 지정하는 방식을 정의한 단일 문자 또는 문자 그룹입니다. 형식 문자열은 개체의 `ToString` 메서드에 매개 변수로 전달되어 개체의 값에 대한 문자열 표현의 표시 방법을 결정합니다.

.NET의 모든 숫자 형식, 날짜/시간 형식 및 열거형 형식은 미리 정의된 형식 지정자 집합을 지원합니다. 형식 문자열을 사용하여 애플리케이션 정의 데이터 형식의 여러 문자열 표현도 정의할 수 있습니다.

표준 서식 문자열

표준 서식 문자열에는 적용되는 개체의 문자열 표현을 정의하는 알파벳 문자인 단일 서식 지정자와 결과 문자열에 표시되는 자릿수에 영향을 주는 선택적 정밀도 지정자가 포함되어 있습니다. 정밀도를 지정하는 포맷이 생략되었거나 지원되지 않는 경우, 표준 형식 지정자는 표준 형식 문자열과 동일합니다.

.NET에서는 모든 숫자 형식, 날짜/시간 형식 및 열거형 형식에 대한 표준 형식 지정자 집합을 정의합니다. 예를 들어, 이러한 각 범주는 해당 형식 값에 대한 일반적인 문자열 표현을 정의하는 "G" 표준 형식 지정자를 지원합니다.

열거형 형식의 표준 형식 문자열은 값의 문자열 표현을 직접 제어합니다. 열거형 값의 `ToString` 메서드에 전달된 형식 문자열은 값이 문자열 이름("G" 및 "F" 형식 지정자), 내부 정수 값("D" 형식 지정자) 또는 16진수 값("X" 형식 지정자)을 사용하여 표시되는지 여부를 결정합니다. 다음 예

제에서는 표준 형식 문자열을 사용하여 DayOfWeek 열거형 값의 형식을 지정하는 방법을 보여 줍니다.

C#

```
DayOfWeek thisDay = DayOfWeek.Monday;
string[] formatStrings = {"G", "F", "D", "X"};

foreach (string formatString in formatStrings)
   Console.WriteLine(thisDay.ToString(formatString));
// The example displays the following output:
//       Monday
//       Monday
//       1
//       00000001
```

열거형 형식 문자열에 대한 자세한 내용은 Enumeration Format Strings을 참조하세요.

일반적으로 숫자 형식의 표준 형식 문자열은 정확한 모양이 하나 이상의 속성 값에 의해 제어되는 결과 문자열을 정의합니다. 예를 들어, "C" 형식 지정자는 숫자의 형식을 통화 값으로 지정합니다. "C" 형식 지정자를 유일한 매개 변수로 사용하여 `ToString` 메서드를 호출하면 현재 문화권의 NumberFormatInfo 개체에 있는 다음 속성 값이 숫자 값의 문자열 표현을 정의하는 데 사용됩니다.

- 현재 문화권의 통화 기호를 지정하는 CurrencySymbol 속성
- 다음과 같은 사항을 결정하는 정수를 반환하는 CurrencyNegativePattern 또는 CurrencyPositivePattern 속성
  - 통화 기호의 위치
  - 음수 값이 선행 음수 기호, 후행 음수 기호 또는 괄호로 표시되는지 여부
  - 숫자 값과 통화 기호 사이에 공백이 표시되는지 여부
- 결과 문자열의 소수 자릿수를 정의하는 CurrencyDecimalDigits 속성
- 결과 문자열의 소수 구분 기호를 정의하는 CurrencyDecimalSeparator 속성
- 그룹 구분 기호를 정의하는 CurrencyGroupSeparator 속성
- 소수점 왼쪽의 각 그룹에 있는 자릿수를 정의하는 CurrencyGroupSizes 속성
- NegativeSign 괄호가 음수 값을 나타내는 데 사용되지 않는 경우 결과 문자열에 사용되는 음수 기호를 결정하는 속성입니다.

또한 숫자 서식 문자열에는 정밀도 지정자가 포함될 수 있습니다. 이 지정자의 의미는 사용되는 형식 문자열에 따라 달라지지만 일반적으로 총 자릿수 또는 결과 문자열에 표시되어야 하는 소

수 자릿수를 나타냅니다. 예를 들어, 다음 예제에서는 "X4" 표준 숫자 문자열과 전체 자릿수 지정자를 사용하여 네 자리의 16진수로 구성된 문자열 값을 만듭니다.

C#

```
byte[] byteValues = { 12, 163, 255 };
foreach (byte byteValue in byteValues)
   Console.WriteLine(byteValue.ToString("X4"));
// The example displays the following output:
//       000C
//       00A3
//       00FF
```

표준 숫자 서식 지정 문자열에 대한 자세한 내용은 표준 날짜 및 시간 형식 문자열을 참조하세요.

날짜 및 시간 값의 표준 형식 문자열은 특정 DateTimeFormatInfo 속성에 저장된 사용자 지정 형식 문자열의 별칭입니다. 예를 들어 "D" 형식 지정자를 사용하여 날짜 및 시간 값의 `ToString` 메서드를 호출하면 현재 문화권의 DateTimeFormatInfo.LongDatePattern 속성에 저장된 사용자 지정 형식 문자열을 사용하여 날짜 및 시간이 표시됩니다. (사용자 지정 서식 문자열에 대한 자세한 내용은 다음 섹션을 참조하세요.) 다음 예제에서는 이러한 관계를 보여 줍니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime date1 = new DateTime(2009, 6, 30);
      Console.WriteLine($"D Format Specifier:     {date1:D}");
      string longPattern =
CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern;
      Console.WriteLine($"'{longPattern}' custom format string:
{date1.ToString(longPattern)}");
   }
}
// The example displays the following output when run on a system whose
// current culture is en-US:
//    D Format Specifier:     Tuesday, June 30, 2009
//    'dddd, MMMM dd, yyyy' custom format string:     Tuesday, June 30, 2009
```

표준 날짜 및 시간 형식 문자열에 대한 자세한 내용은 Standard Date and Time Format Strings을 참조하세요.

표준 형식 문자열을 사용하여 개체의 `ToString(String)` 메서드에 의해 생성되는 애플리케이션 정의 개체의 문자열 표현도 정의할 수 있습니다. 개체에서 지원하는 특정 표준 형식 지정자를 정

의할 수 있으며 대/소문자를 구분하는지 또는 대/소문자를 구분하지 않는지 확인할 수 있습니다. `ToString(String)` 메서드의 구현에서는 다음을 지원해야 합니다.

- 개체의 사용자 지정 또는 일반 형식을 나타내는 "G" 형식 지정자. 개체의 `ToString` 메서드에 대한 매개 변수가 없는 오버로드는 해당 `ToString(String)` 오버로드를 호출하고 이를 "G" 표준 형식 문자열에 전달해야 합니다.

- Visual Basic에서는 null 참조(`Nothing`)와 같은 형식 지정자에 대한 지원이 있습니다. null 참조와 같은 형식 지정자는 "G" 형식 지정자와 동일한 것으로 간주되어야 합니다.

예를 들어 `Temperature` 클래스는 섭씨 온도를 내부적으로 저장하고 형식 지정자를 사용하여 `Temperature` 개체의 값을 섭씨, 화씨 및 캘빈으로 표시할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
using System;

public class Temperature
{
   private decimal m_Temp;

   public Temperature(decimal temperature)
   {
      this.m_Temp = temperature;
   }

   public decimal Celsius
   {
      get { return this.m_Temp; }
   }

   public decimal Kelvin
   {
      get { return this.m_Temp + 273.15m; }
   }

   public decimal Fahrenheit
   {
      get { return Math.Round(((decimal) (this.m_Temp * 9 / 5 + 32)), 2); }
   }

   public override string ToString()
   {
      return this.ToString("C");
   }

   public string ToString(string format)
   {
      // Handle null or empty string.
      if (String.IsNullOrEmpty(format)) format = "C";
```

```
      // Remove spaces and convert to uppercase.
      format = format.Trim().ToUpperInvariant();

      // Convert temperature to Fahrenheit and return string.
      switch (format)
      {
         // Convert temperature to Fahrenheit and return string.
         case "F":
            return this.Fahrenheit.ToString("N2") + " °F";
         // Convert temperature to Kelvin and return string.
         case "K":
            return this.Kelvin.ToString("N2") + " K";
         // return temperature in Celsius.
         case "G":
         case "C":
            return this.Celsius.ToString("N2") + " °C";
         default:
            throw new FormatException(String.Format("The '{0}' format string is not
supported.", format));
      }
   }
}

public class Example1
{
   public static void Main()
   {
      Temperature temp1 = new Temperature(0m);
      Console.WriteLine(temp1.ToString());
      Console.WriteLine(temp1.ToString("G"));
      Console.WriteLine(temp1.ToString("C"));
      Console.WriteLine(temp1.ToString("F"));
      Console.WriteLine(temp1.ToString("K"));

      Temperature temp2 = new Temperature(-40m);
      Console.WriteLine(temp2.ToString());
      Console.WriteLine(temp2.ToString("G"));
      Console.WriteLine(temp2.ToString("C"));
      Console.WriteLine(temp2.ToString("F"));
      Console.WriteLine(temp2.ToString("K"));

      Temperature temp3 = new Temperature(16m);
      Console.WriteLine(temp3.ToString());
      Console.WriteLine(temp3.ToString("G"));
      Console.WriteLine(temp3.ToString("C"));
      Console.WriteLine(temp3.ToString("F"));
      Console.WriteLine(temp3.ToString("K"));

      Console.WriteLine(String.Format("The temperature is now {0:F}.", temp3));
   }
}
// The example displays the following output:
//       0.00 °C
//       0.00 °C
//       0.00 °C
```

```
//       32.00 °F
//       273.15 K
//       -40.00 °C
//       -40.00 °C
//       -40.00 °C
//       -40.00 °F
//       233.15 K
//       16.00 °C
//       16.00 °C
//       16.00 °C
//       60.80 °F
//       289.15 K
//       The temperature is now 16.00 °C.
```

사용자 지정 서식 문자열

표준 형식 문자열 외에도 .NET에서는 숫자 값과 날짜 및 시간 값에 대한 사용자 지정 형식 문자열을 정의합니다. 사용자 지정 형식 문자열은 값의 문자열 표현을 정의하는 하나 이상의 사용자 지정 형식 지정자로 구성됩니다. 예를 들어, en-US 문화권의 경우 사용자 지정 날짜 및 시간 형식 문자열 "yyyy/mm/dd hh:mm:ss t zzz"는 날짜를 "2008/11/15 07:45:00.0000 P -08:00" 형태의 문자열 표현으로 변환합니다. 마찬가지로 사용자 지정 형식 문자열 "0000"은 정수 값 12를 "0012"로 변환합니다. 사용자 지정 서식 문자열의 전체 목록은 사용자 지정 날짜 및 시간 서식 문자열 및 사용자 지정 숫자 형식 문자열을 참조하세요.

형식 문자열이 단일 사용자 지정 형식 지정자로 구성된 경우에는 표준 형식 지정자와 혼동되지 않도록 형식 지정자 앞에 백분율 기호(%)가 와야 합니다. 다음 예제에서는 "M" 사용자 지정 형식 지정자를 사용하여 특정 날짜의 월에 해당하는 한 자리 또는 두 자리 숫자를 표시합니다.

C#

```
DateTime date1 = new DateTime(2009, 9, 8);
Console.WriteLine(date1.ToString("%M"));       // Displays 9
```

많은 표준 형식 문자열은 DateTimeFormatInfo 개체의 속성에 정의된 사용자 지정 형식 문자열의 별칭입니다. 또한 사용자 지정 형식 문자열을 사용하면 상당히 융통성 있게 숫자 값 또는 날짜 및 시간 값에 대한 애플리케이션 정의 형식 지정 기능을 제공할 수 있습니다. 여러 사용자 지정 형식 지정자를 하나의 사용자 지정 형식 문자열로 결합하여 숫자 값과 날짜 및 시간 값에 대한 사용자 지정 결과 문자열을 정의할 수 있습니다. 다음 예제에서는 월 이름, 일, 연도 뒤에 괄호로 묶은 요일을 표시하는 사용자 지정 형식 문자열을 정의합니다.

C#

```
string customFormat = "MMMM dd, yyyy (dddd)";
DateTime date1 = new DateTime(2009, 8, 28);
Console.WriteLine(date1.ToString(customFormat));
// The example displays the following output if run on a system
```

```
// whose language is English:
//       August 28, 2009 (Friday)
```

다음 예제에서는 Int64 값을 7자리 표준 미국 전화 번호로 지역 번호와 함께 표시하는 사용자 지정 서식 문자열을 정의합니다.

C#

```
using System;

public class Example17
{
   public static void Main()
   {
      long number = 8009999999;
      string fmt = "000-000-0000";
      Console.WriteLine(number.ToString(fmt));
   }
}
// The example displays the following output:
//        800-999-9999
```

표준 서식 문자열은 일반적으로 애플리케이션 정의 형식에 대한 대부분의 서식 요구 사항을 처리할 수 있지만 사용자 지정 형식 지정자를 정의하여 형식의 서식을 지정할 수도 있습니다.

서식 문자열 및 .NET 형식

모든 숫자 형식(즉, Byte, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32,UInt64 및 BigInteger 형식)과 DateTime, DateTimeOffset, TimeSpan, Guid 및 모든 열거형 형식은 서식 문자열을 사용한 서식 지정을 지원합니다. 각 형식에서 지원하는 특정 형식 문자열에 대한 자세한 내용은 다음 문서를 참조하세요.

테이블 확장

타이틀	정의
표준 숫자 형식 문자열	숫자 값의 일반적으로 사용되는 문자열 표현을 만드는 표준 형식 문자열에 대해 설명합니다.
사용자 지정 숫자 형식 문자열	숫자 값의 애플리케이션별 형식을 만드는 사용자 지정 형식 문자열에 대해 설명합니다.
표준 날짜 및 시간 형식 문자열	DateTime 및 DateTimeOffset 값의 일반적으로 사용되는 문자열 표현을 만드는 표준 서식 문자열에 대해 설명합니다.
사용자 지정 날짜 및 시간 형식 문자열	DateTime 및 DateTimeOffset 값의 애플리케이션별 서식을 만드는 사용자 지정 서식 문자열에 대해 설명합니다.

타이틀	정의
표준 TimeSpan 서식 문자열	시간 간격의 일반적으로 사용되는 문자열 표현을 만드는 표준 형식 문자열에 대해 설명합니다.
사용자 지정 TimeSpan 서식 문자열	시간 간격의 애플리케이션별 형식을 만드는 사용자 지정 형식 문자열에 대해 설명합니다.
열거형 형식 문자열	열거형 값의 문자열 표현을 만드는 데 사용되는 표준 형식 문자열에 대해 설명합니다.
Guid.ToString(String)	Guid 값에 대한 표준 형식 문자열에 대해 설명합니다.

형식 공급자를 사용한 문화권 구분 형식 지정

형식 지정자를 사용하여 개체의 형식 지정을 사용자 지정할 수 있기는 하지만 의미 있는 개체의 문자열 표현을 만들려면 추가 형식 지정 정보가 필요한 경우가 종종 있습니다. 예를 들어, C" 표준 형식 문자열이나 "$ #,#.00" 같은 사용자 지정 형식 문자열을 사용하여 숫자의 형식을 통화 값으로 지정하려면 최소한 올바른 통화 기호, 그룹 구분 기호 및 소수 구분 기호에 대한 정보를 형식 지정된 문자열에 포함할 수 있어야 합니다. .NET에서 이 추가 서식 지정 정보는 IFormatProvider 인터페이스를 통해 제공되며, 이 인터페이스는 숫자 형식과 날짜 및 시간 형식의 `ToString` 메서드의 하나 이상의 오버로드에 대한 매개 변수로 제공됩니다. IFormatProvider 구현은 .NET에서 문화권별 서식 지정을 지원하는 데 사용됩니다. 다음 예제에서는 서로 다른 문화권을 나타내는 세 IFormatProvider 개의 개체로 서식이 지정될 때 개체의 문자열 표현이 어떻게 변경되는지 보여 줍니다.

C#

```
using System;
using System.Globalization;

public class Example18
{
   public static void Main()
   {
      decimal value = 1603.42m;
      Console.WriteLine(value.ToString("C3", new CultureInfo("en-US")));
      Console.WriteLine(value.ToString("C3", new CultureInfo("fr-FR")));
      Console.WriteLine(value.ToString("C3", new CultureInfo("de-DE")));
   }
}
// The example displays the following output:
//       $1,603.420
//       1 603,420 €
//       1.603,420 €
```

IFormatProvider 인터페이스에는 GetFormat(Type)메서드가 하나 있으며, 이 메서드에는 형식 지정 정보를 제공하는 개체의 형식을 지정하는 매개 변수가 하나 있습니다. 이 메서드는 해당 형식의 개체를 제공할 수 있는 경우 해당 형식의 개체를 반환합니다. 그러지 않으면 null 참조(Visual Basic의 경우 `Nothing`)를 반환합니다.

IFormatProvider.GetFormat 은 콜백 메서드입니다. `ToString` 매개 변수가 포함된 IFormatProvider 메서드 오버로드를 호출하면 해당 GetFormat 개체의 IFormatProvider 메서드가 호출됩니다. GetFormat 메서드는 `formatType` 매개 변수에 지정된 대로 필요한 형식 지정 정보를 제공하는 개체를 `ToString` 메서드에 반환합니다.

여러 형식 지정 또는 문자열 변환 메서드는 형식 IFormatProvider의 매개 변수를 포함하지만 대부분의 경우 메서드가 호출될 때 매개 변수 값이 무시됩니다. 다음 표에서는 매개 변수를 사용하는 형식 지정 메서드와 이러한 메서드가 Type 메서드로 전달하는 IFormatProvider.GetFormat 개체의 형식을 보여 줍니다.

테이블 확장

메서드	`formatType` 매개 변수의 형식
숫자 형식의 `ToString` 메서드	System.Globalization.NumberFormatInfo
날짜 및 시간 형식의 `ToString` 메서드	System.Globalization.DateTimeFormatInfo
String.Format	System.ICustomFormatter
StringBuilder.AppendFormat	System.ICustomFormatter

> **참고**
>
> 숫자 형식과 날짜 및 시간 형식의 `ToString` 메서드는 오버로드되며, 일부 오버로드에만 IFormatProvider 매개 변수가 포함됩니다. 메서드에 형식 IFormatProvider의 매개 변수가 없으면 속성에서 CultureInfo.CurrentCulture 반환되는 개체가 대신 전달됩니다. 예를 들어, 기본 Int32.ToString() 메서드를 호출하면 결과적으로 `Int32.ToString("G", System.Globalization.CultureInfo.CurrentCulture)` 같은 메서드가 호출됩니다.

.NET에는 IFormatProvider를 구현하는 세 가지 클래스가 있습니다.

- 특정 문화권의 날짜 및 시간 값에 대한 형식 지정 정보를 제공하는DateTimeFormatInfo클래스. 이 클래스에 대해 IFormatProvider.GetFormat 을 구현하면 해당 클래스의 인스턴스가 반환됩니다.
- 특정 문화권의 숫자 형식 지정 정보를 제공하는NumberFormatInfo클래스. 이 클래스에 대해 IFormatProvider.GetFormat 을 구현하면 해당 클래스의 인스턴스가 반환됩니다.

- CultureInfo; 이 클래스에 대해 IFormatProvider.GetFormat 을 구현하면 숫자 형식 지정 정보를 제공하는 NumberFormatInfo 개체 또는 날짜 및 시간 값에 대한 형식 지정 정보를 제공하는 DateTimeFormatInfo 개체가 반환될 수 있습니다.

사용자 고유의 형식 공급자를 구현하여 이러한 클래스 중 하나를 대체할 수도 있습니다. 그러나 GetFormat 메서드에 형식 지정 정보를 제공해야 하는 경우에는 구현된 형식 공급자의 `ToString` 메서드에서 위의 표에 나와 있는 형식의 개체를 반환해야 합니다.

숫자 값의 문화적 적응 서식 지정

기본적으로 숫자 값의 형식은 문화에 민감하게 설정되어 있습니다. 서식 지정 메서드를 호출할 때 문화권을 지정하지 않으면 현재 문화권의 서식 규칙이 사용됩니다. 이는 현재 문화권을 4번 변경한 후 Decimal.ToString(String) 메서드를 호출하는 다음 예제에 나와 있습니다. 각각의 경우 결과 문자열은 현재 문화권의 형식 규칙을 반영합니다. 이는 `ToString` 및 `ToString(String)` 메서드가 각 숫자 형식의 `ToString(String, IFormatProvider)` 메서드에 대한 호출을 래핑하기 때문입니다.

C#

```
using System.Globalization;

public class Example6
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
      Decimal value = 1043.17m;

      foreach (var cultureName in cultureNames) {
         // Change the current culture.
         CultureInfo.CurrentCulture =
CultureInfo.CreateSpecificCulture(cultureName);
         Console.WriteLine($"The current culture is
{CultureInfo.CurrentCulture.Name}");
         Console.WriteLine(value.ToString("C2"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The current culture is en-US
//       $1,043.17
//
//       The current culture is fr-FR
//       1 043,17 €
//
//       The current culture is es-MX
//       $1,043.17
//
```

```
//       The current culture is de-DE
//       1.043,17 €
```

`ToString` 매개 변수를 포함하는 `provider` 오버로드를 호출하고 여기에 다음 중 하나를 전달하여 특정 문화권에 대한 숫자 값의 형식을 지정할 수도 있습니다.

- 사용할 형식 규칙을 결정하는 문화에 해당하는 CultureInfo 개체. 해당 CultureInfo.GetFormat 메서드는 숫자 값에 대한 문화권별 형식 정보를 제공하는 CultureInfo.NumberFormat 개체인 NumberFormatInfo 속성의 값을 반환합니다.
- 사용할 문화권별 형식 규칙을 정의하는 NumberFormatInfo 개체. 해당 GetFormat 메서드는 자신의 인스턴스를 반환합니다.

다음 예제에서는 부동 소수점 수의 형식을 지정하기 위해 영어(미국) 및 영어(영국) 문화권과 프랑스어 및 러시아어 중립 문화권을 나타내는 NumberFormatInfo 개체를 사용합니다.

C#

```
using System.Globalization;

public class Example7
{
    public static void Main()
    {
        double value = 1043.62957;
        string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

        foreach (string? name in cultureNames)
        {
            NumberFormatInfo nfi =
CultureInfo.CreateSpecificCulture(name).NumberFormat;
            Console.WriteLine("{0,-6} {1}", name + ":", value.ToString("N3", nfi));
        }
    }
}
// The example displays the following output:
//       en-US: 1,043.630
//       en-GB: 1,043.630
//       ru:    1 043,630
//       fr:    1 043,630
```

날짜 및 시간 값의 문화권 구분 서식 지정

기본적으로 날짜 및 시간 값의 형식은 문화에 민감하게 설정됩니다. 서식 지정 메서드를 호출할 때 문화권을 지정하지 않으면 현재 문화권의 서식 규칙이 사용됩니다. 이는 현재 문화권을 4번 변경한 후 DateTime.ToString(String) 메서드를 호출하는 다음 예제에 나와 있습니다. 각각의 경우 결과 문자열은 현재 문화권의 형식 규칙을 반영합니다. 이는 DateTime.ToString(), DateTime.ToString(String), DateTimeOffset.ToString()및 DateTimeOffset.ToString(String) 메서드

가 DateTime.ToString(String, IFormatProvider) 및 DateTimeOffset.ToString(String, IFormatProvider) 메서드에 대한 호출을 래핑하기 때문입니다.

C#

```
using System.Globalization;

public class Example4
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
      DateTime dateToFormat = new DateTime(2012, 5, 28, 11, 30, 0);

      foreach (var cultureName in cultureNames) {
         // Change the current culture.
         CultureInfo.CurrentCulture =
CultureInfo.CreateSpecificCulture(cultureName);
         Console.WriteLine($"The current culture is
{CultureInfo.CurrentCulture.Name}");
         Console.WriteLine(dateToFormat.ToString("F"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The current culture is en-US
//       Monday, May 28, 2012 11:30:00 AM
//
//       The current culture is fr-FR
//       lundi 28 mai 2012 11:30:00
//
//       The current culture is es-MX
//       lunes, 28 de mayo de 2012 11:30:00 a.m.
//
//       The current culture is de-DE
//       Montag, 28. Mai 2012 11:30:00
```

DateTime.ToString 매개 변수를 포함하는 DateTimeOffset.ToString 또는 `provider` 오버로드를 호출하고 여기에 다음 중 하나를 전달하여 특정 문화권에 대한 날짜 및 시간 값의 형식을 지정할 수도 있습니다.

- 사용할 형식 규칙을 결정하는 문화에 해당하는 CultureInfo 개체. 해당 CultureInfo.GetFormat 메서드는 날짜 및 시간 값에 대한 문화권별 형식 정보를 제공하는 CultureInfo.DateTimeFormat 개체인 DateTimeFormatInfo 속성의 값을 반환합니다.
- 사용할 문화권별 형식 규칙을 정의하는 DateTimeFormatInfo 개체. 해당 GetFormat 메서드는 자신의 인스턴스를 반환합니다.

다음 예에서는 영어(미국) 및 영어(영국) 문화권과 프랑스어 및 러시아어 중립 문화권을 나타내는 DateTimeFormatInfo 개체를 사용하여 날짜 형식을 지정합니다.

C#

```
using System.Globalization;

public class Example5
{
   public static void Main()
   {
      DateTime dat1 = new(2012, 5, 28, 11, 30, 0);
      string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

      foreach (var name in cultureNames) {
         DateTimeFormatInfo dtfi =
CultureInfo.CreateSpecificCulture(name).DateTimeFormat;
         Console.WriteLine($"{name}: {dat1.ToString(dtfi)}");
      }
   }
}
// The example displays the following output:
//       en-US: 5/28/2012 11:30:00 AM
//       en-GB: 28/05/2012 11:30:00
//       ru: 28.05.2012 11:30:00
//       fr: 28/05/2012 11:30:00
```

IFormattable 인터페이스

일반적으로 형식 문자열과 `ToString` 매개 변수로 IFormatProvider 메서드를 오버로드하는 형식은 IFormattable 인터페이스도 구현합니다. 이 인터페이스에는 형식 문자열과 형식 공급자가 매개 변수로 포함되어 있는 IFormattable.ToString(String, IFormatProvider)이라는 단일 멤버가 있습니다.

애플리케이션 정의 클래스에 대해 IFormattable 인터페이스를 구현하면 다음과 같은 두 가지 이점이 있습니다.

- Convert 클래스를 사용한 문자열 변환을 지원합니다. Convert.ToString(Object) 및 Convert.ToString(Object, IFormatProvider) 메서드를 호출하면 IFormattable 구현이 자동으로 호출됩니다.

- 복합 형식 지정을 지원합니다. 형식 문자열을 포함한 형식 항목이 사용자 지정 형식의 형식을 지정하는 데 사용되면 공용 언어 런타임에서 자동으로 IFormattable 구현을 호출하고 이를 형식 문자열에 전달합니다. String.Format 또는 Console.WriteLine같은 메서드를 사용하는 복합 형식 지정에 대한 자세한 내용은 복합 형식 지정 단원을 참조하세요.

다음 예제에서는 `Temperature` 인터페이스를 구현하는 IFormattable 클래스를 정의합니다. 이 클래스는 온도를 섭씨로 표시하는 "C" 또는 "G" 형식 지정자, 온도를 화씨로 표시하는 "F" 형식 지정자 또는 온도를 켈빈으로 표시하는 "K" 형식 지정자를 지원합니다.

C#

```
using System;
using System.Globalization;

namespace HotAndCold
{

    public class Temperature : IFormattable
    {
        private decimal m_Temp;

        public Temperature(decimal temperature)
        {
            this.m_Temp = temperature;
        }

        public decimal Celsius
        {
            get { return this.m_Temp; }
        }

        public decimal Kelvin
        {
            get { return this.m_Temp + 273.15m; }
        }

        public decimal Fahrenheit
        {
            get { return Math.Round((decimal)this.m_Temp * 9 / 5 + 32, 2); }
        }

        public override string ToString()
        {
            return this.ToString("G", null);
        }

        public string ToString(string format)
        {
            return this.ToString(format, null);
        }

        public string ToString(string format, IFormatProvider provider)
        {
            // Handle null or empty arguments.
            if (String.IsNullOrEmpty(format))
                format = "G";
            // Remove any white space and covert to uppercase.
            format = format.Trim().ToUpperInvariant();
```

```
            if (provider == null)
                provider = NumberFormatInfo.CurrentInfo;

            switch (format)
            {
                // Convert temperature to Fahrenheit and return string.
                case "F":
                    return this.Fahrenheit.ToString("N2", provider) + "°F";
                // Convert temperature to Kelvin and return string.
                case "K":
                    return this.Kelvin.ToString("N2", provider) + "K";
                // Return temperature in Celsius.
                case "C":
                case "G":
                    return this.Celsius.ToString("N2", provider) + "°C";
                default:
                    throw new FormatException(String.Format("The '{0}' format string
is not supported.", format));
            }
        }
    }
```

다음 예제에서는 `Temperature` 개체를 인스턴스화합니다. 그런 다음 ToString 메서드를 호출하고 여러 복합 형식 문자열을 사용하여 `Temperature` 개체의 다양한 문자열 표현을 얻습니다. 이 메서드를 호출할 때마다 자동으로 IFormattable 클래스의 `Temperature` 구현도 호출됩니다.

C#

```
public class Example11
{
    public static void Main()
    {
        CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
        Temperature temp = new Temperature(22m);
        Console.WriteLine(Convert.ToString(temp, new CultureInfo("ja-JP")));
        Console.WriteLine($"Temperature: {temp:K}");
        Console.WriteLine($"Temperature: {temp:F}");
        Console.WriteLine(String.Format(new CultureInfo("fr-FR"), "Temperature:
{0:F}", temp));
    }
}
// The example displays the following output:
//       22.00°C
//       Temperature: 295.15K
//       Temperature: 71.60°F
//       Temperature: 71,60°F
```

복합 형식 지정

및 와 같은 String.Format 일부 메서드는 StringBuilder.AppendFormat 지원합니다. 복합 형식 문자열은 0개 이상의 개체에 대한 문자열 표현이 통합된 단일 문자열을 반환하는 일종의 템플릿입니다. 복합 형식 문자열에서는 각 개체가 인덱싱된 형식 항목으로 표현됩니다. 형식 항목의 인덱스는 메서드의 매개 변수 목록에 표시되는 개체의 위치에 해당합니다. 인덱스는 0에서 시작합니다. 예를 들어 다음과 같은 String.Format 메서드 호출에서 첫 번째 형식 항목인 `{0:D}`는 `thatDate`의 문자열 표현으로 바뀌고, 두 번째 형식 항목인 `{1}`은 `item1`의 문자열 표현으로 바뀌고, 세 번째 형식 항목인 `{2:C2}`는 `item1.Value`의 문자열 표현으로 바뀝니다.

C#

```
result = String.Format("On {0:d}, the inventory of {1} was worth {2:C2}.",
                       thatDate, item1, item1.Value);
Console.WriteLine(result);
// The example displays output like the following if run on a system
// whose current culture is en-US:
//       On 5/1/2009, the inventory of WidgetA was worth $107.44.
```

형식 항목을 해당 개체의 문자열 표현으로 바꾸는 것 외에도 형식 항목을 통해 다음을 제어할 수 있습니다.

- 개체가 IFormattable 인터페이스를 구현하고 형식 문자열을 지원하는 경우 개체가 문자열로 표현되는 특정 방식. 이렇게 하려면 형식 항목의 인덱스 뒤에 `:` (콜론) 및 유효한 형식 문자열을 추가합니다. 이전 예제에서는 "d"(짧은 날짜 패턴) 형식 문자열(예: #B0)을 사용하여 날짜 값의 서식을 지정하고 숫자 값을 "C2" 서식 문자열(예: #B1)으로 서식을 지정하여 소수 소수 자릿수가 두 자리인 통화 값으로 표시했습니다.

- 개체의 문자열 표현을 포함하는 필드의 너비 및 해당 필드의 문자열 표현 맞춤. 이렇게 하려면 형식 항목의 인덱스 뒤에 `,`를 쓴 후, 쉼표를 추가하고 필드 너비를 지정합니다. 필드 너비가 양수 값이면 문자열은 필드에서 오른쪽으로 맞춰지고, 필드 너비가 음수 값이면 왼쪽으로 맞춰집니다. 다음 예제에서는 날짜 값을 20자 필드에 왼쪽 맞춤하고, 소수 1자리의 10진수 값을 11자 필드에 오른쪽 맞춤합니다.

  C#

  ```
  DateTime startDate = new DateTime(2015, 8, 28, 6, 0, 0);
  decimal[] temps = { 73.452m, 68.98m, 72.6m, 69.24563m,
                      74.1m, 72.156m, 72.228m };
  Console.WriteLine("{0,-20} {1,11}\n", "Date", "Temperature");
  for (int ctr = 0; ctr < temps.Length; ctr++)
     Console.WriteLine("{0,-20:g} {1,11:N1}", startDate.AddDays(ctr),
  temps[ctr]);

  // The example displays the following output:
  //       Date                 Temperature
  //
  //       8/28/2015 6:00 AM           73.5
  //       8/29/2015 6:00 AM           69.0
  ```

```
//       8/30/2015 6:00 AM           72.6
//       8/31/2015 6:00 AM           69.2
//       9/1/2015 6:00 AM            74.1
//       9/2/2015 6:00 AM            72.2
//       9/3/2015 6:00 AM            72.2
```

맞춤 문자열 구성 요소와 형식 문자열 구성 요소가 모두 있는 경우 전자가 후자보다 우선합니다(예: `{0,-20:g}`).

복합 서식 지정에 대한 자세한 내용은 Composite Formatting을 참조하세요.

ICustomFormatter를 사용한 사용자 지정 서식 지정

두 복합 형식 지정 메서드(String.Format(IFormatProvider, String, Object[]) 및 StringBuilder.AppendFormat(IFormatProvider, String, Object[]))에 사용자 지정 형식을 지원하는 형식 공급자 매개 변수가 포함되어 있습니다. 이러한 형식 지정 메서드 중 하나를 호출하면 Type 인터페이스를 나타내는 ICustomFormatter 개체가 형식 공급자의 GetFormat 메서드에 전달됩니다. 그러면 GetFormat 메서드가 사용자 지정 형식 지정을 제공하는 ICustomFormatter 구현을 반환합니다.

ICustomFormatter 인터페이스에는 복합 형식 지정 메서드에 의해 자동으로 복합 형식 문자열의 각 형식 항목에 대해 한 번씩 호출되는 Format(String, Object, IFormatProvider)라는 메서드가 하나 있습니다. Format(String, Object, IFormatProvider) 메서드에는 세 개의 매개 변수 즉, 형식 항목의 `formatString` 인수를 나타내는 형식 문자열, 형식을 지정할 개체 및 형식 지정 서비스를 제공하는 IFormatProvider 개체가 포함되어 있습니다. 일반적으로 ICustomFormatter 를 구현하는 클래스는 IFormatProvider도 구현하여 이 마지막 매개 변수가 사용자 지정 형식 지정 클래스를 참조하도록 합니다. 이 메서드는 형식을 지정할 개체의 사용자 지정 형식 문자열 표현을 반환합니다. 메서드가 개체의 서식을 지정할 수 없는 경우 null 참조(`Nothing` Visual Basic)를 반환해야 합니다.

다음 예제에서는 정수 값을 뒤에 공백이 오는 두 자리 16진수 값 시퀀스로 표시하는 ICustomFormatter 라는 `ByteByByteFormatter` 구현을 제공합니다.

C#

```
public class ByteByByteFormatter : IFormatProvider, ICustomFormatter
{
    public object? GetFormat(Type? formatType)
    {
        if (formatType == typeof(ICustomFormatter))
            return this;
        else
            return null;
    }

    public string Format(string? format, object? arg,
```

```
                          IFormatProvider? formatProvider)
    {
        if ((formatProvider is not null) && !formatProvider.Equals(this)) return "";

        // Handle only hexadecimal format string.
        if ((format is not null) && !format.StartsWith("X")) return "";

        byte[] bytes;

        // Handle only integral types.
        if (arg is Int16)
            bytes = BitConverter.GetBytes((Int16)arg);
        else if (arg is Int32)
            bytes = BitConverter.GetBytes((Int32)arg);
        else if (arg is Int64)
            bytes = BitConverter.GetBytes((Int64)arg);
        else if (arg is UInt16)
            bytes = BitConverter.GetBytes((UInt16)arg);
        else if (arg is UInt32)
            bytes = BitConverter.GetBytes((UInt32)arg);
        else if (arg is UInt64)
            bytes = BitConverter.GetBytes((UInt64)arg);
        else
            return "";

        string output= "";
        for (int ctr = bytes.Length - 1; ctr >= 0; ctr--)
            output += string.Format("{0:X2} ", bytes[ctr]);

        return output.Trim();
    }
}
```

다음 예제에서는 `ByteByByteFormatter` 클래스를 사용하여 정수 값의 형식을 지정합니다. 이 ICustomFormatter.Format 메서드는 두 번째 String.Format(IFormatProvider, String, Object[]) 메서드 호출에서 두 번 이상 호출되며 NumberFormatInfo 기본 공급자는 세 번째 메서드 호출에서 사용됩니다. `ByteByByteFormatter.Format` 메서드는 "N0" 형식 문자열을 인식하지 못하고 null 참조(`Nothing` Visual Basic)를 반환합니다.

C#

```
public class Example10
{
    public static void Main()
    {
        long value = 3210662321;
        byte value1 = 214;
        byte value2 = 19;

        Console.WriteLine(string.Format(new ByteByByteFormatter(), "{0:X}", value));
        Console.WriteLine(string.Format(new ByteByByteFormatter(), "{0:X} And {1:X}
= {2:X} ({2:000})",
```

```
                                        value1, value2, value1 & value2));
        Console.WriteLine(string.Format(new ByteByByteFormatter(), "{0,10:N0}",
value));
    }
}
// The example displays the following output:
//       00 00 00 00 BF 5E D1 B1
//       00 D6 And 00 13 = 00 12 (018)
//       3,210,662,321
```

참고하십시오

테이블 확장

타이틀	정의
표준 숫자 형식 문자열	숫자 값의 일반적으로 사용되는 문자열 표현을 만드는 표준 형식 문자열에 대해 설명합니다.
사용자 지정 숫자 형식 문자열	숫자 값의 애플리케이션별 형식을 만드는 사용자 지정 형식 문자열에 대해 설명합니다.
표준 날짜 및 시간 형식 문자열	DateTime 값의 일반적으로 사용되는 문자열 표현을 만드는 표준 형식 문자열에 대해 설명합니다.
사용자 지정 날짜 및 시간 형식 문자열	DateTime 값의 애플리케이션별 형식을 만드는 사용자 지정 형식 문자열에 대해 설명합니다.
표준 TimeSpan 서식 문자열	시간 간격의 일반적으로 사용되는 문자열 표현을 만드는 표준 형식 문자열에 대해 설명합니다.
사용자 지정 TimeSpan 서식 문자열	시간 간격의 애플리케이션별 형식을 만드는 사용자 지정 형식 문자열에 대해 설명합니다.
열거형 형식 문자열	열거형 값의 문자열 표현을 만드는 데 사용되는 표준 형식 문자열에 대해 설명합니다.
복합 형식 지정	문자열에 형식이 지정된 하나 이상의 값을 포함시키는 방법에 대해 설명합니다. 그런 후 해당 문자열을 콘솔에 표시하거나 스트림에 쓸 수 있습니다.
문자열 구문 분석	개체를 해당 개체의 문자열 표현으로 표시되는 값으로 초기화하는 방법에 대해 설명합니다. 구문 분석은 형식 지정의 역순으로 진행됩니다.

참조

- System.IFormattable
- System.IFormatProvider
- System.ICustomFormatter

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

표준 숫자 형식 문자열

2025. 06. 30.

표준 숫자 형식 문자열은 일반적인 숫자 형식의 서식을 지정하는 데 사용됩니다. 표준 숫자 형식 문자열은 다음과 같은 형식 `[format specifier][precision specifier]`을 사용합니다.

- *형식 지정자는* 숫자 형식의 형식(예: 통화 또는 백분율)을 지정하는 단일 알파벳 문자입니다. 공백을 포함하여 둘 이상의 알파벳 문자를 포함하는 숫자 서식 문자열은 사용자 지정 숫자 서식 문자열로 해석됩니다. 자세한 내용은 사용자 지정 숫자 형식 문자열을 참조하세요.
- *전체 자릿수 지정자는* 결과 문자열의 숫자 수에 영향을 주는 선택적 정수입니다. .NET 7 이상 버전에서 최대 정밀도 값은 999,999,999입니다. .NET 6에서 최대 전체 자릿수 값은 Int32.MaxValue. 이전 .NET 버전에서 전체 자릿수는 0에서 99까지입니다. 전체 자릿수 지정자는 숫자의 문자열 표현에서 숫자 수를 제어합니다. 숫자 자체를 반올림하지 않습니다. 반올림 작업을 수행하려면 , Math.Ceiling또는 Math.Floor 메서드를 Math.Round사용합니다.

  전체 자릿수 지정자가 결과 문자열의 소수 자릿수를 제어하는 경우 결과 문자열은 무한정 정확한 결과에 가장 가까운 표시 가능한 결과로 반올림되는 숫자를 반영합니다. 두 개의 동일하게 나타낼 수 있는 결과가 있는 경우:
  - **.NET Framework 및 .NET Core에서 .NET Core 2.0까지** 런타임은 유효 자릿수가 가장 낮은 결과(즉, 사용 MidpointRounding.AwayFromZero)를 선택합니다.
  - **.NET Core 2.1 이상에서** 런타임은 유효 자릿수가 가장 낮은 결과(즉, 사용 MidpointRounding.ToEven)를 선택합니다.

  > **참고**
  >
  > 전체 자릿수 지정자는 결과 문자열의 자릿수를 결정합니다. 선행 또는 후행 공백으로 결과 문자열을 패딩하려면 **복합 서식** 지정 기능을 사용하고 서식 항목에서 *너비 구성 요소를* 정의합니다.

표준 숫자 형식 문자열은 다음에서 지원됩니다.

- 모든 숫자 형식의 `ToString` 메서드에 대한 일부 오버로드입니다. 예를 들어 숫자 형식 문자열을 및 Int32.ToString(String) 메서드에 Int32.ToString(String, IFormatProvider) 제공할 수 있습니다.
- `TryFormat` 예를 들어 Int32.TryFormat(Span<Char>, Int32, ReadOnlySpan<Char>, IFormatProvider) 모든 숫자 형식의 메서드입니다 Single.TryFormat(Span<Char>, Int32, ReadOnlySpan<Char>, IFormatProvider).
- 클래스, 메서드 및 메서드의 `Write` `WriteLine`Console 일부 StreamWriter 및 String.Format 메서드에서 사용되는 .NET StringBuilder.AppendFormat입니다. 복합 형식 기능을 사용하면 여러 데이터 항목의 문자열 표현을 단일 문자열에 포함하고, 필드 너비를 지정하고, 필드에 숫자를 맞출 수 있습니다. 자세한 내용은 복합 서식을 참조하세요.
- 복합 형식 문자열과 비교할 때 간소화된 구문을 제공하는 C# 및 Visual Basic의 보간된 문자열입니다.

> **팁**
>
> 숫자 또는 날짜 및 시간 값에 서식 문자열을 적용하고 결과 문자열을 표시할 수 있는 .NET Core Windows Forms 애플리케이션인 서식 **유틸리티**를 다운로드할 수 있습니다. 소스 코드는 **C#** 및 **Visual Basic**에 사용할 수 있습니다.

표준 형식 지정자

다음 표에서는 표준 숫자 형식 지정자에 대해 설명하고 각 형식 지정자가 생성한 샘플 출력을 표시합니다. 표준 숫자 형식 문자열을 사용하는 방법에 대한 자세한 내용은 Notes 섹션을 참조하고, 코드 예제 섹션을 참조하세요.

> 특정 문화권에 대한 형식이 지정된 문자열의 결과는 다음 예제와 다를 수 있습니다. 운영 체제 설정, 사용자 설정, 환경 변수 및 사용 중인 .NET 버전은 모두 형식에 영향을 줄 수 있습니다. 예를 들어 .NET 5부터 .NET은 여러 플랫폼에서 문화 형식을 통합하려고 합니다. 자세한 내용은 .NET 세계화 및 ICU를 참조하세요.

테이블 확장

서식 지정자	이름	설명	예시
"B" 또	바이	결과: 이진 문자열입니다.	42 ("나") -> 101010

서식 지정자	이름	설명	예시
"b"	너리는	지원: 정수 형식만(.NET 8 이상). 전체 자릿수 지정자: 결과 문자열의 숫자 수입니다. 추가 정보: 이진("B") 형식 지정자입니다.	255 ("B16") -> 0000000011111111
"C" 또는 "c"	통화	결과: 통화 값입니다. 지원: 모든 숫자 형식. 전체 자릿수 지정자: 소수 자릿수입니다. 기본 전체 자릿수 지정자: 에 의해 NumberFormatInfo.CurrencyDecimalDigits 정의됩니다. 추가 정보: 통화("C") 형식 지정자입니다.	123.456("C", en-US) -> 123.46 엔$123.456(\text{"C"}, fr-FR) -> 123,46$ 유로$123.456(\text{"C"}, ja-JP) -> 123$ 엔 $-123.456(\text{"C3"}, en$ 123.456엔) -123.456("C3", fr-FR) -> -123,456 유로 -123.456("C3", ja-JP) -> -123.456
"D" 또는 "d"	십진수	결과: 선택적 음수 기호가 있는 정수 숫자입니다. 지원: 정수 계열 형식에만 해당합니다. 전체 자릿수 지정자: 최소 자릿수입니다. 기본 전체 자릿수 지정자: 필요한 최소 자릿수입니다. 추가 정보: Decimal("D") 형식 지정자입니다.	1234("D") -> 1234년 -1234("D6") -> -001234
"E" 또는 "e"	지수(과학)	결과: 지수 표기법입니다. 지원: 모든 숫자 형식. 전체 자릿수 지정자: 소수 자릿수입니다. 기본 전체 자릿수 지정자: 6. 추가 정보: 지수("E") 형식 지정자입니다.	1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756("e", fr-FR) -> 1,052033E+003 -1052.0329112756("e2", en-US) -> -1.05E+003 -1052.0329112756("E2", fr-FR) -> -1,05E+003
"F" 또는 "f"	고정 소수점	결과: 선택적 음수 기호가 있는 정수 및 소수 자릿수입니다. 지원: 모든 숫자 형식. 전체 자릿수 지정자: 소수 자릿수입니다. 기본 전체 자릿수 지정자: 에 의해 NumberFormatInfo.NumberDecimalDigits 정의됩니다. 추가 정보: Fixed-Point("F") 형식 지정자입니다.	1234.567 ("에프", en-US) -> 1234.57 년 1234.567("F", de-DE) -> 1234,57 1234("F1", en-US) -> 1234.0 1234("F1", de-DE) -> 1234,0 -1234.56("F4", en-US) -> -1234.5600 -1234.56("F4", de-DE) -> -1234,5600
"G" 또는 "g"	일반	결과: 고정 소수점 또는 과학적 표기법이 더 압축됩니다. 지원: 모든 숫자 형식. 전체 자릿수 지정자: 유효 자릿수입니다. 기본 전체 자릿수 지정자: 숫자 형식에 따	-123.456("G", en-US) -> -123.456 -123.456("G", sv-SE) -> -123,456 123.4546("G4", en-US) -> 123.5

서식 지정자	이름	설명	예시
		라 다릅니다. 추가 정보: 일반("G") 형식 지정자입니다.	123.4546("G4", sv-SE) -> 123,5 -1.234567890e-25("G", en-US) -> -1.23456789E-25 -1.234567890e-25("G", sv-SE) -> -1,23456789E-25
"N" 또는 "n"	숫자	결과: 정수 및 10진수 숫자, 그룹 구분 기호 및 선택적 음수 기호가 있는 소수 구분 기호입니다. 지원: 모든 숫자 형식. 전체 자릿수 지정자: 원하는 소수 자릿수입니다. 기본 전체 자릿수 지정자: 에 의해 NumberFormatInfo.NumberDecimalDigits 정의됩니다. 추가 정보: 숫자("N") 형식 지정자입니다.	1234.567 ("N", en-US) -> 1,234.57 1234.567("N", ru-RU) -> 1 234,57 1234년 ("N1", en-US) -> 1,234.0 1234("N1", ru-RU) -> 1 234,0 -1234.56("N3", en-US) -> -1,234.560 -1234.56("N3", ru-RU) -> -1 234,560
"P" 또는 "p"	퍼센트	결과: 숫자를 100으로 곱하고 백분율 기호와 함께 표시합니다. 지원: 모든 숫자 형식. 전체 자릿수 지정자: 원하는 소수 자릿수입니다. 기본 전체 자릿수 지정자: 에 의해 NumberFormatInfo.PercentDecimalDigits 정의됩니다. 추가 정보: 백분율("P") 형식 지정자입니다.	1("P", en-US) -> 100.00 % 1("P", fr-FR) -> 100,000 % -0.39678("P1", en-US) -> -39.7% -0.39678("P1", fr-FR) -> -39,7%
"R" 또는 "r"	왕복	결과: 동일한 숫자로 왕복할 수 있는 문자열입니다. 지원되는 형식: Single, Double 및 BigInteger 참고: 형식에 BigInteger 만 권장합니다. 형식의 경우 Double "G17"을 사용하고 형식에는 Single "G9"를 사용합니다. 전체 자릿수 지정자: 무시됩니다. 추가 정보: 왕복("R") 형식 지정자입니다.	123456789.12345678("R") -> 123456789.12345678 -1234567890.12345678("R") -> -1234567890.1234567
"X" 또는 "x"	16진수	결과: 16진수 문자열입니다. 지원: 정수 계열 형식에만 해당합니다. 전체 자릿수 지정자: 결과 문자열의 숫자 수입니다. 추가 정보: 16진수("X") 형식 지정자입니다.	255("X") -> FF (FF) -1("x") -> FF 255("x4") -> 00FF -1("X4") -> 00FF
다른 모	알 수 없	결과: 런타임에 throw FormatException 합니다.	

서식 지정자	이름	설명	예시
든 단일 문자	는 지정자		

표준 숫자 형식 문자열 사용

> ⓘ **참고**
>
> 이 문서의 C# 예제는 Try.NET 인라인 코드 실행기 및 플레이그라운드에서 실행됩니다. **실행** 단추를 선택하여 대화형 창에서 예제를 실행합니다. 코드를 실행하면 코드를 수정하고 **실행을** 다시 선택하여 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나 컴파일에 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

표준 숫자 서식 문자열을 사용하여 다음 방법 중 하나로 숫자 값의 서식을 정의할 수 있습니다.

- 매개 변수가 있는 메서드 또는 메서드 `TryFormat` 의 오버로드에 `ToString` 전달할 `format` 수 있습니다. 다음 예제에서는 현재 문화권에서 숫자 값의 형식을 통화 문자열로 지정합니다(이 경우 en-US 문화권).

  C#

  ```
  decimal value = 123.456m;
  Console.WriteLine(value.ToString("C2"));
  // Displays $123.46
  ```

- 메서드와 같은 메서드와 함께 사용되는 형식 항목에서 인수로 `formatString`String.FormatConsole.WriteLine제공될 수 있습니다.StringBuilder.AppendFormat 자세한 내용은 복합 서식을 참조하세요. 다음 예제에서는 형식 항목을 사용하여 문자열에 통화 값을 삽입합니다.

  C#

  ```
  decimal value = 123.456m;
  Console.WriteLine($"Your account balance is {value:C2}.");
  // Displays "Your account balance is $123.46."
  ```

  필요에 따라 인수를 `alignment` 제공하여 숫자 필드의 너비와 해당 값이 오른쪽 또는 왼쪽 맞춤인지 여부를 지정할 수 있습니다. 다음 예제에서는 28자 필드에 통화 값을 왼쪽 맞춤하고 14자 필드에 통화 값을 오른쪽 맞춤합니다.

  C#

  ```
  decimal[] amounts = { 16305.32m, 18794.16m };
  Console.WriteLine("   Beginning Balance           Ending Balance");
  Console.WriteLine("   {0,-28:C2}{1,14:C2}", amounts[0], amounts[1]);
  // Displays:
  //        Beginning Balance           Ending Balance
  //        $16,305.32                      $18,794.16
  ```

- 보간된 문자열의 `formatString` 보간된 식 항목에서 인수로 제공할 수 있습니다. 자세한 내용은 C# 참조의 문자열 보간 문서 또는 Visual Basic 참조의 보간된 문자열 문서를 참조하세요.

다음 섹션에서는 각 표준 숫자 형식 문자열에 대한 자세한 정보를 제공합니다.

이진 형식 지정자(B)

이진("B") 형식 지정자는 숫자를 이진 숫자 문자열로 변환합니다. 이 형식은 정수 계열 형식에 대해서만 지원되며 .NET 8 이상에서만 지원됩니다.

전체 자릿수 지정자는 결과 문자열에서 원하는 최소 자릿수를 나타냅니다. 필요한 경우 전체 자릿수 지정자가 지정한 자릿수를 생성하기 위해 숫자가 왼쪽에 0으로 채워집니다.

양수 값의 경우 BigInteger항상 음수 값과 구분하기 위한 선행 0이 있습니다. 이렇게 하면 구문 분석 시 출력이 원래 값으로 왕복됩니다. 예를 들어 형식 지정자를 `"B2"` 사용하여 변환된 숫자는 `3` .입니다`"011"`. 이진 번호 `"11"` 는 형식으로 인해 정확히 `2` 비트가 있는 숫자로 해석되므로 음수 값을 `-1`나타내기 때문 `"B2"` 입니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받지 않습니다.

통화 형식 지정자(C)

"C"(또는 통화) 형식 지정자는 숫자를 통화 금액을 나타내는 문자열로 변환합니다. 전체 자릿수 지정자는 결과 문자열에서 원하는 소수 자릿수를 나타냅니다. 전체 자릿수 지정자를 생략하면 기본 전체 자릿수가 속성에 NumberFormatInfo.CurrencyDecimalDigits 의해 정의됩니다.

서식을 지정할 값이 지정하거나 기본값인 소수 자릿수보다 많은 경우 소수 자릿수 값은 결과 문자열에서 반올림됩니다. 지정한 소수 자릿수의 오른쪽에 있는 값이 5 이상이면 결과 문자열의 마지막 숫자가 0에서 둥글게 반올림됩니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 반환된 NumberFormatInfo 문자열의 서식을 제어하는 속성을 나열합니다.

테이블 확장

NumberFormatInfo 속성	설명
CurrencyPositivePattern	양수 값에 대한 통화 기호의 배치를 정의합니다.
CurrencyNegativePattern	음수 값에 대한 통화 기호의 배치를 정의하고 음수 기호가 괄호 또는 NegativeSign 속성으로 표시되는지 여부를 지정합니다.
NegativeSign	괄호가 사용되지 않음을 나타내는 경우 CurrencyNegativePattern 사용되는 음수 기호를 정의합니다.
CurrencySymbol	통화 기호를 정의합니다.
CurrencyDecimalDigits	통화 값의 기본 소수 자릿수를 정의합니다. 전체 자릿수 지정자를 사용하여 이 값을 재정의할 수 있습니다.
CurrencyDecimalSeparator	정수 자릿수와 소수 자릿수를 구분하는 문자열을 정의합니다.
CurrencyGroupSeparator	정수 그룹을 구분하는 문자열을 정의합니다.
CurrencyGroupSizes	그룹에 표시되는 정수 자릿수를 정의합니다.

다음 예제에서는 통화 형식 지정자를 사용하여 Double 값의 서식을 지정합니다.

C#

```
double value = 12345.6789;
Console.WriteLine(value.ToString("C", CultureInfo.CurrentCulture));

Console.WriteLine(value.ToString("C3", CultureInfo.CurrentCulture));

Console.WriteLine(value.ToString("C3",
                  CultureInfo.CreateSpecificCulture("da-DK")));
// The example displays the following output on a system whose
// current culture is English (United States):
//       $12,345.68
//       $12,345.679
//       12.345,679 kr
```

10진수 형식 지정자(D)

"D"(또는 10진수) 형식 지정자는 숫자를 10진수 문자열(0-9)로 변환하고, 숫자가 음수이면 빼기 기호로 접두사를 지정합니다. 이 형식은 정수 형식에 대해서만 지원됩니다.

전체 자릿수 지정자는 결과 문자열에서 원하는 최소 자릿수를 나타냅니다. 필요한 경우 전체 자릿수 지정자가 지정한 자릿수를 생성하기 위해 숫자가 왼쪽에 0으로 채워집니다. 전체 자릿수 지정자가 지정되지 않은 경우 기본값은 선행 0이 없는 정수에 필요한 최소값입니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표와 같이 단일 속성은 결과 문자열의 서식에 영향을 줍니다.

테이블 확장

NumberFormatInfo 속성	설명
NegativeSign	숫자가 음수임을 나타내는 문자열을 정의합니다.

다음은 10진수 서식 지정자를 사용하여 Int32 값의 서식을 지정하는 예제입니다.

C#

```
int value;

value = 12345;
Console.WriteLine(value.ToString("D"));
// Displays 12345
Console.WriteLine(value.ToString("D8"));
// Displays 00012345

value = -12345;
Console.WriteLine(value.ToString("D"));
// Displays -12345
Console.WriteLine(value.ToString("D8"));
// Displays -00012345
```

지수 형식 지정자(E)

지수("E") 형식 지정자는 숫자를 "-d.ddd… E+ddd" 또는 "-d.ddd… e+ddd"입니다. 여기서 각 "d"는 숫자(0-9)를 나타냅니다. 숫자가 음수이면 문자열이 빼기 기호로 시작합니다. 정확히 한 자리는 항상 소수점 앞에 섰습니다.

전체 자릿수 지정자는 소수점 뒤의 원하는 자릿수를 나타냅니다. 전체 자릿수 지정자를 생략하면 소수점 이후의 기본값인 6자리가 사용됩니다.

형식 지정자의 대/소문자는 지수 접두사를 "E" 또는 "e"로 접두사로 지정할지 여부를 나타냅니다. 지수는 항상 더하기 또는 빼기 기호와 최소 3자리 숫자로 구성됩니다. 지수는 필요한 경우 이 최소값을 충족하기 위해 0으로 채워집니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 반환된 NumberFormatInfo 문자열의 서식을 제어하는 속성을 나열합니다.

테이블 확장

NumberFormatInfo 속성	설명
NegativeSign	계수와 지수 모두에 대해 숫자가 음수임을 나타내는 문자열을 정의합니다.
NumberDecimalSeparator	계수의 정수와 소수 자릿수를 구분하는 문자열을 정의합니다.
PositiveSign	지수가 양수임을 나타내는 문자열을 정의합니다.

다음 예제에서는 지수 형식 지정자를 사용하여 Double 값의 서식을 지정합니다.

C#

```
double value = 12345.6789;
Console.WriteLine(value.ToString("E", CultureInfo.InvariantCulture));
// Displays 1.234568E+004

Console.WriteLine(value.ToString("E10", CultureInfo.InvariantCulture));
// Displays 1.2345678900E+004

Console.WriteLine(value.ToString("e4", CultureInfo.InvariantCulture));
// Displays 1.2346e+004

Console.WriteLine(value.ToString("E",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 1,234568E+004
```

고정 소수점 형식 지정자(F)

고정 소수점("F") 형식 지정자는 숫자를 "-ddd.ddd…" 형식의 문자열로 변환합니다. 여기서 각 "d"는 숫자(0-9)를 나타냅니다. 숫자가 음수이면 문자열이 빼기 기호로 시작합니다.

정밀도 지정자는 원하는 소수 자릿수를 나타냅니다. 전체 자릿수 지정자를 생략하면 현재 NumberFormatInfo.NumberDecimalDigits 속성은 숫자 정밀도를 제공합니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 결과 문자열의 서식을 NumberFormatInfo 제어하는 개체의 속성을 나열합니다.

테이블 확장

NumberFormatInfo 속성	설명
NegativeSign	숫자가 음수임을 나타내는 문자열을 정의합니다.
NumberDecimalSeparator	정수 자릿수와 소수 자릿수를 구분하는 문자열을 정의합니다.
NumberDecimalDigits	기본 소수 자릿수를 정의합니다. 전체 자릿수 지정자를 사용하여 이 값을 재정의할 수 있습니다.

다음 예제에서는 고정 소수점 형식 지정자를 사용하여 Double a Int32 및 값의 서식을 지정합니다.

C#

```
int integerNumber;
integerNumber = 17843;
Console.WriteLine(integerNumber.ToString("F",
                  CultureInfo.InvariantCulture));
// Displays 17843.00

integerNumber = -29541;
Console.WriteLine(integerNumber.ToString("F3",
                  CultureInfo.InvariantCulture));
// Displays -29541.000

double doubleNumber;
doubleNumber = 18934.1879;
Console.WriteLine(doubleNumber.ToString("F", CultureInfo.InvariantCulture));
// Displays 18934.19

Console.WriteLine(doubleNumber.ToString("F0", CultureInfo.InvariantCulture));
// Displays 18934

doubleNumber = -1898300.1987;
Console.WriteLine(doubleNumber.ToString("F1", CultureInfo.InvariantCulture));
// Displays -1898300.2

Console.WriteLine(doubleNumber.ToString("F3",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays -1898300,199
```

일반 형식 지정자(G)

일반("G") 형식 지정자는 숫자의 형식과 정밀도 지정자가 있는지 여부에 따라 숫자를 고정 소수점 또는 과학적 표기법의 압축으로 변환합니다. 전체 자릿수 지정자는 결과 문자열에 나타날 수 있는 최대 유효 자릿수를 정의합니다. 전체 자릿수 지정자를 생략하거나 0이면 다음 표에 표시된 대로 숫자의 형식에 따라 기본 전체 자릿수가 결정됩니다.

테이블 확장

숫자 형식	기본 전체 자릿수
Byte 또는 SByte	3자리 숫자
Int16 또는 UInt16	5자리 숫자
Int32 또는 UInt32	10자리 숫자
Int64	19자리 숫자
UInt64	20자리 숫자
BigInteger	무제한("R"과 동일)
Half	숫자를 나타내는 가장 작은 라운드트립 가능 자릿수
Single	숫자를 나타내는 가장 작은 라운드트립 가능 자릿수입니다(.NET Framework에서는 G7이 기본값임).
Double	숫자를 나타내는 가장 작은 라운드트립 가능 자릿수입니다(.NET Framework에서는 G15가 기본값임).
Decimal	숫자를 나타내는 가장 작은 라운드트립 가능 자릿수

고정 소수점 표기법은 과학적 표기법으로 숫자를 표현한 결과 지수가 -5 보다 크고 정밀도 지정자보다 작으면 사용됩니다. 그렇지 않으면 과학적 표기법이 사용됩니다. 필요한 경우 결과에는 소수점이 포함되고 소수점 뒤의 후행 0은 생략됩니다. 전체 자릿수 지정자가 있고 결과의 유효 자릿수가 지정된 전체 자릿수를 초과하면 반올림을 통해 초과 후행 자릿수가 제거됩니다.

그러나 숫자가 a Decimal 이고 전체 자릿수 지정자를 생략하면 고정 소수점 표기법이 항상 사용되며 후행 0이 유지됩니다.

과학적 표기법을 사용하는 경우 형식 지정자가 "G"이면 결과의 지수 앞에 "E", 형식 지정자가 "g"인 경우 "e"가 접두사로 지정됩니다. 지수에는 최소 두 자리 숫자가 포함됩니다. 지수에 최소 3자리 숫자를 포함하는 지수 형식 지정자에 의해 생성되는 과학적 표기법의 형식과 다릅니다.

값과 함께 Double 사용할 경우 "G17" 형식 지정자는 원래 Double 값이 성공적으로 왕복되도록 합니다. Double 이는 IEEE 754-2008 규격 배정밀도(`binary64`) 부동 소수점 숫자로 최대 17자리의 정밀도를 제공하기 때문입니다. .NET Framework에서는 "R" 형식 지정자 대신 사용하는 것이 좋습니다. 경우에 따라 "R"이 왕복 배정밀도 부동 소수점 값을 성공적으로 라운드트립하지 못하기 때문에 사용하는 것이 좋습니다.

값과 함께 Single 사용할 경우 "G9" 형식 지정자는 원래 Single 값이 성공적으로 왕복되도록 합니다. 이는 Single IEEE 754-2008 규격 단정밀도(`binary32`) 부동 소수점 숫자로 최대 9자리의 정밀도를 제공하기 때문입니다. 성능상의 이유로 "R" 형식 지정자 대신 사용하는 것이 좋습니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 결과 문자열의 서식을 제어하는 속성을 나열 NumberFormatInfo 합니다.

테이블 확장

NumberFormatInfo 속성	설명
NegativeSign	숫자가 음수임을 나타내는 문자열을 정의합니다.
NumberDecimalSeparator	정수 자릿수와 소수 자릿수를 구분하는 문자열을 정의합니다.
PositiveSign	지수가 양수임을 나타내는 문자열을 정의합니다.

다음 예제에서는 일반 형식 지정자를 사용하여 다양한 부동 소수점 값의 서식을 지정합니다.

C#

```
double number;

number = 12345.6789;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays  12345.6789
Console.WriteLine(number.ToString("G",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 12345,6789

Console.WriteLine(number.ToString("G7", CultureInfo.InvariantCulture));
// Displays 12345.68

number = .0000023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 2.3E-06
Console.WriteLine(number.ToString("G",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 2,3E-06

number = .0023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 0.0023

number = 1234;
Console.WriteLine(number.ToString("G2", CultureInfo.InvariantCulture));
// Displays 1.2E+03

number = Math.PI;
Console.WriteLine(number.ToString("G5", CultureInfo.InvariantCulture));
// Displays 3.1416
```

숫자 형식 지정자(N)

숫자("N") 형식 지정자는 숫자를 "-d,ddd,ddd.ddd…"형식의 문자열로 변환합니다. 여기서 "-"는 필요한 경우 음수 기호를 나타내고, "d"는 숫자(0-9)를 나타내고, ""는 그룹 구분 기호를 나타내고, "."는 소수점 기호를 나타냅니다. 전체 자릿수 지정자는 소수점 뒤의 원하는 자릿수를 나타냅니다. 전체 자릿수 지정자를 생략하면 소수 자릿수가 현재 NumberFormatInfo.NumberDecimalDigits 속성에 의해 정의됩니다.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 결과 문자열의 서식을 제어하는 속성을 나열 NumberFormatInfo 합니다.

테이블 확장

NumberFormatInfo 속성	설명
NegativeSign	숫자가 음수임을 나타내는 문자열을 정의합니다.
NumberNegativePattern	음수 값의 형식을 정의하고 음수 기호가 괄호로 표시되는지 또는 NegativeSign 속성으로 표시되는지 여부를 지정합니다.
NumberGroupSizes	그룹 구분 기호 사이에 나타나는 정수 자릿수를 정의합니다.
NumberGroupSeparator	정수 그룹을 구분하는 문자열을 정의합니다.
NumberDecimalSeparator	정수 자릿수와 소수 자릿수를 구분하는 문자열을 정의합니다.
NumberDecimalDigits	기본 소수 자릿수를 정의합니다. 정밀도 지정자를 사용하여 이 값을 재정의할 수 있습니다.

다음 예제에서는 숫자 서식 지정자를 사용하여 다양한 부동 소수점 값의 서식을 지정합니다.

C#

```
double dblValue = -12445.6789;
Console.WriteLine(dblValue.ToString("N", CultureInfo.InvariantCulture));
// Displays -12,445.68
Console.WriteLine(dblValue.ToString("N1",
                  CultureInfo.CreateSpecificCulture("sv-SE")));
// Displays -12 445,7

int intValue = 123456789;
Console.WriteLine(intValue.ToString("N1", CultureInfo.InvariantCulture));
// Displays 123,456,789.0
```

백분율 형식 지정자(P)

백분율("P") 형식 지정자는 숫자를 100으로 곱하고 백분율을 나타내는 문자열로 변환합니다. 정밀도 지정자는 원하는 소수 자릿수를 나타냅니다. 전체 자릿수 지정자를 생략하면 현재 PercentDecimalDigits 속성에서 제공하는 기본 숫자 전체 자릿수가 사용됩니다.

다음 표에서는 반환된 NumberFormatInfo 문자열의 서식을 제어하는 속성을 나열합니다.

테이블 확장

NumberFormatInfo 속성	설명
PercentPositivePattern	양수 값에 대한 백분율 기호의 배치를 정의합니다.
PercentNegativePattern	음수 값에 대한 백분율 기호 및 음수 기호의 배치를 정의합니다.
NegativeSign	숫자가 음수임을 나타내는 문자열을 정의합니다.
PercentSymbol	백분율 기호를 정의합니다.
PercentDecimalDigits	백분율 값의 기본 소수 자릿수를 정의합니다. 전체 자릿수 지정자를 사용하여 이 값을 재정의할 수 있습니다.
PercentDecimalSeparator	정수 자릿수와 소수 자릿수를 구분하는 문자열을 정의합니다.
PercentGroupSeparator	정수 그룹을 구분하는 문자열을 정의합니다.
PercentGroupSizes	그룹에 표시되는 정수 자릿수를 정의합니다.

다음 예제에서는 백분율 서식 지정자를 사용하여 부동 소수점 값의 서식을 지정합니다.

C#

```
double number = .2468013;
Console.WriteLine(number.ToString("P", CultureInfo.InvariantCulture));
// Displays 24.68 %
Console.WriteLine(number.ToString("P",
                  CultureInfo.CreateSpecificCulture("hr-HR")));
// Displays 24,68%
Console.WriteLine(number.ToString("P1", CultureInfo.InvariantCulture));
// Displays 24.7 %
```

왕복 형식 지정자(R)

라운드트립("R") 형식 지정자는 문자열로 변환된 숫자 값이 동일한 숫자 값으로 다시 구문 분석되도록 합니다. 이 형식은 , Half, Single및 Double 형식에 BigInteger대해서만 지원됩니다.

.NET Framework 및 3.0 이전의 .NET Core 버전에서 "R" 형식 지정자가 경우에 따라 왕복 값을 성공적으로 반환 Double 하지 못합니다. 값과 Double 값 모두 Single 에 대해 "R" 형식 지정자는 상대적으로 성능이 저하됩니다. 대신 값에 "G17" 형식 지정자와 Double"G9" 형식 지정자를 사용하여 왕복 Single 값을 성공적으로 수행하는 것이 좋습니다.

이 지정자를 BigInteger 사용하여 값의 서식을 지정하면 해당 문자열 표현에는 값의 모든 유효 자릿수가 BigInteger 포함됩니다.

전체 자릿수 지정자를 포함할 수 있지만 무시됩니다. 이 지정자를 사용하는 경우 왕복이 정밀도보다 우선적으로 적용됩니다. 결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 결과 문자열의 서식을 제어하는 속성을 나열 NumberFormatInfo 합니다.

테이블 확장

NumberFormatInfo 속성	설명
NegativeSign	숫자가 음수임을 나타내는 문자열을 정의합니다.
NumberDecimalSeparator	정수 자릿수와 소수 자릿수를 구분하는 문자열을 정의합니다.
PositiveSign	지수가 양수임을 나타내는 문자열을 정의합니다.

다음은 라운드트립 형식 지정자를 사용하여 BigInteger 값의 서식을 지정하는 예제입니다.

C#

```
using System;
using System.Numerics;

public class Example
{
   public static void Main()
   {
      var value = BigInteger.Pow(Int64.MaxValue, 2);
      Console.WriteLine(value.ToString("R"));
   }
}
// The example displays the following output:
//      85070591730234615847396907784232501249
```

중요

경우에 따라 Double "R" 표준 숫자 형식 문자열로 서식이 지정된 값은 64비트 시스템에서 또는 스위치를 사용하여 `/platform:x64` `/platform:anycpu` 컴파일되고 실행되는 경우 왕복에 성공하지 않습니다. 자세한 내용은 다음 단락을 참조하세요.

"R" 표준 숫자 형식 문자열로 서식이 지정된 값이 Double 라운드트립되지 않는 문제를 해결하려면 스위치를 `/platform:x64` 사용하여 `/platform:anycpu` 컴파일하고 64비트 시스템에서 실행하는 경우 "G17" 표준 숫자 형식 문자열을 사용하여 값의 서식을 지정할 Double 수 있습니다. 다음 예제에서는 왕복에 성공하지 않는 값과 함께 Double "R" 형식 문자열을 사용하고 "G17" 형식 문자열을 사용하여 원래 값을 성공적으로 왕복합니다.

C#

```
Console.WriteLine("Attempting to round-trip a Double with 'R':");
double initialValue = 0.6822871999174;
string valueString = initialValue.ToString("R",
                                           CultureInfo.InvariantCulture);
double roundTripped = double.Parse(valueString,
                                   CultureInfo.InvariantCulture);
Console.WriteLine($"{initialValue:R} = {roundTripped:R}: {initialValue.Equals(roundTripped)}\n");

Console.WriteLine("Attempting to round-trip a Double with 'G17':");
string valueString17 = initialValue.ToString("G17",
                                             CultureInfo.InvariantCulture);
double roundTripped17 = double.Parse(valueString17,
                                     CultureInfo.InvariantCulture);
Console.WriteLine($"{initialValue:R} = {roundTripped17:R}: {initialValue.Equals(roundTripped17)}\n");
// If compiled to an application that targets anycpu or x64 and run on an x64 system,
// the example displays the following output:
//       Attempting to round-trip a Double with 'R':
//       .NET Framework:
//       0.6822871999174 = 0.68228719991740006: False
```

```
//       .NET:
//       0.6822871999174 = 0.6822871999174: True
//
//       Attempting to round-trip a Double with 'G17':
//       0.6822871999174 = 0.6822871999174: True
```

16진수 형식 지정자(X)

16진수("X") 형식 지정자는 숫자를 16진수 문자열로 변환합니다. 형식 지정자의 대/소문자는 9보다 큰 16진수에 대문자 또는 소문자를 사용할지 여부를 나타냅니다. 예를 들어 "X"를 사용하여 "ABCDEF"를 생성하고 "x"를 사용하여 "abcdef"를 생성합니다. 이 형식은 정수 형식에 대해서만 지원됩니다.

전체 자릿수 지정자는 결과 문자열에서 원하는 최소 자릿수를 나타냅니다. 필요한 경우 전체 자릿수 지정자가 지정한 자릿수를 생성하기 위해 숫자가 왼쪽에 0으로 채워집니다.

양수 값의 경우 BigInteger항상 음수 값과 구분하기 위한 선행 0이 있습니다. 이렇게 하면 구문 분석 시 출력이 원래 값으로 왕복됩니다. 예를 들어 형식 지정자를 `"X1"` 사용하여 변환된 숫자는 `F` 16진수가 음수 `F` 값을 `-1`나타내기 때문입니다`0F`.

결과 문자열은 현재 NumberFormatInfo 개체의 서식 지정 정보의 영향을 받지 않습니다.

다음은 16진수 서식 Int32 지정자를 사용하여 값의 서식을 지정하는 예제입니다.

C#

```
int value;

value = 0x2045e;
Console.WriteLine(value.ToString("x"));
// Displays 2045e
Console.WriteLine(value.ToString("X"));
// Displays 2045E
Console.WriteLine(value.ToString("X8"));
// Displays 0002045E

value = 123456789;
Console.WriteLine(value.ToString("X"));
// Displays 75BCD15
Console.WriteLine(value.ToString("X2"));
// Displays 75BCD15
```

비고

이 섹션에는 표준 숫자 형식 문자열 사용에 대한 추가 정보가 포함되어 있습니다.

제어판 설정

제어판의 **국가별 및 언어 옵션** 항목의 설정은 서식 지정 작업으로 생성된 결과 문자열에 영향을 미칩니다. 이러한 설정은 서식을 제어하는 데 사용되는 값을 제공하는 현재 문화권과 연결된 개체를 초기화하는 NumberFormatInfo 데 사용됩니다. 다른 설정을 사용하는 컴퓨터는 다른 결과 문자열을 생성합니다.

또한 생성자를 사용하여 현재 시스템 문화권과 동일한 문화권을 나타내는 새 CultureInfo(String) 개체를 인스턴스화하는 경우 CultureInfo 제어판의 **국가 및 언어 옵션** 항목에 의해 설정된 모든 사용자 지정이 새 CultureInfo 개체에 적용됩니다. 생성자를 사용하여 CultureInfo(String, Boolean) 시스템의 사용자 지정을 CultureInfo 반영하지 않는 개체를 만들 수 있습니다.

NumberFormatInfo 속성

서식 지정은 현재 NumberFormatInfo 문화권에 의해 암시적으로 제공되거나 서식을 호출하는 메서드의 매개 변수에 의해 명시적으로 제공되는 현재 개체의 속성에 의해 IFormatProvider 영향을 받습니다. 해당 매개 변수에 NumberFormatInfo 대한 개체 또는 CultureInfo 개체를 지정합니다.

> ⓘ **참고**
>
> 숫자 값 서식 지정에 사용되는 패턴 또는 문자열을 사용자 지정하는 방법에 대한 자세한 내용은 클래스 항목을 참조 NumberFormatInfo 하세요.

정수 및 부동 소수점 숫자 형식

표준 숫자 형식 지정자에 대한 일부 설명은 정수 또는 부동 소수점 숫자 형식을 참조합니다. 정수 계열 숫자 형식은 Byte, SByte,Int16, Int32Int64, UInt16, UInt32UInt64및 BigInteger. 부동 소수점 숫자 형식은 Decimal, Half, Single및 Double.

부동 소수점 무한대 및 NaN

형식 문자열에 관계없이 , 또는 부동 소수점 형식의 Half값이 무한대, 음수 무한대이거나 숫자(NaN)가 아닌 경우 서식이 지정된 문자열은 현재 적용 가능한 Single 개체에 의해 지정된 각 DoublePositiveInfinitySymbolNegativeInfinitySymbol 또는 속성의 값입니다.NaNSymbolNumberFormatInfo

코드 예제

다음 예제에서는 en-US 문화권 및 모든 표준 숫자 형식 지정자를 사용하여 정수 및 부동 소수점 숫자 값의 서식을 지정합니다. 이 예제에서는 두 개의 특정 숫자 형식(Double및Int32)을 사용하지만 다른 숫자 기본 형식(Byte, ,, SByte,Int16Int32Int64UInt16UInt32UInt64BigIntegerDecimalHalf, 및 Single)에 대해 유사한 결과를 생성합니다.

C#

```
// Display string representations of numbers for en-us culture
CultureInfo ci = new CultureInfo("en-us");

// Output floating point values
double floating = 10761.937554;
Console.WriteLine($"C: {floating.ToString("C", ci)}");           // Displays "C: $10,761.94"
Console.WriteLine($"E: {floating.ToString("E03", ci)}");         // Displays "E: 1.076E+004"
Console.WriteLine($"F: {floating.ToString("F04", ci)}");         // Displays "F: 10761.9376"
Console.WriteLine($"G: {floating.ToString("G", ci)}");           // Displays "G: 10761.937554"
Console.WriteLine($"N: {floating.ToString("N03", ci)}");         // Displays "N: 10,761.938"
Console.WriteLine($"P: {(floating/10000).ToString("P02", ci)}"); // Displays "P: 107.62 %"
Console.WriteLine($"R: {floating.ToString("R", ci)}");           // Displays "R: 10761.937554"
Console.WriteLine();

// Output integral values
int integral = 8395;
Console.WriteLine($"C: {integral.ToString("C", ci)}");           // Displays "C: $8,395.00"
Console.WriteLine($"D: {integral.ToString("D6", ci)}");          // Displays "D: 008395"
Console.WriteLine($"E: {integral.ToString("E03", ci)}");         // Displays "E: 8.395E+003"
Console.WriteLine($"F: {integral.ToString("F01", ci)}");         // Displays "F: 8395.0"
Console.WriteLine($"G: {integral.ToString("G", ci)}");           // Displays "G: 8395"
Console.WriteLine($"N: {integral.ToString("N01", ci)}");         // Displays "N: 8,395.0"
Console.WriteLine($"P: {(integral/10000.0).ToString("P02", ci)}"); // Displays "P: 83.95 %"
Console.WriteLine($"X: 0x{integral.ToString("X", ci)}");           // Displays "X: 0x20CB"
Console.WriteLine();
```

참고하십시오

- NumberFormatInfo
- 사용자 지정 숫자 형식 문자열
- 서식 유형
- 방법: 앞에 오는 0으로 숫자 채우기
- 복합 서식 지정
- 샘플: .NET Core WinForms 서식 유틸리티(C#)
- 샘플: .NET Core WinForms 서식 유틸리티(Visual Basic)

사용자 지정 숫자 형식 문자열

아티클 • 2025. 04. 24.

하나 이상의 사용자 지정 숫자 서식 지정자로 구성된 사용자 지정 숫자 서식 문자열을 만들어 숫자 데이터의 서식을 지정하는 방법을 정의할 수 있습니다. 사용자 지정 숫자 서식 문자열은 표준 숫자 서식 문자열이 아닌 모든 형식 문자열입니다.

사용자 지정 숫자 형식 문자열은 모든 숫자 형식의 메서드 오버로드에서 `ToString` 지원됩니다. 예를 들어 형식의 Int32 메서드 및 ToString(String, IFormatProvider) 숫자 형식 문자열을 ToString(String) 제공할 수 있습니다. 사용자 지정 숫자 형식 문자열은 클래스, 메서드 및 StringBuilder.AppendFormat 메서드의 ConsoleStreamWriter 일부 `Write` 및 `WriteLine` 메서드에서 사용되는 .NET 복합 서식 지정 기능에서도 지원됩니다String.Format. 문자열 보간 기능은 사용자 지정 숫자 형식 문자열도 지원합니다.

> **팁**
>
> 숫자 또는 날짜 및 시간 값에 서식 문자열을 적용하고 결과 문자열을 표시할 수 있는 .NET Core Windows Forms 애플리케이션인 서식 **유틸리티**를 다운로드할 수 있습니다. 소스 코드는 C# 및 Visual Basic에 사용할 수 있습니다.

다음 표에서는 사용자 지정 숫자 서식 지정자 및 각 서식 지정자로 생성되는 샘플 출력을 보여 줍니다. 사용자 지정 숫자 형식 문자열 사용에 대한 자세한 내용은 Notes 섹션을 참조하고 예제 섹션을 참조하세요.

테이블 확장

서식 지정자	이름	설명	예시
0	0 자리 표시자	해당 숫자가 있을 경우 0을 해당 숫자로 바꾸고, 그렇지 않으면 결과 문자열에 0을 표시합니다. 추가 정보: "0" 사용자 지정자입니다.	1234.5678 ("00000") -> 01235 0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46
"#"	10진수 자리 표시자	해당 숫자가 있을 경우 "#" 기호를 해당 숫자로 바꾸고, 그렇지 않으면 결과 문자열에 숫자를 표시하지 않습니다. 입력 문자열의 해당 숫자가 중요하지 않은 0이면 결과 문자열에 숫자가 나타나지 않습니다. 예를 들어 0003("####") -> 3입니다.	1234.5678 ("#####") -> 1235 0.45678 ("#.##", en-US) -> .46

서식 지정자	이름	설명	예시
		추가 정보: "#" 사용자 지정 지정자입니다.	0.45678 ("#.##", fr-FR) -> ,46
"."	소수점	결과 문자열에서 소수 구분 기호의 위치를 결정합니다. 추가 정보: "." 사용자 지정 지정자입니다.	0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46
","	그룹 구분 기호 및 숫자 배율	그룹 구분 기호 지정자와 숫자 배율 지정자로 모두 사용됩니다. 그룹 구분 기호로 사용될 경우 각 그룹 사이에 지역화된 그룹 구분 기호 문자를 삽입합니다. 숫자 크기 조정 지정자로 사용될 경우 숫자를 쉼표 단위로 끊어서 1000으로 나눕니다. 추가 정보: "," 사용자 지정자입니다.	그룹 구분 기호 지정자: 2147483647 ("##,#", en-US) -> 2,147,483,647 2147483647 ("##,#", es-ES) -> 2.147.483.647 배율 지정자: 2147483647 ("#,#,,", en-US) -> 2,147 2147483647 ("#,#,,", es-ES) -> 2.147
"%"	백분율 자리 표시자	숫자에 100을 곱하고 결과 문자열에 지역화된 백분율 기호를 삽입합니다. 추가 정보: "%" 사용자 지정자입니다.	0.3697 ("%#0.00", en-US) -> %36.97 0.3697 ("%#0.00", el-GR) -> %36,97 0.3697 ("##.0 %", en-US) -> 37.0 % 0.3697 ("##.0 %", el-GR) -> 37,0 %
"‰"	천분율 자리 표시자	숫자에 1000을 곱하고 결과 문자열에 지역화된 천분율 기호를 삽입합니다. 추가 정보: "‰" 사용자 지정 지정자입니다.	0.03697 ("#0.00‰", en-US) -> 36.97‰ 0.03697 ("#0.00‰", ru-RU) -> 36,97‰
"E0"	지수 표기	적어도 하나의 0이 뒤에 오면 지수 표기법을 사용하여 결과의 서식을 지정합니다. "E" 또는 "e" 문자는 결과 문자열에 표시되	987654("#0.0e0") -> 98.8e4

서식 지정자	이름	설명	예시
"E+0" "E-0" "E0" "E+0" "E-0"	법	는 지수 기호의 대/소문자를 나타냅니다. "E" 또는 "e" 문자 뒤에 오는 0의 수에 따라 지수의 최소 자릿수가 결정됩니다. 더하기 기호(+)는 기호 문자가 항상 지수 앞에 온다는 것을 나타냅니다. 빼기 기호(-)는 기호 문자가 음의 지수 앞에만 온다는 것을 나타냅니다. 추가 정보: "E" 및 "e" 사용자 지정 지정자입니다.	1503.92311 ("0.0##e+00") -> 1.504e+03 1.8901385E-16("0.0e+00") -> 1.9e-16
"\"	이스케이프 문자	다음 문자가 사용자 지정 형식 지정자가 아닌 리터럴로 해석되도록 합니다. 추가 정보: "\" 이스케이프 문자입니다.	987654 ("\###00\#") -> #987654#
'*string*' "*string*"	리터럴 문자열 구분 기호	괄호로 묶인 문자가 변경되지 않은 상태로 결과 문자열에 복사되어야 함을 나타냅니다. 추가 정보: 문자 리터럴.	68 ("# 'degrees'") -> 68도 68 ("#' degrees'") -> 68도
;	섹션 구분 기호	양수, 음수 및 0에 따라 별도의 서식 문자열을 사용하여 섹션을 정의합니다. 추가 정보: ";" 섹션 구분 기호입니다.	12.345 ("#0.0#;(#0.0#);-\0-") -> 12.35 0 ("#0.0#;(#0.0#);-\0-") -> -0- -12.345 ("#0.0#;(#0.0#);-\0-") -> (12.35) 12.345 ("#0.0#;(#0.0#)") -> 12.35 0 ("#0.0#;(#0.0#)") -> 0.0 -12.345 ("#0.0#;(#0.0#)") -> (12.35)
기타	다른 모든 문자	문자가 변경되지 않은 상태로 결과 문자열에 복사됩니다. 추가 정보: 문자 리터럴.	68 ("# °") -> 68 °

다음 단원에서는 각 사용자 지정 숫자 서식 지정자에 대해 자세히 설명합니다.

> **참고**
>
> 이 문서의 C# 예제 중 일부는 Try.NET 인라인 코드 실행기 및 플레이그라운드에서 실행됩니다. **실행** 단추를 선택하여 대화형 창에서 예제를 실행합니다. 코드를 실행하면 코드를 수정하고 **실행을** 다시 선택하여 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나 컴파일에 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

"0" 사용자 지정 지정자

"0" 사용자 지정 지정자는 0 자리 표시자 기호로 사용됩니다. 서식을 지정할 값이 서식 문자열의 0이 표시된 위치에 숫자를 가지고 있으면 해당 숫자가 결과 문자열로 복사되고, 그렇지 않으면 결과 문자열에 0이 표시됩니다. 소수점 앞 가장 왼쪽의 0과 소수점 뒤 가장 오른쪽 0의 위치는 결과 문자열에 항상 표시될 자릿수의 범위를 결정합니다.

"00" 지정자를 사용하면 해당 값이 소수점 뒤 첫째 자리에서 반올림되며 항상 0 이상의 정수 값으로 표시됩니다. 예를 들어, 34.5의 서식을 "00"으로 지정하면 결과는 35가 됩니다.

다음 예제에서는 0 자리 표시자가 포함된 사용자 지정 서식 문자열을 사용하여 여러 가지 값을 표시합니다.

C#

```
double value;

value = 123;
Console.WriteLine(value.ToString("00000"));
Console.WriteLine(String.Format("{0:00000}", value));
// Displays 00123

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value));
// Displays 01.20

CultureInfo daDK = CultureInfo.CreateSpecificCulture("da-DK");
Console.WriteLine(value.ToString("00.00", daDK));
Console.WriteLine(String.Format(daDK, "{0:00.00}", value));
// Displays 01,20

value = .56;
```

```
   Console.WriteLine(value.ToString("0.0", CultureInfo.InvariantCulture));
   Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                   "{0:0.0}", value));
   // Displays 0.6

   value = 1234567890;
   Console.WriteLine(value.ToString("0,0", CultureInfo.InvariantCulture));
   Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                   "{0:0,0}", value));
   // Displays 1,234,567,890

   CultureInfo elGR = CultureInfo.CreateSpecificCulture("el-GR");
   Console.WriteLine(value.ToString("0,0", elGR));
  Console.WriteLine(String.Format(elGR, "{0:0,0}", value));
   // Displays 1.234.567.890

   value = 1234567890.123456;
   Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture));
   Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                   "{0:0,0.0}", value));
   // Displays 1,234,567,890.1

   value = 1234.567890;
   Console.WriteLine(value.ToString("0,0.00", CultureInfo.InvariantCulture));
   Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                   "{0:0,0.00}", value));
   // Displays 1,234.57
```

테이블로 돌아가기

"#" 사용자 지정 지정자

"#" 사용자 지정 지정자는 숫자 표시자 기호로 사용됩니다. 서식을 지정할 값이 서식 문자열의 "#" 기호가 표시된 위치에 숫자를 가지고 있으면 해당 숫자가 결과 문자열로 복사되고, 그렇지 않으면 결과 문자열의 해당 위치에 아무 것도 저장되지 않습니다.

이 지정자는 문자열에서 0이 유일한 숫자인 경우에도 유효 자릿수가 아닌 0을 표시하지 않습니다. 표시되는 숫자에서 유효 자릿수인 경우에만 0이 표시됩니다.

"##" 서식 문자열을 사용하면 해당 값이 소수점 뒤 첫째 자리에서 반올림되며 항상 0 이상의 정수로 표시됩니다. 예를 들어, 34.5의 서식을 "##"으로 지정하면 결과는 35가 됩니다.

다음 예제에서는 숫자 자리 표시자가 포함된 사용자 지정 서식 문자열을 사용하여 여러 가지 값을 표시합니다.

C#

```
double value;
```

```
value = 1.2;
Console.WriteLine(value.ToString("#.##", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#.##}", value));
// Displays 1.2

value = 123;
Console.WriteLine(value.ToString("#####"));
Console.WriteLine(String.Format("{0:#####}", value));
// Displays 123

value = 123456;
Console.WriteLine(value.ToString("[##-##-##]"));
Console.WriteLine(String.Format("{0:[##-##-##]}", value));
 // Displays [12-34-56]

value = 1234567890;
Console.WriteLine(value.ToString("#"));
Console.WriteLine(String.Format("{0:#}", value));
// Displays 1234567890

Console.WriteLine(value.ToString("(###) ###-####"));
Console.WriteLine(String.Format("{0:(###) ###-####}", value));
// Displays (123) 456-7890
```

빈 숫자 또는 선행 0이 공백으로 대체되는 결과 문자열을 반환하려면 다음 예제와 같이 복합 서식 지정 기능을 사용하고 필드 너비를 지정합니다.

C#

```
using System;

public class SpaceOrDigit
{
   public static void Main()
   {
      Double value = .324;
      Console.WriteLine("The value is: '{0,5:#.###}'", value);
   }
}
// The example displays the following output if the current culture
// is en-US:
//      The value is: ' .324'
```

테이블로 돌아가기

"." 사용자 지정 지정자

"." 사용자 지정 서식 지정자는 결과 문자열에 지역화된 소수 구분 기호를 삽입합니다. 서식 문자열의 첫째 마침표 문자는 서식이 지정될 값에서 소수 구분 기호의 위치를 결정하며, 다른 마침표

문자는 무시됩니다. 서식 지정자가 "."로 끝나는 경우 유효 자릿수만 결과 문자열로 서식이 지정됩니다.

결과 문자열에서 소수 구분 기호로 사용되는 문자가 항상 마침표가 아닌 경우 서식을 NumberDecimalSeparator 제어하는 개체의 속성에 NumberFormatInfo 의해 결정됩니다.

다음 예제에서는 "." 서식 지정자를 사용하여 여러 결과 문자열에서 소수점의 위치를 정의합니다.

C#

```
double value;

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value));
// Displays 01.20

Console.WriteLine(value.ToString("00.00",
                  CultureInfo.CreateSpecificCulture("da-DK")));
Console.WriteLine(String.Format(CultureInfo.CreateSpecificCulture("da-DK"),
                  "{0:00.00}", value));
// Displays 01,20

value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%

value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.###E+0}", value));
 // Displays 8.6E+4
```

테이블로 돌아가기

"," 사용자 지정 지정자

"," 문자는 그룹 구분 기호 지정자와 숫자 배율 지정자로 사용됩니다.

- 그룹 구분 기호 지정자: 두 개의 10진수 자리 표시자(0 또는 #) 사이에 정수 계열 자릿수의 서식을 지정하는 하나 이상의 쉼표 문자가 지정된 경우 정수 계열 출력 부분의 각 숫자 그룹 사이에 그룹 구분 문자가 삽입됩니다.

  현재 NumberFormatInfo 개체의 속성 및 NumberGroupSizes 값은 NumberGroupSeparator 숫자 그룹 구분 기호로 사용되는 문자와 각 숫자 그룹의 크기를 결정합니다. 예를 들어, 문자열 "#,#"과 고정 문화권을 사용하여 숫자 1000의 서식을 지정할 경우 "1,000"이 출력됩니다.

- 숫자 배율 지정자: 명시적 또는 암시적 소수점의 바로 왼쪽에 하나 이상의 쉼표가 지정된 경우 서식이 지정될 숫자는 쉼표 단위로 끊어서 1000으로 나뉩니다. 예를 들어, 문자열 "0,,"을 사용하여 숫자 100000000의 서식을 지정할 경우 "100"이 출력됩니다.

동일한 서식 문자열에 그룹 구분 기호와 숫자 배율 지정자를 함께 사용할 수 있습니다. 예를 들어, 문자열 "#,0,,"과 고정 문화권을 사용하여 숫자 1000000000의 서식을 지정할 경우 "1,000"이 출력됩니다.

다음 예제에서는 쉼표를 그룹 구분 기호로 사용하는 방법을 보여 줍니다.

C#

```
double value = 1234567890;
Console.WriteLine(value.ToString("#,#", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,#}", value));
// Displays 1,234,567,890

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value));
// Displays 1,235
```

다음 예제에서는 쉼표를 숫자 크기 조정 지정자로 사용하는 방법을 보여 줍니다.

C#

```
double value = 1234567890;
Console.WriteLine(value.ToString("#,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,,}", value));
// Displays 1235

Console.WriteLine(value.ToString("#,,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,,,}", value));
// Displays 1

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
```

```
                                   "{0:#,##0,,}", value));
// Displays 1,235
```

테이블로 돌아가기

"%" 사용자 지정 지정자

서식 문자열의 백분율 기호(%)를 사용하면 서식이 지정되기 전에 숫자를 100으로 곱합니다. 서식 문자열에서 %가 표시된 위치에 있는 숫자에는 지역화된 백분율 기호가 삽입됩니다. 사용되는 백분율 문자는 현재 NumberFormatInfo 개체의 PercentSymbol 속성에 의해 정의됩니다.

다음 예제에서는 "%" 사용자 지정 지정자를 포함하는 여러 사용자 지정 형식 문자열을 정의합니다.

C#

```
double value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%
```

테이블로 돌아가기

"‰" 사용자 지정 지정자

형식 문자열의 밀리당 문자(‰ 또는 \u2030)를 사용하면 서식이 지정되기 전에 숫자를 1000으로 곱합니다. 서식 문자열에서 ‰가 표시된 위치에 있는 반환된 문자열에는 적절한 천분율 기호가 삽입됩니다. 사용되는 밀리당 문자는 문화권별 서식 정보를 제공하는 개체의 속성에 의해 NumberFormatInfo.PerMilleSymbol 정의됩니다.

다음 예제에서는 "‰" 사용자 지정 지정자가 포함된 사용자 지정 서식 문자열을 정의합니다.

C#

```
double value = .00354;
string perMilleFmt = "#0.## " + '\u2030';
Console.WriteLine(value.ToString(perMilleFmt, CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:" + perMilleFmt + "}", value));
// Displays 3.54‰
```

테이블로 돌아가기

"E" 및 "e" 사용자 지정 지정자

서식 문자열에 "E", "E+", "E-", "e", "e+" 또는 "e-" 문자열이 표시되고 바로 뒤에 적어도 하나의 0이 오면, 해당 수와 지수 사이에 "E" 또는 "e"가 삽입되는 과학적 표기법으로 서식이 지정됩니다. 과학적 표기법 표시기 뒤에 오는 0의 개수는 이 숫자의 지수로 나타낼 최소 자릿수를 결정합니다. "E+" 및 "e+" 서식은 더하기 기호나 빼기 기호가 항상 지수 앞에 와야 한다는 것을 나타냅니다. "E", "E-", "e" 또는 "e-" 서식은 기호 문자가 음의 지수 앞에만 와야 한다는 것을 나타냅니다.

다음 예제에서는 과학적 표기법 지정자를 사용하여 여러 가지 숫자 값에 서식을 지정합니다.

C#

```
double value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+0}", value));
// Displays 8.6E+4

Console.WriteLine(value.ToString("0.###E+000", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+000}", value));
// Displays 8.6E+004

Console.WriteLine(value.ToString("0.###E-000", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E-000}", value));
// Displays 8.6E004
```

테이블로 돌아가기

"\" 이스케이프 문자

서식 문자열의 "#", "0", ".", ",", "%" 및 "‰" 기호는 리터럴 문자가 아닌 서식 지정자로 해석됩니다. 사용자 지정 서식 문자열의 위치에 따라 대문자 및 소문자 "E"뿐만 아니라 + 및 - 기호도 형식 지정자로 해석될 수 있습니다.

문자가 형식 지정자로 해석되지 않도록 하려면 이스케이프 문자인 백슬래시를 앞에 두면 됩니다. 이스케이프 문자는 뒤에 오는 문자가 변경되지 않은 상태로 결과 문자열에 포함되어야 하는 문자 리터럴임을 나타냅니다.

결과 문자열에 백슬래시를 포함하려면 `\\` 처럼 두 개의 백슬래시를 연속해서 입력해야 합니다.

참고

C++ 및 C# 컴파일러 같은 일부 컴파일러에서는 하나의 백슬래시 문자가 이스케이프 문자로 해석될 수도 있습니다. 형식을 지정할 때 문자열이 올바로 해석되도록 하려면 해당 문자열 앞에 축자 문자열 리터럴 문자(@ 문자)를 사용하거나(C#의 경우) 각 백슬래시 앞에 또 다른 백슬래시를 추가하면 됩니다(C# 및 C++의 경우). 다음 C# 예제에서는 이 두 가지 방법을 모두 보여 줍니다.

다음 예제에서는 이스케이프 문자를 사용하여 서식 지정 작업이 "#", "0" 및 "\" 문자를 이스케이프 문자 또는 형식 지정자로 해석하지 못하도록 합니다. C# 예제에서는 백슬래시를 추가로 사용하여 백슬래시를 리터럴 문자로 해석합니다.

C#

```
int value = 123;
Console.WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \\0\\0 cents
\\#\\#\\#"));
Console.WriteLine(String.Format("{0:\\#\\#\\# ##0 dollars and \\0\\0 cents
\\#\\#\\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString(@"\#\#\# ##0 dollars and \0\0 cents \#\#\#"));
Console.WriteLine(String.Format(@"{0:\#\#\# ##0 dollars and \0\0 cents \#\#\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString("\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents
\\\\\\\\\\\\"));
Console.WriteLine(String.Format("{0:\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents
\\\\\\\\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\

Console.WriteLine(value.ToString(@"\\\\\\ ##0 dollars and \0\0 cents \\\\\\"));
Console.WriteLine(String.Format(@"{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\
```

테이블로 돌아가기

";" 섹션 구분 기호

세미콜론(;)은 해당 값이 양수인지 여부에 따라 서로 다른 서식을 숫자에 적용하는 조건부 서식 지정자입니다. 이렇게 하려면 사용자 지정 서식 문자열에 세미콜론으로 구분된 최대 세 개의 섹션이 포함되어야 합니다. 다음 표에서는 이러한 섹션에 대해 설명합니다.

테이블 확장

섹션 수	설명
한 섹션	형식 문자열이 모든 값에 적용됩니다.
두 섹션	첫째 섹션은 양수 값과 0에 적용되고, 둘째 섹션은 음수 값에 적용됩니다. 서식을 지정할 수가 음수였는데 둘째 섹션의 서식에 따라 반올림한 후에 0이 된 경우, 결과값 0은 첫째 섹션에 따라 서식이 지정됩니다.
세 개의 섹션	첫째 섹션은 양수 값에 적용되고, 둘째 섹션은 음수 값에 적용되며, 셋째 섹션은 0에 적용됩니다. 세미콜론 사이에 아무 것도 없어서 둘째 섹션이 비어 있는 경우에는 첫째 섹션을 0이 아닌 모든 값에 적용합니다. 서식을 지정할 수가 0이 아니었는데 첫째 또는 둘째 섹션의 서식에 따라 반올림한 후에 0이 된 경우, 결과값 0은 셋째 섹션에 따라 서식이 지정됩니다.

섹션 구분 기호는 마지막 값의 서식을 지정할 때 숫자와 연관된 기존 서식을 무시합니다. 예를 들어, 섹션 구분 기호가 사용되면 음수 값은 항상 빼기 기호 없이 표시됩니다. 마지막에 서식을 지정한 값에 빼기 기호를 붙이려면 빼기 기호를 사용자 지정 서식 지정자의 일부로 명시적으로 포함시켜야 합니다.

다음 예제에서는 ";" 서식 지정자를 사용하여 양수, 음수 및 0의 서식을 각각 다르게 지정합니다.

C#

```
double posValue = 1234;
double negValue = -1234;
double zeroValue = 0;

string fmt2 = "##;(##)";
string fmt3 = "##;(##);**Zero**";

Console.WriteLine(posValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", posValue));
// Displays 1234

Console.WriteLine(negValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", negValue));
// Displays (1234)

Console.WriteLine(zeroValue.ToString(fmt3));
Console.WriteLine(String.Format("{0:" + fmt3 + "}", zeroValue));
// Displays **Zero**
```

테이블로 돌아가기

문자 리터럴

사용자 지정 숫자 서식 문자열에서 나타나는 서식 지정자는 리터럴 문자가 아닌 서식 지정 문자로 해석됩니다. 여기에는 다음과 같은 문자가 포함됩니다.

- 0
- #
- %
- ‰
- '
- \
- .
- ,
- 형식 문자열의 위치에 따라 E 또는 e입니다.

다른 모든 문자는 항상 문자 리터럴로 해석되며, 형식 지정 작업에서 변경되지 않고 결과 문자열에 포함됩니다. 구문 분석 작업에서는 입력 문자열의 문자와 정확히 일치해야 하며, 비교 시 대/소문자를 구분합니다.

다음 예제에서는 리터럴 문자 단위(이 경우에는 천)의 일반적인 사용을 한 가지 보여줍니다.

C#

```
double n = 123.8;
Console.WriteLine($"{n:#,##0.0K}");
// The example displays the following output:
//       123.8K
```

문자가 결과 문자열에 포함되거나 입력 문자열에서 성공적으로 구문 분석될 수 있도록 하기 위해 문자를 서식 지정 문자가 아니라 리터럴 문자로 해석되도록 지정하는 두 가지 방법이 있습니다.

- 서식 지정 문자를 이스케이프하는 방법입니다. 자세한 내용은 "\" 이스케이프 문자를 참조하세요.
- 전체 리터럴 문자열을 따옴표 아포스트로피로 묶습니다.

다음 예제에서는 사용자 지정 숫자 서식 문자열에서 예약된 문자를 포함하는 두 가지 방법을 사용합니다.

C#

```
double n = 9.3;
Console.WriteLine($@"{n:##.0\%}");
Console.WriteLine($@"{n:\'##\'}");
```

```
Console.WriteLine($@"{n:\\##\\}");
Console.WriteLine();
Console.WriteLine($"{n:##.0'%'}");
Console.WriteLine($@"{n:'\'##'\'}");
// The example displays the following output:
//       9.3%
//       '9'
//       \9\
//
//       9.3%
//       \9\
```

비고

부동 소수점 무한대 및 NaN

형식 문자열에 관계없이 , 또는 부동 소수점 형식의 Half값이 무한대, 음수 무한대 또는 숫자(NaN)가 아닌 경우 서식이 지정된 문자열은 현재 적용 가능한 NumberFormatInfo 개체에서 지정한 각 NegativeInfinitySymbolPositiveInfinitySymbolNaNSymbol 또는 속성의 값입니다.DoubleSingle

제어판 설정

제어판의 **국가별 및 언어 옵션** 항목의 설정은 서식 지정 작업으로 생성된 결과 문자열에 영향을 미칩니다. 이러한 설정은 현재 문화권과 연결된 개체를 NumberFormatInfo 초기화하는 데 사용되며, 현재 문화권은 서식을 제어하는 데 사용되는 값을 제공합니다. 다른 설정을 사용하는 컴퓨터는 다른 결과 문자열을 생성합니다.

또한 생성자를 사용하여 CultureInfo(String) 현재 시스템 문화권과 동일한 문화권을 나타내는 새 CultureInfo 개체를 인스턴스화하는 경우 제어판의 **국가 및 언어 옵션** 항목에 의해 설정된 모든 사용자 지정이 새 CultureInfo 개체에 적용됩니다. 생성자를 사용하여 CultureInfo(String, Boolean) 시스템의 사용자 지정을 CultureInfo 반영하지 않는 개체를 만들 수 있습니다.

반올림 및 고정 소수점 서식 문자열

고정 소수점 서식 문자열(즉, 과학적 표기법 서식 문자를 포함하지 않는 서식 문자열)의 경우 소수점 오른쪽에 숫자 자리 표시자가 있는 만큼 소수 자릿수로 반올림됩니다. 서식 문자열에 소수점이 없으면 숫자가 가장 가까운 정수로 반올림됩니다. 숫자의 자릿수가 소수점 왼쪽의 10진수 자리 표시자보다 많으면, 초과 숫자가 결과 문자열의 첫째 10진수 자리 표시자 바로 앞에 복사됩니다.

테이블로 돌아가기

예시

다음 예제에서는 두 개의 사용자 지정 숫자 서식 문자열을 보여 줍니다. 두 경우 모두에서 숫자 자리 표시자(`#`)는 숫자 데이터를 표시하며, 다른 모든 문자는 결과 문자열에 복사됩니다.

C#

```
double number1 = 1234567890;
string value1 = number1.ToString("(###) ###-####");
Console.WriteLine(value1);

int number2 = 42;
string value2 = number2.ToString("My Number = #");
Console.WriteLine(value2);
// The example displays the following output:
//       (123) 456-7890
//       My Number = 42
```

테이블로 돌아가기

참고하십시오

- System.Globalization.NumberFormatInfo
- 서식 유형
- 표준 숫자 형식 문자열
- 방법: 앞에 오는 0으로 숫자 채우기
- 샘플: .NET Core WinForms 서식 유틸리티(C#)
- 샘플: .NET Core WinForms 서식 유틸리티(Visual Basic)

표준 날짜 및 시간 형식 문자열

표준 날짜 및 시간 형식 문자열은 형식 지정자로 단일 문자를 사용하여 , DateTime또는 DateTimeOffsetDateOnly 값의 텍스트 표현을 TimeOnly정의합니다. 공백을 포함하여 문자를 둘 이상 포함하는 날짜 및 시간 서식은 사용자 지정 날짜 및 시간 서식 문자열로 해석됩니다. 표준 또는 사용자 지정 형식 문자열은 다음 두 가지 방법으로 사용할 수 있습니다.

- 서식 지정 작업에서 발생하는 문자열을 정의합니다.
- 구문 분석 작업으로 , 또는 값으로 변환 DateTimeDateTimeOffsetDateOnlyTimeOnly 할 수 있는 날짜 및 시간 값의 텍스트 표현을 정의합니다.

> **팁**
>
> 숫자 또는 날짜 및 시간 값에 서식 문자열을 적용하고 결과 문자열을 표시할 수 있는 .NET Windows Forms 애플리케이션인 **서식 유틸리티**다운로드할 수 있습니다. 소스 코드는 C# 및 Visual Basic사용할 수 있습니다.

서식 지정자 테이블

다음 표에서는 표준 날짜 및 시간 서식 지정자에 대해 설명합니다. 달리 명시되지 않는 한 특정 표준 날짜 및 시간 형식 지정자는 DateTime 또는 DateTimeOffset 값과 함께 사용되는지 여부에 관계없이 동일한 문자열 표현을 생성합니다. 모든 형식 지정자를 값과 함께 사용할 수 있는 것은 아닙니다. 자세한 내용은 DateOnly을 참조하세요.TimeOnly Windows 및 현재 DateTimeFormatInfo 개체의 국가별 설정이 날짜 및 시간 서식에 미치는 영향에 대한 자세한 내용은 제어판 설정 및 DateTimeFormatInfo 속성을 참조하세요.

테이블 확장

서식 지정자	설명	예시
"d"	간단한 날짜 패턴입니다. 추가 정보:짧은 날짜("d") 형식 지정자가.	2009-06-15T13:45:30 -> 2009년 6월 15일(en-US) 2009-06-15T13:45:30 -> 15/06/2009(fr-FR) 2009-06-15T13:45:30 -> 2009/06/15(ja-JP) DateOnly (2009-06-15) -> 2009년 6월 15일 (en-US)

서식 지정자	설명	예시
"D"	자세한 날짜 패턴입니다. 추가 정보:긴 날짜("D") 형식 지정자가.	2009-06-15T13:45:30 -> 2009년 6월 15일 월요일(en-US) 2009-06-15T13:45:30 -> понедельник, 15 июня 2009 . (ru-RU) 2009-06-15T13:45:30 -> 월요일, 15. Juni 2009 (de-DE)
"f"	전체 날짜/시간 패턴(간단한 시간)입니다. 추가 정보: 전체 날짜 짧은 시간("f") 형식 지정자입니다.	2009-06-15T13:45:30 -> 2009년 6월 15일 월요일 오후 1:45(en-US) 2009-06-15T13:45:30 -> 2009년 6월 15일 13:45 (sv-SE) 2009-06-15T13:45:30 -> Σσσα, 15 σ 2009 1:45 μμ (el-GR)
"F"	전체 날짜/시간 패턴(자세한 시간) 추가 정보: 전체 날짜 긴 시간("F") 형식 지정자입니다.	2009-06-15T13:45:30 -> 2009년 6월 15일 월요일 오후 1:45:30(en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45:30 (sv-SE) 2009-06-15T13:45:30 -> Σσσα, 15 σ 2009 1:45:30 μμ (el-GR)
"g"	일반 날짜/시간 패턴(간단한 시간) 추가 정보: 일반 날짜 짧은 시간("g") 형식 지정자입니다.	2009-06-15T13:45:30 -> 2009년 6월 15일 오후 1:45(en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45(es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45(zh-CN)
"G"	일반 날짜/시간 패턴(자세한 시간)입니다. 추가 정보: 일반 날짜 긴 시간("G") 형식 지정자입니다.	2009-06-15T13:45:30 -> 6/15/2009 1:45:30 PM(en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45:30(es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45:30(zh-CN)
"M", "m"	월/일 패턴입니다. 추가 정보: 월("M", "m") 형식 지정자입니다.	2009-06-15T13:45:30 -> 6월 15일(en-US) 2009-06-15T13:45:30 -> 15. juni (da-DK) 2009-06-15T13:45:30 -> 15 Juni(id-ID)
"O", "o"	왕복 날짜/시간 패턴입니다. 추가 정보: 왕복("O", "o") 형식 지정자입니다.	DateTime 값: 2009-06-15T13:45:30 (DateTimeKind.Local) --> 2009-06-15T13:45:30.0000000-07:00

서식 지정자	설명	예시
		2009-06-15T13:45:30 (DateTimeKind.Utc) --> 2009-06-15T13:45:30.00000000Z 2009-06-15T13:45:30 (DateTimeKind.Unspecified) --> 2009-06-15T13:45:30.0000000 DateTimeOffset 값: 2009-06-15T13:45:30-07:00 --> 2009-06-15T13:45:30.0000000-07:00 DateOnly 값: 2009-06-15 --> 2009-06-15 TimeOnly 값: 13:45:30 --> 13:45:30.0000000
"R", "r"	RFC1123 패턴입니다. 추가 정보: RFC1123("R", "r") 형식 지정자입니다.	DateTimeOffset 입력: 2009-06-15T13:45:30 -> 월, 15 6월 2009 20:45:30 GMT DateTime 입력: 2009-06-15T13:45:30 -> 월, 15 6월 2009 13:45:30 GMT
"s"	정렬 가능한 날짜/시간 패턴입니다. 추가 정보: 정렬 가능한("s") 형식 지정자입니다.	2009-06-15T13:45:30(DateTimeKind.Local) -> 2009-06-15T13:45:30 2009-06-15T13:45:30 (DateTimeKind.Utc) -> 2009-06-15T13:45:30
"t"	간단한 시간 패턴입니다. 추가 정보: 짧은 시간("t") 형식 지정자입니다.	2009-06-15T13:45:30 -> 오후 1:45(en-US) 2009-06-15T13:45:30 -> 13:45(hr-HR) 2009-06-15T13:45:30 -> 01:45 م (ar-EG) TimeOnly (13:45:30) -> 오후 1:45(en-US)
"T"	자세한 시간 패턴 추가 정보: 장시간("T") 형식 지정자입니다.	2009-06-15T13:45:30 -> 오후 1:45:30(en-US) 2009-06-15T13:45:30 -> 13:45:30(hr-HR) 2009-06-15T13:45:30 -> 01:45:30 م(ar-EG) TimeOnly (13:45:30) -> 오후 1:45:30(en-US)
"u"	정렬 가능한 유니버설 날짜/시간 패턴	DateTime 값 사용: 2009-06-15T13:45:30 -> 2009-06-15 13:45:30Z

서식 지정자	설명	예시
	추가 정보: 범용 정렬 가능("u") 형식 지정자입니다.	DateTimeOffset 값 사용: 2009-06-15T13:45:30 -> 2009-06-15 20:45:30Z
"U"	범용 전체 날짜/시간 패턴입니다. 추가 정보: 범용 전체("U") 형식 지정자.	2009-06-15T13:45:30 -> 2009년 6월 15일 월요일 오후 8:45:30(en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 20:45:30 (sv-SE) 2009-06-15T13:45:30 -> Σσσα, 15 σ 2009 8:45:30 μ (el-GR)
"Y", "y"	연도 월 패턴 추가 정보: 연도 월("Y") 형식 지정자입니다.	2009-06-15T13:45:30 -> 2009년 6월(en-US) 2009-06-15T13:45:30 -> juni 2009(da-DK) 2009-06-15T13:45:30 -> Juni 2009(id-ID)
기타 모든 단일 문자	알 수 없는 지정자입니다.	런타임 FormatException을 throw합니다.

표준 서식 문자열의 작동 방법

서식 지정 작업에서 표준 서식 문자열은 단순히 사용자 지정 서식 문자열에 대한 별칭입니다. 별칭을 사용하여 사용자 지정 서식 문자열을 참조할 경우 별칭은 변하지 않지만 사용자 지정 서식 문자열 자체는 변경될 수 있다는 장점이 있습니다. 이는 날짜 및 시간 값에 대한 문자열 표현이 일반적으로 문화권마다 다르다는 점을 감안했을 때 매우 중요한 기능입니다. 예를 들어, "d" 표준 서식 문자열은 날짜 및 시간 값을 간단한 날짜 패턴을 사용하여 표시함을 나타냅니다. 고정 문화권의 경우 이 패턴이 "MM/dd/yyyy"이고 fr-FR 문화권의 경우 "dd/MM/yyyy"이며 ja-JP 문화권의 경우 "yyyy/MM/dd"입니다.

서식 지정 작업에서 표준 서식 문자열이 특정 문화권의 사용자 지정 서식 문자열에 매핑되는 경우 애플리케이션에서는 다음 방법 중 하나로 사용할 사용자 지정 서식 문자열이 포함된 특정 문화권을 정의할 수 있습니다.

- 기본 또는 현재 문화권을 사용할 수 있습니다. 다음 예제에서는 현재 문화권의 간단한 날짜 서식을 사용하여 날짜를 표시합니다. 이 예제의 경우 현재 문화권이 en-US입니다.

  C#

  ```
  // Display using current (en-us) culture's short date format
  DateTime thisDate = new DateTime(2008, 3, 15);
  ```

```
Console.WriteLine(thisDate.ToString("d"));          // Displays 3/15/2008
```

- CultureInfo 매개 변수가 있는 메서드에 서식을 사용할 문화권을 나타내는 IFormatProvider 개체를 전달할 수 있습니다. 다음 예제에서는 pt-BR 문화권의 간단한 날짜 서식을 사용하여 날짜를 표시합니다.

C#

```
// Display using pt-BR culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
CultureInfo culture = new CultureInfo("pt-BR");
Console.WriteLine(thisDate.ToString("d", culture));  // Displays 15/3/2008
```

- DateTimeFormatInfo 매개 변수가 있는 메서드에 서식 정보를 제공하는 IFormatProvider 개체를 전달할 수 있습니다. 다음 예제에서는 hr-HR 문화권에 대 한 DateTimeFormatInfo 개체의 짧은 날짜 형식을 사용 하 여 날짜를 표시 합니다.

C#

```
// Display using date format information from hr-HR culture
DateTime thisDate = new DateTime(2008, 3, 15);
DateTimeFormatInfo fmt = (new CultureInfo("hr-HR")).DateTimeFormat;
Console.WriteLine(thisDate.ToString("d", fmt));      // Displays 15.3.2008
```

> ⓘ **참고 항목**
>
> 날짜 및 시간 값 서식 지정에 사용되는 패턴 또는 문자열을 사용자 지정하는 방법에 대한 자세한 내용은 NumberFormatInfo 클래스 항목을 참조하세요.

표준 서식 문자열을 보다 긴 고정 사용자 지정 서식 문자열에 대한 약식 표현으로 사용하는 경우도 있습니다. 표준 서식 문자열은 "O"(또는 "o"), "R"(또는 "r"), "s" 및 "u"라는 네 범주로 나뉩니다. 이 문자열은 고정 문화권에 정의된 사용자 지정 서식 문자열에 대응되며, 문화권마다 동일하게 인식되는 날짜 및 시간 값에 대한 문자열 표현을 생성합니다. 다음 표에서는 이러한 네 가지 표준 날짜 및 시간 서식 문자열에 대해 설명합니다.

테이블 확장

표준 서식 문자열	DateTimeFormatInfo.InvariantInfo 속성으로 정의됨	사용자 지정 서식 문자열
"O" 또는 "o"	없음	yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffK
"R" 또는 "r"	RFC1123Pattern	ddd, dd MMM yyyy HH':'mm':'ss 'GMT'

표준 서식 문자열	DateTimeFormatInfo.InvariantInfo 속성으로 정의됨	사용자 지정 서식 문자열
"s"	SortableDateTimePattern	yyyy'-'MM'-'dd'T'HH':'mm':'ss
"u"	UniversalSortableDateTimePattern	yyyy'-'MM'-'dd HH':'mm':'ss'Z'

표준 형식 문자열은 구문 분석 작업이 성공하려면 특정 패턴을 정확하게 준수하기 위해 입력 문자열이 필요한 , DateTime.ParseExactDateTimeOffset.ParseExact및 DateOnly.ParseExact 메서드를 사용하여 구TimeOnly.ParseExact문 분석 작업에 사용할 수도 있습니다. 많은 표준 서식 문자열이 여러 사용자 지정 형식 문자열에 매핑되므로 날짜 및 시간 값을 다양한 형식으로 나타낼 수 있으며 구문 분석 작업은 여전히 성공합니다.

DateTimeFormatInfo.GetAllDateTimePatterns(Char) 메서드를 호출하여 표준 형식 문자열에 해당하는 사용자 지정 형식 문자열 또는 문자열을 확인할 수 있습니다. 다음 예제에서는 "d"(짧은 날짜 패턴) 표준 서식 문자열에 매핑되는 사용자 지정 서식 문자열을 표시합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      Console.WriteLine("'d' standard format string:");
      foreach (var customString in
DateTimeFormatInfo.CurrentInfo.GetAllDateTimePatterns('d'))
         Console.WriteLine($"   {customString}");
   }
}
// The example displays the following output:
//       'd' standard format string:
//          M/d/yyyy
//          M/d/yy
//          MM/dd/yy
//          MM/dd/yyyy
//          yy/MM/dd
//          yyyy-MM-dd
//          dd-MMM-yy
```

다음 섹션에서는 , DateTime및 DateTimeOffsetDateOnly 값에 대한 표준 형식 지정자에 대해 TimeOnly설명합니다.

날짜 서식

이 그룹에는 다음 서식이 포함됩니다.

- 간단한 날짜("d") 서식 지정자
- 자세한 날짜("D") 서식 지정자

간단한 날짜("d") 서식 지정자

"d" 표준 형식 지정자는 특정 문화권의 DateTimeFormatInfo.ShortDatePattern 속성으로 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어 고정 문화권의 ShortDatePattern 속성에서 반환되는 사용자 지정 형식 문자열은 "MM/dd/yyyy"입니다.

다음 표에서는 반환된 문자열의 서식을 제어하는 DateTimeFormatInfo 개체 속성을 나열합니다.

테이블 확장

재산	설명
ShortDatePattern	결과 문자열의 전체 형식을 정의합니다.
DateSeparator	날짜의 연도, 월 및 일 구성 요소를 구분하는 문자열을 정의합니다.

다음 예제에서는 "d" 형식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008,4, 10);
Console.WriteLine(date1.ToString("d", DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays 4/10/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("en-NZ")));
// Displays 10/04/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("de-DE")));
// Displays 10.04.2008
```

표로 이동

자세한 날짜("D") 서식 지정자

"D" 표준 형식 지정자는 현재 DateTimeFormatInfo.LongDatePattern 속성으로 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어, 고정 문화권에 대한 사용자 지정 서식 문자열은 "dddd, dd MMMM yyyy"입니다.

다음 표에서는 반환된 문자열의 서식을 제어하는 DateTimeFormatInfo 개체의 속성을 나열합니다.

테이블 확장

재산	설명
LongDatePattern	결과 문자열의 전체 형식을 정의합니다.
DayNames	결과 문자열에 나타날 수 있는 지역화된 날짜 이름을 정의합니다.
MonthNames	결과 문자열에 나타날 수 있는 지역화된 월 이름을 정의합니다.

다음 예제에서는 "D" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10);
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("pt-BR")));
// Displays quinta-feira, 10 de abril de 2008
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("es-MX")));
// Displays jueves, 10 de abril de 2008
```

표로 이동

날짜 및 시간 형식

이 그룹에는 다음 서식이 포함됩니다.

- 전체 날짜 간단한 시간("f") 서식 지정자
- 전체 날짜 자세한 시간("F") 서식 지정자
- 일반 날짜 간단한 시간("g") 서식 지정자
- 일반 날짜 자세한 시간("G") 서식 지정자
- 왕복("O", "o") 서식 지정자
- RFC1123("R", "r") 서식 지정자
- 정렬 가능한("s") 서식 지정자
- 정렬 가능한 유니버설("u") 서식 지정자
- 범용 전체("U") 형식 지정자

전체 날짜 간단한 시간("f") 서식 지정자

"f" 표준 서식 지정자는 공백으로 구분된 자세한 날짜("D")와 간단한 시간("t") 패턴의 조합입니다.

결과 문자열은 특정 DateTimeFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 DateTimeFormatInfo.LongDatePattern 및 DateTimeFormatInfo.ShortTimePattern 속성에서 반환된 사용자 지정 형식 지정자가 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
LongDatePattern	결과 문자열의 날짜 구성 요소 형식을 정의합니다.
ShortTimePattern	결과 문자열의 시간 구성 요소 형식을 정의합니다.
DayNames	결과 문자열에 나타날 수 있는 지역화된 날짜 이름을 정의합니다.
MonthNames	결과 문자열에 나타날 수 있는 지역화된 월 이름을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

다음 예제에서는 "f" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("f",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30 AM
Console.WriteLine(date1.ToString("f",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30
```

표로 이동

전체 날짜 자세한 시간("F") 서식 지정자

"F" 표준 형식 지정자는 현재 DateTimeFormatInfo.FullDateTimePattern 속성에 의해 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어, 고정 문화권에 대한 사용자 지정 서식 문자열은 "dddd, dd MMMM yyyy HH:mm:ss"입니다.

다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 FullDateTimePattern 속성에서 반환되는 사용자 지정 형식 지정자는 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
FullDateTimePattern	결과 문자열의 전체 형식을 정의합니다.
DayNames	결과 문자열에 나타날 수 있는 지역화된 날짜 이름을 정의합니다.
MonthNames	결과 문자열에 나타날 수 있는 지역화된 월 이름을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

다음 예제에서는 "F" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("F",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30:00 AM
Console.WriteLine(date1.ToString("F",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30:00
```

표로 이동

일반 날짜 간단한 시간("g") 서식 지정자

"g" 표준 서식 지정자는 공백으로 구분된 간단한 날짜("d")와 간단한 시간("t") 패턴의 조합입니다.

결과 문자열은 특정 DateTimeFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 DateTimeFormatInfo.ShortDatePattern 및 DateTimeFormatInfo.ShortTimePattern 속성에서 반환되는 사용자 지정 형식 지정자는 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
ShortDatePattern	결과 문자열의 날짜 구성 요소 형식을 정의합니다.
ShortTimePattern	결과 문자열의 시간 구성 요소 형식을 정의합니다.

재산	설명
DateSeparator	날짜의 연도, 월 및 일 구성 요소를 구분하는 문자열을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

다음 예제에서는 "g" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("g",
                  DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30
Console.WriteLine(date1.ToString("g",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30 AM
Console.WriteLine(date1.ToString("g",
                  CultureInfo.CreateSpecificCulture("fr-BE")));
// Displays 10/04/2008 6:30
```

표로 이동

일반 날짜 자세한 시간("G") 서식 지정자

"G" 표준 서식 지정자는 공백으로 구분된 간단한 날짜("d")와 자세한 시간("T") 패턴의 조합입니다.

결과 문자열은 특정 DateTimeFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 DateTimeFormatInfo.ShortDatePattern 및 DateTimeFormatInfo.LongTimePattern 속성에서 반환되는 사용자 지정 형식 지정자는 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
ShortDatePattern	결과 문자열의 날짜 구성 요소 형식을 정의합니다.
LongTimePattern	결과 문자열의 시간 구성 요소 형식을 정의합니다.
DateSeparator	날짜의 연도, 월 및 일 구성 요소를 구분하는 문자열을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.

재산	설명
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

다음 예제에서는 "G" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("G",
                  DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30:00
Console.WriteLine(date1.ToString("G",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30:00 AM
Console.WriteLine(date1.ToString("G",
                  CultureInfo.CreateSpecificCulture("nl-BE")));
// Displays 10/04/2008 6:30:00
```

표로 이동

왕복("O", "o") 서식 지정자

"O" 또는 "o" 표준 서식 지정자는 표준 시간대 정보를 유지하는 패턴을 사용하여 사용자 지정 날짜 및 시간 서식 문자열을 나타내고 ISO 8601을 준수하는 결과 문자열을 생략합니다. DateTime 값의 경우 이 서식 지정자는 텍스트의 DateTime.Kind 속성과 함께 날짜 및 시간 값을 유지하도록 설계되었습니다. 형식이 지정된 문자열은 DateTime.Parse(String, IFormatProvider, DateTimeStyles) 매개 변수가 DateTime.ParseExact설정된 경우 `styles` 또는 DateTimeStyles.RoundtripKind 메서드를 사용하여 다시 구문 분석할 수 있습니다.

값의 경우 DateOnly 이 형식 지정자는 날짜 전용 ISO 8601 문자열을 "yyyy-MM-dd" 형식으로 생성합니다. 값의 경우 TimeOnly 시간 전용 ISO 8601 문자열을 "HH:mm:ss.fffffff" 형식으로 생성합니다.

"O" 또는 "o" 표준 형식 지정자는 "yyyy'-'MM'-'dd'T'HH':'mm':'ss'에 해당합니다.' fffffffK" DateTime 값 및 "yyyy'-'MM'-'dd'T'HH':'mm':'ss'에 대한 사용자 지정 형식 문자열입니다. fffffffzzz" DateTimeOffset 값에 대한 사용자 지정 형식 문자열입니다. 이 문자열에서 하이픈, 콜론 및 문자 "T"와 같은 개별 문자를 구분하는 작은따옴표 쌍은 개별 문자가 변경할 수 없는 리터럴임을 나타냅니다. 아포스트로피는 출력 문자열에 나타나지 않습니다.

"O" 또는 "o" 표준 형식 지정자(및 "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.' fffffffK" 사용자 지정 형식 문자열)은 ISO 8601이 표준 시간대 정보를 나타내는 세 가지 방법을 활용하여 Kind 값의 DateTime 속성을 유지합니다.

- DateTimeKind.Local 날짜 및 시간 값의 표준 시간대 구성 요소는 UTC의 오프셋입니다(예: +01:00, -07:00). 모든 DateTimeOffset 값도 이 형식으로 표시됩니다.
- DateTimeKind.Utc 날짜 및 시간 값의 표준 시간대 구성 요소는 UTC를 나타내기 위해 "Z"(오프셋 0을 의미)를 사용합니다.
- DateTimeKind.Unspecified 날짜 및 시간 값에는 표준 시간대 정보가 없습니다.

"O" 또는 "o" 표준 서식 지정자는 국제 표준을 준수하므로, 이 지정자를 사용하는 서식 지정 또는 구문 분석 작업에서는 항상 고정 문화권 및 양력을 사용합니다.

, , `Parse` 및 `TryParse` `ParseExact` `TryParseExact` DateTimeDateTimeOffset 메서드에 DateOnlyTimeOnly전달되고 이러한 형식 중 하나인 경우 "O" 또는 "o" 형식 지정자를 사용하여 구문 분석할 수 있는 문자열입니다. DateTime 개체의 경우 호출하는 구문 분석 오버로드에는 값이 `styles` DateTimeStyles.RoundtripKind 매개 변수도 포함되어야 합니다. "O" 또는 "o" 형식 지정자에 해당하는 사용자 지정 형식 문자열을 사용하여 구문 분석 메서드를 호출하는 경우 "O" 또는 "o"와 같은 결과가 표시되지 않습니다. 사용자 지정 형식 문자열을 사용하는 구문 분석 메서드는 표준 시간대 구성 요소가 없는 날짜 및 시간 값의 문자열 표현을 구문 분석하거나 "Z"를 사용하여 UTC를 나타낼 수 없기 때문입니다.

다음 예제에서는 "o" 형식 지정자를 사용하여 미국 태평양 표준 시간대의 시스템에 일련의 DateTime 값과 DateTimeOffset 값을 표시합니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
       DateTime dat = new DateTime(2009, 6, 15, 13, 45, 30,
                                   DateTimeKind.Unspecified);
       Console.WriteLine($"{dat} ({dat.Kind}) --> {dat:O}");

       DateTime uDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Utc);
       Console.WriteLine($"{uDat} ({uDat.Kind}) --> {uDat:O}");

       DateTime lDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Local);
       Console.WriteLine($"{lDat} ({lDat.Kind}) --> {lDat:O}\n");

       DateTimeOffset dto = new DateTimeOffset(lDat);
       Console.WriteLine($"{dto} --> {dto:O}");
   }
}
// The example displays the following output:
//    6/15/2009 1:45:30 PM (Unspecified) --> 2009-06-15T13:45:30.0000000
```

```
//    6/15/2009 1:45:30 PM (Utc) --> 2009-06-15T13:45:30.0000000Z
//    6/15/2009 1:45:30 PM (Local) --> 2009-06-15T13:45:30.0000000-07:00
//
//    6/15/2009 1:45:30 PM -07:00 --> 2009-06-15T13:45:30.0000000-07:00
```

다음 예제에서는 "o" 형식 지정자를 사용하여 서식이 지정된 문자열을 만든 다음 날짜 및 시간 `Parse` 메서드를 호출하여 원래 날짜 및 시간 값을 복원합니다.

C#

```
// Round-trip DateTime values.
DateTime originalDate, newDate;
string dateString;
// Round-trip a local time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 10, 6, 30, 0),
DateTimeKind.Local);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine($"Round-tripped {originalDate} {originalDate.Kind} to {newDate}
{newDate.Kind}.");
// Round-trip a UTC time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 12, 9, 30, 0),
DateTimeKind.Utc);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine($"Round-tripped {originalDate} {originalDate.Kind} to {newDate}
{newDate.Kind}.");
// Round-trip time in an unspecified time zone.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 13, 12, 30, 0),
DateTimeKind.Unspecified);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine($"Round-tripped {originalDate} {originalDate.Kind} to {newDate}
{newDate.Kind}.");

// Round-trip a DateTimeOffset value.
DateTimeOffset originalDTO = new DateTimeOffset(2008, 4, 12, 9, 30, 0, new
TimeSpan(-8, 0, 0));
dateString = originalDTO.ToString("o");
DateTimeOffset newDTO = DateTimeOffset.Parse(dateString, null,
DateTimeStyles.RoundtripKind);
Console.WriteLine($"Round-tripped {originalDTO} to {newDTO}.");
// The example displays the following output:
//    Round-tripped 4/10/2008 6:30:00 AM Local to 4/10/2008 6:30:00 AM Local.
//    Round-tripped 4/12/2008 9:30:00 AM Utc to 4/12/2008 9:30:00 AM Utc.
//    Round-tripped 4/13/2008 12:30:00 PM Unspecified to 4/13/2008 12:30:00 PM
Unspecified.
//    Round-tripped 4/12/2008 9:30:00 AM -08:00 to 4/12/2008 9:30:00 AM -08:00.
```

표로 이동

RFC1123("R", "r") 서식 지정자

"R" 또는 "r" 표준 형식 지정자는 DateTimeFormatInfo.RFC1123Pattern 속성에 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 이 패턴은 정의된 표준을 반영하며 해당 속성은 읽기 전용입니다. 따라서 이 패턴은 사용된 문화권이나 제공된 서식 공급자에 관계없이 항상 같습니다. 사용자 지정 서식 문자열은 "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'"입니다. 이 표준 서식 지정자를 사용할 경우 서식 지정 또는 구문 분석 작업에서 항상 고정 문화권이 사용됩니다.

결과 문자열은 고정 문화권을 나타내는 DateTimeFormatInfo 속성에서 반환된 DateTimeFormatInfo.InvariantInfo 개체의 다음 속성에 의해 영향을 받습니다.

테이블 확장

재산	설명
RFC1123Pattern	결과 문자열의 형식을 정의합니다.
AbbreviatedDayNames	결과 문자열에 나타날 수 있는 축약된 일 이름을 정의합니다.
AbbreviatedMonthNames	결과 문자열에 나타날 수 있는 축약된 월 이름을 정의합니다.

RFC 1123 표준은 시간을 UTC(협정 세계시)로 표현하지만 서식 지정 작업은 서식이 지정된 DateTime 개체의 값을 수정하지 않습니다. 따라서 서식 지정 작업을 수행하기 전에 DateTime 메서드를 호출하여 DateTime.ToUniversalTime 값을 UTC로 변환해야 합니다. 반면, DateTimeOffset 값은 이 변환을 자동으로 수행합니다. 서식 지정 작업 전에 DateTimeOffset.ToUniversalTime 메서드를 호출할 필요가 없습니다.

다음 예제에서는 "r" 형식 지정자를 사용하여 미국 태평양 표준 시간대의 시스템에 DateTime 및 DateTimeOffset 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
DateTimeOffset dateOffset = new DateTimeOffset(date1,
                            TimeZoneInfo.Local.GetUtcOffset(date1));
Console.WriteLine(date1.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
Console.WriteLine(dateOffset.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
```

표로 이동

정렬 가능한("s") 서식 지정자

"s" 표준 형식 지정자는 DateTimeFormatInfo.SortableDateTimePattern 속성에 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 패턴은 정의된 표준(ISO 8601)을 반영하며 속성은 읽기 전용입니다. 따라서 이 패턴은 사용된 문화권이나 제공된 서식 공급자에 관계없이 항상 같습니다. 사용자 지정 서식 문자열은 "yyyy'-'MM'-'dd'T'HH':'mm':'ss"입니다.

"s" 형식 지정자는 날짜 및 시간 값에 따라 오름차순 또는 내림차순으로 일관되게 정렬되는 결과 문자열을 생성하기 위한 것입니다. 따라서 "s" 표준 서식 지정자는 날짜 및 시간 값을 일관된 형식으로 나타내지만 서식 지정 작업은 해당 DateTime.Kind 속성 또는 해당 DateTimeOffset.Offset 값을 반영하도록 서식이 지정되는 날짜 및 시간 개체의 값을 수정하지 않습니다. 예를 들어 날짜 및 시간 값 2014-11-15T18:32:17+00:00 및 2014-11-15T18:32:17+08:00의 서식을 지정하여 생성된 결과 문자열은 동일합니다.

이 표준 서식 지정자를 사용할 경우 서식 지정 또는 구문 분석 작업에서 항상 고정 문화권이 사용됩니다.

다음 예제에서는 "s" 형식 지정자를 사용하여 미국 태평양 표준 시간대의 시스템에 DateTime 및 DateTimeOffset 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("s"));
// Displays 2008-04-10T06:30:00
```

표로 이동

정렬 가능한 유니버설("u") 서식 지정자

"u" 표준 형식 지정자는 DateTimeFormatInfo.UniversalSortableDateTimePattern 속성으로 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 이 패턴은 정의된 표준을 반영하며 해당 속성은 읽기 전용입니다. 따라서 이 패턴은 사용된 문화권이나 제공된 서식 공급자에 관계없이 항상 같습니다. 사용자 지정 서식 문자열은 "yyyy'-'MM'-'dd HH':'mm':'ss'Z"입니다. 이 표준 서식 지정자를 사용할 경우 서식 지정 또는 구문 분석 작업에서 항상 고정 문화권이 사용됩니다.

결과 문자열은 시간을 UTC(협정 세계시)로 표현해야 하지만 서식 지정 작업 중에는 원래 DateTime 값의 변환이 수행되지 않습니다. 따라서 서식을 지정하기 전에 DateTime 메서드를 호출하여 DateTime.ToUniversalTime 값을 UTC로 변환해야 합니다. 반면, DateTimeOffset 값은 이 변환을 자동으로 수행합니다. 서식 지정 작업 전에 DateTimeOffset.ToUniversalTime 메서드를 호출할 필요가 없습니다.

다음 예제에서는 "u" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToUniversalTime().ToString("u"));
// Displays 2008-04-10 13:30:00Z
```

표로 이동

범용 전체("U") 형식 지정자

"U" 표준 형식 지정자는 지정된 문화권의 DateTimeFormatInfo.FullDateTimePattern 속성으로 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 패턴은 "F" 패턴과 동일합니다. 그러나 DateTime 값은 서식이 지정되기 전에 자동으로 UTC로 변환됩니다.

다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 FullDateTimePattern 속성에서 반환되는 사용자 지정 형식 지정자는 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
FullDateTimePattern	결과 문자열의 전체 형식을 정의합니다.
DayNames	결과 문자열에 나타날 수 있는 지역화된 날짜 이름을 정의합니다.
MonthNames	결과 문자열에 나타날 수 있는 지역화된 월 이름을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

"U" 형식 지정자는 DateTimeOffset 형식에서 지원되지 않으며 FormatException 값의 서식을 지정하는 데 사용되는 경우 DateTimeOffset throw합니다.

다음 예제에서는 "U" 형식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("U",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 1:30:00 PM
Console.WriteLine(date1.ToString("U",
                  CultureInfo.CreateSpecificCulture("sv-FI")));
// Displays den 10 april 2008 13:30:00
```

표로 이동

시간 형식

이 그룹에는 다음 서식이 포함됩니다.

- 간단한 시간("t") 서식 지정자
- 자세한 시간("T") 서식 지정자

간단한 시간("t") 서식 지정자

"t" 표준 형식 지정자는 현재 DateTimeFormatInfo.ShortTimePattern 속성에 의해 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어, 고정 문화권에 대한 사용자 지정 서식 문자열은 "HH:mm"입니다.

결과 문자열은 특정 DateTimeFormatInfo 개체의 서식 지정 정보의 영향을 받습니다. 다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 DateTimeFormatInfo.ShortTimePattern 속성에서 반환되는 사용자 지정 형식 지정자는 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
ShortTimePattern	결과 문자열의 시간 구성 요소 형식을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

다음 예제에서는 "t" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("t",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30 AM
Console.WriteLine(date1.ToString("t",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30
```

표로 이동

자세한 시간("T") 서식 지정자

"T" 표준 형식 지정자는 특정 문화권의 DateTimeFormatInfo.LongTimePattern 속성에 의해 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어, 고정 문화권에 대한 사용자 지정 서식 문자열은 "HH:mm:ss"입니다.

다음 표에서는 반환된 문자열의 서식을 제어할 수 있는 DateTimeFormatInfo 개체 속성을 나열합니다. 일부 문화권의 DateTimeFormatInfo.LongTimePattern 속성에서 반환되는 사용자 지정 형식 지정자는 일부 속성을 사용하지 않을 수 있습니다.

테이블 확장

재산	설명
LongTimePattern	결과 문자열의 시간 구성 요소 형식을 정의합니다.
TimeSeparator	시간의 시간, 분 및 두 번째 구성 요소를 구분하는 문자열을 정의합니다.
AMDesignator	12시간 시계에서 자정부터 정오 이전까지의 시간을 나타내는 문자열을 정의합니다.
PMDesignator	12시간 시계에서 정오부터 자정 이전까지의 시간을 나타내는 문자열을 정의합니다.

다음 예제에서는 "T" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("T",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30:00 AM
Console.WriteLine(date1.ToString("T",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30:00
```

표로 이동

부분적 날짜 서식

이 그룹에는 다음 서식이 포함됩니다.

- 월("M", "m") 서식 지정자
- 연도 월("Y", "y") 서식 지정자

월("M", "m") 서식 지정자

"M" 또는 "m" 표준 형식 지정자는 현재 DateTimeFormatInfo.MonthDayPattern 속성에 의해 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어, 고정 문화권에 대한 사용자 지정 서식 문자열은 "MMMM dd"입니다.

다음 표에서는 반환된 문자열의 서식을 제어하는 DateTimeFormatInfo 개체 속성을 나열합니다.

테이블 확장

재산	설명
MonthDayPattern	결과 문자열의 전체 형식을 정의합니다.
MonthNames	결과 문자열에 나타날 수 있는 지역화된 월 이름을 정의합니다.

다음 예제에서는 "m" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("m",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays April 10
Console.WriteLine(date1.ToString("m",
                  CultureInfo.CreateSpecificCulture("ms-MY")));
// Displays 10 April
```

표로 이동

연도 월("Y", "y") 서식 지정자

"Y" 또는 "y" 표준 형식 지정자는 지정된 문화권의 DateTimeFormatInfo.YearMonthPattern 속성에 의해 정의된 사용자 지정 날짜 및 시간 형식 문자열을 나타냅니다. 예를 들어, 고정 문화권에 대한 사용자 지정 서식 문자열은 "yyyy MMMM"입니다.

다음 표에서는 반환된 문자열의 서식을 제어하는 DateTimeFormatInfo 개체 속성을 나열합니다.

테이블 확장

재산	설명
YearMonthPattern	결과 문자열의 전체 형식을 정의합니다.
MonthNames	결과 문자열에 나타날 수 있는 지역화된 월 이름을 정의합니다.

다음 예제에서는 "y" 서식 지정자를 사용하여 날짜 및 시간 값을 표시합니다.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("Y",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays April, 2008
Console.WriteLine(date1.ToString("y",
                  CultureInfo.CreateSpecificCulture("af-ZA")));
// Displays April 2008
```

표로 이동

DateOnly 및 TimeOnly 서식 지정

DateOnly 및 TimeOnly 형식은 표준 날짜 및 시간 형식 문자열의 하위 집합을 지원합니다.

- **DateOnly** 는 날짜 관련 형식 지정자를 지원합니다.
  - "d"(짧은 날짜), "D"(긴 날짜)
  - "M" 또는 "m"(월/일)
  - "Y" 또는 "y"(연도/월)
  - "O" 또는 "o"(왕복, 날짜 부분만 해당)
  - "R" 또는 "r"(RFC1123, 날짜 부분만 해당)

- **TimeOnly** 는 시간 관련 형식 지정자를 지원합니다.
  - "t"(짧은 시간), "T"(긴 시간)
  - "O" 또는 "o"(왕복, 시간 부분만 해당)
  - "R" 또는 "r"(RFC1123, 시간 부분만 해당)

날짜 및 시간 정보를 결합하는 형식 지정자(예: "f", "F", "g", "G", "s", "u", "U")는 사용할 FormatException 때 throw DateOnly 합니다TimeOnly.

제어판 설정

Windows에서 제어판의 **국가 및 언어 옵션** 항목의 설정은 서식 지정 작업으로 생성된 결과 문자열에 영향을 줍니다. 이러한 설정은 서식을 제어하는 데 사용되는 값을 제공하는 현재 문화권과 연결된 DateTimeFormatInfo 개체를 초기화하는 데 사용됩니다. 다른 설정을 사용하는 컴퓨터는 다른 결과 문자열을 생성합니다.

또한 CultureInfo(String) 생성자를 사용하여 현재 시스템 문화권과 동일한 문화권을 나타내는 새 CultureInfo 개체를 인스턴스화하는 경우 제어판의 **국가 및 언어 옵션** 항목에 의해 설정된 모든 사용자 지정이 새 CultureInfo 개체에 적용됩니다. CultureInfo(String, Boolean) 생성자를 사용하여 시스템의 사용자 지정을 반영하지 않는 CultureInfo 개체를 만들 수 있습니다.

DateTimeFormatInfo 속성

서식 지정은 현재 문화권에서 암시적으로 제공되거나 서식을 호출하는 메서드의 DateTimeFormatInfo 매개 변수에 의해 명시적으로 제공되는 현재 IFormatProvider 개체의 속성에 의해 영향을 받습니다. IFormatProvider 매개 변수의 경우 애플리케이션은 문화권을 나타내는 CultureInfo 개체 또는 특정 문화권의 날짜 및 시간 서식 규칙을 나타내는 DateTimeFormatInfo 개체를 지정해야 합니다. 대부분의 표준 날짜 및 시간 형식 지정자는 현재 DateTimeFormatInfo 개체의 속성으로 정의된 서식 패턴의 별칭입니다. 애플리케이션은 해당 DateTimeFormatInfo 속성의 해당 날짜 및 시간 형식 패턴을 변경하여 일부 표준 날짜 및 시간 형식 지정자가 생성한 결과를 변경할 수 있습니다.

참고하십시오

- System.DateTime
- System.DateTimeOffset
- 형식 서식 지정
- 사용자 지정 날짜 및 시간 형식 문자열
- 샘플: .NET Core WinForms 서식 유틸리티(C#)
- 샘플: .NET Core WinForms 서식 유틸리티(Visual Basic)

Last updated on 2025. 12. 11.

사용자 지정 날짜 및 시간 서식 문자열

아티클 • 2023. 04. 08.

날짜 및 시간 형식 문자열은 형식 지정 작업에서 생성되는 DateTime 또는 DateTimeOffset 값의 텍스트 표현을 정의합니다. 또한 문자열을 날짜 및 시간으로 성공적으로 변환하기 위해 구문 분석 작업에 필요한 날짜 및 시간 값의 표현을 정의할 수 있습니다. 사용자 지정 형식 문자열은 하나 이상의 사용자 지정 날짜 및 시간 형식 지정자로 구성됩니다. 표준 날짜 및 시간 형식 문자열이 아닌 문자열은 사용자 지정 날짜 및 시간 형식 문자열로 해석됩니다.

> **팁**
>
> 서식 문자열을 숫자 또는 날짜 및 시간 값에 적용할 수 있도록 지원하고 결과 문자열을 표시하는 .NET Core Windows Forms 애플리케이션인 **서식 유틸리티**를 다운로드할 수 있습니다. **C#** 및 **Visual Basic**의 소스 코드를 사용할 수 있습니다.

사용자 지정 날짜 및 시간 형식 문자열은 DateTime과 DateTimeOffset 값 모두에 사용할 수 있습니다.

> **참고**
>
> 이 문서의 일부 C# 예제는 Try.NET 인라인 코드 실행기 및 플레이그라운드에서 실행됩니다. 대화형 창에서 예제를 실행하려면 **실행** 버튼을 선택합니다. 코드를 실행하면 **실행**을 다시 선택하여 코드를 수정하고 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나, 컴파일이 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.
>
> Try.NET 인라인 코드 러너와 플레이그라운드의 **현지 표준 시간대**는 협정 세계시(또는 UTC)입니다. 이는 DateTime, DateTimeOffset 및 TimeZoneInfo 형식과 이러한 형식의 멤버를 보여주는 예제의 동작 및 출력에 영향을 줄 수 있습니다.

형식 작업에서 사용자 지정 날짜 및 시간 형식 문자열은 날짜 및 시간 인스턴스의 `ToString` 메서드 또는 복합 형식을 지원하는 메서드에서 사용할 수 있습니다. 다음 예제에서는 두 가지 사용 방법을 모두 보여 줍니다.

C#

```
DateTime thisDate1 = new DateTime(2011, 6, 10);
Console.WriteLine("Today is " + thisDate1.ToString("MMMM dd, yyyy") + ".");

DateTimeOffset thisDate2 = new DateTimeOffset(2011, 6, 10, 15, 24, 16,
```

```
                                                TimeSpan.Zero);
Console.WriteLine("The current date and time: {0:MM/dd/yy H:mm:ss zzz}",
                  thisDate2);
// The example displays the following output:
//    Today is June 10, 2011.
//    The current date and time: 06/10/11 15:24:16 +00:00
```

구문 분석 작업에서 사용자 지정 날짜 및 시간 형식 문자열은 DateTime.ParseExact, DateTime.TryParseExact, DateTimeOffset.ParseExact 및 DateTimeOffset.TryParseExact 메서드에서 사용할 수 있습니다. 이러한 메서드를 사용하려면 구문 분석 작업에 성공하기 위해 입력 문자열이 특정 패턴을 정확하게 따라야 합니다. 다음 예제에서는 DateTimeOffset.ParseExact(String, String, IFormatProvider) 메서드를 호출하여 일, 월 및 두 자릿수 연도를 포함해야 하는 날짜를 구문 분석하는 방법을 보여 줍니다.

C#

```
using System;
using System.Globalization;

public class Example1
{
    public static void Main()
    {
        string[] dateValues = { "30-12-2011", "12-30-2011",
                                "30-12-11", "12-30-11" };
        string pattern = "MM-dd-yy";
        DateTime parsedDate;

        foreach (var dateValue in dateValues)
        {
            if (DateTime.TryParseExact(dateValue, pattern, null,
                                       DateTimeStyles.None, out parsedDate))
                Console.WriteLine("Converted '{0}' to {1:d}.",
                                  dateValue, parsedDate);
            else
                Console.WriteLine("Unable to convert '{0}' to a date and
time.",
                                  dateValue);
        }
    }
}
// The example displays the following output:
//    Unable to convert '30-12-2011' to a date and time.
//    Unable to convert '12-30-2011' to a date and time.
//    Unable to convert '30-12-11' to a date and time.
//    Converted '12-30-11' to 12/30/2011.
```

다음 표에서는 사용자 지정 날짜 및 시간 형식 지정자에 대해 설명하고 각 형식 지정자가 생성한 결과 문자열을 표시합니다. 기본적으로 결과 문자열은 en-US 문화권의 형식 규칙을 반영합니다. 특정 형식 지정자가 지역화된 결과 문자열을 만드는 동시에 결과 문자열

이 적용되는 문화도 명시합니다. 사용자 지정 날짜 및 시간 형식 문자열을 사용하는 방법에 대한 자세한 내용은 참고 섹션을 참조하세요.

형식 지정자	설명	예제
"d"	1부터 31까지의 일(월 기준)입니다. 추가 정보: "d" 사용자 지정 형식 지정자.	2009-06-01T13:45:30 -> 1 2009-06-15T13:45:30 -> 15
"dd"	01부터 31까지의 일(월 기준)입니다. 추가 정보: "dd" 사용자 지정 형식 지정자.	2009-06-01T13:45:30 -> 01 2009-06-15T13:45:30 -> 15
"ddd"	요일의 약식 이름입니다. 추가 정보: "ddd" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 월(en-US) 2009-06-15T13:45:30 -> Пн(ru-RU) 2009-06-15T13:45:30 -> lun. (fr-FR)
"dddd"	요일의 전체 이름입니다. 추가 정보: "dddd" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 월요일(en-US) 2009-06-15T13:45:30 -> понедельник(ru-RU) 2009-06-15T13:45:30 -> lundi(fr-FR)
"f"	날짜 및 시간 값에서 1/10초입니다. 추가 정보: "f" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.05 -> 0
"ff"	날짜 및 시간 값의 1/100초입니다. 추가 정보: "ff" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> 00
"fff"	날짜 및 시간 값의 1/1000초입니다. 추가 정보: "fff" 사용자 지정 형식 지정자.	2009년 6월 15일 13:45:30.617 -> 617 2009년 6월 15일 13:45:30.0005 -> 000
"ffff"	날짜 및 시간 값의 1/10000초입니다. 추가 정보: "ffff" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175000 -> 6175 2009-06-15T13:45:30.0000500 -> 0000
"fffff"	날짜 및 시간 값의 1/100000초입니다. 추가 정보: "fffff" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175400 -> 61754 2009년 6월 15일 13:45:30.000005 -> 00000

형식 지정자	설명	예제
"ffffff"	날짜 및 시간 값의 1/1000000초입니다. 추가 정보: "ffffff" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> 000000
"fffffff"	날짜 및 시간 값의 1/10000000초입니다. 추가 정보: "fffffff" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 0001150
"F"	0이 아닌 경우 날짜 및 시간 값의 1/10초입니다. 추가 정보: "F" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.0500000 -> (출력 없음)
"FF"	0이 아닌 경우 날짜 및 시간 값의 1/100초입니다. 추가 정보: "FF" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> (출력 없음)
"FFF"	0이 아닌 경우 날짜 및 시간 값의 1/1000초입니다. 추가 정보: “FFF” 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6170000 -> 617 2009-06-15T13:45:30.0005000 -> (출력 없음)
"FFFF"	0이 아닌 경우 날짜 및 시간 값의 1/10000초입니다. 추가 정보: "FFFF" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.5275000 -> 5275 2009-06-15T13:45:30.0000500 -> (출력 없음)
"FFFFF"	0이 아닌 경우 날짜 및 시간 값의 1/100000초입니다. 추가 정보: "FFFFF" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175400 -> 61754 2009-06-15T13:45:30.0000050 -> (출력 없음)
"FFFFFF"	0이 아닌 경우 날짜 및 시간 값의 1/1000000초입니다. 추가 정보: "FFFFFF" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> (출력 없음)

형식 지정자	설명	예제
"FFFFFFF"	0이 아닌 경우 날짜 및 시간 값의 1/10000000초입니다. 추가 정보: "FFFFFFF" 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 000115
"g", "gg"	서기 또는 연대입니다. 추가 정보: “g” 또는 “gg” 사용자 지정 형식 지정자.	2009-06-15T13:45:30.6170000 -> A.D.
"h"	12시간 형식을 사용하는 1부터 12까지의 시간입니다. 추가 정보: "h" 사용자 지정 형식 지정자.	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 1
"hh"	12시간 형식을 사용하는 01부터 12까지의 시간입니다. 추가 정보: "hh" 사용자 지정 형식 지정자.	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 01
"H"	24시간 형식을 사용하는 0부터 23까지의 시간입니다. 추가 정보: “H” 사용자 지정 형식 지정자.	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 13
"HH"	24시간 형식을 사용하는 00부터 23까지의 시간입니다. 추가 정보: “HH” 사용자 지정 형식 지정자.	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 13

형식 지정자	설명	예제
"K"	표준 시간대 정보입니다. 추가 정보: "K" 사용자 지정 형식 지정자.	DateTime 값과 함께 사용하는 경우 2009-06-15T13:45:30, 종류 지정되지 않음 -> 2009-06-15T13:45:30, 종류 Utc -> Z 2009-06-15T13:45:30, Kind Local -> -07:00(로컬 컴퓨터 설정에 따라 다름) DateTimeOffset 값과 함께 사용하는 경우 2009-06-15T01:45:30-07:00 --> -07:00 2009-06-15T08:45:30+00:00 --> +00:00
"m"	0부터 59까지의 분입니다. 추가 정보: "m" 사용자 지정 형식 지정자.	2009-06-15T01:09:30 -> 9 2009-06-15T13:29:30 -> 29
"mm"	00부터 59까지의 분입니다. 추가 정보: "mm" 사용자 지정 형식 지정자.	2009-06-15T01:09:30 -> 09 2009-06-15T01:45:30 -> 45
"M"	1부터 12까지의 월입니다. 추가 정보: "M" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 6
"MM"	01부터 12까지의 월입니다. 추가 정보: "MM" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 06
"MMM"	월의 약식 이름입니다. 추가 정보: "MMM" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 6월(en-US) 2009-06-15T13:45:30 -> juin(fr-FR) 2009-06-15T13:45:30 -> 6월(zu-ZA)
"MMMM"	월의 전체 이름입니다. 추가 정보: "MMMM" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 6월(en-US) 2009-06-15T13:45:30 -> juni(da-DK) 2009-06-15T13:45:30 -> uJuni(zu-ZA)

형식 지정자	설명	예제
"s"	0부터 59까지의 초입니다. 추가 정보: "s" 사용자 지정 형식 지정자.	2009-06-15T13:45:09 -> 9
"ss"	00부터 59까지의 초입니다. 추가 정보: "ss" 사용자 지정 형식 지정자.	2009-06-15T13:45:09 -> 09
"t"	AM/PM 지정자의 첫 문자입니다. 추가 정보: “t” 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> P(en-US) 2009-06-15T13:45:30 -> 午(ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
"tt"	AM/PM 지정자입니다. 추가 정보: “tt” 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> 오후(en-US) 2009-06-15T13:45:30 -> 午後(ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
"y"	0부터 99까지의 연도입니다. 추가 정보: “y” 사용자 지정 형식 지정자.	0001-01-01T00:00:00 -> 1 0900-01-01T00:00:00 -> 0 1900-01-01T00:00:00 -> 0 2009-06-15T13:45:30 -> 9 2019-06-15T13:45:30 -> 19
"yy"	00부터 99까지의 연도입니다. 추가 정보: “yy” 사용자 지정 형식 지정자.	0001-01-01T00:00:00 -> 01 0900-01-01T00:00:00 -> 00 1900-01-01T00:00:00 -> 00 2019-06-15T13:45:30 -> 19
"yyy"	최소 세 자리 숫자로 된 연도입니다. 추가 정보: “yyy” 사용자 지정 형식 지정자.	0001-01-01T00:00:00 -> 001 0900-01-01T00:00:00 -> 900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009

형식 지정자	설명	예제
"yyyy"	네 자리 숫자로 된 연도입니다. 추가 정보: "yyyy" 사용자 지정 형식 지정자.	0001-01-01T00:00:00 -> 0001 0900-01-01T00:00:00 -> 0900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
"yyyyy"	다섯 자리 숫자로 된 연도입니다. 추가 정보: "yyyyy" 사용자 지정 형식 지정자.	0001-01-01T00:00:00 -> 00001 2009-06-15T13:45:30 -> 02009
"z"	앞에 0이 표시되지 않는 UTC에서의 시간 오프셋입니다. 추가 정보: "z" 사용자 지정 형식 지정자.	2009-06-15T13:45:30-07:00 -> -7
"zz"	한 자리 값의 경우 앞에 0이 표시되는 UTC에서의 시간 오프셋입니다. 추가 정보: "zz" 사용자 지정 형식 지정자.	2009-06-15T13:45:30-07:00 -> -07
"zzz"	UTC에서의 시간 및 분 오프셋입니다. 추가 정보: "zzz" 사용자 지정 형식 지정자.	2009-06-15T13:45:30-07:00 -> -07:00
":"	시간 구분 기호입니다. 추가 정보: ":" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> : (en-US) 2009-06-15T13:45:30 -> . (it-IT) 2009-06-15T13:45:30 -> : (ja-JP)
"/"	날짜 구분 기호입니다. 추가 정보: "/" 사용자 지정 형식 지정자.	2009-06-15T13:45:30 -> /(en-US) 2009-06-15T13:45:30 -> -(ar-DZ) 2009-06-15T13:45:30 -> . (tr-TR)
"*string*" '*string*'	리터럴 문자열 구분 기호입니다. 추가 정보: 문자 리터럴.	2009-06-15T13:45:30 ("arr:" h:m t) -> arr: 1:45 P 2009-06-15T13:45:30 ('arr:' h:m t) -> arr: 1:45 P

형식 지정자	설명	예제
%	뒤에 오는 문자를 사용자 지정 형식 지정자로 정의합니다. 추가 정보: 단일 사용자 지정 형식 지정자 사용	2009-06-15T13:45:30 (%h) -> 1
\	이스케이프 문자입니다. 추가 정보: 문자 리터럴 및 이스케이프 문자 사용.	2009-06-15T13:45:30 (h \h) -> 1 h
기타 문자	문자가 변경되지 않은 상태로 결과 문자열에 복사됩니다. 추가 정보: 문자 리터럴.	2009-06-15T01:45:30 (arr hh:mm t) -> arr 01:45 A

다음 단원에서는 각 사용자 지정 날짜 및 시간 형식 지정자에 대한 추가 정보를 제공합니다. 다른 설명이 없는한, 각 지정자는 DateTime 값이나 DateTimeOffset 값에 상관없이 동일한 문자열을 생성합니다.

일 "d" 형식 지정자

"d" 사용자 지정 형식 지정자

"d" 사용자 지정 형식 지정자는 일(월 기준)을 1부터 31까지의 숫자로 나타냅니다. 한 자리 일의 경우 앞에 0이 표시되지 않습니다.

다른 사용자 지정 형식 지정자 없이 "d" 형식 지정자만 사용되면 "d" 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 여러 개의 형식 문자열에 "d" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("d, M",
                  CultureInfo.InvariantCulture));
// Displays 29, 8

Console.WriteLine(date1.ToString("d MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays 29 August
```

```
Console.WriteLine(date1.ToString("d MMMM",
                  CultureInfo.CreateSpecificCulture("es-MX")));
// Displays 29 agosto
```

표로 이동

"dd" 사용자 지정 형식 지정자

"dd" 사용자 지정 형식 문자열은 일(월 기준)을 01부터 31까지의 숫자로 나타냅니다. 한 자리 일의 경우 앞에 0이 표시됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "dd" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
                  CultureInfo.InvariantCulture));
// 02, 01
```

표로 이동

"ddd" 사용자 지정 형식 지정자

"ddd" 사용자 지정 형식 지정자는 요일의 약식 이름을 나타냅니다. 요일의 지역화된 약식 이름은 현재 또는 지정된 문화권의 DateTimeFormatInfo.AbbreviatedDayNames 속성에서 검색됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "ddd" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

표로 이동

"dddd" 사용자 지정 형식 지정자

"dddd" 사용자 지정 형식 지정자는 임의 개수의 추가 "d" 지정자와 함께 요일의 전체 이름을 나타냅니다. 지역화된 요일 이름은 현재 또는 지정된 문화권의 DateTimeFormatInfo.DayNames 속성에서 검색됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "dddd" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

표로 이동

소문자 초 “f” 분수 지정자

"f" 사용자 지정 형식 지정자

"f" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수를 나타냅니다. 즉, 날짜 및 시간 값에서 1/10초까지 표시합니다.

다른 형식 지정자 없이 “f” 형식 지정자만 사용되면 “f” 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

"f" 형식 지정자를 ParseExact, TryParseExact, ParseExact 또는 TryParseExact 메서드에 형식 문자열의 일부로 제공하여 사용할 경우, 사용하는 "f" 형식 지정자의 수는 문자열을 구문 분석하는 데 필요한 초의 소수 부분에 대한 최대 유효 자릿수를 나타냅니다.

다음 예제에서는 사용자 지정 형식 문자열에 "f" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
```

```
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

표로 이동

"ff" 사용자 지정 형식 지정자

"ff" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 2개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/100초까지 표시합니다.

다음 예제에서는 사용자 지정 형식 문자열에 "ff" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

표로 이동

"fff" 사용자 지정 형식 지정자

"fff" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 3개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/1000초까지 표시합니다.

다음 예제에서는 사용자 지정 형식 문자열에 "fff" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

표로 이동

"ffff" 사용자 지정 형식 지정자

"ffff" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 4개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/10000초까지 표시합니다.

1/10000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15 밀리초입니다.

표로 이동

"fffff" 사용자 지정 형식 지정자

"fffff" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 다섯 개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/100000초까지 표시합니다.

1/100000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15 밀리초입니다.

표로 이동

"ffffff" 사용자 지정 형식 지정자

"ffffff" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 6개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/1000000초까지 표시합니다.

1/1000000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15밀리초입니다.

표로 이동

"fffffff" 사용자 지정 형식 지정자

"fffffff" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 7개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/10000000초까지 표시합니다.

1/10000000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15밀리초입니다.

표로 이동

대문자 초 “F” 분수 지정자

"F" 사용자 지정 형식 지정자

"F" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수를 나타냅니다. 즉, 날짜 및 시간 값에서 1/10초까지 표시합니다. 숫자가 0이고 초 수를 따르는 소수점도 표시되지 않으면 아무 것도 표시되지 않습니다.

다른 형식 지정자 없이 “F” 형식 지정자만 사용되면 “F” 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

ParseExact, TryParseExact, ParseExact 또는 TryParseExact 메서드와 함께 F 형식 지정자를 사용할 경우, 사용하는 "F" 형식 지정자의 수는 문자열을 구문 분석하는 데 사용할 수 있는 시간(초)의 최대 유효 자릿수에 대한 최대 개수를 나타냅니다.

다음 예제에서는 사용자 지정 형식 문자열에 "F" 사용자 지정 형식 지정자를 포함합니다.

```
C#
```

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

표로 이동

"FF" 사용자 지정 형식 지정자

"FF" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 2개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/100초까지 표시합니다. 후행 0은 표시되지 않습니다. 두 개의 유효 자릿수가 0이면 아무 것도 표시되지 않으며, 이 경우 초 수를 따르는 소수점도 표시되지 않습니다.

다음 예제에서는 사용자 지정 형식 문자열에 "FF" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

표로 이동

"FFF" 사용자 지정 형식 지정자

"FFF" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 3개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/1000초까지 표시합니다. 후행 0은 표시되지 않습니다. 세 개의 유효 자릿수가 0이면 아무 것도 표시되지 않으며, 이 경우 초 수를 따르는 소수점도 표시되지 않습니다.

다음 예제에서는 사용자 지정 형식 문자열에 "FFF" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

표로 이동

"FFFF" 사용자 지정 형식 지정자

"FFFF" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 4개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/10000초까지 표시합니다. 후행 0은 표시되지 않습니다. 4개의 유효 자릿수가 0이면 아무 것도 표시되지 않으며, 이 경우 초 수를 따르는 소수점도 표시되지 않습니다.

1/10000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15 밀리초입니다.

표로 이동

"FFFFF" 사용자 지정 형식 지정자

"FFFFF" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 5개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/100000초까지 표시합니다. 후행 0은 표시되지 않습니다. 5개의 유효 자릿수가 0이면 아무 것도 표시되지 않으며, 이 경우 초 수를 따르는 소수점도 표시되지 않습니다.

1/100000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15밀리초입니다.

표로 이동

"FFFFFF" 사용자 지정 형식 지정자

"FFFFFF" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 6개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/1000000초까지 표시합니다. 후행 0은 표시되지 않습니다. 6개의 유효 자릿수가 0이면 아무 것도 표시되지 않으며, 이 경우 초 수를 따르는 소수점도 표시되지 않습니다.

1/1000000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15밀리초입니다.

표로 이동

"FFFFFFF" 사용자 지정 형식 지정자

"FFFFFFF" 사용자 지정 형식 지정자는 초의 소수 부분에 대한 최대 유효 자릿수 7개를 나타냅니다. 즉, 날짜 및 시간 값에서 1/10000000초까지 표시합니다. 후행 0은 표시되지 않습니다. 7개의 유효 자릿수가 0이면 아무 것도 표시되지 않으며, 이 경우 초 수를 따르는 소수점도 표시되지 않습니다.

1/10000000초의 시간 값 구성 요소를 표시하는 것이 가능하기는 하지만 이 값은 의미가 없을 수도 있습니다. 날짜 및 시간 값의 자릿수는 시스템 시계의 정밀도에 따라 달라집니다. Windows NT 3.5 이상 버전과 Windows Vista 운영 체제의 경우 시계의 정밀도는 약 10-15밀리초입니다.

표로 이동

연대 “g” 형식 지정자

"g" 또는 "gg" 사용자 지정 형식 지정자

"g" 또는 "gg" 사용자 지정 형식 지정자(추가 "g" 지정자 수 포함)는 A.D와 같은 기간 또는 연대를 나타냅니다. 서식을 지정할 날짜에 연결된 기간 또는 연대 문자열이 없는 경우 서식 지정 작업은 이 지정자를 무시합니다.

다른 사용자 지정 형식 지정자 없이 "g" 형식 지정자만 사용되면 "g" 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "g" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(70, 08, 04);

Console.WriteLine(date1.ToString("MM/dd/yyyy g",
                  CultureInfo.InvariantCulture));
// Displays 08/04/0070 A.D.
Console.WriteLine(date1.ToString("MM/dd/yyyy g",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 08/04/0070 ap. J.-C.
```

표로 이동

소문자 시간 "h" 형식 지정자

"h" 사용자 지정 형식 지정자

"h" 사용자 지정 형식 지정자는 시간을 1부터 12까지의 숫자로 나타냅니다. 즉, 자정 또는 정오 이후의 총 시간을 계산하는 12시간 형식으로 나타냅니다. 자정 이후의 시간과 정오 이후의 같은 시간을 구별할 수 없습니다. 시간은 반올림되지 않으며 한 자리 시간의 경우 앞에 0이 표시되지 않습니다. 예를 들어, 시간이 오전 또는 오후 5:43일 경우 이 사용자 지정 형식 지정자는 "5"를 표시합니다.

다른 사용자 지정 형식 지정자 없이 "h" 형식 지정자만 사용되면 표준 날짜 및 시간 형식 지정자로 해석되고 FormatException이 나타납니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "h" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

표로 이동

"hh" 사용자 지정 형식 지정자

"hh" 사용자 지정 형식 지정자는 임의 개수의 추가 "h" 지정자와 함께 시간을 1부터 12까지의 숫자로 나타냅니다. 즉, 자정 또는 정오 이후의 총 시간을 계산하는 12시간 형식으로 나타냅니다. 자정 이후의 시간과 정오 이후의 같은 시간을 구별할 수 없습니다. 시간은 반올림되지 않으며 한 자리 시간의 경우 앞에 0이 표시됩니다. 예를 들어, 시간이 오전 또는 오후 5:43일 경우 이 형식 지정자는 "05"를 표시합니다.

다음 예제에서는 사용자 지정 형식 문자열에 "hh" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

표로 이동

대문자 시간 “H” 형식 지정자

"H" 사용자 지정 형식 지정자

"H" 사용자 지정 형식 지정자는 시간을 0부터 23까지의 숫자로 나타냅니다. 즉, 자정 이후의 시간을 계산하는 24시간(0부터 시작) 형식으로 나타냅니다. 한 자리 시간의 경우 앞에 0이 표시되지 않습니다.

다른 사용자 지정 형식 지정자 없이 “H” 형식 지정자만 사용되면 표준 날짜 및 시간 형식 지정자로 해석되고 FormatException이 throw됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "H" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("H:mm:ss",
                  CultureInfo.InvariantCulture));
// Displays 6:09:01
```

표로 이동

"HH" 사용자 지정 형식 지정자

"HH" 사용자 지정 형식 지정자는 임의 개수의 추가 "H" 지정자와 함께 시간을 00부터 23까지의 숫자로 나타냅니다. 즉, 자정 이후의 시간을 계산하는 24시간(0부터 시작) 형식으로 나타냅니다. 한 자리 시간의 경우 앞에 0이 표시됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "HH" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("HH:mm:ss",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01
```

표로 이동

표준 시간대 “K” 형식 지정자

"K" 사용자 지정 형식 지정자

"K" 사용자 지정 형식 지정자는 날짜 및 시간 값의 표준 시간대 정보를 나타냅니다. 이 형식 지정자를 DateTime 값과 함께 사용할 경우 결과 문자열은 DateTime.Kind 속성 값에 의해 정의됩니다.

- 현지 표준 시간대(DateTime.Kind 의 속성 값)의 DateTimeKind.Local경우 이 지정자는 UTC(협정 세계시)의 로컬 오프셋을 포함하는 결과 문자열을 생성합니다(예: "-07:00").
- UTC 시간(DateTime.Kind 속성 값이 DateTimeKind.Utc임)의 경우 결과 문자열에 UTC 날짜를 나타내는 "Z" 문자가 포함됩니다.
- 지정되지 않은 표준 시간대(시간의 DateTime.Kind 속성이 DateTimeKind.Unspecified임)의 경우 결과가 String.Empty와 같습니다.

DateTimeOffset 값의 경우 "K" 형식 지정자는 "zzz" 형식 지정자와 같으며 UTC에서의 DateTimeOffset 값 오프셋이 포함된 결과 문자열을 생성합니다.

다른 사용자 지정 형식 지정자 없이 "K" 형식 지정자만 사용되면 표준 날짜 및 시간 형식 지정자로 해석되고 FormatException이 throw됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 미국 태평양 표준 시간대에 있는 시스템의 다양한 DateTime 및 DateTimeOffset 값에 "K" 사용자 지정 형식 지정자를 사용하여 생성된 문자열을 표시합니다.

C#

```
Console.WriteLine(DateTime.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTime.UtcNow.ToString("%K"));
// Displays Z
Console.WriteLine("'{0}'",
                  DateTime.SpecifyKind(DateTime.Now,
                       DateTimeKind.Unspecified).ToString("%K"));
// Displays ''
Console.WriteLine(DateTimeOffset.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTimeOffset.UtcNow.ToString("%K"));
// Displays +00:00
Console.WriteLine(new DateTimeOffset(2008, 5, 1, 6, 30, 0,
                      new TimeSpan(5, 0, 0)).ToString("%K"));
// Displays +05:00
```

표로 이동

분 “m” 형식 지정자

"m" 사용자 지정 형식 지정자

"m" 사용자 지정 형식 지정자는 분을 0부터 59까지의 숫자로 나타냅니다. 분은 마지막 시간 이후 경과한 총 분 수를 나타냅니다. 한 자리 분의 경우 앞에 0이 표시되지 않습니다.

다른 사용자 지정 형식 지정자 없이 “m” 형식 지정자만 사용되면 “m” 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "m" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

표로 이동

"mm" 사용자 지정 형식 지정자

"mm" 사용자 지정 형식 지정자는 임의 개수의 추가 "m" 지정자와 함께 분을 00부터 59까지의 숫자로 나타냅니다. 분은 마지막 시간 이후 경과한 총 분 수를 나타냅니다. 한 자리 분의 경우 앞에 0이 표시됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "mm" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
```

```
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

표로 이동

월 “M” 형식 지정자

"M" 사용자 지정 형식 지정자

"M" 사용자 지정 형식 지정자는 월을 1부터 12까지의 숫자(또는 13월까지 있는 역법의 경우 1부터 13까지의 숫자)로 표현합니다. 한 자리 월의 경우 앞에 0이 표시되지 않습니다.

다른 사용자 지정 형식 지정자 없이 “M” 형식 지정자만 사용되면 “M” 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "M" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 18);
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays (8) Aug, August
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("nl-NL")));
// Displays (8) aug, augustus
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("lv-LV")));
// Displays (8) Aug, augusts
```

표로 이동

"MM" 사용자 지정 형식 지정자

"MM" 사용자 지정 형식 지정자는 월을 01부터 12까지의 숫자(또는 13월까지 있는 역법의 경우 01부터 13까지의 숫자)로 표현합니다. 한 자리 월의 경우 앞에 0이 표시됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "MM" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
                  CultureInfo.InvariantCulture));
// 02, 01
```

표로 이동

"MMM" 사용자 지정 형식 지정자

"MMM" 사용자 지정 형식 지정자는 월의 약식 이름을 나타냅니다. 월의 지역화된 약식 이름은 현재 또는 지정된 문화권의 DateTimeFormatInfo.AbbreviatedMonthNames 속성에서 검색됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "MMM" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

표로 이동

"MMMM" 사용자 지정 형식 지정자

"MMMM" 사용자 지정 형식 지정자는 월의 전체 이름을 나타냅니다. 월의 지역화된 이름은 현재 또는 지정된 문화권의 DateTimeFormatInfo.MonthNames 속성에서 검색됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "MMMM" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

표로 이동

초 "s" 형식 지정자

"s" 사용자 지정 형식 지정자

"s" 사용자 지정 형식 지정자는 초를 0부터 59까지의 숫자로 나타냅니다. 결과는 마지막 분 이후 경과한 총 초 수를 나타냅니다. 한 자리 초의 경우 앞에 0이 표시되지 않습니다.

다른 사용자 지정 형식 지정자 없이 "s" 형식 지정자만 사용되면 "s" 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "s" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

표로 이동

"ss" 사용자 지정 형식 지정자

"ss" 사용자 지정 형식 지정자는 임의 개수의 추가 "s" 지정자와 함께 초를 00부터 59까지의 숫자로 나타냅니다. 결과는 마지막 분 이후 경과한 총 초 수를 나타냅니다. 한 자리 초의 경우 앞에 0이 표시됩니다.

다음 예제에서는 사용자 지정 형식 문자열에 "ss" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

표로 이동

오전/오후 “t” 형식 지정자

"t" 사용자 지정 형식 지정자

"t" 사용자 지정 형식 지정자는 AM/PM 지정자의 첫 문자를 나타냅니다. 적절한 지역화된 지정자는 현재 또는 지정된 문화권의 DateTimeFormatInfo.AMDesignator 또는 DateTimeFormatInfo.PMDesignator 속성에서 검색됩니다. AM 지정자는 0:00:00(자정)부터 11:59:59.999까지의 모든 시간에 사용되고 PM 지정자는 12:00:00(정오)부터 23:59:59.999까지의 모든 시간에 사용됩니다.

다른 사용자 지정 형식 지정자 없이 “t” 형식 지정자만 사용되면 “t” 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "t" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

표로 이동

"tt" 사용자 지정 형식 지정자

"tt" 사용자 지정 형식 지정자는 임의 개수의 추가 "t" 지정자와 함께 전체 AM/PM 지정자를 나타냅니다. 적절한 지역화된 지정자는 현재 또는 지정된 문화권의 DateTimeFormatInfo.AMDesignator 또는 DateTimeFormatInfo.PMDesignator 속성에서 검색됩니다. AM 지정자는 0:00:00(자정)부터 11:59:59.999까지의 모든 시간에 사용되고 PM 지정자는 12:00:00(정오)부터 23:59:59.999까지의 모든 시간에 사용됩니다.

“tt” 지정자는 AM과 PM을 구분해야 하는 언어에만 사용해야 합니다. 예를 들어, 일본어 AM/PM 지정자의 경우 첫 번째 문자가 아니라 두 번째 문자가 서로 다릅니다.

다음 예제에서는 사용자 지정 형식 문자열에 "tt" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

표로 이동

연도 “y” 형식 지정자

"y" 사용자 지정 형식 지정자

"y" 사용자 지정 형식 지정자는 연도를 한 자리 또는 두 자리 숫자로 나타냅니다. 연도가 두 자리를 넘으면 마지막 두 자리 숫자만 결과에 나타납니다. 2008과 같이 두 자리 연도의 첫 번째 숫자가 0으로 시작하면 앞에 0이 표시되지 않습니다.

다른 사용자 지정 형식 지정자 없이 “y” 형식 지정자만 사용되면 “y” 표준 날짜 및 시간 형식 지정자로 해석됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "y" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

표로 이동

"yy" 사용자 지정 형식 지정자

"yy" 사용자 지정 형식 지정자는 연도를 두 자리 숫자로 나타냅니다. 연도가 두 자리를 넘으면 마지막 두 자리 숫자만 결과에 나타납니다. 두 자리 연도의 유효 자릿수가 두 자리 미만인 경우 두 자리가 되도록 앞에 0이 채워집니다.

구문 분석 작업에서 "yy" 사용자 지정 형식 지정자를 사용하여 구문 분석되는 2자리 년도는 형식 공급자의 현재 달력의 Calendar.TwoDigitYearMax 속성을 기준으로 해석됩니다. 다음 예제에서는 en-US 문화권(이 경우 현재 문화권)의 기본 양력을 사용하여 두 자리 년도를 갖는 날짜의 문자열 표현을 구문 분석합니다. 그런 다음 수정된 TwoDigitYearMax를 쓰는 GregorianCalendar를 사용하도록 현재문화권의 CultureInfo 개체를 변경 합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example7
{
    public static void Main()
    {
        string fmt = "dd-MMM-yy";
        string value = "24-Jan-49";

        Calendar cal =
(Calendar)CultureInfo.CurrentCulture.Calendar.Clone();
        Console.WriteLine("Two Digit Year Range: {0} - {1}",
                          cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);

        Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
        Console.WriteLine();

        cal.TwoDigitYearMax = 2099;
        CultureInfo culture =
(CultureInfo)CultureInfo.CurrentCulture.Clone();
        culture.DateTimeFormat.Calendar = cal;
        Thread.CurrentThread.CurrentCulture = culture;

        Console.WriteLine("Two Digit Year Range: {0} - {1}",
                          cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);
        Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
    }
}
// The example displays the following output:
//       Two Digit Year Range: 1930 - 2029
//       1/24/1949
//
//       Two Digit Year Range: 2000 - 2099
//       1/24/2049
```

다음 예제에서는 사용자 지정 형식 문자열에 "yy" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

표로 이동

"yyy" 사용자 지정 형식 지정자

"yyy" 사용자 지정 형식 지정자는 연도를 최소 세 자리 숫자로 나타냅니다. 연도의 유효 자릿수가 세 자리보다 많더라도 결과 문자열에 포함됩니다. 연도가 세 자리 미만인 경우 세 자리가 되도록 앞에 0이 채워집니다.

> **참고**
>
> 연도를 다섯 자리까지 표시할 수 있는 태국 불교식 달력의 경우 이 형식 지정자는 유효 자릿수를 모두 표시합니다.

다음 예제에서는 사용자 지정 형식 문자열에 "yyy" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
```

```
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

표로 이동

"yyyy" 사용자 지정 형식 지정자

"yyyy" 사용자 지정 형식 지정자는 연도를 최소 네 자리 숫자로 나타냅니다. 연도의 유효 자릿수가 네 자리보다 많더라도 결과 문자열에 포함됩니다. 연도가 네 자리 미만인 경우 네 자리가 되도록 앞에 0이 채워집니다.

> ⓘ **참고**
>
> 연도를 다섯 자리까지 표시할 수 있는 태국 불교식 달력의 경우 이 형식 지정자는 최소 네 자리 숫자를 표시합니다.

다음 예제에서는 사용자 지정 형식 문자열에 "yyyy" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
```

```
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

표로 이동

"yyyyy" 사용자 지정 형식 지정자

"yyyyy" 사용자 지정 형식 지정자는 임의 개수의 추가 "y" 지정자와 함께 연도를 최소 다섯 자리 숫자로 나타냅니다. 연도의 유효 자릿수가 다섯 자리보다 많더라도 결과 문자열에 포함됩니다. 연도가 다섯 자리 미만인 경우 다섯 자리가 되도록 앞에 0이 채워집니다.

추가 "y" 지정자가 있는 경우 "y" 지정자의 수에 맞도록 앞에 0이 필요한 만큼 채워집니다.

다음 예제에서는 사용자 지정 형식 문자열에 "yyyyy" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

표로 이동

오프셋 "z" 형식 지정자

"z" 사용자 지정 형식 지정자

값을 사용하여 DateTime "z" 사용자 지정 형식 지정자는 시간 단위로 측정된 UTC(협정 세계시)에서 지정된 표준 시간대의 부호 있는 오프셋을 나타냅니다. 오프셋은 항상 앞에 부호가 표시됩니다. 더하기 기호(+)는 UTC보다 앞선 시간을 나타내고 빼기 기호(-)는 UTC보다 늦은 시간을 나타냅니다. 한 자리 오프셋은 선행 *0 없이* 서식이 지정됩니다.

다음 표에서는 에 따라 오프셋 값이 어떻게 변경되는지 보여줍니다 DateTimeKind.

DateTimeKind 값	**오프셋 값**
Local	UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.
Unspecified	UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.
Utc	`+0` .NET Core 및 .NET 5 이상에서 .NET Framework UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.

DateTimeOffset 값에 사용할 경우 이 형식 지정자는 시간 단위로 측정된 UTC에서의 DateTimeOffset 값 오프셋을 나타냅니다.

다른 사용자 지정 형식 지정자 없이 "z" 형식 지정자만 사용되면 표준 날짜 및 시간 형식 지정자로 해석되고 FormatException이 throw됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

다음 예제에서는 사용자 지정 형식 문자열에 "z" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00 on .NET Framework
// Displays +0, +00, +00:00 on .NET Core and .NET 5+

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
```

```
                    date2));
// Displays +6, +06, +06:00
```

표로 이동

"zz" 사용자 지정 형식 지정자

값을 사용하여 DateTime "zz" 사용자 지정 형식 지정자는 UTC에서 지정된 표준 시간대의 부속 오프셋을 나타내며 시간 단위로 측정됩니다. 오프셋은 항상 앞에 부호가 표시됩니다. 더하기 기호(+)는 UTC보다 앞선 시간을 나타내고 빼기 기호(-)는 UTC보다 늦은 시간을 나타냅니다. 한 자리 오프셋은 선행 0 으로 서식이 지정됩니다.

다음 표에서는 에 따라 오프셋 값이 어떻게 변경되는지 보여줍니다 DateTimeKind.

DateTimeKind 값	**오프셋 값**
Local	UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.
Unspecified	UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.
Utc	`+00` .NET Core 및 .NET 5 이상에서 .NET Framework UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.

DateTimeOffset 값에 사용할 경우 이 형식 지정자는 시간 단위로 측정된 UTC에서의 DateTimeOffset 값 오프셋을 나타냅니다.

다음 예제에서는 사용자 지정 형식 문자열에 "zz" 사용자 지정 형식 지정자를 포함합니다.

C#

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00 on .NET Framework
// Displays +0, +00, +00:00 on .NET Core and .NET 5+

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

표로 이동

"zzz" 사용자 지정 형식 지정자

값을 사용하여 DateTime "zzz" 사용자 지정 형식 지정자는 UTC에서 지정된 표준 시간대의 부호 있는 오프셋을 나타내며 시간 및 분 단위로 측정됩니다. 오프셋은 항상 앞에 부호가 표시됩니다. 더하기 기호(+)는 UTC보다 앞선 시간을 나타내고 빼기 기호(-)는 UTC보다 늦은 시간을 나타냅니다. 한 자리 오프셋의 경우 앞에 0이 표시됩니다.

다음 표에서는 에 따라 오프셋 값이 어떻게 변경되는지 보여줍니다 DateTimeKind.

DateTimeKind 값	**오프셋 값**
Local	UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.
Unspecified	UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.
Utc	`+00:00` .NET Core 및 .NET 5 이상에서 .NET Framework UTC에서 로컬 운영 체제 표준 시간대의 서명된 오프셋입니다.

DateTimeOffset 값에 사용할 경우 이 형식 지정자는 시간 및 분 단위로 측정된 UTC에서의 DateTimeOffset 값 오프셋을 나타냅니다.

다음 예제에서는 사용자 지정 형식 문자열에 "zzz" 사용자 지정 형식 지정자를 포함합니다.

```
C#
```

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00 on .NET Framework
// Displays +0, +00, +00:00 on .NET Core and .NET 5+

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

표로 이동

날짜 및 시간 구분 기호 지정자

":" 사용자 지정 형식 지정자

":" 사용자 지정 형식 지정자는 시, 분, 초를 구분하는 데 사용되는 시간 구분 기호를 나타냅니다. 적절한 지역화된 시간 구분 기호는 현재 또는 지정된 문화권의 DateTimeFormatInfo.TimeSeparator 속성에서 검색됩니다.

> **참고**
>
> 특정 날짜 및 시간 문자열의 시간 구분 기호를 변경하려면 리터럴 문자열 구분 기호 내에서 구분 기호 문자를 지정합니다. 예를 들어 사용자 지정 형식 문자열 `hh'_'dd'_'ss`는 "_"(밑줄)이 항상 시간 구분 기호로 사용되는 결과 문자열을 생성합니다. 특정 문화권의 모든 날짜에 대한 시간 구분 기호를 변경하려면 현재 문화권의 DateTimeFormatInfo.TimeSeparator 속성 값을 변경하거나 DateTimeFormatInfo 개체를 인스턴스화하고 해당 TimeSeparator 속성에 문자를 할당한 다음 IFormatProvider 매개 변수를 포함하는 서식 지정 메서드의 오버로드를 호출합니다.

다른 사용자 지정 형식 지정자 없이 ":" 형식 지정자만 사용되면 표준 날짜 및 시간 형식 지정자로 해석되고 FormatException이 throw됩니다. 단일 형식 지정자를 사용하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

표로 이동

"/" 사용자 지정 형식 지정자

"/" 사용자 지정 형식 지정자는 년, 월, 일을 구분하는 데 사용되는 날짜 구분 기호를 나타냅니다. 적절한 지역화된 날짜 구분 기호는 현재 또는 지정된 문화권의 DateTimeFormatInfo.DateSeparator 속성에서 검색됩니다.

> **참고**
>
> 특정 날짜 및 시간 문자열의 날짜 구분 기호를 변경하려면 리터럴 문자열 구분 기호 내에서 구분 기호 문자를 지정합니다. 예를 들어 사용자 지정 형식 문자열 `mm'/'dd'/'yyyy`는 "/"를 항상 날짜 구분 기호로 사용하는 결과 문자열을 생성합니다. 특정 문화권의 모든 날짜에 대한 날짜 구분 기호를 변경하려면 현재 문화권의 DateTimeFormatInfo.DateSeparator 속성 값을 변경하거나 DateTimeFormatInfo 개체를 인스턴스화하고 해당 DateSeparator 속성에 문자를 할당한 다음 IFormatProvider 매개 변수를 포함하는 서식 지정 메서드의 오버로드를 호출합니다.

다른 사용자 지정 형식 지정자 없이 "/" 형식 지정자만 사용되면 표준 날짜 및 시간 형식 지정자로 해석되고 FormatException이 throw됩니다. 단일 형식 지정자를 사용하는 방법

에 대한 자세한 내용은 이 문서의 뒷부분에 있는 단일 사용자 지정 형식 지정자 사용을 참조하세요.

표로 이동

문자 리터럴

사용자 지정 날짜 및 시간 형식 문자열에서 다음 문자는 예약되었으며 항상 형식 지정 문자 또는 `"`, `'`, `/` 및 `\`의 경우 특수 문자로 해석됩니다.

- `F`
- `H`
- `K`
- `M`
- `d`
- `f`
- `g`
- `h`
- `m`
- `s`
- `t`
- `y`
- `z`
- `%`
- `:`
- `/`
- `"`
- `'`
- `\`

다른 모든 문자는 항상 문자 리터럴로 해석되며, 형식 지정 작업에서 변경되지 않고 결과 문자열에 포함됩니다. 구문 분석 작업에서는 입력 문자열의 문자와 정확히 일치해야 하며, 비교 시 대/소문자를 구분합니다.

다음 예제에는 형식 문자열에서 현지 표준 시간대를 나타내는 리터럴 문자 "PST"(태평양 표준시) 및 "PDT"(태평양 일광 절약 시간)가 포함되어 있습니다. 문자열이 결과 문자열에 포함되고, 현지 표준 시간대 문자열을 포함하는 문자열도 성공적으로 구문 분석되는 것을 확인합니다.

```
C#
```

```
using System;
using System.Globalization;

public class Example5
{
    public static void Main()
    {
        String[] formats = { "dd MMM yyyy hh:mm tt PST",
                             "dd MMM yyyy hh:mm tt PDT" };
        var dat = new DateTime(2016, 8, 18, 16, 50, 0);
        // Display the result string.
        Console.WriteLine(dat.ToString(formats[1]));

        // Parse a string.
        String value = "25 Dec 2016 12:00 pm PST";
        DateTime newDate;
        if (DateTime.TryParseExact(value, formats, null,
                                   DateTimeStyles.None, out newDate))
            Console.WriteLine(newDate);
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

문자가 결과 문자열에 포함되거나 입력 문자열에서 성공적으로 구문 분석될 수 있도록 하기 위해 문자를 예약 문자가 아니라 리터럴 문자로 해석되도록 지정하는 두 가지 방법이 있습니다.

- 각 예약된 문자를 이스케이프합니다. 자세한 내용은 이스케이프 문자 사용을 참조하세요.

다음 예제에는 형식 문자열에서 현지 표준 시간대를 나타내는 리터럴 문자 "pst"(태평양 표준시)가 포함되어 있습니다. "s"와 "t"는 둘 다 사용자 지정 형식 문자열이므로 두 문자를 문자 리터럴로 해석되도록 이스케이프해야 합니다.

C#

```
using System;
using System.Globalization;

public class Example3
{
    public static void Main()
    {
        String format = "dd MMM yyyy hh:mm tt p\\s\\t";
        var dat = new DateTime(2016, 8, 18, 16, 50, 0);
        // Display the result string.
```

```
        Console.WriteLine(dat.ToString(format));

        // Parse a string.
        String value = "25 Dec 2016 12:00 pm pst";
        DateTime newDate;
        if (DateTime.TryParseExact(value, format, null,
                                   DateTimeStyles.None, out newDate))
            Console.WriteLine(newDate);
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM pst
//       12/25/2016 12:00:00 PM
```

- 전체 리터럴 문자열을 따옴표나 아포스트로피로 묶습니다. 다음 예제는 "pst"가 따옴표로 묶여 구분된 전체 문자열을 문자 리터럴로 해석해야 한다는 점을 제외하고 이전 예제와 같습니다.

C#

```
using System;
using System.Globalization;

public class Example6
{
    public static void Main()
    {
        String format = "dd MMM yyyy hh:mm tt \"pst\"";
        var dat = new DateTime(2016, 8, 18, 16, 50, 0);
        // Display the result string.
        Console.WriteLine(dat.ToString(format));

        // Parse a string.
        String value = "25 Dec 2016 12:00 pm pst";
        DateTime newDate;
        if (DateTime.TryParseExact(value, format, null,
                                   DateTimeStyles.None, out newDate))
            Console.WriteLine(newDate);
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM pst
//       12/25/2016 12:00:00 PM
```

참고

단일 사용자 지정 형식 지정자 사용

사용자 지정 날짜 및 시간 형식 문자열은 둘 이상의 문자로 구성됩니다. 날짜 및 시간 형식 지정 메서드는 단일 문자 문자열을 표준 날짜 및 시간 형식 문자열로 해석합니다. 날짜 및 시간 형식 지정 메서드가 문자를 유효한 형식 지정자로 인식하지 못하면 FormatException이 throw됩니다. 예를 들어, "h" 지정자로만 구성된 형식 문자열은 표준 날짜 및 시간 형식 문자열로 해석됩니다. 그러나 이 경우 "h" 표준 날짜 및 시간 형식 지정자가 없으므로 예외가 throw됩니다.

사용자 지정 날짜 및 시간 형식 지정자 중 하나를 형식 문자열의 유일한 지정자로 사용하려면(즉, "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":" 또는 "/" 사용자 지정 형식 지정자를 단독으로 사용하려면) 지정자 앞이나 뒤에 공백을 포함하거나 단일 사용자 지정 날짜 및 시간 지정자 앞에 백분율("%") 형식 지정자를 포함합니다.

예를 들어, "`%h`"는 현재 날짜 및 시간 값이 나타내는 시간을 표시하는 사용자 지정 날짜 및 시간 형식 문자열로 해석됩니다. 결과 문자열에 시간 외에 공백이 포함되기는 하지만 "h" 또는 "h " 형식 지정자를 사용할 수도 있습니다. 다음 예제에서는 이러한 세 가지 형식 문자열을 보여 줍니다.

C#

```
DateTime dat1 = new DateTime(2009, 6, 15, 13, 45, 0);

Console.WriteLine("'{0:%h}'", dat1);
Console.WriteLine("'{0: h}'", dat1);
Console.WriteLine("'{0:h }'", dat1);
// The example displays the following output:
//       '1'
//       ' 1'
//       '1 '
```

이스케이프 문자 사용

형식 문자열의 "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":" 또는 "/" 문자는 리터럴 문자가 아닌 사용자 지정 형식 지정자로 해석됩니다. 문자가 형식 지정자로 해석되지 않도록 하려면 해당 문자 앞에 이스케이프 문자인 백슬래시(\)를 삽입하면 됩니다. 이스케이프 문자는 뒤에 오는 문자가 변경되지 않은 상태로 결과 문자열에 포함되어야 하는 문자 리터럴임을 나타냅니다.

결과 문자열에 백슬래시를 포함하려면`\\`처럼 두 개의 백슬래시를 연속해서 입력해야 합니다.

참고

C++ 및 C# 컴파일러 같은 일부 컴파일러에서는 하나의 백슬래시 문자가 이스케이프 문자로 해석될 수도 있습니다. 형식을 지정할 때 문자열이 올바로 해석되도록 하려면 해당 문자열 앞에 축자 문자열 리터럴 문자(@ 문자)를 사용하거나(C#의 경우) 각 백슬래시 앞에 또 다른 백슬래시를 추가하면 됩니다(C# 및 C++의 경우). 다음 C# 예제에서는 이 두 가지 방법을 모두 보여 줍니다.

다음 예제에서는 이스케이프 문자를 사용하여 형식 지정 작업에서 "h" 및 "m" 문자가 형식 지정자로 해석되지 않도록 합니다.

C#

```
DateTime date = new DateTime(2009, 06, 15, 13, 45, 30, 90);
string fmt1 = "h \\h m \\m";
string fmt2 = @"h \h m \m";

Console.WriteLine("{0} ({1}) -> {2}", date, fmt1, date.ToString(fmt1));
Console.WriteLine("{0} ({1}) -> {2}", date, fmt2, date.ToString(fmt2));
// The example displays the following output:
//       6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
//       6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
```

제어판 설정

제어판의 **국가 및 언어 옵션** 설정은 많은 사용자 지정 날짜 및 시간 형식 지정자를 사용한 형식 지정 작업으로 생성되는 결과 문자열에 영향을 줍니다. 이러한 설정은 서식을 제어하는 데 사용되는 값을 제공하는 현재 문화권과 연결된 개체를 초기화하는 DateTimeFormatInfo 데 사용됩니다. 다른 설정을 사용하는 컴퓨터는 다른 결과 문자열을 생성합니다.

또한 CultureInfo(String) 생성자를 사용하여 현재 시스템 문화권과 동일한 문화권을 나타내는 새 CultureInfo 개체를 인스턴스화하는 경우 제어판의 **국가 및 언어 옵션** 항목을 통해 설정된 사용자 지정 내용이 새 CultureInfo 개체에도 적용됩니다. CultureInfo(String, Boolean) 생성자를 사용하면 시스템의 사용자 지정 내용이 반영되지 않는 CultureInfo 개체를 만들 수 있습니다.

DateTimeFormatInfo 속성

서식 지정은 현재 DateTimeFormatInfo 문화권에서 암시적으로 제공되거나 형식 지정을 호출하는 메서드의 매개 변수에 의해 명시적으로 제공되는 현재 개체의 속성에 의해 IFormatProvider 영향을 받습니다. IFormatProvider 매개 변수의 경우 문화권을 나타내는 CultureInfo 개체나 DateTimeFormatInfo 개체를 지정해야 합니다.

대부분의 사용자 지정 날짜 및 시간 형식 지정자로 생성되는 결과 문자열도 현재 DateTimeFormatInfo 개체의 속성에 따라 달라집니다. 따라서 애플리케이션에서는 해당 DateTimeFormatInfo 속성을 변경하여 일부 사용자 지정 날짜 및 시간 형식 지정자에서 생성된 결과를 변경할 수 있습니다. 예를 들어, "ddd" 형식 지정자는 AbbreviatedDayNames 문자열 배열에서 찾은 약식 요일 이름을 결과 문자열에 추가합니다. 마찬가지로 "MMMM" 형식 지정자는 MonthNames 문자열 배열에서 찾은 전체 월 이름을 결과 문자열에 추가합니다.

참조

- System.DateTime
- System.IFormatProvider
- 형식 서식 지정
- 표준 날짜 및 시간 형식 문자열
- 샘플: .NET Core WinForms 서식 유틸리티(C#)
- 샘플: .NET Core WinForms 서식 유틸리티(Visual Basic)

표준 TimeSpan 서식 문자열

아티클 • 2025. 03. 22.

표준 TimeSpan 서식 문자열은 단일 서식 지정자를 사용하여 서식 지정 작업으로 인한 TimeSpan 값의 텍스트 표현을 정의합니다. 공백을 포함하여 둘 이상의 문자를 포함하는 모든 서식 문자열은 사용자 지정 TimeSpan 서식 문자열로 해석됩니다. 자세한 내용은 사용자 지정 TimeSpan 형식 문자열참조하세요.

TimeSpan 값의 문자열 표현은 TimeSpan.ToString 메서드의 오버로드에 대한 호출과 String.Format같은 복합 서식을 지원하는 메서드에 의해 생성됩니다. 자세한 내용은 형식 및 복합 서식참조하세요. 다음 예제에서는 서식 지정 작업에서 표준 형식 문자열을 사용하는 방법을 보여 줍니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan duration = new TimeSpan(1, 12, 23, 62);
      string output = "Time of Travel: " + duration.ToString("c");
      Console.WriteLine(output);

      Console.WriteLine($"Time of Travel: {duration:c}");
   }
}
// The example displays the following output:
//       Time of Travel: 1.12:24:02
//       Time of Travel: 1.12:24:02
```

표준 TimeSpan 형식 문자열은 TimeSpan.ParseExact 및 TimeSpan.TryParseExact 메서드에서도 구문 분석 작업에 필요한 입력 문자열 형식을 정의하는 데 사용됩니다. 구문 분석하면 값의 문자열 표현이 해당 값으로 변환됩니다. 다음 예제에서는 구문 분석 작업에서 표준 형식 문자열을 사용하는 방법을 보여 줍니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      string value = "1.03:14:56.1667";
      TimeSpan interval;
```

```
      try {
         interval = TimeSpan.ParseExact(value, "c", null);
         Console.WriteLine($"Converted '{value}' to {interval}");
      }
      catch (FormatException) {
         Console.WriteLine($"{value}: Bad Format");
      }
      catch (OverflowException) {
         Console.WriteLine($"{value}: Out of Range");
      }

      if (TimeSpan.TryParseExact(value, "c", null, out interval))
         Console.WriteLine($"Converted '{value}' to {interval}");
      else
         Console.WriteLine($"Unable to convert {value} to a time
interval.");
   }
}
// The example displays the following output:
//       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
//       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
```

다음 표에서는 표준 시간 간격 형식 지정자를 나열합니다.

테이블 확장

서식 지정자	이름	설명	예시
"c"	상수(고정) 형식	이 지정자는 문화권을 구분하지 않습니다. `[-]d'.'hh':'mm':'ss['.'fffffff]` 형식을 사용합니다. ("t" 및 "T" 형식 문자열은 동일한 결과를 생성합니다.) 추가 정보: 상수("c") 서식 지정자 .	`TimeSpan.Zero` -> 00:00:00 `New TimeSpan(0, 0, 30, 0)` -> 00:30:00 `New TimeSpan(3, 17, 25, 30, 500)` -> 3.17:25:30.50000000
"g"	일반 짧은 형식	이 지정자는 필요한 것만 출력합니다. 문화권에 민감하며 `[-][d':']h':'mm':'ss[.FFFFFFF]` 형식을 사용합니다. 추가 정보: 일반 짧음("g") 서식 지정자.	`New TimeSpan(1, 3, 16, 50, 500)` -> 1:3:16:50.5 (en-US) `New TimeSpan(1, 3, 16, 50, 500)` -> 1:3:16:50,5(fr-FR) `New TimeSpan(1, 3, 16, 50, 599)` -> 1:3:16:50.599 (en-US) `New TimeSpan(1, 3, 16, 50, 599)` -> 1:3:16:50,599 (fr-FR)

서식 지정자	이름	설명	예시
"G"	일반 긴 형식	이 지정자는 항상 일 및 7개의 소수 자릿수를 출력합니다. 문화권에 민감하며 `[-]d':'hh':'mm':'ss.fffffff`형식을 사용합니다. 추가 정보: 일반 긴("G") 서식 지정자 .	`New TimeSpan(18, 30, 0)` -> 0:18:30:00.0000000(en-US) `New TimeSpan(18, 30, 0)` -> 0:18:30:00,000000(fr-FR)

상수("c") 서식 지정자

"c" 형식 지정자는 다음 형식으로 TimeSpan 값의 문자열 표현을 반환합니다.

`[-][d.]hh:mm:ss[.fffffff]`

대괄호([및])의 요소는 선택 사항입니다. 마침표(.) 및 콜론(:)은 리터럴 기호입니다. 다음 표에서는 나머지 요소에 대해 설명합니다.

테이블 확장

요소	설명
-	음수 시간 간격을 나타내는 선택적 음수 기호입니다.
d	앞에 오는 0이 없는 선택적 일 수입니다.
hh	"00"에서 "23"에 이르는 시간 수입니다.
MM	"00"에서 "59"에 이르는 분 수입니다.
ss	"0"에서 "59"에 이르는 시간(초)입니다.
fffff	1초의 선택적 소수 부분입니다. 해당 값은 "0000001"(틱 1개 또는 1초의 1000만 분의 1)에서 "9999999"(9,999,999초, 1초 미만 1틱)에 이르기까지 다양할 수 있습니다.

"g" 및 "G" 형식 지정자와 달리 "c" 형식 지정자는 문화권을 구분하지 않습니다. .NET Framework 4 이전 버전에 공통적으로 적용되는 TimeSpan 값의 문자열 표현을 생성합니다. "c"는 기본 TimeSpan 형식 문자열입니다. TimeSpan.ToString() 메서드는 "c" 형식 문자열을 사용하여 시간 간격 값의 형식을 지정합니다.

> **참고**

TimeSpan "t" 및 "T" 표준 형식 문자열도 지원합니다. 이 문자열은 "c" 표준 형식 문자열과 동작이 동일합니다.

다음 예제에서는 두 TimeSpan 개체를 인스턴스화하고 이를 사용하여 산술 연산을 수행하고 결과를 표시합니다. 각 경우에 복합 서식을 사용하여 "c" 형식 지정자를 사용하여 TimeSpan 값을 표시합니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine($"{interval1:c} - {interval2:c} = {interval1 -
interval2:c}");
      Console.WriteLine($"{interval1:c} + {interval2:c} = {interval1 +
interval2:c}");

      interval1 = new TimeSpan(0, 0, 1, 14, 365);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine($"{interval1:c} + {interval2:c} = {interval1 +
interval2:c}");
   }
}
// The example displays the following output:
//       07:45:16 - 18:12:38 = -10:27:22
//       07:45:16 + 18:12:38 = 1.01:57:54
//       00:01:14.3650000 + 00:00:00.2143756 = 00:01:14.5793756
```

일반 짧게("g") 형식 지정자

"g" TimeSpan 형식 지정자는 필요한 요소만 포함하여 TimeSpan 값의 문자열 표현을 압축 형식으로 반환합니다. 형식은 다음과 같습니다.

`[-][d:]h:mm:ss[.FFFFFFF]`

대괄호([및])의 요소는 선택 사항입니다. 콜론(:) 리터럴 기호입니다. 다음 표에서는 나머지 요소에 대해 설명합니다.

테이블 확장

요소	설명
-	음수 시간 간격을 나타내는 선택적 음수 기호입니다.
d	앞에 오는 0이 없는 선택적 일 수입니다.
시간	앞에 오는 0이 없는 "0"에서 "23"까지의 시간 수입니다.
MM	"00"에서 "59"에 이르는 분 수입니다.
ss	"00"에서 "59"에 이르는 시간(초)입니다.
.	소수 자릿수 초 구분 기호입니다. 이는 사용자 재정의 없이 지정된 문화권의 NumberDecimalSeparator 속성과 동일합니다.
FFFFFFF	소수 자릿수 초입니다. 가능한 한 적은 자릿수가 표시됩니다.

"G" 형식 지정자와 마찬가지로 "g" 형식 지정자는 지역화됩니다. 소수 자릿수 초 구분 기호는 현재 문화권 또는 지정된 문화권의 NumberDecimalSeparator 속성을 기반으로 합니다.

다음 예제에서는 두 TimeSpan 개체를 인스턴스화하고 이를 사용하여 산술 연산을 수행하고 결과를 표시합니다. 각 경우에 복합 서식을 사용하여 "g" 형식 지정자를 사용하여 TimeSpan 값을 표시합니다. 또한 현재 시스템 문화권(이 경우 영어 - 미국 또는 en-US) 및 프랑스어 -프랑스(fr-FR) 문화권의 서식 지정 규칙을 사용하여 TimeSpan 값의 서식을 지정합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine($"{interval1:g} - {interval2:g} = {interval1 -
interval2:g}");
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                        "{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2));

      interval1 = new TimeSpan(0, 0, 1, 14, 36);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine($"{interval1:g} + {interval2:g} = {interval1 +
interval2:g}");
   }
```

```
}
// The example displays the following output:
//       7:45:16 - 18:12:38 = -10:27:22
//       7:45:16 + 18:12:38 = 1:1:57:54
//       0:01:14.036 + 0:00:00.2143756 = 0:01:14.2503756
```

일반 Long("G") 형식 지정자

"G" TimeSpan 형식 지정자는 항상 일 및 소수 자릿수 초를 모두 포함하는 긴 형식으로 TimeSpan 값의 문자열 표현을 반환합니다. "G" 표준 형식 지정자에서 생성된 문자열의 형식은 다음과 같습니다.

`[-]d:hh:mm:ss.fffffff`

대괄호([및])의 요소는 선택 사항입니다. 콜론(:) 리터럴 기호입니다. 다음 표에서는 나머지 요소에 대해 설명합니다.

테이블 확장

요소	설명
-	음수 시간 간격을 나타내는 선택적 음수 기호입니다.
d	앞에 오는 0이 없는 일 수입니다.
hh	"00"에서 "23"에 이르는 시간 수입니다.
MM	"00"에서 "59"에 이르는 분 수입니다.
ss	"00"에서 "59"에 이르는 시간(초)입니다.
.	소수 자릿수 초 구분 기호입니다. 이는 사용자 재정의 없이 지정된 문화권의 NumberDecimalSeparator 속성과 동일합니다.
fffff	소수 자릿수 초입니다.

"G" 형식 지정자와 마찬가지로 "g" 형식 지정자는 지역화됩니다. 소수 자릿수 초 구분 기호는 현재 문화권 또는 지정된 문화권의 NumberDecimalSeparator 속성을 기반으로 합니다.

다음 예제에서는 두 TimeSpan 개체를 인스턴스화하고 이를 사용하여 산술 연산을 수행하고 결과를 표시합니다. 각 경우에 복합 서식을 사용하여 "G" 형식 지정자를 사용하여 TimeSpan 값을 표시합니다. 또한 현재 시스템 문화권(이 경우 영어 - 미국 또는 en-US) 및 프랑스어 -프랑스(fr-FR) 문화권의 서식 지정 규칙을 사용하여 TimeSpan 값의 서식을 지정합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine($"{interval1:G} - {interval2:G} = {interval1 -
interval2:G}");
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                        "{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2));

      interval1 = new TimeSpan(0, 0, 1, 14, 36);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine($"{interval1:G} + {interval2:G} = {interval1 +
interval2:G}");
   }
}
// The example displays the following output:
//       0:07:45:16.0000000 - 0:18:12:38.0000000 = -0:10:27:22.0000000
//       0:07:45:16,0000000 + 0:18:12:38,0000000 = 1:01:57:54,0000000
//       0:00:01:14.0360000 + 0:00:00:00.2143756 = 0:00:01:14.2503756
```

참고하십시오

- 형식 서식 지정
- 사용자 지정 TimeSpan 형식 문자열
- 구문 분석 문자열

사용자 지정 TimeSpan 서식 문자열

TimeSpan 서식 문자열은 서식 지정 작업에서 발생하는 TimeSpan 값의 문자열 표현을 정의합니다. 사용자 지정 서식 문자열은 리터럴 문자 수와 함께 하나 이상의 사용자 지정 TimeSpan 형식 지정자로 구성됩니다. 표준 TimeSpan 형식 문자열 아닌 모든 문자열은 사용자 지정 TimeSpan 서식 문자열로 해석됩니다.

> **ⓘ Important**
>
> 사용자 지정 TimeSpan 서식 지정자에는 자리 표시자 구분 기호(예: 일과 시간, 분에서 몇 시간 또는 초에서 소수 자릿수 초)를 구분하는 기호가 포함되지 않습니다. 대신 이러한 기호를 문자열 리터럴로 사용자 지정 형식 문자열에 포함해야 합니다. 예를 들어 `"dd\.hh\:mm"` 마침표(.)를 일과 시간 사이의 구분 기호로 정의하고 콜론(:)을 시간과 분 사이의 구분 기호로 정의합니다.
>
> 또한 사용자 지정 TimeSpan 형식 지정자는 음수 및 양수 시간 간격을 구분할 수 있는 기호를 포함하지 않습니다. 기호 기호를 포함하려면 조건부 논리를 사용하여 형식 문자열을 생성해야 합니다. **기타 문자** 섹션에는 예제가 포함되어 있습니다.

TimeSpan 값의 문자열 표현은 TimeSpan.ToString 메서드의 오버로드에 대한 호출과 String.Format같은 복합 서식을 지원하는 메서드에 의해 생성됩니다. 자세한 내용은 형식 및 복합 서식참조하세요. 다음 예제에서는 서식 지정 작업에서 사용자 지정 형식 문자열을 사용하는 방법을 보여 줍니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan duration = new TimeSpan(1, 12, 23, 62);

      string output = null;
      output = "Time of Travel: " + duration.ToString("%d") + " days";
      Console.WriteLine(output);
      output = "Time of Travel: " + duration.ToString(@"dd\.hh\:mm\:ss");
      Console.WriteLine(output);

      Console.WriteLine($"Time of Travel: {duration:%d} day(s)");
      Console.WriteLine($"Time of Travel: {duration:dd\\.hh\\:mm\\:ss} days");
   }
}
// The example displays the following output:
//       Time of Travel: 1 days
//       Time of Travel: 01.12:24:02
```

```
//       Time of Travel: 1 day(s)
//       Time of Travel: 01.12:24:02 days
```

사용자 지정 TimeSpan 형식 문자열은 TimeSpan.ParseExact 및 TimeSpan.TryParseExact 메서드에서도 구문 분석 작업에 필요한 입력 문자열 형식을 정의하는 데 사용됩니다. 구문 분석하면 값의 문자열 표현이 해당 값으로 변환됩니다. 다음 예제에서는 구문 분석 작업에서 표준 형식 문자열을 사용하는 방법을 보여 줍니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      string value = null;
      TimeSpan interval;

      value = "6";
      if (TimeSpan.TryParseExact(value, "%d", null, out interval))
         Console.WriteLine($"{value} --> {interval.ToString("c")}");
      else
         Console.WriteLine($"Unable to parse '{value}'");

      value = "16:32.05";
      if (TimeSpan.TryParseExact(value, @"mm\:ss\.ff", null, out interval))
         Console.WriteLine($"{value} --> {interval.ToString("c")}");
      else
         Console.WriteLine($"Unable to parse '{value}'");

      value= "12.035";
      if (TimeSpan.TryParseExact(value, "ss\\.fff", null, out interval))
         Console.WriteLine($"{value} --> {interval.ToString("c")}");
      else
         Console.WriteLine($"Unable to parse '{value}'");
   }
}
// The example displays the following output:
//       6 --> 6.00:00:00
//       16:32.05 --> 00:16:32.0500000
//       12.035 --> 00:00:12.0350000
```

다음 표에서는 사용자 지정 날짜 및 시간 형식 지정자에 대해 설명합니다.

테이블 확장

서식 지정자	설명	예시
"d", "%d"	시간 간격의 전체 일 수입니다.	new TimeSpan(6, 14, 32, 17, 685):

서식 지정자	설명	예시
	추가 정보: "d" 사용자 지정 형식 지정자입니다.	`%d` --> "6" `d\.hh\:mm` --> "6.14:32"
`"dd"`-`"ddddddd"`	필요에 따라 앞에 오는 0으로 채워진 시간 간격의 전체 일 수입니다. 추가 정보: "dd"-"ddddddd" 사용자 지정 형식 지정자입니다.	`new TimeSpan(6, 14, 32, 17, 685):` `ddd` --> "006" `dd\.hh\:mm` --> "06.14:32"
`"h"`, `"%h"`	일의 일부로 계산되지 않는 시간 간격의 전체 시간 수입니다. 한 자리 시간은 앞에 0이 없습니다. 추가 정보: "h" 사용자 지정 형식 지정자입니다.	`new TimeSpan(6, 14, 32, 17, 685):` `%h` --> "14" `hh\:mm` --> "14:32"

`"hh"`	일의 일부로 계산되지 않는 시간 간격의 전체 시간 수입니다. 한 자리 시간에는 앞에 0이 있습니다.
	
	추가 정보: "hh" 사용자 지정 형식 지정자입니다.
		`new TimeSpan(6, 14, 32, 17, 685):`
		
		`hh` --> "14"
		
		`new TimeSpan(6, 8, 32, 17, 685):`
		
		`hh` --> 08

`"m"`, `"%m"`	시간 또는 일의 일부로 포함되지 않은 시간 간격의 전체 분 수입니다. 한 자리 분은 앞에 0이 없습니다. 추가 정보: "m" 사용자 지정 형식 지정자입니다.	`new TimeSpan(6, 14, 8, 17, 685):` `%m` --> "8" `h\:m` --> "14:8"

`"mm"`	시간 또는 일의 일부로 포함되지 않은 시간 간격의 전체 분 수입니다. 한 자리 분 앞에 0이 있습니다.
	
	추가 정보: "mm" 사용자 지정 형식 지정자입니다.
		`new TimeSpan(6, 14, 8, 17, 685):`
		
		`mm` --> "08"
		
		`new TimeSpan(6, 8, 5, 17, 685):`
		
		`d\.hh\:mm\:ss` --> 6.08:05:17

`"s"`, `"%s"`	시간, 일 또는 분의 일부로 포함되지 않은 시간 간격의 전체 초 수입니다. 한 자리 초 앞에 0이 없습니다. 추가 정보: "s" 사용자 지정 형식 지정자입니다.	`TimeSpan.FromSeconds(12.965)`: `%s` --> 12 `s\.fff` --> 12.965
`"ss"`	시간, 일 또는 분의 일부로 포함되지 않은 시간 간격의 전체 초 수입니다. 한 자리 초의 앞에 0이 있습니다.	`TimeSpan.FromSeconds(6.965)`: `ss` --> 06

서식 지정자	설명	예시
	추가 정보: "ss" 사용자 지정 형식 지정자입니다.	`ss\.fff` --> 06.965
`"f"`, `"%f"`	시간 간격의 1/10초입니다. 추가 정보: "f" 사용자 지정 형식 지정자입니다.	`TimeSpan.FromSeconds(6.895)`: `f` --> 8 `ss\.f` --> 06.8
`"ff"`	시간 간격의 1/100초입니다. 추가 정보: "ff" 사용자 지정 형식 지정자입니다.	`TimeSpan.FromSeconds(6.895)`: `ff` --> 89 `ss\.ff` --> 06.89
`"fff"`	시간 간격의 밀리초입니다. 추가 정보: "fff" 사용자 지정 형식 지정자입니다.	`TimeSpan.FromSeconds(6.895)`: `fff` --> 895 `ss\.fff` --> 06.895
`"ffff"`	시간 간격의 1/10000초입니다. 추가 정보: "ffff" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("0:0:6.8954321")`: `ffff` --> 8954 `ss\.ffff` --> 06.8954
`"fffff"`	시간 간격의 1/10000초입니다. 추가 정보: "fffff" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("0:0:6.8954321")`: `fffff` --> 89543 `ss\.fffff` --> 06.89543
`"ffffff"`	시간 간격의 100만 초입니다. 추가 정보: "ffffff" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("0:0:6.8954321")`: `ffffff` --> 895432 `ss\.ffffff` --> 06.895432
`"fffffff"`	시간 간격의 1초(또는 소수 틱)의 1,000만 번째입니다. 추가 정보: "fffffff" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("0:0:6.8954321")`: `fffffff` --> 8954321 `ss\.fffffff` --> 06.8954321
`"F"`, `"%F"`	시간 간격의 1/10초입니다. 숫자가 0이면 아무 것도 표시되지 않습니다.	`TimeSpan.Parse("00:00:06.32")`: `%F`: 3

서식 지정자	설명	예시
	추가 정보: "F" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("0:0:3.091")`: `ss\.F`: 03.
`"FF"`	시간 간격의 1/100초입니다. 모든 소수 후행 0 또는 0자리 자리는 포함되지 않습니다. 추가 정보: "FF" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("00:00:06.329")`: `FF`: 32 `TimeSpan.Parse("0:0:3.101")`: `ss\.FF`: 03.1
`"FFF"`	시간 간격의 밀리초입니다. 분수 후행 0은 포함되지 않습니다. 추가 정보:	`TimeSpan.Parse("00:00:06.3291")`: `FFF`: 329 `TimeSpan.Parse("0:0:3.1009")`: `ss\.FFF`: 03.1
`"FFFF"`	시간 간격의 1/10000초입니다. 분수 후행 0은 포함되지 않습니다. 추가 정보: "FFFF" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("00:00:06.32917")`: `FFFFF`: 3291 `TimeSpan.Parse("0:0:3.10009")`: `ss\.FFFF`: 03.1
`"FFFFF"`	시간 간격의 1/10000초입니다. 분수 후행 0은 포함되지 않습니다. 추가 정보: "FFFFF" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("00:00:06.329179")`: `FFFFF`: 32917 `TimeSpan.Parse("0:0:3.100009")`: `ss\.FFFFF`: 03.1
`"FFFFFF"`	시간 간격의 100만 초입니다. 분수 후행 0은 표시되지 않습니다. 추가 정보: "FFFFFF" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("00:00:06.3291791")`: `FFFFFF`: 329179 `TimeSpan.Parse("0:0:3.1000009")`: `ss\.FFFFFF`: 03.1
`"FFFFFFF"`	시간 간격으로 1초의 1000만 초입니다. 분수 후행 0 또는 7개의 0은 표시되지 않습니다.	`TimeSpan.Parse("00:00:06.3291791")`: `FFFFFF`: 3291791

서식 지정자	설명	예시
	추가 정보: "FFFFFFF" 사용자 지정 형식 지정자입니다.	`TimeSpan.Parse("0:0:3.1900000")`: `ss\.FFFFFF`: 03.19
'*문자열*'	리터럴 문자열 구분 기호입니다. 추가 정보: 다른 문자 .	`new TimeSpan(14, 32, 17):` `hh':'mm':'ss` --> "14:32:17"
\	이스케이프 문자입니다. 추가 정보: 다른 문자 .	`new TimeSpan(14, 32, 17):` `hh\:mm\:ss` --> "14:32:17"
기타 문자	다른 모든 이스케이프되지 않은 문자는 사용자 지정 형식 지정자로 해석됩니다. 추가 정보: 다른 문자 .	`new TimeSpan(14, 32, 17):` `hh\:mm\:ss` --> "14:32:17"

`"d"` 사용자 지정 형식 지정자

`"d"` 사용자 지정 형식 지정자는 시간 간격의 TimeSpan.Days 전체 일 수를 나타내는 속성 값을 출력합니다. 값이 두 자리 이상인 경우에도 TimeSpan 값으로 전체 일 수를 출력합니다. TimeSpan.Days 속성 값이 0이면 지정자는 "0"을 출력합니다.

`"d"` 사용자 지정 형식 지정자가 단독으로 사용되는 경우 표준 서식 문자열로 잘못 해석되지 않도록 지정 `"%d"` 합니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
TimeSpan ts1 = new TimeSpan(16, 4, 3, 17, 250);
Console.WriteLine(ts1.ToString("%d"));
// Displays 16
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 `"d"` 보여 줍니다.

C#

```
TimeSpan ts2 = new TimeSpan(4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.hh\:mm\:ss"));

TimeSpan ts3 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts3.ToString(@"d\.hh\:mm\:ss"));
// The example displays the following output:
//       0.04:03:17
//       3.04:03:17
```

표로 이동

"dd" - "ddddddd" 사용자 지정 형식 지정자

"dd", "ddd", "dddd", "ddddd", "dddddd" "ddddddd"및 "dddddddd" 사용자 지정 형식 지정자는 시간 간격의 TimeSpan.Days 전체 일 수를 나타내는 속성 값을 출력합니다.

출력 문자열에는 형식 지정자의 문자 수 d 로 지정된 최소 자릿수가 포함되며 필요에 따라 선행 0으로 채워지고 있습니다. 일 수의 숫자가 형식 지정자의 문자 수를 d 초과하면 전체 일 수가 결과 문자열에 출력됩니다.

다음 예제에서는 이러한 형식 지정자를 사용하여 두 TimeSpan 값의 문자열 표현을 표시합니다. 첫 번째 시간 간격의 days 구성 요소 값은 0입니다. 두 번째 일 구성 요소의 값은 365입니다.

C#

```
TimeSpan ts1 = new TimeSpan(0, 23, 17, 47);
TimeSpan ts2 = new TimeSpan(365, 21, 19, 45);

for (int ctr = 2; ctr <= 8; ctr++)
{
   string fmt = new String('d', ctr) + @"\.hh\:mm\:ss";
   Console.WriteLine($"{fmt} --> {ts1.ToString(fmt)}");
   Console.WriteLine($"{fmt} --> {ts2.ToString(fmt)}");
   Console.WriteLine();
}
// The example displays the following output:
//       dd\.hh\:mm\:ss --> 00.23:17:47
//       dd\.hh\:mm\:ss --> 365.21:19:45
//
//       ddd\.hh\:mm\:ss --> 000.23:17:47
//       ddd\.hh\:mm\:ss --> 365.21:19:45
//
//       dddd\.hh\:mm\:ss --> 0000.23:17:47
//       dddd\.hh\:mm\:ss --> 0365.21:19:45
//
//       ddddd\.hh\:mm\:ss --> 00000.23:17:47
//       ddddd\.hh\:mm\:ss --> 00365.21:19:45
//
//       dddddd\.hh\:mm\:ss --> 000000.23:17:47
//       dddddd\.hh\:mm\:ss --> 000365.21:19:45
//
//       ddddddd\.hh\:mm\:ss --> 0000000.23:17:47
//       ddddddd\.hh\:mm\:ss --> 0000365.21:19:45
//
//       dddddddd\.hh\:mm\:ss --> 00000000.23:17:47
//       dddddddd\.hh\:mm\:ss --> 00000365.21:19:45
```

표로 이동

"h" 사용자 지정 형식 지정자

"h" 사용자 지정 형식 지정자는 해당 날짜 구성 요소의 TimeSpan.Hours 일부로 계산되지 않는 시간 간격의 전체 시간 수를 나타내는 속성 값을 출력합니다. TimeSpan.Hours 속성 값이 0에서 9이면 한 자리 문자열 값을 반환하고, TimeSpan.Hours 속성 값이 10에서 23까지인 경우 두 자리 문자열 값을 반환합니다.

"h" 사용자 지정 형식 지정자가 단독으로 사용되는 경우 표준 서식 문자열로 잘못 해석되지 않도록 지정 "%h" 합니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine($"{ts:%h} hours {ts:%m} minutes");
// The example displays the following output:
//       3 hours 42 minutes
```

일반적으로 구문 분석 작업에서 단일 숫자만 포함하는 입력 문자열은 일 수로 해석됩니다. 대신 사용자 지정 형식 지정자를 사용하여 "%h" 숫자 문자열을 시간 수로 해석할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
string value = "8";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%h", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine($"Unable to convert '{value}' to a time interval");
// The example displays the following output:
//       08:00:00
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 "h" 보여 줍니다.

C#

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.h\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.h\:mm\:ss"));
// The example displays the following output:
//       0.14:03:17
//       3.4:03:17
```

표로 이동

"hh" 사용자 지정 형식 지정자

"hh" 사용자 지정 형식 지정자는 해당 날짜 구성 요소의 TimeSpan.Hours 일부로 계산되지 않는 시간 간격의 전체 시간 수를 나타내는 속성 값을 출력합니다. 0에서 9까지의 값의 경우 출력 문자열에는 선행 0이 포함됩니다.

일반적으로 구문 분석 작업에서 단일 숫자만 포함하는 입력 문자열은 일 수로 해석됩니다. 대신 사용자 지정 형식 지정자를 사용하여 "hh" 숫자 문자열을 시간 수로 해석할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
string value = "08";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "hh", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine($"Unable to convert '{value}' to a time interval");
// The example displays the following output:
//       08:00:00
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 "hh" 보여 줍니다.

C#

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.hh\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.hh\:mm\:ss"));
// The example displays the following output:
//       0.14:03:17
//       3.04:03:17
```

표로 이동

"m" 사용자 지정 형식 지정자

"m" 사용자 지정 형식 지정자는 해당 날짜 구성 요소의 TimeSpan.Minutes 일부로 계산되지 않는 시간 간격의 전체 시간(분)을 나타내는 속성 값을 출력합니다. TimeSpan.Minutes 속성의 값이 0에서 9이면 한 자리 문자열 값을 반환하고, TimeSpan.Minutes 속성의 값이 10에서 59까지인 경우 두 자리 문자열 값을 반환합니다.

"m" 사용자 지정 형식 지정자가 단독으로 사용되는 경우 표준 서식 문자열로 잘못 해석되지 않도록 지정 "%m" 합니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine($"{ts:%h} hours {ts:%m} minutes");
// The example displays the following output:
//       3 hours 42 minutes
```

일반적으로 구문 분석 작업에서 단일 숫자만 포함하는 입력 문자열은 일 수로 해석됩니다. 대신 사용자 지정 형식 지정자를 사용하여 `"%m"` 숫자 문자열을 분 수로 해석할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
string value = "3";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%m", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine($"Unable to convert '{value}' to a time interval");
// The example displays the following output:
//       00:03:00
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 `"m"` 보여 줍니다.

C#

```
TimeSpan ts1 = new TimeSpan(0, 6, 32);
Console.WriteLine($"{ts1:m\\:ss} minutes");

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine($"Elapsed time: {ts2:m\\:ss}");
// The example displays the following output:
//       6:32 minutes
//       Elapsed time: 18:44
```

표로 이동

`"mm"` 사용자 지정 형식 지정자

`"mm"` 사용자 지정 형식 지정자는 시간 또는 일 구성 요소의 TimeSpan.Minutes 일부로 포함되지 않은 시간 간격의 전체 시간(분)을 나타내는 속성 값을 출력합니다. 0에서 9까지의 값의 경우 출력 문자열에는 선행 0이 포함됩니다.

일반적으로 구문 분석 작업에서 단일 숫자만 포함하는 입력 문자열은 일 수로 해석됩니다. 대신 사용자 지정 형식 지정자를 사용하여 `"mm"` 숫자 문자열을 분 수로 해석할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
string value = "07";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "mm", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine($"Unable to convert '{value}' to a time interval");
// The example displays the following output:
//       00:07:00
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 `"mm"` 보여 줍니다.

C#

```
TimeSpan departTime = new TimeSpan(11, 12, 00);
TimeSpan arriveTime = new TimeSpan(16, 28, 00);
Console.WriteLine($"Travel time: {arriveTime - departTime:hh\\:mm}");
// The example displays the following output:
//       Travel time: 05:16
```

표로 이동

`"s"` 사용자 지정 형식 지정자

`"s"` 사용자 지정 형식 지정자는 시간, 일 또는 분 구성 요소의 TimeSpan.Seconds 일부로 포함되지 않은 시간 간격의 전체 초 수를 나타내는 속성 값을 출력합니다. TimeSpan.Seconds 속성의 값이 0에서 9이면 한 자리 문자열 값을 반환하고, TimeSpan.Seconds 속성의 값이 10에서 59까지인 경우 두 자리 문자열 값을 반환합니다.

`"s"` 사용자 지정 형식 지정자가 단독으로 사용되는 경우 표준 서식 문자열로 잘못 해석되지 않도록 지정 `"%s"` 합니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
TimeSpan ts = TimeSpan.FromSeconds(12.465);
Console.WriteLine(ts.ToString("%s"));
// The example displays the following output:
//       12
```

일반적으로 구문 분석 작업에서 단일 숫자만 포함하는 입력 문자열은 일 수로 해석됩니다. 대신 사용자 지정 형식 지정자를 사용하여 `"%s"` 숫자 문자열을 초 수로 해석할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
string value = "9";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%s", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine($"Unable to convert '{value}' to a time interval");
// The example displays the following output:
//       00:00:09
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 `"s"` 보여 줍니다.

C#

```
TimeSpan startTime = new TimeSpan(0, 12, 30, 15, 0);
TimeSpan endTime = new TimeSpan(0, 12, 30, 21, 3);
Console.WriteLine(@"Elapsed Time: {0:s\:fff} seconds",
                  endTime - startTime);
// The example displays the following output:
//       Elapsed Time: 6:003 seconds
```

표로 이동

`"ss"` 사용자 지정 형식 지정자

`"ss"` 사용자 지정 형식 지정자는 시간, 일 또는 분 구성 요소의 TimeSpan.Seconds 일부로 포함되지 않은 시간 간격의 전체 초 수를 나타내는 속성 값을 출력합니다. 0에서 9까지의 값의 경우 출력 문자열에는 선행 0이 포함됩니다.

일반적으로 구문 분석 작업에서 단일 숫자만 포함하는 입력 문자열은 일 수로 해석됩니다. 대신 사용자 지정 형식 지정자를 사용하여 `"ss"` 숫자 문자열을 초 수로 해석할 수 있습니다. 다음 예제에서 이에 대해 설명합니다.

C#

```
string[] values = { "49", "9", "06" };
TimeSpan interval;
foreach (string value in values)
{
   if (TimeSpan.TryParseExact(value, "ss", null, out interval))
      Console.WriteLine(interval.ToString("c"));
   else
      Console.WriteLine($"Unable to convert '{value}' to a time interval");
}
// The example displays the following output:
//       00:00:49
//       Unable to convert '9' to a time interval
//       00:00:06
```

다음 예제에서는 사용자 지정 형식 지정자의 사용을 `"ss"` 보여 줍니다.

C#

```
TimeSpan interval1 = TimeSpan.FromSeconds(12.60);
Console.WriteLine(interval1.ToString(@"ss\.fff"));

TimeSpan interval2 = TimeSpan.FromSeconds(6.485);
Console.WriteLine(interval2.ToString(@"ss\.fff"));
// The example displays the following output:
//       12.600
//       06.485
```

표로 이동

`"f"` 사용자 지정 형식 지정자

`"f"` 사용자 지정 형식 지정자는 시간 간격으로 1/10초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 하나의 소수 자릿수를 포함해야 합니다.

`"f"` 사용자 지정 형식 지정자가 단독으로 사용되는 경우 표준 서식 문자열로 잘못 해석되지 않도록 지정 `"%f"` 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"f"` 1/10초의 값을 표시합니다 TimeSpan . `"f"` 는 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 `"s"` 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//               %f: 8
```

```
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

"ff" 사용자 지정 형식 지정자

"ff" 사용자 지정 형식 지정자는 시간 간격으로 수백 초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 두 개의 소수 자릿수를 포함해야 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 "ff" 1/100초의 TimeSpan 값을 표시합니다. "ff" 는 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 "s" 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
```

```
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

`"fff"` 사용자 지정 형식 지정자

`"fff"` 사용자 지정 형식 지정자(3 `f` 자 포함)는 시간 간격으로 밀리초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 세 개의 소수 자릿수를 포함해야 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"fff"` 값에 밀리초를 표시합니다 TimeSpan . `"fff"` 는 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 `"s"` 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
```

```
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

`"ffff"` 사용자 지정 형식 지정자

`"ffff"` 사용자 지정 형식 지정자(4 `f` 자 포함)는 시간 간격으로 1/10000초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 네 개의 소수 자릿수를 포함해야 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"ffff"` 1/10000초의 TimeSpan 값을 표시합니다. `"ffff"` 는 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 `"s"` 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
```

```
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

"fffff" 사용자 지정 형식 지정자

"fffff" 사용자 지정 형식 지정자(5 f 자 포함)는 시간 간격으로 1/10000초의 10만 분의 1을 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 5개의 소수 자릿수를 포함해야 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 "fffff" 1/10000초의 TimeSpan 값을 표시합니다. "fffff" 는 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 "s" 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
```

```
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

`"ffffff"` 사용자 지정 형식 지정자

`"ffffff"` 사용자 지정 형식 지정자(6 `f` 자 포함)는 시간 간격으로 1초의 백만 번째를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 6개의 소수 자릿수를 포함해야 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"ffffff"` 1초의 백만 번째 값을 값으로 TimeSpan 표시합니다. 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 `"s"` 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//                %f: 8
//                ff: 87
//               fff: 876
//              ffff: 8765
//             fffff: 87654
//            ffffff: 876543
//           fffffff: 8765432
//
```

```
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

`"fffffff"` 사용자 지정 형식 지정자

`"fffffff"` 사용자 지정 형식 지정자(7 `f` 자 포함)는 시간 간격으로 1초의 1,000만 번째(또는 소수 눈금 수)를 출력합니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열은 정확히 7개의 소수 자릿수를 포함해야 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"fffffff"` 소수 자릿수의 틱을 값에 TimeSpan 표시합니다. 먼저 유일한 형식 지정자로 사용된 다음 사용자 지정 형식 문자열의 `"s"` 지정자와 결합됩니다.

C#

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new string('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = $"s\\.{new string('f', ctr)}";
   Console.WriteLine($"{fmt,10}: {ts.ToString(fmt)}");
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
```

```
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

표로 이동

`"F"` 사용자 지정 형식 지정자

`"F"` 사용자 지정 형식 지정자는 시간 간격으로 1/10초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. 시간 간격의 1/10초 값이 0이면 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 두 번째 숫자의 1/10은 선택 사항입니다.

`"F"` 사용자 지정 형식 지정자가 단독으로 사용되는 경우 표준 서식 문자열로 잘못 해석되지 않도록 지정 `"%F"` 합니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"F"` 1/10초의 값을 표시합니다 TimeSpan . 또한 구문 분석 작업에서 이 사용자 지정 형식 지정자를 사용합니다.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.669");
Console.WriteLine($"{ts1} ('%F') --> {ts1:%F}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.091");
Console.WriteLine($"{ts2} ('ss\\.F') --> {ts2:ss\\.F}");
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.12" };
string fmt = @"h\:m\:ss\.F";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6690000 ('%F') --> 6
//       00:00:03.0910000 ('ss\.F') --> 03.
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.F') --> 00:00:03
```

```
//       0:0:03.1 ('h\:m\:ss\.F') --> 00:00:03.1000000
//       Cannot parse 0:0:03.12 with 'h\:m\:ss\.F'.
```

표로 이동

"FF" 사용자 지정 형식 지정자

"FF" 사용자 지정 형식 지정자는 시간 간격으로 수백 초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. 후행 소수 자릿수 0이 있는 경우 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 두 번째 숫자의 1/1000은 선택 사항입니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 "FF" 1/100초의 TimeSpan 값을 표시합니다. 또한 구문 분석 작업에서 이 사용자 지정 형식 지정자를 사용합니다.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697");
Console.WriteLine($"{ts1} ('FF') --> {ts1:FF}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.809");
Console.WriteLine($"{ts2} ('ss\\.FF') --> {ts2:ss\\.FF}");
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.127" };
string fmt = @"h\:m\:ss\.FF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6970000 ('FF') --> 69
//       00:00:03.8090000 ('ss\.FF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FF') --> 00:00:03
//       0:0:03.1 ('h\:m\:ss\.FF') --> 00:00:03.1000000
//       Cannot parse 0:0:03.127 with 'h\:m\:ss\.FF'.
```

표로 이동

`"FFF"` 사용자 지정 형식 지정자

`"FFF"` 사용자 지정 형식 지정자(3 `F` 자 포함)는 시간 간격으로 밀리초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. 후행 소수 자릿수 0이 있는 경우 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 두 번째 숫자의 1/10, 1000, 1/1000은 선택 사항입니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"FFF"` 값에 1/1/10000 TimeSpan 초를 표시합니다. 또한 구문 분석 작업에서 이 사용자 지정 형식 지정자를 사용합니다.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974");
Console.WriteLine($"{ts1} ('FFF') --> {ts1:FFF}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8009");
Console.WriteLine($"{ts2} ('ss\\.FFF') --> {ts2:ss\\.FFF}");
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279" };
string fmt = @"h\:m\:ss\.FFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974000 ('FFF') --> 697
//       00:00:03.8009000 ('ss\.FFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.1279 with 'h\:m\:ss\.FFF'.
```

표로 이동

`"FFFF"` 사용자 지정 형식 지정자

`"FFFF"` 사용자 지정 형식 지정자(4 `F` 자 포함)는 시간 간격으로 1/10000초를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. 후행 소수 자릿수 0이 있는 경우 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문

분석 작업에서 두 번째 숫자의 10분의 1, 100초, 천 번째 및 10,000분의 1이 있는 것은 선택 사항입니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"FFFF"` 1/10000초의 TimeSpan 값을 표시합니다. 또한 구문 분석 작업에서 사용자 지정 형식 지정자를 사용합니다 `"FFFF"` .

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.69749");
Console.WriteLine($"{ts1} ('FFFF') --> {ts1:FFFF}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.80009");
Console.WriteLine($"{ts2} ('ss\\.FFFF') --> {ts2:ss\\.FFFF}");
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.12795" };
string fmt = @"h\:m\:ss\.FFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974900 ('FFFF') --> 6974
//       00:00:03.8000900 ('ss\.FFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.12795 with 'h\:m\:ss\.FFFF'.
```

표로 이동

`"FFFFF"` 사용자 지정 형식 지정자

`"FFFFF"` 사용자 지정 형식 지정자(5 `F` 자 포함)는 시간 간격으로 1/10000초의 10만 분의 1을 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. 후행 소수 자릿수 0이 있는 경우 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 10분의 1, 100초, 1000초, 10000초, 10000초 및 10만 번째 숫자의 존재는 선택 사항입니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"FFFFF"` 1/10000초의 TimeSpan 값을 표시합니다. 또한 구문 분석 작업에서 사용자 지정 형식 지정자를 사용합니다 `"FFFFF"` .

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697497");
Console.WriteLine($"{ts1} ('FFFFF') --> {ts1:FFFFF}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.800009");
Console.WriteLine($"{ts2} ('ss\\.FFFFF') --> {ts2:ss\\.FFFFF}");
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.127956" };
string fmt = @"h\:m\:ss\.FFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974970 ('FFFFF') --> 69749
//       00:00:03.8000090 ('ss\.FFFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.127956 with 'h\:m\:ss\.FFFF'.
```

표로 이동

`"FFFFFF"` 사용자 지정 형식 지정자

`"FFFFFF"` 사용자 지정 형식 지정자(6 `F` 자 포함)는 시간 간격으로 1초의 백만 번째를 출력합니다. 서식 지정 작업에서 나머지 소수 자릿수는 잘립니다. 후행 소수 자릿수 0이 있는 경우 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 1/10, 1000, 1000, 10,000, 10,000번째, 1/1000, 10000초 및 100만 번째 숫자의 존재는 선택 사항입니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"FFFFFF"` 1초의 백만 번째 값을 값으로 TimeSpan 표시합니다. 또한 구문 분석 작업에서 이 사용자 지정 형식 지정자를 사용합니다.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine($"{ts1} ('FFFFFF') --> {ts1:FFFFFF}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8000009");
Console.WriteLine($"{ts2} ('ss\\.FFFFFF') --> {ts2:ss\\.FFFFFF}");
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974974 ('FFFFFF') --> 697497
//       00:00:03.8000009 ('ss\.FFFFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.1279569 with 'h\:m\:ss\.FFFFFF'.
```

표로 이동

`"FFFFFFF"` 사용자 지정 형식 지정자

`"FFFFFFF"` 사용자 지정 형식 지정자(7 `F` 자 포함)는 시간 간격으로 1초의 1,000만 번째(또는 소수 눈금 수)를 출력합니다. 후행 소수 자릿수 0이 있는 경우 결과 문자열에 포함되지 않습니다. TimeSpan.ParseExact 또는 TimeSpan.TryParseExact 메서드를 호출하는 구문 분석 작업에서 입력 문자열에 7개의 소수 자릿수가 있는 것은 선택 사항입니다.

다음 예제에서는 사용자 지정 형식 지정자를 사용하여 `"FFFFFFF"` 1초의 소수 부분을 값에 TimeSpan 표시합니다. 또한 구문 분석 작업에서 이 사용자 지정 형식 지정자를 사용합니다.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine($"{ts1} ('FFFFFFF') --> {ts1:FFFFFFF}");

TimeSpan ts2 = TimeSpan.Parse("0:0:3.9500000");
Console.WriteLine($"{ts2} ('ss\\.FFFFFFF') --> {ts2:ss\\.FFFFFFF}");
```

```
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine($"{input} ('{fmt}') --> {ts3}");
   else
      Console.WriteLine($"Cannot parse {input} with '{fmt}'.");
}
// The example displays the following output:
//    Formatting:
//    00:00:03.6974974 ('FFFFFFF') --> 6974974
//    00:00:03.9500000 ('ss\.FFFFFFF') --> 03.95
//
//    Parsing:
//    0:0:03. ('h\:m\:ss\.FFFFFFF') --> 00:00:03
//    0:0:03.12 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1200000
//    0:0:03.1279569 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1279569
```

표로 이동

기타 문자

공백 문자를 포함하여 서식 문자열의 다른 모든 이스케이프되지 않은 문자는 사용자 지정 형식 지정자로 해석됩니다. 대부분의 경우 다른 이스케이프되지 않은 문자가 있으면 FormatException.

형식 문자열에 리터럴 문자를 포함하는 방법에는 두 가지가 있습니다.

- 작은따옴표(리터럴 문자열 구분 기호)로 묶습니다.
- 이스케이프 문자로 해석되는 백슬래시("\")로 앞에 섰습니다. 즉, C#에서 서식 문자열은 @-quoted이거나 리터럴 문자 앞에 추가 백슬래시가 와야 합니다.

  경우에 따라 조건부 논리를 사용하여 이스케이프된 리터럴을 형식 문자열에 포함해야 할 수 있습니다. 다음 예제에서는 조건부 논리를 사용하여 음수 시간 간격에 대한 기호 기호를 포함합니다.

  C#

  ```
  using System;

  public class Example
  {
     public static void Main()
  ```

```
   {
      TimeSpan result = new DateTime(2010, 01, 01) - DateTime.Now;
      String fmt = (result < TimeSpan.Zero ?  "\\-" : "") + "dd\\.hh\\:mm";

      Console.WriteLine(result.ToString(fmt));
      Console.WriteLine($"Interval: {result.ToString(fmt)}");
   }
}
// The example displays output like the following:
//       -5582.12:21
//       Interval: -5582.12:21
```

.NET은 시간 간격에서 구분 기호에 대한 문법을 정의하지 않습니다. 즉, 일과 시간, 시간, 분, 분, 초, 초 및 초의 분수 사이의 구분 기호는 모두 형식 문자열에서 문자 리터럴로 처리되어야 합니다.

다음 예제에서는 이스케이프 문자와 작은따옴표를 모두 사용하여 출력 문자열에 단어를 `"minutes"` 포함하는 사용자 지정 서식 문자열을 정의합니다.

C#

```
TimeSpan interval = new TimeSpan(0, 32, 45);
// Escape literal characters in a format string.
string fmt = @"mm\:ss\ \m\i\n\u\t\e\s";
Console.WriteLine(interval.ToString(fmt));
// Delimit literal characters in a format string with the ' symbol.
fmt = "mm':'ss' minutes'";
Console.WriteLine(interval.ToString(fmt));
// The example displays the following output:
//       32:45 minutes
//       32:45 minutes
```

표로 이동

참고하십시오

- 형식 서식 지정
- 표준 TimeSpan 형식 문자열

Last updated on 2026. 01. 06.

열거형 형식 문자열

아티클 • 2024. 03. 16.

Enum.ToString 메서드를 사용하여 열거형 멤버의 숫자, 16진수 또는 문자열 값을 나타내는 새 문자열 개체를 만들 수 있습니다. 이 메서드는 열거형 형식 문자열 중 하나를 사용하여 반환할 값을 지정합니다.

다음 섹션에서는 열거형 형식 문자열과 반환되는 값을 보여 줍니다. 이러한 형식 지정자는 대/소문자를 구분하지 않습니다.

G 또는 g

가능한 경우 열거형 항목을 문자열 값으로 표시하고, 가능하지 않은 경우 현재 인스턴스의 정수 값을 표시합니다. FlagsAttribute을 설정하여 열거형을 정의한 경우 유효한 각 항목의 문자열 값이 쉼표로 구분되어 서로 연결됩니다. `Flags` 특성을 설정하지 않은 경우 잘못된 값이 숫자 항목으로 표시됩니다. 다음 예제에서는 `G` 형식 지정자를 보여 줍니다.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("G"));       // 7
Console.WriteLine(ConsoleColor.Red.ToString("G"));     // Red

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("G"));           // Hidden, Archive
```

F 또는 f

열거형 항목을 문자열 값으로 표시합니다(가능한 경우). `Flags` 특성이 없어도 열거형 항목의 총합으로 값을 표시할 수 있는 경우 유효한 각 항목의 문자열 값이 쉼표로 구분되어 서로 연결됩니다. 열거형 항목으로 값을 확인할 수 없는 경우에는 값의 형식이 정수 값으로 지정됩니다. 다음 예제에서는 `F` 형식 지정자를 보여 줍니다.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("F"));       // Monday, Saturday
Console.WriteLine(ConsoleColor.Blue.ToString("F"));    // Blue

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("F"));           // Hidden, Archive
```

D 또는 d

열거형 항목을 가능한 가장 짧은 표현의 정수 값으로 표시합니다. 다음 예제에서는 `D` 형식 지정자를 보여 줍니다.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("D"));        // 7
Console.WriteLine(ConsoleColor.Cyan.ToString("D"));     // 11

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("D"));            // 34
```

X 또는 x

열거형 항목을 16진수 값으로 표시합니다. 결과 문자열이 열거형의 기본 숫자 형식에서 바이트마다 2개 문자를 갖도록 하기 위해 필요에 따라 앞에 0이 오는 값으로 표현됩니다. 다음 예제에서는 X 형식 지정자를 보여 줍니다. 예제에서 DayOfWeek, ConsoleColor 및 FileAttributes의 기본 형식은 Int32 또는 8자 결과 문자열을 생성하는 32비트(또는 4바이트) 정수입니다.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("X"));        // 00000007
Console.WriteLine(ConsoleColor.Cyan.ToString("X"));     // 0000000B

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("X"));            // 00000022
```

예시

다음 예제에서는 세 개의 항목 `Red`, `Blue` 및 `Green`으로 구성된 `Colors`라는 열거형을 정의합니다.

C#

```
public enum Color { Red = 1, Blue = 2, Green = 3 };
```

열거형이 정의되면 다음과 같이 인스턴스를 선언할 수 있습니다.

C#

```
Color myColor = Color.Green;
```

그런 다음 `Color.ToString(System.String)` 메서드를 사용하여 전달된 형식 지정자에 따라 열거형 값을 다양한 방식으로 표시할 수 있습니다.

C#

```
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("G"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("F"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("D"));
Console.WriteLine("The value of myColor is 0x{0}.",
                  myColor.ToString("X"));
// The example displays the following output to the console:
//       The value of myColor is Green.
//       The value of myColor is Green.
//       The value of myColor is 3.
//       The value of myColor is 0x00000003.
```

참고 항목

- 형식 서식 지정

복합 형식 지정

.NET 복합 서식 지정 기능은 개체 목록과 복합 서식 문자열을 입력으로 사용합니다. 복합 서식 문자열은 형식 항목이라고 하는 인덱싱된 자리 표시자와 혼합된 고정 텍스트로 구성됩니다. 이러한 형식 항목은 목록의 개체에 해당합니다. 서식 지정 작업을 통해 원래의 고정 텍스트와 목록에 있는 개체의 문자열 표현이 결합된 형태의 결과 문자열을 얻을 수 있습니다.

> **ⓘ 중요**
>
> 복합 형식 문자열을 사용하는 대신 사용 중인 언어 및 해당 버전이 지원하는 경우 *보간된 문자열*을 사용할 수 있습니다. 보간된 문자열에는 *보간된 식이 포함됩니다*. 보간된 각 식은 식의 값으로 확인되고 문자열이 할당될 때 결과 문자열에 포함됩니다. 자세한 내용은 **문자열 보간(C# 참조)** 및 **보간된 문자열(Visual Basic 참조)을 참조**하세요.

다음 메서드는 복합 서식 지정 기능을 지원합니다.

- String.Format형식이 지정된 결과 문자열을 반환하는 입니다.
- StringBuilder.AppendFormat 형식이 지정된 결과 문자열을 StringBuilder 객체에 추가합니다.
- 형식이 지정된 결과 문자열을 Console.WriteLine 콘솔에 표시하는 메서드의 일부 오버로드입니다.
- 특정 오버로드에서는 형식화된 결과 문자열을 스트림이나 파일에 쓰는 TextWriter.WriteLine 메서드를 사용합니다. 파생된 TextWriter클래스(예: StreamWriter 및 HtmlTextWriter)도 이 기능을 공유합니다.
- Debug.WriteLine(String, Object[])- 추적 수신기에 형식이 지정된 메시지를 출력합니다.
- Trace.TraceError(String, Object[]), Trace.TraceInformation(String, Object[]), 및 Trace.TraceWarning(String, Object[]) 메서드는 추적 수신기에 형식이 지정된 메시지를 출력합니다.
- TraceSource.TraceInformation(String, Object[]) 수신기를 추적하기 위해 정보 메서드를 작성하는 메서드입니다.

복합 형식 문자열

복합 서식 문자열 및 개체 목록은 복합 서식 지정 기능을 지원하는 메서드의 인수로 사용됩니다. 복합 서식 문자열은 하나 이상의 서식 항목과 섞인 고정 텍스트의 0개 이상의 실행으로 구성됩니다. 고정 텍스트는 선택한 문자열이며 각 서식 항목은 목록의 개체 또는 상자 구조체에 해당합니다. 각 개체의 문자열 표현은 해당 형식 항목을 대체합니다.

다음 코드 조각을 고려합니다.Format

```
C#
```

```
string.Format("Name = {0}, hours = {1:hh}", "Fred", DateTime.Now);
```

고정 텍스트는 `Name =` 와 `, hours =` 입니다. 형식 항목은 `{0}` 인덱스 0이 문자열 리터럴 `"Fred"` 에 해당하고 `{1:hh}` 인덱스 1이 값 `DateTime.Now` 에 해당하는 형식 항목입니다.

항목 구문 서식 지정

각 서식 항목의 형태와 구성 요소는 다음과 같습니다.

`{index[,width][:formatString]}`

일치하는 중괄호(`{` 및 `}`)가 필요합니다.

인덱스 구성 요소

매개 변수 지정자라고도 하는 필수 `index` 구성 요소는 개체 목록에서 해당 항목을 식별하는 0부터 시작하는 숫자입니다. 즉, 매개 변수 지정자가 목록의 첫 번째 개체에 서식을 지정하는 서식 항목입니다 `0` . 형식 항목에서 매개 변수 지정자 `1` 는 목록 내 두 번째 개체부터 서식을 지정하며, 이후 항목에도 계속해서 적용됩니다. 다음 예제에서는 10보다 작은 소수를 나타내기 위해 0부터 3까지 번호가 매겨진 4개의 매개 변수 지정자를 포함합니다.

C#

```
string primes = string.Format("Four prime numbers: {0}, {1}, {2}, {3}",
                              2, 3, 5, 7);
Console.WriteLine(primes);

// The example displays the following output:
//       Four prime numbers: 2, 3, 5, 7
```

여러 형식 항목은 동일한 매개 변수 지정자를 지정하여 개체 목록에서 동일한 요소를 참조할 수 있습니다. 예를 들어, 다음 예제와 같이 복합 형식 문자열 `"0x{0:X} {0:E} {0:N}"` 을 지정하여 동일한 숫자 값을 16진수, 과학적, 숫자 형식으로 형식을 지정할 수 있습니다.

C#

```
string multiple = string.Format("0x{0:X} {0:E} {0:N}",
                                Int64.MaxValue);
Console.WriteLine(multiple);

// The example displays the following output:
//       0x7FFFFFFFFFFFFFFF 9.223372E+018 9,223,372,036,854,775,807.00
```

각 형식 항목은 목록의 모든 개체를 참조할 수 있습니다. 예를 들어 세 개의 개체가 있는 경우 같은 복합 서식 문자열 `{1} {0} {2}`을 지정하여 두 번째, 첫 번째 및 세 번째 개체의 서식을 지정할 수 있습니다. 형식 항목에서 참조되지 않는 개체는 무시됩니다. 매개 변수 지정자가 개체 목록의 범위를 벗어난 항목을 지정하는 경우 런타임에 A FormatException 가 throw됩니다.

너비 구성 요소

선택적 `width` 구성 요소는 기본 형식 필드 너비를 나타내는 부호 있는 정수입니다. 값 `width` 이 서식이 지정된 문자열 `width` 의 길이보다 작으면 무시되고 서식이 지정된 문자열의 길이가 필드 너비로 사용됩니다. `width`가 양수이면 필드의 서식이 지정된 데이터는 오른쪽 맞춤이고, `width` 가 음수이면 왼쪽 맞춤입니다. 패딩이 필요한 경우 공백이 사용됩니다. 지정된 경우 `width` 쉼표가 필요합니다.

다음 예제에서는 두 개의 배열을 정의합니다. 하나는 직원의 이름을 포함하고 다른 하나는 2주 동안 근무한 시간을 포함합니다. 복합 형식 문자열은 20자 필드의 이름을 왼쪽으로 정렬하고 5자 필드에서 해당 시간을 오른쪽에 맞춥니다. "N1" 표준 형식 문자열은 소수 자릿수가 한 자리인 시간의 서식을 지정합니다.

C#

```
string[] names = { "Adam", "Bridgette", "Carla", "Daniel",
                   "Ebenezer", "Francine", "George" };
decimal[] hours = { 40, 6.667m, 40.39m, 82,
                    40.333m, 80, 16.75m };

Console.WriteLine("{0,-20} {1,5}\n", "Name", "Hours");

for (int counter = 0; counter < names.Length; counter++)
    Console.WriteLine("{0,-20} {1,5:N1}", names[counter], hours[counter]);

// The example displays the following output:
//       Name                  Hours
//
//       Adam                   40.0
//       Bridgette               6.7
//       Carla                  40.4
//       Daniel                 82.0
//       Ebenezer               40.3
//       Francine               80.0
//       George                 16.8
```

문자열 구성 요소 서식 지정

선택적 `formatString` 구성 요소는 서식이 지정되는 개체의 형식에 적합한 형식 문자열입니다. 다음을 지정할 수 있습니다.

- 해당 개체가 숫자 값인 경우 표준 또는 사용자 지정 숫자 서식 문자열입니다.
- 해당 개체가 DateTime 개체인 경우 표준 또는 사용자 지정 날짜 및 시간 형식 문자열입니다.
- 해당 개체가 열거형 값인 경우, 열거형 형식 문자열입니다.

지정하지 않으면 `formatString` 숫자, 날짜 및 시간 또는 열거형 형식에 대한 일반("G") 형식 지정자가 사용됩니다. 지정된 경우 `formatString` 콜론이 필요합니다.

다음 표에서는 미리 정의된 형식 문자열 집합을 지원하는 .NET 클래스 라이브러리의 형식 또는 형식 범주를 나열하고 지원되는 형식 문자열을 나열하는 아티클에 대한 링크를 제공합니다. 문자열 서식 지정은 기존의 모든 형식에 대해 새 형식 문자열을 정의하고 애플리케이션 정의 형식에서 지원하는 형식 문자열 집합을 정의할 수 있게 해주는 확장 가능한 메커니즘입니다.

자세한 내용은 IFormattable 및 ICustomFormatter 인터페이스 문서를 참조하세요.

테이블 확장

형식 또는 형식 범주	보십시오
날짜 및 시간 형식(DateTime, DateTimeOffset)	표준 날짜 및 시간 형식 문자열 사용자 지정 날짜 및 시간 형식 문자열
열거형 형식 (System.Enum에서 파생된 모든 형식)	열거형 형식 문자열
숫자 형식(BigInteger,,Byte, Decimal, Double, Int16Int32, Int64SByte, SingleUInt16,) UInt32UInt64	표준 숫자 형식 문자열 사용자 지정 숫자 형식 문자열
Guid	Guid.ToString(String)
TimeSpan	표준 TimeSpan 서식 문자열 사용자 지정 TimeSpan 서식 문자열

중괄호 이스케이프

여는 중괄호와 닫는 중괄호는 형식 항목을 시작하고 끝내는 것으로 해석됩니다. 리터럴 여는 중괄호 또는 닫는 중괄호를 표시하려면 이스케이프 시퀀스를 사용해야 합니다. 한 개의 여는 중괄호() 또는 두 개의 닫는 중괄호(`{{` `{`)를 표시하도록 고정 텍스트에 두 개의 여는 중괄호(`}}`)를 지정하여 하나의 닫는 중괄호(`}`)를 표시합니다.

.NET과 .NET Framework에서는 형식 항목을 포함한 이스케이프된 중괄호가 서로 다르게 구문 분석됩니다.

닷넷

중괄호는 서식 항목 주위에서 이스케이프할 수 있습니다. 예를 들어 여는 중괄호, 소수로 서식이 지정된 숫자 값 및 닫는 중괄호를 표시하기 위한 서식 항목을 `{{{0:D}}}` 고려합니다. 형식 항목은 다음과 같은 방식으로 해석됩니다.

1. 처음 두 개의 여는 중괄호(`{{`)는 이스케이프되고 하나의 여는 중괄호를 생성합니다.
2. 다음 세 문자(`{0:`)는 형식 항목의 시작으로 해석됩니다.
3. 다음 문자(`D`)는 10진수 표준 숫자 형식 지정자로 해석됩니다.
4. 다음 중괄호(`}`)는 형식 항목의 끝으로 해석됩니다.
5. 마지막 두 개의 닫는 중괄호는 이스케이프되고 하나의 닫는 중괄호를 생성합니다.
6. 표시되는 최종 결과는 리터럴 문자열 `{6324}` 입니다.

C#

```
int value = 6324;
string output = string.Format("{{{0:D}}}", value);

Console.WriteLine(output);
// The example displays the following output:
//       {6324}
```

.NET Framework

형식 항목의 중괄호는 만나는 순서대로 순차적으로 해석됩니다. 중첩된 중괄호 해석은 지원되지 않습니다.

이탈된 중괄호가 해석되는 방식은 예기치 않은 결과를 초래할 수 있습니다. 예를 들어 여는 중괄호, 소수로 서식이 지정된 숫자 값 및 닫는 중괄호를 표시하기 위한 서식 항목을 `{{{0:D}}}` 고려합니다. 그러나 형식 항목은 다음과 같은 방식으로 해석됩니다.

1. 처음 두 개의 여는 중괄호(`{{`)는 이스케이프되고 하나의 여는 중괄호를 생성합니다.
2. 다음 세 문자(`{0:`)는 형식 항목의 시작으로 해석됩니다.
3. 다음 문자(`D`)는 10진수 표준 숫자 형식 지정자로 해석되지만 다음 두 개의 이스케이프된 중괄호(`}}`)는 단일 중괄호를 생성합니다. 결과 문자열(`D}`)은 표준 숫자 형식 지정자가 아니므로 결과 문자열은 리터럴 문자열을 표시하는 것을 의미하는 사용자 지정 형식 문자열 `D}`로 해석됩니다.
4. 마지막 중괄호(`}`)는 형식 항목의 끝으로 해석됩니다.
5. 표시되는 최종 결과는 리터럴 문자열 `{D}` 입니다. 서식을 지정할 숫자 값이 표시되지 않습니다.

C#

```
int value = 6324;
string output = string.Format("{{{0:D}}}",
                              value);
Console.WriteLine(output);

// The example displays the following output:
//       {D}
```

이스케이프된 중괄호와 서식 항목을 잘못 해석하지 않도록 코드를 작성하는 방법 중 하나는 중괄호와 서식 항목을 각각 따로 포맷팅하는 것입니다. 즉, 첫 번째 형식 처리에서 리터럴 여는 중괄호를 표시합니다. 다음 작업에서 서식 항목의 결과를 표시하고, 최종 작업에서 리터럴한 닫는 중괄호를 표시합니다. 다음 예제에서는 이 방법을 보여 줍니다.

C#

```
int value = 6324;
string output = string.Format("{0}{1:D}{2}",
                              "{", value, "}");
Console.WriteLine(output);

// The example displays the following output:
//       {6324}
```

처리 순서

복합 서식 지정 메서드 호출에 IFormatProvider이 아닌 값을 가진 `null` 인수가 포함된 경우, 런타임은 IFormatProvider.GetFormat 구현을 요청하기 위해 ICustomFormatter 메서드를 호출합니다. 메서드가 ICustomFormatter 구현을 반환할 수 있는 경우, 복합 서식 지정 메서드 호출 중에 캐시됩니다.

형식 항목에 해당하는 매개 변수 목록의 각 값은 다음과 같이 문자열로 변환됩니다.

1. 서식을 지정할 값이 `null`면 빈 문자열 String.Empty 이 반환됩니다.

2. ICustomFormatter 구현을 사용할 수 있는 경우 런타임은 Format 메서드를 호출합니다. 런타임은 형식 항목의 `formatString` 값(또는 `null` 없는 경우)을 메서드에 전달합니다. 또한 런타임은 구현을 IFormatProvider 메서드에 전달합니다. 메서드 호출이 ICustomFormatter.Format 반환 `null`되면 실행이 다음 단계로 진행됩니다. 그렇지 않으면 호출 결과가 ICustomFormatter.Format 반환됩니다.

3. 값이 인터페이스를 구현하는 IFormattable 경우 인터페이스의 ToString(String, IFormatProvider) 메서드가 호출됩니다. 형식 항목 `formatString` 에 값이 있으면 메서드에 값이 전달됩니다. 그렇지 않으면 `null` 전달됩니다. 인수는 IFormatProvider 다음과 같이 결정됩니다.

- 숫자 값의 경우 null IFormatProvider 이 아닌 인수가 있는 복합 서식 지정 메서드가 호출되면 런타임은 해당 NumberFormatInfo 메서드에서 개체를 IFormatProvider.GetFormat 요청합니다. 인수를 제공할 수 없거나, 인수 값이 `null` 이거나, 복합 서식 지정 메서드가 IFormatProvider 매개 변수를 갖고 있지 않은 경우, 현재 문화권의 NumberFormatInfo 개체가 사용됩니다.

- 날짜 및 시간 값의 경우 null IFormatProvider 이 아닌 인수가 있는 복합 서식 지정 메서드가 호출되면 런타임은 해당 DateTimeFormatInfo 메서드에서 개체를 IFormatProvider.GetFormat 요청합니다. 다음 상황에서는 현재 문화권 DateTimeFormatInfo 의 개체가 대신 사용됩니다.
  - 메서드가 IFormatProvider.GetFormat 개체를 제공할 DateTimeFormatInfo 수 없습니다.
  - 인수의 값은 .입니다 `null`.
  - 복합 서식 지정 메서드에는 매개 변수가 IFormatProvider 없습니다.

- 다른 형식의 개체의 경우 복합 서식 메서드를 인수로 IFormatProvider 호출하면 해당 값이 구현에 IFormattable.ToString 직접 전달됩니다. 그렇지 않으면 `null` 구현에 IFormattable.ToString 전달됩니다.

4. 타입의 매개 변수가 없는 `ToString` 메서드가 호출되며, 이 메서드는 Object.ToString()를 재정의하거나 기본 클래스의 동작을 상속합니다. 이 경우 형식 항목의 구성 요소에서 지정한 `formatString` 형식 문자열(있는 경우)은 무시됩니다.

맞춤은 이전 단계가 수행된 후에 적용됩니다.

코드 예제

다음 예제에서는 복합 서식을 사용하여 만든 문자열 하나와 개체의 `ToString` 메서드를 사용하여 만든 문자열을 보여줍니다. 두 형식의 서식은 동일한 결과를 생성합니다.

C#

```
string formatString1 = string.Format("{0:dddd MMMM}", DateTime.Now);
string formatString2 = DateTime.Now.ToString("dddd MMMM");
```

현재 날짜가 5월의 목요일이라고 가정하면 이전 예제의 두 문자열 값은 `Thursday May` 미국 영어 문화권에 있습니다.

Console.WriteLine는 .와 동일한 기능을 노출합니다.String.Format 두 메서드 String.Format 의 유일한 차이점은 결과를 문자열로 반환하고 Console.WriteLine 결과를 개체와 Console 연결된 출력 스트림에 쓰는 것입니다. 다음 예제에서는 이 메서드를 Console.WriteLine 사용하여 값의 `myNumber` 형식을 통화 값으로 지정합니다.

C#

```
int myNumber = 100;
Console.WriteLine($"{myNumber:C}");

// The example displays the following output
// if en-US is the current culture:
//        $100.00
```

다음 예제는 여러 개체 서식 지정과, 하나의 개체를 두 가지 방식으로 서식 지정하는 방법을 보여 줍니다.

C#

```
string myName = "Fred";
Console.WriteLine(string.Format("Name = {0}, hours = {1:hh}, minutes = {1:mm}",
                                myName, DateTime.Now));

// Depending on the current time, the example displays output like the following:
//        Name = Fred, hours = 11, minutes = 30
```

다음 예제에서는 서식에 너비를 사용하는 방법을 보여 줍니다. 서식이 지정된 인수는 세로 막대 문자(`|`)사이에 배치되어 결과 맞춤을 강조 표시합니다.

C#

```
string firstName = "Fred";
string lastName = "Opals";
int myNumber = 100;

string formatFirstName = string.Format("First Name = |{0,10}|", firstName);
string formatLastName = string.Format("Last Name =  |{0,10}|", lastName);
string formatPrice = string.Format("Price =      |{0,10:C}|", myNumber);
Console.WriteLine(formatFirstName);
Console.WriteLine(formatLastName);
Console.WriteLine(formatPrice);
Console.WriteLine();

formatFirstName = string.Format("First Name = |{0,-10}|", firstName);
formatLastName = string.Format("Last Name =  |{0,-10}|", lastName);
formatPrice = string.Format("Price =      |{0,-10:C}|", myNumber);
Console.WriteLine(formatFirstName);
Console.WriteLine(formatLastName);
Console.WriteLine(formatPrice);

// The example displays the following output on a system whose current
// culture is en-US:
//     First Name = |      Fred|
//     Last Name =  |     Opals|
//     Price =      |   $100.00|
//
```

```
//     First Name = |Fred      |
//     Last Name =  |Opals     |
//     Price =      |$100.00   |
```

참고하십시오

- WriteLine
- String.Format
- 문자열 보간(C#)
- 문자열 보간(Visual Basic)
- 형식 유형
- 표준 숫자 형식 문자열
- 사용자 지정 숫자 서식 문자열
- 표준 날짜 및 시간 형식 문자열
- 사용자 지정 날짜 및 시간 형식 문자열
- 표준 TimeSpan 형식 문자열
- 사용자 지정 TimeSpan 형식 문자열
- 열거형 형식 문자열

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

방법: 앞에 오는 0으로 숫자 채우기

아티클 • 2025. 04. 22.

전체 자릿수 지정자와 함께 "D" 표준 숫자 형식 문자열 을 사용하여 앞에 오는 0을 정수에 추가할 수 있습니다. 사용자 지정 숫자 형식 문자열을 사용하여 정수 및 부동 소수점 숫자 모두에 선행 0을 추가할 수 있습니다. 이 문서에서는 두 메서드를 사용하여 앞에 오는 0으로 숫자를 패딩하는 방법을 보여 줍니다.

앞에 오는 0이 있는 정수의 길이를 특정 길이로 채점하려면

1. 정수 값을 표시할 최소 자릿수를 결정합니다. 이 숫자에 선행 숫자를 포함합니다.
2. 정수를 10진수 또는 16진수 값으로 표시할지 여부를 결정합니다.
    - 정수를 10진수 값으로 표시하려면 해당 메서드를 `ToString(String)` 호출하고 문자열 "D*n*"을 매개 변수 값 `format` 으로 전달합니다. 여기서 *n* 은 문자열의 최소 길이를 나타냅니다.
    - 정수를 16진수 값으로 표시하려면 해당 메서드를 `ToString(String)` 호출하고 문자열 "X*n*"을 형식 매개 변수의 값으로 전달합니다. 여기서 *n* 은 문자열의 최소 길이를 나타냅니다.

C# 및 Visual Basic 모두에서 보간된 문자열에서 형식 문자열을 사용할 수도 있습니다. 또는 복합 서식을 사용하는 등의 String.Format 메서드를 호출할 Console.WriteLine 수 있습니다.

다음은 서식이 지정된 숫자의 총 길이가 8자 이상이 되도록 앞에 오는 0으로 여러 정수 값의 서식을 지정하는 예제입니다.

C#

```
byte byteValue = 254;
short shortValue = 10342;
int intValue = 1023983;
long lngValue = 6985321;
ulong ulngValue = UInt64.MaxValue;

// Display integer values by calling the ToString method.
Console.WriteLine("{0,22} {1,22}", byteValue.ToString("D8"),
byteValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", shortValue.ToString("D8"),
shortValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", intValue.ToString("D8"),
intValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", lngValue.ToString("D8"),
lngValue.ToString("X8"));
```

```
Console.WriteLine("{0,22} {1,22}", ulngValue.ToString("D8"),
ulngValue.ToString("X8"));
Console.WriteLine();

// Display the same integer values by using composite formatting.
Console.WriteLine("{0,22:D8} {0,22:X8}", byteValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", shortValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", intValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", lngValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", ulngValue);
// The example displays the following output:
//                       00000254               000000FE
//                       00010342               00002866
//                       01023983               000F9FEF
//                       06985321               006A9669
//         18446744073709551615       FFFFFFFFFFFFFFFF
//
//                       00000254               000000FE
//                       00010342               00002866
//                       01023983               000F9FEF
//                       06985321               006A9669
//         18446744073709551615       FFFFFFFFFFFFFFFF
//         18446744073709551615       FFFFFFFFFFFFFFFF
```

특정 수의 선행 0으로 정수를 채우려면

1. 정수 값을 표시할 선행 0 수를 결정합니다.
2. 정수를 10진수 또는 16진수 값으로 표시할지 여부를 결정합니다.
   - 10진수 값으로 서식을 지정하려면 "D" 표준 형식 지정자가 필요합니다.
   - 16진수 값으로 서식을 지정하려면 "X" 표준 형식 지정자가 필요합니다.
3. 정수 값 또는 `ToString("X").Length` 메서드를 호출하여패딩되지 않은 숫자 문자열의 `ToString("D").Length` 길이를 결정합니다.
4. 패딩되지 않은 숫자 문자열의 길이에 서식이 지정된 문자열에서 원하는 선행 0의 수를 추가합니다. 결과는 패딩된 문자열의 총 길이입니다.
5. 정수 값의 `ToString(String)` 메서드를 호출하고 10진수 문자열의 경우 "D*n*", 16진수 문자열의 경우 "X*n*" 문자열을 전달합니다. 여기서 *n* 은 패딩된 문자열의 총 길이를 나타냅니다. 복합 서식 지정을 지원하는 메서드에서 "D*n*" 또는 "X*n*" 형식 문자열을 사용할 수도 있습니다.

다음 예제에서는 앞에 오는 0이 5개인 정수 값을 패딩합니다.

```
C#
```

```
int value = 160934;
int decimalLength = value.ToString("D").Length + 5;
int hexLength = value.ToString("X").Length + 5;
Console.WriteLine(value.ToString("D" + decimalLength.ToString()));
Console.WriteLine(value.ToString("X" + hexLength.ToString()));
// The example displays the following output:
//       00000160934
//       00000274A6
```

숫자 값을 특정 길이로 맞추기 위해 앞에 0을 추가하려면

1. 숫자의 문자열 표현을 사용할 소수 자릿수를 결정합니다. 이 숫자의 총 자릿수에 선행 0을 포함시키세요.

2. 0 자리 표시자("0")를 사용하여 최소 0 수를 나타내는 사용자 지정 숫자 형식 문자열을 정의합니다.

3. 숫자의 `ToString(String)` 메서드를 호출하고 사용자 지정 형식 문자열을 전달합니다. 문자열 보간 또는 복합 서식 지정을 지원하는 메서드와 함께 사용자 지정 서식 문자열을 사용할 수도 있습니다.

다음 예제에서는 앞에 오는 0을 사용하여 여러 숫자 값의 서식을 지정합니다. 따라서 서식이 지정된 숫자의 총 길이는 소수점 왼쪽에 있는 8자리 이상입니다.

C#

```
string fmt = "00000000.##";
int intValue = 1053240;
decimal decValue = 103932.52m;
float sngValue = 1549230.10873992f;
double dblValue = 9034521202.93217412;

// Display the numbers using the ToString method.
Console.WriteLine(intValue.ToString(fmt));
Console.WriteLine(decValue.ToString(fmt));
Console.WriteLine(sngValue.ToString(fmt));
Console.WriteLine(dblValue.ToString(fmt));
Console.WriteLine();

// Display the numbers using composite formatting.
string formatString = " {0,15:" + fmt + "}";
Console.WriteLine(formatString, intValue);
Console.WriteLine(formatString, decValue);
Console.WriteLine(formatString, sngValue);
Console.WriteLine(formatString, dblValue);
// The example displays the following output:
```

```
//       01053240
//       00103932.52
//       01549230
//       9034521202.93
//
//               01053240
//            00103932.52
//               01549230
//          9034521202.93
```

숫자 값을 특정 수의 선행 0으로 채비하려면

1. 숫자 값에 몇 개의 선행 0을 사용할지 결정하십시오.
2. 패딩되지 않은 숫자 문자열에서 소수점 왼쪽의 자릿수를 결정합니다.
   a. 숫자의 문자열 표현에 소수점 기호가 포함되는지 여부를 확인합니다.
   b. 소수점 기호를 포함하는 경우 소수점 왼쪽의 문자 수를 결정합니다. 소수점 기호가 포함되지 않은 경우 문자열의 길이를 결정합니다.
3. 다음을 사용하는 사용자 지정 형식 문자열을 만듭니다.
   - 문자열에서 선행 0을 나타내기 위한 0 자리 표시자("0")입니다.
   - 기본 문자열에서 각 숫자를 나타내기 위해 숫자 자리 표시자 "#" 또는 0 자리 표시자를 사용합니다.
4. 사용자 지정 서식 문자열을 숫자의 `ToString(String)` 메서드 또는 복합 서식 지정을 지원하는 메서드에 매개 변수로 제공합니다.

다음 예제에서는 앞에 오는 0이 5개인 두 Double 값을 패딩합니다.

C#

```
double[] dblValues = { 9034521202.93217412, 9034521202 };
foreach (double dblValue in dblValues)
{
   string decSeparator =
System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator;
   string fmt, formatString;

   if (dblValue.ToString().Contains(decSeparator))
   {
      int digits = dblValue.ToString().IndexOf(decSeparator);
      fmt = new String('0', 5) + new String('#', digits) + ".##";
   }
   else
   {
      fmt = new String('0', dblValue.ToString().Length);
```

```
    }
    formatString = "{0,20:" + fmt + "}";

    Console.WriteLine(dblValue.ToString(fmt));
    Console.WriteLine(formatString, dblValue);
}
// The example displays the following output:
//       000009034521202.93
//         000009034521202.93
//       9034521202
//                 9034521202
```

참고하십시오

- 사용자 지정 숫자 형식 문자열
- 표준 숫자 형식 문자열
- 복합 형식 지정

방법: 특정 날짜에서 요일 주출

.NET을 사용하면 특정 날짜의 요일을 쉽게 확인하고 특정 날짜의 지역화된 요일 이름을 표시할 수 있습니다. 특정 날짜에 해당하는 요일을 나타내는 열거형 값은 DayOfWeek 속성 또는 DayOfWeek 속성에서 확인할 수 있습니다. 반면, 요일 이름 검색은 날짜 및 시간 값의 `ToString` 메서드 또는 String.Format와 같은 메서드를 호출하여 수행할 수 있는 서식 지정 작업입니다. 이 문서에서는 이러한 서식 지정 작업을 수행하는 방법을 보여 줍니다.

요일을 나타내는 숫자 추출

1. 정적(정적) DateTime.Parse 또는 DateTimeOffset.Parse 메서드를 사용하여 날짜의 문자열 표현을 DateTime 또는 DateTimeOffset 값으로 변환합니다.

2. DateTime.DayOfWeek 또는 DateTimeOffset.DayOfWeek 속성을 사용하여 값을 검색하면 요일을 나타내는 DayOfWeek을 얻을 수 있습니다.

3. 필요한 경우 C#으로 캐스팅하거나(Visual Basic에서) DayOfWeek 값을 정수로 변환합니다.

다음 예제에서는 특정 날짜의 요일을 나타내는 정수입니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine((int) dateValue.DayOfWeek);
   }
}
// The example displays the following output:
//       3
```

축약된 평일 이름 추출

1. 정적 DateTime.Parse 또는 DateTimeOffset.Parse 메서드를 사용하여 날짜의 문자열 표현을 DateTime 또는 DateTimeOffset 값으로 변환합니다.

2. 현재 문화권 또는 특정 문화권의 축약된 평일 이름을 추출할 수 있습니다.

   a. 현재 문화권의 축약된 요일 이름을 추출하려면 날짜 및 시간 값 DateTime.ToString(String) 또는 DateTimeOffset.ToString(String) 인스턴스 메서드를 호

출하고 문자열 `ddd` 을 `format` 매개 변수로 전달합니다. 다음 예제에서는 메서드에 대한 호출을 보여 줍니다 ToString(String) .

C#

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("ddd"));
   }
}
// The example displays the following output:
 //       Wed
```

b. 특정 문화권의 축약된 평일 이름을 추출하려면 날짜 및 시간 값 DateTime.ToString(String, IFormatProvider) 또는 DateTimeOffset.ToString(String, IFormatProvider) 인스턴스 메서드를 호출합니다. 문자열 `ddd` 을 매개 변수로 전달합니다 `format` . `provider` 매개 변수로 검색할 요일 이름이 포함된 문화권을 나타내는 CultureInfo 또는 DateTimeFormatInfo 개체를 전달합니다. 다음 코드는 fr-FR 문화를 나타내는 CultureInfo 개체를 사용하여 ToString(String, IFormatProvider) 메서드를 호출하는 방법을 보여 줍니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("ddd",
                        new CultureInfo("fr-FR")));
   }
}
// The example displays the following output:
//       mer.
```

전체 평일 이름 추출

1. 정적 DateTime.Parse 또는 DateTimeOffset.Parse 메서드를 사용하여 날짜의 문자열 표현을 DateTime 값 또는 DateTimeOffset 값으로 변환합니다.

2. 현재 문화권 또는 특정 문화권의 전체 평일 이름을 추출할 수 있습니다.

   a. 현재 문화권의 평일 이름을 추출하려면 날짜 및 시간 값 DateTime.ToString(String) 또는 DateTimeOffset.ToString(String) 인스턴스 메서드를 호출하고 문자열 `dddd` 을 `format` 매개 변수로 전달합니다. 다음 예제에서는 메서드에 대한 호출을 보여 줍니다 ToString(String) .

   C#

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("dddd"));
   }
}
// The example displays the following output:
//       Wednesday
```

   b. 특정 문화권의 평일 이름을 추출하려면 날짜 및 시간 값 DateTime.ToString(String, IFormatProvider) 또는 DateTimeOffset.ToString(String, IFormatProvider) 인스턴스 메서드를 호출합니다. 문자열 `dddd` 을 매개 변수로 전달합니다 `format` . CultureInfo 나 DateTimeFormatInfo 중 하나의 객체를 `provider` 매개 변수로 전달하여 검색할 평일 이름이 있는 문화권을 나타내도록 합니다. 다음 코드에서는 es-ES 문화권을 나타내는 개체를 사용하여 ToString(String, IFormatProvider) 메서드를 호출 CultureInfo 하는 방법을 보여 줍니다.

   C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("dddd",
                        new CultureInfo("es-ES")));
   }
}
// The example displays the following output:
//       miércoles.
```

예시

다음 예제는 특정 날짜의 요일을 나타내는 숫자를 검색하기 위해 DateTime.DayOfWeek 및 DateTimeOffset.DayOfWeek 속성을 호출하는 방법을 보여줍니다. DateTime.ToString 및 DateTimeOffset.ToString 메서드를 호출하여 약식 평일 이름과 전체 평일 이름을 추출하는 기능도 포함되어 있습니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string dateString = "6/11/2007";
      DateTime dateValue;
      DateTimeOffset dateOffsetValue;

      try
      {
         DateTimeFormatInfo dateTimeFormats;
         // Convert date representation to a date value
         dateValue = DateTime.Parse(dateString, CultureInfo.InvariantCulture);
         dateOffsetValue = new DateTimeOffset(dateValue,
                                   TimeZoneInfo.Local.GetUtcOffset(dateValue));

         // Convert date representation to a number indicating the day of week
         Console.WriteLine((int) dateValue.DayOfWeek);
         Console.WriteLine((int) dateOffsetValue.DayOfWeek);

         // Display abbreviated weekday name using current culture
         Console.WriteLine(dateValue.ToString("ddd"));
         Console.WriteLine(dateOffsetValue.ToString("ddd"));

         // Display full weekday name using current culture
         Console.WriteLine(dateValue.ToString("dddd"));
         Console.WriteLine(dateOffsetValue.ToString("dddd"));

         // Display abbreviated weekday name for de-DE culture
         Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("de-DE")));
         Console.WriteLine(dateOffsetValue.ToString("ddd",
                                                    new CultureInfo("de-DE")));

         // Display abbreviated weekday name with de-DE DateTimeFormatInfo object
         dateTimeFormats = new CultureInfo("de-DE").DateTimeFormat;
         Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats));
         Console.WriteLine(dateOffsetValue.ToString("ddd", dateTimeFormats));

         // Display full weekday name for fr-FR culture
         Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("fr-FR")));
```

```
         Console.WriteLine(dateOffsetValue.ToString("ddd",
                                                    new CultureInfo("fr-FR")));

         // Display abbreviated weekday name with fr-FR DateTimeFormatInfo object
         dateTimeFormats = new CultureInfo("fr-FR").DateTimeFormat;
         Console.WriteLine(dateValue.ToString("dddd", dateTimeFormats));
         Console.WriteLine(dateOffsetValue.ToString("dddd", dateTimeFormats));
      }
      catch (FormatException)
      {
         Console.WriteLine($"Unable to convert {dateString} to a date.");
      }
   }
}
// The example displays the following output:
//       1
//       1
//       Mon
//       Mon
//       Monday
//       Monday
//       Mo
//       Mo
//       Mo
//       Mo
//       lun.
//       lun.
//       lundi
//       lundi
```

개별 언어는 .NET에서 제공하는 기능을 중복하거나 보완하는 기능을 제공할 수 있습니다. 예를 들어 Visual Basic에는 다음 두 가지 함수가 포함됩니다.

- `Weekday` - 특정 날짜의 요일을 나타내는 숫자를 반환합니다. 첫 번째 날의 서수 값을 1로 간주하는 반면, DateTime.DayOfWeek 속성은 이를 0으로 간주합니다.
- `WeekdayName` 현재 문화권에서 특정 요일 번호에 해당하는 주의 이름을 반환하는 입니다.

다음 예제에서는 Visual Basic `Weekday` 및 `WeekdayName` 함수의 사용을 보여 줍니다.

VB

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#

        ' Get weekday number using Visual Basic Weekday function
        Console.WriteLine(Weekday(dateValue))                ' Displays 4
        ' Compare with .NET DateTime.DayOfWeek property
```

```
        Console.WriteLine(dateValue.DayOfWeek)                ' Displays 3

        ' Get weekday name using Weekday and WeekdayName functions
        Console.WriteLine(WeekdayName(Weekday(dateValue)))    ' Displays Wednesday

        ' Change culture to de-DE
        Dim originalCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
        Thread.CurrentThread.CurrentCulture = New CultureInfo("de-DE")
        ' Get weekday name using Weekday and WeekdayName functions
        Console.WriteLine(WeekdayName(Weekday(dateValue)))    ' Displays Donnerstag

        ' Restore original culture
        Thread.CurrentThread.CurrentCulture = originalCulture
    End Sub
End Module
```

속성에서 반환된 DateTime.DayOfWeek 값을 사용하여 특정 날짜의 평일 이름을 검색할 수도 있습니다. 이 프로세스를 수행하려면 속성에서 반환된 값의 ToString 메서드 호출만 필요합니다. 그러나 이 기술은 다음 예제와 같이 현재 문화권에 대한 지역화된 평일 이름을 생성하지 않습니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Change current culture to fr-FR
      CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

      DateTime dateValue = new DateTime(2008, 6, 11);
      // Display the DayOfWeek string representation
      Console.WriteLine(dateValue.DayOfWeek.ToString());
      // Restore original current culture
      Thread.CurrentThread.CurrentCulture = originalCulture;
   }
}
// The example displays the following output:
//       Wednesday
```

참고하십시오

- 표준 날짜 및 시간 형식 문자열
- 사용자 지정 날짜 및 시간 형식 문자열

Last updated on 2026. 03. 31.

방법: 사용자 지정 숫자 형식 공급 기업 정의 및 사용

아티클 • 2023. 05. 10.

.NET에서는 숫자 값의 문자열 표현을 광범위하게 제어할 수 있습니다. 숫자 값의 형식을 사용자 지정하기 위한 다음과 같은 기능을 지원합니다.

- 숫자를 해당 문자열 표현으로 변환하기 위한 미리 정의된 형식 집합을 제공하는 표준 숫자 형식 문자열입니다. `format` 매개 변수가 있는 숫자 서식 지정 메서드(예: Decimal.ToString(String))와 함께 사용할 수 있습니다. 자세한 내용은 표준 숫자 형식 문자열을 참조하세요.

- 함께 결합되어 사용자 지정 숫자 형식 지정자를 정의할 수 있는 기호 집합을 제공하는 사용자 지정 숫자 형식 문자열입니다. 또한 `format` 매개 변수가 있는 숫자 서식 지정 메서드(예: Decimal.ToString(String))와 함께 사용할 수도 있습니다. 자세한 내용은 사용자 지정 숫자 형식 문자열을 참조하세요.

- 숫자 값의 문자열 표현을 표시하는 데 사용되는 기호 및 형식 패턴을 정의하는 사용자 지정 CultureInfo 또는 NumberFormatInfo 개체입니다. `provider` 매개 변수가 있는 숫자 서식 지정 메서드(예: ToString)와 함께 사용할 수 있습니다. 일반적으로 `provider` 매개 변수는 문화권별 서식 지정에 사용됩니다.

애플리케이션에서 형식이 지정된 계정 번호, ID 번호 또는 우편 번호를 표시해야 하는 경우와 같이 이 세 가지 방법이 부적절한 경우도 있습니다. .NET에서는 CultureInfo 및 NumberFormatInfo 개체가 아닌 서식 지정 개체를 정의하여 숫자 값의 서식 지정 방법을 결정할 수도 있습니다. 이 항목에서는 이러한 개체를 구현하기 위한 단계별 지침을 제공하고 전화 번호 형식을 지정하는 예제를 제공합니다.

사용자 지정 형식 공급자 정의

1. IFormatProvider 및 ICustomFormatter 인터페이스를 구현하는 클래스를 정의합니다.

2. IFormatProvider.GetFormat 메서드를 구현합니다. GetFormat은 서식 지정 메서드(예: String.Format(IFormatProvider, String, Object[]) 메서드)가 실제로 사용자 지정 서식 지정을 수행하는 개체를 검색하기 위해 호출하는 콜백 메서드입니다. 일반적인 GetFormat 구현에서는 다음 작업을 수행합니다.

   a. 메서드 매개 변수로 전달되는 Type 개체가 ICustomFormatter 인터페이스를 나타내는지 여부를 결정합니다.

b. 매개 변수가 ICustomFormatter 인터페이스를 나타내지 않는 경우 GetFormat은 사용자 지정 서식 지정을 제공하는 ICustomFormatter 인터페이스를 구현하는 개체를 반환합니다. 일반적으로 사용자 지정 형식 지정 개체 자체가 반환됩니다.

c. 매개 변수가 ICustomFormatter 인터페이스를 나타내지 않는 경우 GetFormat은 `null`을 반환합니다.

3. Format 메서드를 구현합니다. 이 메서드는 String.Format(IFormatProvider, String, Object[]) 메서드에 의해 호출되며 숫자의 문자열 표현을 반환합니다. 일반적으로 메서드를 구현하는 과정은 다음과 같습니다.

a. 필요에 따라 `provider` 매개 변수를 검사하여 메서드가 서식 지정 서비스를 제공하기에 적합한지 확인합니다. IFormatProvider와 ICustomFormatter를 둘 다 구현하는 서식 지정 개체의 경우 `provider` 매개 변수가 현재 서식 지정 개체와 같은지 테스트해야 합니다.

b. 형식 지정 개체가 사용자 지정 형식 지정자를 지원해야 하는지 여부를 결정합니다. 예를 들어 "N" 형식 지정자는 미국 전화 번호가 NANP 형식으로, "I" 형식 지정자는 ITU-T 권장 E.123 형식으로 출력되어야 함을 나타낼 수 있습니다. 형식 지정자가 사용된 경우 메서드에서 특정 형식 지정자를 처리해야 합니다. 지정자는 `format` 매개 변수의 메서드에 전달됩니다. 지정자가 없는 경우 `format` 매개 변수의 값은 String.Empty입니다.

c. 메서드에 `arg` 매개 변수로 전달된 숫자 값을 검색합니다. 문자열 표현으로 변환하는 데 필요한 조작을 수행합니다.

d. `arg` 매개 변수의 문자열 표현을 반환합니다.

사용자 지정 숫자 형식 지정 개체 사용

1. 사용자 지정 형식 지정 클래스의 새 인스턴스를 만듭니다.

2. String.Format(IFormatProvider, String, Object[]) 서식 지정 메서드를 호출하여 사용자 지정 서식 지정 개체, 형식 지정자(또는 지정자를 사용하지 않는 경우 String.Empty) 및 서식을 지정할 숫자 값을 전달합니다.

예제

다음 예제에서는 미국 전화 번호를 나타내는 숫자를 NANP 또는 E.123 형식으로 변환하는 `TelephoneFormatter`라는 사용자 지정 숫자 형식 공급자를 정의합니다. 메서드는 두 가지 형식 지정자 "N"(NANP 형식 출력)과 "I"(국제 E.123 형식 출력)를 처리합니다.

```
```

C#

```
using System;
using System.Globalization;

public class TelephoneFormatter : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(string format, object arg, IFormatProvider
formatProvider)
   {
      // Check whether this is an appropriate callback
      if (! this.Equals(formatProvider))
         return null;

      // Set default format specifier
      if (string.IsNullOrEmpty(format))
         format = "N";

      string numericString = arg.ToString();

      if (format == "N")
      {
         if (numericString.Length <= 4)
            return numericString;
         else if (numericString.Length == 7)
            return numericString.Substring(0, 3) + "-" +
numericString.Substring(3, 4);
         else if (numericString.Length == 10)
               return "(" + numericString.Substring(0, 3) + ") " +
                      numericString.Substring(3, 3) + "-" +
numericString.Substring(6);
         else
            throw new FormatException(
                      string.Format("'{0}' cannot be used to format {1}.",
                                    format, arg.ToString()));
      }
      else if (format == "I")
      {
         if (numericString.Length < 10)
            throw new FormatException(string.Format("{0} does not have 10
digits.", arg.ToString()));
         else
            numericString = "+1 " + numericString.Substring(0, 3) + " " +
numericString.Substring(3, 3) + " " + numericString.Substring(6);
      }
      else
```

```
      {
         throw new FormatException(string.Format("The {0} format specifier
is invalid.", format));
      }
      return numericString;
   }
}

public class TestTelephoneFormatter
{
   public static void Main()
   {
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 0));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
911));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
8490216));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
4257884748));

      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
0));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
911));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
8490216));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
4257884748));

      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:I}",
4257884748));
   }
}
```

사용자 지정 숫자 형식 공급자는 String.Format(IFormatProvider, String, Object[]) 메서드에서만 사용할 수 있습니다. IFormatProvider 형식의 매개 변수가 있는 숫자 서식 지정 메서드의 다른 오버로드(예: `ToString`)는 모두 NumberFormatInfo 형식을 나타내는 Type 개체를 IFormatProvider.GetFormat 구현에 전달합니다. 반환 시 메서드가 NumberFormatInfo 개체를 반환할 것으로 예상합니다. 반환하지 않을 경우 사용자 지정 숫자 형식 공급자는 무시되고 현재 문화권에 대한 NumberFormatInfo 개체가 대신 사용됩니다. 예제에서 `TelephoneFormatter.GetFormat` 메서드는 메서드 매개 변수를 검사하고 ICustomFormatter 이외의 다른 형식을 나타내는 경우 `null`을 반환하여 숫자 서식 지정 메서드에 부적절하게 전달될 수 있는 가능성을 처리합니다.

사용자 지정 숫자 형식 공급자가 형식 지정자 집합을 지원하는 경우 String.Format(IFormatProvider, String, Object[]) 메서드 호출에 사용된 형식 항목에 형식 지정자가 제공되지 않은 경우의 기본 동작을 제공해야 합니다. 예제에서는 "N"이 기본 형식 지정자입니다. 이렇게 하면 명시적 형식 지정자를 제공하여 숫자를 형식이 지정된 전화 번호로 변환할 수 있습니다. 다음 예제에서는 이러한 메서드 호출을 보여 줍니다.

C#

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
4257884748));
```

그러나 형식 지정자가 없는 경우에도 변환을 수행할 수 있습니다. 다음 예제에서는 이러한 메서드 호출을 보여 줍니다.

C#

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
4257884748));
```

기본 형식 지정자가 정의되어 있지 않으면 .NET이 코드에서 지원하지 않는 서식 지정을 제공할 수 있도록 ICustomFormatter.Format 메서드 구현에 다음과 같은 코드를 포함해야 합니다.

C#

```
if (arg is IFormattable)
   s = ((IFormattable)arg).ToString(format, formatProvider);
else if (arg != null)
   s = arg.ToString();
```

이 예제의 경우 ICustomFormatter.Format을 구현하는 메서드는 String.Format(IFormatProvider, String, Object[]) 메서드에 대한 콜백 메서드 역할을 하기 위한 것입니다. 따라서 이 메서드는 `formatProvider` 매개 변수를 검사하여 현재 `TelephoneFormatter` 개체에 대한 참조가 있는지 여부를 확인합니다. 그러나 코드에서 메서드를 직접 호출할 수도 있습니다. 이 경우 `formatProvider` 매개 변수를 사용하여 문화권별 서식 지정 정보를 제공하는 CultureInfo 또는 NumberFormatInfo 개체를 제공할 수 있습니다.

방법: 왕복 날짜 및 시간 값

아티클 • 2025. 03. 25.

많은 애플리케이션에서 날짜 및 시간 값은 단일 시점을 명확하게 식별하기 위한 것입니다. 이 문서에서는 복원된 값이 저장된 값과 동일한 시간을 식별할 수 있도록 DateTime 값, DateTimeOffset 값 및 표준 시간대 정보가 포함된 날짜 및 시간 값을 저장하고 복원하는 방법을 보여줍니다.

DateTime 값 왕복

1. "o" 형식 지정자를 사용하여 DateTime.ToString(String) 메서드를 호출하여 DateTime 값을 문자열 표현으로 변환합니다.
2. DateTime 값의 문자열 표현을 파일에 저장하거나 프로세스, 애플리케이션 도메인 또는 컴퓨터 경계에 전달합니다.
3. DateTime 값을 나타내는 문자열을 검색합니다.
4. DateTime.Parse(String, IFormatProvider, DateTimeStyles) 메서드를 호출하고 DateTimeStyles.RoundtripKind `styles` 매개 변수의 값으로 전달합니다.

다음 예제에서는 DateTime 값을 왕복하는 방법을 보여 줍니다.

C#

```
const string fileName = @".\DateFile.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTime dateToSave = DateTime.SpecifyKind(new DateTime(2008, 6, 12, 18, 45,
15),
                                           DateTimeKind.Local);
string? dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} ({1}) to {2}.",
                  dateToSave.ToString(),
                  dateToSave.Kind.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine($"Wrote {dateString} to {fileName}.");
outFile.Close();

// Restore DateTime value.
DateTime restoredDate;

using StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();
```

```
if (dateString is not null)
{
    restoredDate = DateTime.Parse(dateString, null,
DateTimeStyles.RoundtripKind);
    Console.WriteLine("Read {0} ({2}) from {1}.", restoredDate.ToString(),
                                                  fileName,

restoredDate.Kind.ToString());
}

// The example displays the following output:
//    Converted 6/12/2008 6:45:15 PM (Local) to 2008-06-12T18:45:15.0000000-
05:00.
//    Wrote 2008-06-12T18:45:15.0000000-05:00 to .\DateFile.txt.
//    Read 6/12/2008 6:45:15 PM (Local) from .\DateFile.txt.
```

DateTime 값을 라운드트립할 때 이 기술은 모든 로컬 및 유니버설 시간에 대한 시간을 성공적으로 유지합니다. 예를 들어 로컬 DateTime 값이 미국 태평양 표준 시간대의 시스템에 저장되고 미국 중부 표준 시간대의 시스템에서 복원되는 경우 복원된 날짜와 시간은 원래 시간보다 2시간 늦습니다. 이는 두 표준 시간대 간의 시간 차이를 반영합니다. 그러나 이 기술이 지정되지 않은 시간에 반드시 정확한 것은 아닙니다. 모든 DateTime 값이 Kind 속성이 Unspecified일 경우 로컬 시간으로 간주됩니다. 현지 시간이 아닌 경우 DateTime 올바른 시점을 성공적으로 식별하지 못합니다. 이 제한에 대한 해결 방법은 저장 및 복원 작업을 위해 날짜 및 시간 값을 표준 시간대와 긴밀하게 결합하는 것입니다.

DateTimeOffset 값 왕복

1. "o" 형식 지정자를 사용하여 DateTimeOffset.ToString(String) 메서드를 호출하여 DateTimeOffset 값을 문자열 표현으로 변환합니다.
2. DateTimeOffset 값의 문자열 표현을 파일에 저장하거나 프로세스, 애플리케이션 도메인 또는 컴퓨터 경계에 전달합니다.
3. DateTimeOffset 값을 나타내는 문자열을 검색합니다.
4. DateTimeOffset.Parse(String, IFormatProvider, DateTimeStyles) 메서드를 호출하고 DateTimeStyles.RoundtripKind `styles` 매개 변수의 값으로 전달합니다.

다음 예제에서는 DateTimeOffset 값을 왕복하는 방법을 보여 줍니다.

C#

```
const string fileName = @".\DateOff.txt";

StreamWriter outFile = new StreamWriter(fileName);
```

```
// Save DateTime value.
DateTimeOffset dateToSave = new DateTimeOffset(2008, 6, 12, 18, 45, 15,
                                               new TimeSpan(7, 0, 0));
string? dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} to {1}.", dateToSave.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine($"Wrote {dateString} to {fileName}.");
outFile.Close();

// Restore DateTime value.
DateTimeOffset restoredDateOff;

using StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();

if (dateString is not null)
{
    restoredDateOff = DateTimeOffset.Parse(dateString, null,
                                           DateTimeStyles.RoundtripKind);
    Console.WriteLine("Read {0} from {1}.", restoredDateOff.ToString(),
fileName);
}

// The example displays the following output:
//    Converted 6/12/2008 6:45:15 PM +07:00 to 2008-06-
12T18:45:15.0000000+07:00.
//    Wrote 2008-06-12T18:45:15.0000000+07:00 to .\DateOff.txt.
//    Read 6/12/2008 6:45:15 PM +07:00 from .\DateOff.txt.
```

이 기술은 항상 DateTimeOffset 값을 단일 시점으로 명확하게 식별합니다. 그런 다음 DateTimeOffset.ToUniversalTime 메서드를 호출하여 값을 UTC(협정 세계시)로 변환하거나 DateTimeOffset.ToOffset 또는 TimeZoneInfo.ConvertTime(DateTimeOffset, TimeZoneInfo) 메서드를 호출하여 특정 표준 시간대의 시간으로 변환할 수 있습니다. 이 기술의 주요 제한 사항은 특정 표준 시간대의 시간을 나타내는 DateTimeOffset 값에서 수행되는 날짜 및 시간 산술 연산이 해당 표준 시간대에 대한 정확한 결과를 생성하지 못할 수 있다는 것입니다. DateTimeOffset 값이 인스턴스화되면 해당 표준 시간대에서 연결이 해제되기 때문입니다. 따라서 날짜 및 시간 계산을 수행할 때 표준 시간대의 조정 규칙을 더 이상 적용할 수 없습니다. 날짜 및 시간 값과 함께 제공되는 표준 시간대를 모두 포함하는 사용자 지정 형식을 정의하여 이 문제를 해결할 수 있습니다.

코드 컴파일

이러한 예제에서는 C# `using` 지시문 또는 Visual Basic `Imports` 문을 사용하여 다음 네임스페이스를 가져와야 합니다.

- System(C#에만 해당)
- System.Globalization

- System.IO

참고하십시오

- DateTime, DateTimeOffset, TimeSpan 및 TimeZoneInfo 중에서 선택
- 표준 날짜 및 시간 형식 문자열

방법: 날짜 및 시간 값으로 밀리초 표시

아티클 • 2025. 04. 10.

DateTime.ToString()같은 기본 날짜 및 시간 서식 지정 메서드에는 시간 값의 시간, 분 및 초가 포함되지만 해당 밀리초 구성 요소는 제외됩니다. 이 문서에서는 날짜 및 시간의 밀리초 구성 요소를 형식이 지정된 날짜 및 시간 문자열에 포함하는 방법을 보여 줍니다.

DateTime 값의 밀리초 구성 요소를 표시하려면

1. 날짜의 문자열 표현을 사용하는 경우 정적 DateTime 또는 DateTimeOffset 메서드를 사용하여 DateTime.Parse(String) 또는 DateTimeOffset.Parse(String) 값으로 변환합니다.
2. 시간 단위 구성 요소의 문자열 표현을 추출하려면 날짜 및 시간 값의 DateTime.ToString(String) 또는 ToString 메서드를 호출하고 `fff` 또는 `FFF` 사용자 지정 형식 패턴만 전달하거나 다른 사용자 지정 형식 지정자를 `format` 매개 변수로 전달합니다.

> **팁**
>
> System.Globalization.NumberFormatInfo.NumberDecimalSeparator 속성은 밀리초 구분 기호를 지정합니다.

예시

이 예제는 DateTime과 DateTimeOffset 값의 밀리초 구성 요소를 콘솔에만 표시하거나 더 긴 날짜 및 시간 문자열에 포함한 형태로 보여줍니다.

C#

```
using System.Globalization;
using System.Text.RegularExpressions;

string dateString = "7/16/2008 8:32:45.126 AM";

try
{
    DateTime dateValue = DateTime.Parse(dateString);
    DateTimeOffset dateOffsetValue = DateTimeOffset.Parse(dateString);

    // Display Millisecond component alone.
    Console.WriteLine($"Millisecond component only: {dateValue.ToString("fff")}");
    Console.WriteLine($"Millisecond component only:
{dateOffsetValue.ToString("fff")}");

    // Display Millisecond component with full date and time.
```

```
    Console.WriteLine($"Date and Time with Milliseconds:
{dateValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt")}");
    Console.WriteLine($"Date and Time with Milliseconds:
{dateOffsetValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt")}");

    string fullPattern = DateTimeFormatInfo.CurrentInfo.FullDateTimePattern;

    // Create a format similar to .fff but based on the current culture.
    string millisecondFormat = $"
{NumberFormatInfo.CurrentInfo.NumberDecimalSeparator}fff";

    // Append millisecond pattern to current culture's full date time pattern.
    fullPattern = Regex.Replace(fullPattern, "(:ss|:s)",
$"$1{millisecondFormat}");

    // Display Millisecond component with modified full date and time pattern.
    Console.WriteLine($"Modified full date time pattern:
{dateValue.ToString(fullPattern)}");
    Console.WriteLine($"Modified full date time pattern:
{dateOffsetValue.ToString(fullPattern)}");
}
catch (FormatException)
{
    Console.WriteLine($"Unable to convert {dateString} to a date.");
}
// The example displays the following output if the current culture is en-US:
//    Millisecond component only: 126
//    Millisecond component only: 126
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
```

`fff` 형식 패턴은 밀리초 값에서 뒤따르는 0을 포함합니다. `FFF` 형식 패턴은 이를 억제합니다. 다음 예제에서는 차이점을 보여 줍니다.

C#

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine(dateValue.ToString("fff"));
Console.WriteLine(dateValue.ToString("FFF"));
// The example displays the following output to the console:
//    180
//    18
```

날짜 및 시간의 밀리초 구성 요소를 포함하는 전체 사용자 지정 형식 지정자를 정의하는 데 문제가 있는 것은 애플리케이션의 현재 문화권에서 시간 요소의 배열에 해당하지 않을 수 있는 하드 코딩된 형식을 정의한다는 것입니다. 더 나은 대안은 현재 문화권의 DateTimeFormatInfo 개체에 정의된 날짜 및 시간 표시 패턴 중 하나를 검색하고 밀리초를 포함하도록 수정하는 것입니다. 이 예제에서는 이 방법도 보여 줍니다. DateTimeFormatInfo.FullDateTimePattern 속성에서 현재

문화권의 전체 날짜 및 시간 패턴을 검색한 다음 현재 문화권의 밀리초 구분 기호와 함께 사용자 지정 패턴 `fff` 삽입합니다. 이 예제에서는 정규식을 사용하여 단일 메서드 호출에서 이 작업을 수행합니다.

사용자 지정 형식 지정자를 사용하여 밀리초 이외의 소수 부분을 초 단위로 표시할 수도 있습니다. 예를 들어 `f` 또는 `F` 사용자 지정 서식 지정자는 1/10초, `ff` 또는 `FF` 사용자 지정 형식 지정자는 수백 초를 표시하고, `ffff` 또는 `FFFF` 사용자 지정 형식 지정자는 1/10000초를 표시합니다. 밀리초의 소수 부분은 반올림되지 않고 잘려서 반환된 문자열에 표시됩니다. 이러한 형식 지정자는 다음 예제에서 사용됩니다.

C#

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine($"{dateValue.ToString("s.f")} seconds");
Console.WriteLine($"{dateValue.ToString("s.ff")} seconds");
Console.WriteLine($"{dateValue.ToString("s.ffff")} seconds");
// The example displays the following output to the console:
//    45.1 seconds
//    45.18 seconds
//    45.1800 seconds
```

> **참고**
>
> 초당 10000분의 1초 또는 10만분의 1초와 같이 초의 매우 작은 소수 단위를 표시할 수 있습니다. 그러나 이러한 값은 의미가 없을 수 있습니다. 날짜 및 시간 값의 정밀도는 운영 체제 클록의 해상도에 따라 달라집니다. 자세한 내용은 운영 체제에서 사용하는 API를 참조하세요.
>
> - Windows 7: GetSystemTimeAsFileTime
> - Windows 8 이상: GetSystemTimePreciseAsFileTime
> - Linux 및 macOS: clock_gettime

참고하십시오

- DateTimeFormatInfo
- 사용자 지정 날짜 및 시간 형식 문자열

방법: 양력 이외의 달력에 날짜 표시

아티클 • 2025. 04. 04.

DateTime 및 DateTimeOffset 형식은 그레고리오력을 기본 달력으로 사용합니다. 즉, 날짜 및 시간 값의 `ToString` 메서드를 호출하면 다른 달력을 사용하여 날짜와 시간을 만든 경우에도 해당 날짜와 시간의 문자열 표현이 그레고리오력에 표시됩니다. 다음 예제에서는 페르시아 달력을 사용하여 날짜 및 시간 값을 만드는 두 가지 방법을 사용하지만 메서드를 호출 ToString 할 때 해당 날짜 및 시간 값을 그레고리오력에 계속 표시합니다. 이 예제에서는 특정 달력에 날짜를 표시하기 위해 일반적으로 사용되지만 잘못된 두 가지 기술을 반영합니다.

C#

```
PersianCalendar persianCal = new PersianCalendar();

DateTime persianDate = persianCal.ToDateTime(1387, 3, 18, 12, 0, 0, 0);
Console.WriteLine(persianDate.ToString());

persianDate = new DateTime(1387, 3, 18, persianCal);
Console.WriteLine(persianDate.ToString());
// The example displays the following output to the console:
//       6/7/2008 12:00:00 PM
//       6/7/2008 12:00:00 AM
```

특정 달력에 날짜를 표시하는 데 두 가지 기술을 사용할 수 있습니다. 첫 번째 달력은 특정 문화권의 기본 달력이어야 합니다. 두 번째는 모든 일정과 함께 사용할 수 있습니다.

문화권의 기본 달력에 대한 날짜를 표시하려면

1. 사용할 달력을 나타내는 클래스에서 Calendar 파생된 달력 개체를 인스턴스화합니다.
2. CultureInfo 날짜를 표시하는 데 사용할 문화권의 서식을 나타내는 개체를 인스턴스화합니다.
3. 메서드를 Array.Exists 호출하여 calendar 개체가 속성에서 반환 CultureInfo.OptionalCalendars 된 배열의 멤버인지 여부를 확인합니다. 이는 달력이 개체의 기본 달력 CultureInfo 으로 사용될 수 있음을 나타냅니다. 배열의 멤버가 아닌 경우 "모든 일정에 날짜를 표시하려면" 섹션의 지침을 따릅니다.
4. CultureInfo.DateTimeFormat 속성에서 반환된 DateTimeFormatInfo 개체의 Calendar 속성에 달력 개체를 할당합니다.

> ⓘ **참고**

클래스에는 CultureInfo 속성도 있습니다 Calendar . 그러나 읽기 전용이며 불변입니다. DateTimeFormatInfo.Calendar 속성에 할당된 새 기본 달력을 반영하도록 변경되지 않습니다.

5. ToString 메서드 또는 ToString 메서드 중 하나를 호출하고, 이전 단계에서 기본 달력이 수정된 CultureInfo 개체를 메서드에 전달합니다.

일정에 날짜를 표시하려면

1. 사용할 달력을 나타내는 클래스에서 Calendar 파생된 달력 개체를 인스턴스화합니다.
2. 날짜 및 시간 값의 문자열 표현에 표시할 날짜 및 시간 요소를 결정합니다.
3. 표시하려는 각 날짜 및 시간 요소가 있을 경우, 달력 개체의 `Get` 메소드를 호출하세요. 메서드. 다음 메서드를 사용할 수 있습니다.
   - GetYear- 해당 달력에 연도를 표시합니다.
   - GetMonth- 해당 달력에 월을 표시합니다.
   - GetDayOfMonth- 해당 달력에 해당 월의 일 수를 표시합니다.
   - GetHour- 해당 달력에 하루 중 시간을 표시합니다.
   - GetMinute- 해당 달력의 시간(분)을 표시합니다.
   - GetSecond- 해당 달력에서 분 단위로 초를 표시합니다.
   - GetMilliseconds- 초의 밀리초를 적절한 달력에 표시합니다.

예시

이 예제에서는 두 개의 다른 달력을 사용하여 날짜를 표시합니다. Hijri 달력을 ar-JO 문화권의 기본 달력으로 정의한 후 날짜를 표시하고, fa-IR 문화권에서 선택적 달력으로 지원되지 않는 페르시아 달력을 사용하여 날짜를 표시합니다.

C#

```
using System;
using System.Globalization;

public class CalendarDates
{
   public static void Main()
   {
      HijriCalendar hijriCal = new HijriCalendar();
```

```
      CalendarUtility hijriUtil = new CalendarUtility(hijriCal);
      DateTime dateValue1 = new DateTime(1429, 6, 29, hijriCal);
      DateTimeOffset dateValue2 = new DateTimeOffset(dateValue1,
                                  TimeZoneInfo.Local.GetUtcOffset(dateValue1));
      CultureInfo jc = CultureInfo.CreateSpecificCulture("ar-JO");

      // Display the date using the Gregorian calendar.
      Console.WriteLine($"Using the system default culture:
{dateValue1.ToString("d")}");
      // Display the date using the ar-JO culture's original default calendar.
      Console.WriteLine($"Using the ar-JO culture's original default calendar:
{dateValue1.ToString("d", jc)}");
      // Display the date using the Hijri calendar.
      Console.WriteLine("Using the ar-JO culture with Hijri as the default
calendar:");
      // Display a Date value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue1, jc));
      // Display a DateTimeOffset value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue2, jc));

      Console.WriteLine();

      PersianCalendar persianCal = new PersianCalendar();
      CalendarUtility persianUtil = new CalendarUtility(persianCal);
      CultureInfo ic = CultureInfo.CreateSpecificCulture("fa-IR");

      // Display the date using the ir-FA culture's default calendar.
      Console.WriteLine($"Using the ir-FA culture's default calendar:
{dateValue1.ToString("d", ic)}");
      // Display a Date value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue1, ic));
      // Display a DateTimeOffset value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue2, ic));
   }
}

public class CalendarUtility
{
   private Calendar thisCalendar;
   private CultureInfo targetCulture;

   public CalendarUtility(Calendar cal)
   {
      this.thisCalendar = cal;
   }

   private bool CalendarExists(CultureInfo culture)
   {
      this.targetCulture = culture;
      return Array.Exists(this.targetCulture.OptionalCalendars,
                          this.HasSameName);
   }

   private bool HasSameName(Calendar cal)
   {
```

```
      if (cal.ToString() == thisCalendar.ToString())
         return true;
      else
         return false;
   }

   public string DisplayDate(DateTime dateToDisplay, CultureInfo culture)
   {
      DateTimeOffset displayOffsetDate = dateToDisplay;
      return DisplayDate(displayOffsetDate, culture);
   }

   public string DisplayDate(DateTimeOffset dateToDisplay,
                             CultureInfo culture)
   {
      string specifier = "yyyy/MM/dd";

      if (this.CalendarExists(culture))
      {
         Console.WriteLine($"Displaying date in supported
{this.thisCalendar.GetType().Name} calendar...");
         culture.DateTimeFormat.Calendar = this.thisCalendar;
         return dateToDisplay.ToString(specifier, culture);
      }
      else
      {
         Console.WriteLine($"Displaying date in unsupported
{thisCalendar.GetType().Name} calendar...");

         string separator = targetCulture.DateTimeFormat.DateSeparator;

         return thisCalendar.GetYear(dateToDisplay.DateTime).ToString("0000") +
                separator +
                thisCalendar.GetMonth(dateToDisplay.DateTime).ToString("00") +
                separator +
                thisCalendar.GetDayOfMonth(dateToDisplay.DateTime).ToString("00");
      }
   }
}
// The example displays the following output to the console:
//       Using the system default culture: 7/3/2008
//       Using the ar-JO culture's original default calendar: 03/07/2008
//       Using the ar-JO culture with Hijri as the default calendar:
//       Displaying date in supported HijriCalendar calendar...
//       1429/06/29
//       Displaying date in supported HijriCalendar calendar...
//       1429/06/29
//
//       Using the ir-FA culture's default calendar: 7/3/2008
//       Displaying date in unsupported PersianCalendar calendar...
//       1387/04/13
//       Displaying date in unsupported PersianCalendar calendar...
//       1387/04/13
```

각 CultureInfo 개체는 속성으로 표시되는 OptionalCalendars 하나 이상의 일정을 지원할 수 있습니다. 이 중 하나는 문화권의 기본 달력으로 지정되며 읽기 전용 CultureInfo.Calendar 속성에서 반환됩니다. 다른 선택적 달력을 DateTimeFormatInfo.Calendar 기본 달력으로 지정하려면 CultureInfo.DateTimeFormat 속성이 반환하는 속성에 해당 달력을 나타내는 Calendar 개체를 할당할 수 있습니다. 그러나 클래스가 나타내는 페르시아 달력과 같은 일부 달력은 PersianCalendar 문화권에 대한 선택적 달력으로 사용되지 않습니다.

이 예제에서는 재사용 가능한 달력 유틸리티 클래스 `CalendarUtility`를 정의하여 특정 달력을 사용하여 날짜의 문자열 표현을 생성하는 많은 세부 정보를 처리합니다. `CalendarUtility` 클래스에는 다음 멤버가 있습니다.

- Calendar 개체로 날짜를 나타내는 단일 매개 변수가 있는 매개변수화된 생성자입니다. 클래스의 프라이빗 필드에 할당됩니다.

- `CalendarExists`는 `CalendarUtility` 개체가 나타내는 달력이 메서드에 매개 변수로 전달된 CultureInfo 개체에 의해 지원되는지 여부를 나타내는 부울 값을 반환하는 private 메서드입니다. 메서드는 CultureInfo.OptionalCalendars 배열을 전달하는 Array.Exists 메서드 호출을 래핑합니다.

- `HasSameName`는 Array.Exists 메서드의 매개 변수로 전달된 Predicate<T> 대리자에 할당된 프라이빗 메서드입니다. 배열의 각 멤버는 메서드가 반환될 때까지 메서드에 전달됩니다 `true`. 이 메서드는 선택적 달력의 이름이 개체가 나타내는 `CalendarUtility` 달력과 같은지 여부를 결정합니다.

- `DisplayDate`는 두 개의 매개변수를 전달받는 오버로드된 public 메서드입니다. 첫 번째 매개변수는 `CalendarUtility` 개체로 표현되는 달력에서 표현할 값으로, 이는 DateTime 또는 DateTimeOffset 값일 수 있습니다. 두 번째 매개변수는 서식 규칙을 사용할 문화권입니다. 날짜의 문자열 표현을 반환하는 동작은 서식 규칙을 사용할 문화권에서 대상 달력을 지원하는지 여부에 따라 달라집니다.

이 예제에서 값을 만드는 DateTimeDateTimeOffset 데 사용되는 달력에 관계없이 해당 값은 일반적으로 그레고리오 날짜로 표현됩니다. 이는 DateTime 형식과 DateTimeOffset 형식이 달력 정보를 전혀 유지하지 않기 때문입니다. 내부적으로 0001년 1월 1일 자정 이후 경과된 틱 수로 표시됩니다. 해당 숫자의 해석은 달력에 따라 달라집니다. 대부분의 문화권에서 기본 달력은 그레고리오력입니다.

.NET의 문자 인코딩

아티클 • 2024. 10. 22.

이 문서에서는 .NET에서 사용되는 문자 인코딩 시스템에 대해 소개합니다. 이 문서에서는 String, Char, Rune 및 StringInfo 형식이 유니코드, UTF-16 및 UTF-8에서 작동하는 방식에 대해 설명합니다.

여기서는 *문자*라는 용어를 *판독기가 단일 표시 요소로 인식*한다는 일반적인 의미로 사용합니다. 일반적인 예는 문자 "a", 기호 "@" 및 이모지 "🐂"입니다. 문자소 클러스터에 대한 섹션에서 설명된 것처럼 한 문자가 실제로는 여러 독립적인 표시 요소로 구성되는 경우도 있습니다.

string 및 char 형식

string 클래스의 인스턴스는 일부 텍스트를 나타냅니다. `string`은 논리적으로 16비트 값의 시퀀스이며, 각각은 char 구조체의 인스턴스입니다. string.Length 속성은 `string` 인스턴스의 `char` 인스턴스 수를 반환합니다.

다음 샘플 함수는 `string`에 있는 모든 `char` 인스턴스의 값을 16진수 표기법으로 출력합니다.

C#

```
void PrintChars(string s)
{
    Console.WriteLine($"\"{s}\".Length = {s.Length}");
    for (int i = 0; i < s.Length; i++)
    {
        Console.WriteLine($"s[{i}] = '{s[i]}' ('\\u{(int)s[i]:x4}')");
    }
    Console.WriteLine();
}
```

이 함수에 string “Hello”를 전달하면 다음과 같은 출력이 표시됩니다.

C#

```
PrintChars("Hello");
```

출력

```
"Hello".Length = 5
s[0] = 'H' ('\u0048')
```

```
s[1] = 'e' ('\u0065')
s[2] = 'l' ('\u006c')
s[3] = 'l' ('\u006c')
s[4] = 'o' ('\u006f')
```

각 문자는 단일 `char` 값으로 표현됩니다. 이 패턴은 대부분의 세계 언어에 적용됩니다. 예를 들어 다음은 *nǐ hǎo*로 발음되고 *Hello*를 의미하는 두 개의 중국어 문자의 출력입니다.

C#

```
PrintChars("你好");
```

출력

```
"你好".Length = 2
s[0] = '你' ('\u4f60')
s[1] = '好' ('\u597d')
```

그러나 일부 언어와 일부 기호 및 이모지에서는 두 개의 `char` 인스턴스를 사용하여 단일 문자를 나타냅니다. 예를 들어 오세이지족 언어에서 *Osage*를 의미하는 단어의 문자와 `char` 인스턴스를 비교해 보겠습니다.

C#

```
PrintChars("𐓏𐓘𐓻𐓘𐓻𐓘 𐒻𐓘");
```

출력

```
"𐓏𐓘𐓻𐓘𐓻𐓘 𐒻𐓘".Length = 17
s[0] = '?' ('\ud801')
s[1] = '?' ('\udccf')
s[2] = '?' ('\ud801')
s[3] = '?' ('\udcd8')
s[4] = '?' ('\ud801')
s[5] = '?' ('\udcfb')
s[6] = '?' ('\ud801')
s[7] = '?' ('\udcd8')
s[8] = '?' ('\ud801')
s[9] = '?' ('\udcfb')
s[10] = '?' ('\ud801')
s[11] = '?' ('\udcdf')
s[12] = ' ' ('\u0020')
s[13] = '?' ('\ud801')
s[14] = '?' ('\udcbb')
s[15] = '?' ('\ud801')
s[16] = '?' ('\udcdf')
```

위의 예제에서 공백을 제외한 각 문자는 두 개의 `char` 인스턴스로 표현됩니다.

ox 이모지를 보여 주는 다음 예제에 표시된 것처럼 단일 유니코드 이모지도 두 개의 `char` 로 표현됩니다.

출력

```
"🐂".Length = 2
s[0] = '' ('\ud83d')
s[1] = '' ('\udc02')
```

이러한 예제에서는 `char` 인스턴스의 수를 나타내는 `string.Length` 값이 표시되는 문자 수를 반드시 나타내지는 않음을 보여 줍니다. 단일 `char` 인스턴스 자체가 반드시 문자를 나타내는 것은 아닙니다.

단일 문자에 매핑되는 `char` 쌍을 *서로게이트 쌍*이라고 합니다. 그 작동 방식을 이해하려면 유니코드 및 UTF-16 인코딩을 이해해야 합니다.

유니코드 코드 포인트

유니코드는 다양한 언어 및 스크립트를 사용하여 다양한 플랫폼에서 사용하기 위한 국제 인코딩 표준입니다.

유니코드 표준은 110만 개 이상의 코드 포인트를 정의합니다. 코드 포인트는 0과 `U+10FFFF`(10진수 1,114,111) 사이의 정수 값입니다. 일부 코드 포인트는 문자, 기호 또는 이모지에 할당됩니다. 다른 코드 포인트는 텍스트 또는 문자가 표시되는 방식을 제어하는 작업(예: 새 줄로 이동)에 할당됩니다. 많은 코드 포인트가 아직 할당되지 않은 상태입니다.

다음은 코드 포인트 할당의 몇 가지 예제와 해당 유니코드 차트에 대한 링크입니다.

테이블 확장

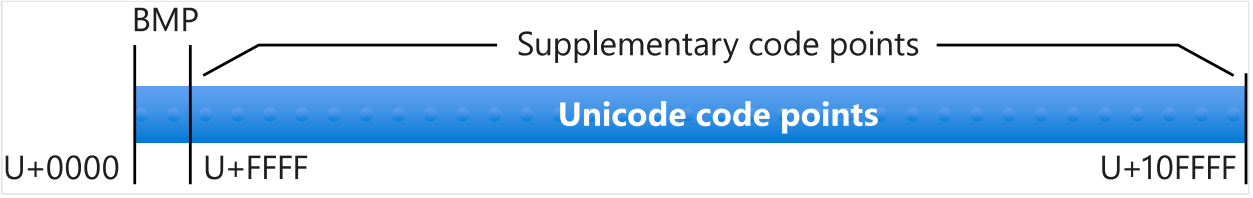
소수	16진수	예제	설명
10	`U+000A`	해당 없음	줄 바꿈
97	`U+0061`	a	라틴 소문자 A
562	`U+0232`	Ȳ	장음 기호를 사용하는 라틴어 대문자 Y
68,675	`U+10C43`	𐱃	고대 튀르크 문자 ORKHON AT
127,801	`U+1F339`	🌹	장미 이모지

코드 포인트는 관례적으로 `U+xxxx` 구문을 사용하여 지칭됩니다. 여기서 `xxxx`는 16진수로 인코딩된 정수 값입니다.

코드 포인트의 전체 범위 내에 두 가지 하위 범위가 있습니다.

- **BMP(기본 다국어 평면)**는 `U+0000..U+FFFF` 범위에 있습니다. 이 16비트 범위는 전 세계 쓰기 시스템을 대부분 포괄하는 데 충분한 65,536개 코드 포인트를 제공합니다.
- **보조 코드 포인트**는 `U+10000..U+10FFFF` 범위에 있습니다. 이 21비트 범위는 덜 알려진 언어와 이모지 같은 다른 용도로 사용할 수 있는 백만 개 이상의 추가 코드 포인트를 제공합니다.

다음 다이어그램에서는 BMP와 보조 코드 포인트의 관계를 보여 줍니다.



UTF-16 코드 단위

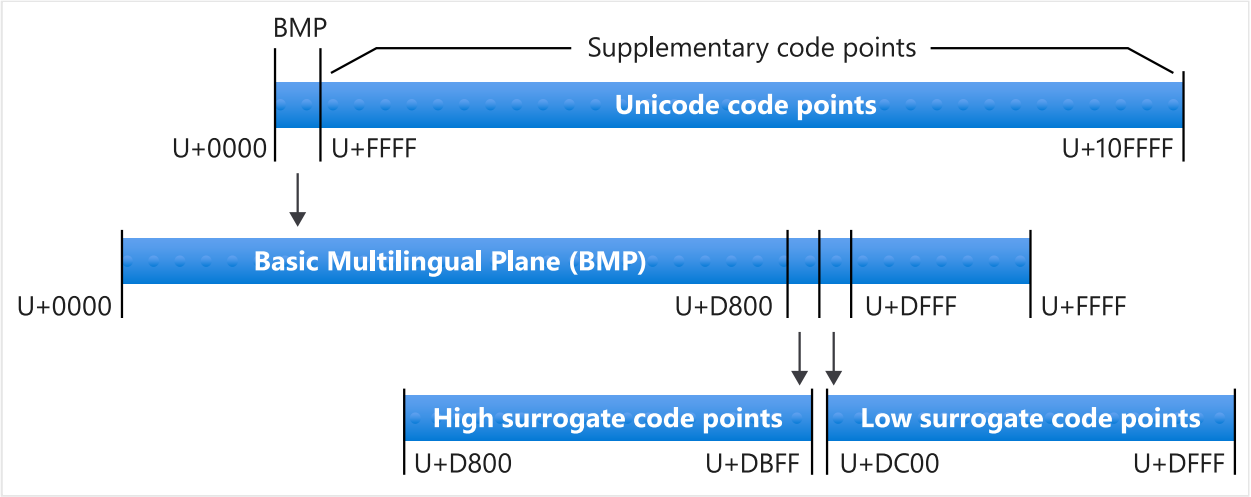
16비트 유니코드 변환 형식(UTF-16)은 유니코드 코드 포인트를 나타내기 위해 16비트 *코드 단위*를 사용하는 문자 인코딩 시스템입니다. .NET에서는 UTF-16을 사용하여 `string`의 텍스트를 인코딩합니다. `char` 인스턴스는 16비트 코드 단위를 나타냅니다.

단일 16비트 코드 단위는 기본 다국어 평면 16비트 범위의 코드 포인트를 나타낼 수 있습니다. 하지만 보조 범위의 코드 포인트의 경우 두 개의 `char` 인스턴스가 필요합니다.

서로게이트 쌍

두 개의 16비트 값을 단일 21비트 값으로 변환하는 과정은 `U+D800`부터 `U+DFFF`까지(10진수 55.296부터 57,343까지)의 특수 범위인 *서로게이트 코드 포인트*를 통해 간단해집니다.

다음 다이어그램에서는 BMP와 서로게이트 코드 포인트의 관계를 보여 줍니다.



상위 서로게이트 코드 포인트(`U+D800..U+DBFF`) 바로 다음에 *하위 서로게이트* 코드 포인트(`U+DC00..U+DFFF`)가 오는 경우 이 쌍은 다음 수식을 사용하여 보조 코드 포인트로 해석됩니다.

```
code point = 0x10000 +
  ((high surrogate code point - 0xD800) * 0x0400) +
  (low surrogate code point - 0xDC00)
```

다음은 10진수 표기법을 사용하는 동일한 수식입니다.

```
code point = 65,536 +
  ((high surrogate code point - 55,296) * 1,024) +
  (low surrogate code point - 56,320)
```

상위 서로게이트 코드 포인트가 *하위* 서로게이트 코드 포인트보다 높은 값을 갖는 것은 아닙니다. 상위 서로게이트 코드 포인트는 20비트 코드 포인트 범위의 상위 차수 10비트를 계산하는 데 사용되기 때문에 "상위"라고 합니다. 하위 서로게이트 코드 포인트는 하위 차수 10비트를 계산하는 데 사용됩니다.

예를 들어 서로게이트 쌍 `0xD83C` 및 `0xDF39`에 해당하는 실제 코드 포인트는 다음과 같이 계산됩니다.

```
actual = 0x10000 + ((0xD83C - 0xD800) * 0x0400) + (0xDF39 - 0xDC00)
       = 0x10000 + (          0x003C  * 0x0400) +           0x0339
       = 0x10000 +                      0xF000  +           0x0339
       = 0x1F339
```

다음은 10진수 표기법을 사용한 동일한 계산입니다.

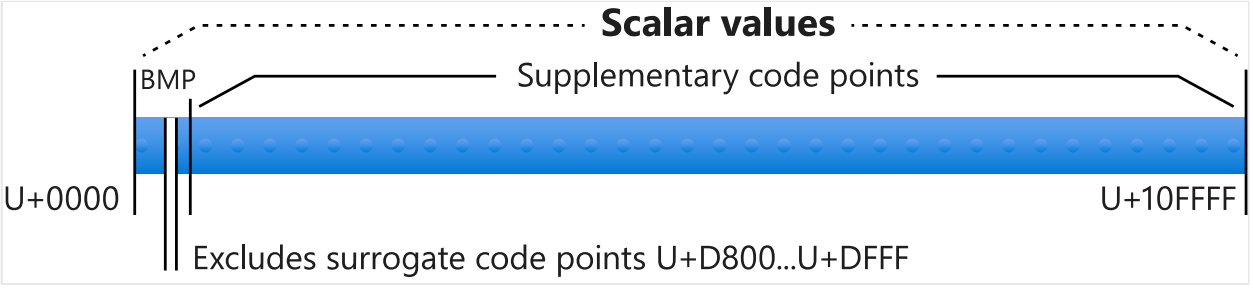
```
actual =  65,536 + ((55,356 - 55,296) * 1,024) + (57,145 - 56320)
       =  65,536 + (              60  * 1,024) +             825
       =  65,536 +                    61,440  +             825
       = 127,801
```

앞의 예제에서는 `"\ud83c\udf39"`가 앞서 언급한 `U+1F339 ROSE ('🌹')` 코드 포인트의 UTF-16 인코딩입니다.

유니코드 스칼라 값

용어 유니코드 스칼라 값은 서로게이트 코드 포인트가 아닌 모든 코드 포인트를 참조합니다. 즉, 스칼라 값은 문자에 할당했거나 나중에 문자에 할당할 수 있는 임의의 코드 포인트입니다. 여기서 "문자"는 텍스트 또는 문자가 표시되는 방식을 제어하는 작업 등을 포함하여 코드 포인트에 할당할 수 있는 모든 것을 나타냅니다.

다음 다이어그램에서는 스칼라 값 코드 포인트를 보여 줍니다.



스칼라 값의 Rune 형식

> **ⓘ 중요**
>
> 형식은 `Rune` .NET Framework에서 사용할 수 없습니다.

.NET에서 형식은 System.Text.Rune 유니코드 스칼라 값을 나타냅니다.

`Rune` 생성자는 결과 인스턴스가 유효한 유니코드 스칼라 값인지 확인합니다. 유효하지 않은 경우에는 예외를 throw합니다. 다음 예제에서는 입력이 유효한 스칼라 값을 나타내므로 `Rune` 인스턴스를 성공적으로 인스턴스화하는 코드를 보여 줍니다.

```
C#
```

```
Rune a = new Rune('a');
Rune b = new Rune(0x0061);
Rune c = new Rune('\u0061');
Rune d = new Rune(0x10421);
Rune e = new Rune('\ud801', '\udc21');
```

다음 예제에서는 코드 포인트가 서로게이트 범위에 있고 서로게이트 쌍에 속하지 않기 때문에 예외를 throw합니다.

C#

```
Rune f = new Rune('\ud801');
```

다음 예제에서는 코드 포인트가 보조 범위를 초과하기 때문에 예외를 throw합니다.

C#

```
Rune g = new Rune(0x12345678);
```

Rune 사용 예제: 문자 대/소문자 변경

`char`를 사용하고 스칼라 값 코드 포인트에서 작동하는 것으로 가정하는 API는 `char`가 서로게이트 쌍의 일부인 경우 올바르게 작동하지 않습니다. 예를 들어 string의 각 char에서 Char.ToUpperInvariant를 호출하는 다음 메서드를 살펴보겠습니다.

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string ConvertToUpperBadExample(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    for (int i = 0; i < input.Length; i++) /* or 'foreach' */
    {
        builder.Append(char.ToUpperInvariant(input[i]));
    }
    return builder.ToString();
}
```

`input` string에 소문자 데저렛 문자 `er`(𐐨)이 포함되는 경우 이 코드는 대문자(𐐠)로 변환되지 않습니다. 이 코드는 각 서로게이트 코드 포인트 `U+D801` 및 `U+DC49`에서 별도로 `char.ToUpperInvariant`를 호출합니다. 그러나 `U+D801` 자체에는 해당 정보를 소문자로 식별하는 데 충분한 정보가 없으므로 `char.ToUpperInvariant`는 이를 그대로 둡니다. 그리고

`U+DC49`도 동일한 방식으로 처리합니다. 그 결과 `input` string의 소문자 'ϕ'가 대문자 'Φ'로 변환되지 않습니다.

string를 올바르게 대문자로 변환하는 두 가지 옵션은 다음과 같습니다.

- `char` 및 `char`를 반복하는 대신 입력 string에서 String.ToUpperInvariant를 호출합니다. `string.ToUpperInvariant` 메서드는 각 서로게이트 쌍의 두 부분에 모두 액세스할 수 있으므로 모든 유니코드 코드 포인트를 올바르게 처리할 수 있습니다.
- 다음 예제와 같이 `char` 인스턴스 대신 `Rune` 인스턴스로 유니코드 스칼라 값을 반복합니다. `Rune` 인스턴스는 유효한 유니코드 스칼라 값이므로 스칼라 값에 대해 작동할 것으로 간주되는 API에 전달될 수 있습니다. 예를 들어 다음 예제와 같이 Rune.ToUpperInvariant를 호출하면 올바른 결과가 반환됩니다.

```
C#
```

```
static string ConvertToUpper(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    foreach (Rune rune in input.EnumerateRunes())
    {
        builder.Append(Rune.ToUpperInvariant(rune));
    }
    return builder.ToString();
}
```

기타 Rune API

`Rune` 형식은 다수의 `char` API의 아날로그를 제공합니다. 예를 들어 다음 메서드는 `char` 형식에 대한 정적 API를 미러링합니다.

- Rune.IsLetter
- Rune.IsWhiteSpace
- Rune.IsLetterOrDigit
- Rune.GetUnicodeCategory

`Rune` 인스턴스에서 원시 스칼라 값을 가져오려면 Rune.Value 속성을 사용합니다.

`Rune` 인스턴스를 `char`의 시퀀스로 다시 변환하려면 Rune.ToString 또는 Rune.EncodeToUtf16 메서드를 사용합니다.

모든 유니코드 스칼라 값은 단일 `char` 또는 서로게이트 쌍으로 표현할 수 있으므로 `Rune` 인스턴스는 최대 2개의 `char` 인스턴스로 나타낼 수 있습니다.

Rune.Utf16SequenceLength를 사용하여 `Rune` 인스턴스를 표현하는 데 필요한 `char` 인스턴스 수를 확인합니다.

.NET `Rune` 형식에 대한 자세한 내용은 Rune API 참조를 참조하세요.

문자소 클러스터

한 문자는 여러 코드 포인트가 조합되어 표시될 수 있으므로 "문자" 대신 자주 사용되는 보다 설명적인 용어가 문자소 클러스터 입니다. .NET에서 해당 용어는 텍스트 요소입니다.

`string` 인스턴스 “a”, “á”, “á”, “🦃” 등을 살펴보겠습니다, 운영 체제가 유니코드 표준에 지정된 대로 이들 항목을 처리하는 경우 각 `string` 인스턴스는 단일 텍스트 요소나 문자소 클러스터로 나타납니다. 하지만 마지막 두 개는 둘 이상의 스칼라 값 코드 포인트로 표현됩니다.

- string "a"는 하나의 스칼라 값으로 표현되고 하나의 `char` 인스턴스를 포함합니다.
  - `U+0061 LATIN SMALL LETTER A`
- string "á"는 하나의 스칼라 값으로 표현되고 하나의 `char` 인스턴스를 포함합니다.
  - `U+00E1 LATIN SMALL LETTER A WITH ACUTE`
- string "á"는 "Á"와 같아 보이지만 두 개의 스칼라 값으로 표현되고 두 개의 `char` 인스턴스를 포함합니다.
  - `U+0061 LATIN SMALL LETTER A`
  - `U+0301 COMBINING ACUTE ACCENT`
- 마지막으로 string "🦃"은 4개의 스칼라 값으로 표현되고 7개의 `char` 인스턴스를 포함합니다.
  - `U+1F469 WOMAN`(보조 범위, 서로게이트 쌍 필요)
  - `U+1F3FD EMOJI MODIFIER FITZPATRICK TYPE-4`(보조 범위, 서로게이트 쌍 필요)
  - `U+200D ZERO WIDTH JOINER`
  - `U+1F692 FIRE ENGINE`(보조 범위, 서로게이트 쌍 필요)

위의 예제 중 일부(예: 결합 악센트 한정자 또는 스킨 톤 한정자)에서는 코드 포인트가 화면에 독립 실행형 요소로 표시되지 않습니다. 대신, 텍스트 요소의 모양을 수정하는 데 사용됩니다. 이러한 예제에서는 단일 "문자" 또는 "문자소 클러스터"로 생각하는 것을 구성하기 위해 여러 스칼라 값이 필요할 수 있음을 보여 줍니다.

`string`의 문자소 클러스터를 열거하려면 다음 예제와 같이 StringInfo 클래스를 사용합니다. Swift에 대해 잘 알고 있는 경우 .NET `StringInfo` 형식은 개념적으로 Swift의 character 형식과 비슷합니다.

예: count char, Rune 및 텍스트 요소 인스턴스

.NET API에서는 문자소 클러스터를 *텍스트 요소*라고 합니다. 다음 메서드는 `string`에서 `char`, `Rune` 및 텍스트 요소 인스턴스 간의 차이점을 보여 줍니다.

C#

```
static void PrintTextElementCount(string s)
{
    Console.WriteLine(s);
    Console.WriteLine($"Number of chars: {s.Length}");
    Console.WriteLine($"Number of runes: {s.EnumerateRunes().Count()}");

    TextElementEnumerator enumerator =
StringInfo.GetTextElementEnumerator(s);

    int textElementCount = 0;
    while (enumerator.MoveNext())
    {
        textElementCount++;
    }

    Console.WriteLine($"Number of text elements: {textElementCount}");
}
```

C#

```
PrintTextElementCount("a");
// Number of chars: 1
// Number of runes: 1
// Number of text elements: 1

PrintTextElementCount("á");
// Number of chars: 2
// Number of runes: 2
// Number of text elements: 1

PrintTextElementCount("🤦");
// Number of chars: 7
// Number of runes: 4
// Number of text elements: 1
```

예: string 인스턴스 분할

`string` 인스턴스를 분할하는 경우 서로게이트 쌍 및 문자소 클러스터를 분할하지 마세요. 다음의 잘못된 코드 예제를 살펴보겠습니다. 이 코드는 string에서 10자마다 줄 바꿈을 삽입하려고 한 것입니다.

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string InsertNewlinesEveryTencharsBadExample(string input)
{
    StringBuilder builder = new StringBuilder();

    // First, append chunks in multiples of 10 chars
    // followed by a newline.
    int i = 0;
    for (; i < input.Length - 10; i += 10)
    {
        builder.Append(input, i, 10);
        builder.AppendLine(); // newline
    }

    // Then append any leftover data followed by
    // a final newline.
    builder.Append(input, i, input.Length - i);
    builder.AppendLine(); // newline

    return builder.ToString();
}
```

이 코드는 `char` 인스턴스를 열거하기 때문에 10개 `char`의 경계에 걸쳐 있는 서로게이트 쌍이 분할되고 그 사이에 줄 바꿈이 삽입됩니다. 서로게이트 코드 포인트는 쌍으로만 의미가 있으므로 이 삽입은 데이터를 손상시킵니다.

`char` 인스턴스 대신 `Rune` 인스턴스(스칼라 값)를 열거하는 경우 데이터 손상 가능성이 제거되지 않습니다. 한 `Rune` 인스턴스 집합이 10개 `char`의 경계에 걸쳐 있는 문자소 클러스터를 구성할 수 있습니다. 문자소 클러스터 집합이 분할된 경우 올바르게 해석할 수 없습니다.

다음 예제와 같이 문자소 클러스터(또는 텍스트 요소)를 계산하여 string을 줄 바꿈하는 것이 더 나은 방법입니다.

C#

```
static string InsertNewlinesEveryTenTextElements(string input)
{
    StringBuilder builder = new StringBuilder();

    // Append chunks in multiples of 10 chars

    TextElementEnumerator enumerator =
StringInfo.GetTextElementEnumerator(input);

    int textElementCount = 1;
    while (enumerator.MoveNext())
```

```
    {
        builder.Append(enumerator.Current);
        if (textElementCount % 10 == 0 && textElementCount > 0)
        {
            builder.AppendLine(); // newline
        }
        textElementCount++;
    }

    // Add a final newline.
    builder.AppendLine(); // newline
    return builder.ToString();

}
```

앞에서 설명한 것처럼 .NET 5 이전에 `StringInfo` 클래스에 버그가 발생하여 일부 그래프 클러스터가 잘못 처리되었습니다.

UTF-8 및 UTF-32

이전 섹션에서는 UTF-16에 초점을 두었습니다. .NET이 `string` 인스턴스를 인코딩하는 데 사용하기 때문이었습니다. 다른 유니코드용 인코딩 시스템 UTF-8 및 UTF-32도 있습니다. 이러한 인코딩은 각각 8비트 코드 단위 및 32비트 코드 단위를 사용합니다.

UTF-16과 마찬가지로 UTF-8은 일부 유니코드 스칼라 값을 나타내기 위해 여러 코드 단위가 필요합니다. UTF-32는 단일 32비트 코드 단위로 모든 스칼라 값을 나타낼 수 있습니다.

다음은 이러한 세 가지 유니코드 인코딩 시스템에서 동일한 유니코드 코드 포인트를 표현하는 방법을 보여 주는 몇 가지 예제입니다.

```
Scalar: U+0061 LATIN SMALL LETTER A ('a')
UTF-8 : [ 61 ]           (1x  8-bit code unit  = 8 bits total)
UTF-16: [ 0061 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 00000061 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+0429 CYRILLIC CAPITAL LETTER SHCHA ('Щ')
UTF-8 : [ D0 A9 ]        (2x  8-bit code units = 16 bits total)
UTF-16: [ 0429 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 00000429 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+A992 JAVANESE LETTER GA ('ꦒ')
UTF-8 : [ EA A6 92 ]     (3x  8-bit code units = 24 bits total)
UTF-16: [ A992 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 0000A992 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+104CC OSAGE CAPITAL LETTER TSHA ('𐓌')
UTF-8 : [ F0 90 93 8C ]  (4x  8-bit code units = 32 bits total)
```

```
UTF-16: [ D801 DCCC ]    (2x 16-bit code units = 32 bits total)
UTF-32: [ 000104CC ]     (1x 32-bit code unit  = 32 bits total)
```

앞서 설명한 것처럼 서로게이트 쌍의 단일 UTF-16 코드 단위는 그 자체로는 의미가 없습니다. 동일한 방식으로 단일 UTF-8 코드 단위가 스칼라 값을 계산하기 위해 2, 3 또는 4개의 시퀀스로 사용되는 경우에는 자체로는 의미가 없습니다.

> ⓘ **참고**
>
> C# 11부터 리터럴 string에 "u8" 접미사를 사용하여 UTF-8 string 리터럴을 나타낼 수 있습니다. UTF-8 string 리터럴에 대한 자세한 내용은 C# 가이드의 **기본 제공 참조 형식**에 대한 문서의 "string 리터럴" 섹션을 참조하세요.

endian

.NET에서 string의 UTF-16 코드 단위는 연속 메모리에 16비트 정수(`char` 인스턴스)의 시퀀스로 저장됩니다. 개별 코드 단위의 비트는 현재 아키텍처의 endian 에 따라 배치됩니다.

Little-Endian 아키텍처에서는 UTF-16 코드 포인트 `[ D801 DCCC ]`로 구성된 string이 바이트 `[ 0x01, 0xD8, 0xCC, 0xDC ]`로 메모리에 배치됩니다. Big-Endian 아키텍처에서는 동일한 string이 바이트 `[ 0xD8, 0x01, 0xDC, 0xCC ]`로 메모리에 배치됩니다.

서로 통신하는 컴퓨터 시스템은 네트워크를 통과하는 데이터의 표현에 동의해야 합니다. 대부분의 네트워크 프로토콜은 텍스트를 전송할 때 UTF-8을 표준으로 사용하여 Little-Endian 컴퓨터와 통신하는 Big-Endian 컴퓨터에서 발생하는 문제를 부분적으로 방지합니다. UTF-8 코드 포인트 `[ F0 90 93 8C ]`로 구성되는 string은 endian에 관계없이 항상 바이트 `[ 0xF0, 0x90, 0x93, 0x8C ]`로 표현됩니다.

텍스트를 전송하는 데 UTF-8을 사용하기 위해 .NET 애플리케이션은 종종 다음 예제와 같은 코드를 사용합니다.

C#

```
string stringToWrite = GetString();
byte[] stringAsUtf8Bytes = Encoding.UTF8.GetBytes(stringToWrite);
await outputStream.WriteAsync(stringAsUtf8Bytes, 0,
stringAsUtf8Bytes.Length);
```

앞의 예제에서 Encoding.UTF8.GetBytes 메서드는 UTF-16 `string`을 일련의 유니코드 스칼라 값으로 다시 디코딩한 다음 해당 스칼라 값을 UTF-8로 인코드하여 결과 시퀀스를

`byte` 배열에 배치합니다. Encoding.UTF8.GetString 메서드는 반대 방향으로 변환을 수행하여 UTF-8 `byte` 배열을 UTF-16 `string`로 변환합니다.

> **⚠ 경고**
>
> UTF-8은 인터넷에서 일반적이므로 네트워크에서 원시 바이트를 읽고 데이터를 UTF-8인 것처럼 처리하는 것이 좋습니다. 그러나 잘 구성된 인코딩인지 유효성을 검사해야 합니다. 악의적인 클라이언트가 잘못 구성된 UTF-8을 서비스에 제출할 수 있습니다. 이러한 데이터를 잘 구성된 것처럼 작업하는 경우 애플리케이션에서 오류 또는 보안 허점이 발생할 수 있습니다. UTF-8 데이터의 유효성을 검사하려면 들어오는 데이터를 `string`으로 변환하는 동안 유효성 검사를 수행하는 `Encoding.UTF8.GetString`와 같은 메서드를 사용할 수 있습니다.

잘 구성된 인코딩

잘 구성된 유니코드 인코딩은 오류 없이 명확하게 디코딩할 수 있는 유니코드 스칼라 값의 시퀀스로 디코딩할 수 있는 코드 단위의 string입니다. 잘 구성된 데이터는 UTF-8, UTF-16 및 UTF-32 사이에서 자유롭게 트랜스코딩될 수 있습니다.

인코딩 시퀀스가 잘 구성되었는지 여부는 컴퓨터 아키텍처의 endian과 관련이 없습니다. 잘못 구성된 UTF-8 시퀀스는 Big-Endian 및 Little-Endian 컴퓨터 모두에서 동일한 방식으로 잘못 구성된 것입니다.

다음은 잘못 구성된 인코딩의 몇 가지 예제입니다.

- UTF-8에서 시퀀스 `[ 6C C2 61 ]`은 잘못 구성된 것입니다. `C2` 뒤에 `61`이 올 수 없기 때문입니다.
- UTF-16에서 시퀀스 `[ DC00 DD00 ]`(또는 C#의 string `"\udc00\udd00"`)는 잘못 구성된 것입니다. 하위 서로게이트 `DC00` 뒤에 다른 하위 서로게이트 `DD00`가 올 수 없기 때문입니다.
- UTF-32에서 시퀀스 `[ 0011ABCD ]`는 잘못 구성된 것입니다. `0011ABCD`가 유니코드 스칼라 값 범위를 벗어나기 때문입니다.

.NET에서 `string` 인스턴스는 거의 항상 잘 구성된 UTF-16 데이터를 포함하지만 이것이 보장되지는 않습니다. 다음 예제에서는 `string` 인스턴스에 잘못 구성된 UTF-16 데이터를 만드는 유효한 C# 코드를 보여 줍니다.

- 잘못 구성된 리터럴:

```
C#
```

```
const string s = "\ud800";
```

- 서로게이트 쌍을 분할하는 substring:

```
C#
string x = "\ud83e\udd70"; // "🥰"
string y = x.Substring(1, 1); // "\udd70" standalone low surrogate
```

Encoding.UTF8.GetString과 같은 API는 잘못 구성된 `string` 인스턴스를 반환하지 않습니다. `Encoding.GetString` 및 `Encoding.GetBytes` 메서드는 입력에서 잘못 구성된 시퀀스를 검색하고 출력을 생성할 때 문자 대체를 수행합니다. 예를 들어 Encoding.ASCII.GetString(byte[])이 입력에서 ASCII가 아닌 바이트(U+0000..U+007F 범위를 벗어남)를 발견할 경우 반환된 `string` 인스턴스에 '?'를 삽입합니다. Encoding.UTF8.GetString(byte[])은 반환된 `string` 인스턴스에서 잘못 구성된 UTF-8 시퀀스를 `U+FFFD REPLACEMENT CHARACTER ('�')`로 바꿉니다. 자세한 내용은 유니코드 표준 섹션 5.22 및 3.9를 참조하세요.

잘못 구성된 시퀀스가 발견될 때 문자 대체를 수행하는 대신 예외를 throw하도록 기본 제공 `Encoding` 클래스를 구성할 수도 있습니다. 이 방법은 문자 대체가 허용되지 않을 수 있는 보안 관련 애플리케이션에서 자주 사용됩니다.

```
C#
byte[] utf8Bytes = ReadFromNetwork();
UTF8Encoding encoding = new UTF8Encoding(encoderShouldEmitUTF8Identifier:
false, throwOnInvalidBytes: true);
string asString = encoding.GetString(utf8Bytes); // will throw if
'utf8Bytes' is ill-formed
```

기본 제공 `Encoding` 클래스를 사용하는 방법에 대한 자세한 내용은 .NET에서 문자 인코딩 클래스를 사용하는 방법을 참조하세요.

참고 항목

- String
- Char
- Rune
- 세계화 및 지역화

.NET에서 문자 인코딩 클래스를 사용하는 방법

아티클 • 2025. 03. 23.

이 문서에서는 다양한 인코딩 체계를 사용하여 .NET에서 제공하는 클래스를 사용하여 텍스트를 인코딩하고 디코딩하는 방법을 설명합니다. 지침에서는 .NET의 문자 인코딩 소개를 읽은 것으로 가정합니다.

인코더 및 디코더

.NET은 다양한 인코딩 시스템을 사용하여 텍스트를 인코딩하고 디코딩하는 인코딩 클래스를 제공합니다. 예를 들어 UTF8Encoding 클래스는 UTF-8에서 인코딩 및 디코딩하는 규칙을 설명합니다. .NET은 `string` 인스턴스에 UTF-16 인코딩(UnicodeEncoding 클래스로 표시됨)을 사용합니다. 인코더 및 디코더는 다른 인코딩 스키마에 사용할 수 있습니다.

인코딩 및 디코딩에는 유효성 검사도 포함될 수 있습니다. 예를 들어 UnicodeEncoding 클래스는 서로게이트 범위의 모든 `char` 인스턴스를 검사하여 유효한 서로게이트 쌍에 있는지 확인합니다. 대체 전략은 인코더가 잘못된 문자를 처리하는 방법 또는 디코더가 잘못된 바이트를 처리하는 방법을 결정합니다.

> **⚠ 경고**
>
> .NET 인코딩 클래스는 문자 데이터를 저장하고 변환하는 방법을 제공합니다. 문자열 형식으로 이진 데이터를 저장하는 데 사용하면 안 됩니다. 사용되는 인코딩에 따라 인코딩 클래스를 사용하여 이진 데이터를 문자열 형식으로 변환하면 예기치 않은 동작이 발생하며 부정확하거나 손상된 데이터가 생성될 수 있습니다. 이진 데이터를 문자열 형식으로 변환하려면 Convert.ToBase64String 메서드를 사용합니다.

.NET의 모든 문자 인코딩 클래스는 모든 문자 인코딩에 공통된 기능을 정의하는 추상 클래스인 System.Text.Encoding 클래스에서 상속됩니다. .NET에서 구현된 개별 인코딩 개체에 액세스하려면 다음을 수행합니다.

- .NET(ASCII, UTF-7, UTF-8, UTF-16 및 UTF-32)에서 사용할 수 있는 표준 문자 인코딩을 나타내는 개체를 반환하는 Encoding 클래스의 정적 속성을 사용합니다. 예를 들어 Encoding.Unicode 속성은 UnicodeEncoding 개체를 반환합니다. 각 개체는 대체 대체를 사용하여 인코딩할 수 없는 문자열과 디코딩할 수 없는 바이트를 처리합니다. 자세한 내용은 대체 옵션을 참조하세요.

- 인코딩의 클래스 생성자를 호출합니다. 이러한 방식으로 ASCII, UTF-7, UTF-8, UTF-16 및 UTF-32 인코딩에 대한 개체를 인스턴스화할 수 있습니다. 기본적으로 각 개체는 대체 대체를 사용하여 인코딩할 수 없는 문자열과 디코딩할 수 없는 바이트를 처리하지만 대신 예외를 throw하도록 지정할 수 있습니다. 자세한 내용은 대체 대체 및 예외 대체참조하세요.

- Encoding(Int32) 생성자를 호출하고 인코딩을 나타내는 정수로 전달합니다. 표준 인코딩 개체는 대체 대체를 사용하며, 코드 페이지와 DBCS(더블바이트 문자 집합) 인코딩 개체는 인코딩할 수 없는 문자열과 디코딩할 수 없는 바이트를 처리하는 데 가장 적합한 대체를 사용합니다. 자세한 내용은 최적 대체참조하세요.

- .NET에서 사용할 수 있는 표준, 코드 페이지 또는 DBCS 인코딩을 반환하는 Encoding.GetEncoding 메서드를 호출합니다. 오버로드를 사용하면 인코더와 디코더 모두에 대한 대체 개체를 지정할 수 있습니다.

Encoding.GetEncodings 메서드를 호출하여 .NET에서 사용할 수 있는 모든 인코딩에 대한 정보를 검색할 수 있습니다. .NET은 다음 표에 나열된 문자 인코딩 구성표를 지원합니다.

테이블 확장

인코딩 클래스	설명
ASCII	바이트의 하위 7비트를 사용하여 제한된 문자 범위를 인코딩합니다. 이 인코딩은 문자 값 `U+0000`부터 `U+007F`까지의 범위만 지원하기 때문에 대부분의 경우 국제화된 애플리케이션에는 적합하지 않습니다.
UTF-7	문자를 7비트 ASCII 문자의 시퀀스로 나타냅니다. 비 ASCII 유니코드 문자는 ASCII 문자의 이스케이프 시퀀스로 표시됩니다. UTF-7은 이메일 및 뉴스 그룹과 같은 프로토콜을 지원합니다. 그러나 UTF-7은 특히 안전하거나 강력하지 않습니다. 경우에 따라 한 비트를 변경하면 전체 UTF-7 문자열의 해석이 근본적으로 변경됩니다. 다른 경우에는 다른 UTF-7 문자열이 동일한 텍스트를 인코딩할 수 있습니다. ASCII가 아닌 문자를 포함하는 시퀀스의 경우 UTF-7에는 UTF-8보다 더 많은 공간이 필요하며 인코딩/디코딩 속도가 느립니다. 따라서 가능하면 UTF-7 대신 UTF-8을 사용해야 합니다.
UTF-8	각 유니코드 코드 지점을 1~4바이트의 시퀀스로 나타냅니다. UTF-8은 8비트 데이터 크기를 지원하며 많은 기존 운영 체제에서 잘 작동합니다. ASCII 문자 범위의 경우 UTF-8은 ASCII 인코딩과 동일하며 더 광범위한 문자 집합을 허용합니다. 그러나 CJK(중국어Japanese-Korean) 스크립트의 경우 UTF-8에는 각 문자에 대해 3바이트가 필요할 수 있으며 UTF-16보다 더 큰 데이터 크기가 발생할 수 있습니다. 경우에 따라 HTML 태그와 같은 ASCII 데이터의 양이 CJK 범위의 증가된 크기를 정당화합니다.
UTF-16	각 유니코드 코드 지점을 하나 또는 두 개의 16비트 정수 시퀀스로 나타냅니다. 유니코드 보조 문자(U+10000 이상)에는 두 개의 UTF-16 서로게이트 코드 포인트가 필요하지만 가장 일반적인 유니코드 문자에는 UTF-16 코드 포인트가 하나만 필요합니다. little-endian 및 big-endian 바이트 순서가 모두 지원됩니다. UTF-16 인코딩은 공용 언

인코딩 클래스	설명
	어 런타임에서 Char 및 String 값을 나타내는 데 사용되며 Windows 운영 체제에서 `WCHAR` 값을 나타내는 데 사용됩니다.
UTF-32	각 유니코드 코드 지점을 32비트 정수로 나타냅니다. little-endian 및 big-endian 바이트 순서가 모두 지원됩니다. UTF-32 인코딩은 인코딩된 공간이 너무 중요한 운영 체제에서 UTF-16 인코딩의 서로게이트 코드 포인트 동작을 방지하려는 경우에 사용됩니다. 디스플레이에 렌더링된 단일 문자 모양은 여전히 둘 이상의 UTF-32 문자로 인코딩할 수 있습니다.
ANSI/ISO 인코딩	다양한 코드 페이지를 지원합니다. Windows 운영 체제에서 코드 페이지는 특정 언어 또는 언어 그룹을 지원하는 데 사용됩니다. .NET에서 지원하는 코드 페이지를 나열하는 테이블은 Encoding 클래스를 참조하세요. Encoding.GetEncoding(Int32) 메서드를 호출하여 특정 코드 페이지에 대한 인코딩 개체를 검색할 수 있습니다. 코드 페이지에는 256개의 코드 포인트가 포함되어 있으며 0부터 시작하는 것입니다. 대부분의 코드 페이지에서 코드 포인트 0~127은 ASCII 문자 집합을 나타내고 코드 포인트 128~255는 코드 페이지 간에 크게 다릅니다. 예를 들어 코드 페이지 1252는 영어, 독일어 및 프랑스어를 비롯한 라틴어 쓰기 시스템에 대한 문자를 제공합니다. 코드 페이지 1252의 마지막 128개 코드 포인트에는 악센트 문자가 포함되어 있습니다. 코드 페이지 1253은 그리스어 쓰기 시스템에 필요한 문자 코드를 제공합니다. 코드 페이지 1253의 마지막 128개 코드 포인트에는 그리스 문자가 포함되어 있습니다. 따라서 ANSI 코드 페이지를 사용하는 애플리케이션은 참조된 코드 페이지를 나타내는 식별자를 포함하지 않는 한 그리스어와 독일어를 동일한 텍스트 스트림에 저장할 수 없습니다.
DBCS(더블 바이트 문자 집합) 인코딩	256자를 초과하는 중국어, 일본어 및 한국어와 같은 언어를 지원합니다. DBCS에서 코드 포인트 쌍(더블 바이트)은 각 문자를 나타냅니다. Encoding.IsSingleByte 속성은 DBCS 인코딩에 대한 `false` 반환합니다. Encoding.GetEncoding(Int32) 메서드를 호출하여 특정 DBCS에 대한 인코딩 개체를 검색할 수 있습니다. 애플리케이션이 DBCS 데이터를 처리할 때 DBCS 문자의 첫 번째 바이트(리드 바이트)는 바로 뒤에 있는 후행 바이트와 함께 처리됩니다. 단일 쌍의 더블 바이트 코드 포인트는 코드 페이지에 따라 다른 문자를 나타낼 수 있으므로 이 체계는 동일한 데이터 스트림에서 일본어 및 중국어와 같은 두 언어의 조합을 허용하지 않습니다.

이러한 인코딩을 사용하면 레거시 애플리케이션에서 가장 일반적으로 사용되는 인코딩뿐만 아니라 유니코드 문자로 작업할 수 있습니다. 또한 Encoding 파생되는 클래스를 정의하고 해당 멤버를 재정의하여 사용자 지정 인코딩을 만들 수 있습니다.

.NET Core 인코딩 지원

기본적으로 .NET Core는 코드 페이지 28591 및 유니코드 인코딩(예: UTF-8 및 UTF-16) 이외의 코드 페이지 인코딩을 사용할 수 없습니다. 그러나 .NET을 대상으로 하는 표준 Windows 앱에 있는 코드 페이지 인코딩을 앱에 추가할 수 있습니다. 자세한 내용은 CodePagesEncodingProvider 항목을 참조하세요.

인코딩 클래스 선택

애플리케이션에서 사용할 인코딩을 선택할 기회가 있는 경우 유니코드 인코딩(UTF8Encoding 또는 UnicodeEncoding)을 사용해야 합니다. (.NET은 세 번째 유니코드 인코딩(UTF32Encoding.)도 지원합니다.

ASCII 인코딩(ASCIIEncoding)을 사용하려는 경우 대신 UTF8Encoding 선택합니다. 두 인코딩은 ASCII 문자 집합에 대해 동일하지만 UTF8Encoding 다음과 같은 이점이 있습니다.

- 모든 유니코드 문자를 나타낼 수 있는 반면, ASCIIEncoding U+0000과 U+007F 사이의 유니코드 문자 값만 지원합니다.
- 오류 검색 및 더 나은 보안을 제공합니다.
- 가능한 한 빨리 조정되었으며 다른 인코딩보다 빨라야 합니다. 완전히 ASCII인 콘텐츠의 경우에도 UTF8Encoding 사용하여 수행되는 작업은 ASCIIEncoding수행되는 작업보다 빠릅니다.

레거시 애플리케이션에만 ASCIIEncoding 사용하는 것이 좋습니다. 그러나 레거시 애플리케이션의 경우에도 다음과 같은 이유로 UTF8Encoding 더 나은 선택일 수 있습니다(기본 설정 가정).

- 애플리케이션에 엄격하게 ASCII가 아닌 콘텐츠가 있고 ASCIIEncoding인코딩하는 경우 각 비 ASCII 문자는 물음표(?)로 인코딩됩니다. 애플리케이션이 이 데이터를 디코딩하면 정보가 손실됩니다.
- 애플리케이션에 엄격하게 ASCII가 아닌 콘텐츠가 있고 UTF8Encoding사용하여 인코딩하는 경우 ASCII로 해석되는 경우 결과를 이해할 수 없는 것처럼 보입니다. 그러나 애플리케이션이 UTF-8 디코더를 사용하여 이 데이터를 디코딩하는 경우 데이터는 왕복을 성공적으로 수행합니다.

웹 애플리케이션에서 웹 요청에 대한 응답으로 클라이언트로 전송된 문자는 클라이언트에 사용되는 인코딩을 반영해야 합니다. 대부분의 경우 사용자가 예상하는 인코딩에 텍스트를 표시하려면 HttpResponse.ContentEncoding 속성을 HttpRequest.ContentEncoding 속성에서 반환된 값으로 설정해야 합니다.

인코딩 개체 사용

인코더는 문자 문자열(가장 일반적으로 유니코드 문자)을 해당하는 숫자(바이트)로 변환합니다. 예를 들어 ASCII 인코더를 사용하여 유니코드 문자를 ASCII로 변환하여 콘솔에 표시할 수 있습니다. 변환을 수행하려면 Encoding.GetBytes 메서드를 호출합니다. 인코딩을 수행하기 전에 인코딩된 문자를 저장하는 데 필요한 바이트 수를 확인하려면 GetByteCount 메서드를 호출할 수 있습니다.

다음 예제에서는 단일 바이트 배열을 사용하여 두 개의 별도 작업에서 문자열을 인코딩합니다. 다음 ASCII로 인코딩된 바이트 집합에 대한 바이트 배열의 시작 위치를 나타내는 인덱스를 유지 관리합니다. 바이트 배열이 인코딩된 문자열을 수용할 수 있을 만큼 충분히 큰지 확인하기 위해 ASCIIEncoding.GetByteCount(String) 메서드를 호출합니다. 그런 다음 ASCIIEncoding.GetBytes(String, Int32, Int32, Byte[], Int32) 메서드를 호출하여 문자열의 문자를 인코딩합니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings= { "This is the first sentence. ",
                          "This is the second sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;

      // Create array of adequate size.
      byte[] bytes = new byte[49];
      // Create index for current position of array.
      int index = 0;

      Console.WriteLine("Strings to encode:");
      foreach (var stringValue in strings) {
         Console.WriteLine($"   {stringValue}");

         int count = asciiEncoding.GetByteCount(stringValue);
         if (count + index >=  bytes.Length)
            Array.Resize(ref bytes, bytes.Length + 50);

         int written = asciiEncoding.GetBytes(stringValue, 0,
                                              stringValue.Length,
                                              bytes, index);

         index = index + written;
      }
      Console.WriteLine("\nEncoded bytes:");
      Console.WriteLine("{0}", ShowByteValues(bytes, index));
      Console.WriteLine();

      // Decode Unicode byte array to a string.
      string newString = asciiEncoding.GetString(bytes, 0, index);
      Console.WriteLine($"Decoded: {newString}");
   }

   private static string ShowByteValues(byte[] bytes, int last )
   {
      string returnString = "   ";
      for (int ctr = 0; ctr <= last - 1; ctr++) {
```

```
         if (ctr % 20 == 0)
            returnString += "\n   ";
         returnString += String.Format("{0:X2} ", bytes[ctr]);
      }
      return returnString;
   }
}
// The example displays the following output:
//       Strings to encode:
//          This is the first sentence.
//          This is the second sentence.
//
//       Encoded bytes:
//
//          54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//          6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
//          73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
//
//       Decoded: This is the first sentence. This is the second sentence.
```

디코더는 특정 문자 인코딩을 반영하는 바이트 배열을 문자 배열 또는 문자열의 문자 집합으로 변환합니다. 바이트 배열을 문자 배열로 디코딩하려면 Encoding.GetChars 메서드를 호출합니다. 바이트 배열을 문자열로 디코딩하려면 GetString 메서드를 호출합니다. 디코딩을 수행하기 전에 디코딩된 바이트를 저장하는 데 필요한 문자 수를 확인하려면 GetCharCount 메서드를 호출할 수 있습니다.

다음 예제에서는 세 개의 문자열을 인코딩한 다음 단일 문자 배열로 디코딩합니다. 디코딩된 다음 문자 집합에 대한 문자 배열의 시작 위치를 나타내는 인덱스를 유지 관리합니다. GetCharCount 메서드를 호출하여 문자 배열이 디코딩된 모든 문자를 수용할 수 있을 만큼 충분히 큰지 확인합니다. 그런 다음 ASCIIEncoding.GetChars(Byte[], Int32, Int32, Char[], Int32) 메서드를 호출하여 바이트 배열을 디코딩합니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings = { "This is the first sentence. ",
                           "This is the second sentence. ",
                           "This is the third sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;
      // Array to hold encoded bytes.
      byte[] bytes;
      // Array to hold decoded characters.
      char[] chars = new char[50];
      // Create index for current position of character array.
```

```
      int index = 0;

      foreach (var stringValue in strings) {
         Console.WriteLine($"String to Encode: {stringValue}");
         // Encode the string to a byte array.
         bytes = asciiEncoding.GetBytes(stringValue);
         // Display the encoded bytes.
         Console.Write("Encoded bytes: ");
         for (int ctr = 0; ctr < bytes.Length; ctr++)
            Console.Write(" {0}{1:X2}",
                          ctr % 20 == 0 ? Environment.NewLine : "",
                          bytes[ctr]);
         Console.WriteLine();

         // Decode the bytes to a single character array.
         int count = asciiEncoding.GetCharCount(bytes);
         if (count + index >=  chars.Length)
            Array.Resize(ref chars, chars.Length + 50);

         int written = asciiEncoding.GetChars(bytes, 0,
                                              bytes.Length,
                                              chars, index);
         index = index + written;
         Console.WriteLine();
      }

      // Instantiate a single string containing the characters.
      string decodedString = new string(chars, 0, index - 1);
      Console.WriteLine("Decoded string: ");
      Console.WriteLine(decodedString);
   }
}
// The example displays the following output:
//    String to Encode: This is the first sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the second sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
//    65 6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the third sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    Decoded string:
//    This is the first sentence. This is the second sentence. This is the
third sentence.
```

Encoding 파생된 클래스의 인코딩 및 디코딩 메서드는 전체 데이터 집합에서 작동하도록 설계되었습니다. 즉, 인코딩 또는 디코딩할 모든 데이터가 단일 메서드 호출에 제공됩니다. 그러나 경우에 따라 스트림에서 데이터를 사용할 수 있으며 인코딩 또는 디코딩할 데이터는 별도의 읽기 작업에서만 사용할 수 있습니다. 이렇게 하려면 이전 호출에서 저장된 상태를 기억하기 위해 인코딩 또는 디코딩 작업이 필요합니다. Encoder 및 Decoder 파생된 클래스의 메서드는 여러 메서드 호출에 걸쳐 있는 인코딩 및 디코딩 작업을 처리할 수 있습니다.

특정 인코딩에 대한 Encoder 개체는 해당 인코딩의 Encoding.GetEncoder 속성에서 사용할 수 있습니다. 특정 인코딩에 대한 Decoder 개체는 해당 인코딩의 Encoding.GetDecoder 속성에서 사용할 수 있습니다. 디코딩 작업의 경우 Decoder 파생된 클래스에는 Decoder.GetChars 메서드가 포함되지만 Encoding.GetString해당하는 메서드는 없습니다.

다음 예제에서는 유니코드 바이트 배열을 디코딩하기 위해 Encoding.GetString 메서드와 Decoder.GetChars 메서드를 사용하는 것의 차이점을 보여 줍니다. 이 예제에서는 일부 유니코드 문자가 포함된 문자열을 파일에 인코딩한 다음 두 디코딩 메서드를 사용하여 한 번에 10바이트를 디코딩합니다. 서로게이트 쌍은 10번째 및 11바이트에서 발생하므로 별도의 메서드 호출에서 디코딩됩니다. 출력에서 볼 수 있듯이 Encoding.GetString 메서드는 바이트를 올바르게 디코딩할 수 없으며 대신 U+FFFD(REPLACEMENT CHARACTER)로 바꿉니다. 반면에 Decoder.GetChars 메서드는 바이트 배열을 디코딩하여 원래 문자열을 가져올 수 있습니다.

C#

```
using System;
using System.IO;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Use default replacement fallback for invalid encoding.
      UnicodeEncoding enc = new UnicodeEncoding(true, false, false);

      // Define a string with various Unicode characters.
      string str1 = "AB YZ 19 \uD800\udc05 \u00e4";
      str1 += "Unicode characters. \u00a9 \u010C s \u0062\u0308";
      Console.WriteLine("Created original string...\n");

      // Convert string to byte array.
      byte[] bytes = enc.GetBytes(str1);

      FileStream fs = File.Create(@".\characters.bin");
      BinaryWriter bw = new BinaryWriter(fs);
      bw.Write(bytes);
      bw.Close();
```

```
      // Read bytes from file.
      FileStream fsIn = File.OpenRead(@".\characters.bin");
      BinaryReader br = new BinaryReader(fsIn);

      const int count = 10;            // Number of bytes to read at a time.
      byte[] bytesRead = new byte[10]; // Buffer (byte array).
      int read;                        // Number of bytes actually read.
      string str2 = String.Empty;      // Decoded string.

      // Try using Encoding object for all operations.
      do {
         read = br.Read(bytesRead, 0, count);
         str2 += enc.GetString(bytesRead, 0, read);
      } while (read == count);
      br.Close();
      Console.WriteLine("Decoded string using
UnicodeEncoding.GetString()...");
      CompareForEquality(str1, str2);
      Console.WriteLine();

      // Use Decoder for all operations.
      fsIn = File.OpenRead(@".\characters.bin");
      br = new BinaryReader(fsIn);
      Decoder decoder = enc.GetDecoder();
      char[] chars = new char[50];
      int index = 0;                   // Next character to write in array.
      int written = 0;                 // Number of chars written to array.
      do {
         read = br.Read(bytesRead, 0, count);
         if (index + decoder.GetCharCount(bytesRead, 0, read) - 1 >=
chars.Length)
            Array.Resize(ref chars, chars.Length + 50);

         written = decoder.GetChars(bytesRead, 0, read, chars, index);
         index += written;
      } while (read == count);
      br.Close();
      // Instantiate a string with the decoded characters.
      string str3 = new String(chars, 0, index);
      Console.WriteLine("Decoded string using
UnicodeEncoding.Decoder.GetString()...");
      CompareForEquality(str1, str3);
   }

   private static void CompareForEquality(string original, string decoded)
   {
      bool result = original.Equals(decoded);
      Console.WriteLine("original = decoded: {0}",
                        original.Equals(decoded, StringComparison.Ordinal));
      if (! result) {
         Console.WriteLine("Code points in original string:");
         foreach (var ch in original)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
         Console.WriteLine();
```

```
         Console.WriteLine("Code points in decoded string:");
         foreach (var ch in decoded)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    Created original string...
//
//    Decoded string using UnicodeEncoding.GetString()...
//    original = decoded: False
//    Code points in original string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055
006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E
0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//    Code points in decoded string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055
006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E
0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//
//    Decoded string using UnicodeEncoding.Decoder.GetString()...
//    original = decoded: True
```

대체 전략 선택

메서드가 문자를 인코딩하거나 디코딩하려고 하지만 매핑이 없는 경우 실패한 매핑을 처리하는 방법을 결정하는 대체 전략을 구현해야 합니다. 대체 전략에는 다음 세 가지 유형이 있습니다.

- 가장 적합한 대체
- 대체 백업
- 예외 폴백

> ⓘ **중요**
>
> 인코딩 작업에서 가장 일반적인 문제는 유니코드 문자를 특정 코드 페이지 인코딩에 매핑할 수 없는 경우에 발생합니다. 디코딩 작업에서 가장 일반적인 문제는 잘못된 바이트 시퀀스를 유효한 유니코드 문자로 변환할 수 없는 경우에 발생합니다. 이러

한 이유로 특정 인코딩 개체에서 사용하는 대체 전략을 알아야 합니다. 가능하면 개체를 인스턴스화할 때 인코딩 개체에서 사용하는 대체 전략을 지정해야 합니다.

Best-Fit 대체

대상 인코딩에서 문자에 정확히 일치하는 항목이 없는 경우 인코더는 유사한 문자에 매핑할 수 있습니다. (가장 적합한 대체는 주로 디코딩 문제가 아닌 인코딩입니다. 유니코드에 성공적으로 매핑할 수 없는 문자가 포함된 코드 페이지는 거의 없습니다.) 가장 적합한 대체는 Encoding.GetEncoding(Int32) 및 Encoding.GetEncoding(String) 오버로드에서 검색되는 코드 페이지 및 더블 바이트 문자 집합 인코딩의 기본값입니다.

> **참고**
>
> 이론적으로 .NET(UTF8Encoding, UnicodeEncoding및 UTF32Encoding)에서 제공되는 유니코드 인코딩 클래스는 모든 문자 집합의 모든 문자를 지원하므로 가장 적합한 대체 문제를 제거하는 데 사용할 수 있습니다.

가장 적합한 전략은 코드 페이지에 따라 다릅니다. 예를 들어 일부 코드 페이지의 경우 전체 너비 라틴 문자는 더 일반적인 반자 라틴 문자에 매핑됩니다. 다른 코드 페이지의 경우 이 매핑이 수행되지 않습니다. 공격적인 베스트 핏 전략에서도 일부 인코딩의 일부 문자에는 상상할 수 있는 적합성이 없습니다. 예를 들어 중국어 표기법은 코드 페이지 1252에 대한 적절한 매핑이 없습니다. 이 경우 대체 문자열이 사용됩니다. 기본적으로 이 문자열은 단일 질문 표시(U+003F)에 불과합니다.

> **참고**
>
> 최적 전략은 자세히 문서화되지 않습니다. 그러나 여러 코드 페이지는 **유니코드 컨소시엄의** 웹 사이트에 설명되어 있습니다. 매핑 파일을 해석하는 방법에 대한 설명은 해당 폴더의 **readme.txt** 파일을 검토하세요.

다음 예제에서는 코드 페이지 1252(서유럽 언어용 Windows 코드 페이지)를 사용하여 가장 적합한 매핑 및 단점을 보여 줍니다. Encoding.GetEncoding(Int32) 메서드는 코드 페이지 1252의 인코딩 개체를 검색하는 데 사용됩니다. 기본적으로 지원하지 않는 유니코드 문자에 가장 적합한 매핑을 사용합니다. 이 예제에서는 세 개의 비 ASCII 문자(CIRCLED LATIN CAPITAL LETTER S(U+24C8), SUPERSCRIPT FIVE(U+2075) 및 INFINITY(U+221E)를 공백으로 구분하여 포함하는 문자열을 인스턴스화합니다. 예제의 출력에서 볼 수 있듯이 문자열이 인코딩되면 세 개의 원래 비공간 문자가 물음표(U+003F), DIGIT FIVE(U+0035) 및 DIGIT EIGHT(U+0038)으로 바뀝니다. DIGIT EIGHT은 지원되지 않는 INFINITY 문자를

대체하지 않으며, QUESTION MARK는 원래 문자에 대한 매핑을 사용할 수 없음을 나타냅니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Get an encoding for code page 1252 (Western Europe character set).
      Encoding cp1252 = Encoding.GetEncoding(1252);

      // Define and display a string.
      string str = "\u24c8 \u2075 \u221e";
      Console.WriteLine("Original string: " + str);
      Console.Write("Code points in string: ");
      foreach (var ch in str)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode a Unicode string.
      Byte[] bytes = cp1252.GetBytes(str);
      Console.Write("Encoded bytes: ");
      foreach (byte byt in bytes)
         Console.Write("{0:X2} ", byt);
      Console.WriteLine("\n");

      // Decode the string.
      string str2 = cp1252.GetString(bytes);
      Console.WriteLine("String round-tripped: {0}", str.Equals(str2));
      if (! str.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
      }
   }
}
// The example displays the following output:
//       Original string: Ⓢ ⁵ ∞
//       Code points in string: 24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 35 20 38
//
//       String round-tripped: False
//       ? 5 8
//       003F 0020 0035 0020 0038
```

가장 적합한 매핑은 유니코드 데이터를 코드 페이지 데이터로 인코딩하는 Encoding 개체의 기본 동작이며 이 동작을 사용하는 레거시 애플리케이션이 있습니다. 그러나 대부분의 새 애플리케이션은 보안상의 이유로 가장 적합한 동작을 피해야 합니다. 예를 들어 애플리케이션은 가장 적합한 인코딩을 통해 도메인 이름을 배치해서는 안 됩니다.

> ⓘ **참고**
>
> 인코딩에 대한 사용자 지정 최적 대체 매핑을 구현할 수도 있습니다. 자세한 내용은 **사용자 지정 대체 전략 구현** 섹션을 참조하세요.

인코딩 객체의 기본 설정이 최적의 대체 옵션인 경우, Encoding.GetEncoding(Int32, EncoderFallback, DecoderFallback) 또는 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 오버로드를 호출하여 Encoding 객체를 검색할 때 다른 대체 전략을 선택할 수 있습니다. 다음 섹션에는 코드 페이지 1252에 매핑할 수 없는 각 문자를 별표(*)로 바꾸는 예제가 포함되어 있습니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding cp1252r = Encoding.GetEncoding(1252,
                                 new EncoderReplacementFallback("*"),
                                 new DecoderReplacementFallback("*"));

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine();

      byte[] bytes = cp1252r.GetBytes(str1);
      string str2 = cp1252r.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
```

```
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

대체 대비책

문자가 대상 구성표에 정확히 일치하지 않지만 매핑할 수 있는 적절한 문자가 없는 경우 애플리케이션은 대체 문자 또는 문자열을 지정할 수 있습니다. 유니코드 디코더의 기본 동작으로, 디코딩할 수 없는 2 바이트 시퀀스를 REPLACEMENT_CHARACTER(U+FFFD)로 바꿉니다. 또한 인코딩하거나 디코딩할 수 없는 각 문자를 물음표로 바꾸는 ASCIIEncoding 클래스의 기본 동작이기도 합니다. 다음 예제에서는 이전 예제의 유니코드 문자열에 대한 문자 대체를 보여 줍니다. 출력에서 알 수 있듯이 ASCII 바이트 값으로 디코딩할 수 없는 각 문자는 물음표의 ASCII 코드인 0x3F 대체됩니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.ASCII;

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode the original string using the ASCII encoder.
      byte[] bytes = enc.GetBytes(str1);
      Console.Write("Encoded bytes: ");
      foreach (var byt in bytes)
         Console.Write("{0:X2} ", byt);
      Console.WriteLine("\n");

      // Decode the ASCII bytes.
      string str2 = enc.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
```

```
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 3F 20 3F
//
//       Round-trip: False
//       ? ? ?
//       003F 0020 003F 0020 003F
```

.NET에는 문자가 인코딩 또는 디코딩 작업에서 정확하게 매핑되지 않는 경우 대체 문자열을 대체하는 EncoderReplacementFallback 및 DecoderReplacementFallback 클래스가 포함됩니다. 기본적으로 이 대체 문자열은 물음표이지만 클래스 생성자 오버로드를 호출하여 다른 문자열을 선택할 수 있습니다. 일반적으로 대체 문자열은 단일 문자이지만 요구 사항은 아닙니다. 다음은 별표(*)를 대체 문자열로 사용하는 EncoderReplacementFallback 개체를 인스턴스화하여 코드 페이지 1252 인코더의 동작을 변경하는 예제입니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding cp1252r = Encoding.GetEncoding(1252,
                                    new EncoderReplacementFallback("*"),
                                    new DecoderReplacementFallback("*"));

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine();

      byte[] bytes = cp1252r.GetBytes(str1);
      string str2 = cp1252r.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
```

```
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

> ⓘ **참고**
>
> 인코딩에 대한 대체 클래스를 구현할 수도 있습니다. 자세한 내용은 **사용자 지정 대체 전략 구현** 섹션을 참조하세요.

물음표(U+003F) 외에도 유니코드 대체 문자(U+FFFD)는 일반적으로 대체 문자열로 사용되며, 특히 유니코드 문자로 변환할 수 없는 바이트 시퀀스를 디코딩하는 경우에 특히 유용합니다. 그러나 대체 문자열을 자유롭게 선택할 수 있으며 여러 문자를 포함할 수 있습니다.

예외 대체

인코더는 가장 적합한 대체 또는 대체 문자열을 제공하는 대신 문자 집합을 인코딩할 수 없는 경우 EncoderFallbackException throw할 수 있으며, 디코더는 바이트 배열을 디코딩할 수 없는 경우 DecoderFallbackException throw할 수 있습니다. 인코딩 및 디코딩 작업에서 예외를 throw하려면 EncoderExceptionFallback 개체와 DecoderExceptionFallback 개체를 각각 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 메서드에 제공합니다. 다음 예제에서는 ASCIIEncoding 클래스를 사용한 예외 대체를 보여 줍니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.GetEncoding("us-ascii",
                                          new EncoderExceptionFallback(),
                                          new DecoderExceptionFallback());

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
```

```
      Console.WriteLine("\n");

      // Encode the original string using the ASCII encoder.
      byte[] bytes = {};
      try {
         bytes = enc.GetBytes(str1);
         Console.Write("Encoded bytes: ");
         foreach (var byt in bytes)
            Console.Write("{0:X2} ", byt);

         Console.WriteLine();
      }
      catch (EncoderFallbackException e) {
         Console.Write("Exception: ");
         if (e.IsUnknownSurrogate())
            Console.WriteLine("Unable to encode surrogate pair 0x{0:X4}
0x{1:X3} at index {2}.",
                              Convert.ToUInt16(e.CharUnknownHigh),
                              Convert.ToUInt16(e.CharUnknownLow),
                              e.Index);
         else
            Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                              Convert.ToUInt16(e.CharUnknown),
                              e.Index);
         return;
      }
      Console.WriteLine();

      // Decode the ASCII bytes.
      try {
         string str2 = enc.GetString(bytes);
         Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
         if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
               Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
         }
      }
      catch (DecoderFallbackException e) {
         Console.Write("Unable to decode byte(s) ");
         foreach (byte unknown in e.BytesUnknown)
            Console.Write("0x{0:X2} ");

         Console.WriteLine($"at index {e.Index}");
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Exception: Unable to encode 0x24C8 at index 0.
```

> ⓘ **참고**
>
> 인코딩 작업에 대한 사용자 지정 예외 처리기를 구현할 수도 있습니다. 자세한 내용은 **사용자 지정 대체 전략 구현** 섹션을 참조하세요.

EncoderFallbackException 및 DecoderFallbackException 개체는 예외를 발생시킨 조건에 대한 다음 정보를 제공합니다.

- EncoderFallbackException 개체에는 인코딩할 수 없는 문자가 알 수 없는 서로게이트 쌍(이 경우 메서드가 `true` 반환) 또는 알 수 없는 단일 문자(이 경우 메서드가 `false` 반환)를 나타내는지 여부를 나타내는 IsUnknownSurrogate 메서드가 포함됩니다. 서로게이트 쌍의 문자는 EncoderFallbackException.CharUnknownHigh 및 EncoderFallbackException.CharUnknownLow 속성에서 사용할 수 있습니다. 알 수 없는 단일 문자는 EncoderFallbackException.CharUnknown 속성에서 사용할 수 있습니다. EncoderFallbackException.Index 속성은 인코딩할 수 없는 첫 번째 문자가 발견된 문자열의 위치를 나타냅니다.

- DecoderFallbackException 개체에는 디코딩할 수 없는 바이트 배열을 반환하는 BytesUnknown 속성이 포함되어 있습니다. DecoderFallbackException.Index 속성은 알 수 없는 바이트의 시작 위치를 나타냅니다.

EncoderFallbackException 및 DecoderFallbackException 개체는 예외에 대한 적절한 진단 정보를 제공하지만 인코딩 또는 디코딩 버퍼에 대한 액세스를 제공하지 않습니다. 따라서 인코딩 또는 디코딩 메서드 내에서 잘못된 데이터를 바꾸거나 수정하는 것을 허용하지 않습니다.

사용자 지정 대체 전략 구현

코드 페이지에서 내부적으로 구현되는 가장 적합한 매핑 외에도 .NET에는 대체 전략을 구현하기 위한 다음 클래스가 포함되어 있습니다.

- EncoderReplacementFallback 및 EncoderReplacementFallbackBuffer 사용하여 인코딩 작업의 문자를 바꿉니다.

- DecoderReplacementFallback 및 DecoderReplacementFallbackBuffer 사용하여 디코딩 작업의 문자를 바꿉니다.

- 문자를 인코딩할 수 없을 때 EncoderExceptionFallback와 EncoderExceptionFallbackBuffer을 사용하여 EncoderFallbackException를 발생시킵니다.

- DecoderExceptionFallback 및 DecoderExceptionFallbackBuffer을 사용하여 문자를 디코딩할 수 없는 경우 DecoderFallbackException를 throw합니다.

또한 다음 단계를 수행하여 가장 적합한 대체, 대체 대체 또는 예외 대체를 사용하는 사용자 지정 솔루션을 구현할 수 있습니다.

1. 인코딩 작업에 대한 EncoderFallback 및 디코딩 작업에 대한 DecoderFallback 클래스를 파생합니다.

2. 인코딩 작업에 대한 EncoderFallbackBuffer 및 디코딩 작업에 대한 DecoderFallbackBuffer 클래스를 파생합니다.

3. 예외 대체의 경우 미리 정의된 EncoderFallbackException 및 DecoderFallbackException 클래스가 요구 사항을 충족하지 않는 경우 Exception 또는 ArgumentException같은 예외 개체에서 클래스를 파생합니다.

EncoderFallback 또는 DecoderFallback에서 파생

사용자 지정 대체 솔루션을 구현하려면 인코딩 작업에 대한 EncoderFallback 및 디코딩 작업에 대한 DecoderFallback 상속하는 클래스를 만들어야 합니다. 이러한 클래스의 인스턴스는 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 메서드에 전달되며 인코딩 클래스와 대체 구현 간의 중개자 역할을 합니다.

인코더 또는 디코더에 대한 사용자 지정 대체 솔루션을 만들 때 다음 멤버를 구현해야 합니다.

- EncoderFallback.MaxCharCount 또는 DecoderFallback.MaxCharCount 속성은 최상의 맞춤, 대체 또는 예외 대체가 단일 문자를 바꾸기 위해 반환할 수 있는 가능한 최대 문자 수를 반환합니다. 사용자 지정 예외 대체의 경우 해당 값은 0입니다.

- 사용자 지정 EncoderFallbackBuffer 또는 DecoderFallbackBuffer 구현을 반환하는 EncoderFallback.CreateFallbackBuffer 또는 DecoderFallback.CreateFallbackBuffer 메서드입니다. 인코더는 성공적으로 인코딩할 수 없는 첫 번째 문자를 만나면 메서드를 호출하고, 디코더는 성공적으로 디코딩할 수 없는 첫 번째 바이트를 만나면 메서드를 호출합니다.

EncoderFallbackBuffer 또는 DecoderFallbackBuffer에서 파생

사용자 지정 대체 솔루션을 구현하려면 인코딩 작업에 대해 EncoderFallbackBuffer에서 상속받는 클래스와 디코딩 작업에 대해 DecoderFallbackBuffer에서 상속받는 클래스를 각각 만들어야 합니다. 이러한 클래스의 인스턴스는 EncoderFallback 및 DecoderFallback 클래스의 CreateFallbackBuffer 메서드에 의해 반환됩니다.

EncoderFallback.CreateFallbackBuffer 메서드는 인코더에서 인코딩할 수 없는 첫 번째 문자가 발견되면 호출되고 디코더에서 디코딩할 수 없는 바이트가 하나 이상 발견되면 DecoderFallback.CreateFallbackBuffer 메서드가 호출됩니다. EncoderFallbackBuffer 및 DecoderFallbackBuffer 클래스는 대체 구현을 제공합니다. 각 인스턴스는 인코딩할 수 없는 문자 또는 디코딩할 수 없는 바이트 시퀀스를 대체할 대체 문자가 포함된 버퍼를 나타냅니다.

인코더 또는 디코더에 대한 사용자 지정 대체 솔루션을 만들 때 다음 멤버를 구현해야 합니다.

- EncoderFallbackBuffer.Fallback 또는 DecoderFallbackBuffer.Fallback 메서드입니다. EncoderFallbackBuffer.Fallback 인코더에서 호출하여 인코딩할 수 없는 문자에 대한 정보를 대체 버퍼에 제공합니다. 인코딩할 문자는 서로게이트 쌍일 수 있으므로 이 메서드는 오버로드됩니다. 인코딩할 문자와 문자열의 인덱스가 하나의 오버로드에 전달됩니다. 두 번째 오버로드는 문자열의 인덱스와 함께 높음 및 낮음 서로게이트를 전달합니다. 디코더에서 DecoderFallbackBuffer.Fallback 메서드를 호출하여 디코딩할 수 없는 바이트에 대한 정보를 대체 버퍼에 제공합니다. 이 메서드는 첫 번째 바이트의 인덱스와 함께 디코딩할 수 없는 바이트 배열을 전달합니다. 대체 메서드는 대체 버퍼가 가장 적합하거나 대체 문자 또는 문자를 제공할 수 있는 경우 `true` 반환해야 합니다. 그렇지 않으면 `false` 반환해야 합니다. 예외 처리의 경우, 대체 메서드는 예외를 던져야 합니다.

- 대체 버퍼에서 다음 문자를 가져오기 위해 인코더 또는 디코더에서 반복적으로 호출되는 EncoderFallbackBuffer.GetNextChar 또는 DecoderFallbackBuffer.GetNextChar 메서드입니다. 모든 대체 문자가 반환되면 메서드는 U+0000을 반환해야 합니다.

- 대체 버퍼에 남아 있는 문자 수를 반환하는 EncoderFallbackBuffer.Remaining 또는 DecoderFallbackBuffer.Remaining 속성입니다.

- 대체 버퍼의 현재 위치를 이전 문자로 이동하는 EncoderFallbackBuffer.MovePrevious 또는 DecoderFallbackBuffer.MovePrevious 메서드입니다.

- 대체 버퍼를 다시 초기화하는 EncoderFallbackBuffer.Reset 또는 DecoderFallbackBuffer.Reset 메서드입니다.

대체 구현이 최적 대체 또는 대체 대체(fallback)인 경우, EncoderFallbackBuffer 및 DecoderFallbackBuffer에서 파생된 클래스는 버퍼에 있는 정확한 문자 수와 반환할 다음 문자 인덱스를 나타내는 두 개의 프라이빗 인스턴스 필드를 유지 관리합니다.

EncoderFallback 예제

이전 예제는 대체 대체를 사용하여 ASCII 문자에 해당하지 않는 유니코드 문자를 별표(*)로 바꿉니다. 다음 예제에서는 대신 사용자 지정 최적 대체 구현을 사용하여 비 ASCII 문자의 더 나은 매핑을 제공합니다.

다음 코드는 비ASCII 문자의 최적 적합 매핑을 처리하기 위해 EncoderFallback로부터 파생된 `CustomMapper` 클래스를 정의합니다. 해당 `CreateFallbackBuffer` 메서드는 EncoderFallbackBuffer 구현을 제공하는 `CustomMapperFallbackBuffer` 개체를 반환합니다. `CustomMapper` 클래스는 Dictionary<TKey,TValue> 개체를 사용하여 지원되지 않는 유니코드 문자(키 값) 및 해당 8비트 문자(64비트 정수에 연속 2바이트로 저장됨)의 매핑을 저장합니다. 대체 버퍼에 이 매핑을 사용할 수 있도록 `CustomMapper` 인스턴스는 `CustomMapperFallbackBuffer` 클래스 생성자에 매개 변수로 전달됩니다. 가장 긴 매핑은 유니코드 문자 U+221E의 문자열 "INF"이므로 `MaxCharCount` 속성은 3을 반환합니다.

C#

```
public class CustomMapper : EncoderFallback
{
   public string DefaultString;
   internal Dictionary<ushort, ulong> mapping;

   public CustomMapper() : this("*")
   {
   }

   public CustomMapper(string defaultString)
   {
      this.DefaultString = defaultString;

      // Create table of mappings
      mapping = new Dictionary<ushort, ulong>();
      mapping.Add(0x24C8, 0x53);
      mapping.Add(0x2075, 0x35);
      mapping.Add(0x221E, 0x49004E0046);
   }

   public override EncoderFallbackBuffer CreateFallbackBuffer()
   {
      return new CustomMapperFallbackBuffer(this);
   }

   public override int MaxCharCount
   {
      get { return 3; }
   }
}
```

다음 코드는 EncoderFallbackBuffer파생되는 `CustomMapperFallbackBuffer` 클래스를 정의합니다. 가장 적합한 매핑을 포함하고 `CustomMapper` 인스턴스에 정의된 사전은 해당 클래

스 생성자에서 사용할 수 있습니다. 해당 `Fallback` 메서드는 ASCII 인코더가 인코딩할 수 없는 유니코드 문자가 매핑 사전에 정의된 경우 `true` 반환합니다. 그렇지 않으면 `false` 반환됩니다. 각 대체에 대해 private `count` 변수는 반환할 문자 수를 나타내고, private `index` 변수는 반환할 다음 문자의 문자열 버퍼(`charsToReturn`)의 위치를 나타냅니다.

C#

```
public class CustomMapperFallbackBuffer : EncoderFallbackBuffer
{
   int count = -1;                  // Number of characters to return
   int index = -1;                  // Index of character to return
   CustomMapper fb;
   string charsToReturn;

   public CustomMapperFallbackBuffer(CustomMapper fallback)
   {
      this.fb = fallback;
   }

   public override bool Fallback(char charUnknownHigh, char charUnknownLow,
int index)
   {
      // Do not try to map surrogates to ASCII.
      return false;
   }

   public override bool Fallback(char charUnknown, int index)
   {
      // Return false if there are already characters to map.
      if (count >= 1) return false;

      // Determine number of characters to return.
      charsToReturn = String.Empty;

      ushort key = Convert.ToUInt16(charUnknown);
      if (fb.mapping.ContainsKey(key)) {
         byte[] bytes = BitConverter.GetBytes(fb.mapping[key]);
         int ctr = 0;
         foreach (var byt in bytes) {
            if (byt > 0) {
               ctr++;
               charsToReturn += (char) byt;
            }
         }
         count = ctr;
      }
      else {
         // Return default.
         charsToReturn = fb.DefaultString;
         count = 1;
      }
      this.index = charsToReturn.Length - 1;
```

```
      return true;
   }

   public override char GetNextChar()
   {
      // We'll return a character if possible, so subtract from the count of
chars to return.
      count--;
      // If count is less than zero, we've returned all characters.
      if (count < 0)
         return '\u0000';

      this.index--;
      return charsToReturn[this.index + 1];
   }

   public override bool MovePrevious()
   {
      // Original: if count >= -1 and pos >= 0
      if (count >= -1) {
         count++;
         return true;
      }
      else {
         return false;
      }
   }

   public override int Remaining
   {
      get { return count < 0 ? 0 : count; }
   }

   public override void Reset()
   {
      count = -1;
      index = -1;
   }
}
```

그런 다음, 다음 코드는 `CustomMapper` 개체를 인스턴스화하고 인스턴스를 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 메서드에 전달합니다. 출력은 가장 적합한 대체 구현이 원래 문자열의 ASCII가 아닌 세 문자를 성공적으로 처리했음을 나타냅니다.

C#

```
using System;
using System.Collections.Generic;
using System.Text;

class Program
```

```
{
   static void Main()
   {
      Encoding enc = Encoding.GetEncoding("us-ascii", new CustomMapper(),
new DecoderExceptionFallback());

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      for (int ctr = 0; ctr <= str1.Length - 1; ctr++) {
         Console.Write("{0} ", Convert.ToUInt16(str1[ctr]).ToString("X4"));
         if (ctr == str1.Length - 1)
            Console.WriteLine();
      }
      Console.WriteLine();

      // Encode the original string using the ASCII encoder.
      byte[] bytes = enc.GetBytes(str1);
      Console.Write("Encoded bytes: ");
      foreach (var byt in bytes)
         Console.Write("{0:X2} ", byt);

      Console.WriteLine("\n");

      // Decode the ASCII bytes.
      string str2 = enc.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
```

참고하십시오

- .NET 문자 인코딩 소개
- Encoder
- Decoder
- DecoderFallback
- Encoding
- EncoderFallback
- 세계화 및 지역화

.NET에서 문자열을 비교하는 모범 사례

.NET은 지역화되고 세계화된 애플리케이션을 개발하기 위한 광범위한 지원을 제공하며, 문자열 정렬 및 표시와 같은 일반적인 작업을 수행할 때 현재 문화권 또는 특정 문화권의 규칙을 쉽게 적용할 수 있습니다. 그러나 문자열을 정렬하거나 비교하는 것이 항상 문화권을 구분하는 작업은 아닙니다. 예를 들어 애플리케이션에서 내부적으로 사용되는 문자열은 일반적으로 모든 문화권에서 동일하게 처리되어야 합니다. XML 태그, HTML 태그, 사용자 이름, 파일 경로 및 시스템 개체 이름과 같은 문화적으로 독립적인 문자열 데이터가 문화권을 구분하는 것처럼 해석되는 경우 애플리케이션 코드는 미묘한 버그, 성능 저하 및 경우에 따라 보안 문제가 발생할 수 있습니다.

이 문서에서는 .NET의 문자열 정렬, 비교 및 대/소문자 구분 메서드를 검토하고, 적절한 문자열 처리 메서드를 선택하기 위한 권장 사항을 제시하며, 문자열 처리 메서드에 대한 추가 정보를 제공합니다.

문자열 사용에 대한 권장 사항

.NET을 사용하여 개발할 때 문자열을 비교할 때 다음 권장 사항을 따릅니다.

> **팁**
>
> 다양한 문자열 관련 메서드가 비교를 수행합니다. 예를 들어 String.Equals, , String.CompareString.IndexOf및 String.StartsWith가 있습니다.

- 문자열 작업에 대한 문자열 비교 규칙을 명시적으로 지정하는 오버로드를 사용합니다. 일반적으로 형식의 매개 변수 StringComparison가 있는 메서드 오버로드를 호출하는 작업이 포함됩니다.
- StringComparison.Ordinal 또는 StringComparison.OrdinalIgnoreCase을 문화권에 구애받지 않은 문자열 일치의 안전한 기본값으로 사용하십시오.
- StringComparison.Ordinal 또는 StringComparison.OrdinalIgnoreCase을 비교에 사용하여 성능을 향상시키십시오.
- 출력을 사용자에게 표시할 때 StringComparison.CurrentCulture에 기반한 문자열 작업을 사용합니다.
- 비교가 언어적으로 관련이 없는 경우(예: 기호 StringComparison.Ordinal)에는 문자열 작업 대신 비언어적 StringComparison.OrdinalIgnoreCase 또는 CultureInfo.InvariantCulture 값을 사용하세요.
- 문자열을 비교하기 위해 정규화할 때 String.ToUpperInvariant 메서드 대신 String.ToLowerInvariant 메서드를 사용하십시오.
- 메서드의 오버로드를 사용하여 두 문자열이 String.Equals 같은지 여부를 테스트합니다.

- 문자열의 같음을 확인하지 않고, 문자열을 정렬하기 위해 String.Compare 및 String.CompareTo 메서드를 사용하세요.
- 문화권 구분 서식을 사용하여 숫자 및 날짜와 같은 문자열이 아닌 데이터를 사용자 인터페이스에 표시합니다. 고정 문화권 서식을 사용하여 문자열이 아닌 데이터를 문자열 형식으로 유지합니다.

문자열을 비교할 때는 다음 방법을 사용하지 않습니다.

- 문자열 작업에 대한 문자열 비교 규칙을 명시적으로 또는 암시적으로 지정하지 않는 오버로드를 사용하지 마세요.
- 대부분의 경우를 기준으로 StringComparison.InvariantCulture 문자열 작업을 사용하지 마세요. 몇 가지 예외 중 하나는 언어적으로 의미 있지만 문화적으로 중립적인 데이터를 유지하는 경우입니다.
- 또는 String.Compare 메서드의 CompareTo 오버로드를 사용하고 0의 반환 값을 테스트하여 두 문자열이 같은지 여부를 확인하지 마세요.

팁

CA1307, CA1309 및 CA1310 코드 분석 규칙은 언어 비교자가 의도치 않게 사용되는 호출 사이트를 식별하는 데 도움이 됩니다. 이를 사용하도록 설정하고 위반을 빌드 오류로 표시하려면 프로젝트 파일에서 다음 속성을 설정합니다.

XML

```
<PropertyGroup>
  <AnalysisMode>All</AnalysisMode>
  <WarningsAsErrors>$(WarningsAsErrors);CA1307;CA1309;CA1310</WarningsAsErrors>
</PropertyGroup>
```

명시적으로 문자열 비교 지정

.NET의 문자열 조작 메서드 대부분은 오버로드됩니다. 일반적으로 하나 이상의 오버로드는 기본 설정을 허용하는 반면, 다른 오버로드는 기본값을 허용하지 않고 대신 문자열을 비교하거나 조작할 정확한 방법을 정의합니다. 기본값을 사용하지 않는 대부분의 메서드에는 문화권 및 대/소문자별 문자열 비교에 대한 규칙을 명시적으로 지정하는 열거형인 형식 StringComparison의 매개 변수가 포함됩니다. 다음 표에서는 열거형 멤버에 StringComparison 대해 설명합니다.

테이블 확장

`StringComparison` 멤버	설명
CurrentCulture	현재 문화를 사용하여 대/소문자를 구분하여 비교합니다.

`StringComparison` 멤버	설명
CurrentCultureIgnoreCase	현재 문화에 따라 대/소문자를 구분하지 않고 비교를 수행합니다.
InvariantCulture	불변 문화 구분을 사용하여 대/소문자를 구분하는 비교를 수행합니다.
InvariantCultureIgnoreCase	고정 문화권을 사용하여 대/소문자를 구분하지 않는 비교를 수행합니다.
Ordinal	서수 비교를 수행합니다.
OrdinalIgnoreCase	대/소문자를 구분하지 않는 순서 비교를 수행합니다.

예를 들어 IndexOf 문자 또는 문자열과 일치하는 개체에서 String 부분 문자열의 인덱스를 반환하는 메서드에는 9개의 오버로드가 있습니다.

- IndexOf(Char), IndexOf(Char, Int32)및 IndexOf(Char, Int32, Int32)- 기본적으로 문자열의 문자에 대한 서수(대/소문자를 구분하고 문화권을 구분하지 않는) 검색을 수행합니다.
- IndexOf(String), IndexOf(String, Int32), 및 IndexOf(String, Int32, Int32)는 기본적으로 문자열 내에서 대/소문자와 문화권을 구분하여 부분 문자열을 검색합니다.
- IndexOf(String, StringComparison), IndexOf(String, Int32, StringComparison)및 IndexOf(String, Int32, Int32, StringComparison)- 비교 형식을 지정할 수 있는 형식 StringComparison 의 매개 변수를 포함합니다.

다음과 같은 이유로 기본값을 사용하지 않는 오버로드를 선택하는 것이 좋습니다.

- 기본 매개 변수로 지정된 일부 오버로드(문자열 인스턴스에서 Char를 검색하는 경우)는 서수 방식으로 비교를 수행하는 반면, 다른 오버로드는 문자열 인스턴스에서 문자열을 검색할 때 문화권에 민감한 방식으로 수행됩니다. 어떤 메서드가 어떤 기본값을 사용하는지 기억하기 어렵고 오버로드를 혼동하기 쉽습니다.

- 메서드 호출의 기본값을 사용하는 코드의 의도는 명확하지 않습니다. 기본값을 사용하는 다음 예제에서는 개발자가 실제로 두 문자열의 서수 또는 언어 비교를 의도했는지, 아니면 대/소문자와 "https"의 `url.Scheme` 대/소문자 차이로 인해 같음 테스트가 반환 `false`되는지 여부를 알기 어렵습니다.

  C#

  ```
  Uri url = new("https://learn.microsoft.com/");

  // Incorrect
  if (string.Equals(url.Scheme, "https"))
  {
      // ...Code to handle HTTPS protocol.
  }
  ```

일반적으로 코드의 의도를 명확하게 만들기 때문에 기본값을 사용하지 않는 메서드를 호출하는 것이 좋습니다. 이렇게 하면 코드를 더 읽기 쉽고 쉽게 디버그하고 유지 관리할 수 있습니다. 다음 예제에서는 이전 예제에 대해 제기된 질문을 다룹니다. 서수 비교가 사용되고 경우에 따른 차이가 무시됨을 분명히 합니다.

C#

```
Uri url = new("https://learn.microsoft.com/");

// Correct
if (string.Equals(url.Scheme, "https", StringComparison.OrdinalIgnoreCase))
{
    // ...Code to handle HTTPS protocol.
}
```

문자열 비교의 세부 정보

문자열 비교는 많은 문자열 관련 작업, 특히 정렬 및 같음 테스트의 핵심입니다. 문자열은 정해진 순서로 정렬됩니다. 정렬된 문자열 목록에서 "my"가 "string" 앞에 나타나면 "my"는 "string"보다 작거나 같아야 합니다. 또한 비교는 같음을 암시적으로 정의합니다. 비교 작업은 같음으로 간주되는 문자열에 대해 0을 반환합니다. 좋은 해석은 두 문자열이 다른 문자열보다 작지 않는다는 것입니다. 문자열과 관련된 가장 의미 있는 작업에는 다른 문자열과 비교하고 잘 정의된 정렬 작업을 실행하는 절차 중 하나 또는 둘 다 포함됩니다.

> ⓘ **참고 항목**
>
> **정렬 가중치 테이블**, Windows 운영 체제의 정렬 및 비교 작업에 사용되는 문자 가중치에 대한 정보가 포함된 텍스트 파일 집합 및 Linux 및 macOS용 정렬 가중치 테이블의 최신 버전인 **기본 유니코드 데이터 정렬 요소 테이블을** 다운로드할 수 있습니다. Linux 및 macOS에서 정렬 가중치 테이블의 특정 버전은 시스템에 설치된 **유니코드용 International Components** 라이브러리의 버전에 따라 달라집니다. ICU 버전 및 구현하는 유니코드 버전에 대한 자세한 내용은 **ICU 다운로드를** 참조하세요.

그러나 같음 또는 정렬 순서에 대해 두 문자열을 평가해도 올바른 단일 결과가 생성되지는 않습니다. 결과는 문자열을 비교하는 데 사용되는 조건에 따라 달라집니다. 특히 서수 비교이거나 현재 문화권 또는 고정된 문화 권(영어 기반의 로캘 비의존적 문화권)의 대/소문자 구분 및 정렬 규칙을 기반으로 하는 문자열 비교는 서로 다른 결과를 생성할 수 있습니다.

또한 다른 버전의 .NET을 사용하거나 다른 운영 체제 또는 운영 체제 버전에서 .NET을 사용하는 문자열 비교는 다른 결과를 반환할 수 있습니다. .NET은 지원되는 모든 플랫폼에서 언어 문자열 비교를 위해 ICU(International Components for Unicode) 라이브러리를 사용합니다. 자세한 내용은 문자열 및 유니코드 표준 및 .NET 세계화 및 ICU를 참조하세요.

현재 문화에 맞춰 문자열 비교

한 가지 기준은 문자열을 비교할 때 현재 문화권의 규칙을 사용하는 것입니다. 현재 문화권을 기반으로 하는 비교는 스레드의 현재 문화권 또는 로캘을 사용합니다. 문화 설정이 사용자가 설정하지 않으면 기본적으로 운영 체제의 설정으로 지정됩니다. 데이터가 언어적으로 관련된 경우와 문화권에 민감한 사용자 상호 작용을 반영하는 경우 현재 문화권을 기반으로 하는 비교를 항상 사용해야 합니다.

그러나 .NET의 비교와 대소문자 처리 방식은 지역이나 언어 설정이 변경될 때 변합니다. 애플리케이션이 개발된 컴퓨터와 문화권이 다른 컴퓨터에서 애플리케이션을 실행하거나 실행 중인 스레드가 문화권을 변경할 때 발생합니다. 이 동작은 의도적이지만 많은 개발자에게 명확하지 않습니다. 다음 예제에서는 미국 영어("en-US")와 스웨덴어("sv-SE") 문화권 간의 정렬 순서 차이를 보여 줍니다. "ångström", "Windows" 및 "Visual Studio"라는 단어는 정렬된 문자열 배열의 다른 위치에 표시됩니다.

C#

```
using System.Globalization;

// Words to sort
string[] values= { "able", "ångström", "apple", "Æble",
                   "Windows", "Visual Studio" };

// Current culture
Array.Sort(values);
DisplayArray(values);

// Change culture to Swedish (Sweden)
string originalCulture = CultureInfo.CurrentCulture.Name;
Thread.CurrentThread.CurrentCulture = new CultureInfo("sv-SE");
Array.Sort(values);
DisplayArray(values);

// Restore the original culture
Thread.CurrentThread.CurrentCulture = new CultureInfo(originalCulture);

static void DisplayArray(string[] values)
{
    Console.WriteLine($"Sorting using the {CultureInfo.CurrentCulture.Name}
culture:");

    foreach (string value in values)
        Console.WriteLine($"   {value}");

    Console.WriteLine();
}

// The example displays the following output:
//     Sorting using the en-US culture:
//        able
```

```
//        Æble
//        ångström
//        apple
//        Visual Studio
//        Windows
//
//     Sorting using the sv-SE culture:
//        able
//        apple
//        Visual Studio
//        Windows
//        ångström
//        Æble
```

현재 문화권을 사용하는 대소문자 구분 없는 비교는 문화권에 민감한 비교와 동일합니다. 다만, 이는 스레드의 현재 문화권에 따라 대소문자를 무시합니다. 이 동작은 정렬 순서에서도 나타날 수 있습니다.

현재 문화권 의미 체계를 사용하는 비교는 다음 메서드의 기본값입니다.

- String.Compare 매개 변수를 포함하지 않는 오버로드입니다 StringComparison .
- String.CompareTo 과부하.
- 기본 String.StartsWith(String) 메서드와 String.StartsWith(String, Boolean, CultureInfo)`null`CultureInfo 매개 변수가 있는 메서드.
- 기본 String.EndsWith(String) 메서드와 String.EndsWith(String, Boolean, CultureInfo)`null`CultureInfo 매개 변수가 있는 메서드.
- String.IndexOf 오버로드는 검색 매개 변수로 String을 허용하며, StringComparison 매개 변수가 없습니다.
- String.LastIndexOf 오버로드는 검색 매개 변수로 String을 허용하며, StringComparison 매개 변수가 없습니다.

어떤 경우든 메서드 호출의 의도를 명확하게 하기 위해 매개 변수가 있는 StringComparison 오버로드를 호출하는 것이 좋습니다.

비언어적 문자열 데이터가 언어적으로 해석되거나 특정 문화권의 문자열 데이터가 다른 문화권의 규칙을 사용하여 해석될 때 미묘하고 미묘한 버그가 나타날 수 있습니다. 정식 예제는 Turkish-I 문제입니다.

미국 영어를 포함한 거의 모든 라틴어 알파벳의 경우 문자 "i"(\u0069)는 문자 "I"(\u0049)의 소문자 버전입니다. 이 대/소문자 구분 규칙은 이러한 문화권에서 프로그래밍하는 사용자의 기본값이 됩니다. 그러나 터키어("tr-TR") 알파벳에는 "i"의 대문자 버전인 "점이 있는 I" 문자 "ý"(\u0130)가 포함됩니다. 터키어에 소문자 '점 없는 i' 문자인 'ı'(\u0131)가 포함되어 있고, 대문자로 변환될 때 'I'가 됩니다. 이 동작은 아제르바이잔어("az") 문화권에서도 발생합니다.

따라서 "i"를 대문자로 사용하거나 "I"를 낮추는 것에 대한 가정은 모든 문화권에서 유효하지 않습니다. 문자열 비교 루틴에 기본 오버로드를 사용하는 경우 문화권 간에 차이가 발생합니다. 비

교할 데이터가 비언어적이면 다음 시도에서 "bill" 및 "BILL" 문자열을 대/소문자를 구분하지 않는 비교를 수행하려고 할 때 기본 오버로드를 사용하면 바람직하지 않은 결과를 생성할 수 있습니다.

C#

```
using System.Globalization;

string name = "Bill";

Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
Console.WriteLine($"Culture = {Thread.CurrentThread.CurrentCulture.DisplayName}");
Console.WriteLine($"   Is 'Bill' the same as 'BILL'? {name.Equals("BILL",
StringComparison.OrdinalIgnoreCase)}");
Console.WriteLine($"   Does 'Bill' start with 'BILL'? {name.StartsWith("BILL",
true, null)}");
Console.WriteLine();

Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
Console.WriteLine($"Culture = {Thread.CurrentThread.CurrentCulture.DisplayName}");
Console.WriteLine($"   Is 'Bill' the same as 'BILL'? {name.Equals("BILL",
StringComparison.OrdinalIgnoreCase)}");
Console.WriteLine($"   Does 'Bill' start with 'BILL'? {name.StartsWith("BILL",
true, null)}");

//' The example displays the following output:
//'
//'     Culture = English (United States)
//'        Is 'Bill' the same as 'BILL'? True
//'        Does 'Bill' start with 'BILL'? True
//'
//'     Culture = Turkish (Türkiye)
//'        Is 'Bill' the same as 'BILL'? True
//'        Does 'Bill' start with 'BILL'? False
```

이 비교는 다음 예제와 같이 문화권이 보안에 민감한 설정에서 실수로 사용되는 경우 심각한 문제를 일으킬 수 있습니다. 현재 문화권이 미국 영어인 경우 `IsFileURI("file:")` 메서드를 호출하면 `true`을 반환하지만, 현재 문화권이 터키어인 경우 `false`을 반환합니다. 따라서 터키 시스템에서는 누군가가 "FILE:"로 시작하는 대/소문자를 구분하지 않는 URI에 대한 액세스를 차단하는 보안 조치를 우회할 수 있습니다.

C#

```
public static bool IsFileURI(string path) =>
    path.StartsWith("FILE:", true, null);
```

이 경우 "file:"은 비언어적이고 문화권을 구분하지 않는 식별자로 해석되므로 다음 예제와 같이 코드를 대신 작성해야 합니다.

C#

```
public static bool IsFileURI(string path) =>
    path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase);
```

서수 문자열 처리

메서드 호출에서 StringComparison.Ordinal 값 또는 StringComparison.OrdinalIgnoreCase 값을 지정하면 자연어의 기능이 무시되는 비언어적 비교를 의미합니다. 이러한 StringComparison 값으로 호출되는 메서드는 문화 기준이 있는 대/소문자 또는 동등 테이블을 이용하는 대신 단순한 바이트 비교에 기반해서 문자열 연산 결정을 내립니다. 대부분의 경우 이 방법은 코드를 더 빠르고 안정적으로 만들면서 의도한 문자열 해석에 가장 적합합니다.

순서 비교는 각 문자열의 각 바이트를 언어 해석 없이 비교하는 작업입니다. 예를 들어 "windows"는 "Windows"와 일치하지 않는다. 이는 기본적으로 C 런타임 `strcmp` 함수에 대한 호출입니다. 컨텍스트에서 문자열을 정확히 일치시켜야 한다고 지시하거나 보수적인 일치 정책을 요구할 때 이 비교를 사용합니다. 또한 서수 비교는 결과를 결정할 때 언어 규칙을 적용하지 않으므로 가장 빠른 비교 작업입니다.

`OrdinalIgnoreCase` 비교자는 여전히 문자 단위로 작동하지만 작업을 수행하는 동안 대/소문자 차이를 제거합니다. 비교기 `OrdinalIgnoreCase` 아래에서 문자 쌍 `'d'`과 `'D'`는 *같게* 비교되고, 문자 쌍 `'á'`와 `'Á'`도 마찬가지입니다. 악센트가 없는 문자 `'a'`는 악센트가 있는 문자 `'á'`와 *같지 않게* 비교됩니다.

이에 대한 몇 가지 예는 다음 표에 나와 있습니다.

테이블 확장

문자열 1	문자열 2	`Ordinal` 비교	`OrdinalIgnoreCase` 비교
`"dog"`	`"dog"`	동일하다	동일하다
`"dog"`	`"Dog"`	동등하지 않음	동일하다
`"resume"`	`"résumé"`	동등하지 않음	동등하지 않음

또한 유니코드를 사용하면 문자열에 여러 가지 메모리 내 표현이 있을 수 있습니다. 예를 들어 e-acute(é)은 다음과 같은 두 가지 방법으로 나타낼 수 있습니다.

- 단일 리터럴 `'é'` 문자(은/는 `'\u00E9'`으로도 표기됩니다).
- 리터럴의 악센트 없는 `'e'` 문자에 이어 결합 악센트 수정자 문자 `'\u0301'`가 있습니다.

즉, 구성 요소가 다르더라도 다음 *네* 개의 문자열이 모두 표시 `"résumé"`됨을 의미합니다. 문자열은 리터럴 `'é'` 문자 또는 악센트가 없는 리터럴 `'e'` 문자에 결합 악센트 수정자 `'\u0301'`를 사

용하여 구성됩니다.

- `"r\u00E9sum\u00E9"`
- `"r\u00E9sume\u0301"`
- `"re\u0301sum\u00E9"`
- `"re\u0301sume\u0301"`

서수 비교자를 사용할 경우, 이러한 문자열 중 어느 것도 서로 같지 않습니다. 이는 모두 서로 다른 기본 문자 시퀀스를 포함하기 때문에 화면에 렌더링될 때 모두 동일하게 표시되기 때문입니다.

작업을 수행할 `string.IndexOf(..., StringComparison.Ordinal)` 때 런타임은 정확한 부분 문자열 일치를 찾습니다. 결과는 다음과 같습니다.

C#

```
Console.WriteLine("resume".IndexOf('e', StringComparison.Ordinal)); // "resume":
prints '1'
Console.WriteLine("r\u00E9sum\u00E9".IndexOf('e', StringComparison.Ordinal)); //
"résumé": prints '-1'
Console.WriteLine("r\u00E9sume\u0301".IndexOf('e', StringComparison.Ordinal)); //
"résumé": prints '5'
Console.WriteLine("re\u0301sum\u00E9".IndexOf('e', StringComparison.Ordinal)); //
"résumé": prints '1'
Console.WriteLine("re\u0301sume\u0301".IndexOf('e', StringComparison.Ordinal)); //
"résumé": prints '1'
Console.WriteLine("resume".IndexOf('e', StringComparison.OrdinalIgnoreCase)); //
"resume": prints '1'
Console.WriteLine("r\u00E9sum\u00E9".IndexOf('e',
StringComparison.OrdinalIgnoreCase)); // "résumé": prints '-1'
Console.WriteLine("r\u00E9sume\u0301".IndexOf('e',
StringComparison.OrdinalIgnoreCase)); // "résumé": prints '5'
Console.WriteLine("re\u0301sum\u00E9".IndexOf('e',
StringComparison.OrdinalIgnoreCase)); // "résumé": prints '1'
Console.WriteLine("re\u0301sume\u0301".IndexOf('e',
StringComparison.OrdinalIgnoreCase)); // "résumé": prints '1'
```

서수 검색 및 비교 루틴은 현재 스레드의 문화권 설정에 영향을 받지 않습니다.

.NET의 문자열에는 포함된 null 문자(및 기타 인쇄하지 않는 문자)가 포함될 수 있습니다. 서수 비교와 문화에 민감한 비교(고정 문화권을 사용하는 비교 포함)의 가장 명확한 차이점 중 하나는 문자열에 포함된 null 문자를 처리하는 방식에 관련이 있습니다. 이러한 문자는 문화권 구분 비교(고정 문화권을 사용하는 비교 포함)를 수행하기 위해 String.Compare 및 String.Equals 메서드를 사용할 때 무시됩니다. 따라서 포함된 null 문자가 포함된 문자열은 그렇지 않은 문자열과 같은 것으로 간주될 수 있습니다. 포함된 인쇄되지 않은 문자는 다음과 같은 String.StartsWith문자열 비교 메서드를 위해 건너뛸 수 있습니다.

> ⓘ **Important**
>
> 문자열 비교 메서드는 포함된 null 문자를 무시하지만, 문자열 검색 메서드 String.Contains, String.EndsWith, String.IndexOf, String.LastIndexOf, String.StartsWith는 무시하지 않습니다.

다음 예제에서는 "A"와 "a" 사이에 포함된 Null 문자가 여러 개 포함된 유사한 문자열을 사용하여 문자열 "Aa"의 문화권 구분 비교를 수행하고 두 문자열이 동일한 것으로 간주되는 방법을 보여 줍니다.

C#

```
string str1 = "Aa";
string str2 = "A" + new string('\u0000', 3) + "a";

Thread.CurrentThread.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("en-us");

Console.WriteLine($"Comparing '{str1}' ({ShowBytes(str1)}) and '{str2}'
({ShowBytes(str2)}):");
Console.WriteLine("   With String.Compare:");
Console.WriteLine($"      Current Culture: {string.Compare(str1, str2,
StringComparison.CurrentCulture)}");
Console.WriteLine($"      Invariant Culture: {string.Compare(str1, str2,
StringComparison.InvariantCulture)}");
Console.WriteLine("   With String.Equals:");
Console.WriteLine($"      Current Culture: {string.Equals(str1, str2,
StringComparison.CurrentCulture)}");
Console.WriteLine($"      Invariant Culture: {string.Equals(str1, str2,
StringComparison.InvariantCulture)}");

string ShowBytes(string value)
{
   string hexString = string.Empty;
   for (int index = 0; index < value.Length; index++)
   {
      string result = Convert.ToInt32(value[index]).ToString("X4");
      result = string.Concat(" ", result.Substring(0,2), " ", result.Substring(2,
2));
      hexString += result;
   }
   return hexString.Trim();
}

// The example displays the following output:
//     Comparing 'Aa' (00 41 00 61) and 'Aa' (00 41 00 00 00 00 00 00 00 61):
//        With String.Compare:
//           Current Culture: 0
//           Invariant Culture: 0
//        With String.Equals:
```

```
//           Current Culture: True
//           Invariant Culture: True
```

그러나 다음 예제와 같이 서수 비교를 사용할 때는 문자열이 같은 것으로 간주되지 않습니다.

C#

```
string str1 = "Aa";
string str2 = "A" + new String('\u0000', 3) + "a";

Console.WriteLine($"Comparing '{str1}' ({ShowBytes(str1)}) and '{str2}'
({ShowBytes(str2)}):");
Console.WriteLine("   With String.Compare:");
Console.WriteLine($"      Ordinal: {string.Compare(str1, str2,
StringComparison.Ordinal)}");
Console.WriteLine("   With String.Equals:");
Console.WriteLine($"      Ordinal: {string.Equals(str1, str2,
StringComparison.Ordinal)}");

string ShowBytes(string str)
{
    string hexString = string.Empty;
    for (int ctr = 0; ctr < str.Length; ctr++)
    {
        string result = Convert.ToInt32(str[ctr]).ToString("X4");
        result = " " + result.Substring(0, 2) + " " + result.Substring(2, 2);
        hexString += result;
    }
    return hexString.Trim();
}

// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Ordinal: 97
//       With String.Equals:
//          Ordinal: False
```

대/소문자를 구분하지 않는 서수 비교는 그다음으로 가장 신중한 접근법입니다. 이러한 비교에서는 대부분의 대/소문자를 무시하여, 예를 들어 "windows"와 "Windows"가 일치합니다. ASCII 문자를 처리할 때 이 정책은 일반적인 ASCII 대/소문자를 무시하는 것을 제외하면 StringComparison.Ordinal와 동일합니다. 따라서 [A, Z](\u0041-\u005A)의 문자는 [a,z](\u0061-\007A)의 해당 문자와 일치합니다. ASCII 범위를 벗어난 문자의 대소문자 변환은 불변 문화권 테이블을 사용합니다. 따라서 다음 비교를 수행합니다.

C#

```
string.Compare(strA, strB, StringComparison.OrdinalIgnoreCase);
```

는 이 비교와 동일하지만 더 빠릅니다.

C#

```
string.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
StringComparison.Ordinal);
```

이러한 비교는 여전히 매우 빠릅니다.

StringComparison.Ordinal 둘 다 StringComparison.OrdinalIgnoreCase 이진 값을 직접 사용하며 일치에 가장 적합합니다. 비교 설정에 대해 잘 모르는 경우 다음 두 값 중 하나를 사용합니다. 그러나 바이트 바이트 비교를 수행하기 때문에 언어 정렬 순서(예: 영어 사전)를 기준으로 정렬하지 않고 이진 정렬 순서로 정렬합니다. 사용자에게 표시되는 경우 대부분의 컨텍스트에서 결과가 이상하게 보일 수 있습니다.

서수적 의미는 String.Equals 인수를 포함하지 않는 오버로드(같음 연산자를 포함)를 위한 기본값 StringComparison입니다. 어떤 경우든 StringComparison 매개 변수를 포함하는 오버로드를 호출하는 것이 좋습니다.

언어 문자열 비교

언어 검색 및 비교 루틴은 문자열을 *데이터 정렬 요소* 로 분해하고 이러한 요소에 대한 검색 또는 비교를 수행합니다. 문자열의 문자와 문자열의 구성 데이터 정렬 요소 사이에 반드시 1:1 매핑이 있는 것은 아닙니다. 예를 들어 길이 2의 문자열은 단일 데이터 정렬 요소로만 구성됩니다. 두 문자열을 언어 인식 방식으로 비교하는 경우 비교자는 문자열의 리터럴 문자가 다르더라도 두 문자열의 데이터 정렬 요소가 동일한 의미 체계 의미를 갖는지 확인합니다.

이전 섹션에서 설명한 문자열 `"résumé"` 및 해당 네 가지 표현을 고려합니다. 다음 표에서는 각 표현을 데이터 정렬 요소로 세분화하여 보여줍니다.

테이블 확장

문자열	데이터 정렬 요소로
`"r\u00E9sum\u00E9"`	`"r" + "\u00E9" + "s" + "u" + "m" + "\u00E9"`
`"r\u00E9sume\u0301"`	`"r" + "\u00E9" + "s" + "u" + "m" + "e\u0301"`
`"re\u0301sum\u00E9"`	`"r" + "e\u0301" + "s" + "u" + "m" + "\u00E9"`
`"re\u0301sume\u0301"`	`"r" + "e\u0301" + "s" + "u" + "m" + "e\u0301"`

데이터 정렬 요소는 독자가 단일 문자 또는 문자 클러스터로 생각하는 내용에 느슨하게 해당합니다. 개념적으로 그래프 클러스터 와 유사하지만 다소 큰 우산을 포함합니다.

언어 비교자에서 정확한 일치는 필요하지 않습니다. 데이터 정렬 요소는 의미에 따라 비교됩니다. 예를 들어 언어 비교자는 부분 문자열을 `"\u00E9"` `"e\u0301"` 동일하게 처리합니다. 둘 다 의미상 "급성 악센트 한정자가 있는 소문자 e"를 의미하기 때문에 동일합니다. 이렇게 하면 다음 코드 샘플과 같이 의미상 동등한 부분 `IndexOf` 문자열을 포함하는 더 큰 문자열 내의 부분 문자열과 `"e\u0301"` 메서드를 일치시킬 수 `"\u00E9"` 있습니다.

C#

```
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("e")); // "résumé": prints '-1' (not
found)
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("\u00E9")); // "résumé": prints '1'
Console.WriteLine("\u00E9".IndexOf("e\u0301")); // prints '0'
```

따라서 언어 비교를 사용하는 경우 길이가 다른 두 문자열을 동일하게 비교할 수 있습니다. 호출자는 이러한 시나리오에서 문자열 길이를 다루는 특수한 경우 논리를 사용하지 않도록 주의해야 합니다.

문화권 인식 검색 및 비교 루틴은 언어 검색 및 비교 루틴의 특별한 형태입니다. 문화권 인식 비교자에서 데이터 정렬 요소의 개념은 지정된 문화권과 관련된 정보를 포함하도록 확장됩니다.

예를 들어 헝가리어 알파벳에서 두 문자 <dz>가 뒤로 나타나면 d< 또는 >z<와 구별되는 >고유한 문자로 간주됩니다. 즉 <, dz> 가 문자열에 표시되면 헝가리 문화권 인식 비교자가 이를 단일 데이터 정렬 요소로 처리합니다.

테이블 확장

문자열	데이터 정렬 요소로	비고
`"endz"`	`"e" + "n" + "d" + "z"`	(표준 언어 비교자 사용)
`"endz"`	`"e" + "n" + "dz"`	헝가리 문화에 맞춘 비교기 사용

헝가리 문화권 인식 비교자를 사용할 때, `"endz"`문자열이 *끝 부분*에 `"z"`부분 문자열이 없는 것으로 간주됩니다. 이것은 <dz>와 <z>가 의미상 서로 다른 정렬 요소로 인식되기 때문입니다.

C#

```
// Set thread culture to Hungarian
CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("hu-HU");
Console.WriteLine("endz".EndsWith("z")); // Prints 'False'

// Set thread culture to invariant culture
CultureInfo.CurrentCulture = CultureInfo.InvariantCulture;
Console.WriteLine("endz".EndsWith("z")); // Prints 'True'
```

> ⓘ **참고 항목**
>
> - **동작**: 언어 및 문화권 인식 비교자는 때때로 동작 조정을 받을 수 있습니다. ICU 및 이전 Windows NLS 기능은 모두 세계 언어의 변경 방식을 고려하여 업데이트됩니다. 자세한 내용은 블로그 게시물 **로캘(문화권) 데이터 변동**을 참조하세요. *Ordinal* 비교기의 동작은 정확한 비트 단위의 검색 및 비교를 수행하여 절대 변경되지 않습니다. 그러나 유니코드가 더 많은 문자 집합을 포괄하게 되고, 기존 대/소문자 데이터의 누락이 수정됨에 따라 *OrdinalIgnoreCase* 비교자의 동작이 변경될 수 있습니다.
> - **사용법**: 비교자 `StringComparison.InvariantCulture` 이며 `StringComparison.InvariantCultureIgnoreCase` 문화권을 인식하지 않는 언어 비교자입니다. 즉, 이러한 비교자는 여러 가지 가능한 기본 표현을 갖는 악센트 문자 é와 같은 개념을 이해하고 이러한 모든 표현을 동일하게 처리해야 합니다. 문화에 의존하지 않는 언어 비교자는 위에 설명된 것처럼 d 또는 z와 별개로 dz에 대한 특수 처리를 포함하지 않습니다. 또한 독일어 Eszett(ß)와 같은 특수 문자는 사용할 수 없습니다.

.NET은 *고정 세계화 모드*도 제공합니다. 이 옵트인 모드는 언어 검색 및 비교 루틴을 처리하는 코드 경로를 사용하지 않도록 설정합니다. 이 모드에서는 호출자가 제공하는 `CultureInfo` 또는 `StringComparison` 인수에 관계없이 모든 작업이 *Ordinal* 또는 *OrdinalIgnoreCase* 동작을 사용합니다. 자세한 내용은 세계화 및 .NET Core 세계화 고정 모드에대한 런타임 구성 옵션을 참조하세요.

고정 문화 설정을 사용하는 문자열 연산

정적 CompareInfo 속성에서 반환된 CultureInfo.InvariantCulture 속성을 사용하여 고정 문화권과 비교합니다. 이 동작은 모든 시스템에서 동일합니다. 범위 밖의 모든 문자를 동일한 고정 문자라고 생각되는 문자로 변환합니다. 이 정책은 문화권 간에 하나의 문자열 동작 집합을 유지하는 데 유용할 수 있지만 예기치 않은 결과를 제공하는 경우가 많습니다.

변하지 않는 문화권과 대/소문자를 구분하지 않는 비교는 비교 조건 정보에도 정적 속성에서 반환된 정적 CompareInfoCultureInfo.InvariantCulture 속성을 사용합니다. 이러한 번역된 문자 간의 대/소문자 차이는 무시됩니다.

ASCII 문자열에서 StringComparison.InvariantCulture 및 StringComparison.Ordinal를 사용하는 비교는 동일하게 작동합니다. 그러나 StringComparison.InvariantCulture 바이트 집합으로 해석해야 하는 문자열에 적합하지 않을 수 있는 언어적 결정을 내립니다.
`CultureInfo.InvariantCulture.CompareInfo` 개체는 Compare 메서드가 특정 문자 집합을 동등한 것으로 해석하도록 합니다. 예를 들어 다음 동등성은 고정 문화권에서 유효합니다.

InvariantCulture: a + ˚ = å

LATIN SMALL LETTER A 문자 "a"(\u0061), "+ " "(\u030a) 위의 결합 링 옆에 있는 경우 라틴 문자 A WITH RING ABOVE 문자 "å"(\u00e5)로 해석됩니다. 다음 예제와 같이 이 동작은 서수 비교와 다릅니다.

C#

```
string separated = "\u0061\u030a";
string combined = "\u00e5";

Console.WriteLine($"Equal sort weight of {separated} and {combined} using
InvariantCulture: {string.Compare(separated, combined,
StringComparison.InvariantCulture) == 0}");

Console.WriteLine($"Equal sort weight of {separated} and {combined} using Ordinal:
{string.Compare(separated, combined, StringComparison.Ordinal) == 0}");

// The example displays the following output:
//     Equal sort weight of a° and å using InvariantCulture: True
//     Equal sort weight of a° and å using Ordinal: False
```

파일 이름, 쿠키 또는 "å"와 같은 조합이 나타날 수 있는 다른 항목을 해석할 때 서수 비교는 여전히 가장 투명하고 적절한 동작을 제공합니다.

분산 문화권에는 비교에 유용한 속성이 거의 없습니다. 언어적으로 관련된 방식으로 비교를 수행하므로 완전한 기호적 동등성을 보장하지 못하지만 문화권에 표시하기 위한 선택은 아닙니다. 비교에 사용하는 StringComparison.InvariantCulture 몇 가지 이유 중 하나는 문화권 간 동일한 디스플레이에 대해 정렬된 데이터를 유지하는 것입니다. 예를 들어, 애플리케이션과 함께 제공되는 표시용 정렬된 식별자 목록이 포함된 큰 데이터 파일에 항목을 추가하려면, 불변 스타일 정렬을 사용하여 삽입해야 합니다.

StringComparison 멤버를 선택하는 방법

의미 문자열 컨텍스트에서 StringComparison 열거형 멤버로의 매핑을 다음 표에 설명합니다.

테이블 확장

데이터	행동	해당하는 문자열 비교 방법 System.StringComparison 가치
대소문자를 구분하는 내부 식별자입니다. XML 및 HTTP와 같은 표준의 대/소문자 구분 식별자.	바이트가 정확히 일치하는 비언어적 식별자입니다.	Ordinal

데이터	행동	해당하는 문자열 비교 방법 System.StringComparison 가치
대/소문자를 구분하는 보안 관련 설정입니다.		
대/소문자를 구분하지 않는 내부 식별자입니다. XML 및 HTTP와 같은 표준의 대/소문자를 구분하지 않는 식별자입니다. 파일 경로입니다. 레지스트리 키 및 값입니다. 환경 변수입니다. 리소스 식별자(예: 이름 처리). 대/소문자를 구분하지 않는 보안 관련 설정입니다.	대소문자가 상관없는 비언어적 식별자입니다.	OrdinalIgnoreCase
일부 지속된, 언어적으로 관련된 데이터입니다. 고정 정렬 순서가 필요한 언어 데이터를 표시합니다.	여전히 언어적으로 관련 있는 문화적으로 중립적인 데이터입니다.	InvariantCulture -또는- InvariantCultureIgnoreCase
사용자에게 표시되는 데이터입니다. 대부분의 사용자 입력.	로컬 언어적 관습이 필요한 데이터입니다.	CurrentCulture -또는- CurrentCultureIgnoreCase

보안 의미

앱이 필터링 또는 액세스 제어에 문자열 API를 사용하는 경우 서수 비교를 사용합니다. 현재 문화권에 따른 언어 비교는 플랫폼 및 로캘에 따라 달라지는 예기치 않은 결과를 생성할 수 있습니다. 다음과 같은 코드 패턴은 보안 악용에 취약할 수 있습니다.

C#

```
//
// THIS SAMPLE CODE IS INCORRECT.
// DO NOT USE IT IN PRODUCTION.
//
bool ContainsHtmlSensitiveCharacters(string input)
{
    if (input.IndexOf("<") >= 0) { return true; }
    if (input.IndexOf("&") >= 0) { return true; }
    return false;
}
```

기본적으로 `string.IndexOf(string)` 메서드는 언어 검색을 사용하므로 문자열에 리터럴 `'<'` 또는 `'&'` 문자가 포함될 수 있으며, `string.IndexOf(string)`이(가) 검색 부분 문자열을 찾지 못했음을 나타내는 `-1`을(를) 반환할 수 있습니다. 코드 분석 규칙 CA1307 및 CA1309는 이러한 호출 사이트에 플래그를 지정하고 개발자에게 잠재적인 문제가 있음을 경고합니다.

.NET의 일반적인 문자열 비교 메서드

다음 섹션에서는 문자열 비교에 가장 일반적으로 사용되는 메서드에 대해 설명합니다.

`String.Compare`

기본 해석: StringComparison.CurrentCulture.

문자열 해석의 가장 중심이 되는 작업인 이러한 메서드 호출의 모든 인스턴스를 검사하여 문자열을 현재 문화권에 따라 해석할지 또는 문화권에서 분리해야 하는지(기호적으로) 결정해야 합니다. 일반적으로 후자가 맞으며, StringComparison.Ordinal 비교를 대신 사용해야 합니다.

System.Globalization.CompareInfo 속성에 의해 반환되는 CultureInfo.CompareInfo 클래스는 플래그 열거형(Compare)을 통해 많은 수의 일치 옵션(서수, 공백 무시, 가나 형식 무시 등)을 제공하는 CompareOptions 메서드를 포함합니다.

`String.CompareTo`

기본 해석: StringComparison.CurrentCulture.

이 메서드는 현재 StringComparison 유형을 지정하는 오버로드를 제공하지 않습니다. 일반적으로 이 메서드를 권장 String.Compare(String, String, StringComparison) 되는 양식으로 변환할 수 있습니다.

IComparable 및 IComparable<T> 인터페이스를 구현하는 형식이 이 메서드를 구현합니다. 형식이 StringComparison 매개변수 옵션을 제공하지 않기 때문에 구현할 때 생성자에서 사용자가 StringComparer를 지정하도록 하는 경우가 많습니다. 다음 예제에서는 `FileName` 클래스의 생성

자가 StringComparer 매개 변수를 포함하는 클래스를 정의합니다. 이 StringComparer 개체는 `FileName.CompareTo` 메서드에서 사용됩니다.

C#

```
class FileName : IComparable
{
    private readonly StringComparer _comparer;

    public string Name { get; }

    public FileName(string name, StringComparer? comparer)
    {
        if (string.IsNullOrEmpty(name)) throw new
ArgumentNullException(nameof(name));

        Name = name;

        if (comparer != null)
            _comparer = comparer;
        else
            _comparer = StringComparer.OrdinalIgnoreCase;
    }

    public int CompareTo(object? obj)
    {
        if (obj == null) return 1;

        if (obj is not FileName)
            return _comparer.Compare(Name, obj.ToString());
        else
            return _comparer.Compare(Name, ((FileName)obj).Name);
    }
}
```

`String.Equals`

기본 해석: StringComparison.Ordinal.

이 String 클래스를 사용하면 정적 또는 인스턴스 Equals 메서드 오버로드를 호출하거나 정적 같음 연산자를 사용하여 같음을 테스트할 수 있습니다. 오버로드 및 연산자는 기본적으로 서수 비교를 사용합니다. 그러나 서수 비교를 수행하려는 경우에도 형식을 명시적으로 지정 StringComparison 하는 오버로드를 호출하는 것이 좋습니다. 이렇게 하면 특정 문자열 해석에 대한 코드를 더 쉽게 검색할 수 있습니다.

`String.ToUpper` 및 `String.ToLower`

기본 해석: StringComparison.CurrentCulture.

문자열을 대문자 또는 소문자로 강제 변환하는 것은 보통 대소문자에 상관없이 문자열을 비교하기 위한 작은 정규화 과정으로 사용되므로, String.ToUpper()와 String.ToLower() 메서드를 사용할 때 주의해야 합니다. 그렇다면 대/소문자를 구분하지 않는 비교를 사용하는 것이 좋습니다.

String.ToUpperInvariant 메서드와 String.ToLowerInvariant 메서드도 사용할 수 있습니다. ToUpperInvariant 는 대/소문자를 정규화하는 표준 방법입니다. StringComparison.OrdinalIgnoreCase를 사용하는 비교는 두 문자열 인수에 대해 ToUpperInvariant를 호출하고 그리고 StringComparison.Ordinal를 사용하여 비교를 수행하는 것으로 되어 있습니다.

오버로드는 해당 문화권을 나타내는 개체를 메서드에 전달 CultureInfo 하여 특정 문화권에서 대문자 및 소문자로 변환할 수도 있습니다.

`Char.ToUpper` 및 `Char.ToLower`

기본 해석: StringComparison.CurrentCulture.

Char.ToUpper(Char) 및 Char.ToLower(Char) 메서드는 이전 섹션에서 설명한 String.ToUpper() 및 String.ToLower() 메서드와 유사하게 작동합니다.

`String.StartsWith` 및 `String.EndsWith`

기본 해석: StringComparison.CurrentCulture (첫 번째 매개 변수가 `string`인 경우) 또는 StringComparison.Ordinal (첫 번째 매개 변수가 `char`인 경우).

이러한 메서드의 기본 오버로드가 비교를 수행하는 방식이 일치하지 않습니다. 매개 변수를 허용하는 `char` 오버로드는 정렬 순서 비교를 수행하지만, `string` 매개 변수를 허용하는 오버로드는 문화에 민감한 비교를 수행하며 인쇄되지 않는 문자를 무시할 수 있습니다.

`String.IndexOf` 및 `String.LastIndexOf`

기본 해석: StringComparison.CurrentCulture.

이러한 메서드의 기본 오버로드가 비교를 수행하는 방식에 일관성이 없습니다. 매개 변수를 포함하는 모든 String.IndexOf 및 String.LastIndexOf 메서드는 서수 비교를 수행하지만, 기본 Char 및 String.IndexOf 메서드 중 매개 변수를 포함한 String.LastIndexOf 는 문화권 구분 비교를 수행합니다.

String.IndexOf(String) 또는 String.LastIndexOf(String) 메서드를 호출하여 현재 인스턴스에서 찾을 문자열을 전달하는 경우, 형식을 명시적으로 지정하는 오버로드를 StringComparison를 호출하는 것이 좋습니다. Char 인수가 포함된 오버로드에서는 StringComparison 형식을 지정할 수 없습니다.

`String.Contains`

기본 해석: StringComparison.Ordinal.

메서드 String.IndexOf와 달리, String.Contains 메서드는 기본적으로 `char` 오버로드와 `string` 오버로드 모두에 대해 서수 비교를 사용합니다. 그러나 호출 사이트에서 동작을 명확하게 하려면 의도가 중요한 경우에도 명시적 StringComparison 인수를 전달해야 합니다.

`MemoryExtensions.AsSpan.IndexOfAny` 및 `SearchValues<T>` 유형

.NET 8에서는 SearchValues<T> 범위 내에서 특정 문자 또는 바이트 집합을 검색하기 위한 최적화된 솔루션을 제공하는 형식을 도입했습니다.

문자열을 알려진 값의 고정 집합과 반복적으로 비교하는 경우 연결된 비교나 LINQ 기반 접근 방식 대신 SearchValues<T>.Contains(T) 메서드를 사용하는 것이 좋습니다. `SearchValues<T>` 는 내부 조회 구조를 미리 계산하고 제공된 값을 기반으로 비교 논리를 최적화할 수 있습니다. 성능상의 이점을 얻으려면 인스턴스를 한 번 생성하고 `SearchValues<string>`로 캐시한 다음, 비교를 위해 다시 사용하세요.

C#

```
using System.Buffers;

namespace ExampleCode;

internal partial class DemoCode
{
    private static readonly SearchValues<string> Commands =
        SearchValues.Create(
            ["start", "run", "go", "begin", "commence"],
            StringComparison.OrdinalIgnoreCase);

    void ProcessCommand(string command)
    {
        if (Commands.Contains(command))
        {
            // ...
        }
    }
}
```

.NET 9 `SearchValues` 에서는 더 큰 문자열 내에서 부분 문자열 검색을 지원하도록 확장되었습니다. 예를 들어 확장을 참조하세요SearchValues.

간접적으로 문자열 비교를 수행하는 메서드

문자열을 중앙 연산으로 비교하는 일부 비 문자열 메서드는 이 형식을 StringComparer 사용합니다. StringComparer 클래스에는 StringComparer 메서드가 다음 형식의 문자열 비교를 수행하는 인스턴스를 반환하는 6개의 정적 속성이 포함되어 있습니다.

- 현재 문화에 민감한 문화 기반 문자열 비교. 이 StringComparer 개체는 속성에서 반환됩니다 StringComparer.CurrentCulture .
- 현재 문화를 사용하여 대소문자를 구분하지 않는 비교입니다. 이 StringComparer 개체는 속성에서 반환됩니다 StringComparer.CurrentCultureIgnoreCase .
- 비교할 문화권에 구애받지 않고 불변 문화의 단어 비교 규칙을 사용하여 비교합니다. 이 StringComparer 개체는 속성에서 반환됩니다 StringComparer.InvariantCulture .
- 고정 문화의 단어 비교 규칙을 사용하여 대소문자와 문화의 영향을 받지 않는 비교입니다. 이 StringComparer 개체는 속성에서 반환됩니다 StringComparer.InvariantCultureIgnoreCase .
- 서수 비교. 이 StringComparer 개체는 속성에서 반환됩니다 StringComparer.Ordinal .
- 대/소문자를 구분하지 않는 서수 비교입니다. 이 StringComparer 개체는 속성에서 반환됩니다 StringComparer.OrdinalIgnoreCase .

`Array.Sort` 및 `Array.BinarySearch`

기본 해석: StringComparison.CurrentCulture.

컬렉션에 데이터를 저장하거나 파일 또는 데이터베이스의 저장된 데이터를 컬렉션으로 읽는 경우에 현재 문화권을 전환하면 컬렉션의 불변 조건이 깨질 수 있습니다. 메서드는 Array.BinarySearch 검색할 배열의 요소가 이미 정렬되어 있다고 가정합니다. 배열의 문자열 요소를 정렬하기 위해 메서드는 Array.Sort 메서드를 String.Compare 호출하여 개별 요소를 정렬합니다. 문화권 구분 비교자를 사용하면 배열이 정렬된 시간과 해당 내용이 검색되는 시간 사이에 문화권이 변경될 경우 위험할 수 있습니다. 예를 들어, 다음 코드에서는 `Thread.CurrentThread.CurrentCulture` 속성이 암시적으로 제공하는 비교자에 대해 스토리지 및 검색이 수행됩니다. 문화가 `StoreNames`와 `DoesNameExist` 사이의 호출 중에 변경될 수 있으며, 특히 배열 내용이 두 메서드 호출 사이에 어딘가에 유지될 경우 이진 검색이 실패할 수 있습니다.

C#

```
// Incorrect
string[] _storedNames;

public void StoreNames(string[] names)
{
    _storedNames = new string[names.Length];

    // Copy the array contents into a new array
    Array.Copy(names, _storedNames, names.Length);

    Array.Sort(_storedNames); // Line A
```

```
}

public bool DoesNameExist(string name) =>
    Array.BinarySearch(_storedNames, name) >= 0; // Line B
```

다음 예제에서는 배열을 정렬하고 검색하는 데 동일한 서수(문화권을 구분하지 않는) 비교 메서드를 사용하는 권장 변형이 나타납니다. 변경 코드는 레이블이 지정된 `Line A` 줄과 `Line B` 두 예제에 반영됩니다.

C#

```
// Correct
string[] _storedNames;

public void StoreNames(string[] names)
{
    _storedNames = new string[names.Length];

    // Copy the array contents into a new array
    Array.Copy(names, _storedNames, names.Length);

    Array.Sort(_storedNames, StringComparer.Ordinal); // Line A
}

public bool DoesNameExist(string name) =>
    Array.BinarySearch(_storedNames, name, StringComparer.Ordinal) >= 0; // Line B
```

이 데이터가 문화권 간에 유지 및 이동되고 이 데이터를 사용자에게 표시하는 데 정렬을 사용하는 경우 더 나은 사용자 출력을 위해 언어적으로 작동하지만 문화권의 변경에 영향을 받지 않는 사용을 고려할 StringComparison.InvariantCulture수 있습니다. 다음 예제에서는 배열을 정렬하고 검색하기 위해 고정 문화권을 사용하도록 이전의 두 예제를 수정합니다.

C#

```
// Correct
string[] _storedNames;

public void StoreNames(string[] names)
{
    _storedNames = new string[names.Length];

    // Copy the array contents into a new array
    Array.Copy(names, _storedNames, names.Length);

    Array.Sort(_storedNames, StringComparer.InvariantCulture); // Line A
}

public bool DoesNameExist(string name) =>
```

```
    Array.BinarySearch(_storedNames, name, StringComparer.InvariantCulture) >= 0;
// Line B
```

컬렉션 예제: `Hashtable` 생성자

해시 문자열은 문자열을 비교하는 방법에 의해 영향을 받는 작업의 두 번째 예제를 제공합니다.

다음은 Hashtable 속성에서 반환된 StringComparer 개체를 전달하여 StringComparer.OrdinalIgnoreCase 개체를 인스턴스화하는 예제입니다. 파생된 StringComparer 클래스 StringComparer 가 인터페이스를 IEqualityComparer 구현하기 때문에 해당 GetHashCode 메서드는 해시 테이블의 문자열 해시 코드를 계산하는 데 사용됩니다.

C#

```
using System.IO;
using System.Collections;

const int InitialCapacity = 100;

Hashtable creationTimeByFile = new(InitialCapacity,
StringComparer.OrdinalIgnoreCase);
string directoryToProcess = Directory.GetCurrentDirectory();

// Fill the hash table
PopulateFileTable(directoryToProcess);

// Get some of the files and try to find them with upper cased names
foreach (var file in Directory.GetFiles(directoryToProcess))
    PrintCreationTime(file.ToUpper());


void PopulateFileTable(string directory)
{
    foreach (string file in Directory.GetFiles(directory))
        creationTimeByFile.Add(file, File.GetCreationTime(file));
}

void PrintCreationTime(string targetFile)
{
    object? dt = creationTimeByFile[targetFile];

    if (dt is DateTime value)
        Console.WriteLine($"File {targetFile} was created at time {value}.");
    else
        Console.WriteLine($"File {targetFile} does not exist.");
}
```

컬렉션 예제: `SortedSet<T>` 및 `List<T>.Sort`

정렬된 문자열 컬렉션을 인스턴스화하거나 기존 문자열 기반 컬렉션을 정렬할 때 동일한 로캘 민감도 문제가 적용됩니다. 항상 명시적 비교자를 지정합니다.

C#

```
// Words to sort
string[] values = [ "able", "ångström", "apple", "Æble",
            "Windows", "Visual Studio" ];

//
// Potentially incorrect code - behavior might vary based on locale.
//
SortedSet<string> mySet = [.. values]; // No comparer specified

List<string> list = [.. values];
list.Sort(); // No comparer specified

//
// Corrected code - uses ordinal sorting; doesn't vary by locale.
//
SortedSet<string> mySet2 = new(values, StringComparer.Ordinal);

List<string> list2 = [.. values];
list2.Sort(StringComparer.Ordinal);
```

.NET과 .NET Framework의 차이점

.NET 및 .NET Framework는 세계화를 다르게 처리합니다. Windows의 .NET Framework는 언어 문자열 비교를 위해 운영 체제의 NLS(국가 언어 지원) 기능을 사용합니다. .NET은 지원되는 모든 플랫폼에서 언어 문자열 비교를 위해 ICU(International Components for Unicode) 라이브러리를 사용합니다.

ICU와 NLS는 언어 비교자에서 서로 다른 논리를 구현하기 때문에 문화권 구분 비교를 사용하는 문자열 메서드의 결과는 .NET과 .NET Framework 간에 다를 수 있습니다. 이는 다음을 포함하여 기본적으로 언어 비교자를 사용하는 모든 메서드에 중요합니다.

- String.Compare
- String.EndsWith (첫 번째 매개 변수가 인 `string`경우)
- String.IndexOf (첫 번째 매개 변수가 인 `string`경우)
- String.StartsWith (첫 번째 매개 변수가 인 `string`경우)
- String.ToLower
- String.ToLowerInvariant
- String.ToUpper
- String.ToUpperInvariant
- System.Globalization.TextInfo (대부분의 멤버)

- System.Globalization.CompareInfo (대부분의 멤버)
- Array.Sort (문자열 배열을 정렬할 때)
- List<T>.Sort() (목록 요소가 문자열인 경우)
- System.Collections.Generic.SortedDictionary<TKey,TValue> (키가 문자열인 경우)
- System.Collections.Generic.SortedList<TKey,TValue> (키가 문자열인 경우)
- System.Collections.Generic.SortedSet<T> (집합에 문자열이 포함된 경우)

> ⓘ **참고 항목**
>
> 영향을 받는 API의 전체 목록은 아닙니다.

한 가지 주목할 만한 차이점은 포함된 null 및 기타 컨트롤 문자의 처리입니다. NLS에서 언어 비교자를 사용하는 경우 null 문자()와 같은 일부 컨트롤 문자는 `\0` 특정 비교 컨텍스트에서 무시할 수 있는 것으로 처리될 수 있습니다. ICU에서 이러한 문자는 문자열의 실제 문자로 처리됩니다. 이렇게 하면 `string.IndexOf(string)` 검색 문자열에 null 문자가 포함된 경우 다른 결과가 반환될 수 있습니다.

예를 들어 다음 코드는 현재 런타임에 따라 다른 답변을 생성할 수 있습니다.

C#

```
const string greeting = "Hel\0lo";
Console.WriteLine($"{greeting.IndexOf("\0")}");

// The snippet prints:
//
// '3' when running on .NET Framework and .NET Core 2.x - 3.x (Windows)
// '0' when running on .NET 5 or later (Windows)
// '0' when running on .NET Core 2.x - 3.x or .NET 5 (non-Windows)
// '3' when running on .NET Core 2.x or .NET 5+ (in invariant mode)
```

이러한 플랫폼 간 및 구현 간 놀라움을 방지하는 가장 좋은 방법은 항상 명시적 StringComparison 인수를 문자열 비교 메서드에 전달하고 비언어적 비교를 사용 StringComparison.Ordinal 하거나 StringComparison.OrdinalIgnoreCase 사용하는 것입니다.

.NET Framework에서 .NET으로 애플리케이션을 마이그레이션하고 Windows에서 레거시 NLS 동작을 사용하는 경우 NLS를 사용하도록 애플리케이션을 구성할 수 있습니다. 자세한 내용은 .NET 세계화 및 ICU를 참조하세요.

참고하십시오

- .NET 앱의 세계화
- .NET 세계화 및 ICU

- C에서 문자열을 비교하는 방법#

Last updated on 2026. 02. 26.

형식이 지정된 데이터 표시 및 유지에 대한 모범 사례

아티클 • 2025. 03. 29.

이 문서에서는 숫자 데이터 및 날짜 및 시간 데이터와 같은 형식이 지정된 데이터를 표시 및 스토리지에 대해 처리하는 방법을 살펴봅니다.

.NET을 사용하여 개발하는 경우 문화권 구분 서식을 사용하여 숫자 및 날짜와 같은 문자열이 아닌 데이터를 사용자 인터페이스에 표시합니다. 고정 문화권 서식을 사용하여 문자열이 아닌 데이터를 문자열 형식으로 유지합니다. 문화권 구분 서식을 사용하여 숫자 또는 날짜 및 시간 데이터를 문자열 형식으로 유지하지 마세요.

서식이 지정된 데이터 표시

숫자, 날짜 및 시간과 같은 문자열이 아닌 데이터를 사용자에게 표시하는 경우 사용자의 문화권 설정을 사용하여 서식을 지정합니다. 기본적으로 다음 항목은 모두 서식 지정 작업에서 현재 문화권을 사용합니다.

- C# 및 Visual Basic 컴파일러에서 지원하는 보간된 문자열입니다.
- C# 연결 연산자 또는 Visual Basic 연결 연산자를 사용하거나 String.Concat 메서드를 직접 호출하는 문자열 연결 작업입니다.
- String.Format 메서드입니다.
- 숫자 형식과 날짜 및 시간 형식의 `ToString` 메서드입니다.

지정된 문화권의 규칙이나 고정 문화권사용하여 문자열의 서식을 명시적으로 지정하려면 다음을 수행할 수 있습니다.

- String.Format 및 `ToString` 메서드를 사용하는 경우 String.Format(IFormatProvider, String, Object[]) 또는 DateTime.ToString(IFormatProvider)같은 `provider` 매개 변수가 있는 오버로드를 호출하고 CultureInfo.CurrentCulture 속성, 원하는 문화권을 나타내는 CultureInfo 인스턴스 또는 CultureInfo.InvariantCulture 속성을 전달합니다.

- 문자열 연결의 경우 컴파일러가 암시적 변환을 수행하도록 허용하지 않습니다. 대신 `provider` 매개 변수가 있는 `ToString` 오버로드를 호출하여 명시적 변환을 수행합니다. 예를 들어 컴파일러는 다음 코드에서 Double 값을 문자열로 변환할 때 현재 문화권을 암시적으로 사용합니다.

  C#

  ```
  string concat1 = "The amount is " + 126.03 + ".";
  ```

```
Console.WriteLine(concat1);
```

대신 다음 코드와 같이 Double.ToString(IFormatProvider) 메서드를 호출하여 변환에 형식 지정 규칙이 사용되는 문화권을 명시적으로 지정할 수 있습니다.

```
C#

string concat2 = "The amount is " +
126.03.ToString(CultureInfo.InvariantCulture) + ".";
Console.WriteLine(concat2);
```

- 문자열 보간을 위해 보간된 문자열을 String 인스턴스 대신 FormattableString에 할당하세요. 그런 다음 FormattableString.ToString() 메서드를 호출하여 현재 문화권의 규칙을 반영하는 결과 문자열을 생성하거나 FormattableString.ToString(IFormatProvider) 메서드를 호출하여 지정된 문화권의 규칙을 반영하는 결과 문자열을 생성할 수 있습니다.

  고정 FormattableString.Invariant 메서드에 서식이 지정된 문자열을 전달하여 고정 문화권의 규칙을 반영하는 결과 문자열을 생성할 수도 있습니다. 다음 예제에서는 이 방법을 보여 줍니다. (예제의 출력은 현재의 `en-US` 문화를 반영합니다.)

  ```
  C#

  using System;
  using System.Globalization;

  class Program
  {
      static void Main()
      {
          Decimal value = 126.03m;
          FormattableString amount = $"The amount is {value:C}";
          Console.WriteLine(amount.ToString());
          Console.WriteLine(amount.ToString(new CultureInfo("fr-FR")));
          Console.WriteLine(FormattableString.Invariant(amount));
      }
  }
  // The example displays the following output:
  //    The amount is $126.03
  //    The amount is 126,03 €
  //    The amount is ¤126.03
  ```

  > ⓘ **참고**
  >
  > C#을 사용하고 고정 문화권을 사용하여 서식을 지정하는 경우 String.Create(IFormatProvider, DefaultInterpolatedStringHandler) 호출하고

첫 번째 매개 변수에 대한 CultureInfo.InvariantCulture 전달하는 것이 더 성능이 높습니다. 자세한 내용은 C# 10 및 .NET 6의 문자열 보간 을 참조하세요.

서식이 지정된 데이터 유지

문자열이 아닌 데이터를 이진 데이터 또는 형식이 지정된 데이터로 유지할 수 있습니다. 서식이 지정된 데이터로 저장하도록 선택한 경우 `provider` 매개 변수를 포함하는 서식 지정 메서드 오버로드를 호출하고 CultureInfo.InvariantCulture 속성을 전달해야 합니다. 불변 문화권은 문화권 및 컴퓨터와 독립적인 서식 있는 데이터에 일관된 형식을 제공합니다. 반면 고정 문화권 이외의 문화권을 사용하여 형식이 지정된 데이터를 유지하면 다음과 같은 여러 제한 사항이 있습니다.

- 다른 문화권이 있는 시스템에서 데이터를 검색하거나 현재 시스템의 사용자가 현재 문화권을 변경하고 데이터를 검색하려고 하면 데이터를 사용할 수 없을 수 있습니다.
- 특정 컴퓨터의 문화권 속성은 표준 값과 다를 수 있습니다. 사용자는 언제든지 문화권에 민감한 디스플레이 설정을 사용자 지정할 수 있습니다. 이 때문에 사용자가 문화권 설정을 사용자 지정한 후에는 시스템에 저장된 서식이 지정된 데이터를 읽을 수 없습니다. 컴퓨터에서 서식이 지정된 데이터의 이식성은 훨씬 더 제한될 수 있습니다.
- 시간 경과에 따라 숫자 또는 날짜 및 시간의 서식을 제어하는 국제, 지역 또는 국가 표준이 변경되며 이러한 변경 내용은 Windows 운영 체제 업데이트에 통합됩니다. 서식 지정 규칙이 변경되면 이전 규칙을 사용하여 서식이 지정된 데이터를 읽을 수 없게 될 수 있습니다.

다음 예제에서는 문화권 구분 서식을 사용하여 데이터를 유지하는 데 발생하는 제한된 이식성을 보여 줍니다. 이 예제에서는 날짜 및 시간 값의 배열을 파일에 저장합니다. 이러한 형식은 영어(미국) 문화권의 규칙을 사용하여 형식이 지정됩니다. 애플리케이션이 현재 문화권을 프랑스어(스위스)로 변경한 후 현재 문화권의 서식 규칙을 사용하여 저장된 값을 읽으려고 합니다. 두 개의 데이터 항목을 읽으려고 시도하면 FormatException 예외가 발생하며 날짜 배열에는 이제 MinValue동일한 두 개의 잘못된 요소가 포함됩니다.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Text;
using System.Threading;

public class Example
{
   private static string filename = @".\dates.dat";
```

```
   public static void Main()
   {
      DateTime[] dates = { new DateTime(1758, 5, 6, 21, 26, 0),
                           new DateTime(1818, 5, 5, 7, 19, 0),
                           new DateTime(1870, 4, 22, 23, 54, 0),
                           new DateTime(1890, 9, 8, 6, 47, 0),
                           new DateTime(1905, 2, 18, 15, 12, 0) };
      // Write the data to a file using the current culture.
      WriteData(dates);
      // Change the current culture.
      Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-CH");
      // Read the data using the current culture.
      DateTime[] newDates = ReadData();
      foreach (var newDate in newDates)
         Console.WriteLine(newDate.ToString("g"));
   }

   private static void WriteData(DateTime[] dates)
   {
      StreamWriter sw = new StreamWriter(filename, false, Encoding.UTF8);
      for (int ctr = 0; ctr < dates.Length; ctr++) {
         sw.Write("{0}", dates[ctr].ToString("g",
CultureInfo.CurrentCulture));
         if (ctr < dates.Length - 1) sw.Write("|");
      }
      sw.Close();
   }

   private static DateTime[] ReadData()
   {
      bool exceptionOccurred = false;

      // Read file contents as a single string, then split it.
      StreamReader sr = new StreamReader(filename, Encoding.UTF8);
      string output = sr.ReadToEnd();
      sr.Close();

      string[] values = output.Split( new char[] { '|' } );
      DateTime[] newDates = new DateTime[values.Length];
      for (int ctr = 0; ctr < values.Length; ctr++) {
         try {
            newDates[ctr] = DateTime.Parse(values[ctr],
CultureInfo.CurrentCulture);
         }
         catch (FormatException) {
            Console.WriteLine($"Failed to parse {values[ctr]}");
            exceptionOccurred = true;
         }
      }
      if (exceptionOccurred) Console.WriteLine();
      return newDates;
   }
}
```

```
// The example displays the following output:
//       Failed to parse 4/22/1870 11:54 PM
//       Failed to parse 2/18/1905 3:12 PM
//
//       05.06.1758 21:26
//       05.05.1818 07:19
//       01.01.0001 00:00
//       09.08.1890 06:47
//       01.01.0001 00:00
//       01.01.0001 00:00
```

그러나 DateTime.ToString(String, IFormatProvider) 및 DateTime.Parse(String, IFormatProvider)호출에서 CultureInfo.CurrentCulture 속성을 CultureInfo.InvariantCulture 바꾸면 다음 출력과 같이 지속형 날짜 및 시간 데이터가 성공적으로 복원됩니다.

콘솔

```
06.05.1758 21:26
05.05.1818 07:19
22.04.1870 23:54
08.09.1890 06:47
18.02.1905 15:12
```

NET에서 새 문자열 만들기

.NET을 사용하면 간단한 할당을 사용하여 문자열을 만들 수 있으며 다양한 매개 변수를 사용하여 문자열 생성을 지원하기 위해 클래스 생성자를 오버로드합니다. .NET은 또한 여러 문자열, 문자열 배열 또는 개체를 결합하여 새 문자열 개체를 만드는 클래스의 여러 메서드 System.String 를 제공합니다.

할당을 사용하여 문자열 만들기

새 String 개체를 만드는 가장 쉬운 방법은 단순히 문자열 리터럴을 개체에 할당하는 것입니다 String .

클래스 생성자를 사용하여 문자열 만들기Create strings using a class constructor

클래스 생성자의 오버로드를 String 사용하여 문자 배열에서 문자열을 만들 수 있습니다. 특정 문자를 지정된 횟수만큼 복제하여 새 문자열을 만들 수도 있습니다.
String(ReadOnlySpan<Char>) 생성자 오버로드는 ReadOnlySpan<T> 또는 스택에 할당된 문자 Span<T>를 받아들여 알려진 크기의 작은 문자열을 빌드할 때 관리되는 힙에 중간 문자 배열을 할당하지 않도록 합니다. 그러나 결과 문자열 인스턴스는 여전히 관리되는 힙에 할당됩니다.

문자열을 반환하는 메서드

다음 표에서는 새 문자열 개체를 반환하는 몇 가지 유용한 메서드를 나열합니다.

테이블 확장

메서드 이름	사용하세요
String.Format	입력 개체 집합에서 형식이 지정된 문자열을 작성합니다.
String.Concat	둘 이상의 문자열에서 문자열을 빌드합니다.
String.Join	문자열 배열을 결합하여 새 문자열을 빌드합니다.
String.Insert	기존 문자열의 지정된 인덱스에 문자열을 삽입하여 새 문자열을 작성합니다.
String.CopyTo	문자열의 지정된 문자를 문자 배열의 지정된 위치에 복사합니다.
String.Create	쓰기 가능 Span<T> 및 호출자 제공 상태 개체를 수신하는 콜백을 통해 문자를 채우는 지정된 길이의 새 문자열을 만듭니다.

`String.Format`

이 `String.Format` 메서드를 사용하여 형식이 지정된 문자열을 만들고 여러 개체를 나타내는 문자열을 연결할 수 있습니다. 이 메서드는 전달된 개체를 문자열로 자동으로 변환합니다. 예를 들어, 응용 프로그램에서 사용자에게 `Int32` 값과 `DateTime` 값을 표시해야 하는 경우, `Format` 메서드를 사용하여 이러한 값을 나타내는 문자열을 쉽게 생성할 수 있습니다. 이 메서드와 함께 사용되는 서식 지정 규칙에 대한 자세한 내용은 복합 서식 지정 섹션을 참조하세요.

다음 예제에서는 메서드를 `Format` 사용하여 정수 변수를 사용하는 문자열을 만듭니다.

C#

```
int numberOfFleas = 12;
string miscInfo = String.Format("Your dog has {0} fleas. " +
                                "It is time to get a flea collar. " +
                                "The current universal date is: {1:u}.",
                                numberOfFleas, DateTime.Now);
Console.WriteLine(miscInfo);
// The example displays the following output:
//       Your dog has 12 fleas. It is time to get a flea collar.
//       The current universal date is: 2008-03-28 13:31:40Z.
```

이 예제 DateTime.Now 에서는 현재 스레드와 연결된 문화권에 지정된 방식으로 현재 날짜 및 시간을 표시합니다.

`String.Concat`

이 `String.Concat` 방법을 사용하여 두 개 이상의 기존 개체에서 새 문자열 개체를 쉽게 만들 수 있습니다. 문자열을 연결할 수 있는 언어 독립적 방법을 제공합니다. 이 메서드는 에서 `System.Object`파생되는 모든 클래스를 허용합니다. 다음 예제에서는 두 개의 기존 문자열 개체와 구분 문자에서 문자열을 만듭니다.

C#

```
string helloString1 = "Hello";
string helloString2 = "World!";
Console.WriteLine(String.Concat(helloString1, ' ', helloString2));
// The example displays the following output:
//      Hello World!
```

`String.Join`

이 `String.Join` 메서드는 문자열 배열과 구분 기호 문자열에서 새 문자열을 만듭니다. 이 메서드는 여러 문자열을 함께 연결하여 목록을 쉼표로 구분하려는 경우에 유용합니다.

다음 예제에서는 공백을 사용하여 문자열 배열을 바인딩합니다.

C#

```
string[] words = {"Hello", "and", "welcome", "to", "my" , "world!"};
Console.WriteLine(String.Join(" ", words));
// The example displays the following output:
//       Hello and welcome to my world!
```

String.Insert

이 `String.Insert` 메서드는 다른 문자열의 지정된 위치에 문자열을 삽입하여 새 문자열을 만듭니다. 이 메서드는 0부터 시작하는 인덱스입니다. 다음은 다섯 번째 인덱스 위치에 `MyString` 문자열을 삽입하고 이 값을 사용하여 새 문자열을 만드는 예제입니다.

C#

```
string sentence = "Once a time.";
 Console.WriteLine(sentence.Insert(4, " upon"));
 // The example displays the following output:
 //       Once upon a time.
```

String.CopyTo

이 `String.CopyTo` 메서드는 문자열의 일부를 문자 배열로 복사합니다. 문자열의 시작 인덱스와 복사할 문자 수를 모두 지정할 수 있습니다. 이 메서드는 원본 인덱스, 문자 배열, 대상 인덱스 및 복사할 문자 수를 사용합니다. 모든 인덱스는 0부터 시작하는 것입니다.

다음 예제에서는 이 `CopyTo` 메서드를 사용하여 문자열 개체에서 문자 배열의 첫 번째 인덱스 위치로 "Hello"라는 단어의 문자를 복사합니다.

C#

```
string greeting = "Hello World!";
char[] charArray = {'W','h','e','r','e'};
Console.WriteLine($"The original character array: {new string(charArray)}");
greeting.CopyTo(0, charArray,0 ,5);
Console.WriteLine($"The new character array: {new string(charArray)}");
// The example displays the following output:
//       The original character array: Where
//       The new character array: Hello
```

String.Create

이 String.Create 메서드를 사용하면 콜백을 사용하여 새 문자열의 문자를 프로그래밍 방식으로 채울 수 있습니다. 콜백은 쓰기 가능한 Span<T> 문자 스트림과 호출자 제공 상태 개체를 수신하므로, 중간 문자 버퍼를 할당하지 않고 문자열의 내용을 작성할 수 있습니다. 콜백 자체는 로컬 변수를 캡처하거나 할당이 많은 다른 API를 호출하는 경우와 같이 여전히 할당될 수 있습니다.

다음 예제에서는 연속된 알파벳 문자에서 5자 문자열을 작성하는 데 사용합니다 `String.Create`

.

C#

```
string result = string.Create(5, 'a', (span, firstChar) =>
{
    for (int i = 0; i < span.Length; i++)
    {
        span[i] = (char)(firstChar + i);
    }
});

Console.WriteLine(result); // abcde
```

`String.Create` 는 최종 문자열 길이를 미리 알고 중간 문자 버퍼를 할당하지 않으려는 성능에 민감한 시나리오를 위해 설계되었습니다. 런타임은 새 문자열을 할당하고, 해당 백업 버퍼를 콜백에 `Span<char>` 직접 전달하고, 콜백이 반환되면 변경할 수 없는 문자열을 반환합니다. 콜백이 완료된 후에는 데이터 복사본이 발생하지 않습니다.

`String.Create` 대 `new String(Span<char>)`

문자열을 효율적으로 빌드하는 또 다른 옵션은 문자 버퍼 `stackalloc` 를 할당하고, 채우고, 생성자에 전달하는 것입니다 `String(ReadOnlySpan<char>)` .

C#

```
static string CreateStringFromSpan()
{
    Span<char> span = stackalloc char[5];
    for (int i = 0; i < 5; i++)
    {
        span[i] = (char)('a' + i);
    }
    return new string(span);
}

Console.WriteLine(CreateStringFromSpan()); // abcde
```

두 방법 모두 최종 문자열을 정확히 한 번 할당합니다. 주요 차이점은 다음과 같습니다.

- **`stackalloc` + `new string(span)`** 는 작업 버퍼를 스택에 배치합니다. 이는 *작은 고정 크기* 버퍼의 경우 가장 빠르지만 스택은 한정된 리소스입니다. 큰 및 깊이 중첩된 할당으로 인해 `StackOverflowException` 문제를 발생시킬 수 있습니다. 이 예제에서는 C# `stackalloc` 패턴을 보여 줍니다. Visual Basic은 `stackalloc`을 지원하지 않지만, `ReadOnlySpan<char>`가 있을 때도 `String(ReadOnlySpan<char>)` 생성자를 호출할 수 있습니다.
- **`String.Create`** 는 문자열 개체 자체의 일부로 힙에 작업 버퍼를 할당하므로 스택 압력이 없습니다. 또한 상태가 참조 형식이거나 캡처되지 않은 구조체인 경우 boxing 할당을 방지하기 위해 런타임이 boxing 없이 콜백에 전달하는 형식 지정 상태 매개 변수를 수용합니다. 일반적으로 알려진 제한된 크기의 작은 문자열(일반적으로 수백 자 미만)을 선호 `stackalloc` + `new String(span)` 합니다. `String.Create`는 크기가 큰 경우, 스택 부담을 피하려는 경우, 아니면 boxing 없이 콜백에 상태를 전달하려는 경우에 사용하세요.

참고하십시오

- 복합 형식 지정

Last updated on 2026. 03. 31.

.NET의 문자열에서 문자 자르기 및 제거

아티클 • 2025. 05. 04.

문장을 개별 단어로 구문 분석하는 경우 단어의 양쪽 끝에 공백(공백이라고도 함)이 있는 단어로 끝날 수 있습니다. 이 경우 클래스의 트리밍 메서드 `System.String` 중 하나를 사용하여 문자열의 지정된 위치에서 임의의 수의 공백 또는 다른 문자를 제거할 수 있습니다. 다음 표에서는 사용 가능한 트리밍 메서드에 대해 설명합니다.

테이블 확장

메서드 이름	사용하세요
String.Trim	문자열의 시작과 끝에서 문자 배열에 지정된 공백 또는 문자를 제거합니다.
String.TrimEnd	문자열의 끝에서 문자 배열에 지정된 문자를 제거합니다.
String.TrimStart	문자열의 시작 부분에서 문자 배열에 지정된 문자를 제거합니다.
String.Remove	문자열의 지정된 인덱스 위치에서 지정된 수의 문자를 제거합니다.

다듬다

다음 예제와 같이 메서드를 사용하여 String.Trim 문자열의 양쪽 끝에서 공백을 쉽게 제거할 수 있습니다.

C#

```
string MyString = " Big   ";
Console.WriteLine($"Hello{MyString}World!");
string TrimString = MyString.Trim();
Console.WriteLine($"Hello{TrimString}World!");
//       The example displays the following output:
//             Hello Big   World!
//             HelloBigWorld!
```

문자열의 시작과 끝에서 문자 배열에 지정한 문자를 제거할 수도 있습니다. 다음 예제에서는 공백 문자, 마침표 및 별표가 제거됩니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
```

```
      String header = "* A Short String. *";
      Console.WriteLine(header);
      Console.WriteLine(header.Trim( new Char[] { ' ', '*', '.' } ));
   }
}
// The example displays the following output:
//       * A Short String. *
//       A Short String
```

TrimEnd (끝 공백 제거)

메서드는 `String.TrimEnd` 문자열의 끝에서 문자를 제거하여 새 문자열 개체를 만듭니다. 제거할 문자를 지정하기 위해 문자 배열이 이 메서드에 전달됩니다. 문자 배열의 요소 순서는 트리밍 작업에 영향을 주지 않습니다. 배열에 지정되지 않은 문자를 찾은 경우 트리밍이 중지됩니다.

다음 예제에서는 메서드를 사용하여 문자열의 마지막 문자를 제거합니다 `TrimEnd` . 이 예제에서는 배열의 `'r'` 문자 순서가 중요하지 않음을 설명하기 위해 문자와 `'W'` 문자의 위치가 반전됩니다. 이 코드는 `MyString`의 마지막 단어와 첫 번째 단어의 일부를 제거합니다.

C#

```
string MyString = "Hello World!";
char[] MyChar = {'r','o','W','l','d','!',' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

이 코드는 `He` 콘솔에 표시됩니다.

다음 예제에서는 메서드를 사용하여 문자열의 마지막 단어를 제거합니다 `TrimEnd` . 이 코드에서 쉼표는 단어를 `Hello` 따르고 쉼표는 트리밍할 문자 배열에 지정되지 않으므로 트리밍은 쉼표로 끝납니다.

C#

```
string MyString = "Hello, World!";
char[] MyChar = {'r','o','W','l','d','!',' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

이 코드는 `Hello,` 콘솔에 표시됩니다.

TrimStart

메서드 `String.TrimStart` 는 기존 문자열 개체의 `String.TrimEnd` 시작 부분에서 문자를 제거 하여 새 문자열을 만드는 것을 제외 하 고 메서드와 비슷합니다. 제거할 문자를 지정하기 위해 `TrimStart` 문자 배열이 메서드에 전달됩니다. 메서드와 `TrimEnd` 마찬가지로 문자 배열의 요소 순서는 트리밍 작업에 영향을 주지 않습니다. 배열에 지정되지 않은 문자를 찾은 경우 트리밍이 중지됩니다.

다음 예제에서는 문자열의 첫 번째 단어를 제거합니다. 이 예제에서는 배열의 `'l'` 문자 순서가 중요하지 않음을 설명하기 위해 문자와 `'H'` 문자의 위치가 반전됩니다.

C#

```
string MyString = "Hello World!";
char[] MyChar = {'e', 'H','l','o',' ' };
string NewString = MyString.TrimStart(MyChar);
Console.WriteLine(NewString);
```

이 코드는 `World!` 콘솔에 표시됩니다.

제거하십시오

이 메서드는 String.Remove 기존 문자열의 지정된 위치에서 시작하는 지정된 수의 문자를 제거합니다. 이 메서드는 0부터 시작하는 인덱스를 가정합니다.

다음 예제에서는 문자열의 0부터 시작하는 인덱스 위치 5부터 시작하여 문자열에서 10자를 제거합니다.

C#

```
string MyString = "Hello Beautiful World!";
Console.WriteLine(MyString.Remove(5,10));
// The example displays the following output:
//         Hello World!
```

교체합니다

메서드를 호출 String.Replace(String, String) 하고 빈 문자열(String.Empty)을 대체로 지정하여 문자열에서 지정된 문자 또는 부분 문자열을 제거할 수도 있습니다. 다음 예제에서는 문자열에서 모든 쉼표가 제거됩니다.

C#

```
using System;
```

```
public class Example
{
   public static void Main()
   {
      String phrase = "a cold, dark night";
      Console.WriteLine($"Before: {phrase}");
      phrase = phrase.Replace(",", "");
      Console.WriteLine($"After: {phrase}");
   }
}
// The example displays the following output:
//       Before: a cold, dark night
//       After: a cold dark night
```

참고하십시오

- 기본 문자열 작업

.NET에서 문자열 채우기

다음 String 메서드 중 하나를 사용하여 지정된 총 길이로 선행 또는 후행 문자로 채워지는 원래 문자열로 구성된 새 문자열을 만듭니다. 안쪽 여백 문자는 공백 또는 지정된 문자일 수 있습니다. 결과 문자열은 오른쪽 맞춤 또는 왼쪽 맞춤으로 나타납니다. 원래 문자열의 길이가 이미 원하는 총 길이보다 크거나 같은 경우, 채우기 메서드는 원래 문자열을 변경하지 않고 반환합니다. 자세한 내용은 와 String.PadLeft 메서드의 두 오버로드에 대한 String.PadRight 섹션을 참조하세요.

테이블 확장

메서드 이름	사용하세요
String.PadLeft	문자열 앞에 선행 문자를 추가하여 지정된 총 길이로 맞춥니다.
String.PadRight	후행 문자가 있는 문자열을 지정된 총 길이로 패딩합니다.

PadLeft (왼쪽으로 공간 채우기)

이 메서드는 String.PadLeft 지정된 총 길이를 달성하기 위해 충분한 선행 패드 문자를 원래 문자열에 연결하여 새 문자열을 만듭니다. 메서드 String.PadLeft(Int32)는 공백을 채움 문자로 사용하며, 메서드 String.PadLeft(Int32, Char)를 사용하면 고유한 채움 문자를 지정할 수 있습니다.

다음 코드 예제에서는 메서드를 PadLeft 사용하여 20자 길이의 새 문자열을 만듭니다. 이 예제에서는 콘솔에 "`--------Hello World!`"를 표시합니다.

C#

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadLeft(20, '-'));
```

패드 오른쪽

이 메서드는 String.PadRight 지정된 총 길이를 얻기 위해 충분한 후행 패드 문자를 원래 문자열에 연결하여 새 문자열을 만듭니다. 메서드 String.PadRight(Int32)는 공백을 채움 문자로 사용하며, 메서드 String.PadRight(Int32, Char)를 사용하면 고유한 채움 문자를 지정할 수 있습니다.

다음 코드 예제에서는 메서드를 PadRight 사용하여 20자 길이의 새 문자열을 만듭니다. 이 예제에서는 콘솔에 "`Hello World!--------`"를 표시합니다.

C#

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadRight(20, '-'));
```

Last updated on 2025. 11. 13.

.NET에서 문자열 비교

아티클 • 2025. 05. 08.

.NET은 문자열 값을 비교하는 몇 가지 메서드를 제공합니다. 다음 표에서는 값 비교 메서드를 나열하고 설명합니다.

테이블 확장

메서드 이름	사용하세요
String.Compare	두 문자열의 값을 비교합니다. 정수 값을 반환합니다.
String.CompareOrdinal	로컬 문화권과 관계없이 두 문자열을 비교합니다. 정수 값을 반환합니다.
String.CompareTo	현재 문자열 개체를 다른 문자열과 비교합니다. 정수 값을 반환합니다.
String.StartsWith	문자열이 전달된 문자열로 시작하는지 여부를 결정합니다. 부울 값을 반환합니다.
String.EndsWith	문자열이 전달된 문자열로 끝나는지 여부를 결정합니다. 부울 값을 반환합니다.
String.Contains	다른 문자열 내에서 문자 또는 문자열이 발생하는지 여부를 결정합니다. 부울 값을 반환합니다.
String.Equals	두 문자열이 같은지 여부를 확인합니다. 부울 값을 반환합니다.
String.IndexOf	검사할 문자열의 시작부터 시작하여 문자 또는 문자열의 인덱스 위치를 반환합니다. 정수 값을 반환합니다.
String.LastIndexOf	검사할 문자열의 끝에서 시작하여 문자 또는 문자열의 인덱스 위치를 반환합니다. 정수 값을 반환합니다.

`Compare` 메서드

정적 String.Compare 메서드는 두 문자열을 비교하는 철저한 방법을 제공합니다. 이 메서드는 문화적으로 인식됩니다. 이 함수를 사용하여 두 문자열 또는 두 문자열의 부분 문자열을 비교할 수 있습니다. 또한 대/소문자와 문화적 차이를 고려하거나 무시하는 오버로드가 제공됩니다. 다음 표에서는 이 메서드가 반환할 수 있는 세 가지 정수 값을 보여 있습니다.

테이블 확장

반환 값	조건
음수 정수	첫 번째 문자열은 정렬 순서로 두 번째 문자열 앞에 섰습니다. -또는-

반환 값	조건
	첫 번째 문자열은 .입니다 `null`.
0	첫 번째 문자열과 두 번째 문자열은 같습니다. -또는- 두 문자열은 모두 .입니다 `null`.
양의 정수 -또는- 1	첫 번째 문자열은 정렬 순서의 두 번째 문자열을 따릅니다. -또는- 두 번째 문자열은 .입니다 `null`.

> **ⓘ 중요**
>
> 이 String.Compare 메서드는 주로 문자열을 정렬하거나 정렬할 때 사용하기 위한 것입니다. 이 메서드를 String.Compare 사용하여 같음을 테스트하면 안 됩니다(즉, 한 문자열이 다른 문자열보다 작거나 큰지 여부에 관계없이 반환 값 0을 명시적으로 찾으려면). 대신 두 문자열이 같은지 여부를 확인하려면 메서드를 String.Equals(String, String, StringComparison) 사용합니다.

다음 예제에서는 메서드를 String.Compare 사용하여 두 문자열의 상대 값을 확인합니다.

C#

```
string string1 = "Hello World!";
Console.WriteLine(String.Compare(string1, "Hello World?"));
```

이 예제는 `-1` 콘솔에 표시됩니다.

앞의 예제는 기본적으로 문화권을 구분합니다. 문화권을 구분하지 않는 문자열 비교를 수행하려면 String.Compare권 매개 변수를 제공하여 사용할 문화권을 지정할 수 있는 메서드의 오버로드를 사용합니다. 메서드를 사용하여 String.Compare 문화권을 구분하지 않는 비교를 수행하는 방법을 보여 주는 예제는 문화권을 구분하지 않는 문자열 비교를 참조하세요.

`CompareOrdinal` 메서드

이 메서드는 String.CompareOrdinal 로컬 문화권을 고려하지 않고 두 문자열 개체를 비교합니다. 이 메서드의 반환 값은 이전 테이블의 메서드에서 반환한 `Compare` 값과 동일합니다.

> ⓘ 중요
>
> 이 String.CompareOrdinal 메서드는 주로 문자열을 정렬하거나 정렬할 때 사용하기 위한 것입니다. 이 메서드를 String.CompareOrdinal 사용하여 같음을 테스트하면 안 됩니다(즉, 한 문자열이 다른 문자열보다 작거나 큰지 여부에 관계없이 반환 값 0을 명시적으로 찾으려면). 대신 두 문자열이 같은지 여부를 확인하려면 메서드를 String.Equals(String, String, StringComparison) 사용합니다.

다음 예제에서는 메서드를 `CompareOrdinal` 사용하여 두 문자열의 값을 비교합니다.

C#

```
string string1 = "Hello World!";
Console.WriteLine(String.CompareOrdinal(string1, "hello world!"));
```

이 예제는 `-32` 콘솔에 표시됩니다.

`CompareTo` 메서드

이 메서드는 String.CompareTo 현재 문자열 개체가 캡슐화하는 문자열을 다른 문자열 또는 개체와 비교합니다. 이 메서드의 반환 값은 이전 테이블의 메서드에서 반환한 String.Compare 값과 동일합니다.

> ⓘ 중요
>
> 이 String.CompareTo 메서드는 주로 문자열을 정렬하거나 정렬할 때 사용하기 위한 것입니다. 이 메서드를 String.CompareTo 사용하여 같음을 테스트하면 안 됩니다(즉, 한 문자열이 다른 문자열보다 작거나 큰지 여부에 관계없이 반환 값 0을 명시적으로 찾으려면). 대신 두 문자열이 같은지 여부를 확인하려면 메서드를 String.Equals(String, String, StringComparison) 사용합니다.

다음 예제에서는 개체와 String.CompareTo 개체를 `string1` 비교 하는 메서드를 `string2` 사용합니다.

C#

```
string string1 = "Hello World";
string string2 = "Hello World!";
int MyInt = string1.CompareTo(string2);
Console.WriteLine( MyInt );
```

이 예제는 `-1` 콘솔에 표시됩니다.

메서드의 String.CompareTo 모든 오버로드는 기본적으로 문화권 구분 및 대/소문자 구분 비교를 수행합니다. 문화권을 구분하지 않는 비교를 수행할 수 있는 이 메서드의 오버로드가 제공되지 않습니다. 코드 명확성을 위해 문화권 구분 작업 또는 `String.Compare` 문화권을 구분하지 않는 작업에 대해 지정하는 CultureInfo.CurrentCulture 대신 메서드를 사용하는 CultureInfo.InvariantCulture 것이 좋습니다. 메서드를 사용하여 `String.Compare` 문화권을 구분하고 문화권을 구분하지 않는 비교를 수행하는 방법을 보여 주는 예제는 Culture-Insensitive 문자열 비교 수행을 참조하세요.

`Equals` 메서드

이 메서드는 String.Equals 두 문자열이 같은지 쉽게 확인할 수 있습니다. 대/소문자를 구분하는 이 메서드는 값 `true` 또는 `false` 부울 값을 반환합니다. 다음 예제와 같이 기존 클래스에서 사용할 수 있습니다. 다음 예제에서는 메서드를 `Equals` 사용하여 문자열 개체에 "Hello World" 구가 포함되어 있는지 여부를 확인합니다.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.Equals("Hello World"));
```

이 예제는 `True` 콘솔에 표시됩니다.

이 메서드를 정적 메서드로 사용할 수도 있습니다. 다음은 정적 메서드를 사용하는 두 문자열 개체를 비교하는 예제입니다.

C#

```
string string1 = "Hello World";
string string2 = "Hello World";
Console.WriteLine(String.Equals(string1, string2));
```

이 예제는 `True` 콘솔에 표시됩니다.

`StartsWith` 및 `EndsWith` 메서드

이 메서드를 String.StartsWith 사용하여 문자열 개체가 다른 문자열을 포함하는 동일한 문자로 시작하는지 여부를 확인할 수 있습니다. 이 대/소문자 구분 메서드는 현재 문자열 개체가 전달된 문자열로 시작하는 경우와 `true` 그렇지 않은 경우 반환 `false` 합니다. 다음 예제에서는 이 메서드를 사용하여 문자열 개체가 "Hello"로 시작하는지 확인합니다.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.StartsWith("Hello"));
```

이 예제는 `True` 콘솔에 표시됩니다.

이 메서드는 String.EndsWith 전달된 문자열을 현재 문자열 개체의 끝에 있는 문자와 비교합니다. 부울 값도 반환합니다. 다음 예제에서는 메서드를 사용하여 문자열의 끝을 확인합니다 `EndsWith` .

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.EndsWith("Hello"));
```

이 예제는 `False` 콘솔에 표시됩니다.

`IndexOf` 및 `LastIndexOf` 메서드

이 메서드를 String.IndexOf 사용하여 문자열 내에서 특정 문자가 처음 나타나는 위치를 확인할 수 있습니다. 이 대/소문자 구분 메서드는 문자열의 시작 부분에서 계산을 시작하고 0부터 시작하는 인덱스를 사용하여 전달된 문자의 위치를 반환합니다. 문자를 찾을 수 없으면 -1 값이 반환됩니다.

다음 예제에서는 메서드를 `IndexOf` 사용하여 문자열에서 '`l`' 문자의 첫 번째 항목을 검색합니다.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.IndexOf('l'));
```

이 예제는 `2` 콘솔에 표시됩니다.

메서드 String.LastIndexOf 는 문자열 내에서 특정 문자가 마지막으로 나타나는 위치를 반환한다는 점을 제외하고 메서드와 비슷합니다 `String.IndexOf` . 대/소문자를 구분하며 0부터 시작하는 인덱스입니다.

다음 예제에서는 메서드를 `LastIndexOf` 사용하여 문자열에서 '`l`' 문자의 마지막 항목을 검색합니다.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.LastIndexOf('l'));
```

이 예제는 `9` 콘솔에 표시됩니다.

두 메서드는 메서드와 함께 String.Remove 사용할 때 유용합니다. 문자 또는 `IndexOf` 메서드를 `LastIndexOf` 사용하여 문자의 위치를 검색한 다음, 해당 문자 또는 해당 문자로 시작하는 단어를 제거하기 위해 메서드에 해당 위치를 `Remove` 제공할 수 있습니다.

참고하십시오

- .NET에서 문자열을 사용하는 모범 사례
- 기본 문자열 작업
- 문화에 구애받지 않는 문자열 작업을 수행하기
- 가중치 테이블 정렬 - Windows에서 .NET Framework 및 .NET Core 1.0-3.1에서 사용
- 기본 유니코드 데이터 정렬 요소 테이블 - 모든 플랫폼의 .NET 5 및 Linux 및 macOS의 .NET Core에서 사용

.NET의 변경 사례

아티클 • 2025. 04. 09.

사용자의 입력을 허용하는 애플리케이션을 작성하는 경우 데이터를 입력하는 데 사용할 대/소문자(위 또는 아래)를 확신할 수 없습니다. 특히 사용자 인터페이스에 문자열을 표시하는 경우 문자열의 대/소문자를 일관되게 지정하려는 경우가 많습니다. 다음 표에서는 세 가지 대/소문자 변경 메서드에 대해 설명합니다. 처음 두 메서드는 문화권을 수용하는 오버로드를 제공합니다.

테이블 확장

메서드 이름	사용하세요
String.ToUpper	문자열의 모든 문자를 대문자로 변환합니다.
String.ToLower	문자열의 모든 문자를 소문자로 변환합니다.
TextInfo.ToTitleCase	문자열을 타이틀 케이스로 변환합니다.

> ⚠ **경고**
>
> String.ToUpper 문자열을 비교하거나 같은지 테스트하기 위해 문자열을 변환하는 데 메서드와 String.ToLower 메서드를 사용하면 안 됩니다. 자세한 내용은 **혼합 사례의 문자열 비교 섹션을 참조하세요** .

혼합된 대/소문자의 문자열 비교

혼합 대소문자의 문자열을 비교하여 그 순서를 결정하려면, `comparisonType` 매개 변수를 사용하여 String.CompareTo 메서드의 오버로드 중 하나를 호출하고, `comparisonType` 인수에 대해 StringComparison.CurrentCultureIgnoreCase, StringComparison.InvariantCultureIgnoreCase, 또는 StringComparison.OrdinalIgnoreCase의 값을 제공합니다. 현재 문화권이 아닌 특정 문화권을 사용하여 비교하려면, String.CompareTo 메서드의 오버로드를 `culture` 및 `options` 매개변수를 사용하여 호출하고, `options` 인수에 CompareOptions.IgnoreCase 값을 제공합니다.

혼합 대소문자 문자열을 비교하여 같은지 여부를 확인하려면, `comparisonType` 매개 변수를 사용하는 String.Equals 메서드의 오버로드 중 하나를 호출하고 `comparisonType` 인수에 대해 StringComparison.CurrentCultureIgnoreCase, StringComparison.InvariantCultureIgnoreCase, StringComparison.OrdinalIgnoreCase 중 하나의 값을 제공하십시오.

자세한 내용은 문자열 사용에 대한 모범 사례를 참조하세요.

`ToUpper` 메서드

이 메서드는 String.ToUpper 문자열의 모든 문자를 대문자로 변경합니다. 다음 예제에서는 문자열 "Hello World!"를 혼합 대/소문자에서 대문자로 변환합니다.

C#

```
string properString = "Hello World!";
Console.WriteLine(properString.ToUpper());
// This example displays the following output:
//       HELLO WORLD!
```

앞의 예제는 기본적으로 문화에 민감하게 반응하며, 현재 문화권의 대/소문자 규칙을 적용합니다. 특정 문화권의 대/소문자 규칙을 적용하거나 문화권을 구분하지 않는 대/소문자 변경을 수행하려면 String.ToUpper(CultureInfo) 메서드 오버로드를 사용하고, 지정된 문화권을 나타내는 값 CultureInfo.InvariantCulture 또는 System.Globalization.CultureInfo 개체를 `culture` 매개 변수로 제공합니다. 메서드를 사용하여 ToUpper 문화권을 구분하지 않는 대/소문자 변경을 수행하는 방법을 보여 주는 예제는 문화권을 구분하지 않는 대/소문자 변경 수행을 참조하세요.

`ToLower` 메서드

이 String.ToLower 메서드는 이전 메서드와 비슷하지만 대신 문자열의 모든 문자를 소문자로 변환합니다. 다음 예제에서는 문자열 "Hello World!"를 소문자로 변환합니다.

C#

```
string properString = "Hello World!";
Console.WriteLine(properString.ToLower());
// This example displays the following output:
//       hello world!
```

앞의 예제는 기본적으로 현재 문화권의 대/소문자 규칙을 적용하여 문화에 민감하게 설계되었습니다. 문화권과 무관하게 대소문자 변환을 수행하거나 특정 문화권의 대소문자 규칙을 적용하려면, 메서드 오버로드를 사용해야 하며, `culture` 매개 변수에 지정된 문화권을 나타내는 값으로 CultureInfo.InvariantCulture이나 System.Globalization.CultureInfo 객체를 제공합니다. 메서드를 사용하여 ToLower(CultureInfo) 문화권을 구분하지 않는 대/소문자 변경을 수행하는 방법을 보여 주는 예제는 문화권을 구분하지 않는 대/소문자 변경 수행을 참조하세요.

`ToTitleCase` 메서드

각 TextInfo.ToTitleCase 단어의 첫 번째 문자를 대문자로 변환하고 나머지 문자를 소문자로 변환합니다. 그러나 완전히 대문자인 단어는 약어로 간주되며 변환되지 않습니다.

메서드 TextInfo.ToTitleCase 는 문화권을 구분합니다. 즉, 특정 문화권의 대/소문자 규칙을 사용합니다. 메서드를 호출하려면 먼저 특정 문화권의 CultureInfo.TextInfo 속성에서 해당 문화권의 대/소문자 규칙을 나타내는 TextInfo 개체를 검색해야 합니다.

다음 예제에서는 배열의 각 문자열을 메서드에 전달합니다 TextInfo.ToTitleCase . 문자열에는 적절한 제목 문자열과 약어가 포함됩니다. 문자열은 영어(미국) 문화권의 대/소문자 규칙에 따라 제목 형식으로 변환됩니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { "a tale of two cities", "gROWL to the rescue",
                          "inside the US government", "sports and MLB baseball",
                          "The Return of Sherlock Holmes", "UNICEF and children"};

      TextInfo ti = CultureInfo.CurrentCulture.TextInfo;
      foreach (var value in values)
         Console.WriteLine($"{value} --> {ti.ToTitleCase(value)}");
   }
}
// The example displays the following output:
//    a tale of two cities --> A Tale Of Two Cities
//    gROWL to the rescue --> Growl To The Rescue
//    inside the US government --> Inside The US Government
//    sports and MLB baseball --> Sports And MLB Baseball
//    The Return of Sherlock Holmes --> The Return Of Sherlock Holmes
//    UNICEF and children --> UNICEF And Children
```

이 메서드는 문화에 민감하지만, 언어적으로 올바른 대/소문자 규칙을 제공하지 않습니다. 예를 들어 이전 예제에서 이 메서드는 "두 도시의 이야기"를 "두 도시의 이야기"로 변환합니다. 그러나 en-US 문화권에서 언어적으로 올바른 제목 표기법은 "두 도시의 이야기"입니다.

참고하십시오

- 기본 문자열 작업
- 문화에 구애받지 않는 문자열 작업을 수행하기

문자열에서 부분 문자열 주출

이 문서에서는 문자열의 일부를 추출하는 몇 가지 방법을 설명합니다.

- 원하는 부분 문자열이 알려진 구분 문자(또는 문자)로 구분된 경우 Split 메서드를 사용합니다.
- 정규식은 문자열이 고정 패턴을 준수하는 경우에 유용합니다.
- 문자열의 모든 부분 문자열을 추출하려는 것이 아니라면 *IndexOf 메서드와 Substring 메서드*를 함께 사용하세요.
- C# 의 범위 및 인덱스를 사용하여 알려진 위치에서 문자를 추출하거나 트리밍합니다.

String.Split 메서드

String.Split은 지정하는 하나 이상의 구분 문자에 따라 문자열을 부분 문자열 그룹으로 분할하는 데 도움이 되는 몇 가지 오버로드를 제공합니다. 최종 결과에서 전체 부분 문자열 수를 제한하거나, 부분 문자열에서 공백 문자를 자르거나, 빈 부분 문자열을 제외할 수 있습니다.

다음 예제는 `String.Split()`의 세 가지 오버로드를 보여 줍니다. 첫 번째 예제는 구분 문자를 전달하지 않고 Split(Char[]) 오버로드를 호출합니다. 구분 문자를 지정하지 않으면 `String.Split()`은 공백 문자인 기본 구분 기호를 사용하여 문자열을 분할합니다.

C#

```
string s = "You win some. You lose some.";

string[] subs = s.Split();

foreach (string sub in subs)
{
    Console.WriteLine($"Substring: {sub}");
}

// This example produces the following output:
//
// Substring: You
// Substring: win
// Substring: some.
// Substring: You
// Substring: lose
// Substring: some.
```

여기에서 볼 수 있듯이 마침표 문자(`.`)가 두 부분 문자열에 포함됩니다. 마침표 문자를 제외하려는 경우 마침표 문자를 추가 구분 문자로 추가할 수 있습니다. 다음 예제는 이 작업을 수행하는 방법을 보여 줍니다.

C#

```
string s = "You win some. You lose some.";

string[] subs = s.Split(' ', '.');

foreach (string sub in subs)
{
    Console.WriteLine($"Substring: {sub}");
}

// This example produces the following output:
//
// Substring: You
// Substring: win
// Substring: some
// Substring:
// Substring: You
// Substring: lose
// Substring: some
// Substring:
```

부분 문자열에서 마침표가 사라졌지만 이제는 두 개의 빈 부분 문자열이 추가로 포함되었습니다. 이러한 빈 부분 문자열은 단어와 그 뒤에 오는 마침표 사이의 부분 문자열을 나타냅니다. 결과 배열에서 빈 부분 문자열을 생략하려면 Split(Char[], StringSplitOptions) 오버로드를 호출하고 StringSplitOptions.RemoveEmptyEntries 매개 변수의 `options`을 지정합니다.

C#

```
string s = "You win some. You lose some.";
char[] separators = new char[] { ' ', '.' };

string[] subs = s.Split(separators, StringSplitOptions.RemoveEmptyEntries);

foreach (string sub in subs)
{
    Console.WriteLine($"Substring: {sub}");
}

// This example produces the following output:
//
// Substring: You
// Substring: win
// Substring: some
// Substring: You
// Substring: lose
// Substring: some
```

정규식

문자열이 고정 패턴을 따르는 경우 정규식을 사용하여 해당 요소를 추출하고 처리할 수 있습니다. 예를 들어 문자열이 "숫자 피연산자 숫자" 형식을 사용하는 경우 정규식을 사용하여 문자열의 요소를 추출하고 처리할 수 있습니다. 예를 들면 다음과 같습니다.

C#

```
String[] expressions = { "16 + 21", "31 * 3", "28 / 3",
                         "42 - 18", "12 * 7",
                         "2, 4, 6, 8" };
String pattern = @"(\d+)\s+([-+*/])\s+(\d+)";

foreach (string expression in expressions)
{
    foreach (System.Text.RegularExpressions.Match m in
    System.Text.RegularExpressions.Regex.Matches(expression, pattern))
    {
        int value1 = Int32.Parse(m.Groups[1].Value);
        int value2 = Int32.Parse(m.Groups[3].Value);
        switch (m.Groups[2].Value)
        {
            case "+":
                Console.WriteLine($"{m.Value} = {value1 + value2}");
                break;
            case "-":
                Console.WriteLine($"{m.Value} = {value1 - value2}");
                break;
            case "*":
                Console.WriteLine($"{m.Value} = {value1 * value2}");
                break;
            case "/":
                Console.WriteLine($"{m.Value} = {value1 / value2:N2}");
                break;
        }
    }
}

// The example displays the following output:
//       16 + 21 = 37
//       31 * 3 = 93
//       28 / 3 = 9.33
//       42 - 18 = 24
//       12 * 7 = 84
```

정규식 패턴 `(\d+)\s+([-+*/])\s+(\d+)`는 다음과 같이 정의됩니다.

테이블 확장

패턴	설명
`(\d+)`	하나 이상의 10진수 숫자가 일치하는지 확인합니다. 이 그룹은 첫 번째 캡처링 그룹입니다.

패턴	설명
`\s+`	하나 이상의 공백 문자를 찾습니다.
`([-+*/])`	산술 연산자 기호(+, -, *, /)를 찾습니다. 이것이 두 번째 캡처링 그룹입니다.
`\s+`	하나 이상의 공백 문자를 찾습니다.
`(\d+)`	하나 이상의 10진수 숫자가 일치하는지 확인합니다. 이 그룹은 세 번째 캡처링 그룹입니다.

정규식을 사용하여 고정 문자 집합이 아닌 패턴을 기반으로 문자열에서 부분 문자열을 추출할 수도 있습니다. 다음은 이러한 조건 중 하나가 발생하는 일반적인 시나리오입니다.

- 하나 이상의 구분 문자가 인스턴스에서 항상 String구분 기호로 사용되지는 않습니다.
- 구분 문자의 시퀀스와 수는 변수이거나 알 수 없습니다.

예를 들어 Split 메서드는 다음 문자열을 분할하는 데 사용할 수 없습니다. `\n`(줄 바꿈) 문자 수가 변수이고 이 문자가 항상 구분 기호로 사용되지는 않기 때문입니다.

text

```
[This is captured\ntext.]\n\n[\n[This is more captured text.]\n]
\n[Some more captured text:\n   Option1\n   Option2][Terse text.]
```

정규식은 다음 예제에서 볼 수 있듯이 이 문자열을 쉽게 분할할 수 있습니다.

C#

```
String input = "[This is captured\ntext.]\n\n[\n" +
               "[This is more captured text.]\n]\n" +
               "[Some more captured text:\n   Option1" +
               "\n   Option2][Terse text.]";
String pattern = @"\[([^\[\]]+)\]";
int ctr = 0;

foreach (System.Text.RegularExpressions.Match m in
   System.Text.RegularExpressions.Regex.Matches(input, pattern))
{
    Console.WriteLine($"{++ctr}: {m.Groups[1].Value}");
}

// The example displays the following output:
//       1: This is captured
//       text.
//       2: This is more captured text.
//       3: Some more captured text:
//          Option1
```

```
//          Option2
//       4: Terse text.
```

정규식 패턴 `\[([^\[\]]+)\]`는 다음과 같이 정의됩니다.

테이블 확장

패턴	설명
`\[`	여는 대괄호를 맞춥니다.
`([^\[\]]+)`	여는 대괄호나 닫는 대괄호가 아닌 문자를 한 번 이상 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\]`	닫는 대괄호를 맞춥니다.

Regex.Split 메서드는 고정 문자 집합 대신 정규식 패턴을 기반으로 문자열을 분할한다는 점을 제외하면 String.Split과 거의 동일합니다. 예를 들어 다음 예제에서는 Regex.Split 메서드를 사용하여 하이픈과 기타 문자의 다양한 조합으로 구분된 부분 문자열이 포함된 문자열을 분할합니다.

C#

```
String input = "abacus -- alabaster - * - atrium -+- " +
               "any -*- actual - + - armoire - - alarm";
String pattern = @"\s-\s?[+*]?\s?-\s";
String[] elements = System.Text.RegularExpressions.Regex.Split(input, pattern);

foreach (string element in elements)
    Console.WriteLine(element);

// The example displays the following output:
//       abacus
//       alabaster
//       atrium
//       any
//       actual
//       armoire
//       alarm
```

정규식 패턴 `\s-\s?[+*]?\s?-\s`는 다음과 같이 정의됩니다.

테이블 확장

패턴	설명
`\s-`	하이픈이 뒤에 오는 공백 문자를 찾습니다.

패턴	설명
`\s?`	0번 또는 1번 나오는 공백 문자와 일치합니다.
`[+*]?`	0번 또는 한 번 나오는 + 또는 * 문자를 찾습니다.
`\s?`	0번 또는 1번 나오는 공백 문자와 일치합니다.
`-\s`	뒤에 공백 문자가 오는 하이픈을 찾습니다.

String.IndexOf 및 String.Substring 메서드

문자열의 모든 부분 문자열에 관심이 있는 것이 아니라면 일치가 시작되는 인덱스를 반환하는 문자열 비교 메서드 중 하나를 사용하는 것이 좋습니다. 그런 다음 Substring 메서드를 호출하여 원하는 부분 문자열을 추출할 수 있습니다. 문자열 비교 메서드는 다음과 같습니다.

- 문자열 인스턴스에서 처음 나오는 문자 또는 문자열의 0부터 시작하는 인덱스를 반환하는 IndexOf
- 문자 배열에서 문자가 처음 나오는 현재 문자열 인스턴스의 0부터 시작하는 인덱스를 반환하는 IndexOfAny
- 문자열 인스턴스에서 마지막으로 나오는 문자 또는 문자열의 0부터 시작하는 인덱스를 반환하는 LastIndexOf
- 문자 배열에서 문자가 마지막으로 나오는 현재 문자열 인스턴스의 0부터 시작하는 인덱스를 반환하는 LastIndexOfAny

다음 예제는 IndexOf 메서드를 사용하여 문자열의 마침표 위치를 찾는 방법을 보여줍니다. 그런 다음 Substring 메서드를 사용하여 전체 문장을 반환합니다.

C#

```
String s = "This is the first sentence in a string. " +
                "More sentences will follow. For example, " +
                "this is the third sentence. This is the " +
                "fourth. And this is the fifth and final " +
                "sentence.";
var sentences = new List<String>();
int start = 0;
int position;

// Extract sentences from the string.
do
{
    position = s.IndexOf('.', start);
    if (position >= 0)
    {
```

```
        sentences.Add(s.Substring(start, position - start + 1).Trim());
        start = position + 1;
    }
} while (position > 0);

// Display the sentences.
foreach (var sentence in sentences)
    Console.WriteLine(sentence);

// The example displays the following output:
//       This is the first sentence in a string.
//       More sentences will follow.
//       For example, this is the third sentence.
//       This is the fourth.
//       And this is the fifth and final sentence.
```

범위 및 인덱스

C# 범위 연산 .. 자와 인덱스-from-end 연산 ^ 자를 사용하면 간결한 구문을 사용하여 부분 문자열을 추출할 수 있습니다. 이러한 연산자를 호출 Substring하지 않고 문자열에 직접 적용할 수 있습니다.

다음 예제에서는 범위를 사용하여 문자열의 일부를 추출하는 여러 가지 방법을 보여 줍니다.

C#

```
string str = "Hello, World!";

// Get the first 5 characters.
string hello = str[..5];
Console.WriteLine(hello);
// Output: Hello

// Get the last 6 characters.
string world = str[^6..];
Console.WriteLine(world);
// Output: World!

// Get characters from index 7 through 11 (exclusive of 12).
string substr = str[7..12];
Console.WriteLine(substr);
// Output: World
```

다음 예제에서는 인덱스-from-end 연산자를 사용하여 경로에서 파일 확장명(마지막 3자)을 제거합니다.

C#

```
string filePath = "C:\\Users\\user1\\bin\\fileA.cs";

// Remove the last 3 characters (.cs extension).
string trimmedPath = filePath[..^3];
Console.WriteLine(trimmedPath);
// Output: C:\Users\user1\bin\fileA
```

> ⓘ **참고 항목**
>
> 범위 및 끝에서부터의 인덱스 연산자는 C#의 기능입니다. Visual Basic은 이 구문을 지원하지 않습니다. 대신 사용합니다 **Substring** .

참고 항목

- .NET 정규식
- C#에서 String.Split을 사용하여 문자열을 구문 분석하는 방법
- 인덱스 및 범위(C# 가이드)

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 24.

.NET에서 StringBuilder 클래스 사용

아티클 • 2025. 05. 08.

String 개체는 변경할 수 없습니다. 클래스의 메서드 중 하나를 사용할 때마다 메모리에 System.String 새 문자열 개체를 만들면 해당 새 개체에 대한 공간을 새로 할당해야 합니다. 문자열을 반복적으로 수정해야 하는 경우 새 String 개체를 만드는 것과 관련된 오버헤드는 비용이 많이 들 수 있습니다. System.Text.StringBuilder 새 개체를 만들지 않고 문자열을 수정하려는 경우 클래스를 사용할 수 있습니다. 예를 들어 루프에서 여러 문자열을 StringBuilder 함께 연결할 때 클래스를 사용하면 성능이 향상될 수 있습니다.

System.Text 네임스페이스 가져오기

클래스는 StringBuilder 네임스페이스에 System.Text 있습니다. 코드에서 정규화된 형식 이름을 제공할 필요가 없도록 네임스페이스를 System.Text 가져올 수 있습니다.

C#

```
using System;
using System.Text;
```

StringBuilder 개체 인스턴스화

다음 예제와 같이 오버로드된 생성자 메서드 중 하나를 사용하여 변수를 초기화하여 클래스의 새 인스턴스 StringBuilder 를 만들 수 있습니다.

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
```

용량 및 길이 설정

StringBuilder 이 개체는 캡슐화된 문자열의 문자 수를 확장할 수 있는 동적 개체이지만 저장할 수 있는 최대 문자 수에 대한 값을 지정할 수 있습니다. 이 값을 개체의 용량이라고 하며 현재 StringBuilder 보유하는 문자열의 길이와 혼동해서는 안 됩니다. 예를 들어 길이가 5인 "Hello" 문자열을 사용하여 클래스의 StringBuilder 새 인스턴스를 만들고 개체의 최대 용량이 25임을 지정할 수 있습니다. 수정할 때 용량에 StringBuilder도달할 때까지 크기 자체를 다시 할당하지 않습니다. 이 경우 새 공간이 자동으로 할당되고 용량이 두 배가 됩니다. 오버로드된 생성자 중 하나를 사용하여 클래스의 StringBuilder 용량을 지정할 수 있습니다. 다음 예제에서는 개체를 `myStringBuilder` 최대 25개의 공백으로 확장할 수 있도록 지정합니다.

```
C#

StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);
```

또한 읽기/쓰기 Capacity 속성을 사용하여 개체의 최대 길이를 설정할 수 있습니다. 다음 예제에서는 **Capacity** 속성을 사용하여 최대 개체 길이를 정의합니다.

```
C#

myStringBuilder.Capacity = 25;
```

이 메서드를 EnsureCapacity 사용하여 현재 **StringBuilder**의 용량을 확인할 수 있습니다. 용량이 전달된 값보다 크면 변경되지 않습니다. 그러나 용량이 전달된 값보다 작으면 전달된 값과 일치하도록 현재 용량이 변경됩니다.

속성을 Length 보거나 설정할 수도 있습니다. **Length** 속성을 **Capacity** 속성보다 큰 값으로 설정하면 **Capacity** 속성이 **Length** 속성과 동일한 값으로 자동으로 변경됩니다. **Length** 속성을 현재 **StringBuilder** 내의 문자열 길이보다 작은 값으로 설정하면 문자열이 줄어듭니다.

StringBuilder 문자열 수정

다음 표에서는 **StringBuilder**의 내용을 수정하는 데 사용할 수 있는 메서드를 나열합니다.

테이블 확장

메서드 이름	사용하세요
StringBuilder.Append	현재 **StringBuilder**의 끝에 정보를 추가합니다.
StringBuilder.AppendFormat	문자열에 전달된 서식 지정자를 서식이 지정된 텍스트로 바꿉니다.
StringBuilder.Insert	문자열 또는 개체를 현재 **StringBuilder**의 지정된 인덱스에 삽입합니다.
StringBuilder.Remove	현재 **StringBuilder**에서 지정된 수의 문자를 제거합니다.
StringBuilder.Replace	현재 **StringBuilder** 에서 지정된 문자 또는 문자열의 모든 항목을 다른 지정된 문자 또는 문자열로 바꿉니다.

추가

Append 메서드를 사용하여 현재 **StringBuilder**가 나타내는 문자열 끝에 개체의 텍스트 또는 문자열 표현을 추가할 수 있습니다. 다음 예제에서는 **StringBuilder** 를 "Hello World"로 초기화한 다음 개체의 끝에 일부 텍스트를 추가합니다. 공간은 필요에 따라 자동으로 할당됩니다.

```
```

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Append(" What a beautiful day.");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World! What a beautiful day.
```

AppendFormat

메서드는 StringBuilder.AppendFormat 개체의 StringBuilder 끝에 텍스트를 추가합니다. 서식을 지정할 개체 또는 개체의 구현을 호출하여 IFormattable 지원합니다. 따라서 숫자, 날짜 및 시간 및 열거형 값에 대한 표준 서식 문자열, 숫자 및 날짜 및 시간 값에 대한 사용자 지정 서식 문자열 및 사용자 지정 형식에 대해 정의된 서식 문자열을 허용합니다. 서식 지정에 대한 자세한 내용은 형식 서식을 참조하세요. 이 메서드를 사용하여 변수 형식을 사용자 지정하고 해당 값을 StringBuilder에 추가할 수 있습니다. 다음 예제에서는 메서드를 AppendFormat 사용 하 여 개체의 끝에 통화 값으로 형식이 지정된 정수 값을 배치 합니다 StringBuilder .

C#

```
int MyInt = 25;
StringBuilder myStringBuilder = new StringBuilder("Your total is ");
myStringBuilder.AppendFormat("{0:C} ", MyInt);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Your total is $25.00
```

삽입

메서드는 Insert 문자열 또는 개체를 현재 StringBuilder 개체의 지정된 위치에 추가합니다. 다음은 이 메서드를 사용하여 개체의 여섯 번째 위치에 단어를 삽입하는 예제입니다 StringBuilder .

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Insert(6,"Beautiful ");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello Beautiful World!
```

제거하십시오

Remove 메서드를 사용하여 지정된 0부터 시작하는 인덱스에서 시작하여 현재 StringBuilder 개체에서 지정된 수의 문자를 제거할 수 있습니다. 다음 예제에서는 **Remove** 메서드를 사용하여 개체를 줄입니다 StringBuilder .

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Remove(5,7);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello
```

교체합니다

Replace 메서드를 사용하여 개체 내의 StringBuilder 문자를 지정된 다른 문자로 바꿀 수 있습니다. 다음 예제에서는 **Replace** 메서드를 사용하여 느낌표 문자(!)의 모든 인스턴스에 대한 개체를 검색 StringBuilder 하고 물음표 문자(?)로 바꿉니다.

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Replace('!', '?');
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World?
```

StringBuilder 개체를 문자열로 변환

개체가 나타내는 문자열을 StringBuilder 매개 변수가 있는 String 메서드 StringBuilder 에 String 전달하거나 사용자 인터페이스에 표시하려면 먼저 개체를 개체로 변환해야 합니다. 메서드를 호출하여 이 변환을 수행합니다 StringBuilder.ToString . 다음 예제에서는 여러 StringBuilder 메서드를 호출한 다음 메서드를 StringBuilder.ToString() 호출하여 문자열을 표시합니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      StringBuilder sb = new StringBuilder();
      bool flag = true;
      string[] spellings = { "recieve", "receeve", "receive" };
```

```
      sb.AppendFormat("Which of the following spellings is {0}:", flag);
      sb.AppendLine();
      for (int ctr = 0; ctr <= spellings.GetUpperBound(0); ctr++) {
         sb.AppendFormat("   {0}. {1}", ctr, spellings[ctr]);
         sb.AppendLine();
      }
      sb.AppendLine();
      Console.WriteLine(sb.ToString());
   }
}
// The example displays the following output:
//       Which of the following spellings is True:
//          0. recieve
//          1. receeve
//          2. receive
```

참고하십시오

- System.Text.StringBuilder
- 기본 문자열 작업
- 서식 유형

방법: .NET에서 기본 문자열 조작 수행

다음 예제에서는 실제 애플리케이션에서 찾을 수 있는 방식으로 문자열 조작을 수행하는 클래스를 생성합니다. 클래스는 `MailToData` 개별의 이름과 주소를 별도의 속성에 저장하고 , `City` 및 `State` 필드를 사용자에게 표시할 단일 문자열로 결합 `Zip` 하는 방법을 제공합니다. 또한 클래스를 사용하면 사용자가 도시, 주 및 우편 번호 정보를 단일 문자열로 입력할 수 있습니다. 애플리케이션은 자동으로 단일 문자열을 구문 분석하고 해당 속성에 적절한 정보를 입력합니다.

간단히 하기 위해 이 예제에서는 명령줄 인터페이스가 있는 콘솔 애플리케이션을 사용합니다.

예시

C#

```
using System;

class MainClass
{
   static void Main()
   {
      MailToData MyData = new MailToData();

      Console.Write("Enter Your Name: ");
      MyData.Name = Console.ReadLine();
      Console.Write("Enter Your Address: ");
      MyData.Address = Console.ReadLine();
      Console.Write("Enter Your City, State, and ZIP Code separated by spaces: ");
      MyData.CityStateZip = Console.ReadLine();
      Console.WriteLine();

      if (MyData.Validated) {
         Console.WriteLine($"Name: {MyData.Name}");
         Console.WriteLine($"Address: {MyData.Address}");
         Console.WriteLine($"City: {MyData.City}");
         Console.WriteLine($"State: {MyData.State}");
         Console.WriteLine($"Zip: {MyData.Zip}");

         Console.WriteLine("\nThe following address will be used:");
         Console.WriteLine(MyData.Address);
         Console.WriteLine(MyData.CityStateZip);
      }
   }
}

public class MailToData
{
   string name = "";
   string address = "";
   string citystatezip = "";
   string city = "";
```

```
   string state = "";
   string zip = "";
   bool parseSucceeded = false;

   public string Name
   {
      get{return name;}
      set{name = value;}
   }

   public string Address
   {
      get{return address;}
      set{address = value;}
   }

   public string CityStateZip
   {
      get {
         return String.Format("{0}, {1} {2}", city, state, zip);
      }
      set {
         citystatezip = value.Trim();
         ParseCityStateZip();
      }
   }

   public string City
   {
      get{return city;}
      set{city = value;}
   }

   public string State
   {
      get{return state;}
      set{state = value;}
   }

   public string Zip
   {
      get{return zip;}
      set{zip = value;}
   }

   public bool Validated
   {
      get { return parseSucceeded; }
   }

   private void ParseCityStateZip()
   {
      string msg = "";
      const string msgEnd = "\nYou must enter spaces between city, state, and zip
code.\n";
```

```
      // Throw a FormatException if the user did not enter the necessary spaces
      // between elements.
      try
      {
         // City may consist of multiple words, so we'll have to parse the
         // string from right to left starting with the zip code.
         int zipIndex = citystatezip.LastIndexOf(" ");
         if (zipIndex == -1) {
            msg = "\nCannot identify a zip code." + msgEnd;
            throw new FormatException(msg);
         }
         zip = citystatezip.Substring(zipIndex + 1);

         int stateIndex = citystatezip.LastIndexOf(" ", zipIndex - 1);
         if (stateIndex == -1) {
            msg = "\nCannot identify a state." + msgEnd;
            throw new FormatException(msg);
         }
         state = citystatezip.Substring(stateIndex + 1, zipIndex - stateIndex - 1);
         state = state.ToUpper();

         city = citystatezip.Substring(0, stateIndex);
         if (city.Length == 0) {
            msg = "\nCannot identify a city." + msgEnd;
            throw new FormatException(msg);
         }
         parseSucceeded = true;
      }
      catch (FormatException ex)
      {
         Console.WriteLine(ex.Message);
      }
   }

   private string ReturnCityStateZip()
    {
        // Make state uppercase.
        state = state.ToUpper();

        // Put the value of city, state, and zip together in the proper manner.
        string MyCityStateZip = String.Concat(city, ", ", state, " ", zip);

        return MyCityStateZip;
    }
}
```

앞의 코드가 실행되면 사용자에게 이름과 주소를 입력하라는 메시지가 표시됩니다. 애플리케이션은 정보를 적절한 속성에 배치하고 정보를 사용자에게 다시 표시하여 도시, 주 및 우편 번호 정보를 표시하는 단일 문자열을 만듭니다.

Last updated on 2025. 11. 13.

.NET에서 문자열 구문 분석

2025. 06. 17.

구문 분석 작업은 .NET 기본 형식을 나타내는 문자열을 해당 기본 형식으로 변환합니다. 예를 들어 구문 분석 작업은 문자열을 부동 소수점 숫자 또는 날짜 및 시간 값으로 변환하는 데 사용됩니다. 구문 분석 작업을 수행하는 데 가장 일반적으로 사용되는 메서드는 메서드입니다 `Parse` . 구문 분석이 기본 형식을 문자열 표현으로 변환하는 것과 관련된 서식 지정의 역방향 작업이므로 동일한 규칙과 규칙이 많이 적용됩니다. 인터페이스 IFormatProvider를 구현하는 개체를 사용하여 문화권에 맞는 서식 정보를 제공하는 것처럼, 문자열 표현을 해석하는 방법을 결정하기 위해 인터페이스 IFormatProvider를 구현하는 개체를 구문 분석에 사용합니다. 자세한 내용은 형식 형식을 참조하세요.

이 섹션 안에

숫자 문자열 파싱
문자열을 .NET 숫자 형식으로 변환하는 방법을 설명합니다.

날짜 및 시간 문자열 구문 분석
문자열을 .NET **DateTime** 형식으로 변환하는 방법을 설명합니다.

다른 문자열을 파싱하기
문자열을 **Char**, **Boolean** 및 **Enum** 형식으로 변환하는 방법을 설명합니다.

관련 섹션

서식 유형
형식 지정자 및 형식 공급자와 같은 기본 서식 개념을 설명합니다.

.NET의 형식 변환
형식을 변환하는 방법을 설명합니다.

.NET에서 숫자 문자열 구문 분석

2025. 06. 28.

모든 숫자 형식에는 `Parse` 및 `TryParse` 두 개의 정적 구문 분석 메서드가 있으며, 이를 통해 숫자의 문자열 표현을 숫자 형식으로 변환할 수 있습니다. 이러한 메서드를 사용하면 표준 숫자 형식 문자열 및 사용자 지정 숫자 형식 문자열에 설명된 형식 문자열을 사용하여 생성된 문자열을 구문 분석할 수 있습니다. 기본적으로 `Parse` 및 `TryParse` 메서드는 정수로만 구성된 문자열을 정수 값으로 변환할 수 있습니다. 정수 및 소수 자릿수, 그룹 구분 기호 및 소수 구분 기호를 포함하는 문자열을 부동 소수점 값으로 성공적으로 변환할 수 있습니다. `Parse` 메서드는 작업이 실패하면 예외를 throw하며, `TryParse` 메서드는 `false`를 반환합니다.

> ⓘ **참고**
>
> .NET 7부터, .NET의 숫자 형식도 System.IParsable<TSelf> 인터페이스를 구현하여 IParsable<TSelf>.Parse 및 IParsable<TSelf>.TryParse 메서드를 정의합니다.

파싱 및 형식 제공자

일반적으로 숫자 값의 문자열 표현은 문화권에 따라 다릅니다. 통화 기호, 그룹(또는 수천) 구분 기호 및 소수 구분 기호와 같은 숫자 문자열의 요소는 모두 문화권에 따라 다릅니다. 구문 분석 방법은 암시적으로 또는 명시적으로 이러한 문화권별 변형을 인식하는 형식 공급자를 사용합니다. `Parse` 또는 `TryParse` 메서드를 호출할 때 형식 공급자를 지정하지 않으면 현재 문화권과 연결된 형식 공급자(NumberFormatInfo 속성에서 반환된 NumberFormatInfo.CurrentInfo 오브젝트)가 사용됩니다.

형식 공급자는 IFormatProvider 구현으로 나타납니다. 이 인터페이스에는 서식을 지정할 형식을 나타내는 Type 개체를 단일 매개 변수로 갖는 단일 멤버 메서드 GetFormat가 있습니다. 이 메서드는 서식 정보를 제공하는 개체를 반환합니다. .NET은 숫자 문자열을 구문 분석하기 위해 다음 두 IFormatProvider 가지 구현을 지원합니다.

- CultureInfo 메서드가 문화권별 서식 지정 정보를 제공하는 NumberFormatInfo 개체를 반환하는 CultureInfo.GetFormat 개체입니다.
- NumberFormatInfo 메서드가 자신을 반환하는 NumberFormatInfo.GetFormat 개체입니다.

다음 예제에서는 배열의 각 문자열을 값으로 Double 변환하려고 시도합니다. 먼저 영어(미국) 문화권의 규칙을 반영하는 형식 공급자를 사용하여 문자열을 구문 분석하려고 합니다. 이 연산이 FormatException를 발생하면, 프랑스어(프랑스) 문화권의 규칙을 반영하는 형식 제공자를 사용하여 문자열을 구문 분석하려고 합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { "1,304.16", "$1,456.78", "1,094", "152",
                          "123,45 €", "1 304,16", "Ae9f" };
      double number;
      CultureInfo culture = null;

      foreach (string value in values) {
         try {
            culture = CultureInfo.CreateSpecificCulture("en-US");
            number = Double.Parse(value, culture);
            Console.WriteLine($"{culture.Name}: {value} --> {number}");
         }
         catch (FormatException) {
            Console.WriteLine($"{culture.Name}: Unable to parse '{value}'.");
            culture = CultureInfo.CreateSpecificCulture("fr-FR");
            try {
               number = Double.Parse(value, culture);
               Console.WriteLine($"{culture.Name}: {value} --> {number}");
            }
            catch (FormatException) {
               Console.WriteLine($"{culture.Name}: Unable to parse '{value}'.");
            }
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    en-US: 1,304.16 --> 1304.16
//
//    en-US: Unable to parse '$1,456.78'.
//    fr-FR: Unable to parse '$1,456.78'.
//
//    en-US: 1,094 --> 1094
//
//    en-US: 152 --> 152
//
//    en-US: Unable to parse '123,45 €'.
//    fr-FR: Unable to parse '123,45 €'.
//
//    en-US: Unable to parse '1 304,16'.
//    fr-FR: 1 304,16 --> 1304.16
//
//    en-US: Unable to parse 'Ae9f'.
//    fr-FR: Unable to parse 'Ae9f'.
```

구문 분석과 NumberStyles 값 설명

구문 분석 작업에서 처리할 수 있는 스타일 요소(예: 공백, 그룹 구분 기호 및 소수 구분 기호)는 열거형 값으로 NumberStyles 정의됩니다. 기본적으로 정수 값을 나타내는 문자열은 숫자, 앞뒤 공백 및 앞에 오는 기호만 허용하는 NumberStyles.Integer 값을 사용하여 분석됩니다. 부동 소수점 값을 나타내는 문자열은 NumberStyles.Float 및 NumberStyles.AllowThousands 값을 결합하여 구문 분석됩니다. 이 결합 스타일은 소수 자릿수뿐만 아니라, 선행 및 후행 공백, 선행 기호, 소수 구분 기호, 그룹 구분 기호 및 지수를 허용합니다. 형식 NumberStyles 의 매개 변수를 포함하는 또는 `TryParse` 메서드의 `Parse` 오버로드를 호출하고 하나 이상의 NumberStyles 플래그를 설정하면 구문 분석 작업이 성공하기 위해 문자열에 있을 수 있는 스타일 요소를 제어할 수 있습니다.

예를 들어 그룹 구분 기호가 포함된 문자열은 메서드를 사용하여 Int32.Parse(String) 값으로 변환 Int32 할 수 없습니다. 그러나 다음 예제와 같이 플래그를 NumberStyles.AllowThousands 사용하면 변환에 성공합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string value = "1,304";
      int number;
      IFormatProvider provider = CultureInfo.CreateSpecificCulture("en-US");
      if (Int32.TryParse(value, out number))
         Console.WriteLine($"{value} --> {number}");
      else
         Console.WriteLine($"Unable to convert '{value}'");

      if (Int32.TryParse(value, NumberStyles.Integer |
NumberStyles.AllowThousands,
                         provider, out number))
         Console.WriteLine($"{value} --> {number}");
      else
         Console.WriteLine($"Unable to convert '{value}'");
   }
}
// The example displays the following output:
//       Unable to convert '1,304'
//       1,304 --> 1304
```

> ⚠ **경고**

구문 분석 작업은 항상 특정 문화권의 서식 규칙을 사용합니다. CultureInfo 또는 NumberFormatInfo 개체를 전달하여 문화권을 지정하지 않으면, 현재 스레드와 연결된 문화권이 사용됩니다.

다음 표에서는 열거형의 멤버를 NumberStyles 나열하고 구문 분석 작업에 미치는 영향을 설명합니다.

테이블 확장

NumberStyles 값	구문 분석할 문자열에 미치는 영향
NumberStyles.None	숫자 숫자만 허용됩니다.
NumberStyles.AllowDecimalPoint	소수 구분 기호와 소수 자릿수가 허용됩니다. 정수 값의 경우 소수 자릿수로 0만 허용됩니다. 유효한 10진수 구분 기호는 NumberFormatInfo.NumberDecimalSeparator 속성 또는 NumberFormatInfo.CurrencyDecimalSeparator 속성에 의해 결정됩니다.
NumberStyles.AllowExponent	"e" 또는 "E" 문자를 사용하여 지수 표기법을 나타낼 수 있습니다. 자세한 내용은 NumberStyles를 참조하세요.
NumberStyles.AllowLeadingWhite	선행 공백이 허용됩니다.
NumberStyles.AllowTrailingWhite	후행 공백이 허용됩니다.
NumberStyles.AllowLeadingSign	양수 또는 음수 기호는 숫자 앞에 올 수 있습니다.
NumberStyles.AllowTrailingSign	양수 또는 음수 기호는 숫자 숫자를 따를 수 있습니다.
NumberStyles.AllowParentheses	괄호는 음수 값을 나타내는 데 사용할 수 있습니다.
NumberStyles.AllowThousands	그룹 구분 기호가 허용됩니다. 그룹 구분 기호 문자는 NumberFormatInfo.NumberGroupSeparator 또는 NumberFormatInfo.CurrencyGroupSeparator 속성에 의해 결정됩니다.
NumberStyles.AllowCurrencySymbol	통화 기호가 허용됩니다. 통화 기호는 속성에 의해 정의됩니다 NumberFormatInfo.CurrencySymbol .
NumberStyles.AllowHexSpecifier	구문 분석할 문자열은 16진수로 해석됩니다. 16진수 0-9, A-F 및 a-f를 포함할 수 있습니다. 이 플래그는 정수 값을 구문 분석하는 데만 사용할 수 있습니다.
NumberStyles.AllowBinarySpecifier	구문 분석할 문자열은 이진 숫자로 해석됩니다. 이진 숫자 0과 1을 포함할 수 있습니다. 이 플래그는 정수 값을 구문 분석하는 데만 사용할 수 있습니다.

또한 NumberStyles 열거형은 여러 NumberStyles 플래그를 포함하는 복합 스타일을 다음과 같이 제공합니다.

테이블 확장

복합 NumberStyles 값	회원 포함
NumberStyles.Integer	NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite및 NumberStyles.AllowLeadingSign 스타일을 포함합니다. 정수 값을 구문 분석하는 데 사용되는 기본 스타일입니다.
NumberStyles.Number	NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite, NumberStyles.AllowLeadingSign, NumberStyles.AllowTrailingSignNumberStyles.AllowDecimalPoint및 NumberStyles.AllowThousands 스타일을 포함합니다.
NumberStyles.Float	NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite, NumberStyles.AllowLeadingSignNumberStyles.AllowDecimalPoint및 NumberStyles.AllowExponent 스타일을 포함합니다.
NumberStyles.Currency	를 제외한 NumberStyles.AllowExponentNumberStyles.AllowHexSpecifier모든 스타일을 포함합니다.
NumberStyles.Any	를 제외한 NumberStyles.AllowHexSpecifier모든 스타일을 포함합니다.
NumberStyles.HexNumber	NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite및 NumberStyles.AllowHexSpecifier 스타일을 포함합니다.
NumberStyles.BinaryNumber	NumberStyles.AllowLeadingWhite, NumberStyles.AllowTrailingWhite및 NumberStyles.AllowBinarySpecifier 스타일을 포함합니다.

바이너리 및 16진수 빅인티저 구문 분석

또는AllowHexSpecifier 또는 AllowBinarySpecifier 플래그로 BigInteger를 구문 분석할 때 입력 문자열은 문자열의 길이대로 정확히 16진수/이진수로 해석됩니다. 예를 들어, `"11"`를 이진 BigInteger로 구문 분석하면, 이는 정확히 2자리의 부호 있는 2의 보수 값으로 해석되는 `11`가 됩니다. 양수 결과를 원하는 경우 다음과 같이 `"011"` 구문 분석되는 선행`0`을 추가합니다`3`.

구문 분석 및 유니코드 숫자

유니코드 표준은 다양한 쓰기 시스템의 숫자 코드 포인트를 정의합니다. 예를 들어 U+0030에서 U+0039까지의 코드 포인트는 기본 라틴 숫자 0부터 9까지, U+09E6에서 U+09EF까지의 코드 포인트는 0부터 9까지의 벵골어 숫자를 나타내고 U+FF10에서 U+FF19까지의 코드 포인트는 0에서 9까지의 전체 위도 숫자를 나타냅니다. 그러나 구문 분석 메서드에서 인식되는 유일한 숫자

숫자는 U+0030에서 U+0039까지의 코드 포인트가 있는 기본 라틴 숫자 0-9입니다. 숫자 구문 분석 메서드에 다른 숫자를 포함한 문자열이 전달되면 메서드는 FormatException를 던집니다.

다음 예제에서는 메서드를 Int32.Parse 사용하여 여러 쓰기 시스템의 숫자로 구성된 문자열을 구문 분석합니다. 예제의 출력에서 알 수 있듯이 기본 라틴어 숫자를 구문 분석하는 시도는 성공하지만 전체 위도, 아랍어-인딕 및 벵골어 숫자를 구문 분석하려는 시도는 실패합니다.

C#

```
using System;

public class Example
{
   public static void Main()
   {
      string value;
      // Define a string of basic Latin digits 1-5.
      value = "\u0031\u0032\u0033\u0034\u0035";
      ParseDigits(value);

      // Define a string of Fullwidth digits 1-5.
      value = "\uFF11\uFF12\uFF13\uFF14\uFF15";
      ParseDigits(value);

      // Define a string of Arabic-Indic digits 1-5.
      value = "\u0661\u0662\u0663\u0664\u0665";
      ParseDigits(value);

      // Define a string of Bangla digits 1-5.
      value = "\u09e7\u09e8\u09e9\u09ea\u09eb";
      ParseDigits(value);
   }

   static void ParseDigits(string value)
   {
      try {
         int number = Int32.Parse(value);
         Console.WriteLine($"'{value}' --> {number}");
      }
      catch (FormatException) {
         Console.WriteLine($"Unable to parse '{value}'.");
      }
   }
}
// The example displays the following output:
//       '12345' --> 12345
//       Unable to parse '１２３４５'.
//       Unable to parse '١٢٣٤٥'.
//       Unable to parse '১২৩৪৫'.
```

참고하십시오

- NumberStyles
- 문자열 구문 분석
- 서식 유형

.NET에서 날짜 및 시간 문자열 구문 분석

.NET은 날짜 및 시간 데이터 작업을 위한 여러 형식을 제공하며, 각 유형은 서로 다른 시나리오에 최적화되어 있습니다.

- DateTime - 날짜와 시간을 함께 나타내며, 두 구성 요소가 모두 필요하거나 레거시 코드로 작업할 때 이상적입니다.
- DateOnly (.NET Framework에서는 사용할 수 없음) - 시간 정보가 없는 날짜만 나타내며 생일, 기념일 또는 비즈니스 날짜에 적합합니다.
- TimeOnly (.NET Framework에서는 사용할 수 없음) - 날짜 정보가 없는 시간만 나타내며 일정, 경보 또는 되풀이되는 일별 이벤트에 적합합니다.

각 형식은 구문 분석 프로세스에 대한 다양한 수준의 유연성과 제어를 통해 문자열을 해당 개체로 변환하는 구문 분석 메서드를 제공합니다.

일반적인 구문 분석 개념

세 날짜 및 시간 유형 모두 비슷한 구문 분석 방법을 공유합니다.

- **`Parse` 및 `TryParse` 메서드** - 현재 문화권 또는 지정된 문화권 설정을 사용하여 많은 공통 문자열 표현을 변환합니다.
- **`ParseExact` 및 `TryParseExact` 메서드** - 특정 형식 패턴을 준수하는 문자열을 변환하여 문화권 설정을 포함하여 예상 형식을 정확하게 제어합니다.
- **형식 문자열** - 표준 또는 사용자 지정 형식 지정자를 사용하여 구문 분석 패턴을 정의합니다.

문화권마다 일, 월 및 연도에 대해 서로 다른 주문을 사용합니다. 일부 시간 표현은 24시간 시계를 사용하고, 다른 시간 표현은 "AM" 및 "PM"을 지정합니다. 구문 분석 메서드는 문화권별 서식 규칙을 통해 이러한 변형을 처리합니다.

개체는 DateTimeFormatInfo 텍스트를 해석하는 방법을 제어합니다. 속성은 날짜 및 시간 구분 기호, 월 이름, 일, 연대 및 "AM" 및 "PM" 지정 형식을 설명합니다. 개체 CultureInfo 또는 개체 DateTimeFormatInfo를 사용하여 IFormatProvider 매개 변수를 통해 문화권을 지정할 수 있습니다.

형식 패턴에 대한 자세한 내용은 표준 날짜 및 시간 서식 문자열과사용자 지정 날짜 및 시간 형식 문자열을 참조하세요.

> ⓘ **Important**
>
> DateOnly와 TimeOnly 형식은 .NET Framework에서 사용할 수 없습니다.

DateTime 구문 분석

DateTime 는 날짜 및 시간 구성 요소를 함께 나타냅니다. 문자열을 `DateTime` 객체로 구문 분석할 때는 `DateTime`에 특화된 몇 가지 측면을 고려해야 합니다.

- **누락된 정보 처리** - `DateTime` 는 입력 문자열에서 파트가 누락된 경우 기본값을 사용합니다.
- **표준 시간대 및 UTC 오프셋 지원** - `DateTime` 는 로컬, UTC 또는 지정되지 않은 표준 시간대를 나타낼 수 있습니다.
- **결합된 날짜 및 시간 구문 분석** - 단일 작업에서 날짜 및 시간 구성 요소를 모두 처리해야 합니다.

누락된 정보 처리

날짜 또는 시간을 나타내는 텍스트에 일부 정보가 누락되었을 수 있습니다. 예를 들어 대부분의 사람들은 "3월 12일"이 현재 연도를 나타낸다고 가정합니다. 마찬가지로 "2018년 3월"은 2018년 3월을 나타냅니다. 시간을 나타내는 텍스트에는 종종 시간, 분 및 AM/PM 지정만 포함됩니다. `DateTime` 구문 분석 메서드는 적절한 기본값을 사용하여 누락된 정보를 처리합니다.

- 시간만 있는 경우 날짜 부분은 현재 날짜를 사용합니다.
- 날짜만 있으면 시간 부분은 자정입니다.
- 날짜에 연도를 지정하지 않으면 현재 연도가 사용됩니다.
- 월의 날짜가 지정되지 않은 경우 해당 월의 첫 번째 날짜가 사용됩니다.

문자열에 날짜가 있는 경우 월과 일 또는 연도 중 하나를 포함해야 합니다. 시간이 있는 경우 시간 및 분 또는 AM/PM 지정자를 포함해야 합니다.

NoCurrentDateDefault 상수를 지정하여 이러한 기본값을 재정의할 수 있습니다. 해당 상수 사용 시 누락된 연도, 월 또는 일 속성은 `1`값으로 설정됩니다. 사용하는 Parse 이 동작을 보여 줍니다.

UTC 오프셋 및 표준 시간대 처리

날짜 및 시간 구성 요소 외에도 날짜 및 시간의 문자열 표현에는 시간이 UTC(협정 세계시)와 얼마나 다른지 나타내는 오프셋이 포함될 수 있습니다. 예를 들어 "2007년 2월 14일 5:32:00 -7:00" 문자열은 UTC보다 7시간 이전의 시간을 정의합니다. 시간의 문자열 표현에서 오프셋이 생략되면 구문 분석 결과로 DateTime 속성이 Kind로 설정된 DateTimeKind.Unspecified 개체가 반환됩니다. 오프셋이 지정되면, 구문 분석 시 DateTime 속성이 Kind로 설정된 DateTimeKind.Local 객체가 반환됩니다. 또한 해당 값은 머신의 현지 표준 시간대로 조정됩니다. 구문 분석 메서드와 함께 DateTimeStyles 값을 사용하여 이 동작을 수정할 수 있습니다.

모호한 날짜 처리

형식 공급자는 모호한 숫자 날짜를 해석하는 데도 사용됩니다. "02/03/04" 문자열이 나타내는 날짜의 구성 요소가 월, 일 및 연도인지는 불분명합니다. 구성 요소는 형식 공급자에서 유사한 날짜 형식의 순서에 따라 해석됩니다.

DateTime.Parse

다음 예제에서는 DateTime.Parse 메서드를 사용하여 `string`을 DateTime으로 변환하는 방법을 보여줍니다. 이 예제에서는 현재 스레드와 연결된 문화권을 사용합니다. 현재 문화권과 연결된 CultureInfo이 입력 문자열을 구문 분석할 수 없는 경우, FormatException 예외가 발생합니다.

C#

```
static void DateTimeParseExample()
{
    // Parse common date and time formats using current culture
    var dateTime1 = DateTime.Parse("1/15/2025 3:30 PM");
    var dateTime2 = DateTime.Parse("January 15, 2025");
    var dateTime3 = DateTime.Parse("15:30:45");

    Console.WriteLine($"Parsed: {dateTime1}");
    Console.WriteLine($"Parsed: {dateTime2}");
    Console.WriteLine($"Parsed: {dateTime3}");

    // Parse with specific culture
    var germanDate = DateTime.Parse("15.01.2025", new CultureInfo("de-DE"));
    Console.WriteLine($"German date parsed: {germanDate}");
}
```

문자열을 구문 분석할 때 형식 지정 규칙이 사용되는 문화권을 명시적으로 정의할 수도 있습니다. DateTimeFormatInfo 속성에서 반환된 표준 CultureInfo.DateTimeFormat 개체 중 하나를 지정합니다. 다음 예제에서는 형식 공급자를 사용하여 독일어 문자열을 특정 형식의 코드인 'DateTime'으로 구문 분석합니다. CultureInfo 문화를 대표하는 `de-DE`를 만듭니다. 이 `CultureInfo` 개체를 사용하면 이 특정 문자열을 성공적으로 구문 분석할 수 있습니다. 이 프로세스는 CurrentCulture의 CurrentThread에 있는 설정을 차단합니다.

C#

```
static void DateTimeParseGermanExample()
{
    var cultureInfo = new CultureInfo("de-DE");
    string dateString = "12 Juni 2008";
    var dateTime = DateTime.Parse(dateString, cultureInfo);
    Console.WriteLine(dateTime);
    // The example displays the following output:
    //       6/12/2008 00:00:00
}
```

그러나 Parse 메서드의 오버로드를 사용하여 사용자 지정 형식 공급자를 지정할 수 있습니다. Parse 메서드는 비표준 형식 구문 분석을 지원하지 않습니다. 비표준 형식으로 표현된 날짜 및 시간을 구문 분석하려면 대신 ParseExact 메서드를 사용합니다.

다음 예제에서는 DateTimeStyles 열거형을 사용하여 지정되지 않은 필드에 대한 현재 날짜 및 시간 정보를 DateTime 추가하지 않도록 지정합니다.

C#

```
static void DateTimeParseNoDefaultExample()
{
    var cultureInfo = new CultureInfo("de-DE");
    string dateString = "12 Juni 2008";
    var dateTime = DateTime.Parse(dateString, cultureInfo,
                                  DateTimeStyles.NoCurrentDateDefault);
    Console.WriteLine(dateTime);
    // The example displays the following output if the current culture is en-US:
    //       6/12/2008 00:00:00
}
```

DateTime.ParseExact

DateTime.ParseExact 메서드는 지정된 문자열 패턴 중 하나를 준수하는 경우 문자열을 DateTime 개체로 변환합니다. 지정된 양식 중 하나가 아닌 문자열이 이 메서드에 전달되면 FormatException throw됩니다. 표준 날짜 및 시간 형식 지정자 또는 사용자 지정 형식 지정자의 조합을 지정할 수 있습니다. 사용자 지정 형식 지정자를 사용하여 사용자 지정 인식 문자열을 생성할 수 있습니다. 지정자에 대한 설명은 표준 날짜 및 시간 서식 문자열 및 사용자 지정 날짜 및 시간 형식 문자열문서를 참조하세요.

다음 예제에서 DateTime.ParseExact 메서드에는 먼저 구문 분석할 문자열 개체가 전달되고, 그 뒤에 형식 지정자와 CultureInfo 개체가 전달됩니다. 이 ParseExact 메서드는 `en-US` 문화권에서 긴 날짜 패턴을 따르는 문자열만 구문 분석할 수 있습니다.

C#

```
static void DateTimeParseExactExample()
{
    // Parse exact format
    var exactDate = DateTime.ParseExact("2025-01-15T14:30:00", "yyyy-MM-
ddTHH:mm:ss", CultureInfo.InvariantCulture);
    Console.WriteLine($"Exact parse: {exactDate}");

    // Parse with custom format
    var customDate = DateTime.ParseExact("15/Jan/2025 2:30 PM", "dd/MMM/yyyy h:mm
tt", CultureInfo.InvariantCulture);
    Console.WriteLine($"Custom format: {customDate}");
}
```

Parse 및 ParseExact 메서드의 각 오버로드에는 문자열의 서식 지정에 대한 문화권별 정보를 제공하는 IFormatProvider 매개 변수도 있습니다. IFormatProvider 개체는 표준 문화권을 나타내는 CultureInfo 개체이거나, CultureInfo.DateTimeFormat 속성에 의해 반환되는 DateTimeFormatInfo 개체입니다. ParseExact 하나 이상의 사용자 지정 날짜 및 시간 형식을 정의하는 추가 문자열 또는 문자열 배열 인수도 사용합니다.

DateOnly 구문 분석

이 구조는 DateOnly 시간 정보가 없는 날짜만 나타내므로 생일, 기념일 또는 비즈니스 날짜와 같은 시나리오에 적합합니다. 시간 구성 요소가 없으므로 하루의 시작부터 종료까지의 날짜를 나타냅니다.

`DateOnly` 에는 날짜 전용 시나리오에 사용하는 `DateTime` 것에 비해 몇 가지 이점이 있습니다.

- `DateTime` 구조가 시간대 조정에 따라 이전 날 또는 다음 날로 넘어갈 수 있습니다. `DateOnly` 는 표준 시간대에 의해 오프셋될 수 없으며 항상 설정된 날짜를 나타냅니다.
- 직렬화에는 `DateOnly`가 `DateTime`보다 적은 데이터가 포함됩니다.
- 코드가 SQL Server와 같은 데이터베이스와 상호 작용하는 경우 전체 날짜는 일반적으로 시간을 포함하지 않는 데이터 형식으로 `date` 저장됩니다. `DateOnly` 는 데이터베이스 형식과 더 잘 일치합니다.

DateOnly.Parse

이 메서드는 DateOnly.Parse 공통 날짜 문자열 표현을 개체로 DateOnly 변환합니다. 메서드는 다양한 형식을 허용하고 구문 분석에 현재 문화권 또는 지정된 문화권을 사용합니다.

C#

```
static void DateOnlyParseExample()
{
    // Parse common date formats
    var date1 = DateOnly.Parse("1/15/2025");
    var date2 = DateOnly.Parse("January 15, 2025", CultureInfo.InvariantCulture);
    var date3 = DateOnly.Parse("2025-01-15");

    Console.WriteLine($"Parsed date: {date1}");
    Console.WriteLine($"Parsed date: {date2.ToString("D")}"); // Long date format
    Console.WriteLine($"Parsed date: {date3.ToString("yyyy-MM-dd")}");

    // Parse with specific culture
    var germanDate = DateOnly.Parse("15.01.2025", new CultureInfo("de-DE"));
    Console.WriteLine($"German date: {germanDate}");
}
```

DateOnly.ParseExact

이 메서드는 DateOnly.ParseExact 입력 문자열의 예상 형식을 정확하게 제어합니다. 날짜 문자열의 정확한 형식을 알고 엄격한 구문 분석을 보장하려는 경우 이 메서드를 사용합니다.

C#

```
static void DateOnlyParseExactExample()
{
    // Parse exact format
    var exactDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture);
    Console.WriteLine($"Exact date: {exactDate}");

    // Parse ISO format
    var isoDate = DateOnly.ParseExact("2025-01-15", "yyyy-MM-dd",
CultureInfo.InvariantCulture);
    Console.WriteLine($"ISO date: {isoDate}");

    // Parse with multiple possible formats
    string[] formats = { "MM/dd/yyyy", "M/d/yyyy", "dd/MM/yyyy" };
    var flexibleDate = DateOnly.ParseExact("1/15/2025", formats,
CultureInfo.InvariantCulture, DateTimeStyles.None);
    Console.WriteLine($"Flexible parse: {flexibleDate}");
}
```

이 메서드는 `ParseExact` 단일 형식 문자열 또는 서식 문자열 배열을 허용하므로 허용 가능한 여러 형식으로 제공되는 날짜를 구문 분석할 수 있습니다.

TimeOnly 구문 분석

구조는 TimeOnly 매일 알람 시계 또는 매일 점심을 먹는 시간과 같은 하루 중 시간 값을 나타냅니다. `TimeOnly` 는 특정 시간인 **00:00:00.00000000 - 23:59:59.99999999** 범위로 제한됩니다.

`TimeOnly` 는 시간 전용 시나리오에 다른 형식을 사용할 때 존재했던 몇 가지 문제를 해결합니다.

- `TimeSpan` 는 경과된 시간을 나타내며 음수이거나 24시간을 초과할 수 있으므로 특정 시간을 나타내는 데 적합하지 않습니다.
- 하루 중 시간을 사용 `DateTime` 하려면 임의의 날짜가 필요하며 계산을 수행할 때 예기치 않은 동작이 발생할 수 있습니다.
- `TimeOnly` 시간 값을 추가하거나 빼는 경우 24시간 롤오버를 자연스럽게 처리합니다.

TimeOnly.Parse

이 메서드는 TimeOnly.Parse 공용 시간 문자열 표현을 개체로 TimeOnly 변환합니다. 이 메서드는 12시간 및 24시간 표기법을 비롯한 다양한 형식을 허용합니다.

C#

```
static void TimeOnlyParseExample()
{
    // Parse common time formats
    var time1 = TimeOnly.Parse("14:30:15");
    var time2 = TimeOnly.Parse("2:30 PM", CultureInfo.InvariantCulture);
    var time3 = TimeOnly.Parse("17:45");

    Console.WriteLine($"Parsed time: {time1}");
    Console.WriteLine($"Parsed time: {time2.ToString("t")}"); // Short time format
    Console.WriteLine($"Parsed time: {time3.ToString("HH:mm")}");

    // Parse with milliseconds
    var preciseTime = TimeOnly.Parse("14:30:15.123");
    Console.WriteLine($"Precise time: {preciseTime.ToString("HH:mm:ss.fff")}");
}
```

TimeOnly.ParseExact

이 메서드는 TimeOnly.ParseExact 입력 시간 문자열의 예상 형식을 정확하게 제어합니다. 정확한 형식을 알고 엄격한 구문 분석을 보장하려는 경우 이 메서드를 사용합니다.

C#

```
static void TimeOnlyParseExactExample()
{
    // Parse exact format
    var exactTime = TimeOnly.ParseExact("5:00 pm", "h:mm tt",
CultureInfo.InvariantCulture);
    Console.WriteLine($"Exact time: {exactTime}");

    // Parse 24-hour format
    var militaryTime = TimeOnly.ParseExact("17:30:25", "HH:mm:ss",
CultureInfo.InvariantCulture);
    Console.WriteLine($"Military time: {militaryTime}");

    // Parse with multiple possible formats
    string[] timeFormats = { "h:mm tt", "HH:mm", "H:mm" };
    var flexibleTime = TimeOnly.ParseExact("2:30 PM", timeFormats,
CultureInfo.InvariantCulture, DateTimeStyles.None);
    Console.WriteLine($"Flexible time parse: {flexibleTime}");
}
```

참고하십시오

- 문자열 구문 분석
- 형식 유형
- .NET에서의 형식 변환

- 표준 날짜 및 시간 형식
- 사용자 지정 날짜 및 시간 형식 문자열
- DateOnly 및 TimeOnly 구조를 사용하는 방법

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 22.

.NET에서 다른 문자열 구문 분석

아티클 • 2025. 05. 04.

숫자 및 DateTime 문자열 외에도 형식을 나타내는 문자열을 데이터 형식 CharBooleanEnum 으로 구문 분석할 수도 있습니다.

Char(문자)

Char 데이터 형식과 연결된 정적 구문 분석 메서드는 단일 문자가 포함된 문자열을 유니코드 값으로 변환하는 데 유용합니다. 다음 코드 예제에서는 문자열을 유니코드 문자로 구문 분석합니다.

C#

```
string MyString1 = "A";
char MyChar = Char.Parse(MyString1);
// MyChar now contains a Unicode "A" character.
```

불리언 (Boolean)

부울 데이터 형식에는 **부울** 값을 나타내는 문자열을 실제 **부울** 형식으로 변환하는 데 사용할 수 있는 **Parse** 메서드가 포함되어 있습니다. 이 메서드는 대/소문자를 구분하지 않으며 "True" 또는 "False"를 포함하는 문자열을 성공적으로 구문 분석할 수 있습니다. **부울** 형식과 연결된 **구문 분석** 메서드는 공백으로 둘러싸인 문자열을 구문 분석할 수도 있습니다. 다른 문자열이 전달되면 FormatException가 발생합니다.

다음 코드 예제에서는 **Parse** 메서드를 사용하여 문자열을 부울 값으로 변환합니다.

C#

```
string MyString2 = "True";
bool MyBool = bool.Parse(MyString2);
// MyBool now contains a True Boolean value.
```

열거

정적 **Parse** 메서드를 사용하여 열거형 형식을 문자열 값으로 초기화할 수 있습니다. 이 메서드는 구문 분석할 열거형 형식, 구문 분석할 문자열 및 구문 분석이 대/소문자를 구분하는지 여부를 나타내는 선택적 부울 플래그를 허용합니다. 구문 분석할 문자열에는 쉼표로 구분된 여러 값이 포함될 수 있으며, 앞에 오거나 뒤에 하나 이상의 빈 공백(공백이라고도 함)이 뒤따를 수 있습니

다. 문자열에 여러 값이 포함된 경우 반환된 개체의 값은 비트 OR 연산과 결합된 지정된 모든 값의 값입니다.

다음 예제에서는 **Parse** 메서드를 사용하여 문자열 표현을 열거형 값으로 변환합니다. DayOfWeek 열거형은 문자열에서 **목요일**로 초기화됩니다.

C#

```
string MyString3 = "Thursday";
DayOfWeek MyDays = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), MyString3);
Console.WriteLine(MyDays);
// The result is Thursday.
```

참고하십시오

- 문자열 구문 분석
- 서식 유형
- .NET의 형식 변환

System.String 클래스

아티클 • 2024. 01. 08.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

문자열은 텍스트를 나타내는 데 사용되는 문자의 순차적 컬렉션입니다. String 개체는 문자열을 나타내는 개체의 System.Char 순차적 컬렉션이며, System.Char 개체는 UTF-16 코드 단위에 해당합니다. 개체의 String 값은 개체의 순차 컬렉션 내용 System.Char 이며 해당 값은 변경할 수 없습니다(즉, 읽기 전용). 문자열의 불변성에 대한 자세한 내용은 불변성 및 StringBuilder 클래스 섹션을 참조하세요. 메모리에 있는 개체의 String 최대 크기는 2GB 또는 약 10억 자입니다.

유니코드, UTF-16, 코드 단위, 코드 포인트 및 CharRune 형식에 대한 자세한 내용은 .NET의 문자 인코딩 소개를 참조하세요.

String 개체 인스턴스화

다음과 같은 방법으로 개체를 String 인스턴스화할 수 있습니다.

- 변수에 문자열 리터럴을 String 할당합니다. 문자열을 만드는 데 가장 일반적으로 사용되는 메서드입니다. 다음 예제에서는 할당을 사용하여 여러 문자열을 만듭니다. C# 및 F#에서는 백슬래시(\)가 이스케이프 문자이기 때문에 문자열의 리터럴 백슬래시를 이스케이프하거나 전체 문자열을 @-quoted해야 합니다.

  C#

  ```
  string string1 = "This is a string created by assignment.";
  Console.WriteLine(string1);
  string string2a = "The path is C:\\PublicDocuments\\Report1.doc";
  Console.WriteLine(string2a);
  string string2b = @"The path is C:\PublicDocuments\Report1.doc";
  Console.WriteLine(string2b);
  // The example displays the following output:
  //       This is a string created by assignment.
  //       The path is C:\PublicDocuments\Report1.doc
  //       The path is C:\PublicDocuments\Report1.doc
  ```

- 클래스 생성자를 호출합니다 String . 다음 예제에서는 여러 클래스 생성자를 호출하여 문자열을 인스턴스화합니다. 일부 생성자에는 문자 배열에 대한 포인터 또는 서명된 바이트 배열이 매개 변수로 포함됩니다. Visual Basic은 이러한 생성자에 대한 호출을 지원하지 않습니다. 생성자에 대한 String 자세한 내용은 생성자 요약을 String 참조하세요.

C#

```
char[] chars = { 'w', 'o', 'r', 'd' };
sbyte[] bytes = { 0x41, 0x42, 0x43, 0x44, 0x45, 0x00 };

// Create a string from a character array.
string string1 = new string(chars);
Console.WriteLine(string1);

// Create a string that consists of a character repeated 20 times.
string string2 = new string('c', 20);
Console.WriteLine(string2);

string stringFromBytes = null;
string stringFromChars = null;
unsafe
{
   fixed (sbyte* pbytes = bytes)
   {
      // Create a string from a pointer to a signed byte array.
      stringFromBytes = new string(pbytes);
   }
   fixed (char* pchars = chars)
   {
      // Create a string from a pointer to a character array.
      stringFromChars = new string(pchars);
   }
}
Console.WriteLine(stringFromBytes);
Console.WriteLine(stringFromChars);
// The example displays the following output:
//       word
//       cccccccccccccccccccc
//       ABCDE
//       word
```

- 문자열 연결 연산자(C# 및 F#의 경우+ 및 Visual Basic의 경우 +)를 사용하여 인스턴스와 문자열 리터럴의 String 조합에서 단일 문자열을 만듭니다. 다음 예제에서는 문자열 연결 연산자의 사용을 보여 줍니다.

C#

```
string string1 = "Today is " + DateTime.Now.ToString("D") + ".";
Console.WriteLine(string1);

string string2 = "This is one sentence. " + "This is a second. ";
string2 += "This is a third sentence.";
Console.WriteLine(string2);
// The example displays output like the following:
//    Today is Tuesday, July 06, 2011.
//    This is one sentence. This is a second. This is a third sentence.
```

- 속성을 검색하거나 문자열을 반환하는 메서드를 호출합니다. 다음 예제에서는 클래스의 메서드를 String 사용하여 더 큰 문자열에서 부분 문자열을 추출합니다.

```
C#
```

```
string sentence = "This sentence has five words.";
// Extract the second word.
int startPosition = sentence.IndexOf(" ") + 1;
string word2 = sentence.Substring(startPosition,
                                  sentence.IndexOf(" ", startPosition)
- startPosition);
Console.WriteLine("Second word: " + word2);
// The example displays the following output:
//       Second word: sentence
```

- 형식 지정 메서드를 호출하여 값 또는 개체를 문자열 표현으로 변환합니다. 다음 예제에서는 복합 서식 지정 기능을 사용하여 두 개체의 문자열 표현을 문자열에 포함합니다.

```
C#
```

```
DateTime dateAndTime = new DateTime(2011, 7, 6, 7, 32, 0);
double temperature = 68.3;
string result = String.Format("At {0:t} on {0:D}, the temperature was
{1:F1} degrees Fahrenheit.",
                              dateAndTime, temperature);
Console.WriteLine(result);
// The example displays the following output:
//       At 7:32 AM on Wednesday, July 06, 2011, the temperature was
68.3 degrees Fahrenheit.
```

Char 개체 및 유니코드 문자

문자열의 각 문자는 유니코드 코드 포인트 또는 유니코드 문자의 서수(숫자) 값이라고도 하는 유니코드 스칼라 값으로 정의됩니다. 각 코드 포인트는 UTF-16 인코딩을 사용하여 인코딩되고 인코딩의 각 요소의 숫자 값은 개체로 Char 표시됩니다.

> **ⓘ 참고**
>
> 인스턴스는 String UTF-16 코드 단위의 순차 컬렉션으로 구성되므로 올바른 형식의 유니코드 문자열이 아닌 개체를 만들 String 수 있습니다. 예를 들어 해당 상위 서로게이트 없이 서로게이트가 낮은 문자열을 만들 수 있습니다. 네임스페이스에서 개체 System.Text 를 인코딩 및 디코딩하는 메서드와 같은 일부 메서드는 문자열이 올바

른 형식인지 확인하기 위해 검사 수행할 수 있지만 클래스 String 멤버는 문자열이 올바른 형식인지를 보장하지 않습니다.

단일 Char 개체는 일반적으로 단일 코드 포인트를 나타냅니다. 즉, 숫자 값 Char 은 코드 포인트와 같습니다. 예를 들어 문자 "a"의 코드 포인트는 U+0061입니다. 그러나 코드 포인트에는 둘 이상의 인코딩된 요소(둘 Char 이상의 개체)가 필요할 수 있습니다. 유니코드 표준은 여러 Char 개체에 해당하는 두 가지 유형의 문자를 정의합니다. 즉, 그래프와 유니코드 보조 평면의 문자에 해당하는 유니코드 보조 코드 포인트입니다.

- 그래프는 기본 문자 뒤에 하나 이상의 결합 문자로 표시됩니다. 예를 들어 ä 문자는 코드 포인트가 U+0061인 개체와 Char 코드 포인트가 U+0308인 개체로 표시됩니다 Char. 이 문자는 코드 포인트가 U+00E4인 단일 Char 개체로 정의할 수도 있습니다. 다음 예제와 같이 같음의 문화권 구분 비교는 일반 서수 비교는 그렇지 않지만 이러한 두 표현이 같음을 나타냅니다. 그러나 두 문자열이 정규화된 경우 서수 비교는 해당 문자열이 같음도 나타냅니다. (문자열 정규화에 대한 자세한 내용은 다음을 참조하세요 .정규화 섹션.)

```
C#
```

```
using System;
using System.Globalization;
using System.IO;

public class Example5
{
   public static void Main()
   {
      StreamWriter sw = new StreamWriter(@".\graphemes.txt");
      string grapheme = "\u0061\u0308";
      sw.WriteLine(grapheme);

      string singleChar = "\u00e4";
      sw.WriteLine(singleChar);

      sw.WriteLine("{0} = {1} (Culture-sensitive): {2}", grapheme,
singleChar,
                   String.Equals(grapheme, singleChar,
                                 StringComparison.CurrentCulture));
      sw.WriteLine("{0} = {1} (Ordinal): {2}", grapheme, singleChar,
                   String.Equals(grapheme, singleChar,
                                 StringComparison.Ordinal));
      sw.WriteLine("{0} = {1} (Normalized Ordinal): {2}", grapheme,
singleChar,
                   String.Equals(grapheme.Normalize(),
                                 singleChar.Normalize(),
                                 StringComparison.Ordinal));
      sw.Close();
   }
}
```

```
// The example produces the following output:
//       ä
//       ä
//       ä = ä (Culture-sensitive): True
//       ä = ä (Ordinal): False
//       ä = ä (Normalized Ordinal): True
```

- 유니코드 보조 코드 지점(서로게이트 쌍)은 코드 포인트가 높은 서로게이트인 개체와 Char 코드 포인트가 낮은 서로게이트인 개체로 표시됩니다Char. 상위 서로게이트의 코드 단위는 U+D800에서 U+DBFF까지 다양합니다. 낮은 서로게이트의 코드 단위는 U+DC00에서 U+DFFF까지 다양합니다. 서로게이트 쌍은 16개의 유니코드 보조 평면의 문자를 나타내는 데 사용됩니다. 다음 예제에서는 서로게이트 문자를 만들고 메서드에 Char.IsSurrogatePair(Char, Char) 전달하여 서로게이트 쌍인지 여부를 확인합니다.

C#

```
string surrogate = "\uD800\uDC03";
for (int ctr = 0; ctr < surrogate.Length; ctr++)
   Console.Write($"U+{(ushort)surrogate[ctr]:X2} ");

Console.WriteLine();
Console.WriteLine("   Is Surrogate Pair: {0}",
                  Char.IsSurrogatePair(surrogate[0], surrogate[1]));
// The example displays the following output:
//       U+D800 U+DC03
//          Is Surrogate Pair: True
```

유니코드 표준

문자열의 문자는 값에 해당하는 UTF-16으로 인코딩된 코드 단위로 Char 표시됩니다.

문자열의 각 문자에는 열거형으로 .NET에 표시되는 연결된 유니코드 문자 범주가 UnicodeCategory 있습니다. 문자 또는 서로게이트 쌍의 범주는 메서드를 호출 CharUnicodeInfo.GetUnicodeCategory 하여 확인할 수 있습니다.

.NET에서는 문자 및 해당 범주의 고유한 테이블을 유지 관리합니다. 그러면 다른 플랫폼에서 실행되는 특정 버전의 .NET 구현이 동일한 문자 범주 정보를 반환하게 됩니다. 모든 .NET 버전 및 모든 OS 플랫폼에서 문자 범주 정보는 유니코드 문자 데이터베이스에서 제공됩니다.

다음 표에서는 해당 문자 범주의 기반이 되는 .NET 버전 및 유니코드 표준 버전을 나열합니다.

테이블 확장

.NET 버전	유니코드 표준 버전
.NET Framework 1.1	유니코드 표준, 버전 4.0.0
.NET Framework 2.0	유니코드 표준, 버전 5.0.0
.NET Framework 3.5	유니코드 표준, 버전 5.0.0
.NET Framework 4	유니코드 표준, 버전 5.0.0
.NET Framework 4.5	유니코드 표준, 버전 6.3.0
.NET Framework 4.5.1	유니코드 표준, 버전 6.3.0
.NET Framework 4.5.2	유니코드 표준, 버전 6.3.0
.NET Framework 4.6	유니코드 표준, 버전 6.3.0
.NET Framework 4.6.1	유니코드 표준, 버전 6.3.0
.NET Framework 4.6.2 이전 버전	유니코드 표준, 버전 8.0.0
.NET Core 2.1	유니코드 표준, 버전 8.0.0
.NET Core 3.1	유니코드 표준 버전 11.0.0
.NET 5	유니코드 표준 버전 13.0.0

또한 .NET은 유니코드 표준에 따라 문자열 비교 및 정렬을 지원합니다. .NET Framework 4 및 이전 버전은 고유한 문자열 데이터 테이블을 기본. Windows 7에서 실행되는 .NET Framework 4.5부터 시작하는 .NET Framework 버전도 마찬가지입니다. Windows 8 이상 버전의 Windows 운영 체제에서 실행되는 .NET Framework 4.5부터 런타임은 문자열 비교 및 정렬 작업을 운영 체제에 위임합니다. .NET Core 및 .NET 5 이상에서는 유니코드 라이브러리용 International Components(Windows 10 2019년 5월 업데이트 이전 Windows 버전 제외)에서 문자열 비교 및 정렬 정보를 제공합니다. 다음 표에서는 .NET 버전과 문자 비교 및 정렬의 기반이 되는 유니코드 표준 버전을 나열합니다.

테이블 확장

.NET 버전	유니코드 표준 버전
.NET Framework 1.1	유니코드 표준, 버전 4.0.0
.NET Framework 2.0	유니코드 표준, 버전 5.0.0
.NET Framework 3.5	유니코드 표준, 버전 5.0.0
.NET Framework 4	유니코드 표준, 버전 5.0.0

.NET 버전	유니코드 표준 버전
Windows 7의 .NET Framework 4.5 이상	유니코드 표준, 버전 5.0.0
Windows 8 이상 Windows 운영 체제의 .NET Framework 4.5 이상	유니코드 표준, 버전 6.3.0
.NET Core 및 .NET 5 이상	기본 운영 체제에서 지원되는 유니코드 표준의 버전에 따라 달라집니다.

포함된 null 문자

.NET에서 개체는 String 문자열 길이의 일부로 계산되는 포함된 null 문자를 포함할 수 있습니다. 그러나 C 및 C++와 같은 일부 언어에서는 null 문자가 문자열의 끝을 나타냅니다. 문자열의 일부로 간주되지 않으며 문자열 길이의 일부로 계산되지 않습니다. 즉, C 및 C++ 프로그래머 또는 C 또는 C++로 작성된 라이브러리가 문자열에 대해 만들 수 있는 다음과 같은 일반적인 가정은 개체에 적용할 String 때 반드시 유효하지는 않습니다.

- 또는 `wcslen` 함수에서 반환된 값이 `strlen` 반드시 같은 String.Length것은 아닙니다.
- 또는 `wcscpy_s` 함수에서 `strcpy_s` 만든 문자열이 메서드에서 만든 String.Copy 문자열과 반드시 동일하지는 않습니다.

개체를 인스턴스화하는 String 네이티브 C 및 C++ 코드와 플랫폼 호출을 통해 전달된 String 개체가 포함된 null 문자가 문자열의 끝을 표시한다고 가정하지 않도록 해야 합니다.

문자열에 포함된 null 문자는 문자열을 정렬(또는 비교)하고 문자열을 검색할 때도 다르게 처리됩니다. 고정 문화권을 사용한 비교를 포함하여 두 문자열 간에 문화권 구분 비교를 수행하는 경우 Null 문자는 무시됩니다. 서수 또는 대/소문자를 구분하지 않는 서수 비교에 대해서만 고려됩니다. 반면에 포함된 null 문자는 메서드(예 Contains: , StartsWith및 IndexOf)를 사용하여 문자열을 검색할 때 항상 고려됩니다.

문자열 및 인덱스

인덱스는 .에서 유니코드 문자가 아닌 개체의 Char 위치입니다 String. 인덱스는 문자열의 첫 번째 위치(인덱스 위치 0)에서 시작하는 0부터 시작하는 음수입니다. 문자열 인스턴스에서 문자 또는 부분 문자열의 인덱스와 같은 IndexOfLastIndexOf다양한 검색 메서드를 반환합니다.

이 Chars[] 속성을 사용하면 문자열의 인덱스 위치로 개별 Char 개체에 액세스할 수 있습니다. Chars[] 속성은 기본 속성(Visual Basic) 또는 인덱서(C# 및 F#)이므로 다음과 같은 코

드를 사용하여 문자열의 개별 Char 개체에 액세스할 수 있습니다. 이 코드는 문자열에서 공백 또는 문장 부호 문자를 검색하여 문자열에 포함된 단어 수를 결정합니다.

C#

```
string s1 = "This string consists of a single short sentence.";
int nWords = 0;

s1 = s1.Trim();
for (int ctr = 0; ctr < s1.Length; ctr++) {
   if (Char.IsPunctuation(s1[ctr]) | Char.IsWhiteSpace(s1[ctr]))
      nWords++;
}
Console.WriteLine("The sentence\n   {0}\nhas {1} words.",
                  s1, nWords);
// The example displays the following output:
//       The sentence
//          This string consists of a single short sentence.
//       has 8 words.
```

클래스는 String 인터페이스를 IEnumerable 구현하기 때문에 다음 예제와 같이 구문을 사용하여 `foreach` 문자열의 개체를 반복할 Char 수도 있습니다.

C#

```
string s1 = "This string consists of a single short sentence.";
int nWords = 0;

s1 = s1.Trim();
foreach (var ch in s1) {
   if (Char.IsPunctuation(ch) | Char.IsWhiteSpace(ch))
      nWords++;
}
Console.WriteLine("The sentence\n   {0}\nhas {1} words.",
                  s1, nWords);
// The example displays the following output:
//       The sentence
//          This string consists of a single short sentence.
//       has 8 words.
```

유니코드 문자가 둘 Char 이상의 개체로 인코딩될 수 있으므로 연속 인덱스 값은 연속 유니코드 문자에 해당하지 않을 수 있습니다. 특히 문자열에는 기본 문자 다음에 하나 이상의 결합 문자 또는 서로게이트 쌍에 의해 형성되는 텍스트의 여러 문자 단위가 포함될 수 있습니다. 개체 대신 Char 유니코드 문자를 사용하려면 및 TextElementEnumerator 클래스 또는 String.EnumerateRunes 메서드와 구조체를 Rune 사용합니다 System.Globalization.StringInfo. 다음 예제에서는 개체와 함께 작동하는 코드와 유니코드 문자로 Char 작동하는 코드의 차이점을 보여 줍니다. 문장의 각 단어에 있는 문자 또는 텍

스트 요소의 수를 비교합니다. 문자열에는 기본 문자의 두 시퀀스 뒤에 결합 문자가 포함됩니다.

C#

```
// First sentence of The Mystery of the Yellow Room, by Leroux.
string opening = "Ce n'est pas sans une certaine émotion que "+
                 "je commence à raconter ici les aventures " +
                 "extraordinaires de Joseph Rouletabille.";
// Character counters.
int nChars = 0;
// Objects to store word count.
List<int> chars = new List<int>();
List<int> elements = new List<int>();

foreach (var ch in opening) {
   // Skip the ' character.
   if (ch == '\u0027') continue;

   if (Char.IsWhiteSpace(ch) | (Char.IsPunctuation(ch))) {
      chars.Add(nChars);
      nChars = 0;
   }
   else {
      nChars++;
   }
}

System.Globalization.TextElementEnumerator te =
   System.Globalization.StringInfo.GetTextElementEnumerator(opening);
while (te.MoveNext()) {
   string s = te.GetTextElement();
   // Skip the ' character.
   if (s == "\u0027") continue;
   if ( String.IsNullOrEmpty(s.Trim()) | (s.Length == 1 &&
Char.IsPunctuation(Convert.ToChar(s)))) {
      elements.Add(nChars);
      nChars = 0;
   }
   else {
      nChars++;
   }
}

// Display character counts.
Console.WriteLine("{0,6} {1,20} {2,20}",
                  "Word #", "Char Objects", "Characters");
for (int ctr = 0; ctr < chars.Count; ctr++)
   Console.WriteLine("{0,6} {1,20} {2,20}",
                     ctr, chars[ctr], elements[ctr]);
// The example displays the following output:
//       Word #         Char Objects           Characters
//            0                    2                    2
//            1                    4                    4
```

```
//          2                3                3
//          3                4                4
//          4                3                3
//          5                8                8
//          6                8                7
//          7                3                3
//          8                2                2
//          9                8                8
//         10                2                1
//         11                8                8
//         12                3                3
//         13                3                3
//         14                9                9
//         15               15               15
//         16                2                2
//         17                6                6
//         18               12               12
```

이 예제에서는 메서드와 클래스를 사용하여 문자열의 StringInfo.GetTextElementEnumeratorTextElementEnumerator 모든 텍스트 요소를 열거하여 텍스트 요소에서 작동합니다. 메서드를 호출하여 각 텍스트 요소의 시작 인덱스가 포함된 배열을 검색할 StringInfo.ParseCombiningCharacters 수도 있습니다.

개별 Char 값이 아닌 텍스트 단위를 사용하는 방법에 대한 자세한 내용은 .NET의 문자 인코딩 소개를 참조하세요.

Null 문자열 및 빈 문자열

선언되었지만 값이 할당되지 않은 문자열은 다음과 입니다 `null`. 해당 문자열에서 메서드를 호출하려고 시도하면 .NullReferenceException null 문자열은 값이 "" 또는 String.Empty"인 문자열인 빈 문자열과 다릅니다. 경우에 따라 null 문자열 또는 빈 문자열을 메서드 호출의 인수로 전달하면 예외가 throw됩니다. 예를 들어 메서드 ArgumentNullException에 null 문자열을 전달하면 Int32.Parse 빈 문자열이 throw되고 빈 문자열을 전달하면 .FormatException 다른 경우에는 메서드 인수가 null 문자열이거나 빈 문자열일 수 있습니다. 예를 들어 클래스에 대한 구현을 IFormattable 제공하는 경우 null 문자열과 빈 문자열을 모두 일반("G") 형식 지정자와 동일시하려고 합니다.

이 String 클래스에는 문자열이 비어 있는지 여부를 테스트할 수 있는 다음과 같은 두 가지 편리한 메서드가 포함되어 있습니다 `null` .

- IsNullOrEmpty문자열 `null` 이 같거나 같은지 여부를 나타내는 String.Empty입니다. 이 메서드는 다음과 같은 코드를 사용할 필요가 없습니다.

  ```
  C#
  ```

```
if (str == null || str.Equals(String.Empty))
```

- IsNullOrWhiteSpace문자열이 공백 문자로만 구성되어 있는지 여부를 나타내는 입니다`null`String.Empty. 이 메서드는 다음과 같은 코드를 사용할 필요가 없습니다.

```
C#

if (str == null || str.Equals(String.Empty) ||
str.Trim().Equals(String.Empty))
```

다음 예제에서는 사용자 지정 `Temperature` 클래스의 구현에서 IFormattable.ToString 메서드를 사용합니다IsNullOrEmpty. 이 메서드는 "G", "C", "F" 및 "K" 형식 문자열을 지원합니다. 빈 서식 문자열 또는 값이 `null` 메서드에 전달되는 서식 문자열인 경우 해당 값은 "G" 형식 문자열로 변경됩니다.

```
C#

public string ToString(string format, IFormatProvider provider)
{
   if (String.IsNullOrEmpty(format)) format = "G";
   if (provider == null) provider = CultureInfo.CurrentCulture;

   switch (format.ToUpperInvariant())
   {
      // Return degrees in Celsius.
      case "G":
      case "C":
         return temp.ToString("F2", provider) + "°C";
      // Return degrees in Fahrenheit.
      case "F":
         return (temp * 9 / 5 + 32).ToString("F2", provider) + "°F";
      // Return degrees in Kelvin.
      case "K":
         return (temp + 273.15).ToString();
      default:
         throw new FormatException(
               String.Format("The {0} format string is not supported.",
                             format));
   }
}
```

불변성 및 StringBuilder 클래스

String 개체를 만든 후에는 값을 수정할 수 없으므로 변경할 수 없음(읽기 전용)이라고 합니다. 개체를 수정하는 것처럼 보이는 메서드는 String 실제로 수정 내용이 포함된 새

String 개체를 반환합니다.

문자열은 변경할 수 없으므로 단일 문자열로 보이는 항목에 반복 추가 또는 삭제를 수행하는 문자열 조작 루틴은 상당한 성능 저하를 초래합니다. 예를 들어 다음 코드는 난수 생성기를 사용하여 0x052F 범위 0x0001 1000자의 문자열을 만듭니다. 코드는 문자열 연결을 사용하여 이름이 지정된 `str` 기존 문자열에 새 문자를 추가하는 것처럼 보이지만 실제로 각 연결 작업에 대해 새 String 개체를 만듭니다.

C#

```
using System;
using System.IO;
using System.Text;

public class Example6
{
   public static void Main()
   {
      Random rnd = new Random();

      string str = String.Empty;
      StreamWriter sw = new StreamWriter(@".\StringFile.txt",
                           false, Encoding.Unicode);

      for (int ctr = 0; ctr <= 1000; ctr++) {
         str += (char)rnd.Next(1, 0x0530);
         if (str.Length % 60 == 0)
            str += Environment.NewLine;
      }
      sw.Write(str);
      sw.Close();
   }
}
```

문자열 값을 여러 개 변경하는 작업에 클래스 대신 String 클래스를 사용할 StringBuilder 수 있습니다. 클래스 StringBuilder 의 String 인스턴스와 달리 개체는 변경할 수 있습니다. 문자열에서 부분 문자열을 연결, 추가 또는 삭제하면 단일 문자열에서 작업이 수행됩니다. 개체 값 수정을 마쳤으면 메서드 StringBuilder.ToString 를 StringBuilder 호출하여 문자열로 변환할 수 있습니다. 다음 예제에서는 개체로 0x052F 위해 0x0001 범위에서 1000개의 임의 문자를 연결하기 위해 이전 예제에서 StringBuilder 사용된 문자를 바꿉 String 니다.

C#

```
using System;
using System.IO;
using System.Text;

public class Example10
```

```
{
   public static void Main()
   {
      Random rnd = new Random();
      StringBuilder sb = new StringBuilder();
      StreamWriter sw = new StreamWriter(@".\StringFile.txt",
                                         false, Encoding.Unicode);

      for (int ctr = 0; ctr <= 1000; ctr++) {
         sb.Append((char)rnd.Next(1, 0x0530));
         if (sb.Length % 60 == 0)
            sb.AppendLine();
      }
      sw.Write(sb.ToString());
      sw.Close();
   }
}
```

서수 및 문화권 구분 작업

클래스의 멤버는 String 개체에 대해 서수 또는 문화권 구분(언어) 작업을 수행합니다 String . 서수 작업은 각 Char 개체의 숫자 값에 대해 작동합니다. 문화권 구분 작업은 개체의 String 값에 대해 작동하며 문화권별 대/소문자 구분, 정렬, 서식 지정 및 구문 분석 규칙을 고려합니다. 문화권 구분 작업은 명시적으로 선언된 문화권 또는 암시적 현재 문화권의 컨텍스트에서 실행됩니다. 두 종류의 연산은 동일한 문자열에서 수행될 때 매우 다른 결과를 생성할 수 있습니다.

또한 .NET은 지역과 독립적인 영어의 문화권 설정을 기반으로 하는 고정 문화권 ()CultureInfo.InvariantCulture을 사용하여 문화권을 구분하지 않는 언어 문자열 작업을 지원합니다. 다른 System.Globalization.CultureInfo 설정과 달리 고정 문화권의 설정은 단일 컴퓨터, 시스템 간 및 .NET 버전 간에 일관성을 다시 기본 보장됩니다. 고정 문화권은 모든 문화권에서 문자열 비교 및 순서의 안정성을 보장하는 블랙 박스의 일종으로 볼 수 있습니다.

> **ⓘ 중요**
>
> 애플리케이션 파일 이름과 같은 기호 식별자에 대 한 보안 결정을 내리는 또는 명명된 파이프 하는 경우, XML 파일에 텍스트 기반 데이터와 같은 지속형된 데이터에 대 한 작업 대신 문화권 구분 비교는 서 수 비교를 사용 해야 합니다. 이는 문화권 구분 비교가 적용된 문화권에 따라 다른 결과를 얻을 수 있는 반면 서수 비교는 비교된 문자의 이진 값에만 의존하기 때문입니다.

> **ⓘ 중요**

문자열 작업을 수행하는 대부분의 메서드에는 형식 StringComparison매개 변수가 있는 오버로드가 포함되며, 이를 통해 메서드가 서수 또는 문화권 구분 연산을 수행할지 여부를 지정할 수 있습니다. 일반적으로 메서드 호출의 의도를 명확하게 하려면 이 오버로드를 호출해야 합니다. 문자열에서 서수 및 문화권 구분 작업을 사용하는 모범 사례 및 지침은 문자열 사용에 대한 모범 사례를 참조 **하세요**.

대/소문자 구분, 구문 분석 및 서식 지정, 비교 및 정렬 및 같음 테스트에 대한 작업은 서수 또는 문화권에 민감할 수 있습니다. 다음 섹션에서는 각 작업 범주에 대해 설명합니다.

> **팁**
>
> 항상 메서드 호출의 의도를 명확하게 하는 메서드 오버로드를 호출해야 합니다. 예를 들어 현재 문화권의 규칙을 사용하여 두 문자열의 문화권 구분 비교를 수행하도록 메서드를 호출 **Compare(String, String)** 하는 대신 인수 값 **StringComparison.CurrentCulture** `comparisonType` 이 있는 메서드를 호출 **Compare(String, String, StringComparison)** 해야 합니다. 자세한 내용은 **문자열 사용에 대한 모범 사례**를 참조하세요.

정렬 및 비교 작업에 사용되는 문자 가중치에 대한 정보가 포함된 텍스트 파일 집합인 정렬 가중치 테이블을 다음 링크에서 다운로드할 수 있습니다.

- Windows(.NET Framework 및 .NET Core): 가중치 테이블 정렬
- Windows 10 2019년 5월 업데이트 이상(.NET 5 이상) 및 Linux 및 macOS(.NET Core 및 .NET 5 이상): 기본 유니코드 데이터 정렬 요소 테이블

대/소문자 구분

대/소문자 규칙은 유니코드 문자의 대문자를 변경하는 방법을 결정합니다. 예를 들어 소문자에서 대문자로 변환합니다. 대/소문자 구분 연산은 문자열 비교 전에 수행되는 경우가 많습니다. 예를 들어 문자열을 대문자로 변환하여 다른 대문자 문자열과 비교할 수 있습니다. 또는 ToLowerInvariant 메서드를 호출 ToLower 하여 문자열의 문자를 소문자로 변환할 수 있으며 또는 메서드를 호출 ToUpperToUpperInvariant 하여 대문자로 변환할 수 있습니다. 또한 메서드를 사용하여 문자열을 TextInfo.ToTitleCase 제목 대/소문자로 변환할 수 있습니다.

> **참고**
>
> **Linux 및 macOS 시스템에서만 실행되는 .NET Core:** C 및 Posix 문화권에 대한 데이터 정렬 동작은 이러한 문화권에서 예상되는 유니코드 데이터 정렬 순서를 사용하지

않으므로 항상 대/소문자를 구분합니다. C 또는 Posix 이외의 문화권을 사용하여 문화권 구분, 대/소문자 비구분 정렬 작업을 수행하는 것이 좋습니다.

대/소문자 연산은 현재 문화권, 지정된 문화권 또는 고정 문화권의 규칙을 기반으로 할 수 있습니다. 대/소문자 매핑은 사용되는 문화권에 따라 달라질 수 있으므로 대/소문자 연산의 결과는 문화권에 따라 달라질 수 있습니다. 대/소문자의 실제 차이점은 세 가지 종류입니다.

- 라틴 문자 I(U+0049), LATIN SMALL LETTER I(U+0069), LATIN CAPITAL LETTER I WITH DOT ABOVE(U+0130) 및 LATIN SMALL LETTER DOTLESS I(U+0131)의 사례 매핑의 차이점입니다. tr-TR(터키어(터키)) 및 az-Latn-AZ(아제르바이잔, 라틴어) 문화권 및 tr, az 및 az-Latn 중립 문화권에서는 LATIN CAPITAL LETTER I의 소문자 I는 LATIN SMALL LETTER DOTLESS I이고 라틴어 SMALL LETTER I의 대문자로는 라틴 문자 I WITH DOT ABOVE입니다. 고정 문화권, LATIN SMALL LETTER I 및 LATIN CAPITAL LETTER I를 포함한 다른 모든 문화권에서는 소문자 및 대문자 등가물입니다.

  다음 예제에서는 문화권 구분 대/소문자 비교를 사용하는 경우 파일 시스템 액세스를 방지하기 위해 설계된 문자열 비교가 실패할 수 있는 방법을 보여 줍니다. 고정 문화권의 대/소문자 구분 규칙을 사용해야 합니다.

  C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example1
{
   const string disallowed = "file";

   public static void Main()
   {

IsAccessAllowed(@"FILE:\\\c:\users\user001\documents\FinancialInfo.txt"
);
   }

   private static void IsAccessAllowed(String resource)
   {
      CultureInfo[] cultures = { CultureInfo.CreateSpecificCulture("en-
US"),
                                 CultureInfo.CreateSpecificCulture("tr-
TR") };
      String scheme = null;
      int index = resource.IndexOfAny( new Char[] { '\\', '/' } );
      if (index > 0)
         scheme = resource.Substring(0, index - 1);
```

```
      // Change the current culture and perform the comparison.
      foreach (var culture in cultures) {
         Thread.CurrentThread.CurrentCulture = culture;
         Console.WriteLine("Culture: {0}",
CultureInfo.CurrentCulture.DisplayName);
         Console.WriteLine(resource);
         Console.WriteLine("Access allowed: {0}",
                           ! String.Equals(disallowed, scheme,
StringComparison.CurrentCultureIgnoreCase));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Culture: English (United States)
//       FILE:\\\c:\users\user001\documents\FinancialInfo.txt
//       Access allowed: False
//
//       Culture: Turkish (Turkey)
//       FILE:\\\c:\users\user001\documents\FinancialInfo.txt
//       Access allowed: True
```

- 고정 문화권과 다른 모든 문화권 간의 대/소문자 매핑의 차이입니다. 이러한 경우 고정 문화권의 대/소문자 규칙을 사용하여 문자를 대문자 또는 소문자로 변경하면 동일한 문자가 반환됩니다. 다른 모든 문화권의 경우 다른 문자를 반환합니다. 영향을 받는 문자 중 일부는 다음 표에 나와 있습니다.

테이블 확장

캐릭터	다음으로 변경된 경우	반환
MICRON SIGN(U+00B5)	대문자	그리스어 대문자 MU(U+-39C)
위에 점이 있는 라틴 문자 I(U+0130)	소문자	라틴 문자 I(U+0069)
LATIN SMALL LETTER DOTLESS I(U+0131)	대문자	라틴 문자 I(U+0049)
LATIN SMALL LETTER LONG S(U+017F)	대문자	라틴 문자 S(U+0053)
CARON이 있는 작은 문자 Z가 있는 라틴 문자 D(U+01C5)	소문자	CARON을 사용하여 라틴 문자 DZ(U+01C6)
그리스어 YPOGEGRAMMENI 결합 (U+0345)	대문자	그리스어 대문자 IOTA(U+0399)

- ASCII 문자 범위에서 두 문자 혼합 대/소문자 쌍의 대/소문자 매핑의 차이점입니다. 대부분의 문화권에서 두 문자 혼합 대/소문자 쌍은 해당하는 2자 대문자 또는 소문

자 쌍과 같습니다. 각 경우에 digraph와 비교되기 때문에 다음 문화권에서 다음 두 문자 쌍에 대해서는 그렇지 않습니다.

- hr-HR(크로아티아어(크로아티아) 문화권의 "lJ" 및 "nJ"입니다.
- cs-CZ(체코)와 sk-SK(슬로바키아)의 "cH".
- da-DK(덴마크어(덴마크)) 문화권의 "aA".
- hu-HU(헝가리어)의 문화권에서 "cS", "dZ", "dZS", "nY", "sZ", "tY" 및 "zS"입니다.
- es-ES_tradnl(스페인, 전통 정렬) 문화권의 "cH" 및 "lL"입니다.
- vi-VN(베트남어)의 "cH", "gI", "kH", "nG" "nH", "pH", "qU', "tH" 및 "tR"입니다.

그러나 이러한 쌍은 고정 문자열 또는 식별자에서 일반적이지 않으므로 이러한 쌍의 문화권 구분 비교로 인해 문제가 발생하는 경우는 이례적인 일입니다.

다음 예제에서는 문자열을 대문자로 변환할 때 문화권 간의 대/소문자 구분 규칙의 몇 가지 차이점을 보여 줍니다.

C#

```
using System;
using System.Globalization;
using System.IO;

public class Example
{
   public static void Main()
   {
      StreamWriter sw = new StreamWriter(@".\case.txt");
      string[] words = { "file", "sıfır", "Ďenana" };
      CultureInfo[] cultures = { CultureInfo.InvariantCulture,
                                 new CultureInfo("en-US"),
                                 new CultureInfo("tr-TR") };

      foreach (var word in words) {
         sw.WriteLine("{0}:", word);
         foreach (var culture in cultures) {
            string name = String.IsNullOrEmpty(culture.Name) ?
                                 "Invariant" : culture.Name;
            string upperWord = word.ToUpper(culture);
            sw.WriteLine("   {0,10}: {1,7} {2, 38}", name,
                         upperWord, ShowHexValue(upperWord));
         }
         sw.WriteLine();
      }
      sw.Close();
   }

   private static string ShowHexValue(string s)
   {
      string retval = null;
      foreach (var ch in s) {
         byte[] bytes = BitConverter.GetBytes(ch);
```

```
         retval += String.Format("{0:X2} {1:X2} ", bytes[1], bytes[0]);
      }
      return retval;
   }
}
// The example displays the following output:
//    file:
//        Invariant:    FILE               00 46 00 49 00 4C 00 45
//            en-US:    FILE               00 46 00 49 00 4C 00 45
//            tr-TR:    FİLE               00 46 01 30 00 4C 00 45
//
//    sıfır:
//        Invariant:   SıFıR         00 53 01 31 00 46 01 31 00 52
//            en-US:   SIFIR         00 53 00 49 00 46 00 49 00 52
//            tr-TR:   SIFIR         00 53 00 49 00 46 00 49 00 52
//
//    Ǆenana:
//        Invariant:  ǄENANA   01 C5 00 45 00 4E 00 41 00 4E 00 41
//            en-US:  ǄENANA   01 C4 00 45 00 4E 00 41 00 4E 00 41
//            tr-TR:  ǄENANA   01 C4 00 45 00 4E 00 41 00 4E 00 41
```

구문 분석 및 서식 지정

서식 지정 및 구문 분석은 역 연산입니다. 서식 지정 규칙은 날짜, 시간 또는 숫자와 같은 값을 해당 문자열 표현으로 변환하는 방법을 결정하는 반면 구문 분석 규칙은 문자열 표현을 날짜 및 시간과 같은 값으로 변환하는 방법을 결정합니다. 서식 지정 및 구문 분석 규칙은 모두 문화권 규칙에 따라 달라집니다. 다음 예제에서는 문화권별 날짜 문자열을 해석할 때 발생할 수 있는 모호성을 보여 줍니다. 날짜 문자열을 생성하는 데 사용된 문화권의 규칙을 알지 못하면 2011년 3월 1일, 2011년 3월 1일 및 2011년 1월 3일 또는 2011년 3월 1일을 나타내는지 여부를 알 수 없습니다.

C#

```
using System;
using System.Globalization;

public class Example9
{
   public static void Main()
   {
      DateTime date = new DateTime(2011, 3, 1);
      CultureInfo[] cultures = { CultureInfo.InvariantCulture,
                                 new CultureInfo("en-US"),
                                 new CultureInfo("fr-FR") };

      foreach (var culture in cultures)
         Console.WriteLine("{0,-12} {1}", String.IsNullOrEmpty(culture.Name)
?
                           "Invariant" : culture.Name,
                           date.ToString("d", culture));
```

```
   }
}
// The example displays the following output:
//       Invariant    03/01/2011
//       en-US        3/1/2011
//       fr-FR        01/03/2011
```

마찬가지로, 다음 예제와 같이 단일 문자열은 구문 분석 작업에 규칙이 사용되는 문화권에 따라 다른 날짜를 생성할 수 있습니다.

C#

```
using System;
using System.Globalization;

public class Example15
{
   public static void Main()
   {
      string dateString = "07/10/2011";
      CultureInfo[] cultures = { CultureInfo.InvariantCulture,
                                 CultureInfo.CreateSpecificCulture("en-GB"),
                                 CultureInfo.CreateSpecificCulture("en-US")
};
      Console.WriteLine("{0,-12} {1,10} {2,8} {3,8}\n", "Date String",
"Culture",
                                                        "Month", "Day");
      foreach (var culture in cultures) {
         DateTime date = DateTime.Parse(dateString, culture);
         Console.WriteLine("{0,-12} {1,10} {2,8} {3,8}", dateString,
                           String.IsNullOrEmpty(culture.Name) ?
                           "Invariant" : culture.Name,
                           date.Month, date.Day);
      }
   }
}
// The example displays the following output:
//       Date String     Culture    Month      Day
//
//       07/10/2011    Invariant        7       10
//       07/10/2011        en-GB       10        7
//       07/10/2011        en-US        7       10
```

문자열 비교 및 정렬

문자열을 비교하고 정렬하는 규칙은 문화권마다 다릅니다. 예를 들어 정렬 순서는 윗주 또는 문자의 시각적 표현을 기반으로 할 수 있습니다. 동아시아 언어에서는 표의 문자의 부수와 획에 따라 문자가 정렬됩니다. 언어와 문화권이 알파벳에 사용하는 순서에 따라 정렬 순서가 달라지기도 합니다. 예를 들어 덴마크어 알파벳의 "Æ" 문자는 "Z" 다음에 옵

니다. 또한 비교는 대/소문자를 구분하거나 대/소문자를 구분하지 않을 수 있으며 대/소문자 구분 규칙은 문화권에 따라 다를 수 있습니다. 반면 서수 비교는 문자열을 비교하고 정렬할 때 문자열에 있는 개별 문자의 유니코드 코드 요소를 사용합니다.

정렬 규칙은 유니코드 문자의 알파벳 순서와 두 문자열이 서로 비교되는 방식을 결정합니다. 예를 들어 메서드는 String.Compare(String, String, StringComparison) 매개 변수를 기반으로 두 문자열을 StringComparison 비교합니다. 매개 변수 값이면 메서드는 StringComparison.CurrentCulture현재 문화권의 규칙을 사용하는 언어 비교를 수행합니다. 매개 변수 값이 StringComparison.Ordinal면 메서드가 서수 비교를 수행합니다. 따라서 다음 예제에서 볼 수 있듯이 현재 문화권이 미국 영어인 경우 메서드에 대한 첫 번째 호출 String.Compare(String, String, StringComparison) (문화권 구분 비교 사용)은 "a"가 "A"보다 작지만 동일한 메서드(서수 비교 사용)에 대한 두 번째 호출은 "A"보다 큰 "a"를 고려합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example2
{
   public static void Main()
   {
      Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
      Console.WriteLine(String.Compare("A", "a",
StringComparison.CurrentCulture));
      Console.WriteLine(String.Compare("A", "a", StringComparison.Ordinal));
   }
}
// The example displays the following output:
//       1
//       -32
```

.NET은 단어, 문자열 및 서수 정렬 규칙을 지원합니다.

- 단어 정렬은 특정 무수 유니코드 문자에 특수 가중치가 할당될 수 있는 문자열의 문화권 구분 비교를 수행합니다. 예를 들어 하이픈(-)에는 매우 작은 가중치가 할당되어 정렬된 목록에서 "coop" 및 "co-op"이 나란히 표시될 수 있습니다. 단어 정렬 규칙을 사용하여 두 문자열을 비교하는 메서드 목록은 String 범주별 문자열 연산 섹션을 참조하세요.

- 문자열 정렬은 문화권 구분 비교도 수행합니다. 특수한 경우가 없고 모든 영숫자가 아닌 기호가 모든 영숫자 유니코드 문자 앞에 온다는 점을 제외하고 단어 정렬과 유사합니다. 값CompareOptions.StringSort이 제공된 매개 변수가 있는 메서드 오버로

드 `options` 를 호출 CompareInfo.Compare 하여 문자열 정렬 규칙을 사용하여 두 문자열을 비교할 수 있습니다. 문자열 정렬 규칙을 사용하여 두 문자열을 비교하기 위해 .NET에서 제공하는 유일한 메서드입니다.

- 서수 정렬은 문자열에 있는 각 Char 개체의 숫자 값을 기준으로 문자열을 비교합니다. 서수 비교는 문자의 소문자 및 대문자 버전에 다른 코드 포인트가 있기 때문에 자동으로 대/소문자를 구분합니다. 그러나 대/소문자를 중요하지 않은 경우 대/소문자를 무시하는 서수 비교를 지정할 수 있습니다. 이는 고정 문화권을 사용한 다음 결과에 대한 서수 비교를 수행하여 문자열을 대문자로 변환하는 것과 같습니다. 서수 정렬 규칙을 사용하여 두 문자열을 비교하는 메서드 목록은 String 범주별 문자열 작업을 참조하세요.

문화권 구분 비교는 속성에 지정된 CultureInfo.InvariantCulture 고정 문화권을 포함하여 개체를 CultureInfo 명시적으로 또는 암시적으로 사용하는 비교입니다. 암시적 문화권은 현재 문화권이며, 이 문화권은 및 CultureInfo.CurrentCulture 속성에 Thread.CurrentCulture 의해 지정됩니다. 문화권 간에 알파벳 문자의 정렬 순서(즉, 속성이 반환`true`하는 Char.IsLetter 문자)에 상당한 변형이 있습니다. 같은 문자열 비교 메서드 Compare(String, String, CultureInfo, CompareOptions)에 개체를 제공하여 CultureInfo 특정 문화권의 규칙을 사용하는 문화권 구분 비교를 지정할 수 있습니다. 메서드의 적절한 오버로드 Compare 가 아닌 CompareOptions.OrdinalIgnoreCaseCompareOptions.Ordinal 열거형의 멤버를 제공하여 StringComparison.CurrentCultureStringComparison.CurrentCultureIgnoreCase현재 문화권의 CompareOptions 규칙을 사용하는 문화권 구분 비교를 지정할 수 있습니다. 문화권 구분 비교는 일반적으로 정렬에 적합하지만 서수 비교는 그렇지 않습니다. 서수 비교는 일반적으로 두 문자열이 같은지(즉, ID를 결정하기 위해) 결정하는 데 적합하지만 문화권 구분 비교는 그렇지 않습니다.

다음 예제에서는 문화권 구분과 서수 비교의 차이를 보여 줍니다. 이 예제에서는 da-DK 및 en-US 문화권의 서수 비교 및 규칙(각각 메서드가 호출되는 시점 Compare 의 기본 문화권)을 사용하여 세 개의 문자열인 "Apple", "Æble" 및 "AEble"을 평가합니다. 덴마크어 언어는 문자 "Æ"를 개별 문자로 취급하고 알파벳의 "Z" 뒤를 정렬하기 때문에 문자열 "Æble"은 "Apple"보다 큽니다. 그러나 "Æble"은 "AEble"에 해당하는 것으로 간주되지 않으므로 "Æble"도 "AEble"보다 큽니다. en-US 문화권에는 문자 "Æ"가 포함되지 않지만 "Æble"이 "Apple"보다 작지만 "AEble"이 같은 이유를 설명하는 "AE"와 동일하게 처리됩니다. 반면 서수 비교에서는 "Apple"이 "Æble"보다 작고 "Æble"이 "AEble"보다 큰 것으로 간주합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;
```

```
public class CompareStringSample
{
   public static void Main()
   {
      string str1 = "Apple";
      string str2 = "Æble";
      string str3 = "AEble";

      // Set the current culture to Danish in Denmark.
      Thread.CurrentThread.CurrentCulture = new CultureInfo("da-DK");
      Console.WriteLine("Current culture: {0}",
                        CultureInfo.CurrentCulture.Name);
      Console.WriteLine("Comparison of {0} with {1}: {2}",
                        str1, str2, String.Compare(str1, str2));
      Console.WriteLine("Comparison of {0} with {1}: {2}\n",
                        str2, str3, String.Compare(str2, str3));

      // Set the current culture to English in the U.S.
      Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
      Console.WriteLine("Current culture: {0}",
                        CultureInfo.CurrentCulture.Name);
      Console.WriteLine("Comparison of {0} with {1}: {2}",
                        str1, str2, String.Compare(str1, str2));
      Console.WriteLine("Comparison of {0} with {1}: {2}\n",
                        str2, str3, String.Compare(str2, str3));

      // Perform an ordinal comparison.
      Console.WriteLine("Ordinal comparison");
      Console.WriteLine("Comparison of {0} with {1}: {2}",
                        str1, str2,
                        String.Compare(str1, str2,
StringComparison.Ordinal));
      Console.WriteLine("Comparison of {0} with {1}: {2}",
                        str2, str3,
                        String.Compare(str2, str3,
StringComparison.Ordinal));
   }
}
// The example displays the following output:
//       Current culture: da-DK
//       Comparison of Apple with Æble: -1
//       Comparison of Æble with AEble: 1
//
//       Current culture: en-US
//       Comparison of Apple with Æble: 1
//       Comparison of Æble with AEble: 0
//
//       Ordinal comparison
//       Comparison of Apple with Æble: -133
//       Comparison of Æble with AEble: 133
```

적절한 정렬 또는 문자열 비교 방법을 선택하려면 다음 일반 지침을 사용합니다.

- 사용자의 문화권에 따라 문자열을 정렬하려면 현재 문화권의 규칙에 따라 순서를 지정해야 합니다. 사용자의 문화권이 변경되면 정렬된 문자열의 순서도 그에 따라 변경됩니다. 예를 들어 동의어 사전 애플리케이션을 사용자의 문화권을 기준으로 단어를 항상 정렬 해야 합니다.
- 특정 문화권의 규칙에 따라 문자열을 정렬하려면 해당 문화권을 나타내는 개체를 CultureInfo 비교 메서드에 제공하여 순서를 지정해야 합니다. 예를 들어, 학생에 게 특정 언어를 설명 하도록 애플리케이션에서 원하는 문자열을 정렬할 익히면 해당 하는 문화권 중 하나로의 규칙에 따라 합니다.
- 문화권 간에 문자열 순서를 변경하지 기본 고정 문화권의 규칙에 따라 순서를 지정 하거나 서수 비교를 사용해야 합니다. 예를 들어 서수 정렬을 사용하여 파일, 프로세 스, 뮤텍스 또는 명명된 파이프의 이름을 구성합니다.
- 보안 결정(예: 사용자 이름이 유효한지 여부)을 포함하는 비교의 경우 메서드의 Equals 오버로드를 호출하여 항상 같음 서수 테스트를 수행해야 합니다.

> **참고**
>
> 문자열 비교에 사용되는 문화권 구분 정렬 및 대/소문자 규칙은 .NET 버전에 따라 달라집니다. .NET Core에서 문자열 비교는 기본 운영 체제에서 지원하는 유니코드 표준의 버전에 따라 달라집니다. Windows 8 이상에서 실행되는 .NET Framework 4.5 이상 버전에서는 정렬, 대/소문자, 정규화 및 유니코드 문자 정보가 유니코드 6.0 표준을 준수합니다. 다른 Windows 운영 체제에서는 유니코드 5.0 표준을 준수합니다.

단어, 문자열 및 서수 정렬 규칙에 대한 자세한 내용은 항목을 참조하세요 System.Globalization.CompareOptions . 각 규칙을 사용하는 경우에 대한 추가 권장 사항은 문자열 사용에 대한 모범 사례를 참조 하세요.

일반적으로 문자열의 정렬 순서를 확인하기 위해 직접 같은 Compare 문자열 비교 메서드를 호출하지 않습니다. 대신 비교 메서드는 다음과 같은 Array.Sort 메서드를 정렬하여 호출됩니다 List<T>.Sort. 다음 예제에서는 문자열 비교 메서드를 명시적으로 호출하지 않고 네 가지 정렬 작업(현재 문화권을 사용하여 단어 정렬, 고정 문화권을 사용한 단어 정렬, 서수 정렬 및 문자열 정렬)을 수행하지만 사용할 비교 형식을 지정합니다. 정렬의 각 형식은 배열에서 문자열의 고유한 순서를 생성합니다.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;

public class Example3
```

```
{
   public static void Main()
   {
      string[] strings = { "coop", "co-op", "cooperative",
                           "co\u00ADoperative", "cœur", "coeur" };

      // Perform a word sort using the current (en-US) culture.
      string[] current = new string[strings.Length];
      strings.CopyTo(current, 0);
      Array.Sort(current, StringComparer.CurrentCulture);

      // Perform a word sort using the invariant culture.
      string[] invariant = new string[strings.Length];
      strings.CopyTo(invariant, 0);
      Array.Sort(invariant, StringComparer.InvariantCulture);

      // Perform an ordinal sort.
      string[] ordinal = new string[strings.Length];
      strings.CopyTo(ordinal, 0);
      Array.Sort(ordinal, StringComparer.Ordinal);

      // Perform a string sort using the current culture.
      string[] stringSort = new string[strings.Length];
      strings.CopyTo(stringSort, 0);
      Array.Sort(stringSort, new SCompare());

      // Display array values
      Console.WriteLine("{0,13} {1,13} {2,15} {3,13} {4,13}\n",
                        "Original", "Word Sort", "Invariant Word",
                        "Ordinal Sort", "String Sort");
      for (int ctr = 0; ctr < strings.Length; ctr++)
         Console.WriteLine("{0,13} {1,13} {2,15} {3,13} {4,13}",
                           strings[ctr], current[ctr], invariant[ctr],
                           ordinal[ctr], stringSort[ctr] );
   }
}

// IComparer<String> implementation to perform string sort.
internal class SCompare : IComparer<String>
{
   public int Compare(string x, string y)
   {
      return CultureInfo.CurrentCulture.CompareInfo.Compare(x, y,
CompareOptions.StringSort);
   }
}
// The example displays the following output:
//         Original     Word Sort  Invariant Word  Ordinal Sort   String
Sort
//
//             coop          cœur            cœur         co-op         co-
op
//            co-op         coeur           coeur         coeur
cœur
//      cooperative          coop            coop          coop
```

```
coeur
//     cooperative         co-op           co-op   cooperative          coop
//             cœur   cooperative     cooperative  cooperative   cooperative
//            coeur  cooperative    cooperative          cœur  cooperative
```

> **팁**
>
> 내부적으로 .NET은 정렬 키를 사용하여 문화적으로 중요한 문자열 비교를 지원합니다. 문자열의 각 문자에는 사전순, 대/소문자 및 분음 부호를 포함하여 여러 범주의 정렬 가중치가 적용됩니다. 클래스가 **SortKey** 나타내는 정렬 키는 특정 문자열에 대해 이러한 가중치의 리포지토리를 제공합니다. 앱이 동일한 문자열 집합에서 많은 수의 검색 또는 정렬 작업을 수행하는 경우 사용하는 모든 문자열에 대한 정렬 키를 생성하고 저장하여 성능을 향상시킬 수 있습니다. 정렬 또는 비교 작업이 필요한 경우 문자열 대신 정렬 키를 사용합니다. 자세한 내용은 **SortKey** 클래스를 참조하세요.

문자열 비교 규칙을 지정하지 않으면 문자열에 대해 문화권을 구분하고 대/소문자를 구분하는 정렬을 수행하는 등의 Array.Sort(Array) 정렬 메서드가 있습니다. 다음 예제에서는 현재 문화권을 변경하면 배열에서 정렬된 문자열의 순서에 미치는 영향을 보여 줍니다. 세 문자열의 배열을 만듭니다. 우선 `System.Threading.Thread.CurrentThread.CurrentCulture` 속성을 "en-US"로 설정하고 Array.Sort(Array) 메서드를 호출합니다. 결과 정렬 순서는 영어(미국) 문화권에 대한 정렬 규칙을 기반으로 합니다. 다음으로 예제에서는 `System.Threading.Thread.CurrentThread.CurrentCulture` 속성을 da-DK로 설정하고 Array.Sort 메서드를 다시 호출합니다. 덴마크어(덴마크)의 정렬 규칙을 사용하는 경우 결과 정렬 순서가 en-US 결과와 어떻게 다른지 확인해 봅니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class ArraySort
{
   public static void Main(String[] args)
   {
      // Create and initialize a new array to store the strings.
      string[] stringArray = { "Apple", "Æble", "Zebra"};

      // Display the values of the array.
      Console.WriteLine( "The original string array:");
      PrintIndexAndValues(stringArray);

      // Set the CurrentCulture to "en-US".
      Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
      // Sort the values of the array.
```

```
      Array.Sort(stringArray);

      // Display the values of the array.
      Console.WriteLine("After sorting for the culture \"en-US\":");
      PrintIndexAndValues(stringArray);

      // Set the CurrentCulture to "da-DK".
      Thread.CurrentThread.CurrentCulture = new CultureInfo("da-DK");
      // Sort the values of the Array.
      Array.Sort(stringArray);

      // Display the values of the array.
      Console.WriteLine("After sorting for the culture \"da-DK\":");
      PrintIndexAndValues(stringArray);
   }
   public static void PrintIndexAndValues(string[] myArray)
   {
      for (int i = myArray.GetLowerBound(0); i <=
            myArray.GetUpperBound(0); i++ )
         Console.WriteLine("[{0}]: {1}", i, myArray[i]);
      Console.WriteLine();
   }
}
// The example displays the following output:
//       The original string array:
//       [0]: Apple
//       [1]: Æble
//       [2]: Zebra
//
//       After sorting for the "en-US" culture:
//       [0]: Æble
//       [1]: Apple
//       [2]: Zebra
//
//       After sorting for the culture "da-DK":
//       [0]: Apple
//       [1]: Zebra
//       [2]: Æble
```

⚠ 경고

문자열을 비교하는 주된 목적이 문자열이 같은지 여부를 확인하는 경우 메서드를 **String.Equals** 호출해야 합니다. 일반적으로 서수 비교를 수행하는 데 사용해야 **Equals** 합니다. 이 **String.Compare** 메서드는 주로 문자열을 정렬하기 위한 것입니다.

문자열 검색 메서드(예: String.StartsWith 및 String.IndexOf)는 문화권 구분 또는 서수 문자열 비교를 수행할 수도 있습니다. 다음 예제에서는 메서드를 사용하여 서수와 문화권 구분 비교의 차이점을 IndexOf 보여 줍니다. 현재 문화권이 영어(미국)인 문화권 구분 검색은 합자 "ø"와 일치하도록 부분 문자열 "oe"를 고려합니다. 소프트 하이픈(U+00AD)은

너비가 0인 문자이므로 검색은 소프트 하이픈을 동일한 String.Empty 것으로 처리하고 문자열의 시작 부분에서 일치 항목을 찾습니다. 반면 서수 검색은 두 경우 모두 일치 항목을 찾을 수 없습니다.

C#

```
using System;

public class Example8
{
   public static void Main()
   {
      // Search for "oe" and "œu" in "œufs" and "oeufs".
      string s1 = "œufs";
      string s2 = "oeufs";
      FindInString(s1, "oe", StringComparison.CurrentCulture);
      FindInString(s1, "oe", StringComparison.Ordinal);
      FindInString(s2, "œu", StringComparison.CurrentCulture);
      FindInString(s2, "œu", StringComparison.Ordinal);
      Console.WriteLine();

      string s3 = "co\u00ADoperative";
      FindInString(s3, "\u00AD", StringComparison.CurrentCulture);
      FindInString(s3, "\u00AD", StringComparison.Ordinal);
   }

   private static void FindInString(string s, string substring,
StringComparison options)
   {
      int result = s.IndexOf(substring, options);
      if (result != -1)
         Console.WriteLine("'{0}' found in {1} at position {2}",
                           substring, s, result);
      else
         Console.WriteLine("'{0}' not found in {1}",
                           substring, s);
   }
}
// The example displays the following output:
//       'oe' found in œufs at position 0
//       'oe' not found in œufs
//       'œu' found in oeufs at position 0
//       'œu' not found in oeufs
//
//       '' found in cooperative at position 0
//       '' found in cooperative at position 2
```

문자열에서 검색

문자열 검색 메서드(예: String.StartsWithString.IndexOf및)는 문화권 구분 또는 서수 문자열 비교를 수행하여 지정된 문자열에서 문자 또는 부분 문자열을 찾을 수 있는지 여부를 확인할 수도 있습니다.

메서드와 같은 개별 문자 또는 메서드와 같은 IndexOf 문자 IndexOfAny 집합 중 하나를 검색하는 클래스의 검색 String 메서드는 모두 서수 검색을 수행합니다. 문자에 대한 문화권 구분 검색을 수행하려면 메서드(예: CompareInfo.IndexOf(String, Char) 또는 CompareInfo.LastIndexOf(String, Char))를 호출 CompareInfo 해야 합니다. 서수 및 문화권 구분 비교를 사용하여 문자를 검색한 결과는 매우 다를 수 있습니다. 예를 들어 합자 "Æ"(U+00C6)와 같은 미리 컴파일된 유니코드 문자를 검색하면 문화권에 따라 "AE"(U+041U+0045)와 같은 올바른 시퀀스에서 해당 구성 요소의 발생과 일치할 수 있습니다. 다음 예제에서는 개별 문자를 검색할 String.IndexOf(Char) 때와 CompareInfo.IndexOf(String, Char) 메서드의 차이점을 보여 줍니다. 합자 "æ"(U+00E6)는 en-US 문화권의 규칙을 사용하는 경우 문자열 "aerial"에서 찾을 수 있지만 da-DK 문화권의 규칙을 사용하거나 서수 비교를 수행할 때는 찾을 수 없습니다.

C#

```
using System;
using System.Globalization;

public class Example17
{
   public static void Main()
   {
      String[] cultureNames = { "da-DK", "en-US" };
      CompareInfo ci;
      String str = "aerial";
      Char ch = 'æ';  // U+00E6

      Console.Write("Ordinal comparison -- ");
      Console.WriteLine("Position of '{0}' in {1}: {2}", ch, str,
                        str.IndexOf(ch));

      foreach (var cultureName in cultureNames) {
         ci = CultureInfo.CreateSpecificCulture(cultureName).CompareInfo;
         Console.Write("{0} cultural comparison -- ", cultureName);
         Console.WriteLine("Position of '{0}' in {1}: {2}", ch, str,
                           ci.IndexOf(str, ch));
      }
   }
}
// The example displays the following output:
//       Ordinal comparison -- Position of 'æ' in aerial: -1
//       da-DK cultural comparison -- Position of 'æ' in aerial: -1
//       en-US cultural comparison -- Position of 'æ' in aerial: 0
```

반면, String 검색 옵션이 형식 StringComparison의 매개 변수로 명시적으로 지정되지 않은 경우 문자가 아닌 문자열을 검색하는 클래스 메서드는 문화권 구분 검색을 수행합니다. 유일한 예외는 Contains서수 검색을 수행하는 것입니다.

같음 테스트

이 메서드를 String.Compare 사용하여 정렬 순서에서 두 문자열의 관계를 확인합니다. 일반적으로 이 작업은 문화권에 민감한 작업입니다. 반면, 메서드를 String.Equals 호출하여 같음을 테스트합니다. 같음 테스트는 일반적으로 사용자 입력을 유효한 사용자 이름, 암호 또는 파일 시스템 경로와 같은 알려진 문자열과 비교하기 때문에 일반적으로 서수 작업입니다.

> **⚠ 경고**
>
> 메서드를 호출 **String.Compare** 하고 반환 값이 0인지 여부를 확인하여 같음을 테스트할 수 있습니다. 그러나 이 방법은 권장되지 않습니다. 두 문자열이 같은지 여부를 확인하려면 메서드의 **String.Equals** 오버로드 중 하나를 호출해야 합니다. 두 메서드 모두 비교 형식을 명시적으로 지정하는 매개 변수를 포함 **System.StringComparison** 하므로 호출할 기본 오버로드는 인스턴스 **Equals(String, StringComparison)** 메서드 또는 정적 **Equals(String, String, StringComparison)** 메서드입니다.

다음 예제에서는 서수가 대신 사용해야 하는 경우 문화권에 민감한 비교를 수행하는 위험을 보여 줍니다. 이 경우 코드의 의도는 "FILE://" 또는 "file://"로 시작하는 URL에서 파일 시스템 액세스를 금지하기 위해 URL 시작 부분과 문자열 "FILE://"의 대/소문자를 구분하지 않는 비교를 수행합니다. 그러나 "file://"로 시작하는 URL에서 터키어(터키) 문화권을 사용하여 문화권 구분 비교를 수행하는 경우 소문자 "i"에 해당하는 터키어 대문자 값이 "I" 대신 "5"이므로 같음 비교가 실패합니다. 따라서 파일 시스템 액세스가 실수로 허용됩니다. 반면 서수 비교가 수행되면 같음 비교에 성공하고 파일 시스템 액세스가 거부됩니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example4
{
   public static void Main()
   {
      Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("tr-TR");

      string filePath = "file://c:/notes.txt";
```

```
      Console.WriteLine("Culture-sensitive test for equality:");
      if (! TestForEquality(filePath,
StringComparison.CurrentCultureIgnoreCase))
         Console.WriteLine("Access to {0} is allowed.", filePath);
      else
         Console.WriteLine("Access to {0} is not allowed.", filePath);

      Console.WriteLine("\nOrdinal test for equality:");
      if (! TestForEquality(filePath, StringComparison.OrdinalIgnoreCase))
         Console.WriteLine("Access to {0} is allowed.", filePath);
      else
         Console.WriteLine("Access to {0} is not allowed.", filePath);
   }

   private static bool TestForEquality(string str, StringComparison cmp)
   {
      int position = str.IndexOf("://");
      if (position < 0) return false;

      string substring = str.Substring(0, position);
      return substring.Equals("FILE", cmp);
   }
}
// The example displays the following output:
//       Culture-sensitive test for equality:
//       Access to file://c:/notes.txt is allowed.
//
//       Ordinal test for equality:
//       Access to file://c:/notes.txt is not allowed.
```

표준화

일부 유니코드 문자에는 여러 표현이 있습니다. 예를 들어 다음 코드 포인트 중 어느 것이든 문자 "1"을 나타낼 수 있습니다.

- U+1EAF
- U+0103 U+0301
- U+0061 U+0306 U+0301

단일 문자에 대한 여러 표현은 검색, 정렬, 일치 및 기타 문자열 작업을 복잡하게 만듭니다.

유니코드 표준은 해당하는 이진 표현에 대해 유니코드 문자의 이진 표현 하나를 반환하는 정규화라는 프로세스를 정의합니다. 정규화는 다른 규칙을 따르는 정규화 양식이라는 여러 알고리즘을 사용할 수 있습니다. .NET은 유니코드 정규화 형식 C, D, KC 및 KD를 지원합니다. 문자열이 동일한 정규화 형식으로 정규화된 경우 서수 비교를 사용하여 비교할 수 있습니다.

서수 비교는 각 문자열에 있는 해당 Char 개체의 유니코드 스칼라 값에 대한 이진 비교입니다. 클래스에는 String 다음을 포함하여 서수 비교를 수행할 수 있는 여러 메서드가 포함됩니다.

- 매개 변수를 Compare포함하는 메서드, , StartsWithEquals, EndsWithIndexOf및 LastIndexOf 메서드의 모든 오버로드입니다StringComparison. 이 매개 변수의 값을 StringComparison.OrdinalOrdinalIgnoreCase 제공하는 경우 메서드는 서수 비교를 수행합니다.
- 메서드의 오버로드입니다 CompareOrdinal .
- 서수 비교를 사용하는 메서드(예: Contains, Replace및 Split.
- 문자열 인스턴스의 Char 배열에서 값 또는 요소를 검색하는 Char 메서드입니다. 이러한 메서드에는 포함 IndexOf(Char) 및 Split(Char[]).

메서드를 호출 String.IsNormalized() 하여 문자열이 정규화 형식 C로 정규화되는지 여부를 확인하거나 메서드를 호출 String.IsNormalized(NormalizationForm) 하여 문자열이 지정된 정규화 형식으로 정규화되는지 여부를 확인할 수 있습니다. 메서드를 String.Normalize() 호출하여 문자열을 정규화 형식 C로 변환하거나 메서드를 String.Normalize(NormalizationForm) 호출하여 문자열을 지정된 정규화 양식으로 변환할 수도 있습니다. 문자열을 정규화하고 비교하는 방법에 대한 단계별 정보는 및 Normalize(NormalizationForm) 메서드를 Normalize() 참조하세요.

다음 간단한 예제에서는 문자열 정규화를 보여 줍니다. 세 가지 다른 문자열에서 세 가지 방법으로 문자 ""를 정의하고 같음의 서수 비교를 사용하여 각 문자열이 다른 두 문자열과 다른지 확인합니다. 그런 다음 각 문자열을 지원되는 정규화 형식으로 변환하고 지정된 정규화 형식으로 각 문자열의 서수 비교를 다시 수행합니다. 각 경우에서 두 번째 같음 테스트는 문자열이 같음을 보여 줍니다.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Text;

public class Example13
{
   private static StreamWriter sw;

   public static void Main()
   {
      sw = new StreamWriter(@".\TestNorm1.txt");

      // Define three versions of the same word.
      string s1 = "sống";        // create word with U+1ED1
```

```
      string s2 = "s\u00F4\u0301ng";
      string s3 = "so\u0302\u0301ng";

      TestForEquality(s1, s2, s3);
      sw.WriteLine();

      // Normalize and compare strings using each normalization form.
      foreach (string formName in Enum.GetNames(typeof(NormalizationForm)))
      {
         sw.WriteLine("Normalization {0}:\n", formName);
         NormalizationForm nf = (NormalizationForm)
Enum.Parse(typeof(NormalizationForm), formName);
         string[] sn = NormalizeStrings(nf, s1, s2, s3);
         TestForEquality(sn);
         sw.WriteLine("\n");
      }

      sw.Close();
   }

   private static void TestForEquality(params string[] words)
   {
      for (int ctr = 0; ctr <= words.Length - 2; ctr++)
         for (int ctr2 = ctr + 1; ctr2 <= words.Length - 1; ctr2++)
            sw.WriteLine("{0} ({1}) = {2} ({3}): {4}",
                         words[ctr], ShowBytes(words[ctr]),
                         words[ctr2], ShowBytes(words[ctr2]),
                         words[ctr].Equals(words[ctr2],
StringComparison.Ordinal));
   }

   private static string ShowBytes(string str)
   {
      string result = null;
      foreach (var ch in str)
         result += $"{(ushort)ch:X4} ";
      return result.Trim();
   }

   private static string[] NormalizeStrings(NormalizationForm nf, params
string[] words)
   {
      for (int ctr = 0; ctr < words.Length; ctr++)
         if (! words[ctr].IsNormalized(nf))
            words[ctr] = words[ctr].Normalize(nf);
      return words;
   }
}
// The example displays the following output:
//       sống (0073 1ED1 006E 0067) = sống (0073 00F4 0301 006E 0067): False
//       sống (0073 1ED1 006E 0067) = sống (0073 006F 0302 0301 006E 0067):
False
//       sống (0073 00F4 0301 006E 0067) = sống (0073 006F 0302 0301 006E
0067): False
//
```

```
//       Normalization FormC:
//
//       sống (0073 1ED1 006E 0067) = sống (0073 1ED1 006E 0067): True
//       sống (0073 1ED1 006E 0067) = sống (0073 1ED1 006E 0067): True
//       sống (0073 1ED1 006E 0067) = sống (0073 1ED1 006E 0067): True
//
//
//       Normalization FormD:
//
//       sống (0073 006F 0302 0301 006E 0067) = sống (0073 006F 0302 0301
006E 0067): True
//       sống (0073 006F 0302 0301 006E 0067) = sống (0073 006F 0302 0301
006E 0067): True
//       sống (0073 006F 0302 0301 006E 0067) = sống (0073 006F 0302 0301
006E 0067): True
//
//
//       Normalization FormKC:
//
//       sống (0073 1ED1 006E 0067) = sống (0073 1ED1 006E 0067): True
//       sống (0073 1ED1 006E 0067) = sống (0073 1ED1 006E 0067): True
//       sống (0073 1ED1 006E 0067) = sống (0073 1ED1 006E 0067): True
//
//
//       Normalization FormKD:
//
//       sống (0073 006F 0302 0301 006E 0067) = sống (0073 006F 0302 0301
006E 0067): True
//       sống (0073 006F 0302 0301 006E 0067) = sống (0073 006F 0302 0301
006E 0067): True
//       sống (0073 006F 0302 0301 006E 0067) = sống (0073 006F 0302 0301
006E 0067): True
```

정규화 및 정규화 양식에 대한 자세한 내용은 System.Text.NormalizationForm유니코드 표준 부록 #15: 유니코드 정규화 양식 및 unicode.org 웹 사이트의 정규화 FAQ를 참조하세요.

범주별 문자열 작업

클래스는 String 문자열 비교, 같음 문자열 테스트, 문자열에서 문자 또는 부분 문자열 찾기, 문자열 수정, 문자열에서 부분 문자열 추출, 문자열 결합, 값 서식 지정, 문자열 복사 및 문자열 정규화에 대한 멤버를 제공합니다.

문자열 비교

다음 String 메서드를 사용하여 문자열을 비교하여 정렬 순서에서 상대 위치를 확인할 수 있습니다.

- Compare 는 정렬 순서에서 한 문자열과 두 번째 문자열의 관계를 나타내는 정수입니다.
- CompareOrdinal 는 해당 코드 요소의 비교를 기반으로 한 문자열과 두 번째 문자열의 관계를 나타내는 정수입니다.
- CompareTo 는 정렬 순서에서 현재 문자열 인스턴스와 두 번째 문자열의 관계를 나타내는 정수입니다. 메서드는 CompareTo(String) 클래스에 IComparable 대한 String 구현 및 IComparable<T> 기능을 제공합니다.

문자열이 같은지 테스트

메서드를 Equals 호출하여 두 문자열이 같은지 여부를 확인합니다. 인스턴스 Equals(String, String, StringComparison) 및 정적 Equals(String, StringComparison) 오버로드를 사용하면 비교가 문화권 구분 또는 서수인지 여부와 대/소문자를 고려하거나 무시할지 여부를 지정할 수 있습니다. 대부분의 같음 테스트는 서수이며, 시스템 리소스(예: 파일 시스템 개체)에 대한 액세스를 결정하는 같음 비교는 항상 서수여야 합니다.

문자열에서 문자 찾기

클래스에는 String 두 가지 종류의 검색 메서드가 포함됩니다.

- 특정 부분 문자열이 문자열 인스턴스에 있는지 여부를 나타내는 값을 반환 Boolean 하는 메서드입니다. 여기에는 , EndsWith및 메서드가 StartsWith 포함Contains됩니다.
- 문자열 인스턴스에서 부분 문자열의 시작 위치를 나타내는 메서드입니다. 여기에는 , IndexOfAny, LastIndexOf및 메서드가 LastIndexOfAny 포함IndexOf됩니다.

> ⚠ **경고**
>
> 특정 부분 문자열이 아닌 특정 패턴에 대한 문자열을 검색하려면 정규식을 사용해야 합니다. 자세한 내용은 .NET 정규식을 참조 **하세요**.

문자열 수정

클래스에는 String 문자열 값을 수정하는 것처럼 보이는 다음 메서드가 포함됩니다.

- Insert 는 문자열을 현재 String 인스턴스에 삽입합니다.
- PadLeft 는 문자열의 시작 부분에 지정된 문자를 하나 이상 삽입합니다.

- PadRight 문자열의 끝에 지정된 문자가 하나 이상 삽입됩니다.
- Remove 는 현재 String 인스턴스에서 부분 문자열을 삭제합니다.
- Replace 는 하위 문자열을 현재 String 인스턴스의 다른 부분 문자열로 바꿉니다.
- ToLower 문자열 ToLowerInvariant 의 모든 문자를 소문자로 변환합니다.
- ToUpper 문자열 ToUpperInvariant 의 모든 문자를 대문자로 변환합니다.
- Trim 는 문자열의 시작과 끝에서 문자의 모든 발생을 제거합니다.
- TrimEnd 는 문자열의 끝에서 문자의 모든 발생을 제거합니다.
- TrimStart 는 문자열의 시작 부분에서 문자의 모든 발생을 제거합니다.

> ⓘ **중요**
>
> 모든 문자열 수정 메서드는 새 **String** 개체를 반환합니다. 현재 인스턴스의 값은 수정하지 않습니다.

문자열에서 부분 문자열 추출

메서드는 String.Split 단일 문자열을 여러 문자열로 구분합니다. 메서드의 오버로드를 사용하면 여러 구분 기호를 지정하고, 메서드가 추출하는 부분 문자열 수를 제한하고, 부분 문자열에서 공백을 트리밍하고, 빈 문자열(구분 기호가 인접할 때 발생함)이 반환된 문자열에 포함되는지 여부를 지정할 수 있습니다.

문자열 결합

문자열 연결에는 다음 String 메서드를 사용할 수 있습니다.

- Concat 는 하나 이상의 하위 문자열을 단일 문자열로 결합합니다.
- Join 하나 이상의 부분 문자열을 단일 요소에 연결하고 각 부분 문자열 사이에 구분 기호를 추가합니다.

값 서식 지정

이 메서드는 String.Format 복합 서식 지정 기능을 사용하여 문자열에 있는 하나 이상의 자리 표시자를 일부 개체 또는 값의 문자열 표현으로 대체합니다. 이 Format 메서드는 종종 다음을 수행하는 데 사용됩니다.

- 문자열에 숫자 값의 문자열 표현을 포함하려면

- 문자열에 날짜 및 시간 값의 문자열 표현을 포함하려면
- 문자열에 열거형 값의 문자열 표현을 포함하려면
- 문자열의 인터페이스를 지원하는 일부 개체의 문자열 표현을 IFormattable 포함하려면
- 더 큰 문자열 내의 필드에서 부분 문자열을 오른쪽 맞춤 또는 왼쪽 맞춤하려면

서식 지정 작업 및 예제에 대한 자세한 내용은 오버로드 요약을 Format 참조하세요.

문자열을 복사합니다.

다음 String 메서드를 호출하여 문자열의 복사본을 만들 수 있습니다.

- Clone 는 기존 개체에 대한 참조를 반환합니다 String .
- Copy 는 기존 문자열의 복사본을 만듭니다.
- CopyTo 는 문자열의 일부를 문자 배열에 복사합니다.

문자열 정규화

유니코드에서 단일 문자에는 여러 코드 포인트가 있을 수 있습니다. 정규화는 이러한 해당 문자를 동일한 이진 표현으로 변환합니다. 메서드는 String.Normalize 정규화를 수행하고 메서드는 String.IsNormalized 문자열이 정규화되는지 여부를 결정합니다.

자세한 내용과 예제는 이 문서의 앞부분에 있는 정규화 섹션을 참조하세요.

System.String 생성자

> ⓘ **참고 항목**
>
> 이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

오버로드된 생성자 구문

문자열 생성자는 포인터 매개 변수가 없는 범주와 포인터 매개 변수가 있는 범주의 두 가지 범주로 구분됩니다. 포인터를 사용하는 생성자는 CLS 규격이 아닙니다. 또한 Visual Basic은 포인터 사용을 지원하지 않으며 C#에는 안전하지 않은 컨텍스트에서 실행하기 위해 포인터를 사용하는 코드가 필요합니다. 자세한 내용은 unsafe를 참조하세요.

오버로드를 선택하는 방법에 대한 추가 지침은 어떤 메서드를 호출하나요?를 참조하세요.

`String(Char[] value)`
유니코드 문자 배열로 표시되는 값으로 새 인스턴스를 초기화합니다. 이 생성자는 유니코드 문자를 복사합니다(예제 2: 문자 배열 사용).

`String(Char[] value, Int32 startIndex, Int32 length)`
유니코드 문자 배열, 해당 배열 내의 시작 문자 위치 및 길이로 표시된 값으로 새 인스턴스를 초기화합니다(예 3: 문자 배열의 일부를 사용하고 단일 문자를 반복).

`String(Char c, Int32 count)`
지정된 유니코드 문자가 지정된 횟수만큼 반복되는 값으로 새 인스턴스를 초기화합니다(예 3: 문자 배열의 일부 사용 및 단일 문자 반복).

`String(char* value)`
(CLS 규격 아님) null 문자(U+0000 또는 '\0')로 종료되는 유니코드 문자 배열에 대한 포인터로 표시되는 값으로 새 인스턴스를 초기화합니다. (예제 4: 문자 배열에 대한 포인터 사용).

권한: SecurityCriticalAttribute, 즉시 호출자에게 완전한 신뢰가 필요합니다. 이 멤버는 부분적으로 신뢰할 수 있는 또는 투명 코드에서 사용할 수 없습니다.

`String(char* value, Int32 startIndex, Int32 length)`
(CLS 규격 아님) 유니코드 문자 배열, 해당 배열 내의 시작 문자 위치 및 길이에 대한 포인터로 표시되는 값으로 새 인스턴스를 초기화합니다. 생성자는 의 유니코드 문자를 인덱스 에서 시작하여 인덱스 - 1에서 종료까지 복사합니다 (예 5: 포인터 및 배열 범위에서 문자열을 인스턴스화).

권한: SecurityCriticalAttribute, 즉시 호출자에게 완전한 신뢰가 필요합니다. 이 멤버는 부분적으로 신뢰할 수 있는 또는 투명 코드에서 사용할 수 없습니다.

`String(SByte* value)`

(CLS 규격 아님) 새 인스턴스를 8비트 부가 정수 배열에 대한 포인터로 표시된 값으로 초기화합니다. 배열은 현재 시스템 코드 페이지(즉, 지정된 인코딩)를 사용하여 인코딩된 Encoding.Default문자열을 나타내는 것으로 간주됩니다. 생성자는 포인터가 `value` 지정한 위치에서 시작하여 null 문자(0x00)에 도달할 때까지 문자를 처리합니다(예 6: 포인터에서 부호 있는 바이트 배열로 문자열 인스턴스화).

권한: SecurityCriticalAttribute, 즉시 호출자에게 완전한 신뢰가 필요합니다. 이 멤버는 부분적으로 신뢰할 수 있는 또는 투명 코드에서 사용할 수 없습니다.

`String(SByte* value, Int32 startIndex, Int32 length)`

(CLS 규격 아님) 새 인스턴스를 8비트 부가 정수의 배열, 해당 배열 내의 시작 위치 및 길이에 대한 포인터로 표시된 값으로 초기화합니다. 배열은 현재 시스템 코드 페이지(즉, 지정된 인코딩)를 사용하여 인코딩된 Encoding.Default문자열을 나타내는 것으로 간주됩니다. 생성자는 -1에서 `startIndex` 시작하여 끝나는 `startIndex` + `length`값의 문자를 처리합니다(예 6: 포인터에서 부호 있는 바이트 배열로 문자열 인스턴스화).

권한: SecurityCriticalAttribute, 즉시 호출자에게 완전한 신뢰가 필요합니다. 이 멤버는 부분적으로 신뢰할 수 있는 또는 투명 코드에서 사용할 수 없습니다.

`String(SByte* value, Int32 startIndex, Int32 length, Encoding enc)`

(CLS 규격 아님) 새 인스턴스를 8비트 부가 정수의 배열, 해당 배열 내의 시작 위치, 길이 및 Encoding 개체에 대한 포인터로 표시된 값으로 초기화합니다.

권한: SecurityCriticalAttribute, 즉시 호출자에게 완전한 신뢰가 필요합니다. 이 멤버는 부분적으로 신뢰할 수 있는 또는 투명 코드에서 사용할 수 없습니다.

매개 변수

다음은 포인터 매개 변수를 포함하지 않는 생성자가 사용하는 String 매개 변수의 전체 목록입니다. 각 오버로드에서 사용되는 매개 변수는 위의 오버로드 구문을 참조하세요.

테이블 확장

매개 변수	타입	설명
`value`	Boolean(불리언)[]	유니코드 문자 배열입니다.
`c`	Char	유니코드 문자입니다.
`startIndex`	Int32	새 문자열의 `value` 첫 번째 문자의 시작 위치입니다. 기본값: 0

매개 변수	타입	설명
`length`	Int32	새 문자열에 `value` 포함할 문자 수입니다. 기본값: Array.Length
`count`	Int32	새 문자열에서 문자 `c` 가 반복되는 횟수입니다. `count` 값이 0이면 새 개체의 값은 String.Empty입니다.

다음은 포인터 매개 변수를 포함하는 생성자가 사용하는 String 매개 변수의 전체 목록입니다. 각 오버로드에서 사용되는 매개 변수는 위의 오버로드 구문을 참조하세요.

테이블 확장

매개 변수	타입	설명
`value`	Char* 또는 SByte*	Null로 끝나는 유니코드 문자 배열 또는 부호 있는 8비트 정수 배열에 대한 포인터입니다. `value`가 배열이거나 빈 배열인 경우, 새 문자열의 값은 `null`String.Empty입니다.
`startIndex`	Int32	새 문자열의 첫 번째 문자를 정의하는 배열 요소의 인덱스입니다. 기본값: 0
`length`	Int32	새 문자열을 만드는 데 사용할 배열 요소의 수입니다. 길이가 0이면 생성자는 값 String.Empty이 .인 문자열을 만듭니다. 기본값: Array.Length
`enc`	Encoding	배열을 인코딩하는 방법을 `value` 지정하는 개체입니다. 기본값: Encoding.Default또는 시스템의 현재 ANSI 코드 페이지

예외

다음은 포인터 매개 변수를 포함하지 않는 생성자가 throw한 예외 목록입니다.

테이블 확장

예외	조건	에 의해 발생됨:
ArgumentNullException	`value`은 `null`입니다.	String(Char[], Int32, Int32)

예외	조건	에 의해 발생됨:
ArgumentOutOfRangeException	`startIndex`, `length` 또는 `count`가 0보다 작습니다. 또는 `startIndex`와 `length`의 합계가 `value`의 요소 수보다 큽니다. 또는 `count`가 0보다 작습니다.	String(Char, Int32) String(Char[], Int32, Int32)

다음은 포인터 매개 변수를 포함하는 생성자가 throw한 예외 목록입니다.

테이블 확장

예외	조건	에 의해 발생됨:
ArgumentException	`value` 은 잘못된 유니코드 문자를 포함하는 배열을 지정합니다. 또는 `value` 또는 `value` + `startIndex` 64K 미만의 주소를 지정합니다. 또는 기본 코드 페이지 인코딩을 String 사용하지 않으므로 바이트 배열 `value` 에서 새 `value` 인스턴스를 초기화할 수 없습니다.	포인터가 있는 모든 생성자입니다.
ArgumentNullException	`value`가 null입니다.	String(SByte*) String(SByte*, Int32, Int32) String(SByte*, Int32, Int32, Encoding)
ArgumentOutOfRangeException	현재 프로세스에 주소가 지정된 모든 문자에 대한 읽기 액세스 권한이 있는 것은 아닙니다. 또는 `startIndex` 또는 `length`가 0보다 작거나 `value` +	포인터가 있는 모든 생성자입니다.

예외	조건	에 의해 발생됨:
	`startIndex`로 인해 포인터 오버플로가 발생하거나, 현재 프로세스에서 주소가 지정된 모든 문자에 대한 읽기 액세스 권한을 갖지는 않습니다. 또는 새 문자열의 길이가 너무 커서 할당할 수 없습니다.	
AccessViolationException	`value`또는 `value` + `startIndex` + `length` - 1은 잘못된 주소를 지정합니다.	String(SByte*) String(SByte*, Int32, Int32) String(SByte*, Int32, Int32, Encoding)

어떤 메서드를 호출합니까?

테이블 확장

수신:	통화 또는 사용
문자열을 만듭니다.	문자열 리터럴 또는 기존 문자열에서 할당(예제 1: 문자열 할당 사용)
전체 문자 배열에서 문자열을 만듭니다.	String(Char[]) (예제 2: 문자 배열 사용)
문자 배열의 일부에서 문자열을 만듭니다.	String(Char[], Int32, Int32) (예제 3: 문자 배열의 일부를 사용하고 단일 문자를 반복)
동일한 문자를 여러 번 반복하는 문자열을 만듭니다.	String(Char, Int32) (예제 3: 문자 배열의 일부를 사용하고 단일 문자를 반복)
포인터에서 유니코드 또는 와이드 문자 배열에 대한 문자열을 만듭니다.	String(Char*)
포인터를 사용하여 유니코드 또는 와이드 문자 배열의 일부에서 문자열을 만듭니다.	String(Char*, Int32, Int32)
C++ `char` 배열에서 문자열을 만듭니다.	String(SByte*), String(SByte*, Int32, Int32) 또는 String(SByte*, Int32, Int32, Encoding)

수신:	통화 또는 사용
ASCII 문자에서 문자열을 만듭니다.	ASCIIEncoding.GetString

문자열 만들기

프로그래밍 방식으로 문자열을 만드는 데 가장 일반적으로 사용되는 기술은 예제 1에 설명된 것처럼 간단한 할당입니다. 클래스에는 String 다음 값에서 문자열을 만들 수 있는 4가지 유형의 생성자 오버로드도 포함됩니다.

- 문자 배열에서(UTF-16으로 인코딩된 문자의 배열) 전체 배열의 문자 또는 일부에서 새 String 개체를 만들 수 있습니다. String(Char[]) 생성자는 배열의 모든 문자를 새 문자열에 복사합니다. String(Char[], Int32, Int32) 생성자는 인덱스 `startIndex`부터 인덱스 `startIndex` + `length` - 1까지의 문자를 새 문자열로 복사합니다. `length`이 0이면, 새 문자열의 값은 String.Empty입니다.

  코드 반복적으로 동일한 값이 있는 문자열을 인스턴스화하는 경우 문자열을 작성 하는 대체 방법을 사용 하 여 애플리케이션 성능을 개선할 수 있습니다. 자세한 내용은 반복 문자열 처리를 참조 하세요.

- 생성자 String(Char, Int32)를 사용하여, 단일 문자가 0번, 1번 또는 여러 번 중복될 수 있습니다. `count`이 0이면, 새 문자열의 값은 String.Empty입니다.

- 포인터를 null로 종료되는 문자 배열에 사용하는 경우 String(Char*) 생성자를 사용합니다 String(Char*, Int32, Int32). 전체 배열 또는 지정된 범위를 사용하여 문자열을 초기화할 수 있습니다. 생성자는 지정된 포인터에서 시작하거나 지정된 포인터에 `startIndex`를 더한 위치에서 시작하여, 배열의 끝까지 또는 `length` 문자를 복사하는 유니코드 문자 시퀀스를 생성합니다. `value`가 null 포인터이거나 `length`가 0인 경우, 생성자는 값이 String.Empty인 문자열을 만듭니다. 복사 작업이 배열의 끝으로 진행되고 배열이 null로 종료되지 않은 경우 생성자 동작은 시스템에 종속됩니다. 이러한 조건으로 인해 액세스 위반이 발생할 수 있습니다.

  배열에 포함된 null 문자(U+0000 또는 '\0')가 포함되어 있고 String(Char*, Int32, Int32) 오버로드가 호출되는 경우 문자열 인스턴스에는 포함된 null을 `length` 포함한 문자가 포함됩니다. 다음 예제에서는 두 개의 null 문자를 포함하는 10개 요소의 배열에 대한 포인터가 메서드에 String(Char*, Int32, Int32) 전달될 때 발생하는 작업을 보여줍니다. 주소는 배열의 시작 부분이며 배열의 모든 요소를 문자열에 추가해야 하므로 생성자는 포함된 null 2개를 포함하여 10자로 문자열을 인스턴스화합니다. 반면, 동일한 배열이 생성자에 전달 String(Char*) 되는 경우 결과는 첫 번째 null 문자를 포함하지 않는 4자 문자열입니다.

  C#

```
using System;

public class Example2
{
   public unsafe static void Main()
   {
      char[] chars = { 'a', 'b', 'c', 'd', '\0', 'A', 'B', 'C', 'D', '\0' };
      string s = null;

      fixed(char* chPtr = chars) {
         s = new string(chPtr, 0, chars.Length);
      }

      foreach (var ch in s)
         Console.Write($"{(ushort)ch:X4} ");
      Console.WriteLine();

      fixed(char* chPtr = chars) {
         s = new string(chPtr);
      }

      foreach (var ch in s)
         Console.Write($"{(ushort)ch:X4} ");
      Console.WriteLine();
   }
}
// The example displays the following output:
//       0061 0062 0063 0064 0000 0041 0042 0043 0044 0000
//       0061 0062 0063 0064
```

배열에는 유니코드 문자가 포함되어야 합니다. C++에서는 문자 배열을 관리 Char되는 [] 형식 또는 관리`wchar_t`되지 않는[] 형식으로 정의해야 합니다.

오버로드가 String(Char*) 호출되고 배열이 null로 종료되지 않거나 오버로드가 호출되고 String(Char*, Int32, Int32)`startIndex` + -1에 문자 시퀀스에 할당된 메모리 외부의 범위가 포함된 경우 `length` 생성자의 동작은 시스템에 따라 달라지며 액세스 위반이 발생할 수 있습니다.

- 포인터에서 시작하여 부호 있는 바이트 배열로. 전체 배열 또는 지정된 범위를 사용하여 문자열을 초기화할 수 있습니다. 기본 코드 페이지 인코딩을 사용하여 바이트 시퀀스를 해석하거나 생성자 호출에서 인코딩을 지정할 수 있습니다. 생성자가 null로 종료되지 않은 전체 배열에서 문자열을 인스턴스화하려고 하거나 배열 범위가 -1에서 `value` + `startIndex` -1까지`value` + `startIndex` + `length`의 범위가 배열에 할당된 메모리 외부에 있는 경우 이 생성자의 동작은 시스템에 종속되며 액세스 위반이 발생할 수 있습니다.

  부호 있는 바이트 배열을 매개 변수로 포함하는 세 개의 생성자는 이 예제와 같이 주로 C++ `char` 배열을 문자열로 변환하도록 설계되었습니다.

```
```

C++

```
using namespace System;

void main()
{
      char chars[] = { 'a', 'b', 'c', 'd', '\x00' };

      char* charPtr = chars;
      String^ value = gcnew String(charPtr);

      Console::WriteLine(value);
}
// The example displays the following output:
//       abcd
```

배열에 null 문자('\0') 또는 값이 0인 바이트가 포함된 경우 String(SByte*, Int32, Int32) 오버로드가 호출되면, 문자열 인스턴스는 포함된 null 문자를 포함한 `length` 문자를 갖습니다. 다음 예제에서는 두 개의 null 문자를 포함하는 10개 요소의 배열에 대한 포인터가 메서드에 String(SByte*, Int32, Int32) 전달될 때 발생하는 작업을 보여줍니다. 주소는 배열의 시작 부분이며 배열의 모든 요소를 문자열에 추가해야 하므로 생성자는 포함된 null 2개를 포함하여 10자로 문자열을 인스턴스화합니다. 반면, 동일한 배열이 생성자에 전달 String(SByte*) 되는 경우 결과는 첫 번째 null 문자를 포함하지 않는 4자 문자열입니다.

C#

```
using System;

public class Example5
{
   public unsafe static void Main()
   {
      sbyte[] bytes = { 0x61, 0x62, 0x063, 0x064, 0x00, 0x41, 0x42, 0x43,
0x44, 0x00 };

      string s = null;
      fixed (sbyte* bytePtr = bytes) {
         s = new string(bytePtr, 0, bytes.Length);
      }

      foreach (var ch in s)
         Console.Write($"{(ushort)ch:X4} ");

      Console.WriteLine();

      fixed(sbyte* bytePtr = bytes) {
         s = new string(bytePtr);
      }

      foreach (var ch in s)
         Console.Write($"{(ushort)ch:X4} ");
```

```
      Console.WriteLine();
   }
}
// The example displays the following output:
//       0061 0062 0063 0064 0000 0041 0042 0043 0044 0000
//       0061 0062 0063 0064
```

String(SByte*) 및 String(SByte*, Int32, Int32) 생성자는 기본 ANSI 코드 페이지를 사용하여 해석 `value` 하기 때문에 바이트 배열이 동일한 이러한 생성자를 호출하면 다른 시스템에서 값이 다른 문자열을 만들 수 있습니다.

반복 문자열 처리

텍스트 스트림을 구문 분석하거나 디코딩하는 앱은 종종 생성자 또는 String(Char[], Int32, Int32) 메서드를 사용하여 StringBuilder.Append(Char[], Int32, Int32) 문자 시퀀스를 문자열로 변환합니다. 하나의 문자열을 만들고 다시 사용하는 대신 동일한 값으로 새 문자열을 반복적으로 만들면 메모리가 낭비됩니다. 생성자를 호출 String(Char[], Int32, Int32) 하여 동일한 문자열 값을 반복적으로 만들 가능성이 있는 경우 동일한 문자열 값이 무엇인지 미리 모르는 경우에도 대신 조회 테이블을 사용할 수 있습니다.

예를 들어 XML 태그 및 특성이 포함된 파일에서 문자 스트림을 읽고 구문 분석한다고 가정합니다. 스트림을 구문 분석할 때 특정 토큰(즉, 기호적 의미가 있는 문자 시퀀스)이 반복적으로 발생합니다. "0", "1", "true" 및 "false" 문자열에 해당하는 토큰은 XML 스트림에서 자주 발생할 수 있습니다.

각 토큰을 새 문자열로 변환하는 대신 일반적으로 발생하는 문자열을 System.Xml.NameTable 저장할 개체를 만들 수 있습니다. 개체는 NameTable 임시 메모리를 할당하지 않고 저장된 문자열을 검색하기 때문에 성능을 향상시킵니다. 토큰이 발견되면 메서드를 NameTable.Get(Char[], Int32, Int32) 사용하여 테이블에서 토큰을 검색합니다. 토큰이 있는 경우 메서드는 해당 문자열을 반환합니다. 토큰이 없으면 메서드를 NameTable.Add(Char[], Int32, Int32) 사용하여 테이블에 토큰을 삽입하고 해당 문자열을 가져옵니다.

예제 1: 문자열 할당 사용

다음 예제에서는 문자열 리터럴을 할당하여 새 문자열을 만듭니다. 첫 번째 문자열의 값을 할당하여 두 번째 문자열을 만듭니다. 다음은 새 String 개체를 인스턴스화하는 가장 일반적인 두 가지 방법입니다.

C#

```
using System;

public class Example3
```

```
{
   public static void Main()
   {
      String value1 = "This is a string.";
      String value2 = value1;
      Console.WriteLine(value1);
      Console.WriteLine(value2);
   }
}
// The example displays the following output:
//    This is a string.
//    This is a string.
```

VB

```
Module Example
   Public Sub Main()
      Dim value1 As String = "This is a string."
      Dim value2 As String = value1
      Console.WriteLine(value1)
      Console.WriteLine(value2)
   End Sub
End Module
' The example displays the following output:
'    This is a string.
'    This is a string.
```

예제 2: 문자 배열 사용

다음 예제에서는 문자 배열에서 새 String 개체를 만드는 방법을 보여 줍니다.

C#

```
// Unicode Mathematical operators
char [] charArr1 = {'\u2200','\u2202','\u200F','\u2205'};
String szMathSymbols = new String(charArr1);

// Unicode Letterlike Symbols
char [] charArr2 = {'\u2111','\u2118','\u2122','\u2126'};
String szLetterLike = new String (charArr2);

// Compare Strings - the result is false
Console.WriteLine("The Strings are equal? " +
    (String.Compare(szMathSymbols, szLetterLike)==0?"true":"false") );
```

VB

```
' Unicode Mathematical operators
Dim charArr1() As Char = {ChrW(&H2200), ChrW(&H2202), _
```

```
                          ChrW(&H200F), ChrW(&H2205)}
Dim szMathSymbols As New String(charArr1)

' Unicode Letterlike Symbols
Dim charArr2() As Char = {ChrW(&H2111), ChrW(&H2118), _
                          ChrW(&H2122), ChrW(&H2126)}
Dim szLetterLike As New String(charArr2)

' Compare Strings - the result is false
Console.WriteLine("The strings are equal? " & _
        CStr(szMathSymbols.Equals(szLetterLike)))
```

예제 3: 문자 배열의 일부를 사용하고 단일 문자를 반복합니다.

다음 예제에서는 문자 배열의 일부에서 새 String 개체를 만드는 방법과 단일 문자가 여러 번 나타나는 새 String 개체를 만드는 방법을 보여 줍니다.

C#

```
// Create a Unicode String with 5 Greek Alpha characters
String szGreekAlpha = new String('\u0391',5);
// Create a Unicode String with a Greek Omega character
String szGreekOmega = new String(new char [] {'\u03A9','\u03A9','\u03A9'},2,1);

String szGreekLetters = String.Concat(szGreekOmega, szGreekAlpha,
szGreekOmega.Clone());

// Examine the result
Console.WriteLine(szGreekLetters);

// The first index of Alpha
int ialpha = szGreekLetters.IndexOf('\u0391');
// The last index of Omega
int iomega = szGreekLetters.LastIndexOf('\u03A9');

Console.WriteLine("The Greek letter Alpha first appears at index " + ialpha +
    " and Omega last appears at index " + iomega + " in this String.");
```

VB

```
' Create a Unicode String with 5 Greek Alpha characters
Dim szGreekAlpha As New String(ChrW(&H0391), 5)
' Create a Unicode String with a Greek Omega character
Dim szGreekOmega As New String(New Char() {ChrW(&H03A9), ChrW(&H03A9), _
                                           ChrW(&H03A9)}, 2, 1)

Dim szGreekLetters As String = String.Concat(szGreekOmega, szGreekAlpha, _
                                             szGreekOmega.Clone())
```

```
' Examine the result
Console.WriteLine(szGreekLetters)

' The first index of Alpha
Dim iAlpha As Integer = szGreekLetters.IndexOf(ChrW(&H0391))
' The last index of Omega
Dim iomega As Integer = szGreekLetters.LastIndexOf(ChrW(&H03A9))

Console.WriteLine("The Greek letter Alpha first appears at index {0}.", _
                  ialpha)
Console.WriteLIne("The Greek letter Omega last appears at index {0}.", _
                  iomega)
```

예제 4: 문자 배열에 대한 포인터 사용

다음 예제에서는 포인터에서 문자 배열에 대한 새 String 개체를 만드는 방법을 보여 줍니다. C# 예제는 컴파일러 스위치를 `/unsafe` 사용하여 컴파일해야 합니다.

C#

```
using System;

public class Example4
{
   public static unsafe void Main()
   {
      char[] characters = { 'H', 'e', 'l', 'l', 'o', ' ',
                            'w', 'o', 'r', 'l', 'd', '!', '\u0000' };
      string value;

      fixed (char* charPtr = characters) {
         value = new String(charPtr);
      }
      Console.WriteLine(value);
   }
}
// The example displays the following output:
//       Hello world!
```

예제 5: 포인터 및 배열 범위에서 문자열 인스턴스화

다음 예제에서는 마침표 또는 느낌표에 대한 문자 배열의 요소를 검사합니다. 하나가 발견되면, 문장 부호 앞에 오는 배열의 문자들로부터 문자열 객체를 생성합니다. 그렇지 않은 경우 배열의 전체 내용으로 문자열을 인스턴스화합니다. C# 예제는 컴파일러 스위치를 `/unsafe` 사용하여 컴파일해야 합니다.

C#

```
using System;

public class Example1
{
   public static unsafe void Main()
   {
      char[] characters = { 'H', 'e', 'l', 'l', 'o', ' ',
                            'w', 'o', 'r', 'l', 'd', '!', '\u0000' };
      String value;

      fixed (char* charPtr = characters) {
         int length = 0;
         Char* iterator = charPtr;

         while (*iterator != '\x0000')
         {
            if (*iterator == '!' || *iterator == '.')
               break;
            iterator++;
            length++;
         }
         value = new String(charPtr, 0, length);
      }
      Console.WriteLine(value);
   }
}
// The example displays the following output:
//      Hello World
```

예제 6: 포인터에서 부호 있는 바이트 배열로 문자열 인스턴스화

다음 예제에서는 생성자를 사용하여 클래스 String 의 인스턴스를 String(SByte*) 만드는 방법을 보여 줍니다.

C#

```
unsafe
{
    // Null terminated ASCII characters in an sbyte array
    String szAsciiUpper = null;
    sbyte[] sbArr1 = new sbyte[] { 0x41, 0x42, 0x43, 0x00 };
    // Instruct the Garbage Collector not to move the memory
    fixed(sbyte* pAsciiUpper = sbArr1)
    {
        szAsciiUpper = new String(pAsciiUpper);
    }
    String szAsciiLower = null;
```

```
    sbyte[] sbArr2 = { 0x61, 0x62, 0x63, 0x00 };
    // Instruct the Garbage Collector not to move the memory
    fixed(sbyte* pAsciiLower = sbArr2)
    {
        szAsciiLower = new String(pAsciiLower, 0, sbArr2.Length);
    }
    // Prints "ABC abc"
    Console.WriteLine(szAsciiUpper + " " + szAsciiLower);

    // Compare Strings - the result is true
    Console.WriteLine("The Strings are equal when capitalized ? " +
        (String.Compare(szAsciiUpper.ToUpper(),
szAsciiLower.ToUpper())==0?"true":"false") );

    // This is the effective equivalent of another Compare method, which ignores
case
    Console.WriteLine("The Strings are equal when capitalized ? " +
        (String.Compare(szAsciiUpper, szAsciiLower, true)==0?"true":"false") );
}
```

Last updated on 2026. 02. 12.

System.String.Format 메서드

아티클 • 2025. 03. 26.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> ⓘ **중요**
>
> **String.Format** 메서드를 호출하거나 복합 형식 문자열을 사용하는 대신, 언어에서 지원하는 경우 *보간된 문자열*을 사용할 수 있습니다. 보간된 문자열은 *보간된 식*을 포함하는 문자열입니다. 보간된 각 식은 식의 값으로 확인되고 문자열이 할당될 때 결과 문자열에 포함됩니다. 자세한 내용은 **문자열 보간(C# 참조)** 및 보간된 문자열 **(Visual Basic 참조)**참조하세요.

예시

Format 메서드를 호출하는 수많은 예제는 이 문서 전체에 산재되어 있습니다. C# .NET Core 프로젝트가 포함된 전체 `String.Format` 예제 집합을 다운로드할 수도 있습니다.

다음은 문서에 포함된 몇 가지 예입니다.

형식 문자열 만들기

문자열 삽입
서식 항목
동일한 인덱스를 가진 항목 서식 지정

형식이 지정된 출력 제어

제어 서식
컨트롤 간격
컨트롤 정렬
정수 자릿수를 제어하다
소수 구분 기호 뒤의 숫자 수를 제어합니다.
결과 문자열에 리터럴 중괄호를 포함

서식 문자열을 문화에 민감하게 만들기

서식 문자열 문화권 구분

서식 지정 작업 사용자 지정

사용자 지정 서식 지정 작업
인터셉트 제공자 및 로마 숫자 포맷터

String.Format 메서드 시작

개체, 변수 또는 식의 값을 다른 문자열에 삽입해야 하는 경우 String.Format 사용합니다. 예를 들어 Decimal 값의 값을 문자열에 삽입하여 사용자에게 단일 문자열로 표시할 수 있습니다.

C#

```
Decimal pricePerOunce = 17.36m;
String s = String.Format("The current price is {0} per ounce.",
                         pricePerOunce);
Console.WriteLine(s);
// Result: The current price is 17.36 per ounce.
```

또한 해당 값의 서식을 제어할 수 있습니다.

C#

```
Decimal pricePerOunce = 17.36m;
String s = String.Format("The current price is {0:C2} per ounce.",
                         pricePerOunce);
Console.WriteLine(s);
// Result if current culture is en-US:
//      The current price is $17.36 per ounce.
```

서식 외에도 맞춤 및 간격을 제어할 수 있습니다.

문자열 삽입

String.Format 형식 문자열로 시작하고 문자열로 변환되고 서식 문자열의 지정된 위치에 삽입되는 하나 이상의 개체 또는 식으로 시작합니다. 다음은 그 예입니다.

C#

```
decimal temp = 20.4m;
string s = String.Format("The temperature is {0}°C.", temp);
Console.WriteLine(s);
// Displays 'The temperature is 20.4°C.'
```

형식 문자열의 `{0}` 형식 항목입니다. `0` 문자열 값이 해당 위치에 삽입되는 개체의 인덱스입니다. (인덱스는 0부터 시작합니다.) 삽입할 개체가 문자열이 아니면 결과 문자열에 삽입하기 전에 해당 `ToString` 메서드를 호출하여 1로 변환합니다.

다음은 개체 목록에서 두 개의 형식 항목과 두 개의 개체를 사용하는 또 다른 예입니다.

C#

```
string s = String.Format("At {0}, the temperature is {1}°C.",
                         DateTime.Now, 20.4);
Console.WriteLine(s);
// Output similar to: 'At 4/10/2015 9:29:41 AM, the temperature is 20.4°C.'
```

모든 서식 항목의 인덱스가 개체 목록에 일치하는 개체를 갖는 한 원하는 만큼의 서식 항목과 개체를 개체 목록에 포함할 수 있습니다. 또한 호출하는 오버로드에 대해 걱정할 필요가 없습니다. 컴파일러가 적절한 오버로드를 선택해 줍니다.

컨트롤 서식 지정

서식 문자열이 있는 서식 항목의 인덱스로 개체의 서식을 지정하는 방법을 제어할 수 있습니다. 예를 들어 `{0:d}` 개체 목록의 첫 번째 개체에 "d" 서식 문자열을 적용합니다. 다음은 단일 개체와 두 개의 형식 항목이 있는 예제입니다.

C#

```
string s = String.Format("It is now {0:d} at {0:t}", DateTime.Now);
Console.WriteLine(s);
// Output similar to: 'It is now 4/10/2015 at 10:04 AM'
```

다양한 형식이 형식 문자열을 지원합니다. 모든 숫자 형식(표준 및 사용자 지정 형식 문자열 모두), 모든 날짜 및 시간(표준 및 사용자 지정 형식 문자열 모두) 및 시간 포함 간격(표준 및 사용자 지정 형식 문자열), 모든 열거형 형식 열거형 형식및 GUID. 형식 문자열에 대한 지원을 사용자 고유의 형식에 추가할 수도 있습니다.

간격 제어

12자 문자열을 삽입하는 `{0,12}`같은 구문을 사용하여 결과 문자열에 삽입되는 문자열의 너비를 정의할 수 있습니다. 이 경우 첫 번째 개체의 문자열 표현은 12자 필드에서 오른쪽 맞춤됩니다. (첫 번째 개체의 문자열 표현 길이가 12자를 초과하는 경우 기본 설정 필드 너비는 무시되고 전체 문자열이 결과 문자열에 삽입됩니다.)

다음 예제에서는 문자열 "Year"와 일부 연도 문자열을 저장할 6자 필드와 문자열 "Population" 및 일부 모집단 데이터를 저장할 15자 필드를 정의합니다. 주의하세요, 문자

가 필드에서 오른쪽으로 정렬되어 있습니다.

C#

```
int[] years = { 2013, 2014, 2015 };
int[] population = { 1025632, 1105967, 1148203 };
var sb = new System.Text.StringBuilder();
sb.Append(String.Format("{0,6} {1,15}\n\n", "Year", "Population"));
for (int index = 0; index < years.Length; index++)
    sb.Append(String.Format("{0,6} {1,15:N0}\n", years[index],
population[index]));

Console.WriteLine(sb);

// Result:
//      Year      Population
//
//      2013       1,025,632
//      2014       1,105,967
//      2015       1,148,203
```

컨트롤 정렬

기본적으로 필드 너비를 지정하는 경우 문자열은 해당 필드 내에서 오른쪽 맞춤됩니다. 필드의 문자열을 왼쪽에 맞추려면 필드 너비 앞에 음수 기호(예: `{0,-12}`)를 추가하여 12자 왼쪽 맞춤 필드를 정의합니다.

다음 예제는 레이블과 데이터를 모두 왼쪽으로 정렬한다는 점을 제외하고 이전 예제와 비슷합니다.

C#

```
int[] years = { 2013, 2014, 2015 };
int[] population = { 1025632, 1105967, 1148203 };
String s = String.Format("{0,-10} {1,-10}\n\n", "Year", "Population");
for (int index = 0; index < years.Length; index++)
    s += String.Format("{0,-10} {1,-10:N0}\n",
                       years[index], population[index]);
Console.WriteLine($"\n{s}");
// Result:
//    Year       Population
//
//    2013       1,025,632
//    2014       1,105,967
//    2015       1,148,203
```

String.Format 복합 서식 기능을 사용합니다. 자세한 내용은 복합 서식 지정을 참조하세요.

어떤 메서드를 호출합니까?

테이블 확장

에게	전화
현재 문화권의 규칙을 사용하여 하나 이상의 개체 서식을 지정합니다.	`provider` 매개 변수를 포함하는 오버로드를 제외하고 나머지 Format 오버로드는 하나 이상의 개체 매개 변수 뒤에 String 매개 변수를 포함합니다. 이 때문에 호출하려는 Format 오버로드를 확인할 필요가 없습니다. 언어 컴파일러는 인수 목록에 따라 `provider` 매개 변수가 없는 오버로드 중에서 적절한 오버로드를 선택합니다. 예를 들어 인수 목록에 5개의 인수가 있는 경우 컴파일러는 Format(String, Object[]) 메서드를 호출합니다.
특정 문화권의 규칙을 사용하여 하나 이상의 개체 서식을 지정합니다.	`provider` 매개 변수로 시작하는 각 Format 오버로드 뒤에는 String 매개 변수와 하나 이상의 개체 매개 변수가 잇습니다. 이 때문에 호출하려는 특정 Format 오버로드를 확인할 필요가 없습니다. 언어 컴파일러는 인수 목록에 따라 `provider` 매개 변수가 있는 오버로드 중에서 적절한 오버로드를 선택합니다. 예를 들어 인수 목록에 5개의 인수가 있는 경우 컴파일러는 Format(IFormatProvider, String, Object[]) 메서드를 호출합니다.
ICustomFormatter 구현 또는 IFormattable 구현을 사용하여 사용자 지정 서식 지정 작업을 수행합니다.	`provider` 매개 변수가 있는 4개의 오버로드 중 어느 것이든지. 컴파일러는 인수 목록에 따라 `provider` 매개 변수가 있는 오버로드 중에서 적절한 오버로드를 선택합니다.

Format 메서드를 간략하게 설명합니다.

Format 메서드의 각 오버로드는 복합 서식 지정 기능를 사용하여 복합 형식 문자열에 0부터 시작하는 인덱싱된 자리 표시자인 *형식 항목*를 포함합니다. 런타임에 각 형식 항목은 매개 변수 목록에서 해당 인수의 문자열 표현으로 바뀝 있습니다. 인수 값이 `null`경우 서식 항목이 String.Empty바뀝다. 예를 들어 Format(String, Object, Object, Object) 메서드에 대한 다음 호출에는 {0}, {1}및 {2}세 개의 형식 항목이 있는 형식 문자열과 세 개의 항목이 있는 인수 목록이 포함됩니다.

C#

```
DateTime dat = new DateTime(2012, 1, 17, 9, 30, 0);
string city = "Chicago";
int temp = -16;
string output = String.Format("At {0} in {1}, the temperature was {2}
degrees.",
                              dat, city, temp);
Console.WriteLine(output);
```

```
// The example displays output like the following:
//    At 1/17/2012 9:30:00 AM in Chicago, the temperature was -16 degrees.
```

서식 항목

형식 항목에는 다음 구문이 있습니다.

```txt
{index[,alignment][:formatString]}
```

대괄호는 선택적 요소를 나타냅니다. 여는 중괄호와 닫는 중괄호가 필요합니다. 서식 문자열에 리터럴 여는 중괄호 또는 닫는 중괄호를 포함하려면 이스케이프 중괄호 섹션을 복합 서식 문서에서 참조하세요.

예를 들어 통화 값의 서식을 지정하는 서식 항목은 다음과 같이 표시될 수 있습니다.

```C#
var value = String.Format("{0,-10:C}", 126347.89m);
Console.WriteLine(value);
```

형식 항목에는 다음과 같은 요소가 있습니다.

인덱스
문자열 표현을 문자열의 이 위치에 포함할 인수의 인덱스(0부터 시작)입니다. 이 인수가 `null`경우 빈 문자열이 문자열의 이 위치에 포함됩니다.

정렬
선택 사항. 인수가 삽입되는 필드의 총 길이와 인수가 오른쪽 맞춤(양수) 또는 왼쪽 맞춤(음수 정수)인지 여부를 나타내는 부호 있는 정수입니다. 맞춤 생략하면 선행 또는 후행 공백이 없는 필드에 해당 인수의 문자열 표현이 삽입됩니다.

맞춤 값이 삽입할 인수의 길이보다 작으면 *맞춤* 무시되고 인수의 문자열 표현 길이가 필드 너비로 사용됩니다.

formatString
선택 사항. 해당 인수의 결과 문자열 형식을 지정하는 문자열입니다. formatString 생략하면 해당 인수의 매개 변수가 없는 `ToString` 메서드가 호출되어 문자열 표현이 생성됩니다. formatString 지정하는 경우 형식 항목에서 참조하는 인수는 IFormattable 인터페이스를 구현해야 합니다. 형식 문자열을 지원하는 형식은 다음과 같습니다.

- 모든 정수 및 부동 소수점 타입. (표준 숫자 서식 문자열 및 사용자 지정 숫자 서식 문자열 참조하세요.)
- DateTime 및 DateTimeOffset. (표준 날짜 및 시간 형식 문자열 참조하고 사용자 지정 날짜 및 시간 서식 문자열 .)
- 모든 열거형 (열거형 형식 문자열참조하세요.)
- TimeSpan 값입니다. 표준 TimeSpan 형식 문자열 및 사용자 지정 TimeSpan 형식 문자열참조하세요.
- GUID. (Guid.ToString(String) 메서드를 참조하세요.)

그러나 모든 사용자 지정 형식은 IFormattable 구현하거나 기존 형식의 IFormattable 구현을 확장할 수 있습니다.

다음 예제에서는 `alignment` 및 `formatString` 인수를 사용하여 형식이 지정된 출력을 생성합니다.

C#

```
// Create array of 5-tuples with population data for three U.S. cities,
1940-1950.
Tuple<string, DateTime, int, DateTime, int>[] cities =
    { Tuple.Create("Los Angeles", new DateTime(1940, 1, 1), 1504277,
                  new DateTime(1950, 1, 1), 1970358),
      Tuple.Create("New York", new DateTime(1940, 1, 1), 7454995,
                  new DateTime(1950, 1, 1), 7891957),
      Tuple.Create("Chicago", new DateTime(1940, 1, 1), 3396808,
                  new DateTime(1950, 1, 1), 3620962),
      Tuple.Create("Detroit", new DateTime(1940, 1, 1), 1623452,
                  new DateTime(1950, 1, 1), 1849568) };

// Display header
var header = String.Format("{0,-12}{1,8}{2,12}{1,8}{2,12}{3,14}\n",
                           "City", "Year", "Population", "Change (%)");
Console.WriteLine(header);
foreach (var city in cities) {
   var output = String.Format("{0,-12}{1,8:yyyy}{2,12:N0}{3,8:yyyy}{4,12:N0}
{5,14:P1}",
                        city.Item1, city.Item2, city.Item3, city.Item4,
city.Item5,
                        (city.Item5 - city.Item3)/ (double)city.Item3);
   Console.WriteLine(output);
}
// The example displays the following output:
//    City            Year  Population    Year  Population    Change (%)
//
//    Los Angeles     1940   1,504,277    1950   1,970,358        31.0 %
//    New York        1940   7,454,995    1950   7,891,957         5.9 %
```

```
//    Chicago          1940   3,396,808    1950   3,620,962         6.6 %
//    Detroit          1940   1,623,452    1950   1,849,568        13.9 %
```

인수 형식 지정 방법

형식 항목은 문자열의 시작 부분에서 순차적으로 처리됩니다. 각 형식 항목에는 메서드의 인수 목록에 있는 개체에 해당하는 인덱스가 있습니다. Format 메서드는 인수를 검색하고 다음과 같이 문자열 표현을 파생합니다.

- 인수가 `null`경우 메서드는 결과 문자열에 String.Empty 삽입합니다. null 인수에 대해 NullReferenceException를 처리할 필요가 없습니다.
- Format(IFormatProvider, String, Object[]) 오버로드를 호출하고 `provider` 개체의 IFormatProvider.GetFormat 구현이 null이 아닌 ICustomFormatter 구현을 반환하면 인수가 해당 ICustomFormatter.Format(String, Object, IFormatProvider) 메서드로 전달됩니다. 형식 항목에 *formatString* 인수가 포함된 경우 메서드에 첫 번째 인수로 전달됩니다. ICustomFormatter 구현을 사용할 수 있고 null이 아닌 문자열을 생성하는 경우 해당 문자열은 인수의 문자열 표현으로 반환됩니다. 그렇지 않으면 다음 단계가 실행됩니다.
- 인수가 IFormattable 인터페이스를 구현하는 경우 해당 IFormattable.ToString 구현이 호출됩니다.
- 기본 클래스 구현에서 재정의하거나 상속하는 인수의 매개 변수 없는 `ToString` 메서드가 호출됩니다.

ICustomFormatter.Format 메서드에 대한 호출을 가로채고 Format 메서드가 복합 형식 문자열의 각 서식 항목에 대한 서식 지정 메서드에 전달하는 정보를 확인할 수 있는 예제는 예제: 절편 공급자 및 로마 숫자 포맷터참조하세요.

자세한 내용은 처리 순서참조하세요.

인덱스가 같은 항목 서식 지정

인덱스 항목의 인덱스가 인수 목록의 인수 수보다 크거나 같은 경우 Format 메서드는 FormatException 예외를 throw합니다. 그러나 `format` 여러 서식 항목의 인덱스가 같으면 인수보다 더 많은 형식 항목을 포함할 수 있습니다. 다음 예제의 Format(String, Object) 메서드 호출에서 인수 목록에는 단일 인수가 있지만 형식 문자열에는 두 개의 형식 항목이 포함됩니다. 하나는 숫자의 10진수를 표시하고 다른 하나는 16진수 값을 표시합니다.

```
C#
```

```
short[] values= { Int16.MinValue, -27, 0, 1042, Int16.MaxValue };
Console.WriteLine("{0,10}  {1,10}\n", "Decimal", "Hex");
foreach (short value in values)
{
   string formatString = String.Format("{0,10:G}: {0,10:X}", value);
   Console.WriteLine(formatString);
}
// The example displays the following output:
//       Decimal         Hex
//
//        -32768:       8000
//           -27:       FFE5
//             0:          0
//          1042:        412
//         32767:       7FFF
```

서식과 문화

일반적으로 인수 목록의 개체는 CultureInfo.CurrentCulture 속성에서 반환되는 현재 문화권의 규칙을 사용하여 문자열 표현으로 변환됩니다. `provider` 매개 변수를 포함하는 Format 오버로드 중 하나를 호출하여 이 동작을 제어할 수 있습니다. `provider` 매개 변수는 서식 지정 프로세스를 조정하는 데 사용되는 사용자 지정 및 문화권별 서식 정보를 제공하는 IFormatProvider 구현입니다.

IFormatProvider 인터페이스에는 서식 정보를 제공하는 개체를 반환하는 단일 멤버인 GetFormat있습니다. .NET에는 문화권별 서식을 제공하는 세 가지 IFormatProvider 구현이 있습니다.

- CultureInfo; 해당 GetFormat 메서드는 숫자 값의 서식을 지정하기 위한 문화권별 NumberFormatInfo 개체와 날짜 및 시간 값의 서식을 지정하기 위한 문화권별 DateTimeFormatInfo 개체를 반환합니다.
- DateTimeFormatInfo- 날짜 및 시간 값의 문화권별 서식 지정에 사용됩니다. 해당 GetFormat 메서드는 자신을 반환합니다.
- NumberFormatInfo- 숫자 값의 문화권별 서식 지정에 사용됩니다. 해당 GetFormat(Type) 메서드는 자신을 반환합니다.

사용자 지정 서식 지정 작업

IFormatProvider 형식의 `provider` 매개 변수가 있는 Format 메서드의 오버로드를 호출하여 사용자 지정 서식 지정 작업을 수행할 수도 있습니다. 예를 들어 정수를 ID 번호 또는 전화 번호로 서식을 지정할 수 있습니다. 사용자 지정 서식을 수행하려면 `provider` 인수가 IFormatProvider 인터페이스와 ICustomFormatter 인터페이스를 모두 구현해야 합니다. Format 메서드가 ICustomFormatter 구현을 `provider` 인수로 전달하면 Format 메서드는

IFormatProvider.GetFormat 구현을 호출하고 ICustomFormatter형식의 개체를 요청합니다. 그런 다음 반환된 ICustomFormatter 개체의 Format 메서드를 호출하여 전달된 복합 문자열의 각 서식 항목에 서식을 지정합니다.

사용자 지정 서식 지정 솔루션을 제공하는 방법에 대한 자세한 내용은 방법: 사용자 지정 숫자 형식 공급자 정의 및 사용 및 ICustomFormatter참조하세요. 정수를 서식이 지정된 사용자 지정 숫자로 변환하는 예제는 예제: 사용자 지정 서식 지정 작업참조하세요. 부호 없는 바이트를 로마 숫자로 변환하는 예제는 예제: 절편 공급자 및 로마 숫자 포맷터참조하세요.

예: 사용자 지정 서식 지정 작업

이 예제에서는 x-xxxxx-xx 형식의 정수 값 형식을 고객 계정 번호로 지정하는 형식 공급자를 정의합니다.

C#

```
using System;

public class TestFormatter
{
   public static void Main()
   {
      int acctNumber = 79203159;
      Console.WriteLine(String.Format(new CustomerFormatter(), "{0}",
acctNumber));
      Console.WriteLine(String.Format(new CustomerFormatter(), "{0:G}",
acctNumber));
      Console.WriteLine(String.Format(new CustomerFormatter(), "{0:S}",
acctNumber));
      Console.WriteLine(String.Format(new CustomerFormatter(), "{0:P}",
acctNumber));
      try {
         Console.WriteLine(String.Format(new CustomerFormatter(), "{0:X}",
acctNumber));
      }
      catch (FormatException e) {
         Console.WriteLine(e.Message);
      }
   }
}

public class CustomerFormatter : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
```

```
   }

   public string Format(string format,
                        object arg,
                        IFormatProvider formatProvider)
   {
      if (! this.Equals(formatProvider))
      {
         return null;
      }
      else
      {
         if (String.IsNullOrEmpty(format))
            format = "G";

         string customerString = arg.ToString();
         if (customerString.Length < 8)
            customerString = customerString.PadLeft(8, '0');

         format = format.ToUpper();
         switch (format)
         {
            case "G":
               return customerString.Substring(0, 1) + "-" +
                                    customerString.Substring(1, 5) + "-" +
                                    customerString.Substring(6);
            case "S":
               return customerString.Substring(0, 1) + "/" +
                                    customerString.Substring(1, 5) + "/" +
                                    customerString.Substring(6);
            case "P":
               return customerString.Substring(0, 1) + "." +
                                    customerString.Substring(1, 5) + "." +
                                    customerString.Substring(6);
            default:
               throw new FormatException(
                        String.Format("The '{0}' format specifier is not
supported.", format));
         }
      }
   }
}
// The example displays the following output:
//       7-92031-59
//       7-92031-59
//       7/92031/59
//       7.92031.59
//       The 'X' format specifier is not supported.
```

예: 인터셉트 공급자 및 로마 숫자 포맷터

이 예제에서는 두 가지 작업을 수행하는 ICustomFormatter 및 IFormatProvider 인터페이스를 구현하는 사용자 지정 형식 공급자를 정의합니다.

- ICustomFormatter.Format 구현에 전달된 매개 변수를 표시합니다. 이렇게 하면 Format(IFormatProvider, String, Object[]) 메서드가 서식을 지정하려는 각 개체에 대한 사용자 지정 서식 구현으로 전달되는 매개 변수를 확인할 수 있습니다. 이 기능은 애플리케이션을 디버깅할 때 유용할 수 있습니다.

- 서식을 지정할 개체가 "R" 표준 서식 문자열을 사용하여 서식을 지정할 부호 없는 바이트 값인 경우 사용자 지정 포맷터는 숫자 값의 서식을 로마 숫자로 지정합니다.

C#

```
using System;
using System.Globalization;

public class InterceptProvider : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(String format, Object obj, IFormatProvider provider)
   {
      // Display information about method call.
      string formatString = format ?? "<null>";
      Console.WriteLine("Provider: {0}, Object: {1}, Format String: {2}",
                        provider.GetType().Name, obj ?? "<null>",
formatString);

      if (obj == null) return String.Empty;

      // If this is a byte and the "R" format string, format it with Roman
numerals.
      if (obj is Byte && formatString.ToUpper().Equals("R")) {
         Byte value = (Byte) obj;
         int remainder;
         int result;
         String returnString = String.Empty;

         // Get the hundreds digit(s)
         result = Math.DivRem(value, 100, out remainder);
         if (result > 0)
            returnString = new String('C', result);
         value = (Byte) remainder;
         // Get the 50s digit
         result = Math.DivRem(value, 50, out remainder);
         if (result == 1)
```

```
            returnString += "L";
         value = (Byte) remainder;
         // Get the tens digit.
         result = Math.DivRem(value, 10, out remainder);
         if (result > 0)
            returnString += new String('X', result);
         value = (Byte) remainder;
         // Get the fives digit.
         result = Math.DivRem(value, 5, out remainder);
         if (result > 0)
            returnString += "V";
         value = (Byte) remainder;
         // Add the ones digit.
         if (remainder > 0)
            returnString += new String('I', remainder);

         // Check whether we have too many X characters.
         int pos = returnString.IndexOf("XXXX");
         if (pos >= 0) {
            int xPos = returnString.IndexOf("L");
            if (xPos >= 0 & xPos == pos - 1)
               returnString = returnString.Replace("LXXXX", "XC");
            else
               returnString = returnString.Replace("XXXX", "XL");
         }
         // Check whether we have too many I characters
         pos = returnString.IndexOf("IIII");
         if (pos >= 0)
            if (returnString.IndexOf("V") >= 0)
               returnString = returnString.Replace("VIIII", "IX");
            else
               returnString = returnString.Replace("IIII", "IV");

         return returnString;
      }

      // Use default for all other formatting.
      if (obj is IFormattable)
         return ((IFormattable) obj).ToString(format,
CultureInfo.CurrentCulture);
      else
         return obj.ToString();
   }
}

public class Example
{
   public static void Main()
   {
      int n = 10;
      double value = 16.935;
      DateTime day = DateTime.Now;
      InterceptProvider provider = new InterceptProvider();
      Console.WriteLine(String.Format(provider, "{0:N0}: {1:C2} on {2:d}\n",
n, value, day));
```

```
      Console.WriteLine(String.Format(provider, "{0}: {1:F}\n", "Today: ",
                                      (DayOfWeek) DateTime.Now.DayOfWeek));
      Console.WriteLine(String.Format(provider, "{0:X}, {1}, {2}\n",
                                      (Byte) 2, (Byte) 12, (Byte) 199));
      Console.WriteLine(String.Format(provider, "{0:R}, {1:R}, {2:R}\n",
                                      (Byte) 2, (Byte) 12, (Byte) 199));
   }
}
// The example displays the following output:
//    Provider: InterceptProvider, Object: 10, Format String: N0
//    Provider: InterceptProvider, Object: 16.935, Format String: C2
//    Provider: InterceptProvider, Object: 1/31/2013 6:10:28 PM, Format
String: d
//    10: $16.94 on 1/31/2013
//
//    Provider: InterceptProvider, Object: Today: , Format String: <null>
//    Provider: InterceptProvider, Object: Thursday, Format String: F
//    Today: : Thursday
//
//    Provider: InterceptProvider, Object: 2, Format String: X
//    Provider: InterceptProvider, Object: 12, Format String: <null>
//    Provider: InterceptProvider, Object: 199, Format String: <null>
//    2, 12, 199
//
//    Provider: InterceptProvider, Object: 2, Format String: R
//    Provider: InterceptProvider, Object: 12, Format String: R
//    Provider: InterceptProvider, Object: 199, Format String: R
//    II, XII, CXCIX
```

자주 묻는 질문(FAQ)

`String.Format` 메서드 호출에 대해 문자열 보간을 권장하는 이유는 무엇인가요?

문자열 보간은 다음과 같습니다.

- 더 유연합니다. 복합 서식을 지원하는 메서드를 호출할 필요 없이 문자열에서 사용할 수 있습니다. 그렇지 않으면 Format 메서드 또는 복합 서식을 지원하는 다른 메서드(예: Console.WriteLine 또는 StringBuilder.AppendFormat)를 호출해야 합니다.
- 더 읽기 쉬움. 문자열에 삽입할 식이 인수 목록이 아닌 보간된 식에 표시되므로 보간된 문자열은 코딩 및 읽기가 훨씬 쉽습니다. 가독성이 높기 때문에 보간된 문자열은 복합 형식 메서드에 대한 호출을 대체할 수 있을 뿐만 아니라 문자열 연결 작업에도 사용하여 보다 간결하고 명확한 코드를 생성할 수 있습니다.

다음 두 코드 예제를 비교하면 문자열 연결과 복합 서식 지정 메서드 호출보다 보간된 문자열의 우위를 보여 줍니다. 다음 예제에서는 여러 문자열 연결 작업을 사용하여 코드가

장황해지고 읽기 어려워집니다.

C#

```
string[] names = { "Balto", "Vanya", "Dakota", "Samuel", "Koani", "Yiska",
"Yuma" };
string output = names[0] + ", " + names[1] + ", " + names[2] + ", " +
                names[3] + ", " + names[4] + ", " + names[5] + ", " +
                names[6];

output += "\n";
var date = DateTime.Now;
output += String.Format("It is {0:t} on {0:d}. The day of the week is {1}.",
                        date, date.DayOfWeek);
Console.WriteLine(output);
// The example displays the following output:
//     Balto, Vanya, Dakota, Samuel, Koani, Yiska, Yuma
//     It is 10:29 AM on 1/8/2018. The day of the week is Monday.
```

반면, 다음 예제에서 보간된 문자열을 사용하면 문자열 연결 문과 이전 예제의 Format 메서드 호출보다 훨씬 더 명확하고 간결한 코드를 생성합니다.

C#

```
string[] names = { "Balto", "Vanya", "Dakota", "Samuel", "Koani", "Yiska",
"Yuma" };
string output = $"{names[0]}, {names[1]}, {names[2]}, {names[3]},
{names[4]}, " +
                $"{names[5]}, {names[6]}";

var date = DateTime.Now;
output += $"\nIt is {date:t} on {date:d}. The day of the week is
{date.DayOfWeek}.";
Console.WriteLine(output);
// The example displays the following output:
//     Balto, Vanya, Dakota, Samuel, Koani, Yiska, Yuma
//     It is 10:29 AM on 1/8/2018. The day of the week is Monday.
```

미리 정의된 형식 문자열은 어디에서 찾을 수 있나요?

- 모든 정수 및 부동 소수점 형식은 표준 숫자 서식 문자열 사용자 지정 숫자 서식 문자열 참조하세요.
- 날짜 및 시간 값은 표준 날짜 및 시간 형식 문자열 사용자 지정 날짜 및 시간 서식 문자열 참조하세요.
- 열거형 값은 열거형 형식 문자열참조하세요.

- TimeSpan 값은 표준 TimeSpan 형식 문자열 및 사용자 지정 TimeSpan 형식 문자열를 참조하세요.
- Guid 값은 Guid.ToString(String) 참조 페이지의 주의 섹션을 참조하세요.

형식 항목을 대체하는 결과 문자열의 맞춤을 제어하려면 어떻게 해야 하나요?

형식 항목의 일반적인 구문은 다음과 같습니다.

```txt
{index[,alignment][: formatString]}
```

여기서 *맞춤*은 필드 너비를 정의하는 부호 있는 정수입니다. 이 값이 음수이면 필드의 텍스트가 왼쪽 맞춤됩니다. 양수이면 텍스트가 오른쪽 맞춤됩니다.

소수 구분 기호 뒤의 자릿수를 제어하려면 어떻게 해야 하나요?

모든 표준 숫자 형식 문자열은 "D"(정수에만 사용), "G", "R", 및 "X"를 제외하고, 결과 문자열의 소수 자릿수를 정의하는 정밀도 지정자를 허용합니다. 다음 예제에서는 표준 숫자 형식 문자열을 사용하여 결과 문자열의 소수 자릿수를 제어합니다.

```C#
object[] values = { 1603, 1794.68235, 15436.14 };
string result;
foreach (var value in values)
{
    result = String.Format("{0,12:C2}   {0,12:E3}   {0,12:F4}   {0,12:N3}
{1,12:P2}\n",
                           Convert.ToDouble(value), Convert.ToDouble(value)
/ 10000);
    Console.WriteLine(result);
}
// The example displays output like the following:
//       $1,603.00     1.603E+003      1603.0000      1,603.000       16.03
%
//
//       $1,794.68     1.795E+003      1794.6824      1,794.682       17.95
%
//
//      $15,436.14     1.544E+004     15436.1400     15,436.140      154.36
%
```

사용자 지정 숫자 서식 문자열사용하는 경우 다음 예제와 같이 "0" 형식 지정자를 사용하여 결과 문자열의 소수 자릿수를 제어합니다.

C#

```
decimal value = 16309.5436m;
string result = String.Format("{0,12:#.00000} {0,12:0,000.00}
{0,12:000.00#}",
                              value);
Console.WriteLine(result);
// The example displays the following output:
//        16309.54360    16,309.54    16309.544
```

정수 자릿수를 제어하려면 어떻게 해야 하나요?

기본적으로 서식 지정 작업에는 0이 아닌 정수만 표시됩니다. 정수를 서식 지정할 때, "D" 및 "X" 표준 서식 문자열과 함께 정밀도 지정자를 사용하여 자릿수를 제어할 수 있습니다.

C#

```
int value = 1326;
string result = String.Format("{0,10:D6} {0,10:X8}", value);
Console.WriteLine(result);
// The example displays the following output:
//     001326   0000052E
```

다음 예제와 같이 "0" 사용자 지정 숫자 형식 지정자사용하여 정수 또는 부동 소수점 숫자를 선행 0으로 채워 지정된 수의 정수로 결과 문자열을 생성할 수 있습니다.

C#

```
int value = 16342;
string result = String.Format("{0,18:00000000} {0,18:00000000.000}
{0,18:000,0000,000.0}",
                              value);
Console.WriteLine(result);
// The example displays the following output:
//           00016342       00016342.000    0,000,016,342.0
```

서식 목록에 포함할 수 있는 항목은 몇 개입니까?

실질적인 제한은 없습니다. Format(IFormatProvider, String, Object[]) 메서드의 두 번째 매개 변수는 ParamArrayAttribute 특성으로 태그가 지정되므로 구분된 목록 또는 개체 배열을 서식 목록으로 포함할 수 있습니다.

결과 문자열에 리터럴 중괄호("{" 및 "}")를 포함하려면 어떻게 해야 하나요?

예를 들어 다음 메서드 호출이 FormatException 예외를 throw하지 않도록 하려면 어떻게 해야 할까요?

```
C#

result = String.Format("The text has {0} '{' characters and {1} '}'
characters.",
                       nOpen, nClose);
```

단일 여는 중괄호 또는 닫는 중괄호는 항상 형식 항목의 시작 또는 끝으로 해석됩니다. 문자 그대로 해석되려면 이스케이프되어야 합니다. 다음 메서드 호출과 같이 "{" 및 "}" 대신 다른 중괄호("{{" 및 "}}")를 추가하여 중괄호를 이스케이프합니다.

```
C#

string result;
int nOpen = 1;
int nClose = 2;
result = String.Format("The text has {0} '{{' characters and {1} '}}'
characters.",
                       nOpen, nClose);
Console.WriteLine(result);
```

그러나 이스케이프된 중괄호조차도 쉽게 잘못 해석됩니다. 다음 예제와 같이 서식 목록에 중괄호를 포함하고 서식 항목을 사용하여 결과 문자열에 삽입하는 것이 좋습니다.

```
C#

string result;
int nOpen = 1;
int nClose = 2;
result = String.Format("The text has {0} '{1}' characters and {2} '{3}'
characters.",
                       nOpen, "{", nClose, "}");
Console.WriteLine(result);
```

String.Format 메서드를 호출하면 FormatException이 throw되는 이유는 무엇인가요?

예외의 가장 일반적인 원인은 서식 항목의 인덱스가 서식 목록의 개체에 해당하지 않는다는 것입니다. 일반적으로 형식 항목의 인덱스를 잘못 열거했거나 서식 목록에 개체를 포함하는 것을 잊어버렸습니다. 이스케이프되지 않은 왼쪽 또는 오른쪽 중괄호 문자를 포함하

려고 하면 FormatException예외가 발생합니다. 경우에 따라 예외는 오타의 결과입니다. 예를 들어 일반적인 실수는 "{"(왼쪽 중괄호) 대신 "["(왼쪽 대괄호)를 잘못 입력하는 것입니다.

Format(System.IFormatProvider,System.String,System.Object[]) 메서드가 매개 변수 배열을 지원하는 경우 배열을 사용할 때 내 코드가 예외를 throw하는 이유는 무엇인가요?

예를 들어 다음 코드는 FormatException 예외를 throw합니다.

C#

```
Random rnd = new Random();
int[] numbers = new int[4];
int total = 0;
for (int ctr = 0; ctr <= 2; ctr++)
{
    int number = rnd.Next(1001);
    numbers[ctr] = number;
    total += number;
}
numbers[3] = total;
Console.WriteLine($"{numbers} + {1} + {2} = {3}");
```

컴파일러 오버로드 해석의 문제입니다. 컴파일러는 정수 배열을 개체 배열로 변환할 수 없으므로 정수 배열을 단일 인수로 처리하므로 Format(String, Object) 메서드를 호출합니다. 형식 목록에는 하나의 항목만 있는데 4개의 형식 항목이 있어 예외가 발생합니다.

Visual Basic과 C#은 모두 정수 배열을 개체 배열로 변환할 수 없으므로 Format(String, Object[]) 메서드를 호출하기 전에 직접 변환을 수행해야 합니다. 다음 예제에서는 하나의 구현을 제공합니다.

C#

```
Random rnd = new Random();
int[] numbers = new int[4];
int total = 0;
for (int ctr = 0; ctr <= 2; ctr++)
{
    int number = rnd.Next(1001);
    numbers[ctr] = number;
    total += number;
}
numbers[3] = total;
object[] values = new object[numbers.Length];
numbers.CopyTo(values, 0);
Console.WriteLine($"{values} + {1} + {2} = {3}");
```

System.String.Intern 메서드

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

공용 언어 런타임은 각 고유 문자열 값에 대한 단일 참조를 보유하는 *인턴 풀*이라는 테이블을 유지 관리합니다. 이 메서드는 Intern 인턴 풀을 사용하여 값 `str`과 같은 문자열을 검색합니다. 이러한 문자열이 없으면 참조 `str` 가 풀에 추가되고 해당 참조가 반환됩니다. 반면 요청된 문자열이 IsInterned(String) 인턴 풀에 없는 경우 메서드는 null 참조를 반환합니다.

인턴 풀은 런타임에서 문자열 스토리지를 절약하는 데 사용할 수 있습니다. 그러나 문자열 리터럴의 자동 인턴팅은 보장되지 않습니다. 어셈블리가 컴파일되고 실행되는 방법에 따라 일부 리터럴이 풀에 추가되지 않을 수 있습니다.

다음 예제에서 문자열 `s1` 의 값은 "MyTest"입니다. 클래스는 System.Text.StringBuilder 값이 같은 `s1`새 문자열 개체를 생성합니다. 해당 문자열에 대한 참조가 .에 `s2`할당됩니다. 메서드는 Intern 값 `s2`이 같은 문자열을 검색합니다. 이미 인턴된 경우(예: 어셈블리에 문자열 리터럴 인턴팅이 필요하기 때문에) 메서드는 `s1`과 동일한 참조를 반환하며, 이 참조는 `s3`에 할당됩니다. 그리고 `s1`와 `s3`는 같음으로 비교됩니다. 그렇지 않으면 `s2`에 대한 새로운 인턴된 항목이 만들어지고 `s3`에 할당되며, `s1`와 `s3`는 같지 않음을 비교합니다. `s1`와 `s2`는 두 경우 모두 서로 다른 개체를 참조하므로 같지 않다고 비교합니다.

C#

```
string s1 = "MyTest";
string s2 = new StringBuilder().Append("My").Append("Test").ToString();
string s3 = String.Intern(s2);
Console.WriteLine((Object)s2==(Object)s1); // Different references.
Console.WriteLine((Object)s3==(Object)s1); // The same reference.
```

성능 고려 사항

애플리케이션에서 할당하는 총 메모리 양을 줄이려는 경우 문자열 인턴팅에는 두 가지 원치 않는 부작용이 있음을 명심하세요. 첫째, 인턴 String 된 개체에 할당된 메모리는 CLR(공용 언어 런타임)이 종료될 때까지 해제될 가능성이 없습니다. 그 이유는 애플리케이션 또는 애플리케이션 도메인이 종료된 후에도 CLR이 인턴 String 된 개체에 대한 참조를 유지할 수 있기 때문입니다. 둘째, 문자열을 인터닝하려면 먼저 문자열을 만들어야 합니다. String 개체에서 사용하는 메모리는 결국 가비지 수집될 것이지만, 여전히 할당되어야 합니다.

열거형 멤버는 CompilationRelaxations.NoStringInterning 문자열 리터럴 인턴링을 요구하지 않는 어셈블리를 표시합니다. 기본적으로 C# 컴파일러는 성능 향상을 위해 각 어셈블리마다 CompilationRelaxationsAttribute를 NoStringInterning 플래그와 함께 생성합니다. 이는 문자열 리터럴이 인턴 풀에 추가될 것을 보장하지 않음을 의미합니다. NoStringInterning에 CompilationRelaxationsAttribute 특성을 사용하여 어셈블리를 사용자 지정할 수 있습니다.

네이티브 AOT를 사용하여 앱을 게시하는 경우 해제 NoStringInterning 는 지원되지 않습니다. 네이티브 AOT에서는 문자열 리터럴이 인턴 풀에 추가되도록 보장되지 않으므로 Intern 소스 코드에서 리터럴로 보이는 문자열에 대한 일치 항목을 찾을 수 없습니다.

Last updated on 2026. 03. 31.

System.String.IsNullOrEmpty 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

IsNullOrEmpty는 String이 `null`인지 아니면 그 값이 String.Empty인지 동시에 테스트할 수 있는 편리한 메서드입니다. 다음 코드와 동일합니다.

C#

```
bool TestForNullOrEmpty(string s)
{
    bool result;
    result = s == null || s == string.Empty;
    return result;
}

string s1 = null;
string s2 = "";
Console.WriteLine(TestForNullOrEmpty(s1));
Console.WriteLine(TestForNullOrEmpty(s2));

// The example displays the following output:
//    True
//    True
```

메서드를 IsNullOrWhiteSpace 사용하여 문자열 `null`이 있는지, 문자열 값 String.Empty인지 또는 공백 문자로만 구성되어 있는지 테스트할 수 있습니다.

null 문자열이란?

문자열은 C++ 또는 Visual Basic에서 값이 할당되지 않았거나 명시적으로 `null` 값이 할당된 경우 `null`입니다. 복합 서식 지정 기능은 다음 예제와 같이 null 문자열을 정상적으로 처리할 수 있지만, 그 멤버 중 하나를 호출하려고 할 때 NullReferenceException이 발생합니다.

C#

```
  String s = null;

  Console.WriteLine($"The value of the string is '{s}'");

  try
  {
      Console.WriteLine($"String length is {s.Length}");
  }
  catch (NullReferenceException e)
  {
```

```
        Console.WriteLine(e.Message);
    }

    // The example displays the following output:
    //     The value of the string is ''
    //     Object reference not set to an instance of an object.
```

빈 문자열이란?

문자열이 빈 문자열("") 또는 String.Empty.을 명시적으로 할당한 경우 비어 있습니다. 빈 문자열의 값은 0입니다 Length . 다음 예제에서는 빈 문자열을 만들고 해당 값과 길이를 표시합니다.

C#

```
String s = "";
Console.WriteLine($"The length of '{s}' is {s.Length}.");

// The example displays the following output:
//       The length of '' is 0.
```

System.Char 구조체

아티클 • 2024. 01. 08.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

구조체는 Char UTF-16 인코딩을 사용하여 유니코드 코드 포인트를 나타냅니다. 개체의 Char 값은 16비트 숫자(서수) 값입니다.

유니코드, 스칼라 값, 코드 포인트, 서로게이트 쌍, UTF-16 및 Rune 형식에 익숙하지 않은 경우 .NET의 문자 인코딩 소개를 참조하세요.

이 문서에서는 개체와 문자 간의 Char 관계를 살펴보고 인스턴스와 함께 Char 수행되는 몇 가지 일반적인 작업에 대해 설명합니다. 이러한 작업 중 일부를 수행하기 위한 대안으로 .NET Core 3.0에 도입된 형식을 Char 고려하는 Rune 것이 좋습니다.

Char 개체, 유니코드 문자 및 문자열

String 개체는 텍스트 문자열을 나타내는 구조체의 Char 순차적 컬렉션입니다. 대부분의 유니코드 문자는 단일 Char 개체로 나타낼 수 있지만 기본 문자, 서로게이트 쌍 및/또는 결합 문자 시퀀스로 인코딩된 문자는 여러 Char 개체로 표시됩니다. 이러한 이유로 개체의 Char 구조 String 체가 반드시 단일 유니코드 문자와 동일하지는 않습니다.

여러 16비트 코드 단위는 다음과 같은 경우 단일 유니코드 문자를 나타내는 데 사용됩니다.

- 문자 모양- 단일 문자 또는 기본 문자와 하나 이상의 결합 문자로 구성될 수 있습니다. 예를 들어 ä 문자는 코드 단위가 U+0061인 개체와 Char 코드 단위가 U+0308인 개체로 표시됩니다Char. (ä 문자는 U+00E4의 코드 단위가 있는 단일 Char 개체로 정의할 수도 있습니다.) 다음 예제에서는 ä 문자가 두 Char 개의 개체로 구성되어 있음을 보여 줍니다.

```
C#

using System;
using System.IO;

public class Example1
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter("chars1.txt");
        char[] chars = { '\u0061', '\u0308' };
        string strng = new String(chars);
        sw.WriteLine(strng);
        sw.Close();
```

```
    }
}
// The example produces the following output:
//       ä
```

- 유니코드 BMP(Basic Multilingual Plane) 외부의 문자입니다. 유니코드는 평면 0을 나타내는 BMP 외에 16개의 평면을 지원합니다. 유니코드 코드 포인트는 평면을 포함하는 21비트 값으로 UTF-32로 표시됩니다. 예를 들어 U+1D160은 MUSICAL SYMBOL EIGHTH NOTE 문자를 나타냅니다. UTF-16 인코딩에는 16비트만 있으므로 BMP 외부의 문자는 UTF-16의 서로게이트 쌍으로 표시됩니다. 다음 예제에서는 U+1D160에 해당하는 UTF-32(MUSICAL SYMBOL EIGHTH NOTE 문자)가 U+D834 U+DD60임을 보여 줍니다. U+D834는 상위 서로게이트입니다. 상위 서로게이트 범위는 U+D800부터 U+DBFF까지입니다. U+DD60은 낮은 서로게이트입니다. 하위 서로게이트 범위는 U+DC00부터 U+DFFF까지입니다.

C#

```
using System;
using System.IO;

public class Example3
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\chars2.txt");
        int utf32 = 0x1D160;
        string surrogate = Char.ConvertFromUtf32(utf32);
        sw.WriteLine("U+{0:X6} UTF-32 = {1} ({2}) UTF-16",
                     utf32, surrogate, ShowCodePoints(surrogate));
        sw.Close();
    }

    private static string ShowCodePoints(string value)
    {
        string retval = null;
        foreach (var ch in value)
            retval += String.Format("U+{0:X4} ", Convert.ToUInt16(ch));

        return retval.Trim();
    }
}
// The example produces the following output:
//       U+01D160 UTF-32 = ð (U+D834 U+DD60) UTF-16
```

문자 및 문자 범주

각 유니코드 문자 또는 유효한 서로게이트 쌍은 유니코드 범주에 속합니다. .NET에서 유니코드 범주는 열거형의 UnicodeCategory 멤버로 표시되며, 예를 들어 , UnicodeCategory.LowercaseLetter및 UnicodeCategory.SpaceSeparator등의 값을 UnicodeCategory.CurrencySymbol포함합니다.

문자의 유니코드 범주를 확인하려면 메서드를 호출합니다 GetUnicodeCategory . 예를 들어 다음 예제에서는 문자열에서 GetUnicodeCategory 각 문자의 유니코드 범주를 표시하도록 호출합니다. 이 예제는 인스턴스에 String 서로게이트 쌍이 없는 경우에만 올바르게 작동합니다.

C#

```
using System;
using System.Globalization;

class Example
{
   public static void Main()
   {
      // Define a string with a variety of character categories.
      String s = "The red car drove down the long, narrow, secluded road.";
      // Determine the category of each character.
      foreach (var ch in s)
         Console.WriteLine("'{0}': {1}", ch, Char.GetUnicodeCategory(ch));
   }
}
// The example displays the following output:
//       'T': UppercaseLetter
//       'h': LowercaseLetter
//       'e': LowercaseLetter
//       ' ': SpaceSeparator
//       'r': LowercaseLetter
//       'e': LowercaseLetter
//       'd': LowercaseLetter
//       ' ': SpaceSeparator
//       'c': LowercaseLetter
//       'a': LowercaseLetter
//       'r': LowercaseLetter
//       ' ': SpaceSeparator
//       'd': LowercaseLetter
//       'r': LowercaseLetter
//       'o': LowercaseLetter
//       'v': LowercaseLetter
//       'e': LowercaseLetter
//       ' ': SpaceSeparator
//       'd': LowercaseLetter
//       'o': LowercaseLetter
//       'w': LowercaseLetter
//       'n': LowercaseLetter
//       ' ': SpaceSeparator
//       't': LowercaseLetter
//       'h': LowercaseLetter
```

```
//      'e': LowercaseLetter
//      ' ': SpaceSeparator
//      'l': LowercaseLetter
//      'o': LowercaseLetter
//      'n': LowercaseLetter
//      'g': LowercaseLetter
//      ',': OtherPunctuation
//      ' ': SpaceSeparator
//      'n': LowercaseLetter
//      'a': LowercaseLetter
//      'r': LowercaseLetter
//      'r': LowercaseLetter
//      'o': LowercaseLetter
//      'w': LowercaseLetter
//      ',': OtherPunctuation
//      ' ': SpaceSeparator
//      's': LowercaseLetter
//      'e': LowercaseLetter
//      'c': LowercaseLetter
//      'l': LowercaseLetter
//      'u': LowercaseLetter
//      'd': LowercaseLetter
//      'e': LowercaseLetter
//      'd': LowercaseLetter
//      ' ': SpaceSeparator
//      'r': LowercaseLetter
//      'o': LowercaseLetter
//      'a': LowercaseLetter
//      'd': LowercaseLetter
//      '.': OtherPunctuation
```

내부적으로 ASCII 범위(U+0000~ U+00FF)를 벗어난 문자의 GetUnicodeCategory 경우 메서드는 클래스에서 보고한 CharUnicodeInfo 유니코드 범주에 따라 달라집니다. .NET Framework 4.6.2부터 유니코드 문자는 유니코드 표준 버전 8.0.0을 기반으로 분류됩니다. .NET Framework 4에서 .NET Framework 4.6.1로의 .NET Framework 버전에서는 유니코드 표준 버전 6.3.0을 기반으로 분류됩니다.

문자 및 텍스트 요소

단일 문자는 여러 Char 개체로 나타낼 수 있으므로 개별 Char 개체로 작업하는 것이 항상 의미가 있는 것은 아닙니다. 예를 들어 다음 예제에서는 0부터 9까지의 에게 해를 나타내는 유니코드 코드 요소를 UTF-16으로 인코딩된 코드 단위로 변환합니다. 개체와 문자가 잘못 동일 Char 하기 때문에 결과 문자열에 20자가 있다고 부정확하게 보고합니다.

C#

```
using System;

public class Example5
```

```
{
    public static void Main()
    {
        string result = String.Empty;
        for (int ctr = 0x10107; ctr <= 0x10110; ctr++)  // Range of Aegean
numbers.
            result += Char.ConvertFromUtf32(ctr);

        Console.WriteLine("The string contains {0} characters.",
result.Length);
    }
}
// The example displays the following output:
//     The string contains 20 characters.
```

개체가 단일 문자를 나타낸다는 Char 가정이 없도록 다음을 수행할 수 있습니다.

- 개별 문자로 String 작업하는 대신 개체 전체를 사용하여 언어 콘텐츠를 나타내고 분석할 수 있습니다.
- 다음 예제와 같이 사용할 String.EnumerateRunes 수 있습니다.

  C#

  ```
  int CountLetters(string s)
  {
      int letterCount = 0;

      foreach (Rune rune in s.EnumerateRunes())
      {
          if (Rune.IsLetter(rune))
          { letterCount++; }
      }

      return letterCount;
  }
  ```

- 클래스를 StringInfo 사용하여 개별 Char 개체 대신 텍스트 요소를 사용할 수 있습니다. 다음 예제에서는 개체를 StringInfo 사용하여 에게 해 숫자 0에서 9로 구성된 문자열의 텍스트 요소 수를 계산합니다. 서로게이트 쌍을 단일 문자로 간주하므로 문자열에 10자가 포함되어 있음을 올바르게 보고합니다.

  C#

  ```
  using System;
  using System.Globalization;

  public class Example4
  {
      public static void Main()
  ```

```
    {
        string result = String.Empty;
        for (int ctr = 0x10107; ctr <= 0x10110; ctr++)  // Range of
Aegean numbers.
            result += Char.ConvertFromUtf32(ctr);

        StringInfo si = new StringInfo(result);
        Console.WriteLine("The string contains {0} characters.",
                          si.LengthInTextElements);
    }
}
// The example displays the following output:
//       The string contains 10 characters.
```

- 문자열에 하나 이상의 결합 문자가 있는 기본 문자가 포함된 경우 메서드를 호출 String.Normalize 하여 부분 문자열을 단일 UTF-16 인코딩 코드 단위로 변환할 수 있습니다. 다음 예제에서는 메서드를 호출 String.Normalize 하여 기본 문자 U+0061(LATIN SMALL LETTER A)을 변환하고 U+0308(DIAERESIS 결합)을 U+00E4(LATIN SMALL LETTER A WITH DIAERESIS)로 결합합니다.

C#

```
using System;

public class Example2
{
    public static void Main()
    {
        string combining = "\u0061\u0308";
        ShowString(combining);

        string normalized = combining.Normalize();
        ShowString(normalized);
    }

    private static void ShowString(string s)
    {
        Console.Write("Length of string: {0} (", s.Length);
        for (int ctr = 0; ctr < s.Length; ctr++)
        {
            Console.Write("U+{0:X4}", Convert.ToUInt16(s[ctr]));
            if (ctr != s.Length - 1) Console.Write(" ");
        }
        Console.WriteLine(")\n");
    }
}
// The example displays the following output:
//       Length of string: 2 (U+0061 U+0308)
//
//       Length of string: 1 (U+00E4)
```

일반적인 작업

구조체는 Char 개체를 비교 Char 하고, 현재 Char 개체의 값을 다른 형식의 개체로 변환하고, 개체의 유니코드 범주를 결정하는 메서드를 Char 제공합니다.

테이블 확장

원하는 작업	이러한 `System.Char` 메서드 사용
개체 비교 Char	CompareTo 및 Equals
코드 지점을 문자열로 변환	ConvertFromUtf32 형식도 참조하세요 Rune .
Char 개체 또는 서로게이트 개체 쌍 Char 을 코드 포인트로 변환	단일 문자의 경우: Convert.ToInt32(Char) 서로게이트 쌍 또는 문자열의 문자: Char.ConvertToUtf32 형식도 참조하세요 Rune .
문자의 유니코드 범주 가져오기	GetUnicodeCategory Rune.GetUnicodeCategory을 참조하세요.
문자가 숫자, 문자, 문장 부호, 컨트롤 문자 등과 같은 특정 유니코드 범주에 있는지 확인	IsControl, IsDigit, IsHighSurrogate, IsLetter, IsLetterOrDigit, IsLower, , IsLowSurrogate, IsNumber, IsSeparatorIsPunctuation, IsSurrogateIsSurrogatePair, IsSymbolIsUpper및IsWhiteSpace 형식에 대한 해당 메서드도 참조하세요 Rune .
Char 숫자를 나타내는 개체를 숫자 값 형식으로 변환	GetNumericValue Rune.GetNumericValue을 참조하세요.
문자열의 문자를 개체로 Char 변환	Parse 및 TryParse
개체를 Char 개체로 String 변환	ToString
개체의 대/소문자 Char 변경	ToLower, ToLowerInvariant, ToUpper 및 ToUpperInvariant 형식에 대한 해당 메서드도 참조하세요 Rune .

Char 값 및 interop

유니코드 UTF-16으로 인코딩된 코드 단위로 표현되는 관리 Char 되는 형식이 관리되지 않는 코드에 전달되면 interop 마샬러는 기본적으로 문자 집합을 ANSI로 변환합니다. 플랫폼 호출 선언에 특성을 적용하고 StructLayoutAttribute COM interop 선언에 특성을 적용 DllImportAttribute 하여 마샬링된 Char 형식이 사용하는 문자 집합을 제어할 수 있습니다.

System.StringComparer 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스에서 StringComparer 파생된 개체는 대/소문자 및 문화권별 비교 규칙을 모두 고려하는 문자열 기반의 비교, 동등성, 및 해시 코드 연산을 구현합니다. 클래스를 StringComparer 사용하여 형식별 비교를 만들어 제네릭 컬렉션의 요소를 정렬할 수 있습니다. Hashtable, Dictionary<TKey,TValue>, SortedList, 및 SortedList<TKey,TValue>와 같은 클래스는 정렬 용도로 StringComparer 클래스를 사용합니다.

클래스에서 나타내는 StringComparer 비교 작업은 대/소문자를 구분하거나 대/소문자를 구분하지 않는 것으로 정의되며 단어(문화권 구분) 또는 서수(문화권을 구분하지 않는) 비교 규칙을 사용합니다. 단어 및 서수 비교 규칙에 대한 자세한 내용은 다음을 참조하세요 System.Globalization.CompareOptions.

> **참고**
>
> 정렬 가중치 테이블의 최신 버전인 **기본 유니코드 데이터 정렬 요소** 테이블을 다운로드할 수 있습니다. 정렬 가중치 테이블의 특정 버전은 시스템에 설치된 **유니코드용 International Components** 라이브러리의 버전에 따라 달라집니다. ICU 버전 및 구현하는 유니코드 버전에 대한 자세한 내용은 **ICU 다운로드를** 참조하세요.
>
> Windows의 .NET Framework의 경우 정렬 및 비교 작업에 사용되는 문자 **가중치**에 대한 정보가 포함된 텍스트 파일 집합인 정렬 가중치 테이블을 다운로드할 수 있습니다.

구현된 속성

모순된 것처럼 보이는 클래스 속성을 사용하는 StringComparer 방법에 대해 혼동할 수 있습니다. 클래스는 StringComparer로 선언됩니다 (`abstract`는 Visual Basic에서 `MustInherit`). 즉, StringComparer 클래스에서 파생된 클래스의 개체에서만 해당 멤버를 호출할 수 있습니다. 모순은 StringComparer 클래스의 각 속성이 `static`으로 선언된다는 점입니다(`Shared`는 Visual Basic에서). 즉, 먼저 파생 클래스를 만들지 않고도 속성을 호출할 수 있다는 것입니다.

속성을 직접 호출할 수 있습니다. 왜냐하면 각 속성이 실제로 StringComparer 클래스에서 파생된 익명 클래스의 인스턴스를 반환하기 때문입니다. 따라서 각 속성 값의 형식은 StringComparer익명 클래스 자체의 형식이 아니라 익명 클래스의 기본 클래스입니다. 각 StringComparer 클래스 속성은 미리 정의된 대/소문자 및 비교 규칙을 지원하는 StringComparer 개체를 반환합니다.

System.Text.Encoding 클래스

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스는 Encoding 문자 인코딩을 나타냅니다.

인코딩은 유니코드 문자 집합을 바이트 시퀀스로 변환하는 프로세스입니다. 반면 디코딩은 인코딩된 바이트 시퀀스를 유니코드 문자 집합으로 변환하는 프로세스입니다. UTF(유니코드 변환 형식) 및 지원되는 Encoding기타 인코딩에 대한 자세한 내용은 .NET의 문자 인코딩을 참조하세요.

Encoding 는 바이트 배열과 같은 임의의 이진 데이터 대신 유니코드 문자에서 작동하기 위한 것입니다. 임의의 이진 데이터를 텍스트로 인코딩해야 하는 경우 uuencode와 같은 프로토콜을 사용해야 합니다. 이 프로토콜은 다음과 같은 Convert.ToBase64CharArray메서드에 의해 구현됩니다.

.NET은 현재 유니코드 인코딩 및 기타 인코딩을 지원하기 위해 클래스의 Encoding 다음 구현을 제공합니다.

- ASCIIEncoding 유니코드 문자를 단일 7비트 ASCII 문자로 인코딩합니다. 이 인코딩은 U+0000과 U+007F 사이의 문자 값만 지원합니다. 코드 페이지 20127. 속성을 통해 ASCII에서도 사용할 수 있습니다.

- UTF7Encoding UTF-7 인코딩을 사용하여 유니코드 문자를 인코딩합니다. 이 인코딩은 모든 유니코드 문자 값을 지원합니다. 코드 페이지 65000. 속성에서도 UTF7를 통해 사용할 수 있습니다.

- UTF8Encoding UTF-8 인코딩을 사용하여 유니코드 문자를 인코딩합니다. 이 인코딩은 모든 유니코드 문자 값을 지원합니다. 코드 페이지 65001. UTF8 속성을 통해서도 사용할 수 있습니다.

- UnicodeEncoding UTF-16 인코딩을 사용하여 유니코드 문자를 인코딩합니다. 리틀 엔디안 및 빅 엔디안 바이트 순서가 모두 지원됩니다. Unicode 속성과 BigEndianUnicode 속성을 통해서도 사용할 수 있습니다.

- UTF32Encoding UTF-32 인코딩을 사용하여 유니코드 문자를 인코딩합니다. little endian(코드 페이지 12000) 및 big endian(코드 페이지 12001) 바이트 주문이 모두 지원됩니다. UTF32 속성을 통해서도 사용할 수 있습니다.

Encoding 클래스는 주로 서로 다른 인코딩과 유니코드 간에 변환하기 위한 것입니다. 파생 유니코드 클래스 중 하나가 앱에 적합한 선택인 경우가 많습니다.

메서드를 GetEncoding 사용하여 다른 인코딩을 가져오고 메서드를 GetEncodings 호출하여 모든 인코딩 목록을 가져옵니다.

인코딩 목록

다음 표에서는 .NET에서 지원하는 인코딩을 나열합니다. 각 인코딩의 코드 페이지 번호와 인코딩의 EncodingInfo.Name 및 EncodingInfo.DisplayName 속성의 값을 나열합니다. **.NET Framework 지원**, **.NET Core 지원 또는 .NET5 이상 지원** 열의 확인 표시는 기본 플랫폼에 관계없이 해당 .NET 구현에서 코드 페이지가 기본적으로 지원됨을 나타냅니다. .NET Framework의 경우 테이블에 나열된 다른 인코딩의 가용성은 운영 체제에 따라 달라집니다. .NET Core 및 .NET 5 이상의 버전에서는 System.Text.CodePagesEncodingProvider 클래스를 사용하거나 System.Text.EncodingProvider 클래스에서 파생하여 다른 인코딩을 사용할 수 있습니다.

> **참고**
>
> 해당 속성이 EncodingInfo.Name 국제 표준에 해당하는 코드 페이지가 반드시 해당 표준을 완전히 준수하지는 않습니다.

테이블 확장

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
37	IBM037	IBM EBCDIC(US-Canada)			
437	IBM437	OEM 미국			
500	IBM500	IBM EBCDIC(International)			
708	ASMO-708	아랍어(ASMO 708)			
720	DOS-720	아랍어(DOS)			
737	ibm737	그리스어(DOS)			
775	ibm775	발틱 (DOS)			
850	ibm850	서유럽어(DOS)			
852	ibm852	중부 유럽(DOS)			
855	IBM855	OEM 키릴 문자			
857	ibm857	터키어(DOS)			
858	IBM00858	OEM 다국어 라틴어 I			
860	IBM860	포르투갈어(DOS)			

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
861	ibm861	아이슬란드어(DOS)			
862	DOS-862	히브리어(DOS)			
863	IBM863	프랑스어 캐나다어(DOS)			
864	IBM864	아랍어(864)			
865	IBM865	북유럽어(DOS)			
866	cp866	키릴 자모(DOS)			
869	ibm869	그리스어, 현대식(DOS)			
870	IBM870	IBM EBCDIC(다국어 라틴어-2)			
874	windows-874	태국어(Windows)			
875	cp875	IBM EBCDIC (그리스어 현대)			
932	shift_jis	일본어(Shift-JIS)			
936	gb2312	중국어 간체(GB2312)	✓		
949	ks_c_5601-1987	한국어			
950	big5	중국어 번체(Big5)			
1026	IBM1026	IBM EBCDIC(터키어 라틴어-5)			
1047	IBM01047	IBM Latin-1			
1140	IBM01140	IBM EBCDIC(미국-Canada-Euro)			
1141	IBM01141	IBM EBCDIC(Germany-Euro)			
1142	IBM01142	IBM EBCDIC(덴마크-Norway-Euro)			
1143	IBM01143	IBM EBCDIC(핀란드-Sweden-Euro)			
1144	IBM01144	IBM EBCDIC(Italy-Euro)			
1145	IBM01145	IBM EBCDIC(Spain-Euro)			

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
1146	IBM01146	IBM EBCDIC(UK-Euro)			
1147	IBM01147	IBM EBCDIC(France-Euro)			
1148	IBM01148	IBM EBCDIC(International-Euro)			
1149	IBM01149	IBM EBCDIC(Icelandic-Euro)			
1200	utf-16	유니코드	✓	✓	✓
1201	unicodeFFFE	유니코드(빅 엔디안)	✓	✓	✓
1250	windows-1250	중부 유럽(Windows)			
1251	windows-1251	키릴 자모(Windows)			
1252	Windows-1252	서유럽어(Windows)	✓		
1253	windows-1253	그리스어(Windows)			
1254	windows-1254	터키어(Windows)			
1255	windows-1255	히브리어(Windows)			
1256	windows-1256	아랍어(Windows)			
1257	windows-1257	발트(Windows)			
1258	windows-1258	베트남어(Windows)			
1361	Johab	한국어(조합)			
1만	매킨토시	서유럽어(Mac)			
10001	x-mac-japanese	일본어(Mac)			
10002	x-mac-chinesetrad	중국어 번체 (Mac)			
10003	x-mac-korean	한국어(Mac)	✓		
10004	x-mac-arabic	아랍어(Mac)			
10005	x-mac-hebrew	히브리어(Mac)			
10006	x-mac-greek	그리스어(Mac)			
10007	x-mac-키릴 문자	키릴 문자(Mac)			

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
10008	x-mac-chinesesimp	중국어 간체(Mac)	✓		
10010	x-mac-루마니아어	루마니아어(Mac)			
10017	x-mac-ukrainian	우크라이나어(Mac)			
10021	x-mac-thai	태국어(Mac)			
10029	x-mac-ce	중부 유럽(Mac)			
10079	x-mac-icelandic	아이슬란드어(Mac)			
10081	x-mac-터키어	터키어(Mac)			
10082	x-mac-크로아티아어	크로아티아어(Mac)			
12000	utf-32	유니코드(UTF-32)	✓	✓	✓
12001	utf-32BE	유니코드(UTF-32 빅 엔디안)	✓	✓	✓
20000	x-Chinese-CNS	중국어 번체(CNS)			
20001	x-cp20001	TCA 대만			
20002	x-Chinese-Eten	중국어 번체(에텐)			
20003	x-cp20003	대만 IBM5550			
20004	x-cp20004	TeleText 대만			
20005	x-cp20005	왕 대만			
20105	x-IA5	서유럽어(IA5)			
20106	x-IA5-German	독일어(IA5)			
20107	x-IA5-Swedish	스웨덴어(IA5)			
20108	x-IA5-Norwegian	노르웨이어(IA5)			
20127	us-ascii	US-ASCII	✓	✓	✓
20261	x-cp20261	T.61			
20269	x-cp20269	ISO-6937			
20273	IBM273	IBM EBCDIC(독일)			

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
20277	IBM277	IBM EBCDIC(Denmark-Norway)			
20278	IBM278	IBM EBCDIC(Finland-Sweden)			
20280	IBM280	IBM EBCDIC(이탈리아)			
20284	IBM284	IBM EBCDIC(스페인)			
20285	IBM285	IBM EBCDIC(영국)			
20290	IBM290	IBM EBCDIC(일본어 가타카나)			
20297	IBM297	IBM EBCDIC(프랑스)			
20420	IBM420	IBM EBCDIC(아랍어)			
20423	IBM423	IBM EBCDIC(그리스어)			
20424	IBM424	IBM EBCDIC(히브리어)			
20833	x-EBCDIC-KoreanExtended	IBM EBCDIC(한국어 확장)			
20838	IBM-Thai	IBM EBCDIC(태국어)			
20866	koi8-r	키릴 자모 (KOI8-R)			
20871	IBM871	IBM EBCDIC(아이슬란드어)			
20880	IBM880	IBM EBCDIC(러시아어 키릴 문자)			
20905	IBM905	IBM EBCDIC(터키어)			
20924	IBM00924	IBM Latin-1			
20932	EUC-JP	일본어(JIS 0208-1990 및 0212-1990)			
20936	x-cp20936	중국어 간체(GB2312-80)	✓		
20949	x-cp20949	한국어 Wansung	✓		
21025	cp1025	IBM EBCDIC (키릴 문자 Serbian-Bulgarian)			
21866	koi8-u	키릴 자모 (KOI8-U)			

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
28591	iso-8859-1	서유럽어(ISO)	✓	✓	✓
28592	iso-8859-2	중앙 유럽(ISO)			
28593	iso-8859-3	라틴어 3(ISO)			
28594	iso-8859-4	발트어(ISO)			
28595	iso-8859-5	키릴 자모(ISO)			
28596	iso-8859-6	아랍어(ISO)			
28597	iso-8859-7	그리스어(ISO)			
28598	iso-8859-8	히브리어(ISO-Visual)	✓		
28599	iso-8859-9	터키어(ISO)			
28603	iso-8859-13	에스토니아어(ISO)			
28605	iso-8859-15	라틴어 9(ISO)			
29001	x-Europa	에우로파			
38598	iso-8859-8-i	히브리어(ISO-Logical)	✓		
50220	iso-2022-jp	일본어(JIS)	✓		
50221	csISO2022JP	일본어(JIS-Allow 1바이트 가나)	✓		
50222	iso-2022-jp	일본어(JIS-Allow 1바이트 가나 - SO/SI)	✓		
50225	iso-2022-kr	한국어(ISO)	✓		
50227	x-cp50227	중국어 간체(ISO-2022)	✓		
51932	euc-jp	일본어(EUC)	✓		
51936	EUC-CN	중국어 간체(EUC)	✓		
51949	euc-kr	한국어(EUC)	✓		
52,936	hz-gb-2312	중국어 간체(HZ)	✓		
54936	GB18030	중국어 간체(GB18030)	✓		
57002	x-iscii-de	ISCII 데바나가리	✓		

코드 페이지	이름	표시 이름	.NET Framework 지원	.NET Core 지원	.NET 5 이상 지원
57003	x-iscii-be	ISCII 벵골어	✓		
57004	x-iscii-ta	ISCII 타밀어	✓		
57005	x-iscii-te	ISCII 텔루구어	✓		
57006	x-iscii-as	ISCII 아삼어	✓		
57007	x-iscii-or	ISCII 오리야	✓		
57008	x-iscii-ka	ISCII Kannada	✓		
57009	x-iscii-ma	ISCII 말라얄람	✓		
57010	x-iscii-gu	ISCII 구자라트어	✓		
57011	x-iscii-pa	ISCII 펀자브어	✓		
65000	utf-7	유니코드(UTF-7)	✓	✓	
65001	utf-8	유니코드(UTF-8)	✓	✓	✓

다음 예제에서는 GetEncoding(Int32) 메서드 및 GetEncoding(String) 메서드를 호출하여 그리스어(Windows) 코드 페이지 인코딩을 가져옵니다. 메서드 호출에서 반환된 개체를 비교하여 Encoding 동일한 것을 표시한 다음, 맵은 그리스어 알파벳의 각 문자에 대한 유니코드 코드 포인트와 해당 코드 페이지 값을 표시합니다.

C#

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.GetEncoding(1253);
      Encoding altEnc = Encoding.GetEncoding("windows-1253");
      Console.WriteLine($"{enc.EncodingName} = Code Page {altEnc.CodePage}:
{enc.Equals(altEnc)}");
      string greekAlphabet = "Α α Β β Γ γ Δ δ Ε ε Ζ ζ Η η " +
                             "Θ θ Ι ι Κ κ Λ λ Μ μ Ν ν Ξ ξ " +
                             "Ο ο Π π Ρ ρ Σ σ ς Τ τ Υ υ " +
                             "Φ ϕ Χ χ Ψ ψ Ω ω";
      Console.OutputEncoding = Encoding.UTF8;
      byte[] bytes = enc.GetBytes(greekAlphabet);
      Console.WriteLine("{0,-12} {1,20} {2,20:X2}", "Character",
                        "Unicode Code Point", "Code Page 1253");
```

```
      for (int ctr = 0; ctr < bytes.Length; ctr++) {
         if (greekAlphabet[ctr].Equals(' '))
            continue;

         Console.WriteLine("{0,-12} {1,20} {2,20:X2}", greekAlphabet[ctr],
                           GetCodePoint(greekAlphabet[ctr]), bytes[ctr]);
      }
   }

   private static string GetCodePoint(char ch)
   {
      string retVal = "u+";
      byte[] bytes = Encoding.Unicode.GetBytes(ch.ToString());
      for (int ctr = bytes.Length - 1; ctr >= 0; ctr--)
         retVal += bytes[ctr].ToString("X2");

      return retVal;
   }
}
// The example displays the following output:
//       Character      Unicode Code Point       Code Page 1253
//       Α                         u+0391                   C1
//       α                         u+03B1                   E1
//       Β                         u+0392                   C2
//       β                         u+03B2                   E2
//       Γ                         u+0393                   C3
//       γ                         u+03B3                   E3
//       Δ                         u+0394                   C4
//       δ                         u+03B4                   E4
//       Ε                         u+0395                   C5
//       ε                         u+03B5                   E5
//       Ζ                         u+0396                   C6
//       ζ                         u+03B6                   E6
//       Η                         u+0397                   C7
//       η                         u+03B7                   E7
//       Θ                         u+0398                   C8
//       θ                         u+03B8                   E8
//       Ι                         u+0399                   C9
//       ι                         u+03B9                   E9
//       Κ                         u+039A                   CA
//       κ                         u+03BA                   EA
//       Λ                         u+039B                   CB
//       λ                         u+03BB                   EB
//       Μ                         u+039C                   CC
//       μ                         u+03BC                   EC
//       Ν                         u+039D                   CD
//       ν                         u+03BD                   ED
//       Ξ                         u+039E                   CE
//       ξ                         u+03BE                   EE
//       Ο                         u+039F                   CF
//       ο                         u+03BF                   EF
//       Π                         u+03A0                   D0
//       π                         u+03C0                   F0
//       Ρ                         u+03A1                   D1
//       ρ                         u+03C1                   F1
```

```
//      Σ                       u+03A3                  D3
//      σ                       u+03C3                  F3
//      ς                       u+03C2                  F2
//      T                       u+03A4                  D4
//      τ                       u+03C4                  F4
//      Y                       u+03A5                  D5
//      υ                       u+03C5                  F5
//      Φ                       u+03A6                  D6
//      φ                       u+03C6                  F6
//      X                       u+03A7                  D7
//      χ                       u+03C7                  F7
//      Ψ                       u+03A8                  D8
//      ψ                       u+03C8                  F8
//      Ω                       u+03A9                  D9
//      ω                       u+03C9                  F9
```

변환할 데이터를 순차 블록(예: 스트림에서 읽은 데이터)에서만 사용할 수 있거나, 데이터의 양이 너무 많아 더 작은 블록으로 나누어야 하는 경우에는 GetDecoder 메서드의 Decoder 또는 GetEncoder 메서드의 Encoder를 각각 제공하는 파생 클래스의 메서드를 사용해야 합니다.

UTF-16 및 UTF-32 인코더는 big endian 바이트 순서(가장 중요한 바이트 우선) 또는 작은 엔디안 바이트 순서(가장 중요하지 않은 바이트 우선)를 사용할 수 있습니다. 예를 들어 라틴 문자 A(U+0041)는 다음과 같이 직렬화됩니다(16진수).

- UTF-16 빅 엔디언 바이트 순서: 00 41
- UTF-16 리틀 엔디언 바이트 순서: 41 00
- UTF-32 빅 엔디언 바이트 순서: 00 00 00 41
- UTF-32 little endian 바이트 순서: 41 00 00 00

일반적으로 네이티브 바이트 순서를 사용하여 유니코드 문자를 저장하는 것이 더 효율적입니다. 예를 들어 Intel 컴퓨터와 같은 little endian 플랫폼에서 작은 엔디안 바이트 순서를 사용하는 것이 좋습니다.

메서드는 GetPreamble BOM(바이트 순서 표시)을 포함하는 바이트 배열을 검색합니다. 이 바이트 배열이 인코딩된 스트림에 접두사를 지정하면 디코더가 사용되는 인코딩 형식을 식별하는 데 도움이 됩니다.

바이트 순서 및 바이트 순서 표시에 대한 자세한 내용은 유니코드 홈페이지의 유니코드 표준을 참조하세요.

인코딩 클래스는 다음과 같은 오류를 허용합니다.

- 자동으로 "?" 문자로 변경합니다.
- "최적" 문자를 사용합니다.
- U+FFFD 유니코드 대체 문자를 사용하여 EncoderFallback 및 DecoderFallback 클래스를 통해 애플리케이션에 특정한 동작으로 변경합니다.

데이터 스트림 오류가 발생할 경우 예외를 발생시켜야 합니다. 앱은 해당하는 경우 "throwonerror" 플래그를 사용하거나 EncoderExceptionFallback 및 DecoderExceptionFallback 클래스를 사용합니다. 가장 적합한 대체(fallback)는 데이터 손실이나 혼동을 일으킬 수 있고 단순 문자 교체보다 느리기 때문에 권장되지 않는 경우가 많습니다. ANSI 인코딩의 경우 가장 적합한 동작이 기본값입니다.

System.Text.Encoding.Default 속성

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> ⚠ **경고**
>
> 컴퓨터별로 다른 인코딩을 기본값으로 사용할 수 있으며, 기본 인코딩은 단일 컴퓨터에서 변경될 수 있습니다. 인코딩을 사용하여 **Encoding.Default** 컴퓨터 간에 스트리밍되거나 동일한 컴퓨터에서 다른 시간에 검색된 데이터를 인코딩 및 디코딩하는 경우 해당 데이터가 잘못 변환될 수 있습니다. 또한 속성에서 **Default** 반환된 인코딩은 가장 적합한 대체를 사용하여 지원되지 않는 문자를 코드 페이지에서 지원하는 문자에 매핑합니다. 이러한 이유로 기본 인코딩을 사용하지 않는 것이 좋습니다. 인코딩된 바이트가 제대로 디코딩되도록 하려면 유니코드 인코딩(예: **UTF8Encoding** 또는 **UnicodeEncoding**.)을 사용해야 합니다. 더 높은 수준의 프로토콜을 사용하여 동일한 형식이 인코딩 및 디코딩에 사용되는지 확인할 수도 있습니다.

.NET Framework

.NET Framework에서 Default 속성은 항상 시스템의 활성 코드 페이지를 가져와 그것에 해당하는 Encoding 개체를 만듭니다. 활성 코드 페이지는 코드 페이지에 따라 달라지는 추가 문자와 함께 ASCII 문자 집합을 포함하는 ANSI 코드 페이지일 수 있습니다. ANSI 코드 페이지를 기반으로 하는 모든 Default 인코딩에서 데이터가 손실되므로 대신 인코딩을 Encoding.UTF8 사용하는 것이 좋습니다. UTF-8은 U+00에서 U+7F 범위로 동일하지만 손실 없이 ASCII 범위 외부의 문자를 인코딩할 수 있습니다.

.NET 코어

.NET Core에서, Default 속성은 항상 UTF8Encoding 값을 반환합니다. UTF-8은 .NET Core 애플리케이션이 실행되는 모든 운영 체제(Windows, Linux 및 macOS)에서 지원됩니다.

System.Text.RegularExpressions.Regex 클래스

아티클 • 2024. 01. 08.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

클래스는 Regex .를 나타냅니다. NET의 정규식 엔진입니다. 많은 양의 텍스트를 신속하게 구문 분석하여 특정 문자 패턴을 찾는 데 사용할 수 있습니다. 텍스트 부분 문자열을 추출, 편집, 바꾸기 또는 삭제하려면 추출된 문자열을 컬렉션에 추가하여 보고서를 생성합니다.

> **참고**
>
> 문자열이 특정 패턴을 준수하는지 여부를 확인하여 문자열의 유효성을 검사하려는 경우 클래스를 System.Configuration.RegexStringValidator 사용할 수 있습니다.

정규식을 사용하려면 정규식 언어 - 빠른 참조로 문서화된 구문을 사용하여 텍스트 스트림에서 식별하려는 패턴을 정의합니다. 다음으로 필요에 따라 개체를 인스턴스화할 Regex 수 있습니다. 마지막으로 정규식 패턴과 일치하는 텍스트를 바꾸거나 패턴 일치를 식별하는 등의 일부 작업을 수행하는 메서드를 호출합니다.

> **참고**
>
> 몇 가지 일반적인 정규식 패턴은 정규식 예제를 참조 **하세요**. Regular-Expressions.info 같은 정규식 패턴 의 온라인 라이브러리도 많이 있습니다.

정규식 언어에 대한 자세한 내용은 정규식 언어 - 빠른 참조를 참조하거나, 다음 브로슈어 중 하나를 다운로드하여 인쇄하세요.

Word(.docx) 형식 의 빠른 참조(PDF) 형식의 빠른 참조

Regex 및 문자열 메서드

이 System.String 클래스에는 텍스트와 패턴 일치를 수행하는 데 사용할 수 있는 몇 가지 검색 및 비교 메서드가 포함되어 있습니다. 예를 들어 String.Contains, String.EndsWith및 String.StartsWith 메서드는 문자열 인스턴스에 지정된 부분 문자열String.IndexOf이 포함되어 있는지 여부를 결정하고 , , String.LastIndexOfString.IndexOfAny및 String.LastIndexOfAny 메서드는 문자열에서 지정된 부분 문자열의 시작 위치를 반환합니다. 특정 문자열을 검색할 System.String 때 클래스의 메서드를 사용합니다. 문자열에서

Regex 특정 패턴을 검색할 때 클래스를 사용합니다. 자세한 내용 및 예제는 .NET 정규식을 참조 하세요.

정적 메서드와 인스턴스 메서드 비교

정규식 패턴을 정의한 후 다음 두 가지 방법 중 하나를 사용하여 정규식 엔진에 제공할 수 있습니다.

- 정규식을 나타내는 개체를 인스턴스화 Regex 합니다. 이렇게 하려면 정규식 패턴을 Regex 생성자에 전달합니다. Regex 개체는 변경할 수 없습니다. 정규식을 사용하여 개체를 Regex 인스턴스화하면 해당 개체의 정규식을 변경할 수 없습니다.

- (Visual Basic에서) Regex 메서드로 검색할 `static` `Shared` 정규식과 텍스트를 모두 제공합니다. 이렇게 하면 개체를 명시적으로 만들지 않고 정규식을 사용할 수 있습니다 Regex .

모든 Regex 패턴 식별 메서드에는 정적 오버로드와 인스턴스 오버로드가 모두 포함됩니다.

정규식 엔진은 패턴을 사용하려면 먼저 특정 패턴을 컴파일해야 합니다. 개체는 변경할 수 있으므로 Regex 클래스 생성자 또는 정적 메서드를 호출할 때 Regex 발생하는 일회성 프로시저입니다. 단일 정규식을 반복적으로 컴파일할 필요가 없도록 정규식 엔진은 정적 메서드 호출에 사용되는 컴파일된 정규식을 캐시합니다. 따라서 정규식 패턴 일치 메서드는 정적 및 인스턴스 메서드에 대해 비슷한 성능을 제공합니다. 그러나 캐싱은 다음 두 경우의 성능에 부정적인 영향을 줄 수 있습니다.

- 많은 수의 정규식과 함께 정적 메서드 호출을 사용하는 경우 기본적으로 정규식 엔진은 가장 최근에 사용한 15개의 정적 정규식을 캐시합니다. 15 개 이상의 정적 정규식을 사용 하는 애플리케이션에서 일부 정규식을 컴파일해야 합니다. 이 다시 컴파일을 방지하려면 속성을 늘릴 Regex.CacheSize 수 있습니다.

- 이전에 컴파일된 정규식을 사용하여 새 Regex 개체를 인스턴스화하는 경우 예를 들어 다음 코드는 텍스트 스트림에서 중복된 단어를 찾는 정규식을 정의합니다. 이 예제에서는 단일 정규식을 사용하지만 새 개체를 Regex 인스턴스화하여 각 텍스트 줄을 처리합니다. 이렇게 하면 루프의 각 반복과 함께 정규식이 다시 컴파일됩니다.

```
C#

StreamReader sr = new StreamReader(filename);
string input;
string pattern = @"\b(\w+)\s\1\b";
while (sr.Peek() >= 0)
{
   input = sr.ReadLine();
   Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase);
```

```
   MatchCollection matches = rgx.Matches(input);
   if (matches.Count > 0)
   {
      Console.WriteLine("{0} ({1} matches):", input, matches.Count);
      foreach (Match match in matches)
         Console.WriteLine("   " + match.Value);
   }
}
sr.Close();
```

다시 컴파일을 방지하려면 다음 다시 작성된 예제와 같이 필요한 모든 코드에서 액세스할 수 있는 단일 Regex 개체를 인스턴스화해야 합니다.

C#

```
StreamReader sr = new StreamReader(filename);
string input;
string pattern = @"\b(\w+)\s\1\b";
Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase);

while (sr.Peek() >= 0)
{
   input = sr.ReadLine();
   MatchCollection matches = rgx.Matches(input);
   if (matches.Count > 0)
   {
      Console.WriteLine("{0} ({1} matches):", input, matches.Count);
      foreach (Match match in matches)
         Console.WriteLine("   " + match.Value);
   }
}
sr.Close();
```

정규식 작업 수행

개체를 Regex 인스턴스화하고 해당 메서드를 호출하거나 정적 메서드 Regex 를 호출하기로 결정하든 클래스는 다음과 같은 패턴 일치 기능을 제공합니다.

- 일치 항목의 유효성 검사입니다. 메서드를 IsMatch 호출하여 일치 항목이 있는지 여부를 확인합니다.
- 단일 일치 항목을 검색합니다. 메서드를 Match 호출하여 문자열 또는 문자열의 일부에서 첫 번째 일치 항목을 나타내는 개체를 검색 Match 합니다. 메서드를 호출하여 후속 일치 항목을 검색할 Match.NextMatch 수 있습니다.
- 모든 일치 항목을 검색합니다. 메서드를 Matches 호출하여 문자열 또는 문자열의 일부에서 찾은 모든 일치 항목을 나타내는 개체를 검색

System.Text.RegularExpressions.MatchCollection 합니다.

- 일치하는 텍스트를 대체합니다. 일치하는 텍스트를 바꾸기 위해 메서드를 호출 Replace 합니다. 대체 텍스트는 정규식으로 정의할 수도 있습니다. 또한 일부 Replace 메서드에는 대체 텍스트를 프로그래밍 방식으로 정의할 수 있는 매개 변수가 포함 MatchEvaluator 됩니다.

- 입력 문자열의 일부에서 형성된 문자열 배열을 만듭니다. 메서드를 Split 호출하여 정규식으로 정의된 위치에서 입력 문자열을 분할합니다.

패턴 일치 메서드 Regex 외에도 클래스에는 다음과 같은 몇 가지 특수 용도의 메서드가 포함됩니다.

- 메서드는 Escape 정규식 또는 입력 문자열에서 정규식 연산자로 해석될 수 있는 모든 문자를 이스케이프합니다.

- 이 메서드는 이러한 이 Unescape 스케이프 문자를 제거합니다.

- 메서드는 CompileToAssembly 미리 정의된 정규식을 포함하는 어셈블리를 만듭니다. .NET에는 네임스페이스에서 이러한 특수 용도 어셈블리의 예제가 System.Web.RegularExpressions 포함되어 있습니다.

제한 시간 값 정의

.NET은 패턴 일치에서 상당한 성능과 유연성을 제공하는 완전한 기능을 갖춘 정규식 언어를 지원합니다. 그러나 성능과 유연성은 성능 저하의 위험이 있습니다. 제대로 수행되지 않는 정규식은 놀라울 정도로 쉽게 만들 수 있습니다. 경우에 따라 과도한 역추적을 사용하는 정규식 작업은 정규식 패턴과 거의 일치하는 텍스트를 처리할 때 응답을 중지하는 것처럼 보일 수 있습니다. .NET 정규식 엔진에 대한 자세한 내용은 정규식 동작의 세부 정보를 참조 하세요. 과도한 역추적에 대한 자세한 내용은 역추적을 참조 하세요.

.NET Framework 4.5부터 정규식 일치에 대한 제한 시간 간격을 정의하여 과도한 역추적을 제한할 수 있습니다. 정규식 패턴 및 입력 텍스트에 따라 실행 시간이 지정된 제한 시간 간격을 초과할 수 있지만 지정된 제한 시간 간격보다 역추적하는 데 더 많은 시간을 소비하지는 않습니다. 정규식 엔진의 시간이 초과되면 예외가 RegexMatchTimeoutException throw됩니다. 대부분의 경우 정규식 엔진이 정규식 패턴과 거의 일치하는 텍스트를 일치시키려고 시도하여 처리 능력을 낭비하지 못하게 합니다. 그러나 시간 제한 간격이 너무 낮게 설정되었거나 현재 컴퓨터 부하로 인해 전반적인 성능 저하가 발생했음을 나타낼 수도 있습니다.

예외를 처리하는 방법은 예외의 원인에 따라 달라집니다. 시간 제한 간격이 너무 낮게 설정되거나 과도한 컴퓨터 로드로 인해 예외가 발생하는 경우 제한 시간 간격을 늘리고 일치 작업을 다시 시도할 수 있습니다. 정규식이 과도한 역추적을 사용하므로 예외가 발생

하는 경우 일치 항목이 없다고 가정할 수 있으며, 필요에 따라 정규식 패턴을 수정하는 데 도움이 되는 정보를 기록할 수 있습니다.

정규식 개체를 인스턴스화할 때 생성자를 호출 Regex(String, RegexOptions, TimeSpan) 하여 제한 시간 간격을 설정할 수 있습니다. 정적 메서드의 경우 매개 변수가 있는 일치하는 메서드의 오버로드를 호출하여 제한 시간 간격을 `matchTimeout` 설정할 수 있습니다. 제한 시간 값을 명시적으로 설정하지 않으면 기본 제한 시간 값이 다음과 같이 결정됩니다.

- 애플리케이션 수준 시간 제한을 사용 하 여 값 하나 있습니다. 이 애플리케이션 도메인에 적용 되는 제한 시간 값 수는 Regex 개체가 인스턴스화되거나 정적 메서드를 호출 합니다. 호출 하 여 애플리케이션 수준 시간 제한 값을 설정할 수 있습니다는 AppDomain.SetData 의 문자열 표현에 할당할 메서드를 TimeSpan "REGEX_DEFAULT_MATCH_TIMEOUT" 속성 값입니다.
- 값을 사용 하 여 InfiniteMatchTimeout없는 애플리케이션 수준 시간 제한 값이 설정된 경우.

> **ⓘ 중요**
>
> 모든 정규식 패턴 일치 작업에서 제한 시간 값을 설정하는 것이 좋습니다. 자세한 내용은 정규식 모범 사례를 참조 **하세요**.

예제

다음 예제에서는 정규식을 사용하여 문자열에서 반복되는 단어 발생에 대해 검사. 정규식 `\b(?<word>\w+)\s+(\k<word>)\b` 은 다음 표와 같이 해석할 수 있습니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 일치를 시작합니다.
`(?<word>\w+)`	하나 이상의 단어 문자를 단어 경계까지 찾습니다. 캡처된 이 그룹의 `word` 이름을 지정합니다.
`\s+`	하나 이상의 공백 문자를 찾습니다.
`(\k<word>)`	이름이 `word` 지정된 캡처된 그룹과 일치합니다.
`\b`	단어 경계를 찾습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Test
{
    public static void Main ()
    {
        // Define a regular expression for repeated words.
        Regex rx = new Regex(@"\b(?<word>\w+)\s+(\k<word>)\b",
          RegexOptions.Compiled | RegexOptions.IgnoreCase);

        // Define a test string.
        string text = "The the quick brown fox  fox jumps over the lazy dog
dog.";

        // Find matches.
        MatchCollection matches = rx.Matches(text);

        // Report the number of matches found.
        Console.WriteLine("{0} matches found in:\n   {1}",
                          matches.Count,
                          text);

        // Report on each match.
        foreach (Match match in matches)
        {
            GroupCollection groups = match.Groups;
            Console.WriteLine("'{0}' repeated at positions {1} and {2}",
                              groups["word"].Value,
                              groups[0].Index,
                              groups[1].Index);
        }
    }
}

// The example produces the following output to the console:
//       3 matches found in:
//          The the quick brown fox  fox jumps over the lazy dog dog.
//       'The' repeated at positions 0 and 4
//       'fox' repeated at positions 20 and 25
//       'dog' repeated at positions 49 and 53
```

다음 예제에서는 정규식을 사용하여 문자열이 통화 값을 나타내는지 또는 통화 값을 나타내는 올바른 형식인지를 검사 방법을 보여 줍니다. 이 경우 정규식은 en-US 문화권에 NumberFormatInfo.CurrencyDecimalSeparator대한 , CurrencyDecimalDigits, NumberFormatInfo.CurrencySymbolNumberFormatInfo.NegativeSign및 NumberFormatInfo.PositiveSign 속성에서 동적으로 빌드됩니다. 결과 정규식은 .입니다 `^\s*[\+-]?\s?\$?\s?(\d*\.?\d{2}?){1}$`. 이 정규식은 다음 표와 같이 해석할 수 있습니다.

테이블 확장

패턴	설명
`^`	문자열의 시작 부분에서 시작합니다.
`\s*`	0개 이상의 공백 문자가 일치하는지 확인합니다.
`[\+-]?`	양수 기호 또는 음수 기호가 0개 또는 1번 일치하는지 확인합니다.
`\s?`	0번 이상 나오는 공백 문자를 찾습니다.
`\$?`	달러 기호가 0개 또는 1번 일치하는지 확인합니다.
`\s?`	0번 이상 나오는 공백 문자를 찾습니다.
`\d*`	0번 이상 나오는 10진수를 찾습니다.
`\.?`	0 또는 10진수 기호와 일치합니다.
`(\d{2})?`	그룹 1 캡처: 10진수 2자리를 0개 또는 1회 일치시킵니다.
`(\d*\.?(\d{2})?){1}`	소수점 기호로 구분된 정수 및 소수 자릿수의 패턴을 한 번 이상 일치시킵니다.
`$`	문자열의 끝과 일치합니다.

이 경우 정규식은 유효한 통화 문자열에 그룹 구분 기호가 없고 소수 자릿수 또는 지정된 문화권의 CurrencyDecimalDigits 속성에 정의된 소수 자릿수가 없다고 가정합니다.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        // Get the en-US NumberFormatInfo object to build the regular
        // expression pattern dynamically.
        NumberFormatInfo nfi = CultureInfo.GetCultureInfo("en-
US").NumberFormat;

        // Define the regular expression pattern.
        string pattern;
        pattern = @"^\s*[";
        // Get the positive and negative sign symbols.
        pattern += Regex.Escape(nfi.PositiveSign + nfi.NegativeSign) + @"]?
\s?";
        // Get the currency symbol.
```

```
        pattern += Regex.Escape(nfi.CurrencySymbol) + @"?\s?";
        // Add integral digits to the pattern.
        pattern += @"(\d*";
        // Add the decimal separator.
        pattern += Regex.Escape(nfi.CurrencyDecimalSeparator) + "?";
        // Add the fractional digits.
        pattern += @"(\d{";
        // Determine the number of fractional digits in currency values.
        pattern += nfi.CurrencyDecimalDigits.ToString() + "})?){1}$";

        Console.WriteLine($"Pattern is {pattern}\n");

        Regex rgx = new Regex(pattern);

        // Define some test strings.
        string[] tests = { "-42", "19.99", "0.001", "100 USD",
                           ".34", "0.34", "1,052.21", "$10.62",
                           "+1.43", "-$0.23" };

        // Check each test string against the regular expression.
        foreach (string test in tests)
        {
            if (rgx.IsMatch(test))
                Console.WriteLine($"{test} is a currency value.");
            else
                Console.WriteLine($"{test} is not a currency value.");
        }
    }
}
// The example displays the following output:
//       Pattern is ^\s*[\+-]?\s?\$?\s?(\d*\.?(\d{2})?){1}$
//
//       -42 is a currency value.
//       19.99 is a currency value.
//       0.001 is not a currency value.
//       100 USD is not a currency value.
//       .34 is a currency value.
//       0.34 is a currency value.
//       1,052.21 is not a currency value.
//       $10.62 is a currency value.
//       +1.43 is a currency value.
//       -$0.23 is a currency value.
```

이 예제의 정규식은 동적으로 작성되므로 디자인 타임에 정규식 엔진에서 정규식 언어 연산자로 지정된 문화권의 통화 기호, 10진수 기호 또는 양수 및 음수 기호를 잘못 해석할 수 있는지 여부를 알 수 없습니다. 잘못된 해석을 방지하기 위해 예제에서는 동적으로 생성된 각 문자열을 메서드에 Escape 전달합니다.

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

System.Text.RegularExpressions.Regex.Match 메서드

아티클 • 2025. 03. 26.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Match(String, Int32) 메서드는 입력 문자열에서 `startat` 문자 위치 이후부터 정규식 패턴과 일치하는 첫 번째 부분 문자열을 반환합니다. Match(String, Int32) 메서드가 검색하는 정규식 패턴은 Regex 클래스 생성자 중 하나에 대한 호출에 의해 정의됩니다. 정규식 패턴을 작성하는 데 사용되는 언어 요소에 대한 자세한 내용은 정규식 언어 - 빠른 참조참조하세요.

`startat` 매개 변수

필요에 따라 `startat` 매개 변수를 사용하여 문자열의 시작 위치를 지정할 수 있습니다. 문자열에서 `startat` 이전에 시작된 일치 항목은 무시됩니다. 시작 위치를 지정하지 않으면 왼쪽에서 오른쪽 검색의 `input` 왼쪽 끝인 기본 위치에서 검색이 시작되고 오른쪽에서 왼쪽 검색에서는 `input` 오른쪽 끝에서 시작됩니다. `startat`시작에도 불구하고 반환된 일치 항목의 인덱스는 문자열의 시작에 상대적입니다.

정규식 엔진은 `startat`전에 시작하는 일치 항목을 반환하지 않지만 `startat`전에 문자열을 무시하지 않습니다. 즉, 앵커 또는 lookbehind 어설션과 같은 어설션은 입력 전체에 계속 적용될 있습니다. 예를 들어, 다음 코드에는 입력 문자열에서 인덱스 `startat`의 5 이전에 발생해도 충족되는 후방 탐색 어설션을 포함한 패턴이 포함되어 있습니다.

C#

```
using System;
using System.Text.RegularExpressions;

namespace Examples
{
    public class Example3
    {
        public static void Main()
        {
            string input = "Zip code: 98052";
            var regex = new Regex(@"(?<=Zip code: )\d{5}");
            Match match = regex.Match(input, 5);
            if (match.Success)
                Console.WriteLine($"Match found: {match.Value}");
        }
    }
}
```

```
// This code prints the following output:
// Match found: 98052
```

> **팁**
>
> - 패턴이 `^` 앵커로 시작하지만 `startat` 0보다 크면 인덱스 0에서 시작하도록 `^` 제한되므로 한 줄 검색에서 일치하는 항목을 찾을 수 없습니다.
> - **\G 앵커**는 `startat`에서 만족합니다. 따라서 문자열의 특정 문자 위치에서 정확하게 시작되도록 일치 항목을 제한하려면, 정규식을 왼쪽에서 오른쪽으로 진행하는 패턴의 시작 위치에 `\G`로 고정합니다. 이것은 일치 조건을 제한하여 반드시 `startat`에서 시작하도록 하며, 여러 일치 항목이 필요할 경우에는 일치 항목들이 연속적으로 나타나도록 합니다.

오른쪽에서 왼쪽으로 검색

오른쪽에서 왼쪽 검색, 즉 정규식 패턴이 RegexOptions.RightToLeft 옵션을 사용하여 생성될 때 다음과 같은 방식으로 동작합니다.

- 스캔은 반대 방향으로 이동하며 패턴을 뒤에서(오른쪽) 앞으로(왼쪽) 일치시킵니다.
- 기본 시작 위치는 입력 문자열의 오른쪽 끝입니다.
- `startat` 지정하면 오른쪽에서 왼쪽으로 스캔이 `startat` - 1(`startat`아님)의 문자에서 시작됩니다.
- 패턴의 오른쪽 끝에 `\G` 앵커를 지정하면 (첫 번째) 일치 항목이 정확히 `startat` - 1로 끝나도록 제한합니다.

오른쪽에서 왼쪽으로 검색하는 방법에 대한 자세한 내용은 오른쪽에서 왼쪽 모드참조하세요.

일치하는 항목을 찾을 수 있는지 확인

반환된 Match 개체의 Success 속성 값을 확인하여 입력 문자열에서 정규식 패턴이 발견되었는지 여부를 확인할 수 있습니다. 일치 항목이 발견되면 반환된 Match 개체의 Value 속성에는 정규식 패턴과 일치하는 `input` 부분 문자열이 포함됩니다. 일치하는 항목이 없으면 값이 String.Empty입니다.

첫 번째 또는 여러 개의 일치

이 메서드는 정규식 패턴과 일치하는 `input` `startat` 문자 위치 또는 그 뒤의 첫 번째 부분 문자열을 반환합니다. 반환된 Match 개체의 Match.NextMatch 메서드를 반복적으로 호출하여 후속 일치 항목을 검색할 수 있습니다. Regex.Matches(String, Int32) 메서드를 호출하여 단일 메서드 호출에서 모든 일치 항목을 검색할 수도 있습니다.

타임아웃 예외

일치하는 작업의 실행 시간이 Regex.Regex(String, RegexOptions, TimeSpan) 생성자가 지정한 제한 시간 간격을 초과하면 RegexMatchTimeoutException 예외가 throw됩니다. 생성자를 호출할 때 제한 시간 간격을 설정하지 않으면 작업이 Regex 개체가 만들어진 애플리케이션 도메인에 대해 설정된 제한 시간 값을 초과하면 예외가 throw됩니다. Regex 생성자 호출 또는 애플리케이션 도메인의 속성에 제한 시간이 정의되지 않거나 시간 제한 값이 Regex.InfiniteMatchTimeout경우 예외가 throw되지 않습니다.

System.Text.Rune 구조체

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

인스턴스는 Rune 서로게이트 범위(U+D800에서 U+DFFF)를 제외한 모든 코드 포인트를 의미하는 유니코드 스칼라 값을 나타냅니다. 형식의 생성자 및 변환 연산자는 입력의 유효성을 검사하므로 소비자는 기본 인스턴스가 잘 형성되어 있다고 가정하여 API를 Rune 호출할 수 있습니다.

유니코드 스칼라 값, 코드 포인트, 서로게이트 범위 및 잘 구성된 용어에 익숙하지 않은 경우 .NET의 문자 인코딩 소개를 참조하세요.

Rune 형식을 사용하는 경우

코드가 다음과 같은 경우 형식을 `Rune` 사용하는 것이 좋습니다.

- 유니코드 스칼라 값이 필요한 API 호출
- 유니코드 대리쌍을 명시적으로 처리합니다.

유니코드 스칼라 값이 필요한 API

코드가 `char` 또는 `string` 내의 `ReadOnlySpan<char>` 인스턴스를 반복하는 경우, 일부 `char` 메서드는 서로게이트 범위에 있는 `char` 인스턴스에서 제대로 작동하지 않습니다. 예를 들어 다음 API는 스칼라 값 `char` 이 올바르게 작동해야 합니다.

- Char.GetNumericValue
- Char.GetUnicodeCategory
- Char.IsDigit
- Char.IsLetter
- Char.IsLetterOrDigit
- Char.IsLower
- Char.IsNumber
- Char.IsPunctuation
- Char.IsSymbol
- Char.IsUpper

다음 예제에서는 각 인스턴스가 서로게이트 코드 포인트일 경우 제대로 작동하지 않는 코드를 보여 줍니다.

```
C#
```

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
int CountLettersBadExample(string s)
{
    int letterCount = 0;

    foreach (char ch in s)
    {
        if (char.IsLetter(ch))
        { letterCount++; }
    }

    return letterCount;
}
```

다음은 `ReadOnlySpan<char>`와 작동하는 동등한 코드입니다.

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static int CountLettersBadExample(ReadOnlySpan<char> span)
{
    int letterCount = 0;

    foreach (char ch in span)
    {
        if (char.IsLetter(ch))
        { letterCount++; }
    }

    return letterCount;
}
```

앞의 코드는 영어와 같은 일부 언어에서 올바르게 작동합니다.

C#

```
CountLettersInString("Hello")
// Returns 5
```

그러나 Osage와 같은 기본 다국어 평면 외부의 언어에서는 제대로 작동하지 않습니다.

C#

```
CountLettersInString("𐓷𐓘𐓻𐓘𐓻𐓢 𐓍𐓢")
// Returns 0
```

이 메서드가 Osage 텍스트에 대해 잘못된 결과를 반환하는 이유는 Osage 문자의 `char` 인스턴스가 서로게이트 코드 포인트이기 때문입니다. 단일 서로게이트 코드 지점에는 문자인지 여부를 확인하기에 충분한 정보가 없습니다.

`Rune`를 대신 사용하도록 이 코드를 변경하면 메서드는 기본 다국어 평면을 벗어난 코드 포인트에서도 올바르게 작동합니다.

C#

```
int CountLetters(string s)
{
    int letterCount = 0;

    foreach (Rune rune in s.EnumerateRunes())
    {
        if (Rune.IsLetter(rune))
        { letterCount++; }
    }

    return letterCount;
}
```

다음은 `ReadOnlySpan<char>`와 작동하는 동등한 코드입니다.

C#

```
static int CountLetters(ReadOnlySpan<char> span)
{
    int letterCount = 0;

    foreach (Rune rune in span.EnumerateRunes())
    {
        if (Rune.IsLetter(rune))
        { letterCount++; }
    }

    return letterCount;
}
```

앞의 코드는 Osage 문자를 올바르게 계산합니다.

C#

```
CountLettersInString("𐓵𐓘𐓵𐓘 𐓂𐓪")
// Returns 8
```

서로게이트 쌍을 명시적으로 처리하는 코드

코드가 `Rune` 서로게이트 코드 지점에서 명시적으로 작동하는 API를 호출하는 경우(예: 다음 메서드) 형식을 사용하는 것이 좋습니다.

- Char.IsSurrogate
- Char.IsSurrogatePair
- Char.IsHighSurrogate
- Char.IsLowSurrogate
- Char.ConvertFromUtf32
- Char.ConvertToUtf32

예를 들어 다음 메서드에는 서로게이트 `char` 쌍을 처리하는 특수 논리가 있습니다.

C#

```
static void ProcessStringUseChar(string s)
{
    Console.WriteLine("Using char");

    for (int i = 0; i < s.Length; i++)
    {
        if (!char.IsSurrogate(s[i]))
        {
            Console.WriteLine($"Code point: {(int)(s[i])}");
        }
        else if (i + 1 < s.Length && char.IsSurrogatePair(s[i], s[i + 1]))
        {
            int codePoint = char.ConvertToUtf32(s[i], s[i + 1]);
            Console.WriteLine($"Code point: {codePoint}");
            i++; // so that when the loop iterates it's actually +2
        }
        else
        {
            throw new Exception("String was not well-formed UTF-16.");
        }
    }
}
```

이러한 코드는 다음 예제와 같이 사용하는 `Rune`경우 더 간단합니다.

C#

```
static void ProcessStringUseRune(string s)
{
    Console.WriteLine("Using Rune");

    for (int i = 0; i < s.Length;)
    {
        if (!Rune.TryGetRuneAt(s, i, out Rune rune))
        {
            throw new Exception("String was not well-formed UTF-16.");
```

```
        }

        Console.WriteLine($"Code point: {rune.Value}");
        i += rune.Utf16SequenceLength; // increment the iterator by the number of
chars in this Rune
    }
}
```

사용할 필요가 없는 경우 `Rune`

코드에서 `Rune` 타입을 사용할 필요가 없습니다.

- 정확한 `char` 일치 항목을 찾습니다.
- 알려진 char 값에 문자열을 분할합니다.

`Rune` 형식을 사용하면 코드가 다음과 같은 경우 잘못된 결과를 반환할 수 있습니다.

- 의 표시 문자 수를 계산합니다. `string`

정확한 `char` 일치 항목 찾기

다음 코드는 검색된 특정 문자를 반복 `string` 하여 첫 번째 일치 항목의 인덱스를 반환합니다. 코드가 단일 `Rune` 문자로 표현되는 문자를 찾고 있으므로 이 코드를 사용하도록 `char` 변경할 필요가 없습니다.

C#

```
int GetIndexOfFirstAToZ(string s)
{
    for (int i = 0; i < s.Length; i++)
    {
        char thisChar = s[i];
        if ('A' <= thisChar && thisChar <= 'Z')
        {
            return i; // found a match
        }
    }

    return -1; // didn't find 'A' - 'Z' in the input string
}
```

알려진 조건에 따라 문자열을 분할합니다 `char`

다음 예제와 같이 (공간) 또는 `string.Split` (쉼표)와 같은 `' '` 구분 기호를 호출 `','` 하고 사용하는 것이 일반적입니다.

C#

```
string inputString = "🐂, 🐄, 🐆";
string[] splitOnSpace = inputString.Split(' ');
string[] splitOnComma = inputString.Split(',');
```

코드에서 단일 `Rune` 문자로 표현되는 문자를 찾고 있으므로 여기서 사용할 `char` 필요가 없습니다.

디스플레이 문자 수를 계산하기 `string`

문자열의 `Rune` 인스턴스 수가 문자열을 표시할 때 표시되는 사용자 인식 가능 문자 수와 일치하지 않을 수 있습니다.

`Rune` 인스턴스는 유니코드 스칼라 값을 나타내므로 유니코드 텍스트 구분 지침을 따르는 구성 요소는 표시 문자를 계산하기 위한 구성 요소로 사용할 `Rune` 수 있습니다.

이 형식은 StringInfo 표시 문자 수를 계산하는 데 사용할 수 있지만 .NET 5 이상 이외의 .NET 구현에 대한 모든 시나리오에서 올바르게 계산되지는 않습니다.

자세한 내용은 Grapheme 클러스터를 참조하세요.

`Rune`를 인스턴스화하는 방법

인스턴스를 가져오는 방법에는 여러 가지가 있습니다 `Rune` . 생성자를 사용하여 `Rune`를 직접 만들 수 있습니다.

- 코드 포인트입니다.

  C#

  ```
  Rune a = new Rune(0x0061); // LATIN SMALL LETTER A
  Rune b = new Rune(0x10421); // DESERET CAPITAL LETTER ER
  ```

- 단일 `char`.

  C#

  ```
  Rune c = new Rune('a');
  ```

- 서로게이트 `char` 쌍입니다.

  C#

```
Rune d = new Rune('\ud83d', '\udd2e'); // U+1F52E CRYSTAL BALL
```

입력이 유효한 유니코드 스칼라 값을 나타내지 않으면 모든 생성자가 throw `ArgumentException` 됩니다.

Rune.TryCreate 실패 시 예외를 발생시키지 않으려는 호출자에게 사용할 수 있는 방법이 있습니다.

`Rune` 기존 입력 시퀀스에서 인스턴스를 읽을 수도 있습니다. 예를 들어 `ReadOnlySpan<char>` UTF-16 데이터를 나타내는 경우 메서드는 Rune.DecodeFromUtf16 입력 범위의 시작 부분에 첫 번째 `Rune` 인스턴스를 반환합니다. 메서드도 Rune.DecodeFromUtf8 마찬가지로 작동하여 `ReadOnlySpan<byte>` UTF-8 데이터를 나타내는 매개 변수를 허용합니다. 범위의 시작이 아니라 범위의 끝에서 읽는 것과 동일한 메서드가 있습니다.

의 쿼리 속성 `Rune`

인스턴스의 정수 코드 포인트 값을 `Rune` 얻으려면 속성을 사용합니다 Rune.Value .

C#

```
Rune rune = new Rune('\ud83d', '\udd2e'); // U+1F52E CRYSTAL BALL
int codePoint = rune.Value; // = 128302 decimal (= 0x1F52E)
```

`char` 형식에서 사용할 수 있는 많은 정적 API는 `Rune` 형식에서도 사용할 수 있습니다. 예를 들어, Rune.IsWhiteSpace와 Rune.GetUnicodeCategory는 Char.IsWhiteSpace와 Char.GetUnicodeCategory 메서드와 동일합니다. 해당 `Rune` 메서드는 서로게이트 쌍을 올바르게 처리합니다.

다음 예제 코드는 입력으로 `ReadOnlySpan<char>` 사용하고 문자나 숫자가 아닌 모든 `Rune` 범위의 시작과 끝에서 트리밍합니다.

C#

```
static ReadOnlySpan<char> TrimNonLettersAndNonDigits(ReadOnlySpan<char> span)
{
    // First, trim from the front.
    // If any Rune can't be decoded
    // (return value is anything other than "Done"),
    // or if the Rune is a letter or digit,
    // stop trimming from the front and
    // instead work from the end.
    while (Rune.DecodeFromUtf16(span, out Rune rune, out int charsConsumed) ==
OperationStatus.Done)
    {
```

```
        if (Rune.IsLetterOrDigit(rune))
        { break; }
        span = span[charsConsumed..];
    }

    // Next, trim from the end.
    // If any Rune can't be decoded,
    // or if the Rune is a letter or digit,
    // break from the loop, and we're finished.
    while (Rune.DecodeLastFromUtf16(span, out Rune rune, out int charsConsumed) ==
OperationStatus.Done)
    {
        if (Rune.IsLetterOrDigit(rune))
        { break; }
        span = span[..^charsConsumed];
    }

    return span;
}
```

API 간에는 몇 가지 API 차이점이 있습니다 `char` `Rune`. 다음은 그 예입니다.

- 정의상 `Rune` 인스턴스는 절대 서로게이트 코드 포인트가 될 수 없으므로 Char.IsSurrogate(Char)와 동일한 `Rune`는 없습니다.
- 항상 Rune.GetUnicodeCategory .와 동일한 결과를 Char.GetUnicodeCategory반환하지는 않습니다. 와 동일한 값을 CharUnicodeInfo.GetUnicodeCategory반환합니다. **비고**에 대한 자세한 내용은 Char.GetUnicodeCategory를 참조하세요.

`Rune` UTF-8 또는 UTF-16으로 변환

유니 `Rune` 코드 스칼라 값이므로 UTF-8, UTF-16 또는 UTF-32 인코딩으로 변환할 수 있습니다. 이 `Rune` 형식은 UTF-8 및 UTF-16으로의 변환을 기본적으로 지원합니다.

Rune.EncodeToUtf16는 `Rune` 인스턴스를 `char` 인스턴스로 변환합니다. UTF-16으로 `char` 인스턴스를 변환할 때 생성되는 `Rune` 인스턴스 수를 쿼리하려면 Rune.Utf16SequenceLength 속성을 사용하십시오. UTF-8 변환에서도 비슷한 메서드가 있습니다.

다음 예제에서는 인스턴스를 `Rune` 배열로 `char` 변환합니다. 이 코드는 변수에 인스턴스가 `Rune` 있다고 가정합니다 `rune` .

C#

```
char[] chars = new char[rune.Utf16SequenceLength];
int numCharsWritten = rune.EncodeToUtf16(chars);
```

A `string`는 UTF-16 문자 시퀀스이므로, `Rune` 인스턴스도 같은 예제에서 UTF-16으로 변환됩니다.

```
C#

string theString = rune.ToString();
```

다음 예제에서는 인스턴스를 `Rune` 바이트 배열로 `UTF-8` 변환합니다.

```
C#

byte[] bytes = new byte[rune.Utf8SequenceLength];
int numBytesWritten = rune.EncodeToUtf8(bytes);
```

및 Rune.EncodeToUtf16 메서드는 Rune.EncodeToUtf8 작성된 실제 요소 수를 반환합니다. 대상 버퍼가 너무 짧아 결과를 포함할 수 없는 경우 예외를 throw합니다. 예외를 피하려는 호출자를 위해 예외를 던지지 않는 TryEncodeToUtf8 및 TryEncodeToUtf16 메서드도 있습니다.

.NET의 다른 언어와의 Rune 비교

"rune"이라는 용어는 유니코드 표준에 정의되어 있지 않습니다. 이 용어는 UTF-8 생성으로 거슬러 올라갑니다. 롭 파이크와 켄 톰슨은 결국 코드 포인트로 알려질 것을 설명하는 용어를 찾고 있었다. 그들은 "rune"이라는 용어에 정착했고, Rob Pike는 나중에 Go 프로그래밍 언어에 대한 영향력을 통해 이 용어를 대중화하는 데 도움을 주었습니다.

그러나 .NET `Rune` 형식은 Go `rune` 형식과 동일하지 않습니다. Go에서 `rune` 형식은 int32에 대한 별칭입니다. Go rune은 유니코드 코드 포인트를 나타내려는 것이지만, 유효하지 않은 유니코드 코드 포인트와 서로게이트 코드 포인트를 포함하여 32비트 값이라면 어떤 값도 될 수 있습니다.

다른 프로그래밍 언어의 유사한 형식은 Rust의 기본 char 형식 또는 Swift의 Unicode.Scalar 형식을 참조하세요. 둘 다 유니코드 스칼라 값을 나타냅니다. 다음과 유사한 기능을 제공합니다. NET의 `Rune` 형식이며, 합법적인 유니코드 스칼라 값이 아닌 값의 인스턴스화를 허용하지 않습니다.

System.Text.StringBuilder 클래스

아티클 • 2024. 01. 08.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

클래스는 StringBuilder 값이 변경 가능한 문자 시퀀스인 문자열과 유사한 개체를 나타냅니다.

StringBuilder 및 문자열 형식

String 둘 다 문자 시퀀스를 나타내지만 StringBuilder 다르게 구현됩니다. String 은 변경할 수 없는 형식입니다. 즉, 개체를 수정 String 하는 것처럼 보이는 각 작업은 실제로 새 문자열을 만듭니다.

예를 들어 다음 C# 예제에서 메서드를 호출 String.Concat 하면 이름이 지정된 `value` 문자열 변수의 값이 변경됩니다. 실제로 메서드는 Concat 메서드에 `value` 전달된 개체와 다른 값과 주소를 `value` 가진 개체를 반환합니다. 예제는 컴파일러 옵션을 사용하여 `/unsafe` 컴파일해야 합니다.

C#

```
using System;

public class Example7
{
    public unsafe static void Main()
    {
        string value = "This is the first sentence" + ".";
        fixed (char* start = value)
        {
            value = String.Concat(value, "This is the second sentence. ");
            fixed (char* current = value)
            {
                Console.WriteLine(start == current);
            }
        }
    }
}
// The example displays the following output:
//       False
```

광범위한 문자열 조작(예: 루프에서 문자열을 여러 번 수정하는 앱)을 수행하는 루틴의 경우 문자열을 반복적으로 수정하면 성능이 크게 저하될 수 있습니다. 대안은 변경 가능한 문자열 클래스인 사용을 StringBuilder위한 것입니다. 변경 가능성은 클래스의 인스턴스가 만들어지면 문자를 추가, 제거, 바꾸기 또는 삽입하여 수정할 수 있음을 의미합니다. 개

체는 StringBuilder 문자열에 대한 확장을 수용하기 위해 버퍼를 기본. 공간을 사용할 수 있는 경우 새 데이터가 버퍼에 추가됩니다. 그렇지 않으면 더 큰 새 버퍼가 할당되고 원래 버퍼의 데이터가 새 버퍼에 복사되고 새 데이터가 새 버퍼에 추가됩니다.

> **ⓘ 중요**
>
> 클래스는 **StringBuilder** 일반적으로 클래스보다 **String** 더 나은 성능을 제공하지만 문자열을 조작할 때마다 자동으로 **StringStringBuilder** 대체해서는 안 됩니다. 성능은 문자열의 크기, 새 문자열에 할당할 메모리 양, 코드가 실행되는 시스템 및 작업 유형에 따라 달라집니다. 실제로 상당한 성능 향상을 제공하는지 여부를 **StringBuilder** 확인하기 위해 코드를 테스트할 준비가 되어 있어야 합니다.

다음 조건에서 클래스를 String 사용하는 것이 좋습니다.

- 코드에서 문자열에 적용할 변경 횟수가 작은 경우 이러한 경우 StringBuilder 성능이 미미하거나 성능이 향상 String되지 않을 수 있습니다.
- 특히 문자열 리터럴을 사용하여 고정된 수의 연결 작업을 수행하는 경우 이 경우 컴파일러는 연결 작업을 단일 작업으로 결합할 수 있습니다.
- 문자열을 빌드하는 동안 광범위한 검색 작업을 수행해야 하는 경우 클래스에 StringBuilder 와 같은 `IndexOf` 검색 메서드가 `StartsWith`없습니다. 개체를 StringBuilder 이러한 작업에 대한 개체로 String 변환해야 하며, 이렇게 하면 사용 StringBuilder으로 인한 성능 이점이 무효화됩니다. 자세한 내용은 StringBuilder 개체 섹션의 텍스트 검색을 참조하세요.

다음 조건에서 클래스를 StringBuilder 사용하는 것이 좋습니다.

- 코드가 디자인 타임에 알 수 없는 개수의 문자열을 변경할 것으로 예상하는 경우(예: 루프를 사용하여 사용자 입력이 포함된 난수의 문자열을 연결할 때)
- 코드에서 문자열을 상당히 많이 변경할 것으로 예상되는 경우

StringBuilder 작동 방식

이 속성은 StringBuilder.Length 개체에 StringBuilder 현재 포함된 문자 수를 나타냅니다. 개체에 StringBuilder 문자를 추가하면 개체가 포함할 수 있는 문자 수를 정의하는 속성의 StringBuilder.Capacity 크기와 같을 때까지 길이가 증가합니다. 추가된 문자 수가 개체의 StringBuilder 길이가 현재 용량을 초과하면 새 메모리가 할당되고, 속성 값 Capacity 이 두 배로 늘어나고, 새 문자가 개체에 StringBuilder 추가되고, 해당 Length 속성이 조정됩니다. 개체에 StringBuilder 대한 추가 메모리는 속성에 정의된 StringBuilder.MaxCapacity 값에 도달할 때까지 동적으로 할당됩니다. 최대 용량에 도달하면 개체에 더 이상 메모리를

StringBuilder 할당할 수 없으며 문자를 추가하거나 최대 용량을 초과하여 확장하려고 하면 예외가 ArgumentOutOfRangeExceptionOutOfMemoryException throw됩니다.

다음 예제에서는 개체가 StringBuilder 새 메모리를 할당하고 개체에 할당된 문자열이 확장됨에 따라 동적으로 용량을 늘리는 방법을 보여 줍니다. 이 코드는 기본(매개 변수 없는) 생성자를 호출하여 개체를 만듭니다 StringBuilder . 이 개체의 기본 용량은 16자이며 최대 용량은 20억 자 이상입니다. "이것은 문장입니다." 문자열을 추가하면 문자열 길이(19자)가 개체의 기본 용량을 초과하기 때문에 새 메모리 할당이 StringBuilder 발생합니다. 개체의 용량이 32자로 두 배로 늘어나고, 새 문자열이 추가되고, 개체의 길이가 이제 19자입니다. 그런 다음 코드는 개체의 값에 "이것은 추가 문장입니다."라는 문자열을 StringBuilder 11번 추가합니다. 추가 작업으로 인해 개체의 StringBuilder 길이가 용량을 초과할 때마다 기존 용량이 두 배로 늘어나고 Append 작업이 성공합니다.

C#

```
using System;
using System.Reflection;
using System.Text;

public class Example4
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder();
        ShowSBInfo(sb);
        sb.Append("This is a sentence.");
        ShowSBInfo(sb);
        for (int ctr = 0; ctr <= 10; ctr++)
        {
            sb.Append("This is an additional sentence.");
            ShowSBInfo(sb);
        }
    }

    private static void ShowSBInfo(StringBuilder sb)
    {
        foreach (var prop in sb.GetType().GetProperties())
        {
            if (prop.GetIndexParameters().Length == 0)
                Console.Write("{0}: {1:N0}    ", prop.Name,
prop.GetValue(sb));
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    Capacity: 16    MaxCapacity: 2,147,483,647    Length: 0
//    Capacity: 32    MaxCapacity: 2,147,483,647    Length: 19
//    Capacity: 64    MaxCapacity: 2,147,483,647    Length: 50
//    Capacity: 128    MaxCapacity: 2,147,483,647    Length: 81
//    Capacity: 128    MaxCapacity: 2,147,483,647    Length: 112
```

```
//    Capacity: 256    MaxCapacity: 2,147,483,647    Length: 143
//    Capacity: 256    MaxCapacity: 2,147,483,647    Length: 174
//    Capacity: 256    MaxCapacity: 2,147,483,647    Length: 205
//    Capacity: 256    MaxCapacity: 2,147,483,647    Length: 236
//    Capacity: 512    MaxCapacity: 2,147,483,647    Length: 267
//    Capacity: 512    MaxCapacity: 2,147,483,647    Length: 298
//    Capacity: 512    MaxCapacity: 2,147,483,647    Length: 329
//    Capacity: 512    MaxCapacity: 2,147,483,647    Length: 360
```

메모리 할당

개체의 StringBuilder 기본 용량은 16자이며 기본 최대 용량은 Int32.MaxValue입니다. 이러한 기본값은 생성 StringBuilder(String) 자를 호출하는 StringBuilder() 경우에 사용됩니다.

다음과 같은 방법으로 개체의 초기 용량을 StringBuilder 명시적으로 정의할 수 있습니다.

- 개체를 StringBuilder 만들 때 매개 변수를 `capacity` 포함하는 생성자를 호출합니다.
- 속성에 새 값을 명시적으로 할당하여 StringBuilder.Capacity 기존 StringBuilder 개체를 확장합니다. 새 용량이 기존 용량보다 작거나 개체의 최대 용량보다 큰 경우 속성이 예외를 StringBuilder throw합니다.
- 새 용량으로 StringBuilder.EnsureCapacity 메서드를 호출합니다. 새 용량은 개체의 최대 용량보다 StringBuilder 크지 않아야 합니다. 그러나 속성 EnsureCapacity 에 Capacity 대한 할당과 달리 원하는 새 용량이 기존 용량보다 작으면 예외를 throw하지 않습니다. 이 경우 메서드 호출은 영향을 주지 않습니다.

생성자 호출에서 개체에 StringBuilder 할당된 문자열의 길이가 기본 용량 또는 지정된 용량 Capacity 을 초과하는 경우 속성은 매개 변수로 지정된 문자열의 길이로 `value` 설정됩니다.

생성자를 호출하여 개체의 최대 용량을 StringBuilder 명시적으로 정의할 StringBuilder(Int32, Int32) 수 있습니다. 읽기 전용이므로 속성에 새 값을 MaxCapacity 할당하여 최대 용량을 변경할 수 없습니다.

이전 섹션에서 보여 주듯이 기존 용량이 부족할 때마다 추가 메모리가 할당되고 개체의 StringBuilder 용량이 속성에 정의된 MaxCapacity 값으로 두 배가 됩니다.

일반적으로 기본 용량과 최대 용량은 대부분의 앱에 적합합니다. 다음 조건에서 이러한 값을 설정하는 것이 좋습니다.

- 개체의 StringBuilder 최종 크기가 크게 증가할 가능성이 있는 경우 일반적으로 몇 메가바이트 초과합니다. 이 경우 너무 많은 메모리 재할당이 필요하지 않도록 초기 Capacity 속성을 상당히 높은 값으로 설정하면 성능이 약간 향상될 수 있습니다.

- 코드가 메모리가 제한된 시스템에서 실행 중인 경우 이 경우 코드가 메모리 제한 환경에서 실행될 수 있는 큰 문자열을 처리하는 경우보다 Int32.MaxValue 작게 속성을 설정하는 MaxCapacity 것이 좋습니다.

StringBuilder 개체 인스턴스화

다음 표에 StringBuilder 나열된 6개의 오버로드된 클래스 생성자 중 하나를 호출하여 개체를 인스턴스화합니다. 생성자 중 3개는 값이 빈 문자열인 개체를 인스턴스화 StringBuilder 하지만 값 Capacity 과 MaxCapacity 값을 다르게 설정합니다. 다시 기본 세 개의 생성자는 특정 문자열 값과 용량이 있는 개체를 정의 StringBuilder 합니다. 세 생성자 중 2개는 기본 최대 용량을 Int32.MaxValue사용하는 반면, 세 번째 생성자는 최대 용량을 설정할 수 있습니다.

테이블 확장

생성자	문자열 값	용량	최대 생산 능력
StringBuilder()	String.Empty	16	Int32.MaxValue
StringBuilder(Int32)	String.Empty	매개 변수에 `capacity` 의해 정의됨	Int32.MaxValue
StringBuilder(Int32, Int32)	String.Empty	매개 변수에 `capacity` 의해 정의됨	매개 변수에 `maxCapacity` 의해 정의됨
StringBuilder(String)	매개 변수에 `value` 의해 정의됨	16 또는 `value`. Length중 더 큰 값	Int32.MaxValue
StringBuilder(String, Int32)	매개 변수에 `value` 의해 정의됨	매개 변수 또는 `value`.에 `capacity` 의해 정의됩니다. Length을 선택합니다.	Int32.MaxValue
StringBuilder(String, Int32, Int32, Int32)	`value`에 의해 정의됩니다. Substring(`startIndex`, `length`)	매개 변수 또는 `value`.에 `capacity` 의해 정의됩니다. Length을 선택합니다.	Int32.MaxValue

다음 예제에서는 이러한 생성자 오버로드 중 세 가지를 사용하여 개체를 인스턴스화 StringBuilder 합니다.

```
C#
```

```
using System;
using System.Text;

public class Example8
{
    public static void Main()
    {
        string value = "An ordinary string";
        int index = value.IndexOf("An ") + 3;
        int capacity = 0xFFFF;

        // Instantiate a StringBuilder from a string.
        StringBuilder sb1 = new StringBuilder(value);
        ShowSBInfo(sb1);

        // Instantiate a StringBuilder from string and define a capacity.
        StringBuilder sb2 = new StringBuilder(value, capacity);
        ShowSBInfo(sb2);

        // Instantiate a StringBuilder from substring and define a capacity.
        StringBuilder sb3 = new StringBuilder(value, index,
                                              value.Length - index,
                                              capacity);
        ShowSBInfo(sb3);
    }

    public static void ShowSBInfo(StringBuilder sb)
    {
        Console.WriteLine("\nValue: {0}", sb.ToString());
        foreach (var prop in sb.GetType().GetProperties())
        {
            if (prop.GetIndexParameters().Length == 0)
                Console.Write("{0}: {1:N0}    ", prop.Name,
prop.GetValue(sb));
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    Value: An ordinary string
//    Capacity: 18    MaxCapacity: 2,147,483,647    Length: 18
//
//    Value: An ordinary string
//    Capacity: 65,535    MaxCapacity: 2,147,483,647    Length: 18
//
//    Value: ordinary string
//    Capacity: 65,535    MaxCapacity: 2,147,483,647    Length: 15
```

StringBuilder 메서드 호출

인스턴스에서 문자열을 수정하는 대부분의 메서드는 StringBuilder 동일한 인스턴스에 대한 참조를 반환합니다. 이렇게 하면 다음 두 가지 방법으로 메서드를 호출 StringBuilder 할 수 있습니다.

- 다음 예제와 같이 개별 메서드를 호출하고 반환 값을 무시할 수 있습니다.

```
C#
```

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      StringBuilder sb = new StringBuilder();
      sb.Append("This is the beginning of a sentence, ");
      sb.Replace("the beginning of ", "");
      sb.Insert(sb.ToString().IndexOf("a ") + 2, "complete ");
      sb.Replace(",", ".");
      Console.WriteLine(sb.ToString());
   }
}
// The example displays the following output:
//        This is a complete sentence.
```

- 단일 문에서 일련의 메서드 호출을 수행할 수 있습니다. 연속 작업을 연결 하는 단일 문을 작성 하려는 경우에 편리할 수 있습니다. 다음 예제에서는 이전 예제의 세 가지 메서드 호출을 한 줄의 코드로 통합합니다.

```
C#
```

```
using System;
using System.Text;

public class Example2
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("This is the beginning of
a sentence, ");
        sb.Replace("the beginning of ",
"").Insert(sb.ToString().IndexOf("a ") + 2,
                                                "complete
").Replace(",", ".");
        Console.WriteLine(sb.ToString());
    }
}
// The example displays the following output:
//        This is a complete sentence.
```

StringBuilder 작업 수행

클래스의 메서드를 StringBuilder 사용하여 개체의 문자를 StringBuilder 반복, 추가, 삭제 또는 수정할 수 있습니다.

StringBuilder 문자 반복

속성을 사용하여 StringBuilder.Chars[] 개체의 문자 StringBuilder 에 액세스할 수 있습니다. C# Chars[] 에서는 인덱서입니다. Visual Basic에서는 클래스의 StringBuilder 기본 속성입니다. 이렇게 하면 속성을 명시적으로 참조하지 않고 인덱스만 사용하여 개별 문자를 설정하거나 검색할 Chars[] 수 있습니다. 개체의 문자는 StringBuilder 인덱스 0(0)에서 시작하여 인덱 Length 스 - 1을 계속합니다.

다음 예제에서는 속성을 보여 줍니다 Chars[] . 개체에 10개의 난수를 추가한 StringBuilder 다음 각 문자를 반복합니다. 문자의 유니코드 범주인 UnicodeCategory.DecimalDigitNumber경우 숫자를 1로 줄이거나 값이 0인 경우 숫자를 9로 변경합니다. 이 예제에서는 개별 문자의 StringBuilder 값이 변경되기 전과 후에 개체의 내용을 표시합니다.

C#

```
using System;
using System.Globalization;
using System.Text;

public class Example3
{
    public static void Main()
    {
        Random rnd = new Random();
        StringBuilder sb = new StringBuilder();

        // Generate 10 random numbers and store them in a StringBuilder.
        for (int ctr = 0; ctr <= 9; ctr++)
            sb.Append(rnd.Next().ToString("N5"));

        Console.WriteLine("The original string:");
        Console.WriteLine(sb.ToString());

        // Decrease each number by one.
        for (int ctr = 0; ctr < sb.Length; ctr++)
        {
            if (Char.GetUnicodeCategory(sb[ctr]) ==
UnicodeCategory.DecimalDigitNumber)
            {
                int number = (int)Char.GetNumericValue(sb[ctr]);
                number--;
                if (number < 0) number = 9;
```

```
                sb[ctr] = number.ToString()[0];
            }
        }
        Console.WriteLine("\nThe new string:");
        Console.WriteLine(sb.ToString());
    }
}
// The example displays the following output:
//    The original string:
//
1,457,531,530.00000940,522,609.000001,668,113,564.000001,998,992,883.000001,
792,660,834.00
//
000101,203,251.000002,051,183,075.000002,066,000,067.000001,643,701,043.0000
01,702,382,508
//    .00000
//
//    The new string:
//
0,346,420,429.99999839,411,598.999990,557,002,453.999990,887,881,772.999990,
681,559,723.99
//
999090,192,140.999991,940,072,964.999991,955,999,956.999990,532,690,932.9999
90,691,271,497
//    .99999
```

Chars[] 속성에서 문자 기반 인덱싱을 사용하면 다음과 같은 조건 하에서 성능이 매우 느려질 수 있습니다.

- StringBuilder 인스턴스는 대규모입니다(예: 수만 개의 문자로 구성됨).
- "StringBuilder두툼한"입니다. 즉, 개체의 StringBuilder.Capacity 속성을 자동으로 확장하고 새 메모리 청크를 할당하는 등의 StringBuilder.Append 메서드 호출이 반복됩니다.

각 문자 액세스가 인덱싱할 올바른 버퍼를 찾기 위해 연결된 전체 청크 목록을 확인하기 때문에 성능이 심각하게 저하됩니다.

> ⓘ **참고**
>
> 큰 "청키" StringBuilder 개체의 경우에도 하나 또는 적은 수의 문자에 대한 인덱스 기반 액세스를 위해 속성을 사용하면 Chars[] 성능에 미치는 영향이 미미합니다. 일반적으로 O(n) **작업입니다**. StringBuilder 개체에서 문자를 반복할 때 성능에 상당한 영향이 발생합니다. **O(n^2)**개 작업에 영향을 줍니다.

StringBuilder 개체에서 문자 기반 인덱싱을 사용할 때 성능 문제가 발생하는 경우 다음 방법 중 하나를 사용할 수 있습니다.

- ToString 메서드를 호출하여 StringBuilder 인스턴스를 String으로 변환한 다음, 문자열의 문자에 액세스합니다.
- 기존 StringBuilder 개체의 콘텐츠를 크기가 미리 지정된 새로운 StringBuilder 개체로 복사합니다. 새로운 StringBuilder 개체가 청크가 아니기 때문에 성능이 향상됩니다. 예시:

```
C#
```

```
// sbOriginal is the existing StringBuilder object
var sbNew = new StringBuilder(sbOriginal.ToString(),
sbOriginal.Length);
```

- StringBuilder(Int32) 생성자를 호출하여 StringBuilder 개체의 초기 용량을 예상된 최대 크기와 대략 동일한 값으로 설정합니다. StringBuilder가 거의 최대 용량에 도달하더라도 메모리의 전체 블록을 할당합니다.

StringBuilder 개체에 텍스트 추가

클래스에는 StringBuilder 개체의 내용을 확장하기 위한 다음 메서드가 StringBuilder 포함됩니다.

- 이 메서드는 Append 문자열, 부분 문자열, 문자 배열, 문자 배열의 일부, 여러 번 반복되는 단일 문자 또는 기본 데이터 형식의 문자열 표현을 개체에 StringBuilder 추가합니다.
- 이 메서드는 AppendLine 줄 종결자 또는 문자열과 줄 종결자를 개체에 StringBuilder 추가합니다.
- 메서드는 AppendFormat 개체에 복합 형식 문자열 을 StringBuilder 추가합니다. 결과 문자열에 포함된 개체의 문자열 표현은 현재 시스템 문화권 또는 지정된 문화권의 서식 규칙을 반영할 수 있습니다.
- 메서드는 Insert 문자열, 부분 문자열, 문자열의 여러 반복, 문자 배열, 문자 배열의 일부 또는 개체의 지정된 위치에 기본 데이터 형식의 문자열 표현을 StringBuilder 삽입합니다. 위치는 0부터 시작하는 인덱스로 정의됩니다.

다음 예제에서는 , AppendLine및 InsertAppendFormat메서드를 사용하여 Append개체의 StringBuilder 텍스트를 확장합니다.

```
C#
```

```
using System;
using System.Text;
```

```
public class Example6
{
    public static void Main()
    {
        // Create a StringBuilder object with no text.
        StringBuilder sb = new StringBuilder();
        // Append some text.
        sb.Append('*', 10).Append(" Adding Text to a StringBuilder Object
").Append('*', 10);
        sb.AppendLine("\n");
        sb.AppendLine("Some code points and their corresponding
characters:");
        // Append some formatted text.
        for (int ctr = 50; ctr <= 60; ctr++)
        {
            sb.AppendFormat("{0,12:X4} {1,12}", ctr, Convert.ToChar(ctr));
            sb.AppendLine();
        }
        // Find the end of the introduction to the column.
        int pos = sb.ToString().IndexOf("characters:") + 11 +
                  Environment.NewLine.Length;
        // Insert a column header.
        sb.Insert(pos, String.Format("{2}{0,12:X4} {1,12}{2}", "Code Unit",
                                     "Character", "\n"));

        // Convert the StringBuilder to a string and display it.
        Console.WriteLine(sb.ToString());
    }
}
// The example displays the following output:
//    ********** Adding Text to a StringBuilder Object **********
//
//    Some code points and their corresponding characters:
//
//       Code Unit    Character
//            0032            2
//            0033            3
//            0034            4
//            0035            5
//            0036            6
//            0037            7
//            0038            8
//            0039            9
//            003A            :
//            003B            ;
//            003C            <
```

StringBuilder 개체에서 텍스트 삭제

클래스에는 StringBuilder 현재 StringBuilder 인스턴스의 크기를 줄일 수 있는 메서드가 포함되어 있습니다. 이 메서드는 Clear 모든 문자를 제거하고 속성을 0으로 설정합니다 Length . 메서드는 Remove 특정 인덱스 위치에서 시작하는 지정된 수의 문자를 삭제합니

다. 또한 개체의 속성을 현재 인스턴스의 StringBuilder 길이보다 작은 값으로 설정 Length 하여 개체의 끝에서 문자를 제거할 수 있습니다.

다음은 개체에서 StringBuilder 일부 텍스트를 제거하고 결과 용량, 최대 용량 및 길이 속성 값을 표시한 다음 메서드를 호출 Clear 하여 개체에서 StringBuilder 모든 문자를 제거하는 예제입니다.

C#

```
using System;
using System.Text;

public class Example5
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("A StringBuilder object");
        ShowSBInfo(sb);
        // Remove "object" from the text.
        string textToRemove = "object";
        int pos = sb.ToString().IndexOf(textToRemove);
        if (pos >= 0)
        {
            sb.Remove(pos, textToRemove.Length);
            ShowSBInfo(sb);
        }
        // Clear the StringBuilder contents.
        sb.Clear();
        ShowSBInfo(sb);
    }

    public static void ShowSBInfo(StringBuilder sb)
    {
        Console.WriteLine("\nValue: {0}", sb.ToString());
        foreach (var prop in sb.GetType().GetProperties())
        {
            if (prop.GetIndexParameters().Length == 0)
                Console.Write("{0}: {1:N0}    ", prop.Name,
prop.GetValue(sb));
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    Value: A StringBuilder object
//    Capacity: 22    MaxCapacity: 2,147,483,647    Length: 22
//
//    Value: A StringBuilder
//    Capacity: 22    MaxCapacity: 2,147,483,647    Length: 16
//
//    Value:
//    Capacity: 22    MaxCapacity: 2,147,483,647    Length: 0
```

StringBuilder 개체의 텍스트 수정

메서드는 StringBuilder.Replace 전체 StringBuilder 개체 또는 특정 문자 범위에서 문자 또는 문자열의 모든 발생을 대체합니다. 다음 예제에서는 메서드를 사용하여 개체의 Replace 모든 느낌표(!)를 물음표(?) StringBuilder 로 바꿉니다.

C#

```
using System;
using System.Text;

public class Example13
{
    public static void Main()
    {
        StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
        MyStringBuilder.Replace('!', '?');
        Console.WriteLine(MyStringBuilder);
    }
}
// The example displays the following output:
//       Hello World?
```

StringBuilder 개체의 텍스트 검색

클래스에는 StringBuilder 특정 문자 또는 부분 문자열에 대한 개체를 검색할 수 있도록 클래스에서 제공하는 String 메서드 및 클래스와 String.StartsWith 유사한 String.ContainsString.IndexOf메서드가 포함되지 않습니다. 부분 문자열의 현재 상태 또는 시작 문자 위치를 확인하려면 문자열 검색 메서드 또는 정규식 메서드를 사용하여 값을 검색 String 해야 합니다. 다음 표와 같이 이러한 검색을 구현하는 네 가지 방법이 있습니다.

테이블 확장

기법	장점	단점
개체에 추가 StringBuilder 하기 전에 문자열 값을 검색합니다.	부분 문자열이 있는지 여부를 확인하는 데 유용합니다.	부분 문자열의 인덱스 위치가 중요한 경우 사용할 수 없습니다.
반환 String 된 개체를 호출 ToString 하고 검색합니다.	개체에 모든 텍스트를 할당한 다음 수정을 StringBuilder 시작하는 경우 사용하기 쉽습니다.	개체에 모든 텍스트를 추가하기 전에 수정해야 하는 경우 반복적으로 호출 ToString 하는 것이 번거롭 StringBuilder 습니다. 변경하려는 경우 개체 텍스트의

기법	장점	단점
		StringBuilder 끝에서 작업해야 합니다.
Chars[] 속성을 사용하여 문자 범위를 순차적으로 검색합니다.	개별 문자 또는 작은 부분 문자열에 관심이 있는 경우에 유용합니다.	검색할 문자 수가 크거나 검색 논리가 복잡한 경우 번거롭습니다. 반복된 메서드 호출을 통해 매우 크게 성장한 개체의 성능이 매우 저하됩니다.
개체를 StringBuilder 개체로 String 변환하고 개체를 String 수정합니다.	수정 횟수가 작은 경우에 유용합니다.	수정 횟수가 큰 경우 클래스의 StringBuilder 성능 이점을 부정합니다.

이러한 기술을 좀 더 자세히 살펴보겠습니다.

- 검색의 목표가 특정 부분 문자열이 있는지 여부를 확인하는 것인 경우(즉, 부분 문자열의 위치에 관심이 없는 경우) 개체에 StringBuilder 저장하기 전에 문자열을 검색할 수 있습니다. 다음 예제에서는 하나의 가능한 구현을 제공합니다. 생성자가 개체에 `StringBuilderFinder` 대한 참조 StringBuilder 와 문자열에서 찾을 부분 문자열을 전달하는 클래스를 정의합니다. 이 경우 예제에서는 기록된 온도가 화씨 또는 섭씨인지 여부를 확인하고 개체의 StringBuilder 시작 부분에 적절한 입문 텍스트를 추가합니다. 난수 생성기는 섭씨 또는 화씨 도의 데이터를 포함하는 배열을 선택하는 데 사용됩니다.

C#

```
using System;
using System.Text;

public class Example9
{
    public static void Main()
    {
        Random rnd = new Random();
        string[] tempF = { "47.6F", "51.3F", "49.5F", "62.3F" };
        string[] tempC = { "21.2C", "16.1C", "23.5C", "22.9C" };
        string[][] temps = { tempF, tempC };

        StringBuilder sb = new StringBuilder();
        var f = new StringBuilderFinder(sb, "F");
        var baseDate = new DateTime(2013, 5, 1);
        String[] temperatures = temps[rnd.Next(2)];
        bool isFahrenheit = false;
        foreach (var temperature in temperatures)
        {
            if (isFahrenheit)
                sb.AppendFormat("{0:d}: {1}\n", baseDate, temperature);
```

```
            else
                isFahrenheit = f.SearchAndAppend(String.Format("{0:d}:
{1}\n",
                                                 baseDate,
temperature));
            baseDate = baseDate.AddDays(1);
        }
        if (isFahrenheit)
        {
            sb.Insert(0, "Average Daily Temperature in Degrees
Fahrenheit");
            sb.Insert(47, "\n\n");
        }
        else
        {
            sb.Insert(0, "Average Daily Temperature in Degrees
Celsius");
            sb.Insert(44, "\n\n");
        }
        Console.WriteLine(sb.ToString());
    }
}

public class StringBuilderFinder
{
    private StringBuilder sb;
    private String text;

    public StringBuilderFinder(StringBuilder sb, String textToFind)
    {
        this.sb = sb;
        this.text = textToFind;
    }

    public bool SearchAndAppend(String stringToSearch)
    {
        sb.Append(stringToSearch);
        return stringToSearch.Contains(text);
    }
}
// The example displays output similar to the following:
//    Average Daily Temperature in Degrees Celsius
//
//    5/1/2013: 21.2C
//    5/2/2013: 16.1C
//    5/3/2013: 23.5C
//    5/4/2013: 22.9C
```

- 메서드를 StringBuilder.ToString 호출하여 개체를 StringBuilder 개체로 String 변환합니다. 같은 String.LastIndexOf 메서드를 사용하여 문자열을 검색하거나 String.StartsWith정규식 및 클래스를 Regex 사용하여 패턴을 검색할 수 있습니다. 두 개체 모두 StringBuilderString UTF-16 인코딩을 사용하여 문자를 저장하기 때문

에 문자, 부분 문자열 및 정규식 일치의 인덱스 위치는 두 개체 모두에서 동일합니다. 이렇게 하면 메서드를 사용하여 StringBuilder 개체에서 해당 텍스트를 찾을 String 때와 동일한 위치에서 변경할 수 있습니다.

> ⓘ **참고**
>
> 이 방법을 채택하는 경우 개체를 문자열로 반복적으로 변환 **StringBuilder** 할 필요가 없도록 개체의 **StringBuilder** 끝에서 시작 부분으로 작업해야 합니다.

다음 예제에서 이 방법을 보여 줍니다. 개체에 영어 알파벳의 각 문자의 네 개의 발생을 StringBuilder 저장합니다. 그런 다음 텍스트를 개체로 String 변환하고 정규식을 사용하여 각 4자 시퀀스의 시작 위치를 식별합니다. 마지막으로 첫 번째 시퀀스를 제외한 각 4자 시퀀스 앞에 밑줄을 추가하고 시퀀스의 첫 번째 문자를 대문자로 변환합니다.

C#

```
using System;
using System.Text;
using System.Text.RegularExpressions;

public class Example10
{
    public static void Main()
    {
        // Create a StringBuilder object with 4 successive occurrences
        // of each character in the English alphabet.
        StringBuilder sb = new StringBuilder();
        for (ushort ctr = (ushort)'a'; ctr <= (ushort)'z'; ctr++)
            sb.Append(Convert.ToChar(ctr), 4);

        // Create a parallel string object.
        String sbString = sb.ToString();
        // Determine where each new character sequence begins.
        String pattern = @"(\w)\1+";
        MatchCollection matches = Regex.Matches(sbString, pattern);

        // Uppercase the first occurrence of the sequence, and separate
it
        // from the previous sequence by an underscore character.
        for (int ctr = matches.Count - 1; ctr >= 0; ctr--)
        {
            Match m = matches[ctr];
            sb[m.Index] = Char.ToUpper(sb[m.Index]);
            if (m.Index > 0) sb.Insert(m.Index, "_");
        }
        // Display the resulting string.
        sbString = sb.ToString();
        int line = 0;
```

```
        do
        {
            int nChars = line * 80 + 79 <= sbString.Length ?
                                 80 : sbString.Length - line * 80;
            Console.WriteLine(sbString.Substring(line * 80, nChars));
            line++;
        } while (line * 80 < sbString.Length);
    }
}
// The example displays the following output:
//
Aaaa_Bbbb_Cccc_Dddd_Eeee_Ffff_Gggg_Hhhh_Iiii_Jjjj_Kkkk_Llll_Mmmm_Nnnn_O
ooo_Pppp_
//    Qqqq_Rrrr_Ssss_Tttt_Uuuu_Vvvv_Wwww_Xxxx_Yyyy_Zzzz
```

- 개체의 StringBuilder.Chars[] 문자 StringBuilder 범위를 순차적으로 검색하려면 이 속성을 사용합니다. 검색할 문자 수가 크거나 검색 논리가 특히 복잡한 경우에는 이 방법이 실용적이지 않을 수 있습니다. 매우 큰 청크 StringBuilder 분할 개체에 대한 문자별 인덱스 기반 액세스의 성능에 미치는 영향은 속성에 대한 StringBuilder.Chars[] 설명서를 참조하세요.

  다음 예제는 이전 예제와 기능에서 동일하지만 구현은 다릅니다. 속성을 사용하여 Chars[] 문자 값이 변경된 시기를 감지하고, 해당 위치에 밑줄을 삽입하고, 새 시퀀스의 첫 번째 문자를 대문자로 변환합니다.

  C#

```
using System;
using System.Text;

public class Example11
{
    public static void Main()
    {
        // Create a StringBuilder object with 4 successive occurrences
        // of each character in the English alphabet.
        StringBuilder sb = new StringBuilder();
        for (ushort ctr = (ushort)'a'; ctr <= (ushort)'z'; ctr++)
            sb.Append(Convert.ToChar(ctr), 4);

        // Iterate the text to determine when a new character sequence
occurs.
        int position = 0;
        Char current = '\u0000';
        do
        {
            if (sb[position] != current)
            {
                current = sb[position];
                sb[position] = Char.ToUpper(sb[position]);
                if (position > 0)
```

```
                    sb.Insert(position, "_");
                position += 2;
            }
            else
            {
                position++;
            }
        } while (position <= sb.Length - 1);
        // Display the resulting string.
        String sbString = sb.ToString();
        int line = 0;
        do
        {
            int nChars = line * 80 + 79 <= sbString.Length ?
                                 80 : sbString.Length - line * 80;
            Console.WriteLine(sbString.Substring(line * 80, nChars));
            line++;
        } while (line * 80 < sbString.Length);
    }
}
// The example displays the following output:
//
Aaaa_Bbbb_Cccc_Dddd_Eeee_Ffff_Gggg_Hhhh_Iiii_Jjjj_Kkkk_Llll_Mmmm_Nnnn_O
ooo_Pppp_
//    Qqqq_Rrrr_Ssss_Tttt_Uuuu_Vvvv_Wwww_Xxxx_Yyyy_Zzzz
```

- 수정되지 않은 모든 텍스트를 개체에 StringBuilder 저장하고, 메서드를 StringBuilder.ToString 호출하여 개체를 개체로 String 변환 StringBuilder 하고, 개체에 대한 String 수정을 수행합니다. 몇 가지 수정 사항만 있는 경우 이 방법을 사용할 수 있습니다. 그렇지 않으면 변경할 수 없는 문자열을 사용하는 비용이 개체 사용 StringBuilder 의 성능 이점을 부정할 수 있습니다.

  다음 예제는 이전의 두 예제와 기능에서 동일하지만 구현은 다릅니다. 개체를 StringBuilder 만들고 개체로 String 변환한 다음 정규식을 사용하여 문자열에 대한 모든 재기본 수정을 수행합니다. 메서드는 Regex.Replace(String, String, MatchEvaluator) 람다 식을 사용하여 각 일치 항목에서 대체를 수행합니다.

  C#

```
using System;
using System.Text;
using System.Text.RegularExpressions;

public class Example12
{
    public static void Main()
    {
        // Create a StringBuilder object with 4 successive occurrences
        // of each character in the English alphabet.
        StringBuilder sb = new StringBuilder();
        for (ushort ctr = (ushort)'a'; ctr <= (ushort)'z'; ctr++)
```

```
            sb.Append(Convert.ToChar(ctr), 4);

        // Convert it to a string.
        String sbString = sb.ToString();

        // Use a regex to uppercase the first occurrence of the
sequence,
        // and separate it from the previous sequence by an underscore.
        string pattern = @"(\w)(\1+)";
        sbString = Regex.Replace(sbString, pattern,
                                 m => (m.Index > 0 ? "_" : "") +
                                 m.Groups[1].Value.ToUpper() +
                                 m.Groups[2].Value);

        // Display the resulting string.
        int line = 0;
        do
        {
            int nChars = line * 80 + 79 <= sbString.Length ?
                                  80 : sbString.Length - line * 80;
            Console.WriteLine(sbString.Substring(line * 80, nChars));
            line++;
        } while (line * 80 < sbString.Length);
    }
}
// The example displays the following output:
//
Aaaa_Bbbb_Cccc_Dddd_Eeee_Ffff_Gggg_Hhhh_Iiii_Jjjj_Kkkk_Llll_Mmmm_Nnnn_O
ooo_Pppp_
//    Qqqq_Rrrr_Ssss_Tttt_Uuuu_Vvvv_Wwww_Xxxx_Yyyy_Zzzz
```

StringBuilder 개체를 문자열로 변환

StringBuilder 개체에 표시되는 문자열을 String 매개 변수를 가진 메서드에 전달하거나 사용자 인터페이스에 표시하려면 StringBuilder 개체를 String 개체로 변환해야 합니다. 메서드를 호출하여 이 변환을 수행합니다 StringBuilder.ToString . 예를 들어 정규식 메서드에 전달할 수 있도록 개체를 문자열로 변환 StringBuilder 하기 위해 메서드를 호출 ToString 하는 이전 예제를 참조하세요.

제품 사용자 의견 제공

.NET의 규정 준수 라이브러리

아티클 • 2025. 04. 01.

.NET은 .NET 애플리케이션에서 데이터 분류 및 수정과 같은 규정 준수 기능을 구현하기 위한 기본 구성 요소 및 추상화 기능을 제공하는 라이브러리를 제공합니다. 이러한 추상화는 개발자가 표준화된 방식으로 데이터를 만들고 관리하는 데 도움이 됩니다. 이 문서에서는 데이터 분류 및 수정 준수 라이브러리에 대한 개요를 확인합니다.

.NET의 데이터 분류

데이터 분류는 DataClassification 구조를 사용하여 민감도 및 보호 수준에 따라 데이터를 분류하는 데 도움이 됩니다. 이렇게 하면 중요한 정보에 레이블을 지정하고 이러한 레이블에 따라 정책을 적용할 수 있습니다. 사용자 지정 분류 및 특성을 만들어 데이터에 적절하게 태그를 지정할 수 있습니다.

에 대한 자세한 내용은 . NET의 데이터 분류 라이브러리는 .NET 데이터 분류를 참조하세요.

.NET의 데이터 편집

데이터 편집은 개인 정보 규칙을 준수하고 중요한 데이터를 보호하기 위해 로그, 오류 메시지 또는 기타 출력의 중요한 정보를 보호하는 데 도움이 됩니다.
Microsoft.Extensions.Compliance.Redaction 라이브러리는 ErasingRedactor 및 HmacRedactor같은 다양한 편집기를 제공합니다. 이러한 편집기를 구성하고 `AddRedaction` 메서드를 사용하여 등록할 수 있습니다. 또한 특정 요구 사항에 맞게 사용자 지정 편집기 및 재배포 공급자를 만들 수 있습니다.

.NET의 데이터 삭제 라이브러리에 대한 자세한 내용은 .NET의 데이터 삭제에서 참조하세요.

.NET의 데이터 분류

2025. 05. 21.

데이터 분류는 민감도 및 보호 수준에 따라 데이터를 분류(또는 분류)하는 데 도움이 됩니다. DataClassification 구조를 사용하면 중요한 정보에 레이블을 지정하고 이러한 레이블에 따라 정책을 적용할 수 있습니다.

- DataClassification.TaxonomyName: 분류 시스템을 식별합니다.
- DataClassification.Value: 분류 내의 특정 레이블을 나타냅니다.

경우에 따라 데이터에 명시적으로 데이터 분류가 없음을 지정해야 할 수 있습니다. 이 작업은 DataClassification.None을 사용하여 수행됩니다. 마찬가지로 데이터 분류를 알 수 없음을 지정해야 할 수 있습니다. 이러한 경우 DataClassification.Unknown 사용합니다.

패키지 설치

시작하려면 Microsoft.Extensions.Compliance.Abstractions NuGet 패키지를 설치합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Compliance.Abstractions
```

또는 .NET 10+ SDK를 사용하는 경우:

.NET CLI

```
dotnet package add Microsoft.Extensions.Compliance.Abstractions
```

사용자 지정 분류 만들기

다양한 유형의 중요한 데이터에 대한 `static` 멤버를 만들어 사용자 지정 분류를 정의합니다. 이렇게 하면 앱 전체에서 데이터에 레이블을 지정하고 처리할 수 있는 일관된 방법이 제공됩니다. 다음 예제 클래스를 고려합니다.

C#

```
using Microsoft.Extensions.Compliance.Classification;

internal static class MyTaxonomyClassifications
```

```
{
    internal static string Name => "MyTaxonomy";

    internal static DataClassification PrivateInformation => new(Name,
nameof(PrivateInformation));
    internal static DataClassification CreditCardNumber => new(Name,
nameof(CreditCardNumber));
    internal static DataClassification SocialSecurityNumber => new(Name,
nameof(SocialSecurityNumber));

    internal static DataClassificationSet PrivateAndSocialSet =>
new(PrivateInformation, SocialSecurityNumber);
}
```

사용자 지정 분류를 다른 앱과 공유하려는 경우 이 클래스와 해당 멤버는 `public` 대신 `internal` 합니다. 예를 들어 여러 애플리케이션에서 사용할 수 있는 사용자 지정 분류를 포함하는 공유 라이브러리를 가질 수 있습니다.

DataClassificationSet 여러 데이터 분류를 단일 집합으로 작성할 수 있습니다. 이렇게 하면 여러 데이터 분류를 사용하여 데이터를 분류할 수 있습니다. 또한 .NET 편집 API는 DataClassificationSet사용합니다.

> ⓘ **참고**
>
> 여러 데이터 분류가 함께 DataClassificationSet로 결합되어 단일 분류로 처리됩니다. 논리 `AND` 작업으로 생각할 수 있습니다. 예를 들어, DataClassificationSet와 `PrivateInformation`, `SocialSecurityNumber`로 분류된 데이터에 대한 편집을 구성한 경우, 이는 `PrivateInformation`로만 분류되거나 `SocialSecurityNumber`로만 분류된 데이터에는 적용되지 않습니다.

사용자 지정 분류 특성 만들기

사용자 지정 분류에 따라 사용자 지정 특성을 만듭니다. 이러한 특성을 사용하여 올바른 분류를 사용하여 데이터에 태그를 지정합니다. 다음 사용자 지정 특성 클래스 정의를 고려합니다.

C#

```
public sealed class PrivateInformationAttribute : DataClassificationAttribute
{
    public PrivateInformationAttribute()
        : base(MyTaxonomyClassifications.PrivateInformation)
    {
    }
}
```

앞의 코드는 DataClassificationAttribute 형식의 하위 클래스인 개인 정보 특성을 선언합니다. 매개변수가 없는 생성자를 정의하고, 그 생성자가 사용자 지정 DataClassification를 해당 `base`로 전달합니다.

데이터 분류 설정을 연결하기

데이터 분류 설정을 바인딩하려면 .NET 구성 시스템을 사용합니다. 예를 들어 JSON 구성 공급자를 사용 중이라고 가정하면 다음과 같이 *appsettings.json* 정의할 수 있습니다.

JSON

```
{
    "Key": {
        "PhoneNumber": "MyTaxonomy:PrivateInformation",
        "ExampleDictionary": {
            "CreditCard": "MyTaxonomy:CreditCardNumber",
            "SSN": "MyTaxonomy:SocialSecurityNumber"
        }
    }
}
```

이제 이러한 구성 설정을 `TestOptions` 개체에 바인딩하는 다음 옵션 패턴 방법을 고려합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Compliance.Classification;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Options;

public class TestOptions
{
    public DataClassification? PhoneNumber { get; set; }
    public IDictionary<string, DataClassification> ExampleDictionary { get; set; }
= new Dictionary<string, DataClassification>();
}

class Program
{
    static void Main(string[] args)
    {
        // Build configuration from an external json file.
        IConfiguration configuration = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
true)
            .Build();

        // Setup DI container and bind the configuration section "Key" to
TestOptions.
```

```
        IServiceCollection services = new ServiceCollection();
        services.Configure<TestOptions>(configuration.GetSection("Key"));

        // Build the service provider.
        IServiceProvider serviceProvider = services.BuildServiceProvider();

        // Get the bound options.
        TestOptions options =
serviceProvider.GetRequiredService<IOptions<TestOptions>>().Value;

        // Simple output demonstrating binding results.
        Console.WriteLine("Configuration bound to TestOptions:");
        Console.WriteLine($"PhoneNumber: {options.PhoneNumber}");
        foreach (var item in options.ExampleDictionary)
        {
            Console.WriteLine($"{item.Key}: {item.Value}");
        }
    }
}
```

.NET의 데이터 편집

2025. 06. 07.

편집을 사용하면 로그, 오류 메시지 또는 기타 출력에서 중요한 정보를 삭제하거나 마스킹할 수 있습니다. 이렇게 하면 개인 정보 보호 규칙을 준수하고 중요한 데이터를 보호합니다. 개인 데이터, 재무 정보 또는 기타 기밀 데이터 요소를 처리하는 앱에서 유용합니다.

편집 패키지 설치

시작하려면 Microsoft.Extensions.Compliance.Redaction NuGet 패키지를 설치 합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Compliance.Redaction
```

또는 .NET 10+ SDK를 사용하는 경우:

.NET CLI

```
dotnet package add Microsoft.Extensions.Compliance.Redaction
```

사용 가능한 편집기

편집기는 중요한 데이터를 수정하는 작업을 담당합니다. 중요한 정보를 수정, 바꾸기 또는 마스크합니다. 라이브러리에서 제공하는 다음과 같은 사용 가능한 편집기를 고려합니다.

- 입력 ErasingRedactor 을 빈 문자열로 바꿉니다.
- HmacRedactor 편집 중인 데이터를 인코딩하는 데 사용됩니다`HMACSHA256`.

사용 예제

기본 제공 편집기를 사용하려면 필요한 서비스를 등록해야 합니다. 다음 목록에 설명된 대로 사용 가능한 `AddRedaction` 방법 중 하나를 사용하여 서비스를 등록합니다.

- AddRedaction(IServiceCollection): IRedactorProvider에 IServiceCollection의 구현을 등록합니다.

- AddRedaction(IServiceCollection, Action<IRedactionBuilder>): IRedactorProvider에 IServiceCollection의 구현을 등록하고, 주어진 `configure` 대리자를 사용하여 이용 가능한 수정자를 구성합니다.

편집기 구성

IRedactorProvider을(를) 사용하여 런타임에 편집기를 가져옵니다. 사용자 고유의 공급자를 구현하고 호출 내에 `AddRedaction` 등록하거나 기본 공급자를 사용할 수 있습니다. 다음 IRedactionBuilder 메서드를 사용하여 편집기를 구성합니다.

C#

```
// This will use the default redactor, which is the ErasingRedactor
var serviceCollection = new ServiceCollection();
serviceCollection.AddRedaction();

// Using the default redactor provider:
serviceCollection.AddRedaction(redactionBuilder =>
{
    // Assign a redactor to use for a set of data classifications.
    redactionBuilder.SetRedactor<StarRedactor>(
        MyTaxonomyClassifications.Private,
        MyTaxonomyClassifications.Personal);
    // Assign a fallback redactor to use when processing classified data for which
no specific redactor has been registered.
    // The `ErasingRedactor` is the default fallback redactor. If no redactor is
configured for a data classification then the data will be erased.
    redactionBuilder.SetFallbackRedactor<MyFallbackRedactor>();
});

// Using a custom redactor provider:
builder.Services.AddSingleton<IRedactorProvider, StarRedactorProvider>();
```

코드에서 이 데이터 분류를 지정합니다.

C#

```
public static class MyTaxonomyClassifications
{
    public static string Name => "MyTaxonomy";

    public static DataClassification Private => new(Name, nameof(Private));
    public static DataClassification Public => new(Name, nameof(Public));
    public static DataClassification Personal => new(Name, nameof(Personal));
}
```

HMAC 적출기 구성

다음 IRedactionBuilder 메서드를 사용하여 HMAC 편집기를 구성합니다.

C#

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddRedaction(builder =>
{
    builder.SetHmacRedactor(
        options =>
        {
            options.KeyId = 1234567890;
            options.Key =
Convert.ToBase64String("1234567890abcdefghijklmnopqrstuvwxyz");
        },

        // Any data tagged with Personal or Private attributes will be redacted by
the Hmac redactor.
        MyTaxonomyClassifications.Personal, MyTaxonomyClassifications.Private,

        // "DataClassificationSet" lets you compose multiple data classifications:
        // For example, here the Hmac redactor will be used for data tagged
        // with BOTH Personal and Private (but not one without the other).
        new DataClassificationSet(MyTaxonomyClassifications.Personal,
                                  MyTaxonomyClassifications.Private));
});
```

또는 다음과 같이 구성합니다.

C#

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddRedaction(builder =>
{
    builder.SetHmacRedactor(
        Configuration.GetSection("HmacRedactorOptions"),
MyTaxonomyClassifications.Personal);
});
```

JSON 구성 파일에 다음 섹션을 포함합니다.

JSON

```
{
    "HmacRedactorOptions": {
        "KeyId": 1234567890,
        "Key": "1234567890abcdefghijklmnopqrstuvwxyz"
    }
}
```

- HmacRedactorOptions에는 해당 HmacRedactorOptions.Key 속성과 HmacRedactorOptions.KeyId 속성이 설정되어야 합니다.
- `Key`는 base 64 형식이어야 하며, 길이가 최소 44자 이상이어야 합니다. 서비스의 각 주요 배포에 고유한 키를 사용합니다. 키 자료를 비밀로 유지하고 정기적으로 변경합니다.
- `KeyId`는 데이터를 해시하는 데 사용된 키를 식별하기 위해 수정된 각 값에 추가됩니다.
- 키 ID가 다르면 값이 관련이 없으며 상관 관계에 사용할 수 없습니다.

> ⓘ **참고**
>
> HmacRedactor는 여전히 실험 단계에 있으므로, 앞서 언급한 방법으로 인해 `EXTEXP0002` 경고가 발생하며 안정 상태에 있지 않음을 나타냅니다. 이를 사용하려면 프로젝트 파일에 `<NoWarn>$(NoWarn);EXTEXP0002</NoWarn>`을 추가하거나 `#pragma warning disable EXTEXP0002`을 호출 앞뒤에 추가하십시오 `SetHmacRedactor`.

사용자 지정 편집기 구성

사용자 지정 편집기를 만들려면 Redactor로부터 상속하는 하위 클래스를 정의하세요.

C#

```
public sealed class StarRedactor : Redactor

public class StarRedactor : Redactor
{
    private const string Stars = "****";

    public override int GetRedactedLength(ReadOnlySpan<char> input) =>
Stars.Length;

    public override int Redact(ReadOnlySpan<char> source, Span<char> destination)
    {
        Stars.CopyTo(destination);

        return Stars.Length;
    }
}
```

사용자 지정 편집자 공급자 만들기

인터페이스는 IRedactorProvider 데이터 분류에 따라 편집기의 인스턴스를 제공합니다. 사용자 지정 편집기 공급자를 만들려면 다음 예제와 IRedactorProvider 같이 상속합니다.

C#

```
using Microsoft.Extensions.Compliance.Classification;
using Microsoft.Extensions.Compliance.Redaction;

public sealed class StarRedactorProvider : IRedactorProvider
{
    private static readonly StarRedactor _starRedactor = new();

    public static StarRedactorProvider Instance { get; } = new();

    public Redactor GetRedactor(DataClassificationSet classifications) =>
_starRedactor;
}
```

중요한 정보 로깅

로깅은 실수로 인한 데이터 노출의 일반적인 소스입니다. 개인 데이터, 자격 증명 또는 재무 세부 정보와 같은 중요한 정보는 일반 텍스트로 로그에 기록해서는 안 됩니다. 이를 방지하려면 잠재적으로 중요한 데이터를 로깅할 때 항상 편집을 사용합니다.

중요한 데이터를 로깅하는 단계

1. **원격 분석 확장 패키지 설치**: 확장 로거를 사용하여 수정 기능을 사용하도록 설정할 수 있도록 Microsoft.Extensions.Telemetry 를 설치합니다.
2. **수정 설정**: 메서드를 호출 AddRedaction(IServiceCollection) 하여 편집기를 로깅 파이프라인과 통합하여 중요한 필드가 로그에 기록되기 전에 자동으로 삭제하거나 마스킹합니다.
3. **중요한 필드 식별**: 애플리케이션에서 중요한 데이터와 보호가 필요한 데이터를 알고 적절한 데이터 분류로 표시합니다.
4. **로그 출력 검토**: 중요한 데이터가 노출되지 않도록 로그를 정기적으로 감사하세요.

예: 로그에서 데이터 수정

Microsoft.Extensions.Logging을 사용하는 경우 다음과 같이 편집을 로깅과 결합할 수 있습니다.

C#

```
using Microsoft.Extensions.Telemetry;
using Microsoft.Extensions.Compliance.Redaction;

var services = new ServiceCollection();
services.AddLogging(builder =>
{
    // Enable redaction.
    builder.EnableRedaction();
```

```
});

services.AddRedaction(builder =>
{
    // configure redactors for your data classifications
    builder.SetRedactor<StarRedactor>(MyTaxonomyClassifications.Private);
});
// Use annotations to mark sensitive data.
// For example, apply the Private classification to SSN data.
[LoggerMessage(0, LogLevel.Information, "User SSN: {SSN}")]
public static partial void LogPrivateInformation(
    this ILogger logger,
    [MyTaxonomyClassifications.Private] string SSN);

public void TestLogging()
{
    LogPrivateInformation("MySSN");
}
```

출력은 다음과 같아야 합니다.

```
User SSN: *****
```

이렇게 하면 중요한 데이터가 기록되기 전에 수정되어 데이터 유출 위험이 줄어듭니다.

.NET 정규식

정규식은 텍스트를 처리하는 강력하고 유연하며 효율적인 방법을 제공합니다. 정규식의 광범위한 패턴 일치 표기법을 사용하면 대량의 텍스트를 신속하게 구문 분석하여 다음을 수행할 수 있습니다.

- 특정 문자 패턴을 찾습니다.
- 텍스트의 유효성을 검사하여 미리 정의된 패턴(예: 전자 메일 주소)과 일치하는지 확인합니다.
- 텍스트 하위 문자열을 추출, 편집, 바꾸기 또는 삭제합니다.
- 보고서를 생성하기 위해 추출된 문자열을 컬렉션에 추가합니다.

문자열을 처리하거나 큰 텍스트 블록을 구문 분석하는 많은 애플리케이션에서 정규식은 필수 도구입니다.

정규식의 작동 방식

정규식을 사용한 텍스트 처리의 중심은 .NET의 System.Text.RegularExpressions.Regex 개체로 표현되는 정규식 엔진입니다. 최소한 정규식을 사용하여 텍스트를 처리하려면 정규식 엔진에 다음 두 가지 정보 항목이 제공되어야 합니다.

- 텍스트에서 식별할 정규식 패턴입니다.

  .NET에서 정규식 패턴은 Perl 5 정규식과 호환되는 특수 구문 또는 언어로 정의되며 오른쪽에서 왼쪽 일치와 같은 다른 기능을 추가합니다. 자세한 내용은 정규식 언어 - 빠른 참조참조하세요.

- 정규식 패턴을 위한 구문 분석 텍스트입니다.

Regex 클래스의 메서드를 사용하면 다음 작업을 수행할 수 있습니다.

- Regex.IsMatch 메서드를 호출하여 입력 텍스트에서 정규식 패턴이 발생하는지 여부를 확인합니다. IsMatch 메서드를 사용하여 텍스트의 유효성을 검사하는 예제는 방법: 문자열이 유효한 전자 메일 형식인지 확인.

- Regex.Match 또는 Regex.Matches 메서드를 호출하여 정규식 패턴과 일치하는 텍스트를 하나 또는 모두 검색합니다. 이전 메서드는 일치하는 텍스트에 대한 정보를 제공하는 System.Text.RegularExpressions.Match 개체를 반환합니다. 후자는 구문 분석된 텍스트에 있는 각 일치 항목에 대해 하나의 MatchCollection 개체를 포함하는 System.Text.RegularExpressions.Match 개체를 반환합니다.

- Regex.Replace 메서드를 호출하여 정규식 패턴과 일치하는 텍스트를 바꿉다. Replace 메서드를 사용하여 날짜 서식을 변경하고 문자열에서 잘못된 문자를 제거하는 예제는 방법: 문자열 잘못된 문자 제거 및 예제: 날짜 서식 변경참조하세요.

정규식 개체 모델에 대한 개요는 정규식 개체 모델참조하세요.

정규식 언어에 대한 자세한 내용은 정규식 언어 - 빠른 참조 참조하거나 다음 브로셔 중 하나를 다운로드하여 인쇄합니다.

- Word 포맷(.docx) 형식 빠른 참조
- 빠른 참조 PDF (.pdf) 형식

정규식 예제

String 클래스에는 더 큰 문자열에서 리터럴 문자열을 찾으려는 경우 사용할 수 있는 문자열 검색 및 대체 메서드가 포함됩니다. 정규식은 더 큰 문자열에서 여러 부분 문자열 중 하나를 찾으려는 경우 또는 다음 예제와 같이 문자열의 패턴을 식별하려는 경우에 가장 유용합니다.

> ⚠ **Warning**
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions` 입력을 제공하여 **서비스 거부 공격** 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

> 💡 **팁**
>
> 네임스페이스에는 System.Web.RegularExpressions HTML, XML 및 ASP.NET 문서에서 문자열을 구문 분석하기 위해 미리 정의된 정규식 패턴을 구현하는 많은 정규식 개체가 포함되어 있습니다. 예를 들어 TagRegex 클래스는 문자열의 시작 태그를 식별하고 CommentRegex 클래스는 문자열의 ASP.NET 주석을 식별합니다.

예제 1: 부분 문자열 바꾸기

메일링 목록에 이름 및 성과 함께 제목(Mr., Mrs., Miss 또는 Ms.)이 포함된 이름이 포함되어 있다고 가정합니다. 목록에서 봉투 레이블을 생성할 때 제목을 포함하지 않으려는 경우를 가정해 보겠습니다. 이 경우 다음 예제와 같이 정규식을 사용하여 제목을 제거할 수 있습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
```

```
   public static void Main()
   {
      string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";
      string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",
                         "Abraham Adams", "Ms. Nicole Norris" };
      foreach (string name in names)
         Console.WriteLine(Regex.Replace(name, pattern, String.Empty));
   }
}
// The example displays the following output:
//    Henry Hunt
//    Sara Samuels
//    Abraham Adams
//    Nicole Norris
```

정규식 패턴 `(Mr\.? |Mrs\.? |Miss |Ms\.? )`은 "Mr ", "Mr. ", "Mrs ", "Mrs. ", "Miss ", "Ms ", 또는 "Ms. "에 해당하는 모든 경우를 찾습니다. Regex.Replace 메서드에 대한 호출은 일치하는 문자열을 String.Empty; 로 바꿉니다. 즉, 원래 문자열에서 제거합니다.

예제 2: 중복 단어 식별

실수로 단어를 복제하는 것은 작성자가 만드는 일반적인 오류입니다. 다음 예제와 같이 정규식을 사용하여 중복된 단어를 식별합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Class1
{
   public static void Main()
   {
      string pattern = @"\b(\w+?)\s\1\b";
      string input = "This this is a nice day. What about this? This tastes good. I
saw a a dog.";
      foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
         Console.WriteLine($"{match.Value} (duplicates '{match.Groups[1].Value}')
at position {match.Index}");
   }
}
// The example displays the following output:
//       This this (duplicates 'This') at position 0
//       a a (duplicates 'a') at position 66
```

정규식 패턴 `\b(\w+?)\s\1\b` 다음과 같이 해석할 수 있습니다.

테이블 확장

패턴	해석
`\b`	시작점은 단어 경계에서입니다.
`(\w+?)`	하나 이상의 단어 문자를 가능한 한 적게 일치시킵니다. 함께 `\1`라고 할 수 있는 그룹을 형성합니다.
`\s`	공백 문자를 일치시킵니다.
`\1`	이름이 `\1`인 그룹에 해당하는 부분 문자열과 일치합니다.
`\b`	단어 경계를 찾습니다.

Regex.Matches 메서드는 정규식 옵션을 RegexOptions.IgnoreCase설정하여 호출됩니다. 따라서 일치 작업은 대/소문자를 구분하지 않으며, 이 예제에서는 부분 문자열 "This this"를 중복으로 식별합니다.

입력 문자열에 "this?"라는 하위 문자열이 포함되어 있습니까? 이". 그러나 중간 문장 부호 때문에 중복으로 식별되지 않습니다.

예제 3: 문화권 구분 정규식을 동적으로 빌드

다음 예제에서는 .NET 세계화 기능이 제공하는 유연성과 결합된 정규식의 기능을 보여 줍니다. NumberFormatInfo 개체를 사용하여 미국 영어 문화권에서 통화 값의 형식을 결정합니다. 그런 다음 해당 정보를 사용하여 텍스트에서 통화 값을 추출하는 정규식을 동적으로 생성합니다. 각 일치 항목에 대해 숫자 문자열만 포함된 하위 그룹을 추출하고, Decimal 값으로 변환하고, 실행 합계를 계산합니다. 이 예제에서는 en-US 문화권을 사용하지만 정규식을 생성하고 적절한 CultureInfo 설정으로 논리를 구문 분석하여 다른 문화권에 동일한 기술을 적용할 수 있습니다.

C#

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define text to be parsed.
      string input = "Office expenses on 2/13/2008:\n" +
                     "Paper (500 sheets)                    $3.95\n" +
                     "Pencils (box of 10)                   $1.00\n" +
                     "Pens (box of 10)                      $4.49\n" +
                     "Erasers                               $2.19\n" +
```

```
                     "Ink jet printer                          $69.95\n\n" +
                     "Total Expenses                         $ 81.58\n";

      // Get the en-US culture and its NumberFormatInfo object.
      CultureInfo enUsCulture = CultureInfo.CreateSpecificCulture("en-US");
      NumberFormatInfo nfi = enUsCulture.NumberFormat;
      // Assign needed property values to variables.
      string currencySymbol = nfi.CurrencySymbol;
      bool symbolPrecedesIfPositive = nfi.CurrencyPositivePattern % 2 == 0;
      string groupSeparator = nfi.CurrencyGroupSeparator;
      string decimalSeparator = nfi.CurrencyDecimalSeparator;

      // Form regular expression pattern.
      string pattern = Regex.Escape( symbolPrecedesIfPositive ? currencySymbol :
"") +
                       @"\s*[-+]?" + "([0-9]{0,3}(" + groupSeparator + "[0-9]{3})*
(" +
                       Regex.Escape(decimalSeparator) + "[0-9]+)?)" +
                       (! symbolPrecedesIfPositive ? currencySymbol : "");
      Console.WriteLine( "The regular expression pattern is:");
      Console.WriteLine("   " + pattern);

      // Get text that matches regular expression pattern.
      MatchCollection matches = Regex.Matches(input, pattern,

RegexOptions.IgnorePatternWhitespace);
      Console.WriteLine($"Found {matches.Count} matches.");

      // Get numeric string, convert it to a value, and add it to List object.
      List<decimal> expenses = new List<Decimal>();

      foreach (Match match in matches)
         expenses.Add(Decimal.Parse(match.Groups[1].Value, NumberStyles.Number,
nfi));

      // Determine whether total is present and if present, whether it is correct.
      decimal total = 0;
      foreach (decimal value in expenses)
         total += value;

      if (total / 2 == expenses[expenses.Count - 1])
         Console.WriteLine(string.Format(enUsCulture, "The expenses total {0:C2}.",
expenses[expenses.Count - 1]));
      else
         Console.WriteLine(string.Format(enUsCulture, "The expenses total {0:C2}.",
total));
   }
}
// The example displays the following output:
//       The regular expression pattern is:
//          \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)
//       Found 6 matches.
//       The expenses total $81.58.
```

이 예제에서는 en-US 문화권을 사용하고 정규식을 `\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)` 동적으로 작성합니다. 이 정규식 패턴은 다음과 같이 해석할 수 있습니다.

테이블 확장

패턴	해석
`\$`	입력 문자열에서 달러 기호(`$`)의 단일 항목을 찾습니다. 정규식 패턴 문자열에는 달러 기호가 정규식 앵커가 아닌 문자 그대로 해석되어야 함을 나타내는 백슬래시를 포함합니다. `$` 기호만으로 정규식 엔진이 문자열의 끝에서 일치를 시작해야 함을 나타냅니다. 현재 문화권의 통화 기호가 정규식 기호로 잘못 해석되지 않도록 하기 위해 예제에서는 Regex.Escape 메서드를 호출하여 문자를 이스케이프합니다.
`\s*`	공백 문자가 0개 이상 포함되어 있는지 찾습니다.
`[-+]?`	양수 또는 음수 부호가 0개 또는 1개 있는지를 찾습니다.
`([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)`	외부 괄호는 이 식을 캡처링 그룹 또는 하위 식으로 정의합니다. 일치 항목이 발견되면 일치하는 문자열의 이 부분에 대한 정보를 Group 속성에서 반환된 GroupCollection 개체의 두 번째 Match.Groups 개체에서 검색할 수 있습니다. 컬렉션의 첫 번째 요소는 전체 일치를 나타냅니다.
`[0-9]{0,3}`	0에서 9까지의 소수 자릿수를 0~3번 찾습니다.
`(,[0-9]{3})*`	0개 이상의 그룹 구분 기호에 이어지는 세 자리 숫자를 찾습니다.
`\.`	소수 구분 기호가 한 번만 나타나는 경우를 찾으세요.
`[0-9]+`	하나 이상의 소수 자릿수를 찾습니다.
`(\.[0-9]+)?`	소수 구분 기호가 0개 또는 1개, 10진수 이상인 경우를 찾습니다.

입력 문자열 내에 각 하위 패턴이 있으면 일치 성공이며, 일치에 대한 정보를 포함하는 Match 개체가 MatchCollection 개체에 추가됩니다.

관련 문서

테이블 확장

제목	설명
정규식 언어 - 빠른 참조	정규식을 정의하는 데 사용할 수 있는 문자, 연산자 및 구문 집합에 대한 정보를 제공합니다.
정규 표현식 객체 모델	정규식 클래스를 사용하는 방법을 보여 주는 정보 및 코드 예제를 제공합니다.

제목	설명
정규식 동작에 대한 의 세부 정보	.NET 정규식의 기능 및 동작에 대한 정보를 제공합니다.
Visual Studio에서 정규식 사용	

참고 문헌

- System.Text.RegularExpressions
- System.Text.RegularExpressions.Regex
- 정규 표현식 - 빠른 참조(Word 문서 형식으로 다운로드)
- 정규식 - 빠른 참조(PDF 형식으로 다운로드)

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 18.

정규식 언어 - 빠른 참조

아티클 • 2024. 11. 21.

정규식은 정규식 엔진이 입력 텍스트에서 찾으려고 하는 패턴입니다. 패턴은 하나 이상의 문자 리터럴, 연산자 또는 구문으로 구성됩니다. 간략하게 살펴보려면 .NET 정규식을 참조하세요.

이 빠른 참조의 각 단원에서는 정규식을 정의하는 데 사용할 수 있는 특정 범주의 문자, 연산자 및 구문을 보여줍니다.

또한 쉽게 참조할 수 있도록 다운로드 및 인쇄할 수 있는 다음과 같은 두 가지 형식으로 이 정보를 제공했습니다.

- Word(.docx) 형식으로 다운로드
- PDF(.pdf) 형식으로 다운로드

문자 이스케이프

정규식의 백슬래시 문자(\)는 뒤에 있는 문자가 다음 표와 같이 특수 문자이거나 문자 그대로 해석되어야 했음을 나타냅니다. 자세한 내용은 문자 이스케이프를 참조하세요.

테이블 확장

이스케이프된 문자	설명	패턴	일치
`\a`	벨 문자인 \u0007을 찾습니다.	`\a`	`"Error!" + '\u0007'`의 `"\u0007"`
`\b`	문자 클래스에서 백스페이스 문자인 \u0008을 찾습니다.	`[\b]{3,}`	`"\b\b\b\b"`의 `"\b\b\b\b"`
`\t`	탭 문자인 \u0009를 찾습니다.	`(\w+)\t`	`"item1\titem2\t"`의 `"item1\t"`, `"item2\t"`
`\r`	캐리지 리턴 문자인 \u000D를 찾습니다. `\r` 은 줄 바꿈 문자인 `\n`과 다릅니다.	`\r\n(\w+)`	`"\r\nThese are\ntwo lines."`의 `"\r\nThese"`
`\v`	세로 탭 문자인 \u000B를 찾습니다.	`[\v]{2,}`	`"\v\v\v"`의 `"\v\v\v"`
`\f`	용지 공급 문자인 \u000C를 찾습니다.	`[\f]{2,}`	`"\f\f\f"`의 `"\f\f\f"`
`\n`	줄 바꿈 문자인 \u000A를 찾습니다.	`\r\n(\w+)`	`"\r\nThese are\ntwo lines."`의 `"\r\nThese"`

이스케이프된 문자	설명	패턴	일치
`\e`	이스케이프 문자인 \u001B를 찾습니다.	`\e`	`"\x001B"`의 `"\x001B"`
`\` *nnn*	8진수 표현을 사용하여 문자를 지정합니다.*nnn*은 두 자리 또는 세 자리로 구성됩니다.	`\w\040\w`	`"a bc d"`의 `"a b"`, `"c d"`
`\x` *nn*	16진수 표현을 사용하여 문자를 지정합니다. (*nn*은 정확히 두 자리로 구성됩니다.)	`\w\x20\w`	`"a bc d"`의 `"a b"`, `"c d"`
`\c` *X* `\c` *x*	*X* 또는 *x*로 지정한 ASCII 제어 문자를 찾습니다. 여기서 *X* 또는 *x* 는 제어 문자를 나타내는 문자입니다.	`\cC`	`"\x0003"`의 `"\x0003"` (Ctrl-C)
`\u` *nnnn*	16진수 표현(정확히 네 자리로 구성되는 *nnnn*)을 사용하여 유니코드 문자를 찾습니다.	`\w\u0020\w`	`"a bc d"`의 `"a b"`, `"c d"`
`\`	이 표나 이 항목의 다른 표에 있는 이스케이프된 문자로 인식되지 않는 문자가 뒤에 나올 경우 이 문자를 찾습니다. 예를 들어, `\*`는 `\x2A`와 같고 `\.`는 `\x2E`와 같습니다. 이를 통해 정규식 엔진은 * 또는 ?와 같은 언어 요소와 `\*` 또는 `\?`로 표현되는 문자 리터럴을 구분합니다.	`\d+[\+-x\*]\d+`	`"(2+2) * 3*9"`의 `"2+2"` 및 `"3*9"`

문자 클래스

문자 클래스는 문자 집합 중 하나를 찾습니다. 문자 클래스에는 다음 표에 나와 있는 언어 요소가 포함됩니다. 자세한 내용은 문자 클래스를 참조하세요.

테이블 확장

문자 클래스	설명	패턴	일치

`[` *character_group* `]`	character_group *모든 단일 문자와* 일치합니다. 기본적으로 일치 항목 찾기에서는 대/소문자를 구분합니다.	`[ae]`	`"gray"`의 `"a"`
			
			`"lane"`의 `"a"`, `"e"`

`[^` *character_group* `]`	부정: character_group *없는 모든 단일 문자와* 일치합니다. 기본적으로 *character_group* 의 문자는 대/소문자를 구분합니다.	`[^aei]`	`"reign"`의 `"r"`, `"g"`, `"n"`
`[` *first* `-` *last* `]`	문자 범위: 첫 번째*부터* 마지막*까지 범위의 모든 단일 문자와 일치합니다.*	`[A-Z]`	`"AB123"`의 `"A"`, `"B"`

문자 클래스	설명	패턴	일치

`.`	와일드카드: 을 제외한 `\n`모든 단일 문자와 일치합니다.
	
	리터럴 마침표 문자(또는 `\u002E`)를 일치하려면 이스케이프 문자(`\.`)로 앞에 와야 합니다.
		`a.e`	`"nave"`의 `"ave"`
			
			`"water"`의 `"ate"`

`\p{` *name* `}`	유니코드 일반 범주 또는 이름으로 *지정된 명명된* 블록의 모든 단일 문자와 일치합니다.	`\p{Lu}`
		
		`\p{IsCyrillic}`
			`"City Lights"`의 `"C"`, `"L"`
			
			`"ДЖem"`의 `"Д"`, `"Ж"`

`\P{` *name* `}`	유니코드 일반 범주 또는 이름으로 *지정된 명명된* 블록에 없는 단일 문자와 일치합니다.	`\P{Lu}`
		
		`\P{IsCyrillic}`
			`"City"`의 `"i"`, `"t"`, `"y"`
			
			`"ДЖem"`의 `"e"`, `"m"`

`\w`	단어 문자를 찾습니다.	`\w`	`"ID A1.3"`의 `"I"`, `"D"`, `"A"`, `"1"`, `"3"`
`\W`	단어가 아닌 문자를 찾습니다.	`\W`	`"ID A1.3"`의 `" "`, `"."`
`\s`	공백 문자를 찾습니다.	`\w\s`	`"ID A1.3"`의 `"D "`
`\S`	공백이 아닌 문자를 찾습니다.	`\s\S`	`"int __ctr"`의 `" _"`
`\d`	소수 자릿수와 일치합니다.	`\d`	`"4 = IV"`의 `"4"`
`\D`	10진수 이외의 문자를 찾습니다.	`\D`	`"4 = IV"`의 `" "`, `"="`, `" "`, `"I"`, `"V"`

기준 위치

앵커 또는 너비가 0인 원자적 어설션은 문자열에서 일치 항목의 현재 위치에 따라 일치의 성공 또는 실패 여부를 결정하지만 엔진에서 문자열을 따라 가거나 문자를 소비하도록 하지는 않습니다. 다음 표에 나와 있는 메타문자는 앵커입니다. 자세한 내용은 앵커를 참조하세요.

테이블 확장

Assertion	설명	패턴	일치
`^`	기본적으로 일치 항목은 문자열의 시작 부분에서 시작되어야 합니다. 다중 선에서는 줄의 시작 부분에서 시작되어야 합니다.	`^\d{3}`	`"901-333-"`의 `"901"`
`$`	기본적으로 일치 항목은 문자열의 끝부분 또는 문자열의 끝부분 `\n` 앞에서 발생해야 합니다. 다중 선에서는 줄의 끝부분 또는 줄의 끝 `\n` 앞에서 발생해야 합니다.	`-\d{3}$`	`"-901-333"`의 `"-333"`
`\A`	일치 항목이 문자열의 시작 부분에 있어야 합니다.	`\A\d{3}`	`"901-333-"`의 `"901"`
`\Z`	일치 항목이 문자열의 끝이나 문자열의 끝에 있는 `\n` 앞에 있어야 합니다.	`-\d{3}\Z`	`"-901-333"`의 `"-333"`
`\z`	일치 항목이 문자열의 끝에 있어야 합니다.	`-\d{3}\z`	`"-901-333"`의 `"-333"`
`\G`	일치는 이전 일치가 종료된 지점이나 이전 일치 항목이 없는 경우 일치가 시작된 문자열의 위치에서 발생해야 합니다.	`\G\(\d\)`	`"(1)(3)(5)[7](9)"`의 `"(1)"`, `"(3)"`, `"(5)"`
`\b`	일치 항목이 `\w` (영숫자) 문자와 `\W` (영숫자가 아닌 문자) 문자 사이의 경계에 있어야 합니다.	`\b\w+\s\w+\b`	`"them theme them them"`의 `"them theme"`, `"them them"`
`\B`	일치 항목이 `\b` 경계에 있어야 합니다.	`\Bend\w*\b`	`"end sends endure lender"`의 `"ends"`, `"ender"`

정규식의 그룹화 구문

그룹화 구문은 정규식의 하위 식을 나타내며 대개 입력 문자열의 부분 문자열을 캡처합니다. 그룹화 구문에는 다음 표에 나와 있는 언어 요소가 포함됩니다. 자세한 내용은 그룹화 구문을 참조하세요.

테이블 확장

그룹화 구문	설명	패턴	일치
`(` *subexpression* `)`	일치하는 하위 식을 캡처하고 서수(1부터 시작)를 할당합니다.	`(\w)\1`	`"deep"`의 `"ee"`

그룹화 구문	설명	패턴	일치

`(?<` *name* `>` *subexpression* `)`
또는
`(?'` *name* `'` *subexpression* `)`
	일치하는 하위 식을 명령된 그룹에 캡처합니다.	`(?<double>\w)\k<double>`	`"deep"`의 `"ee"`

`(?<` *name1* `-` *name2* `>` *subexpression* `)`
또는
`(?'` *name1* `-` *name2* `'` *subexpression* `)`
	균형 조정 그룹 정의를 정의합니다. 자세한 내용은 Grouping Constructs의 "균형 조정 그룹 정의" 섹션을 참조하세요.	`(((?'Open'\()[^\(\)]*)+((?'Close-Open'\))[^\(\)]*)+)*(?(Open)(?!))$`	`"3+2^((1-3)*(3-1))"`의 `"((1-3)*(3-1))"`

`(?:` *subexpression* `)`	비캡처 그룹을 정의합니다.	`Write(?:Line)?`	`"Console.WriteLine()"`의 `"WriteLine"`
			
			`"Console.Write(value)"`의 `"Write"`

`(?imnsx-imnsx:` *subexpression* `)`	*subexpression* 내에서 지정된 옵션을 적용하거나 사용하지 않도록 설정합니다. 자세한 내용은 Regular Expression Options을 참조하세요.	`A\d{2}(?i:\w+)\b`	`"A12xl A12XL a12xl"`의 `"A12xl"`, `"A12XL"`
`(?=` *subexpression* `)`	너비가 0인 긍정 우측 어설션입니다.	`\b\w+\b(?=.+and.+)`	`"cats"`, `"dogs"` in `"cats, dogs and some mice."`
`(?!` *subexpression* `)`	너비가 0인 부정 우측 어설션입니다.	`\b\w+\b(?!.+and.+)`	`"and"`, , `"some"` `"mice"` in `"cats, dogs and some mice."`

그룹화 구문	설명	패턴	일치

`(?<=` *subexpression* `)`	너비가 0인 긍정 좌측 어설션입니다.	`\b\w+\b(?<=.+and.+)`
		
		--- 	
		
		`\b\w+\b(?<=.+and.*)`
			`"some"`, `"mice"` in `"cats, dogs and some mice."`
			
			--- 	
			
			`"and"`, , `"some"` `"mice"` in `"cats, dogs and some mice."`

`(?<!` *subexpression* `)`	너비가 0인 부정 좌측 어설션입니다.	`\b\w+\b(?<!.+and.+)`
		
		--- 	
		
		`\b\w+\b(?<!.+and.*)`
			`"cats"`, , `"dogs"` `"and"` in `"cats, dogs and some mice."`
			
			--- 	
			
			`"cats"`, `"dogs"` in `"cats, dogs and some mice."`

`(?>` *subexpression* `)`	원자성 그룹입니다.	'(?>a|ab)c	`"ac"` 안으로 `"ac"` *아무 것도 없습니다* . `"abc"`

한눈에 살펴보기

정규식 엔진이 조회 식에 도달**하면 현재 위치에서 원래 문자열의 시작(lookbehind) 또는 끝(lookahead)에 도달하는 부분 문자열을 취한 다음, 조회 패턴을 사용하여 해당 부분 문자열에서 실행됩니다**Regex.IsMatch**.** 이 하위 식 결과의 성공은 긍정 또는 부정 어설션인지 여부에 따라 결정됩니다.

테이블 확장

해결 방법	속성	함수
`(?=check)`	긍정 Lookahead	문자열의 현재 위치 바로 뒤에 있는 것이 "check"임을 어설션합니다.
`(?<=check)`	Positive Lookbehind	문자열의 현재 위치 바로 앞에 오는 것이 "check"임을 어설션합니다.
`(?!check)`	네거티브 Lookahead	문자열의 현재 위치 바로 뒤에 있는 것이 "check"가 아니라는 것을 어설션합니다.

해결 방법	속성	함수
`(?<!check)`	네거티브 Lookbehind	문자열의 현재 위치 바로 앞에 오는 것이 "check"가 아니라는 것을 어설션합니다.

일치 **하면 일치로 인해 패턴의 나머지가 실패하더라도 원자성 그룹은** 다시 평가되지 않습니다. 이렇게 하면 원자성 그룹 또는 패턴의 나머지 부분에서 수량자가 발생할 때 성능이 크게 향상될 수 있습니다.

수량자

수량자는 이전 요소(문자, 그룹 또는 문자 클래스)의 인스턴스가 입력 문자열에 몇 개 있어야 일치 항목으로 간주되는지를 지정합니다. 수량자에는 다음 표에 나와 있는 언어 요소가 포함됩니다. 자세한 내용은 수량자를 참조하세요.

테이블 확장

수량자	설명	패턴	일치
`*`	이전 요소를 0개 이상 찾습니다.	`a.*c`	`"abcbc"`의 `"abcbc"`
`+`	이전 요소를 1개 이상 찾습니다.	`"be+"`	`"been"`의 `"bee"`, `"bent"`의 `"be"`
`?`	이전 요소를 0개 또는 1개 찾습니다.	`"rai?"`	`"rain"`의 `"rai"`
`{` *n* `}`	이전 요소를 정확히 *n*회 찾습니다.	`",\d{3}"`	`"1,043.6"`의 `",043"`, `"9,876,543,210"`의 `",876"`, `",543"` 및 `",210"`
`{` *n* `,}`	이전 요소를 최소한 *n*회 찾습니다.	`"\d{2,}"`	`"166"`, , `"29"` `"1930"`
`{` *n* `,` *m* `}`	이전 요소를 *n* 회 이상 *m* 회 이하로 찾습니다.	`"\d{3,5}"`	`"166"`, `"17668"` `"193024"`의 `"19302"`
`*?`	이전 요소를 0개 이상 가능한 한 적은 개수로 찾습니다.	`a.*?c`	`"abcbc"`의 `"abc"`
`+?`	이전 요소를 1개 이상 가능한 한 적은 개수로 찾습니다.	`"be+?"`	`"been"`의 `"be"`, `"bent"`의 `"be"`
`??`	이전 요소를 가능한 한 적은 개수로 0개 또는 1개 찾습니다.	`"rai??"`	`"rain"`의 `"ra"`

수량자	설명	패턴	일치
`{` *n* `}?`	이전 요소를 정확히 *n*회 찾습니다.	`",\d{3}?"`	`"1,043.6"`의 `",043"`, `"9,876,543,210"`의 `",876"`, `",543"` 및 `",210"`
`{` *n* `,}?`	이전 요소를 최소한 *n*회 이상 가능한 한 적은 개수로 찾습니다.	`"\d{2,}?"`	`"166"`, , `"29"` `"1930"`

`{` *n* `,` *m* `}?`	이전 요소를 *n* 회에서 *m* 회 사이에서 찾으며, 가능한 한 적은 개수로 찾습니다.	`"\d{3,5}?"`	`"166"`, `"17668"`
			
			`"193024"`의 `"193"`, `"024"`

역참조 구문

역참조를 사용하면 이전에 찾은 하위 식을 이후에 동일한 정규식에서 식별할 수 있습니다. 다음 표에서는 .NET의 정규식에서 지원하는 역참조 구문을 보여줍니다. 자세한 내용은 Backreference Constructs을 참조하세요.

테이블 확장

역참조 구문	설명	패턴	일치
`\` *number*	역참조입니다. 번호가 매겨진 하위 식의 값을 찾습니다.	`(\w)\1`	`"seek"`의 `"ee"`
`\k<` *name* `>`	명명된 역참조입니다. 명명된 식의 값을 찾습니다.	`(?<char>\w)\k<char>`	`"seek"`의 `"ee"`

Alternation Constructs

교체 구문은 일치를 허용하도록 정규식을 수정합니다. 이러한 구문에는 다음 표에 나와 있는 언어 요소가 포함됩니다. 자세한 내용은 교체 구문을 참조하세요.

테이블 확장

교체 구문	설명	패턴	일치
`|`	세로 막대(`|`)로 구분된 한 가지 요소와 일치합니다.	`th(e|is|at)`	`"this is the day."`의 `"the"`, `"this"`
`(?(` *expression* `)` *yes* `|` *no*	*expression* 으로 지정한 정규식 패턴이 일치하면 *yes*와 일치합니다. 그렇지 않으면 선	`(?(A)A\d{2}\b|\b\d{3}\b)`	`"A10 C103 910"`의 `"A10"`, `"910"`

교체 구문	설명	패턴	일치
`)` 또는 `(?(` *expression* `)` *yes* `)`	택 사항 *no* 부분과 일치합니다. *expression* 은 너비가 0인 어설션으로 해석됩니다. 명명되거나 번호가 매겨진 캡처링 그룹의 모호성을 방지하기 위해 다음과 같이 명시적 어설션을 선택적으로 사용할 수 있습니다. `(?( (?=` *expression* `) )` *yes* `|` *no* `)`		
`(?(` *name* `)` *yes* `|` *no* `)` 또는 `(?(` *name* `)` *yes* `)`	명명되거나 번호가 매겨진 캡처링 그룹인 *name* 에 일치하는 항목이 있으면 *yes*와 일치합니다. 그렇지 않으면 선택 사항 *no*와 일치합니다.	`(?<quoted>")?(?(quoted).+?"|\S+\s)`	`"Dogs.jpg \"Yiska playing.jpg\""`의 `"Dogs.jpg "`, `"\"Yiska playing.jpg\""`

대체

대체는 바꾸기 패턴에서 지원하는 정규식 언어 요소입니다. 자세한 내용은 대체를 참조하세요. 다음 표에 나와 있는 메타문자는 너비가 0인 원자성 어설션입니다.

테이블 확장

문자	설명	패턴	대체 패턴	입력 문자열	결과 문자열
`$` *number*	그룹 *number*와 일치하는 부분 문자열을 대체합니다.	`\b(\w+)(\s)(\w+)\b`	`$3$2$1`	`"one two"`	`"two one"`
`${` *name* `}`	명명된 그룹 *name*과 일치하는 부분 문자열을 대체합니다.	`\b(?<word1>\w+)(\s)(?<word2>\w+)\b`	`${word2}${word1}`	`"one two"`	`"two one"`
`$$`	"$" 리터럴을 대체합니다.	`\b(\d+)\s?USD`	`$$$1`	`"103 USD"`	`"$103"`
`$&`	일치하는 전체 문자열의 복사본을 대체합니다.	`\$?\d*\.?\d+`	`**$&**`	`"$1.30"`	`"**$1.30**"`
`$``	일치하는 문자열 앞에 있는 입력 문자열	`B+`	`$``	`"AABBCC"`	`"AAAACC"`

문자	설명	패턴	대체 패턴	입력 문자열	결과 문자열
	의 모든 텍스트를 대체합니다.				
`$'`	일치하는 문자열 뒤에 있는 입력 문자열의 모든 텍스트를 대체합니다.	`B+`	`$'`	`"AABBCC"`	`"AACCCC"`
`$+`	캡처된 마지막 그룹을 대체합니다.	`B+(C+)`	`$+`	`"AABBCCDD"`	`"AACCDD"`
`$_`	전체 입력 문자열을 대체합니다.	`B+`	`$_`	`"AABBCC"`	`"AAAABBCCCC"`

정규식 옵션

정규식 엔진이 정규식 패턴을 해석하는 방법을 제어하는 옵션을 지정할 수 있습니다. 옵션의 대부분은 인라인(정규식 패턴)에서 또는 1개 이상의 RegexOptions 상수로 지정될 수 있습니다. 이 빠른 참조는 인라인 옵션만 나열합니다. 인라인 및 RegexOptions 옵션에 대한 자세한 내용은 정규식 옵션를 참조하세요.

인라인 옵션을 두 가지 방법으로 지정할 수 있습니다.

- 기타 구문 `(?imnsx-imnsx)`을 사용하여 옵션이나 옵션 집합 앞에 빼기 기호(-)를 추가해 해당 옵션을 해제할 수 있습니다. 예를 들어 `(?i-mn)`은 대/소문자를 구분하지 않는 일치 조건(`i`)을 설정하고 여러 줄 모드(`m`)를 해제하고 명명되지 않은 그룹 캡처(`n`)를 해제합니다. 이 옵션은 옵션이 정의되는 지점부터 정규식 패턴에 적용되고, 패턴 끝 또는 다른 구문이 옵션을 되돌리는 지점까지 유효합니다.
- 그룹화 구문 `(?imnsx-imnsx:`*subexpression*`)`을 사용하여 지정된 그룹에 대해서만 옵션을 정의할 수 있습니다.

.NET 정규식 엔진은 다음 인라인 옵션을 지원합니다.

테이블 확장

옵션	설명	패턴	일치
`i`	대/소문자를 구분하지 않는 일치를 사용합니다.	`\b(?i)a(?-i)a\w+\b`	`"aardvark AAAuto aaaAuto Adam breakfast"`의 `"aardvark"`, `"aaaAuto"`

옵션	설명	패턴	일치
`m`	여러 줄 모드를 사용합니다. `^` 및 `$`는 각 줄의 시작 및 끝과 일치합니다(문자열의 시작 및 끝이 아님).	예제를 보려면 정규식 옵션에서 "여러 줄 모드" 섹션을 참조하세요.	
`n`	명명되지 않은 그룹을 캡처하지 않습니다.	예를 들어 정규식 옵션에서 "명시적 캡처만 해당" 섹션을 참조하세요.	
`s`	한 줄 모드를 사용합니다.	예제를 보려면 정규식 옵션에서 "한 줄 모드" 섹션을 참조하세요.	
`x`	정규식 패턴에서 이스케이프되지 않은 공백은 무시합니다.	`\b(?x) \d+ \s \w+`	`"1 aardvark 2 cats IV centurions"`의 `"1 aardvark"`, `"2 cats"`

기타 구문

기타 구문은 정규식 패턴을 수정하거나 정규식 패턴에 대한 정보를 제공합니다. 다음 표에서는 .NET에서 지원하는 기타 구문을 보여줍니다. 자세한 내용은 기타 구문을 참조하세요.

테이블 확장

구문	정의	예시
`(?imnsx-imnsx)`	패턴 중간에 대/소문자 구분하지 않음과 같은 옵션을 설정하거나 해제합니다. 자세한 내용은 정규식 옵션을 참조하세요.	`\bA(?i)b\w+\b`는 `"ABA Able Act"`의 `"ABA"`, `"Able"`과 일치합니다.
`(?#` *comment* `)`	인라인 주석입니다. 주석이 첫 번째 닫는 괄호 문자에서 끝납니다.	`\bA(?#Matches words starting with A)\w+\b`
`#` [줄의 끝]	X-모드 주석입니다. 주석이 이스케이프되지 않은 `#` 에서 시작하여 줄 끝까지 이어집니다.	`(?x)\bA\w+\b#Matches words starting with A`

참고 항목

- System.Text.RegularExpressions
- System.Text.RegularExpressions.Regex
- 정규식

- 정규식 클래스
- 정규식 - 빠른 참조(Word 형식으로 다운로드)
- 정규식 - 빠른 참조(PDF 형식으로 다운로드)

정규식의 문자 이스케이프

아티클 • 2024. 03. 17.

정규식의 백슬래시(\)는 다음 중 하나를 나타냅니다.

- 다음 섹션의 표에 나와 있는 대로 뒤에 나오는 문자는 특수 문자입니다. 예를 들어 `\b`는 단어 경계에서 정규식 일치가 시작되어야 함을 나타내는 앵커이고, `\t`는 탭을 나타내고, `\x020`은 공백을 나타냅니다.
- 이스케이프되지 않은 언어 구문으로 해석되는 문자는 문자 그대로 해석되어야 합니다. 예를 들어 중괄호(`{`)는 수량자 정의를 시작하지만 중괄호 뒤에 백슬래시가 있으면(`\{`) 정규식 엔진이 중괄호와 일치해야 함을 나타냅니다. 마찬가지로 단일 백슬래시는 이스케이프된 언어 구문의 시작을 표시하지만, 이중 백슬래시(`\\`)는 정규식 엔진이 백슬래시와 일치해야 함을 나타냅니다.

> **참고**
>
> 문자 이스케이프는 정규식 패턴에서는 인식되지만 대체 패턴에서 인식되지 않습니다.

.NET의 문자 이스케이프

다음 표에서는 .NET의 정규식에서 지원하는 문자 이스케이프를 보여 줍니다.

테이블 확장

문자 또는 시퀀스	설명
다음을 제외한 모든 문자입니다. . $ ^ { [(|) * + ? \	**문자 또는 시퀀스** 열에 나열된 문자 이외의 문자는 정규식에서 특별한 의미를 가지지 않습니다. 문자 그대로 해석됩니다. **문자 또는 시퀀스** 열에 포함된 문자는 특수 정규식 언어 요소입니다. 정규식에서 이들 문자를 찾으려면 문자를 이스케이프하거나 긍정 문자 그룹에 포함해야 합니다. 예를 들어 정규식 `\$\d+` 또는 `[$]\d+`는 "$1200"을 찾습니다.
`\a`	벨 문자인 `\u0007`을 찾습니다.
`\b`	`[`*character_group*`]` 문자 클래스에서 백스페이스 문자인 `\u0008`을 찾습니다. 문자 클래스를 참조하세요. 문자 클래스 이외에 `\b`는 단어 경계와 일치하는 앵커입니다. 앵커를 참조하세요.

문자 또는 시퀀스	설명
`\t`	탭 문자인 `\u0009`를 찾습니다.
`\r`	캐리지 리턴 문자인 `\u000D`를 찾습니다. `\r`은 줄 바꿈 문자인 `\n`과 다릅니다.
`\v`	세로 탭 문자인 `\u000B`를 찾습니다.
`\f`	용지 공급 문자인 `\u000C`를 찾습니다.
`\n`	줄 바꿈 문자인 `\u000A`를 찾습니다.
`\e`	이스케이프 문자인 `\u001B`를 찾습니다.
`\`*nnn*	ASCII 문자를 찾습니다. 여기서 *nnn*은 8진수 문자 코드를 나타내는 두 자리 또는 세 자리 숫자로 구성됩니다. 예를 들어 `\040`은 공백 문자를 나타냅니다. 이 생성자가 한 개의 숫자만 포함하거나(예: `\2`) 캡처링 그룹의 수와 일치하는 경우에는 역참조로 해석됩니다. 역참조 구문을 참조하세요.
`\x`*nn*	ASCII 문자를 찾습니다. 여기서 *nn*은 두 자리 16진수 문자 코드입니다.
`\c`*X*	ASCII 제어 문자를 찾습니다. 여기서 X는 제어 문자를 나타내는 문자입니다. 예를 들어, `\cC`는 CTRL-C입니다.
`\u`*nnnn*	단위 값이 *nnnn* 16진수인 UTF-16 코드 단위를 찾습니다. **참고:** 유니코드를 지정하는 데 사용되는 Perl 5 문자 이스케이프는 .NET에서 지원되지 않습니다. Perl 5 문자 이스케이프는 `\x{####…}` 형식입니다. 여기서 `####…`는 일련의 16진수입니다. 대신에 `\u`*nnnn*을 사용합니다.
`\`	이스케이프된 문자로 인식되지 않는 문자가 뒤에 나올 경우 이 문자를 찾습니다. 예를 들어 `\*`는 별표(*)와 일치하고 `\x2A`와 같습니다.

예제

다음 예제에서는 정규식에서 문자 이스케이프를 사용하는 방법을 보여 줍니다. 세계 최대 도시의 이름과 2009년 인구가 포함된 문자열을 구문 분석합니다. 각 도시 이름과 해당 인구는 탭(`\t`) 또는 세로 막대(| 또는 `\u007c`)로 구분됩니다. 개별 도시 및 해당 인구는 캐리지 리턴 및 줄 바꿈으로 서로 구분됩니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
```

```
      string delimited = @"\G(.+)[\t\u007c](.+)\r?\n";
      string input = "Mumbai, India|13,922,125\t\n" +
                           "Shanghai, China\t13,831,900\n" +
                           "Karachi, Pakistan|12,991,000\n" +
                           "Delhi, India\t12,259,230\n" +
                           "Istanbul, Türkiye|11,372,613\n";
      Console.WriteLine("Population of the World's Largest Cities, 2009");
      Console.WriteLine();
      Console.WriteLine("{0,-20} {1,10}", "City", "Population");
      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, delimited))
         Console.WriteLine("{0,-20} {1,10}", match.Groups[1].Value,
                                             match.Groups[2].Value);
   }
}
// The example displays the following output:
//       Population of the World's Largest Cities, 2009
//
//       City                 Population
//
//       Mumbai, India        13,922,125
//       Shanghai, China      13,831,900
//       Karachi, Pakistan    12,991,000
//       Delhi, India         12,259,230
//       Istanbul, Türkiye     11,372,613
```

정규식 `\G(.+)[\t\u007c](.+)\r?\n`는 다음 테이블과 같이 해석됩니다.

테이블 확장

패턴	설명
`\G`	마지막 일치가 종료되면 일치를 시작합니다.
`(.+)`	임의 문자를 한 번 이상 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`[\t\u007c]`	탭 문자(`\t`) 또는 세로 막대(|)를 찾습니다.
`(.+)`	임의 문자를 한 번 이상 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`\r?\n`	캐리지 리턴, 줄 바꿈이 차례로 나타나는 발생 0개 또는 1개를 찾습니다.

참고 항목

- 정규식 언어 - 빠른 참조

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

정규식의 문자 클래스

문자 클래스는 문자 집합을 정의하며, 이 중 하나가 입력 문자열에서 발생하면 일치하는 것으로 판정할 수 있습니다. .NET의 정규식 언어는 다음과 같은 문자 클래스를 지원합니다.

- 긍정 문자 그룹. 입력 문자열의 문자는 지정된 문자 집합 중 하나와 일치해야 합니다. 자세한 내용은 긍정 문자 그룹을 참조하세요.
- 부정 문자 그룹. 입력 문자열의 문자는 지정된 문자 집합 중 하나와 일치하면 안 됩니다. 자세한 내용은 부정 문자 그룹을 참조하세요.
- 모든 character입니다. 정규식의 `.`(점 또는 마침표) 문자는 `\n`을 제외한 모든 문자와 일치하는 와일드카드 문자입니다. 자세한 내용은 임의의 문자를 참조하세요.
- 일반 유니코드 범주 또는 명명된 블록입니다. 입력 문자열의 문자는 일치하는 것으로 판정하려면 특정 유니코드 범주의 멤버이거나 유니코드 문자의 연속된 범위 내에 있어야 합니다. 자세한 내용은 유니코드 범주 또는 유니코드 블록을 참조하세요.
- 부정적 일반 유니코드 범주 또는 이름이 지정된 블록입니다. 입력 문자열의 문자는 일치하는 것으로 판정하려면 특정 유니코드 범주의 멤버가 아니거나 유니코드 문자의 연속된 범위 내에 있으면 안 됩니다. 자세한 내용은 부정 유니코드 범주 또는 유니코드 블록을 참조하세요.
- 단어 문자입니다. 입력 문자열의 문자는 단어에 있는 문자에 적합한 유니코드 범주에 속할 수 있습니다. 자세한 내용은 단어 문자를 참조하세요.
- 비단어 문자입니다. 입력 문자열의 문자는 단어 문자가 아닌 유니코드 범주에 속할 수 있습니다. 자세한 내용은 단어가 아닌 문자를 참조하세요.
- 공백 문자. 입력 문자열의 문자는 유니코드 구분 기호 문자와 여러 컨트롤 문자 중 하나일 수 있습니다. 자세한 내용은 공백 문자를 참조하세요.
- 공백이 아닌 문자. 입력 문자열의 문자는 공백 문자가 아닌 문자일 수 있습니다. 자세한 내용은 공백이 아닌 문자를 참조하세요.
- 10진수입니다. 입력 문자열의 문자는 유니코드 10진수로 분류된 문자일 수 있습니다. 자세한 내용은 10진수 숫자 문자를 참조하세요.
- 10진수가 아닙니다. 입력 문자열의 문자는 유니코드 10진수 이외의 문자는 모두 사용할 수 있습니다. 자세한 내용은 10진수 숫자 문자를 참조하세요.

.NET에서는 한 문자 클래스에서 다른 문자 클래스를 제외한 결과로 문자 집합을 정의하는 데 사용할 수 있는 문자 클래스 빼기 식을 지원합니다. 자세한 내용은 문자 클래스 빼기를 참조하세요.

> ⓘ **참고 항목**

일치하는 단어 문자를 검색하는 \w 또는 일치하는 유니코드 범주를 검색하는 \p{}와 같이 범주별로 일치하는 문자를 검색하는 문자 클래스는 CharUnicodeInfo 클래스를 활용하여 문자 범주에 대한 정보를 제공합니다. .NET Framework 4.6.2 이상 버전에서 문자 범주는 **유니코드 표준 버전 8.0.0**을 기반으로 합니다.

긍정 문자 그룹: [ ]

양수 문자 그룹은 문자 목록을 지정하며, 그 중 하나가 입력 문자열에 표시되어 일치 항목이 발생할 수 있습니다. 이 문자 목록은 개별적으로, 범위 또는 둘 다로 지정될 수 있습니다.

개별 문자 목록을 지정하는 구문은 다음과 같습니다.

```
[*character_group*]
```

여기서 *character_group* 일치가 성공하기 위해 입력 문자열에 나타날 수 있는 개별 문자 목록입니다. *character_group*은 하나 이상의 리터럴 문자, 이스케이프 문자 또는 문자 클래스로 이루어진 조합으로 구성될 수 있습니다.

문자의 범위를 지정하는 구문은 다음과 같습니다.

```
[firstCharacter-lastCharacter]
```

여기서 *firstCharacter* 는 범위를 시작하는 문자이고 *lastCharacter* 는 범위를 종료하는 문자입니다. 문자 범위는 일련의 문자로서, 문자 범위를 정의하려면 연속된 문자 중 첫 번째 문자와 마지막 문자를 하이픈(-)으로 연결하여 지정합니다. 두 문자의 유니코드 코드 포인트가 연속되면 이 두 문자는 연속된 문자입니다. *firstCharacter*는 낮은 코드 포인트를 가진 문자여야 하며 *lastCharacter*는 높은 코드 포인트를 가진 문자여야 합니다.

> **참고 항목**
>
> 양수 문자 그룹에는 문자 집합과 문자 범위가 모두 포함될 수 있으므로 하이픈 문자(`-`)는 그룹의 첫 번째 또는 마지막 문자가 아닌 한 항상 범위 구분 기호로 해석됩니다.

하이픈을 문자 그룹의 변두리 멤버가 아닌 내부 멤버로 포함하려면 하이픈을 이스케이프 처리합니다. 예를 들어 `a` 문자와 `-`에서 `/`까지의 문자로 구성된 문자 그룹을 만들려면 올바른 구문은 `[a\--/]`입니다.

긍정 문자 클래스가 포함된 몇 가지 일반적인 정규식 패턴은 다음과 같습니다.

테이블 확장

패턴	설명
`[aeiou]`	모든 모음을 찾습니다.
`[\p{P}\d]`	모든 문장 부호와 10진수 문자와 일치시키다.
`[\s\p{P}]`	모든 공백 및 문장 부호를 일치시킵니다.

다음 예제에서는 "a"와 "e" 문자를 포함하는 긍정 문자 그룹을 정의하여 입력 문자열에 "grey" 또는 "gray"라는 단어가 있고 그 후에 다른 단어가 있어야 일치가 발생하도록 합니다.

C#

```
static void PositiveCharacterGroup()
{
    string pattern = @"gr[ae]y\s\S+?[\s\p{P}]";
    string input = "The gray wolf jumped over the grey wall.";
    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match match in matches)
        Console.WriteLine($"'{match.Value}'");
}
// The example displays the following output:
//       'gray wolf '
//       'grey wall.'
```

`gr[ae]y\s\S+?[\s|\p{P}]` 정규식은 다음과 같이 정의됩니다.

테이블 확장

패턴	설명
`gr`	리터럴 문자 "gr"과 일치시킵니다.
`[ae]`	"a" 또는 "e"를 찾습니다.
`y\s`	리터럴 문자 "y" 다음에 오는 공백 문자를 매치합니다.
`\S+?`	가능한 한 적은 수의 공백이 아닌 문자를 찾습니다.
`[\s\p{P}]`	공백 문자 또는 문장 부호를 찾습니다.

다음 예제에서는 모든 대문자로 시작하는 단어를 찾습니다. A-Z의 대문자로 범위를 나타내기 위해 `[A-Z]` 하위 식을 사용합니다.

C#

```
static void CharacterRange()
{
    string pattern = @"\b[A-Z]\w*\b";
```

```
    string input = "A city Albany Zulu maritime Marseilles";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine(match.Value);
}
// The example displays the following output:
//       A
//       Albany
//       Zulu
//       Marseilles
```

`\b[A-Z]\w*\b` 정규식은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하세요.
`[A-Z]`	A-Z의 대문자를 모두 찾습니다.
`\w*`	0개 이상의 단어 문자를 찾습니다.
`\b`	단어 경계를 찾습니다.

부정 문자 그룹: [^]

부정 문자 그룹은 일치 항목으로 간주되려면 입력 문자열에 나타나서는 안 되는 문자 목록을 지정합니다. 문자 목록을 개별적으로, 범위 또는 둘 다로 지정할 수 있습니다.

개별 문자 목록을 지정하는 구문은 다음과 같습니다.

`[*^character_group*]`

여기서 *character_group* 일치가 성공하기 위해 입력 문자열에 표시할 수 없는 개별 문자 목록입니다. *character_group*은 하나 이상의 리터럴 문자, 이스케이프 문자 또는 문자 클래스로 이루어진 조합으로 구성될 수 있습니다.

문자의 범위를 지정하는 구문은 다음과 같습니다.

`[^*firstCharacter*-*lastCharacter*]`

여기서 *firstCharacter* 는 범위를 시작하는 문자이고 *lastCharacter* 는 범위를 종료하는 문자입니다. 문자 범위는 일련의 문자로서, 문자 범위를 정의하려면 연속된 문자 중 첫 번째 문자와 마지막 문자를 하이픈(-)으로 연결하여 지정합니다. 두 문자의 유니코드 코드 포인트가 연속되면 이 두 문자는 연속된 문자입니다. *firstCharacter*는 낮은 코드 포인트를 가진 문자여야 하며 *lastCharacter*는 높은 코드 포인트를 가진 문자여야 합니다.

> **참고 항목**
>
> 음수 문자 그룹에는 문자 집합과 문자 범위가 모두 포함될 수 있으므로 하이픈 문자(`-`)는 그룹의 첫 번째 또는 마지막 문자가 아닌 한 항상 범위 구분 기호로 해석됩니다.

두 개 이상의 문자 범위를 연결할 수도 있습니다. 예를 들어, "0"부터 "9"까지의 10진수 범위, "a"부터 "f"까지의 소문자 범위 및 "A"부터 "F"까지의 대문자 범위를 지정하려면 `[0-9a-fA-F]`를 사용합니다.

부정 문자 그룹에서 맨 앞의 캐럿 문자(`^`)는 필수 문자로서, 해당 문자 그룹이 긍정 문자 그룹이 아니라 부정 문자 그룹임을 나타냅니다.

> **Important**
>
> 더 큰 정규식 패턴의 음수 문자 그룹은 너비가 0인 어설션이 아닙니다. 즉, 부정 문자 그룹을 평가한 후 정규식 엔진은 입력 문자열에서 한 문자를 이동합니다.

부정 문자 그룹이 포함된 몇 가지 일반적인 정규식 패턴은 다음과 같습니다.

테이블 확장

패턴	설명
`[^aeiou]`	모음을 제외한 모든 문자를 찾습니다.
`[^\p{P}\d]`	문장 부호와 숫자를 제외하고 모든 문자를 일치시킵니다.

다음 예제에서는 문자 "th"로 시작하고 뒤에 "o"가 오지 않는 단어를 찾습니다.

C#

```
static void NegativeCharacterGroup()
{
    string pattern = @"\bth[^o]\w+\b";
    string input = "thought thing though them through thus thorough this";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine(match.Value);
}
// The example displays the following output:
//       thing
//       them
//       through
//       thus
//       this
```

`\bth[^o]\w+\b` 정규식은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하십시오.
`th`	리터럴 문자 "th"를 찾습니다.
`[^o]`	"o"가 아닌 문자와 일치합니다.
`\w+`	하나 이상의 단어 문자를 매치합니다.
`\b`	단어 경계에서 멈춥니다.

임의의 문자: .

마침표 문자(.)는 `\n`(줄 바꿈 문자)를 제외한 모든 문자와 일치하며 다음 두 가지 조건이 있습니다.

- 정규식 패턴이 RegexOptions.Singleline 옵션에 의해 수정되거나 `.` 문자 클래스를 포함하는 패턴의 일부가 `s` 옵션에 의해 수정되는 경우 `.`가 문자를 일치시킵니다. 자세한 내용은 Regular Expression Options을 참조하세요.

  다음 예제에서는 기본 및 `.` 옵션으로 RegexOptions.Singleline 문자 클래스의 다른 동작을 보여줍니다. 정규식 `^.+`는 문자열의 시작에서 모든 문자를 일치시킵니다. 기본적으로 일치 항목은 첫 번째 줄의 끝에 종료됩니다. 정규식 패턴은 캐리지 리턴 문자 `\r`와는 일치하지만, `\n`와는 일치하지 않습니다. RegexOptions.Singleline 옵션은 전체 입력 문자열을 한 줄로 해석하기 때문에 `\n`을 포함하여 입력 문자열의 모든 문자와 일치합니다.

  C#

```
static void AnyCharacterMultiline()
{
    string pattern = "^.+";
    string input = "This is one line and" + Environment.NewLine + "this is the
second.";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine(Regex.Escape(match.Value));

    Console.WriteLine();
    foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Singleline))
        Console.WriteLine(Regex.Escape(match.Value));
}
// The example displays the following output:
```

```
//       This\ is\ one\ line\ and\r
//
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

> ⓘ **참고 항목**
>
> `\n`을(를) 제외한 모든 문자와 일치하기 때문에 `.` 문자 클래스는 `\r`(캐리지 리턴 문자)과(와)도 일치합니다.

- 긍정 또는 부정 문자 그룹의 마침표는 문자 클래스가 아니라 리터럴 마침표 문자로 처리됩니다. 자세한 내용은 이 문서의 앞부분에 있는 긍정 문자 그룹 및 부정 문자 그룹을 참조하세요. 다음 예제는 문자 클래스와 긍정 문자 그룹으로 마침표 문자(`.`)를 포함하는 정규식을 정의하는 그림을 제공합니다. `\b.*[.?!;:](\s|\z)` 정규식은 단어 경계에서 시작하여 마침표를 포함한 다섯 가지 문장 부호 중 하나를 만날 때까지 문자와 일치하고, 그 이후에 공백 문자 또는 문자열의 끝과 일치합니다.

C#

```
static void AnyCharacterSingleline()
{
    string pattern = @"\b.*[.?!;:](\s|\z)";
    string input = "this. what: is? go, thing.";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine(match.Value);
}
// The example displays the following output:
//       this. what: is? go, thing.
```

> ⓘ **참고 항목**
>
> 모든 문자와 일치하기 때문에 `.` 언어 요소는 정규식 패턴이 모든 문자를 여러 번 일치하도록 시도할 경우 종종 게으른 수량자와 함께 사용됩니다. 자세한 내용은 **수량자**를 참조하세요.

유니코드 범주 또는 유니코드 블록: \p{}

유니코드 표준에서는 각 문자를 일반 범주에 할당합니다. 예를 들어 특정 문자는 대문자(범주로 `Lu` 표시), 소수 자릿수(범주), 수학 기호(`Nd` `Sm` 범주) 또는 단락 구분 기호(`Zl` 범주)일 수 있습니다. 유니코드 표준의 특정 문자 집합이 특정 범위 또는 연속된 코드 포인트의 블록도 차지합니다. 예를 들어, 기본 라틴 문자 집합은 \u0000부터 \u007F까지에서 발견되고, 아랍어 문자 집합이 \u0600부터 \u06FF까지에서 발견됩니다.

정규 표현식 구조

`\p{` *이름* `}`

유니코드 일반 범주 또는 명명된 블록에 속하는 모든 문자와 일치합니다. 여기서 *이름*은 범주 약어 또는 명명된 블록 이름입니다. 범주 약어 목록은 이 문서의 뒷부분에 있는 지원되는 유니코드 일반 범주 섹션을 참조하세요. 명명된 블록 목록은 이 문서의 뒷부분에 있는 지원되는 명명된 블록 섹션을 참조하세요.

> **팁**
>
> 문자열을 먼저 정규화하기 위해 String.Normalize 메서드를 호출하면 일치률을 개선할 수 있습니다.

다음 예제에서는 `\p{`*name*`}` 구조를 사용하여 유니코드 일반 범주(이 경우 `Pd` 또는 문장 부호, 대시 범주)와 명명된 블록들인 `IsGreek` 및 `IsBasicLatin`을 모두 매칭합니다.

C#

```
static void UnicodeCategory()
{
    string pattern = @"\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+";
    string input = "Ελληνική Γλώσσα - Greek Language";

    Console.WriteLine(Regex.IsMatch(input, pattern));        // Displays True.
}
```

`\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+` 정규식은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작합니다.
`\p{IsGreek}+`	둘 이상의 그리스어 문자를 찾습니다.
`(\s)?`	0개 또는 1개의 공백 문자를 찾습니다.
`(\p{IsGreek}+(\s)?)+`	하나 이상의 그리스 문자 다음에 0개 또는 1개의 공백 문자가 한 번 이상 나타나는 패턴을 찾습니다.
`\p{Pd}`	문장 부호, 대시 문자를 일치시킵니다.
`\s`	공백 문자를 찾습니다.

패턴	설명
`\p{IsBasicLatin}+`	하나 이상의 기본 라틴 문자를 찾습니다.
`(\s)?`	0번 또는 1번 나오는 공백 문자를 찾습니다.
`(\p{IsBasicLatin}+(\s)?)+`	하나 이상의 기본 라틴 문자 다음에 0개 또는 1개의 공백 문자가 한 번 이상 나타나는 패턴을 찾습니다.

부정 유니코드 범주 또는 유니코드 블록: \P{}

유니코드 표준에서는 각 문자를 일반 범주에 할당합니다. 예를 들어, 특정 문자는 대문자(`Lu` 범주로 표현), 10진수(`Nd` 범주), 수학 기호(`Sm` 범주) 또는 단락 구분 기호(`Zl` 범주)가 될 수 있습니다. 유니코드 표준의 특정 문자 집합이 특정 범위 또는 연속된 코드 포인트의 블록도 차지합니다. 예를 들어, 기본 라틴 문자 집합은 \u0000부터 \u007F까지에서 발견되고, 아랍어 문자 집합이 \u0600부터 \u06FF까지에서 발견됩니다.

정규식 구문

`\P{` *이름* `}`

유니코드 일반 범주 또는 명명된 블록에 속하지 않는 모든 문자와 일치합니다. 여기서 *이름*은 범주 약어 또는 명명된 블록 이름입니다. 범주 약어 목록은 이 문서의 뒷부분에 있는 지원되는 유니코드 일반 범주 섹션을 참조하세요. 명명된 블록 목록은 이 문서의 뒷부분에 있는 지원되는 명명된 블록 섹션을 참조하세요.

> **팁**
>
> 문자열을 먼저 정규화하기 위해 String.Normalize 메서드를 호출하면 일치률을 개선할 수 있습니다.

다음 예제에서는 `\P{`*name*`}` 구문을 사용하여 숫자 문자열에서 모든 통화 기호(이 경우, `Sc` 또는 기호, 통화 범주)를 제거합니다.

C#

```
static void NegativeUnicodeCategory()
{
    string pattern = @"(\P{Sc})+";

    string[] values = { "$164,091.78", "£1,073,142.68", "73¢", "€120" };
    foreach (string value in values)
        Console.WriteLine(Regex.Match(value, pattern).Value);
}
// The example displays the following output:
```

```
//       164,091.78
//       1,073,142.68
//       73
//       120
```

정규식 패턴 `(\P{Sc})+` 은 통화 기호가 아닌 하나 이상의 문자와 일치하며 결과 문자열에서 통화 기호를 효과적으로 제거합니다.

단어 문자: \w

`\w`는 단어 문자와 일치합니다. 단어 문자는 다음 표에 나열된 유니코드 범주의 멤버입니다.

테이블 확장

범주	설명
Ll	글자, 소문자
Lu	문자, 대문자
Lt	대문자, 제목 스타일
Lo	문자, 기타
Lm	문자, 수정자
Mn	표시, 공백 없음
Nd	숫자, 10진수
Pc	문장 부호, 연결자. 이 범주는 가장 일반적으로 사용되는 LOWLINE 문자 (_), U + 005F인 10개의 문자를 포함합니다.

ECMAScript와 호환되는 동작을 지정한 경우 `\w`는 `[a-zA-Z_0-9]`와 같습니다. ECMAScript 정규식에 대한 자세한 내용은 정규식 옵션에서 "ECMAScript 일치 동작" 섹션을 참조하세요.

> **참고 항목**
>
> 모든 단어 문자와 일치하기 때문에 `\w` 언어 요소는 정규식 패턴이 뒤에 특정 단어 문자가 있는 임의의 단어 문자를 여러 번 일치시키려는 경우에 lazy 수량자와 함께 많이 사용됩니다. 자세한 내용은 **수량자**를 참조하세요.

다음 예제에서는 `\w` 언어 요소를 사용하여 단어의 중복 문자를 찾습니다. 예제는 다음과 같이 해석될 수 있는 정규식 패턴 `(\w)\1`을 정의합니다.

테이블 확장

요소	설명
(\w)	단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
\1	첫 번째 캡처의 값을 찾습니다.

C#

```
static void WordCharacter()
{
    string pattern = @"(\w)\1";
    string[] words = { "trellis", "seer", "latter", "summer",
                       "hoarse", "lesser", "aardvark", "stunned" };
    foreach (string word in words)
    {
        Match match = Regex.Match(word, pattern);
        if (match.Success)
            Console.WriteLine($"'{match.Value}' found in '{word}' at position
{match.Index}.");
        else
            Console.WriteLine($"No double characters in '{word}'.");
    }
}
// The example displays the following output:
//       'll' found in 'trellis' at position 3.
//       'ee' found in 'seer' at position 1.
//       'tt' found in 'latter' at position 2.
//       'mm' found in 'summer' at position 2.
//       No double characters in 'hoarse'.
//       'ss' found in 'lesser' at position 2.
//       'aa' found in 'aardvark' at position 0.
//       'nn' found in 'stunned' at position 3.
```

단어가 아닌 문자: \W

`\W`는 단어가 아닌 문자를 찾습니다. \W 언어 요소는 다음 문자 클래스에 해당합니다.

`[^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}]`

즉, 다음 표에 나열된 유니코드 범주의 문자를 제외한 모든 문자를 찾습니다.

테이블 확장

범주	설명
Ll	문자, 소문자
Lu	글자, 대문자
Lt	문자, 제목 스타일
Lo	문자, 기타
Lm	문자, 수정자
Mn	표시, 공백 없음
Nd	숫자, 10진수
Pc	문장 부호, 연결자. 이 범주는 가장 일반적으로 사용되는 LOWLINE 문자 (_), U + 005F인 10개의 문자를 포함합니다.

ECMAScript와 호환되는 동작을 지정한 경우 `\W`는 `[^a-zA-Z_0-9]`와 같습니다. ECMAScript 정규식에 대한 자세한 내용은 정규식 옵션에서 "ECMAScript 일치 동작" 섹션을 참조하세요.

> **참고 항목**
>
> 모든 비단어 문자와 일치하기 때문에 `\W` 언어 요소는 정규식 패턴이 특정 비단어 문자가 따라오는 임의의 비단어 문자를 여러 번 찾으려 하는 경우 lazy 수량자와 함께 사용되는 경우가 많습니다. 자세한 내용은 **수량자**를 참조하세요.

다음 예제에서는 `\W` 문자 클래스를 보여 줍니다. 공백 또는 문장 부호와 같은 한두 개의 비단어 문자가 따라오는 단어와 일치하는 정규식 패턴 `\b(\w+)(\W){1,2}`를 정의합니다. 정규식 는 다음 테이블과 같이 해석됩니다.

테이블 확장

요소	설명
\b	단어 경계에서 일치 항목 찾기를 시작합니다.
(\w+)	하나 이상의 단어 문자를 매치합니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
(\W){1,2}	비단어 문자를 한 번 또는 두 번 매치합니다. 이 그룹은 두 번째 캡처링 그룹입니다.

C#

```
static void NonWordCharacter()
{
```

```
    string pattern = @"\b(\w+)(\W){1,2}";
    string input = "The old, grey mare slowly walked across the narrow, green
pasture.";
    foreach (Match match in Regex.Matches(input, pattern))
    {
        Console.WriteLine(match.Value);
        Console.Write("   Non-word character(s):");
        CaptureCollection captures = match.Groups[2].Captures;
        for (int ctr = 0; ctr < captures.Count; ctr++)
            Console.Write(@"'{0}' (\u{1}){2}", captures[ctr].Value,
                          Convert.ToUInt16(captures[ctr].Value[0]).ToString("X4"),
                          ctr < captures.Count - 1 ? ", " : "");
        Console.WriteLine();
    }
}
// The example displays the following output:
//       The
//          Non-word character(s):' ' (\u0020)
//       old,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//       grey
//          Non-word character(s):' ' (\u0020)
//       mare
//          Non-word character(s):' ' (\u0020)
//       slowly
//          Non-word character(s):' ' (\u0020)
//       walked
//          Non-word character(s):' ' (\u0020)
//       across
//          Non-word character(s):' ' (\u0020)
//       the
//          Non-word character(s):' ' (\u0020)
//       narrow,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//       green
//          Non-word character(s):' ' (\u0020)
//       pasture.
//          Non-word character(s):'.' (\u002E)
```

두 번째 캡처링 그룹의 Group 개체가 단일 캡처된 비단어 문자만 포함하기 때문에 예제에서는 CaptureCollection 속성에 의해 반환되는 Group.Captures 개체에서 모든 캡처된 비단어 문자를 검색합니다.

공백 문자: \s

`\s`는 공백 문자에 일치합니다. 다음 표에 나열된 이스케이프 시퀀스 및 유니코드 범주와 동일합니다.

테이블 확장

범주	설명
`\f`	폼 피드 문자, \u000C.
`\n`	줄 바꿈 문자, \u000A.
`\r`	캐리지 리턴 문자 \u000D입니다.
`\t`	탭 문자, \u0009.
`\v`	세로 탭 문자, \u000B.
`\x85`	NEL(NEXT LINE) 문자, \u0085입니다.
`\p{Z}`	모든 구분 문자와 일치합니다. 여기에는 `Zs`, `Zl` 및 `Zp` 범주가 포함됩니다.

ECMAScript와 호환되는 동작을 지정한 경우 `\s`는 `[ \f\n\r\t\v]`와 같습니다. ECMAScript 정규식에 대한 자세한 내용은 정규식 옵션에서 "ECMAScript 일치 동작" 섹션을 참조하세요.

다음 예제에서는 `\s` 문자 클래스를 보여 줍니다. "s" 또는 "es"로 끝나고 그 뒤에 공백 문자나 입력 문자열의 끝이 있는 단어와 일치하는 정규식 패턴 `\b\w+(e)?s(\s|$)`를 정의합니다. 정규식는 다음 테이블과 같이 해석됩니다.

테이블 확장

요소	설명
\b	단어 경계에서 일치 항목 찾기를 시작합니다.
\w+	하나 이상의 단어 문자를 매치합니다.
(e)?	"e"를 0번 또는 1번 매칭합니다.
s	"s"를 찾습니다.
(\s|$)	공백 문자 또는 입력 문자열의 끝을 찾습니다.

C#

```
static void WhitespaceCharacter()
{
    string pattern = @"\b\w+(e)?s(\s|$)";
    string input = "matches stores stops leave leaves";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine(match.Value);
}
// The example displays the following output:
//       matches
//       stores
```

```
//       stops
//       leaves
```

공백이 아닌 문자: \S

`\S`는 비공백 문자를 매칭합니다. 정규식 패턴과 동일 `[^\f\n\r\t\v\x85\p{Z}]` 하거나 공백 문자와 일치하는 정규식 패턴과 반대 `\s`입니다. 자세한 내용은 공백 문자: \s를 참조하세요.

ECMAScript와 호환되는 동작을 지정한 경우 `\S`는 `[^ \f\n\r\t\v]`와 같습니다. ECMAScript 정규식에 대한 자세한 내용은 정규식 옵션에서 "ECMAScript 일치 동작" 섹션을 참조하세요.

다음 예제에서는 `\S` 언어 요소를 보여 줍니다. 정규식 패턴 `\b(\S+)\s?`는 공백 문자로 구분된 문자열을 찾습니다. 일치 GroupCollection 개체의 두 번째 요소는 일치하는 문자열을 포함합니다. 정규식은 다음 표에 나와 있는 것처럼 해석할 수 있습니다.

테이블 확장

요소	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(\S+)`	공백이 아닌 문자를 하나 이상 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\s?`	0번 또는 1번 나오는 공백 문자를 찾습니다.

C#

```
static void NonWhitespaceCharacter()
{
    string pattern = @"\b(\S+)\s?";
    string input = "This is the first sentence of the first paragraph. " +
                           "This is the second sentence.\n" +
                           "This is the only sentence of the second paragraph.";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine(match.Groups[1]);
}
// The example displays the following output:
//    This
//    is
//    the
//    first
//    sentence
//    of
//    the
//    first
//    paragraph.
//    This
//    is
```

```
//    the
//    second
//    sentence.
//    This
//    is
//    the
//    only
//    sentence
//    of
//    the
//    second
//    paragraph.
```

10진수 문자: \d

`\d`는 10진수와 일치합니다. 표준 10진수 0-9와 다른 많은 문자 집합의 10진수를 포함하는 정규식 패턴과 동일합니다 `\p{Nd}` .

ECMAScript와 호환되는 동작을 지정한 경우 `\d`는 `[0-9]`와 같습니다. ECMAScript 정규식에 대한 자세한 내용은 정규식 옵션에서 "ECMAScript 일치 동작" 섹션을 참조하세요.

다음 예제에서는 `\d` 언어 요소를 보여 줍니다. 입력 문자열이 미국 및 캐나다의 올바른 전화 번호를 나타내는지 여부를 테스트합니다. 정규식 패턴 `^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$`는 다음 테이블과 같이 정의됩니다.

테이블 확장

요소	설명
`^`	입력 문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
`\(?`	0개 또는 한 개의 리터럴 "(" 문자를 찾습니다.
`\d{3}`	세 개의 10진수를 찾습니다.
`\)?`	0개 또는 한 개의 리터럴 ")" 문자를 찾습니다.
`[\s-]`	하이픈 또는 공백 문자를 찾습니다.
`(\(?\d{3}\)?[\s-])?`	옵션 여는 괄호에 이어 세 개의 10진수, 옵션 닫는 괄호 및 공백 문자나 하이픈 0개 또는 1개를 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\d{3}-\d{4}`	세 개의 10진수와 하이픈 다음에 이어지는 네 개의 10진수를 찾습니다.
`$`	입력 문자열의 끝 부분을 찾습니다.

C#

```
static void DigitCharacter()
{
    string pattern = @"^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$";
    string[] inputs = { "111 111-1111", "222-2222", "222 333-444",
                        "(212) 111-1111", "111-AB1-1111",
                        "212-111-1111", "01 999-9999" };

    foreach (string input in inputs)
    {
        if (Regex.IsMatch(input, pattern))
            Console.WriteLine(input + ": matched");
        else
            Console.WriteLine(input + ": match failed");
    }
}
// The example displays the following output:
//       111 111-1111: matched
//       222-2222: matched
//       222 333-444: match failed
//       (212) 111-1111: matched
//       111-AB1-1111: match failed
//       212-111-1111: matched
//       01 999-9999: match failed
```

숫자가 아닌 문자: \D

`\D`는 숫자가 아닌 모든 문자와 일치합니다. 정규식 패턴과 `\P{Nd}` 동일합니다.

ECMAScript와 호환되는 동작을 지정한 경우 `\D`는 `[^0-9]`와 같습니다. ECMAScript 정규식에 대한 자세한 내용은 정규식 옵션에서 "ECMAScript 일치 동작" 섹션을 참조하세요.

다음 예제에서는 \D 언어 요소를 보여 줍니다. 부품 번호 같은 문자열이 10진수 문자 및 10진수가 아닌 문자의 적절한 조합으로 구성되어 있는지 여부를 테스트합니다. 정규식 패턴 `^\D\d{1,5}\D*$`는 다음 테이블과 같이 정의됩니다.

테이블 확장

요소	설명
`^`	입력 문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
`\D`	숫자가 아닌 문자를 찾습니다.
`\d{1,5}`	1~5의 10진수를 찾습니다.
`\D*`	0개 또는 하나 이상의 10진수가 아닌 문자를 찾습니다.
`$`	입력 문자열의 끝 부분을 찾습니다.

C#

```
static void NonDigitCharacter()
{
    string pattern = @"^\D\d{1,5}\D*$";
    string[] inputs = { "A1039C", "AA0001", "C18A", "Y938518" };

    foreach (string input in inputs)
    {
        if (Regex.IsMatch(input, pattern))
            Console.WriteLine(input + ": matched");
        else
            Console.WriteLine(input + ": match failed");
    }
}
// The example displays the following output:
//       A1039C: matched
//       AA0001: match failed
//       C18A: matched
//       Y938518: match failed
```

지원되는 유니코드 일반 범주

유니코드는 다음 표에 나와 있는 일반 범주를 정의합니다. 자세한 내용은 유니코드 문자 데이터베이스 Sec. 5.7.1, 표 12에서 하위 항목인 "UCD 파일 형식"과 "일반 범주 값"을 참조하세요.

테이블 확장

범주	설명
`Lu`	문자, 대문자
`Ll`	글자, 소문자
`Lt`	대문자, 제목 스타일
`Lm`	문자, 수정자
`Lo`	문자, 기타
`L`	모든 문자 여기에는 `Lu`, `Ll`, `Lt`, `Lm` 및 `Lo` 문자가 포함됩니다.
`Mn`	표시, 공백 없음
`Mc`	표시, 공백 조합
`Me`	표시, 묶기
`M`	모든 결합 기호입니다. 여기에는 `Mn`, `Mc` 및 `Me` 범주가 포함됩니다.

범주	설명
`Nd`	숫자, 10진수
`Nl`	숫자, 문자
`No`	숫자, 기타
`N`	모든 숫자. 여기에는 `Nd`, `Nl` 및 `No` 범주가 포함됩니다.
`Pc`	문장 부호, 연결자
`Pd`	문장 부호, 대시
`Ps`	문장 부호, 열기
`Pe`	문장 부호, 닫기
`Pi`	문장 부호, 초기 따옴표(사용량에 따라 Ps 또는 Pe처럼 동작할 수 있습니다).
`Pf`	문장 부호, 마지막 따옴표(사용량에 따라 Ps 또는 Pe처럼 동작할 수 있습니다).
`Po`	문장 부호, 기타
`P`	모든 구두점 문자. 여기에는 `Pc`, `Pd`, `Ps`, `Pe`, `Pi`, `Pf` 및 `Po` 범주가 포함됩니다.
`Sm`	기호, 수학
`Sc`	기호, 통화
`Sk`	기호, 한정자
`So`	기호, 기타
`S`	모든 기호. 여기에는 `Sm`, `Sc`, `Sk` 및 `So` 범주가 포함됩니다.
`Zs`	구분 기호, 공백
`Zl`	행 구분자, 선
`Zp`	구분 기호, 단락
`Z`	모든 구분 기호 문자. 여기에는 `Zs`, `Zl` 및 `Zp` 범주가 포함됩니다.
`Cc`	기타, 제어
`Cf`	기타, 서식
`Cs`	기타, 대리자
`Co`	기타, 개인 사용
`Cn`	기타, 할당되지 않음 또는 비문자

범주	설명
`C`	다른 모든 문자. 여기에는 `Cc`, `Cf`, `Cs`, `Co` 및 `Cn` 범주가 포함됩니다.

해당 문자를 GetUnicodeCategory 메서드로 전달하여 특정 문자의 유니코드 범주를 확인할 수 있습니다. 다음 예제에서는 GetUnicodeCategory 메서드를 사용하여 선택한 라틴 문자를 포함하는 배열에서 각 요소의 범주를 결정합니다.

C#

```
static void GetUnicodeCategory()
{
    char[] chars = { 'a', 'X', '8', ',', ' ', '\u0009', '!' };

    foreach (char ch in chars)
        Console.WriteLine($"'{Regex.Escape(ch.ToString())}':
{Char.GetUnicodeCategory(ch)}");
}
// The example displays the following output:
//       'a': LowercaseLetter
//       'X': UppercaseLetter
//       '8': DecimalDigitNumber
//       ',': OtherPunctuation
//       '\ ': SpaceSeparator
//       '\t': Control
//       '!': OtherPunctuation
```

지원되는 명명된 블록

.NET에서는 다음 표에 나와 있는 명명된 블록을 제공합니다. 지원되는 명명된 블록 집합은 유니코드 4.0 및 Perl 5.6을 기반으로 합니다. 명명된 블록을 사용하는 정규식에 대해서는 유니코드 범주 또는 유니코드 블록: \p{} 섹션을 참조하세요.

테이블 확장

코드 포인트 범위	블록 이름
0000 - 007F	`IsBasicLatin`
0080 - 00FF	`IsLatin-1Supplement`
0100 - 017F	`IsLatinExtended-A`
0180 - 024F	`IsLatinExtended-B`
0250 - 02AF	`IsIPAExtensions`

코드 포인트 범위	블록 이름
02B0 - 02FF	`IsSpacingModifierLetters`
0300 - 036F	`IsCombiningDiacriticalMarks`

0370 - 03FF	`IsGreek`
	
	또는
	
	`IsGreekandCoptic`

0400 - 04FF	`IsCyrillic`
0500 - 052F	`IsCyrillicSupplement`
0530 - 058F	`IsArmenian`
0590 - 05FF	`IsHebrew`
0600 - 06FF	`IsArabic`
0700 - 074F	`IsSyriac`
0780 - 07BF	`IsThaana`
0900 - 097F	`IsDevanagari`
0980 - 09FF	`IsBengali`
0A00 - 0A7F	`IsGurmukhi`
0A80 - 0AFF	`IsGujarati`
0B00 - 0B7F	`IsOriya`
0B80 - 0BFF	`IsTamil`
0C00 - 0C7F	`IsTelugu`
0C80 - 0CFF	`IsKannada`
0D00 - 0D7F	`IsMalayalam`
0D80 - 0DFF	`IsSinhala`
0E00 - 0E7F	`IsThai`
0E80 - 0EFF	`IsLao`
0F00 - 0FFF	`IsTibetan`

코드 포인트 범위	블록 이름
1000 - 109F	`IsMyanmar`
10A0 - 10FF	`IsGeorgian`
1100 - 11FF	`IsHangulJamo`
1200 - 137F	`IsEthiopic`
13A0 - 13FF	`IsCherokee`
1400 - 167F	`IsUnifiedCanadianAboriginalSyllabics`
1680 - 169F	`IsOgham`
16A0 - 16FF	`IsRunic`
1700 - 171F	`IsTagalog`
1720 - 173F	`IsHanunoo`
1740 - 175F	`IsBuhid`
1760 - 177F	`IsTagbanwa`
1780 - 17FF	`IsKhmer`
1800 - 18AF	`IsMongolian`
1900 - 194F	`IsLimbu`
1950년 - 197F	`IsTaiLe`
19E0 - 19FF	`IsKhmerSymbols`
1D00 - 1D7F	`IsPhoneticExtensions`
1E00 - 1EFF	`IsLatinExtendedAdditional`
1F00 - 1FFF	`IsGreekExtended`
2000 - 206F	`IsGeneralPunctuation`
2070 - 209F	`IsSuperscriptsandSubscripts`
20A0 - 20CF	`IsCurrencySymbols`
20D0 - 20FF	`IsCombiningDiacriticalMarksforSymbols` 또는

코드 포인트 범위	블록 이름
	`IsCombiningMarksforSymbols`
2100 - 214F	`IsLetterlikeSymbols`
2150 - 218F	`IsNumberForms`
2190 - 21FF	`IsArrows`
2200 - 22FF	`IsMathematicalOperators`
2300 - 23FF	`IsMiscellaneousTechnical`
2400 - 243F	`IsControlPictures`
2440 - 245F	`IsOpticalCharacterRecognition`
2460 - 24FF	`IsEnclosedAlphanumerics`
2500 - 257F	`IsBoxDrawing`
2580 - 259F	`IsBlockElements`
25A0 - 25FF	`IsGeometricShapes`
2600 - 26FF	`IsMiscellaneousSymbols`
2700 - 27BF	`IsDingbats`
27C0 - 27EF	`IsMiscellaneousMathematicalSymbols-A`
27F0 - 27FF	`IsSupplementalArrows-A`
2800 - 28FF	`IsBraillePatterns`
2900 - 297F	`IsSupplementalArrows-B`
2980 - 29FF	`IsMiscellaneousMathematicalSymbols-B`
2A00 - 2AFF	`IsSupplementalMathematicalOperators`
2B00 - 2BFF	`IsMiscellaneousSymbolsandArrows`
2E80 - 2EFF	`IsCJKRadicalsSupplement`
2F00 - 2FDF	`IsKangxiRadicals`
2FF0 - 2FFF	`IsIdeographicDescriptionCharacters`
3000 - 303F	`IsCJKSymbolsandPunctuation`

코드 포인트 범위	블록 이름
3040 - 309F	`IsHiragana`
30A0 - 30FF	`IsKatakana`
3100 - 312F	`IsBopomofo`
3130 - 318F	`IsHangulCompatibilityJamo`
3190 - 319F	`IsKanbun`
31A0 - 31BF	`IsBopomofoExtended`
31F0 - 31FF	`IsKatakanaPhoneticExtensions`
3200 - 32FF	`IsEnclosedCJKLettersandMonths`
3300 - 33FF	`IsCJKCompatibility`
3400 - 4DBF	`IsCJKUnifiedIdeographsExtensionA`
4DC0 - 4DFF	`IsYijingHexagramSymbols`
4E00 - 9FFF	`IsCJKUnifiedIdeographs`
A000 - A48F	`IsYiSyllables`
A490 - A4CF	`IsYiRadicals`
AC00 - D7AF	`IsHangulSyllables`
D800 - DB7F	`IsHighSurrogates`
DB80 - DBFF	`IsHighPrivateUseSurrogates`
DC00 - DFFF	`IsLowSurrogates`
E000 - F8FF	`IsPrivateUse` 또는 `IsPrivateUseArea`
F900 - FAFF	`IsCJKCompatibilityIdeographs`
FB00 - FB4F	`IsAlphabeticPresentationForms`
FB50 - FDFF	`IsArabicPresentationForms-A`
FE00 - FE0F	`IsVariationSelectors`
FE20 - FE2F	`IsCombiningHalfMarks`
FE30 - FE4F	`IsCJKCompatibilityForms`
FE50 - FE6F	`IsSmallFormVariants`

코드 포인트 범위	블록 이름
FE70 - FEFF	`IsArabicPresentationForms-B`
FF00 - FFEF	`IsHalfwidthandFullwidthForms`
FFF0 - FFFF	`IsSpecials`

문자 클래스 빼기: [base_group - [excluded_group]]

문자 클래스는 문자 집합을 정의합니다. 문자 클래스 빼기는 한 문자 클래스에서 다른 문자 클래스의 문자를 제외한 결과로 문자 집합을 생성합니다.

문자 클래스 빼기 식의 형식은 다음과 같습니다.

`[` *base_group* `-[` *excluded_group* `]]`

대괄호(`[]`)와 하이픈(`-`)은 필수 요소입니다. *base_group*은 긍정 문자 그룹 또는 부정 문자 그룹입니다. *excluded_group* 구성 요소는 다른 긍정 또는 부정 문자 그룹이거나 다른 문자 클래스 빼기 식입니다. 즉, 문자 클래스 빼기 식을 중첩할 수 있습니다.

예를 들어, "a"부터 "z"까지의 문자 범위로 구성된 기본 그룹이 있다고 가정합니다. 문자 "m"을 제외한 기본 그룹으로 구성된 문자 집합을 정의하려면 `[a-z-[m]]`을 사용합니다. 문자 "d", "j" 및 "p"를 제외한 기본 그룹으로 구성된 문자 집합을 정의하려면 `[a-z-[djp]]`를 사용합니다. "m"부터 "p"까지의 문자 범위를 제외한 기본 그룹으로 구성된 문자 집합을 정의하려면 `[a-z-[m-p]]`를 사용합니다.

중첩된 문자 클래스 빼기 식인 `[a-z-[d-w-[m-o]]]`를 고려합니다. 가장 안쪽 문자 범위에서 바깥쪽으로 식이 계산됩니다. 먼저 'd'부터 'w'까지의 문자 범위에서 "m"부터 "o"까지의 문자 범위를 뺍니다. 그러면 "d"부터 "l"까지와 "p"부터 "w"까지의 문자로 구성된 문자 집합이 생성됩니다. 그런 다음 "a"부터 "z"까지의 문자 범위에서 이 집합을 뺍니다. 그러면 문자 집합 `[abcmnoxyz]`가 생성됩니다.

모든 문자 클래스에 문자 클래스 빼기를 사용할 수 있습니다. 모든 유니코드 문자(\u0000에서 \uFFFF까지) 중 공백 문자(`\s`), 문장 부호 일반 범주의 문자(`\p{P}`), 명명된 블록 `IsGreek`의 문자(`\p{IsGreek}`), 그리고 유니코드 NEXT LINE 제어 문자(\x85)를 제외한 문자 집합을 정의하려면 `[\u0000-\uFFFF-[\s\p{P}\p{IsGreek}\x85]]`를 사용하십시오.

유용한 결과를 생성하는 문자 클래스 빼기 식에 대한 문자 클래스를 선택합니다. 아무것도 일치시킬 수 없는 빈 문자 집합을 생성하는 식이나 원래 기본 그룹과 동일한 식을 사용하지 마세요. 예를 들어, `[\p{IsBasicLatin}-[\x00-\x7F]]` 일반 범주에서 `IsBasicLatin` 문자 범위의 모든 문자를 빼는 `IsBasicLatin` 식을 사용하면 빈 집합이 생성됩니다. 마찬가지로 `[a-z-[0-9]]` 식을 사용

하면 원래 기본 그룹이 생성됩니다. "a"에서 "z"의 문자 범위인 기본 그룹에는 "0"에서 "9"의 10진수 문자 범위인 제외된 그룹에 문자가 포함되지 않기 때문입니다.

다음 예제에서는 입력 문자열에서 0과 홀수를 찾는 정규식 `^[0-9-[2468]]+$`를 정의합니다. 정규식 는 다음 테이블과 같이 해석됩니다.

테이블 확장

요소	설명
^	입력 문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
`[0-9-[2468]]+`	0에서 9까지의 문자 중에서 2, 4, 6, 8을 제외한 숫자가 하나 이상 있는 경우를 일치시키세요. 0 또는 홀수 자릿수가 한 번 이상 나타나는 경우를 찾습니다.
$	입력 문자열의 끝에서 매칭을 종료합니다.

C#

```
static void CharacterClassSubtraction()
{
    string[] inputs = { "123", "13579753", "3557798", "335599901" };
    string pattern = @"^[0-9-[2468]]+$";

    foreach (string input in inputs)
    {
        Match match = Regex.Match(input, pattern);
        if (match.Success)
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//       13579753
//       335599901
```

참고하기

- GetUnicodeCategory
- 정규식 언어 - 빠른 참조
- 정규식 옵션

Last updated on 2026. 03. 31.

정규 표현식의 앵커 기능

아티클 • 2025. 03. 31.

앵커 또는 원자성 0 너비 어설션은 일치가 발생해야 하는 문자열의 위치를 지정합니다. 검색 식에서 앵커를 사용하는 경우 정규식 엔진은 문자열을 통과하거나 문자를 사용하지 않습니다. 지정된 위치에서만 일치 항목을 찾습니다. 예를 들어 `^` 일치 항목이 줄 또는 문자열의 시작 부분에서 시작되도록 지정합니다. 따라서 정규식 `^http:`은 "http:"가 줄의 시작 부분에 있을 때만 일치합니다. 다음 표에서는 .NET의 정규식에서 지원하는 앵커를 나열합니다.

테이블 확장

닻	설명
`^`	기본적으로 일치 항목은 문자열의 시작 부분에서 발생해야 합니다. 여러 줄 모드에서는 줄의 시작 부분에서 발생해야 합니다. 자세한 내용은 문자열 시작 또는 줄참조하세요.
`$`	기본적으로 일치는 문자열의 끝에서 또는 문자열의 끝에 `\n` 전에 발생해야 합니다. 여러 줄 모드에서는 줄의 끝에서 또는 줄의 끝에서 `\n` 전에 발생해야 합니다. 자세한 내용은 문자열 끝 또는 줄참조하세요.
`\A`	일치 항목은 문자열의 시작 부분에서만 발생해야 합니다(다중 줄 지원 없음). 자세한 내용은 문자열 시작 전용 을 참조하세요.
`\Z`	일치는 문자열의 끝에서 또는 문자열의 끝에 `\n` 전에 발생해야 합니다. 자세한 내용은 문자열 끝이나 줄의 끝 앞을 참조하세요.
`\z`	일치는 문자열의 끝에만 발생해야 합니다. 자세한 내용은 문자열의 끝만참조하세요.
`\G`	일치 항목은 이전 일치가 종료된 위치에서 시작하거나 이전 일치 항목이 없는 경우 일치가 시작된 문자열의 위치에서 시작해야 합니다. 자세한 내용은 연속된 일치을 참조하세요.
`\b`	매치는 단어 경계에 있어야 합니다. 자세한 내용은 단어 경계를 확인하세요.
`\B`	단어 경계에서 매칭이 발생하지 않아야 합니다. 자세한 내용은 비단어 경계 을 참조하세요.

문자열 또는 줄의 시작: ^

기본적으로 `^` 앵커는 문자열의 첫 번째 문자 위치에서 다음 패턴을 시작해야 한다고 지정합니다. `^`을 RegexOptions.Multiline 옵션과 함께 사용하는 경우(정규식 옵션참조), 각 줄의 시작 부분에서 반드시 일치가 발생해야 합니다.

다음 예제에서는 일부 프로 야구 팀이 존재했던 연도에 대한 정보를 추출하는 정규식에서 `^` 앵커를 사용합니다. 이 예제에서는 Regex.Matches 메서드의 두 오버로드를 호출합

니다.

- Matches(String, String) 오버로드에 대한 호출은 정규식 패턴과 일치하는 입력 문자열의 첫 번째 부분 문자열만 찾습니다.
- Matches(String, String, RegexOptions) 오버로드에 `options` 매개 변수를 RegexOptions.Multiline로 설정하여 호출하면 5개의 부분 문자열을 모두 찾습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-
1957\n" +
                       "Chicago Cubs, National League, 1903-present\n" +
                       "Detroit Tigers, American League, 1901-present\n" +
                       "New York Giants, National League, 1885-1957\n" +
                       "Washington Senators, American League, 1901-1960\n";
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+";
        Match match;

        match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();
    }
```

```
}
// The example displays the following output:
//    The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//
//    The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
```

`^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+` 정규식 패턴은 다음 표와 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	입력 문자열의 시작 부분에서 일치를 시작합니다(또는 메서드가 RegexOptions.Multiline 옵션으로 호출되는 경우 줄의 시작 부분).
`((\w+(\s?)){2,}`	단어 문자가 하나 이상 있고 그 뒤에 0개 또는 1개의 공백이 있는 패턴을 최소 두 번 이상 매치합니다. 이 그룹은 첫 번째 캡처 그룹입니다. 또한 이 식은 두 번째 및 세 번째 캡처링 그룹을 정의합니다. 두 번째 캡처 그룹은 캡처된 단어로 구성되고 세 번째는 캡처된 공백으로 구성됩니다.
`,\s`	쉼표와 공백 문자에 일치하는 것을 찾습니다.
`(\w+\s\w+)`	하나 이상의 단어 문자와 공백, 그리고 다시 하나 이상의 단어 문자가 있는지 확인하세요. 네 번째 캡처링 그룹입니다.
`,`	쉼표와 일치하기
`\s\d{4}`	공백 뒤에 오는 4자리 10진수를 찾습니다.
`(-(\d{4}|present))?`	0개 또는 1개의 하이픈이 이어지는 네 자리 숫자 또는 문자열 "present"를 매칭합니다. 이는 여섯 번째 캡처링 그룹입니다. 또한 일곱 번째 캡처링 그룹도 포함됩니다.
`,?`	쉼표가 0개 또는 1개 있는지 확인합니다.
`(\s\d{4}(-(\d{4}|present))?,?)+`	공백, 4개의 숫자, 하이픈 뒤에 4개의 숫자 또는 문자열 "present"가 뒤따를 수 있으며, 마지막으로 0개 또는 1개의 쉼표가 나타나는 패턴을 하나 이상 찾습니다. 다섯 번째 캡처 그룹입니다.

문자열 또는 줄의 끝: $

`$` 앵커는 입력 문자열의 끝에서 또는 입력 문자열의 끝에 `\n` 전에 이전 패턴이 발생해야 하므로 지정합니다.

`$`를 RegexOptions.Multiline 옵션과 함께 사용하면, 줄의 끝에서도 매치가 발생할 수 있습니다. `$`은 `\n`에서는 충족되지만, `\r\n`(캐리지 리턴과 줄 바꿈 문자의 조합, 즉 CR/LF)에서는 충족되지 않습니다. CR/LF 문자 조합을 처리하려면 정규식 패턴에 `\r?$` 포함합니다. `\r?$`과의 일치 항목에 `\r`이 포함됩니다.

다음 예제에서는 문자열 시작 또는 줄 섹션의 예제에서 사용되는 정규식 패턴에 `$` 앵커를 추가합니다. 텍스트 5줄을 포함하는 원래 입력 문자열과 함께 사용하면 첫 번째 줄의 끝이 `$` 패턴과 일치하지 않으므로 Regex.Matches(String, String) 메서드가 일치 항목을 찾을 수 없습니다. 원래 입력 문자열이 문자열 배열로 분할되면 Regex.Matches(String, String) 메서드는 5개 줄 각각을 일치시키는 데 성공합니다. `options` 매개 변수를 RegexOptions.Multiline설정하여 Regex.Matches(String, String, RegexOptions) 메서드를 호출하면 정규식 패턴이 캐리지 리턴 문자 `\r`고려하지 않기 때문에 일치하는 항목을 찾을 수 없습니다. 그러나 정규식 패턴에서 `$`을(를) `\r?$`으로 대체하여 수정하면, `options` 매개 변수가 RegexOptions.Multiline로 설정된 상태로 Regex.Matches(String, String, RegexOptions) 메서드를 호출할 때 다시 5개의 일치 항목을 찾을 수 있습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string cr = Environment.NewLine;
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-
1957" + cr +
                       "Chicago Cubs, National League, 1903-present" + cr +
                       "Detroit Tigers, American League, 1901-present" + cr
+
                       "New York Giants, National League, 1885-1957" + cr +
                       "Washington Senators, American League, 1901-1960" +
cr;
        Match match;

        string basePattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+";
        string pattern = basePattern + "$";
        Console.WriteLine("Attempting to match the entire input string:");
        match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
```

```
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        string[] teams = input.Split(new String[] { cr },
StringSplitOptions.RemoveEmptyEntries);
        Console.WriteLine("Attempting to match each element in a string
array:");
        foreach (string team in teams)
        {
            match = Regex.Match(team, pattern);
            if (match.Success)
            {
                Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
                foreach (Capture capture in match.Groups[5].Captures)
                    Console.Write(capture.Value);
                Console.WriteLine(".");
            }
        }
        Console.WriteLine();

        Console.WriteLine("Attempting to match each line of an input string
with '$':");
        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        pattern = basePattern + "\r?$";
        Console.WriteLine(@"Attempting to match each line of an input string
with '\r?$':");
        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
```

```
        Console.WriteLine();
    }
}
// The example displays the following output:
//    Attempting to match the entire input string:
//
//    Attempting to match each element in a string array:
//    The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
//
//    Attempting to match each line of an input string with '$':
//
//    Attempting to match each line of an input string with '\r?$':
//    The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
```

문자열 시작 전용: \A

`\A` 앵커는 입력 문자열의 시작 부분에서 일치 항목이 발생해야 한다고 지정합니다. `\A` RegexOptions.Multiline 옵션을 무시한다는 점을 제외하고 `^` 앵커와 동일합니다. 따라서 여러 줄 입력 문자열에서 첫 번째 줄의 시작 부분만 일치시킬 수 있습니다.

다음 예제는 `^` 및 `$` 앵커에 대한 예제와 유사합니다. 일부 프로 야구 팀이 존재했던 연도에 대한 정보를 추출하기 위해 정규식에서 `\A` 앵커를 사용합니다. 입력 문자열에는 5개의 줄이 포함됩니다. Regex.Matches(String, String, RegexOptions) 메서드에 대한 호출은 정규식 패턴과 일치하는 입력 문자열의 첫 번째 부분 문자열만 찾습니다. 예제에서와 같이 Multiline 옵션은 효과가 없습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-
1957\n" +
                       "Chicago Cubs, National League, 1903-present\n" +
                       "Detroit Tigers, American League, 1901-present\n" +
```

```
                      "New York Giants, National League, 1885-1957\n" +
                      "Washington Senators, American League, 1901-1960\n";

        string pattern = @"\A((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+";

        Match match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
```

문자열 끝 또는 마지막 줄바꿈 전: \Z

`\Z` 앵커는 입력 문자열의 끝에서 또는 입력 문자열의 끝에 `\n` 전에 일치가 발생해야 하므로 지정합니다. `\Z` RegexOptions.Multiline 옵션을 무시한다는 점을 제외하고 `$` 앵커와 동일합니다. 따라서 여러 줄 문자열에서는 마지막 줄의 끝이나 `\n`앞의 마지막 줄만 충족할 수 있습니다.

유의하십시오: `\Z`는 `\n`에서 충족되지만 `\r\n`(CR/LF 문자 조합)에서는 충족되지 않습니다. CR/LF를 `\n`것처럼 처리하려면 정규식 패턴에 `\r?\Z` 포함합니다. 이렇게 하면 `\r`이 일치의 일부가 됩니다.

다음 예제에서는 문자열 시작 또는 줄 섹션의 예제와 유사한 정규식에서 `\Z` 앵커를 사용합니다. 이 섹션에서는 일부 프로 야구 팀이 존재했던 연도에 대한 정보를 추출합니다. 정규식 `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z`의 하위 식 `\r?\Z`은 문자열 끝 부분이나 `\n` 또는 `\r\n`로 끝나는 문자열의 끝에서 충족됩니다. 결과적으로 배열의 각 요소는 정규식 패턴과 일치합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
```

```
{
    public static void Main()
    {
        string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912,
1932-1957",
                            "Chicago Cubs, National League, 1903-present" +
Environment.NewLine,
                            "Detroit Tigers, American League, 1901-present" +
Regex.Unescape(@"\n"),
                            "New York Giants, National League, 1885-1957",
                            "Washington Senators, American League, 1901-1960"
+ Environment.NewLine};
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+\r?\Z";

        foreach (string input in inputs)
        {
            Console.WriteLine(Regex.Escape(input));
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine("   Match succeeded.");
            else
                Console.WriteLine("   Match failed.");
        }
    }
}
// The example displays the following output:
//    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match succeeded.
//    Detroit\ Tigers,\ American\ League,\ 1901-present\n
//       Match succeeded.
//    New\ York\ Giants,\ National\ League,\ 1885-1957
//       Match succeeded.
//    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//       Match succeeded.
```

문자열의 끝만: \z

`\z` 앵커는 입력 문자열의 끝에서 일치가 발생해야 임을 지정합니다. `$` 언어 요소와 마찬가지로 `\z` RegexOptions.Multiline 옵션을 무시합니다. `\Z` 언어 요소와 달리 `\z` 문자열 끝에 있는 `\n` 문자에 의해 충족되지 않습니다. 따라서 입력 문자열의 끝만 일치시킬 수 있습니다.

다음 예제에서는 이전 섹션의 예제와 동일한 정규식에서 `\z` 앵커를 사용하여 일부 프로 야구 팀이 존재했던 연도에 대한 정보를 추출합니다. 이 예제에서는 문자열 배열의 5개 요소를 각각 정규식 패턴 `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z` 일치시키려고 시도합니다. 문자열 중 2개는 캐리지 리턴과 줄

바꿈 문자로 끝나고, 하나는 줄 바꿈 문자로 끝나며, 두 개는 캐리지 리턴이나 줄 바꿈 문자 없이 끝납니다. 출력에서 알 수 있듯이 캐리지 리턴 또는 줄 바꿈 문자가 없는 문자열만 패턴과 일치합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912,
1932-1957",
                            "Chicago Cubs, National League, 1903-present" +
Environment.NewLine,
                            "Detroit Tigers, American League, 1901-present\n",
                            "New York Giants, National League, 1885-1957",
                            "Washington Senators, American League, 1901-1960"
+ Environment.NewLine };
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+\r?\z";

        foreach (string input in inputs)
        {
            Console.WriteLine(Regex.Escape(input));
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine("   Match succeeded.");
            else
                Console.WriteLine("   Match failed.");
        }
    }
}
// The example displays the following output:
//    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match failed.
//    Detroit\ Tigers,\ American\ League,\ 1901-present\n
//       Match failed.
//    New\ York\ Giants,\ National\ League,\ 1885-1957
//       Match succeeded.
//    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//       Match failed.
```

연속 일치: \G

`\G` 앵커는 일치가 시작된 문자열의 위치에서 이전 일치 항목이 종료된 지점이나 이전 일치 항목이 없는 지점에서 일치가 발생해야 한다고 지정합니다. 이 앵커를 Regex.Matches 또는 Match.NextMatch 메서드와 함께 사용하면 모든 일치 항목이 연속되도록 합니다.

> **팁**
>
> 일반적으로 패턴의 왼쪽 끝에 `\G` 앵커를 배치합니다. 드물게 오른쪽에서 왼쪽으로 검색을 수행하는 경우 패턴의 오른쪽 끝에 `\G` 앵커를 배치합니다.

다음 예제에서는 정규식을 사용하여 쉼표로 구분된 문자열에서 설치류 종의 이름을 추출합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "capybara,squirrel,chipmunk,porcupine,gopher," +
                       "beaver,groundhog,hamster,guinea pig,gerbil," +
                       "chinchilla,prairie dog,mouse,rat";
        string pattern = @"\G(\w+\s?\w*),?";
        Match match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.WriteLine(match.Groups[1].Value);
            match = match.NextMatch();
        }
    }
}
// The example displays the following output:
//       capybara
//       squirrel
//       chipmunk
//       porcupine
//       gopher
//       beaver
//       groundhog
//       hamster
//       guinea pig
//       gerbil
//       chinchilla
//       prairie dog
//       mouse
//       rat
```

`\G(\w+\s?\w*),?` 정규식은 다음 표와 같이 해석됩니다.

테이블 확장

패턴	설명
`\G`	마지막 경기가 끝난 위치를 시작합니다.
`\w+`	하나 이상의 단어 문자를 일치시킵니다.
`\s?`	0개 또는 1개의 공백을 찾습니다.
`\w*`	0개 이상의 단어 문자를 찾습니다.
`(\w+\s?\w*)`	하나 이상의 단어 문자 뒤에 0개 또는 1개의 공백이 있고, 그 후에 0개 이상의 단어 문자가 이어집니다. 이 그룹은 첫 번째 캡처 그룹입니다.
`,?`	리터럴 쉼표 문자를 0개 또는 1개씩 일치시킵니다.

단어 경계: \b

`\b` 앵커는 단어 문자(`\w` 언어 요소)와 단어가 아닌 문자(`\W` 언어 요소) 사이의 경계에서 일치가 발생해야 임을 지정합니다. 단어 문자는 영숫자 문자와 밑줄로 구성됩니다. 단어가 아닌 문자는 영숫자 또는 밑줄이 아닌 문자입니다. 자세한 내용은 문자 클래스참조하세요. 문자열의 시작 또는 끝에 있는 단어 경계에서도 일치가 발생할 수 있습니다.

`\b` 앵커는 하위 식이 단어의 시작이나 끝이 아닌 전체 단어와 일치하는지 확인하는 데 자주 사용됩니다. 다음 예제의 정규식 `\bare\w*\b` 이 사용량을 보여 줍니다. 부분 문자열 "are"로 시작하는 모든 단어와 일치합니다. 이 예제의 출력은 `\b` 입력 문자열의 시작과 끝 모두와 일치한다는 것을 보여 줍니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "area bare arena mare";
        string pattern = @"\bare\w*\b";
        Console.WriteLine("Words that begin with 'are':");
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine($"'{match.Value}' found at position
{match.Index}");
    }
}
```

```
// The example displays the following output:
//       Words that begin with 'are':
//       'area' found at position 0
//       'arena' found at position 10
```

정규식 패턴은 다음 표와 같이 해석됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 일치를 시작합니다.
`are`	부분 문자열 "are"를 찾기.
`\w*`	0개 이상의 단어 문자를 찾습니다.
`\b`	단어 경계에서 매치를 종료합니다.

비 단어 경계: \B

`\B` 앵커는 단어 경계에서 일치가 발생하지 않도록 지정합니다. `\b` 앵커와 반대입니다.

다음 예제에서는 `\B` 앵커를 사용하여 단어에서 부분 문자열 "qu"의 발생 항목을 찾습니다. 정규식 패턴 `\Bqu\w+` 단어를 시작하지 않고 단어의 끝까지 계속되는 "qu"로 시작하는 부분 문자열과 일치합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "equity queen equip acquaint quiet";
        string pattern = @"\Bqu\w+";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine($"'{match.Value}' found at position
{match.Index}");
    }
}
// The example displays the following output:
//       'quity' found at position 1
//       'quip' found at position 14
//       'quaint' found at position 21
```

정규식 패턴은 다음 표와 같이 해석됩니다.

테이블 확장

패턴	설명
`\B`	단어 경계에서 매치를 시작하지 마세요.
`qu`	부분 문자열 "qu"를 일치시키십시오.
`\w+`	하나 이상의 단어 문자를 찾습니다.

참고하십시오

- 정규식 언어 - 빠른 참조
- 정규식 옵션

정규식의 그룹화 구문

아티클 • 2023. 04. 08.

그룹화 구문은 정규식의 하위 식을 나타내며 입력 문자열의 부분 문자열을 캡처합니다. 그룹화 구문은 다음과 같은 경우에 사용할 수 있습니다.

- 입력 문자열에서 반복되는 하위 식과 일치합니다.
- 여러 정규식 언어 요소가 있는 하위 식에 수량자를 적용합니다. 수량자에 대한 자세한 내용은 Quantifiers를 참조하세요.
- 및 Match.Result 메서드에서 반환되는 문자열에 하위 식이 Regex.Replace 포함됩니다.
- Match.Groups 속성에서 개별 하위 식을 검색하여 전체적으로 일치하는 텍스트와 별도로 처리합니다.

다음 표에는 .NET 정규식 엔진에서 지원하는 그룹화 구문이 나열되어 있으며 캡처 중인지 아니면 캡슐화되지 않는지를 나타냅니다.

그룹화 구문	캡처링 또는 비 캡처링
일치하는 하위 식	캡처 중
명명된 일치하는 하위 식	캡처 중
균형 조정 그룹 정의	캡처 중
비 캡처링 그룹	비 캡처링
그룹 옵션	비 캡처링
너비가 0인 긍정 lookahead 어설션	비 캡처링
너비가 0인 부정 lookahead 어설션	비 캡처링
너비가 0인 긍정 lookbehind 어설션	비 캡처링
너비가 0인 부정 lookbehind 어설션	비 캡처링
원자성 그룹	비 캡처링

그룹 및 정규식 개체 모델에 대한 자세한 내용은 그룹화 구문 및 정규식 개체를 참조하세요.

일치하는 하위 식

다음 그룹화 구문은 일치하는 하위 식을 캡처합니다.

`(부분식)`

여기서 *하위 식* 은 유효한 정규식 패턴입니다. 괄호를 사용하는 캡처는 1부터 정규식의 여는 괄호 순서에 따라 왼쪽에서 오른쪽으로 자동으로 번호가 매겨집니다. 그러나 명명된 캡처 그룹은 이름이 지정되지 않은 캡처 그룹 다음에 항상 마지막 순서로 정렬됩니다. 번호가 0인 캡처는 전체 정규식 패턴과 일치하는 텍스트입니다.

> **참고**
>
> 기본적으로 `(subexpression)` 언어 요소는 일치하는 하위 식을 캡처합니다. RegexOptions 그러나 정규식 패턴 일치 메서드의 매개 변수에 플래그가 포함 RegexOptions.ExplicitCapture 되거나 이 하위 식에 옵션이 적용되는 경우 `n` (이 문서의 뒷부분에 있는 **그룹 옵션** 참조) 일치하는 하위 식이 캡처되지 않습니다.

캡처된 그룹에는 다음과 같은 4가지 방법으로 액세스할 수 있습니다.

- 정규식 내의 역참조 구문을 사용합니다. 일치하는 하위 식은 `\number` 구문을 통해 동일한 정규식에서 참조됩니다. 여기서 *number* 는 캡처된 하위 식의 서수입니다.
- 정규식 내의 명명된 역참조 구문을 사용합니다. 일치하는 하위 식은 `\k<name>` 구문을 통해 동일한 정규식에서 참조됩니다. 여기서 *name* 은 캡처링 그룹의 이름) 또는 `\k<number>` 구문을 통해 동일한 정규식에서 참조됩니다. 여기서 *number* 는 캡처링 그룹의 서수)을 통해 동일한 정규식에서 참조됩니다. 캡처링 그룹은 해당 서수와 같은 기본 이름을 갖고 있습니다. 자세한 내용은 이 항목 뒷부분의 명명된 일치하는 하위 식 을 참조하세요.
- Regex.Replace 또는 Match.Result 메서드 호출에서 `$number` 바꾸기 시퀀스를 사용합니다. 여기서 *number*는 캡처된 하위 식의 서수입니다.
- 프로그래밍 방식으로 GroupCollection 속성에서 반환하는 Match.Groups 개체를 사용합니다. 컬렉션에서 위치 0에 있는 멤버는 전체 정규식 일치를 나타냅니다. 각 후속 멤버는 일치하는 하위 식을 나타냅니다. 자세한 내용은 Grouping Constructs and Regular Expression Objects 섹션을 참조하세요.

다음 예제에서는 텍스트에서 중복된 단어를 식별하는 정규식을 보여 줍니다. 정규식 패턴의 두 캡처링 그룹은 중복된 단어의 두 인스턴스를 나타냅니다. 두 번째 인스턴스는 입력 문자열의 해당 시작 위치를 보고하기 위해 캡처됩니다.

C#

```
using System;
using System.Text.RegularExpressions;
```

```
public class Example
{
   public static void Main()
   {
      string pattern = @"(\w+)\s(\1)\W";
      string input = "He said that that was the the correct answer.";
      foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
         Console.WriteLine("Duplicate '{0}' found at positions {1} and
{2}.",
                           match.Groups[1].Value, match.Groups[1].Index,
match.Groups[2].Index);
   }
}
// The example displays the following output:
//       Duplicate 'that' found at positions 8 and 13.
//       Duplicate 'the' found at positions 22 and 26.
```

정규식 패턴은 다음과 같습니다.

`(\w+)\s(\1)\W`

다음 테이블은 정규식 패턴이 해석되는 방법을 보여 줍니다.

무늬	설명
`(\w+)`	하나 이상의 단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\s`	공백 문자를 찾습니다.
`(\1)`	캡처된 첫 번째 그룹의 문자열을 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다. 이 예제에서는 `Match.Index` 개체를 사용합니다.
`\W`	공백 및 문장 부호를 포함하여 비단어 문자를 찾습니다. 그러면 정규식 패턴이 캡처된 첫 번째 그룹의 단어로 시작하는 단어를 찾지 못합니다.

명명된 일치하는 하위 식

다음 그룹화 구문은 일치하는 하위 식을 캡처하고 사용자가 이름 또는 번호로 해당 하위 식에 액세스할 수 있게 해줍니다.

`(?<name>subexpression)`

또는

`(?'name'subexpression)`

여기서 *name* 은 유효한 그룹 이름이고 *하위* 식은 유효한 정규식 패턴입니다. *name* 은 문장 부호 문자를 포함해서는 안 되며 숫자로 시작할 수 없습니다.

> ⓘ **참고**
>
> 정규식 패턴 일치 메서드의 **RegexOptions** 매개 변수가 **RegexOptions.ExplicitCapture** 플래그를 포함하거나, `n` 옵션이 이 하위 식에 적용된 경우(이 항목 뒷부분의 **그룹 옵션** 참조) 하위 식을 캡처하는 유일한 방법은 명시적으로 캡처링 그룹의 이름을 지정하는 것입니다.

명명된 캡처된 그룹에는 다음과 같은 방법으로 액세스할 수 있습니다.

- 정규식 내의 명명된 역참조 구문을 사용합니다. 일치하는 하위 식은 `\k<`*name*`>`구문을 통해 동일한 정규식에서 참조됩니다. 여기서 *name* 은 캡처된 하위 식의 이름입니다.

- 정규식 내의 역참조 구문을 사용합니다. 일치하는 하위 식은 `\`*number*구문을 통해 동일한 정규식에서 참조됩니다. 여기서 *number* 는 캡처된 하위 식의 서수입니다. 명명된 일치하는 하위 식은 하위 식을 일치시킨 후 왼쪽에서 오른쪽으로 연속적으로 번호가 매겨집니다.

- Regex.Replace 또는 Match.Result 메서드 호출에서 `${`*name*`}` 바꾸기 시퀀스를 사용합니다. 여기서 *name*은 캡처된 하위 식의 이름입니다.

- Regex.Replace 또는 Match.Result 메서드 호출에서 `$`*number* 바꾸기 시퀀스를 사용합니다. 여기서 *number*는 캡처된 하위 식의 서수입니다.

- 프로그래밍 방식으로 GroupCollection 속성에서 반환하는 Match.Groups 개체를 사용합니다. 컬렉션에서 위치 0에 있는 멤버는 전체 정규식 일치를 나타냅니다. 각 후속 멤버는 일치하는 하위 식을 나타냅니다. 명명된 캡처된 그룹은 캡처된 그룹의 번호를 매긴 후 컬렉션에 저장됩니다.

- 프로그래밍 방식으로 하위 식 이름을 GroupCollection 개체의 인덱서(C#의 경우) 또는 해당 Item[] 속성(Visual Basic의 경우)에 제공합니다.

단순 정규식 패턴이 프로그래밍 방식으로 또는 정규식 언어 구문을 사용하여, 번호가 매겨진(명명되지 않은) 그룹 및 명명된 그룹을 참조할 수 있는 방법을 보여 줍니다. 정규식 `((?<One>abc)\d+)?(?<Two>xyz)(.*)` 는 번호 및 이름으로 다음 캡처링 그룹을 생성합니다. 첫 번째 캡처링 그룹(번호 0)은 항상 전체 패턴을 지칭합니다. (명명된 그룹은 항상 마지막 순서로 정렬됩니다.)

number	이름	무늬

number	이름	무늬
0	0(기본 이름)	`((?<One>abc)\d+)?(?<Two>xyz)(.*)`
1	1(기본 이름)	`((?<One>abc)\d+)`
2	2(기본 이름)	`(.*)`
3	하나	`(?<One>abc)`
4	둘	`(?<Two>xyz)`

다음 예제에서는 중복된 단어와 중복된 각 단어 바로 뒤에 나오는 단어를 식별하는 정규식을 보여 줍니다. 정규식 패턴은 중복된 단어를 나타내는 라는 두 개의 하위 `duplicateWord`식과 중복된 단어를 따르는 단어를 나타내는 및 `nextWord`을 정의합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?
<nextWord>\w+)";
      string input = "He said that that was the the correct answer.";
      foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
         Console.WriteLine("A duplicate '{0}' at position {1} is followed by
'{2}'.",
                           match.Groups["duplicateWord"].Value,
match.Groups["duplicateWord"].Index,
                           match.Groups["nextWord"].Value);
   }
}
// The example displays the following output:
//       A duplicate 'that' at position 8 is followed by 'was'.
//       A duplicate 'the' at position 22 is followed by 'correct'.
```

정규식 패턴은 다음과 같습니다.

`(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)`

다음 테이블은 정규식이 해석되는 방법을 보여 줍니다.

무늬	설명

무늬	설명
`(?<duplicateWord>\w+)`	하나 이상의 단어 문자를 찾습니다. 이 캡처링 그룹의 이름을 `duplicateWord`로 지정합니다.
`\s`	공백 문자를 찾습니다.
`\k<duplicateWord>`	`duplicateWord`라는 캡처된 그룹에서 문자열을 찾습니다.
`\W`	공백 및 문장 부호를 포함하여 비단어 문자를 찾습니다. 그러면 정규식 패턴이 캡처된 첫 번째 그룹의 단어로 시작하는 단어를 찾지 못합니다.
`(?<nextWord>\w+)`	하나 이상의 단어 문자를 찾습니다. 이 캡처링 그룹의 이름을 `nextWord`로 지정합니다.

그룹 이름은 정규식에서 반복할 수 있습니다. 예를 들어 다음 예제와 같이 둘 이상의 그룹 이름을 로 지정할 `digit`수 있습니다. 중복된 이름의 경우 Group 개체의 값은 입력 문자열에서 마지막으로 성공한 캡처에 의해 결정됩니다. 또한 그룹 이름이 중복되지 않은 경우와 마찬가지로 CaptureCollection 에 각 캡처에 대한 정보가 채워집니다.

다음 예제에서 `\D+(?<digit>\d+)\D+(?<digit>\d+)?` 정규식에는 `digit`라는 그룹 발생이 두 개 포함됩니다. 첫 번째 `digit` 그룹은 하나 이상의 숫자 문자를 캡처합니다. 두 번째 `digit` 그룹은 하나 이상 숫자 문자의 0개 또는 1개 발생을 캡처합니다. 예제의 출력에 표시된 것처럼 두 번째 캡처 그룹이 텍스트와 일치하면 해당 텍스트의 값이 Group 개체의 값을 정의합니다. 두 번째 캡처링 그룹이 입력 문자열과 일치하지 않는 경우 마지막으로 성공한 일치 값은 개체의 Group 값을 정의합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      String pattern = @"\D+(?<digit>\d+)\D+(?<digit>\d+)?";
      String[] inputs = { "abc123def456", "abc123def" };
      foreach (var input in inputs) {
         Match m = Regex.Match(input, pattern);
         if (m.Success) {
            Console.WriteLine("Match: {0}", m.Value);
            for (int grpCtr = 1; grpCtr < m.Groups.Count; grpCtr++) {
               Group grp = m.Groups[grpCtr];
               Console.WriteLine("Group {0}: {1}", grpCtr, grp.Value);
               for (int capCtr = 0; capCtr < grp.Captures.Count; capCtr++)
                  Console.WriteLine("   Capture {0}: {1}", capCtr,
                                    grp.Captures[capCtr].Value);
            }
```

```
            }
            else {
               Console.WriteLine("The match failed.");
            }
            Console.WriteLine();
         }
      }
   }
// The example displays the following output:
//       Match: abc123def456
//       Group 1: 456
//          Capture 0: 123
//          Capture 1: 456
//
//       Match: abc123def
//       Group 1: 123
//          Capture 0: 123
```

다음 테이블은 정규식이 해석되는 방법을 보여 줍니다.

무늬	설명
`\D+`	하나 이상의 10진수가 아닌 문자를 찾습니다.
`(?<digit>\d+)`	하나 이상의 10진수 문자를 찾습니다. 명명된 그룹 `digit`에 일치를 할당합니다.
`\D+`	하나 이상의 10진수가 아닌 문자를 찾습니다.
`(?<digit>\d+)?`	하나 이상 10진수 문자의 0개 또는 1개 발생을 찾습니다. 명명된 그룹 `digit` 에 일치를 할당합니다.

균형 조정 그룹 정의

균형 조정 그룹 정의는 이전에 정의된 그룹의 정의를 삭제하고 현재 그룹에 이전에 정의된 그룹과 현재 그룹 간의 간격을 저장합니다. 이 그룹화 구문의 형식은 다음과 같습니다.

`(?<name1-name2>subexpression)`

또는

`(?'name1-name2' subexpression)`

여기서 *name1* 은 현재 그룹(선택 사항), *name2* 는 이전에 정의된 그룹이며 *하위* 식은 유효한 정규식 패턴입니다. 균형 조정 그룹 정의는 *name2* 의 정의를 삭제하고 *name1* 에 *name2* 와 *name1*간의 간격을 저장합니다. *name2* 그룹이 정의되어 있지 않으면 일치에서 역추적합니다. *name2* 의 마지막 정의를 삭제하면 *name2*의 이전 정의가 표시되므로 이

구문을 통해 그룹 *name2* 에 대한 캡처 스택을 괄호 또는 여는 대괄호 및 닫는 대괄호와 같은 중첩된 구문을 추적하기 위한 카운터로 사용할 수 있습니다.

균형 조정 그룹 정의에서는 *name2* 를 스택으로 사용합니다. 각 중첩된 구문의 시작 문자는 그룹 및 해당 Group.Captures 컬렉션에 배치됩니다. 닫는 문자가 일치하면 해당 여는 문자가 그룹에서 제거되고 Captures 컬렉션이 하나 감소합니다. 모든 중첩된 구문의 여는 문자와 닫는 문자가 일치하고 나면 *name2*가 비어 있습니다.

> **참고**
>
> 다음 예제에서 중첩된 구문의 적절한 여는 문자와 닫는 문자를 사용하도록 정규식을 수정하고 나면 이러한 정규식을 사용하여 가장 많이 중첩된 구문(예: 여러 중첩 메서드 호출을 포함하는 프로그램 코드 줄 또는 수학식)을 처리할 수 있습니다.

다음 예제에서는 분산 그룹 정의를 사용하여 입력 문자열의 왼쪽 및 오른쪽 꺾쇠 괄호(<>)를 일치합니다. 이 예제에서는 일치하는 꺾쇠괄호 쌍을 추적하는 데 스택처럼 사용되는 두 개의 명명된 그룹 `Open` 및 `Close`를 정의합니다. 캡처된 각 왼쪽 꺾쇠괄호는 `Open` 그룹의 캡처 컬렉션에 푸시되고, 캡처된 각 오른쪽 꺾쇠괄호는 `Close` 그룹의 캡처 컬렉션에 푸시됩니다. 균형 조정 그룹 정의는 각 왼쪽 꺾쇠괄호에 대해 일치하는 오른쪽 꺾쇠괄호가 있도록 보장합니다. 일치하는 오른쪽 꺾쇠괄호가 없는 경우 최종 하위 패턴 `(?(Open)(?!))`는 `Open` 그룹이 비어 있지 않은 경우(및 따라서 모든 중첩된 구문이 닫히지 않은 경우)에만 평가됩니다. 최종 하위 패턴이 평가되면 찾기가 실패하는데, `(?!)` 하위 패턴이 항상 실패하는 너비가 0인 부정 lookahead 어설션이기 때문입니다.

C#

```
using System;
using System.Text.RegularExpressions;

class Example
{
   public static void Main()
   {
      string pattern = "^[^<>]*" +
                       "(" +
                       "((?'Open'<)[^<>]*)+" +
                       "((?'Close-Open'>)[^<>]*)+" +
                       ")*" +
                       "(?(Open)(?!))$";
      string input = "<abc><mno<xyz>>";

      Match m = Regex.Match(input, pattern);
      if (m.Success == true)
      {
         Console.WriteLine("Input: \"{0}\" \nMatch: \"{1}\"", input, m);
         int grpCtr = 0;
```

```
          foreach (Group grp in m.Groups)
          {
             Console.WriteLine("   Group {0}: {1}", grpCtr, grp.Value);
             grpCtr++;
             int capCtr = 0;
             foreach (Capture cap in grp.Captures)
             {
                 Console.WriteLine("      Capture {0}: {1}", capCtr,
cap.Value);
                 capCtr++;
             }
           }
       }
       else
       {
          Console.WriteLine("Match failed.");
       }
    }
}
// The example displays the following output:
//    Input: "<abc><mno<xyz>>"
//    Match: "<abc><mno<xyz>>"
//       Group 0: <abc><mno<xyz>>
//          Capture 0: <abc><mno<xyz>>
//       Group 1: <mno<xyz>>
//          Capture 0: <abc>
//          Capture 1: <mno<xyz>>
//       Group 2: <xyz
//          Capture 0: <abc
//          Capture 1: <mno
//          Capture 2: <xyz
//       Group 3: >
//          Capture 0: >
//          Capture 1: >
//          Capture 2: >
//       Group 4:
//       Group 5: mno<xyz>
//          Capture 0: abc
//          Capture 1: xyz
//          Capture 2: mno<xyz>
```

정규식 패턴은 다음과 같습니다.

```
^[^<>]*(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*(?(Open)(?!))$
```

정규식은 다음과 같이 해석됩니다.

무늬	설명
`^`	문자열의 시작 부분에서 시작합니다.
`[^<>]*`	왼쪽 또는 오른쪽 꺾쇠괄호가 아닌 0개 이상의 문자를 찾습니다.

무늬	설명
`(?'Open'<)`	왼쪽 꺾쇠괄호를 찾아 `Open`이라는 그룹에 할당합니다.
`[^<>]*`	왼쪽 또는 오른쪽 꺾쇠괄호가 아닌 0개 이상의 문자를 찾습니다.
`((?'Open'<)[^<>]*)+`	하나 이상의 왼쪽 꺾쇠괄호 다음에 왼쪽 또는 오른쪽 꺾쇠괄호가 아닌 0개 이상의 문자가 있는 일치 항목을 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`(?'Close-Open'>)`	오른쪽 꺾쇠괄호를 찾고 `Open` 그룹과 현재 그룹 간의 부분 문자열을 `Close` 그룹에 할당한 다음 `Open` 그룹의 정의를 삭제합니다.
`[^<>]*`	왼쪽 및 오른쪽 꺾쇠 괄호가 아닌 문자를 0번 이상 찾습니다.
`((?'Close-Open'>)[^<>]*)+`	하나 이상의 오른쪽 꺾쇠괄호 다음에 왼쪽 및 오른쪽 꺾쇠괄호가 아닌 0개 이상의 문자가 있는 일치 항목을 찾습니다. 오른쪽 꺾쇠괄호를 찾을 때 `Open` 그룹과 현재 그룹 간의 부분 문자열을 `Close` 그룹에 할당하고 `Open` 그룹의 정의를 삭제합니다. 이 그룹은 세 번째 캡처링 그룹입니다.
`(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*`	하나 이상의 왼쪽 꺾쇠괄호가 있고, 다음에 0개 이상의 꺾쇠괄호가 아닌 문자가 있고, 다음에 하나 이상의 오른쪽 꺾쇠괄호가 있고, 다음에 0개 이상의 꺾쇠괄호가 아닌 문자가 있는 패턴을 0개 이상 찾습니다. 오른쪽 꺾쇠괄호를 찾을 때 `Open` 그룹의 정의를 삭제하고 `Open` 그룹과 현재 그룹 간의 부분 문자열을 `Close` 그룹에 할당합니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`(?(Open)(?!))`	`Open` 그룹이 있는 경우 빈 문자열을 찾을 수 있으면 찾기를 중단하되, 문자열에서 정규식 엔진의 위치를 앞으로 이동하지 마세요. 이는 너비가 0인 부정 lookahead 어설션입니다. 입력 문자열에는 항상 암시적으로 빈 문자열이 있기 때문에 이 찾기는 항상 실패합니다. 이 찾기의 실패는 꺾쇠괄호의 짝이 맞지 않음을 의미합니다.
`$`	입력 문자열의 끝 부분을 찾습니다.

최종 하위 식 `(?(Open)(?!))`는 입력 문자열의 중첩 구문이 짝이 올바르게 맞는지 여부를 나타냅니다(예: 각 왼쪽 꺾쇠괄호가 오른쪽 꺾쇠괄호와 일치하는지 여부). 최종 하위 식에서는 유효한 캡처된 그룹을 기반으로 조건부 일치를 사용합니다. 자세한 내용은 정규식의 교체 구문을 참조하세요. `Open` 그룹이 정의되어 있는 경우 정규식 엔진은 입력 문자열에서 하위 식 `(?!)` 를 찾으려고 시도합니다. `Open` 그룹은 중첩 구문이 짝이 맞지 않는 경우에만 정의해야 합니다. 따라서 입력 문자열에서 찾을 패턴은 항상 찾기가 실패하도록 만드는 패턴이어야 합니다. 이 경우 `(?!)`는 항상 실패하는 너비가 0인 부정 lookahead 어설션인데, 입력 문자열의 다음 위치에는 항상 암시적으로 빈 문자열이 있기 때문입니다.

예제에서 정규식 엔진은 다음 표와 같이 입력 문자열 "<abc><mno<xyz>>"를 평가합니다.

단계	무늬	결과

단계	무늬	결과
1	`^`	입력 문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
2	`[^<>]*`	왼쪽 꺾쇠괄호 앞에서 꺾쇠괄호가 아닌 문자를 찾습니다. 일치 항목이 없습니다.
3	`(((?'Open'<)`	"<abc>"의 왼쪽 꺾쇠 괄호를 일치시키고 `Open` 그룹에 할당합니다.
4	`[^<>]*`	"abc"를 찾습니다.
5	`)+`	"<abc"는 캡처된 두 번째 그룹의 값입니다. 입력 문자열의 다음 문자가 왼쪽 꺾쇠괄호가 아니므로 정규식 엔진은 `(?'Open'<)[^<>]*)` 하위 패턴으로 루프백하지 않습니다.
6	`((?'Close-Open'>)`	"<abc>"의 오른쪽 꺾쇠 괄호와 일치하고, 그룹과 오른쪽 꺾쇠 괄호 `Close` 사이의 `Open` 부분 문자열인 "abc"를 그룹에 할당하고, 그룹의 현재 값(""<)`Open`을 삭제하여 비워 둡니다.
7	`[^<>]*`	오른쪽 꺾쇠괄호 뒤에서 꺾쇠괄호가 아닌 문자를 찾습니다. 일치 항목이 없습니다.
8	`)+`	캡처된 세 번째 그룹의 값은 ">"입니다. 입력 문자열의 다음 문자가 오른쪽 꺾쇠괄호가 아니므로 정규식 엔진은 `((?'Close-Open'>)[^<>]*)` 하위 패턴으로 루프백하지 않습니다.
9	`)*`	캡처된 첫 번째 그룹의 값은 "<abc>"입니다. 입력 문자열의 다음 문자가 왼쪽 꺾쇠 괄호이므로 정규식 엔진은 `(((?'Open'<)` 하위 패턴으로 루프백합니다.
10	`(((?'Open'<)`	“<mno”에서 왼쪽 꺾쇠괄호를 찾아 `Open` 그룹에 할당합니다. 이제 컬렉션 Group.Captures 에 단일 값 "<"이 있습니다.
11	`[^<>]*`	"mno"를 찾습니다.
12	`)+`	"<mno"는 캡처된 두 번째 그룹의 값입니다. 입력 문자열의 다음 문자가 왼쪽 꺾쇠괄호이므로 정규식 엔진은 `(?'Open'<)[^<>]*)` 하위 패턴으로 루프백합니다.
13	`(((?'Open'<)`	"<xyz>"의 왼쪽 꺾쇠 괄호를 일치시키고 그룹에 할당합니다 `Open` . 이제 그룹의 컬렉션에는 Group.Captures "mno"의 `Open` 왼쪽 꺾쇠 괄호와 "<<xyz>"의 왼쪽 꺾쇠 괄호라는 두 개의 캡처가 포함됩니다.
14	`[^<>]*`	"xyz"를 찾습니다.

단계	무늬	결과
15	`)+`	"<xyz"는 캡처된 두 번째 그룹의 값입니다. 입력 문자열의 다음 문자가 왼쪽 꺾쇠괄호가 아니므로 정규식 엔진은 `(?'Open'<)[^<>]*)` 하위 패턴으로 루프백하지 않습니다.
16	`((?'Close-Open'>)`	"<xyz>"의 오른쪽 꺾쇠 괄호와 일치합니다. "xyz"는 `Open` 그룹과 오른쪽 꺾쇠 괄호 간의 부분 문자열을 `Close` 그룹에 할당하고 `Open` 그룹의 현재 값을 삭제합니다. 이전 캡처의 값("<mno"의 왼쪽 꺾쇠괄호)이 `Open` 그룹의 현재 값이 됩니다. 이제 그룹의 컬렉션에는 Captures "<xyz>"의 `Open` 왼쪽 꺾쇠 괄호인 단일 캡처가 포함됩니다.
17	`[^<>]*`	꺾쇠괄호가 아닌 문자를 찾습니다. 일치 항목이 없습니다.
18	`)+`	캡처된 세 번째 그룹의 값은 ">"입니다. 입력 문자열의 다음 문자가 오른쪽 꺾쇠괄호이므로 정규식 엔진은 `((?'Close-Open'>)[^<>]*)` 하위 패턴으로 루프백합니다.
19	`((?'Close-Open'>)`	"xyz>>"의 마지막 오른쪽 꺾쇠 괄호와 일치하고 그룹에 "mno<xyz>"(그룹과 오른쪽 꺾쇠 괄호 사이의 `Open` 부분 문자열) `Close` 를 할당하고 그룹의 현재 값을 `Open` 삭제합니다. 이제 `Open` 그룹이 비어 있습니다.
20	`[^<>]*`	꺾쇠괄호가 아닌 문자를 찾습니다. 일치 항목이 없습니다.
21	`)+`	캡처된 세 번째 그룹의 값은 ">"입니다. 입력 문자열의 다음 문자가 오른쪽 꺾쇠괄호가 아니므로 정규식 엔진은 `((?'Close-Open'>)[^<>]*)` 하위 패턴으로 루프백하지 않습니다.
22	`)*`	캡처된 첫 번째 그룹의 값은 "<mno<xyz>>"입니다. 입력 문자열의 다음 문자가 왼쪽 꺾쇠괄호가 아니므로 정규식 엔진은 `(((?'Open'<)` 하위 패턴으로 루프백하지 않습니다.
23	`(?(Open)(?!))`	`Open` 그룹이 정의되어 있지 않으므로 찾으려는 시도가 이루어지지 않습니다.
24	`$`	입력 문자열의 끝 부분을 찾습니다.

비 캡처링 그룹

다음 그룹화 구문은 하위 식과 일치하는 부분 문자열을 캡처하지 않습니다.

`(?:subexpression)`

여기서 *하위 식* 은 유효한 정규식 패턴입니다. 비 캡처링 그룹 구문은 일반적으로 수량자가 그룹에 적용될 때 사용되지만 그룹에 의해 캡처된 부분 문자열에는 관심을 두지 않습니다.

> **참고**
>
> 정규식에 중첩된 그룹화 구문이 포함된 경우 외부 비 캡처링 그룹 구문은 내부 중첩된 그룹 구문에 적용되지 않습니다.

다음 예제에서는 비 캡처링 그룹을 포함하는 정규식을 보여 줍니다. 출력에는 캡처된 그룹이 포함되지 않습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?:\b(?:\w+)\W*)+\.";
      string input = "This is a short sentence.";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: {0}", match.Value);
      for (int ctr = 1; ctr < match.Groups.Count; ctr++)
         Console.WriteLine("   Group {0}: {1}", ctr,
match.Groups[ctr].Value);
   }
}
// The example displays the following output:
//       Match: This is a short sentence.
```

정규식 `(?:\b(?:\w+)\W*)+\.` 는 마침표로 종료된 문장을 찾습니다. 정규식은 개별 단어가 아닌 문장에 초점을 두므로 그룹화 구문은 단독으로 수량자로 사용됩니다. 정규식 패턴은 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(?:\w+)`	하나 이상의 단어 문자를 찾습니다. 일치하는 텍스트를 캡처된 그룹에 할당하지 않습니다.
`\W*`	0개 이상의 단어가 아닌 문자를 찾습니다.

무늬	설명
`(?:\b(?:\w+)\W*)+`	단어 경계에서 시작하는 하나 이상의 단어 문자 뒤에 0개 이상의 단어가 아닌 문자가 한 번 이상 나타나는 패턴을 찾습니다. 일치하는 텍스트를 캡처된 그룹에 할당하지 않습니다.
`\.`	마침표를 찾습니다.

그룹 옵션

다음 그룹화 구문은 하위 식 내에서 지정된 옵션을 적용하거나 사용하지 않도록 설정합니다.

`(?imnsx-imnsx:`*부분식*`)`

여기서 *하위 식* 은 유효한 정규식 패턴입니다. 예를 들어, `(?i-s:)` 는 대/소문자 구분하지 않음을 설정하고 한 줄 모드를 사용하지 않도록 설정합니다. 지정할 수 있는 인라인 옵션에 대한 자세한 내용은 정규식 옵션을 참조하세요.

> **참고**
>
> System.Text.RegularExpressions.Regex 클래스 생성자 또는 정적 메서드를 사용하여 하위 식이 아니라 전체 정규식에 적용되는 옵션을 지정할 수 있습니다. 또한 `(?imnsx-imnsx)` 언어 구문을 사용하여 정규식에서 특정 지점 뒤에 적용되는 인라인 옵션을 지정할 수 있습니다.

그룹 옵션 구문은 캡처링 그룹이 아닙니다. 즉, *subexpression* 에 의해 캡처된 문자열의 일부가 일치 항목에 포함되더라도 캡처된 그룹에 포함되지 않으며 GroupCollection 개체를 채우는 데 사용되지도 않습니다.

예를 들어, 다음 예제의 정규식 `\b(?ix: d \w+)\s`는 문자 "d"로 시작하는 모든 단어를 식별할 때 대/소문자를 구분하지 않는 일치를 사용하도록 설정하고 패턴 공백을 무시하기 위해 그룹화 구문에 인라인 옵션을 사용합니다. 정규식은 다음 테이블과 같이 정의됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(?ix: d \w+)`	이 패턴에서 대/소문자를 구분하지 않는 일치를 사용하고 공백을 무시하여 "d" 다음에 하나 이상의 단어 문자가 있는 일치 항목을 찾습니다.
`\s`	공백 문자를 찾습니다.

C#

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("'{0}// found at index {1}.", match.Value,
match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

너비가 0인 긍정 lookahead 어설션

다음 그룹화 구문은 너비가 0인 긍정 lookahead 어설션을 정의합니다.

`(?=`*부분식*`)`

여기서 *하위 식* 은 정규식 패턴입니다. 찾기가 성공하려면 일치하는 부분 문자열이 일치 결과에 포함되지 않더라도 입력 문자열이 *subexpression*의 정규식 패턴과 일치해야 합니다. 너비가 0인 긍정 lookahead 어설션은 역추적하지 않습니다.

일반적으로 너비가 0인 긍정 lookahead 어설션은 정규식 패턴의 끝 부분에서 찾을 수 있으며, 일치가 발생하도록 문자열의 끝 부분에 있어야 하지만 일치에 포함되어서는 안 되는 부분 문자열을 정의합니다. 또한 이 어설션은 과도한 역추적을 방지하는 데 유용합니다. 너비가 0인 긍정 lookahead 어설션을 사용하여, 특정 캡처된 그룹이 해당 캡처된 그룹에 대해 정의된 패턴의 하위 집합과 일치하는 텍스트로 시작하도록 할 수 있습니다. 예를 들어, 캡처링 그룹이 연속 단어 문자와 일치하는 경우 너비가 0인 긍정 lookahead 어설션을 사용하여 첫 번째 문자가 대문자 영문자가 되도록 할 수 있습니다.

다음 예제에서는 너비가 0인 긍정 lookahead 어설션을 사용하여 입력 문자열에서 동사 "is" 앞에 오는 단어를 찾습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+(?=\sis\b)";
      string[] inputs = { "The dog is a Malamute.",
                          "The island has beautiful birds.",
                          "The pitch missed home plate.",
                          "Sunday is a weekend day." };
```

```
      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine("'{0}' precedes 'is'.", match.Value);
         else
            Console.WriteLine("'{0}' does not match the pattern.", input);
      }
   }
}
// The example displays the following output:
//    'dog' precedes 'is'.
//    'The island has beautiful birds.' does not match the pattern.
//    'The pitch missed home plate.' does not match the pattern.
//    'Sunday' precedes 'is'.
```

정규식 `\b\w+(?=\sis\b)` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`(?=\sis\b)`	단어 문자 뒤에 공백 문자와 문자열 "is"(단어 경계에서 끝남)가 오는지 여부를 확인합니다. 그럴 경우 찾기가 성공합니다.

너비가 0인 부정 lookahead 어설션

다음 그룹화 구문은 너비가 0인 부정 lookahead 어설션을 정의합니다.

`(?!`*부분식*`)`

여기서 *하위 식* 은 정규식 패턴입니다. 찾기가 성공하려면 일치하는 문자열이 일치 결과에 포함되지 않더라도 입력 문자열이 *subexpression*의 정규식 패턴과 일치해서는 안 됩니다.

너비가 0인 부정 lookahead 어설션은 일반적으로 정규식의 시작 부분이나 끝 부분에 사용됩니다. 정규식의 시작 부분에서 이 어설션은 정규식의 시작 부분이 유사하지만 더 일반적인 찾으려는 패턴을 정의할 때 일치해서는 안 되는 특정 패턴을 정의할 수 있습니다. 이 경우 이 어설션은 흔히 역추적을 제한하는 데 사용됩니다. 정규식의 끝 부분에서 이 어설션은 일치의 끝 부분에서 발생할 수 없는 하위 식을 정의할 수 있습니다.

다음 예제에서는 "un"으로 시작하지 않는 단어를 찾기 위해 정규식의 시작 부분에서 너비가 0인 lookahead 어설션을 사용하는 정규식을 정의합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?!un)\w+\b";
      string input = "unite one unethical ethics use untie ultimate";
      foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       one
//       ethics
//       use
//       ultimate
```

정규식 `\b(?!un)\w+\b` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(?!un)`	다음 두 문자가 "un"인지 확인합니다. 그러지 않은 경우 일치가 가능합니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

다음 예제에서는 문장 부호 문자로 끝나지 않는 단어를 찾기 위해 정규식의 끝 부분에서 너비가 0인 lookahead 어설션을 사용하는 정규식을 정의합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+\b(?!\p{P})";
      string input = "Disconnected, disjointed thoughts in a sentence
fragment.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
```

```
   }
}
// The example displays the following output:
//       disjointed
//       thoughts
//       in
//       a
//       sentence
```

정규식 `\b\w+\b(?!\p{P})` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.
`\p{P})`	다음 문자가 문장 부호 기호(예: 마침표 또는 쉼표)가 아닌 경우 찾기가 성공합니다.

너비가 0인 긍정 lookbehind 어설션

다음 그룹화 구문은 너비가 0인 긍정 lookbehind 어설션을 정의합니다.

`(?<=`*부분식*`)`

여기서 *하위 식* 은 정규식 패턴입니다. 찾기가 성공하려면 `subexpression`이 일치 결과에 포함되지 않더라도 *subexpression*이 입력 문자열에서 현재 위치의 왼쪽에서 발생해야 합니다. 너비가 0인 긍정 lookbehind 어설션은 역추적하지 않습니다.

너비가 0인 긍정 lookbehind 어설션은 일반적으로 정규식의 시작 부분에 사용됩니다. 이 어설션이 정의하는 패턴은 이 패턴이 일치 결과에 포함되지 않더라도 일치에 대한 사전 조건입니다.

예를 들어, 다음 예제에서는 21세기 연도의 마지막 두 자리를 찾습니다(즉, 숫자 "20"이 일치하는 문자열 앞에 와야 함).

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
```

```
      string input = "2010 1999 1861 2140 2009";
      string pattern = @"(?<=\b20)\d{2}\b";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       10
//       09
```

정규식 패턴 `(?<=\b20)\d{2}\b` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\d{2}`	두 개의 10진수를 찾습니다.
`(?<=\b20)`	단어 경계에서 두 개의 10진수 앞에 10진수 "20"이 있는 경우 일치 항목 찾기를 계속합니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

너비가 0인 긍정 lookbehind 어설션은 캡처된 그룹의 마지막 문자(하나 또는 여러 개)가 해당 그룹의 정규식 패턴과 일치하는 문자의 하위 집합이어야 하는 경우 역추적을 제한하는 데도 사용됩니다. 예를 들어, 그룹이 모든 연속 단어 문자를 캡처하는 경우 너비가 0인 긍정 lookbehind 어설션을 사용하여 마지막 문자가 사전순이 되도록 할 수 있습니다.

너비가 0인 부정 lookbehind 어설션

다음 그룹화 구문은 너비가 0인 부정 lookbehind 어설션을 정의합니다.

`(?<!`*부분식*`)`

여기서 *하위 식* 은 정규식 패턴입니다. 찾기가 성공하려면 *subexpression* 이 입력 문자열에서 현재 위치의 왼쪽에서 발생해서는 안 됩니다. 그러나 `subexpression` 과 일치하지 않는 모든 부분 문자열은 일치 결과에 포함되지 않습니다.

너비가 0인 부정 lookbehind 어설션은 일반적으로 정규식의 시작 부분에 사용됩니다. 이 어설션이 정의하는 패턴은 뒤에 오는 문자열에서 일치가 불가능하도록 합니다. 또한 이 어설션은 캡처된 그룹의 마지막 문자(하나 또는 여러 개)가 해당 그룹의 정규식 패턴과 일치하는 하나 이상의 문자가 아니어야 하는 경우 역추적을 제한하는 데도 사용됩니다. 예를 들어, 그룹이 모든 연속 단어 문자를 캡처하는 경우 너비가 0인 긍정 lookbehind 어설션을 사용하여 마지막 문자가 밑줄(_)이 아니도록 할 수 있습니다.

다음 예제에서는 주말이 아닌 평일인 날짜를 찾습니다(즉, 토요일 및 일요일이 아님).

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] dates = { "Monday February 1, 2010",
                         "Wednesday February 3, 2010",
                         "Saturday February 6, 2010",
                         "Sunday February 7, 2010",
                         "Monday, February 8, 2010" };
      string pattern = @"(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b";

      foreach (string dateValue in dates)
      {
         Match match = Regex.Match(dateValue, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
      }
   }
}
// The example displays the following output:
//       February 1, 2010
//       February 3, 2010
//       February 8, 2010
```

정규식 패턴 `(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`\w+`	하나 이상의 단어 문자 다음에 공백 문자가 있는 일치 항목을 찾습니다.
`\d{1,2},`	한 개 또는 두 개의 10진수 다음에 공백 문자 하나와 쉼표 하나가 있는 일치 항목을 찾습니다.
`\d{4}\b`	네 개의 10진수를 찾고 단어 경계에서 일치 항목 찾기를 끝냅니다.
`(?<!(Saturday|Sunday) )`	일치 항목 앞에 문자열 "Saturday" 또는 "Sunday"가 아닌 문자열이 있고 해당 문자열 뒤에 공백이 하나 있으면 찾기가 성공한 것입니다.

원자성 그룹

다음 그룹화 구문은 원자성 그룹(다른 정규식 엔진에서 역추적하지 않는 하위 식, 원자성 하위 식 또는 한 번만 하위 식이라고 함)을 나타냅니다.

`(?>`*부분식*`)`

여기서 *하위 식* 은 정규식 패턴입니다.

대개 정규식에 선택적 또는 대체 일치 패턴이 포함되어 있고 찾기가 실패하는 경우 정규식 엔진이 입력 문자열을 패턴과 일치시키기 위해 여러 방향으로 분기할 수 있습니다. 첫 번째 분기를 사용하여 일치 항목을 찾을 수 없는 경우 정규식 엔진은 첫 번째 일치 항목을 사용한 지점을 백업하거나 해당 지점으로 역추적한 다음 두 번째 분기를 사용하여 찾기를 시도할 수 있습니다. 이 프로세스는 모든 분기가 시도될 때까지 계속될 수 있습니다.

이제 `(?>`*subexpression*`)` 언어 구문은 역추적을 사용하지 않도록 설정합니다. 정규식 엔진은 입력 문자열에서 가능한 한 많은 문자를 찾습니다. 일치가 더 이상 불가능한 경우 정규식 엔진은 대체 패턴 일치를 시도하기 위해 역추적하지 않습니다. (즉, 하위 식은 하위 식 단독으로 일치하는 문자열과만 일치합니다. 정규식 엔진은 하위 식을 기반으로 한 문자열 및 이러한 문자열 뒤에 나오는 하위 식은 찾으려고 시도하지 않습니다.)

이 옵션은 역추적이 실패할 것임을 아는 경우에 사용하는 것이 좋습니다. 정규식 엔진이 불필요한 검색을 수행하지 않도록 하면 성능이 향상됩니다.

다음 예제에서는 원자성 그룹이 패턴 일치 결과를 수정하는 방법을 보여 줍니다. 역추적 정규식은 단어 경계에서 뒤에 동일한 문자가 하나 더 있는 일련의 반복된 문자를 찾지만 역추적하지 않는 정규식은 그렇지 못합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "cccd.", "aaad", "aaaa" };
      string back = @"(\w)\1+.\b";
      string noback = @"(?>(\w)\1+).\b";

      foreach (string input in inputs)
      {
         Match match1 = Regex.Match(input, back);
         Match match2 = Regex.Match(input, noback);
         Console.WriteLine("{0}: ", input);

         Console.Write("   Backtracking : ");
         if (match1.Success)
            Console.WriteLine(match1.Value);
```

```
            else
               Console.WriteLine("No match");

            Console.Write("   Nonbacktracking: ");
            if (match2.Success)
               Console.WriteLine(match2.Value);
            else
               Console.WriteLine("No match");
         }
      }
   }
// The example displays the following output:
//    cccd.:
//       Backtracking : cccd
//       Nonbacktracking: cccd
//    aaad:
//       Backtracking : aaad
//       Nonbacktracking: aaad
//    aaaa:
//       Backtracking : aaaa
//       Nonbacktracking: No match
```

역추적하지 않는 정규식 `(?>(\w)\1+).\b` 는 다음 테이블과 같이 정의됩니다.

무늬	설명
`(\w)`	단일 단어 문자를 찾아 첫 번째 캡처링 그룹에 할당합니다.
`\1+`	캡처된 첫 번째 부분 문자열의 값을 한 번 이상 찾습니다.
`.`	임의의 문자를 찾습니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.
`(?>(\w)\1+)`	중복된 단어 문자를 하나 이상 일치하지만 단어 경계의 마지막 문자와 일치하도록 역추적하지 마세요.

그룹화 구문 및 정규식 개체

정규식 캡처링 그룹과 일치하는 부분 문자열은 System.Text.RegularExpressions.Group 개체로 표현됩니다. 이 개체는 System.Text.RegularExpressions.GroupCollection 속성에서 반환하는 Match.Groups 개체에서 검색할 수 있습니다. GroupCollection 개체는 다음과 같이 채워집니다.

- 컬렉션의 첫 번째 Group 개체(인덱스 0에 있는 개체)는 전체 일치를 나타냅니다.
- 다음 Group 개체 집합은 명명되지 않은(번호가 매겨진) 캡처링 그룹을 나타냅니다. 이러한 그룹은 정규식에서 정의된 순서로 왼쪽에서 오른쪽으로 나타납니다. 이러한 그룹의 인덱스 값 범위는 1에서 컬렉션에 있는 명명되지 않은 캡처링 그룹 수까지입

니다. (특정 그룹의 인덱스는 번호가 매겨진 역참조와 동일합니다. 역참조에 대한 자세한 내용은 역참조 구문을 참조하세요.)

- 최종 Group 개체 집합은 명명된 캡처링 그룹을 나타냅니다. 이러한 그룹은 정규식에서 정의된 순서로 왼쪽에서 오른쪽으로 나타납니다. 명명된 첫 번째 캡처링 그룹의 인덱스 값은 명명되지 않은 마지막 캡처링 그룹의 인덱스보다 1 큽니다. 정규식에 명명되지 않은 캡처링 그룹이 없는 경우 명명된 첫 번째 캡처링 그룹의 인덱스 값은 1입니다.

캡처링 그룹에 수량자를 적용하는 경우 해당 Group 개체의 Capture.Value, Capture.Index 및 Capture.Length 속성은 캡처링 그룹에 의해 캡처된 마지막 부분 문자열을 반영합니다. CaptureCollection 속성에서 반환하는 Group.Captures 개체의 수량자를 포함하는 그룹에 의해 캡처된 부분 문자열의 전체 집합을 검색할 수 있습니다.

다음 예제에서는 Group 개체와 Capture 개체 간의 관계를 명확하게 보여 줍니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\b(\w+)\W+)+";
      string input = "This is a short sentence.";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}'", match.Value);
      for (int ctr = 1; ctr < match.Groups.Count; ctr++)
      {
         Console.WriteLine("   Group {0}: '{1}'", ctr,
match.Groups[ctr].Value);
         int capCtr = 0;
         foreach (Capture capture in match.Groups[ctr].Captures)
         {
            Console.WriteLine("      Capture {0}: '{1}'", capCtr,
capture.Value);
            capCtr++;
         }
      }
   }
}
// The example displays the following output:
//       Match: 'This is a short sentence.'
//          Group 1: 'sentence.'
//             Capture 0: 'This '
//             Capture 1: 'is '
//             Capture 2: 'a '
//             Capture 3: 'short '
//             Capture 4: 'sentence.'
```

```
//          Group 2: 'sentence'
//             Capture 0: 'This'
//             Capture 1: 'is'
//             Capture 2: 'a'
//             Capture 3: 'short'
//             Capture 4: 'sentence'
```

정규식 패턴 `(\b(\w+)\W+)+` 는 문자열에서 개별 단어를 추출합니다. 이 패턴은 다음 표에서와 같이 정의됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(\w+)`	하나 이상의 단어 문자를 찾습니다. 이러한 문자는 함께 하나의 단어를 형성합니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`\W+`	하나 이상의 단어가 아닌 문자를 찾습니다.
`(\b(\w+)\W+)`	하나 이상의 단어 문자 다음에 하나 이상의 단어가 아닌 문자가 한 번 이상 나타나는 패턴을 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.

두 번째 캡처 그룹은 문장의 각 단어를 찾습니다. 첫 번째 캡처 그룹은 각 단어와 이 단어 뒤에 나오는 문장 부호 및 공백을 함께 찾습니다. 인덱스가 2인 Group 개체는 두 번째 캡처링 그룹과 일치하는 텍스트에 대한 정보를 제공합니다. 캡처링 그룹에 의해 캡처된 단어의 전체 집합은 CaptureCollection 속성에서 반환하는 Group.Captures 개체에서 제공합니다.

참조

- 정규식 언어 - 빠른 참조
- 역추적

정규식의 수량자

아티클 • 2025. 03. 26.

수량자는 찾을 일치 항목의 입력에 있어야 하는 문자, 그룹 또는 문자 클래스의 인스턴스 수를 지정합니다. 다음 테이블에서는 .NET에서 지원하는 수량자를 보여 줍니다.

테이블 확장

탐욕적 수량자	게으른 수량자	설명
`*`	`*?`	0번 이상 일치합니다.
`+`	`+?`	패턴이 한 번 이상 일치합니다.
`?`	`??`	0번 또는 한 번 일치합니다.
`{`*n*`}`	`{`*n*`}?`	*n*번 정확히 일치합니다.
`{`*n*`,}`	`{`*n*`,}?`	최소 *n*번 이상 일치합니다.
`{`*n*`,`*m*`}`	`{`*n*`,`*m*`}?`	*n*번에서 *m*번까지 일치합니다.

수량 `n` 및 `m`은 정수 상수입니다. 일반적으로 수량자는 탐욕적입니다. 이를 통해 정규식 엔진은 특정 패턴의 발생을 가능한 한 많이 일치시킵니다. 수량자에 `?` 문자를 추가하면 게으르게 됩니다. 따라서 정규식 엔진이 가능한 한 적은 수의 항목과 일치하게 됩니다. 탐욕적 수량자와 게으른 수량자 간의 차이에 대한 설명은 이 문서의 뒷부분에 나오는 탐욕적 및 게으른 수량자 섹션을 참조하세요.

> **중요**
>
> 정규식 패턴 `(a*)*`와 같은 중첩 수량자는 정규식 엔진이 수행해야 하는 비교 횟수를 늘릴 수 있습니다. 비교 횟수는 입력 문자열의 문자 수에 대한 지수 함수로 증가할 수 있습니다. 이 동작 및 해결 방법에 대한 자세한 내용은 **역추적**을 참조하세요.

정규식 수량자

다음 섹션에는 .NET의 정규식에서 지원하는 수량자를 보여 줍니다.

> **참고**
>
> *, +, ?, { 및 } 문자가 정규식 패턴에 나타난 경우 **문자 클래스**에 포함된 경우가 아니면 정규식 엔진은 이러한 문자를 수량자 또는 수량자 구문의 일부로 해석합니다. 이

를 문자 클래스 외부의 리터럴 문자로 해석하려면 앞에 백슬래시를 추가하여 이스케이프해야 합니다. 예를 들어 정규식 패턴의 `\*` 문자열은 리터럴 별표("*") 문자로 해석됩니다.

0번 이상 일치: *

`*` 수량자는 이전 요소를 0번 이상 일치시킵니다. 이는 `{0,}` 수량자와 동등합니다. `*`는 그 게으른 형태가 `*?`인 탐욕적 수량자입니다.

다음 예제에서는 이 정규식을 설명합니다. 입력 문자열의 9개 숫자 그룹 중 5개는 패턴과 일치하고 4개(`95`, `929`, `9219` 및 `9919`)는 패턴과 일치하지 않습니다.

C#

```
string pattern = @"\b91*9*\b";
string input = "99 95 919 929 9119 9219 999 9919 91119";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

// The example displays the following output:
//       '99' found at position 0.
//       '919' found at position 6.
//       '9119' found at position 14.
//       '999' found at position 24.
//       '91119' found at position 33.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	일치가 단어 경계에서 시작되어야 함을 지정합니다.
`91*`	`9` 뒤에 0개 이상의 `1` 문자가 오는 것과 일치합니다.
`9*`	0개 이상의 `9` 문자와 일치합니다.
`\b`	일치가 단어 경계에서 끝나야 함을 지정합니다.

1번 이상 일치: +

`+` 수량자는 이전 요소를 1번 이상 찾습니다. 이는 `{1,}`과 동등합니다. `+`는 탐욕적 수량자이며, 그 게으른 형태는 `+?`입니다.

예를 들어, 정규식 `\ban+\w*?\b`은 `a` 문자의 인스턴스가 하나 이상 뒤에 오는 `n` 문자로 시작하는 전체 단어를 찾으려고 합니다. 다음 예제에서는 이 정규식을 설명합니다. 정규식은 `an`, `annual`, `announcement` 및 `antique` 단어와 일치하며, `autumn` 및 `all`와는 정확히 일치하지 않습니다.

C#

```
string pattern = @"\ban+\w*?\b";

string input = "Autumn is a great time for an annual announcement to all
antique collectors.";
foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

// The example displays the following output:
//       'an' found at position 27.
//       'annual' found at position 30.
//       'announcement' found at position 37.
//       'antique' found at position 57.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	시작점은 단어 경계에서입니다.
`an+`	`a` 뒤에 하나 이상의 `n` 문자가 오는 것과 일치합니다.
`\w*?`	단어 문자와 0번 이상의 일치하지만 가능한 적은 수로 일치합니다.
`\b`	단어 경계에서 종료합니다.

0회 또는 1회 매칭 여부: ?

`?` 수량자는 이전 요소와 0번 또는 1번 일치시킵니다. 이는 `{0,1}`과 동등합니다. `?`는 탐욕적 수량자이며 그 게으른 대체는 `??`입니다.

예를 들어, 정규식 `\ban?\b`은 `a` 문자의 인스턴스가 0개 또는 1개 뒤에 오는 `n` 문자로 시작하는 전체 단어를 찾으려고 합니다. 즉, 단어 `a` 및 `an`를 찾으려고 합니다. 다음 예제에서는 이 정규식을 설명합니다.

C#

```
string pattern = @"\ban?\b";
string input = "An amiable animal with a large snout and an animated nose.";
foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

// The example displays the following output:
//        'An' found at position 0.
//        'a' found at position 23.
//        'an' found at position 42.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계 지점에서 시작합니다.
`an?`	`a` 뒤에 0개 또는 1개의 `n` 문자가 오는 것과 일치합니다.
`\b`	단어 경계에서 종료합니다.

정확히 n번 일치: {n}

`{ n }` 수량자는 정확하게 *n*번 이전 요소와 일치하며 여기서 *n*은 정수입니다. `{ n }`은 게으른 수량자가 `{n}?`인 탐욕적 수량자입니다.

예를 들어, 정규식 `\b\d+\,\d{3}\b`는 단어 경계, 하나 이상의 10진수 숫자, 세 개의 10진수 숫자, 그리고 단어 경계를 순서대로 찾으려고 합니다. 다음 예제에서는 이 정규식을 설명합니다.

C#

```
string pattern = @"\b\d+\,\d{3}\b";
string input = "Sales totaled 103,524 million in January, " +
                      "106,971 million in February, but only " +
                      "943 million in March.";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        '103,524' found at position 14.
//        '106,971' found at position 45.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하세요.
`\d+`	하나 이상의 10진수와 일치합니다.
`\,`	쉼표 문자를 일치시킵니다.
`\d{3}`	10진수 3자리와 일치합니다.
`\b`	단어 경계에서 끝냅니다.

적어도 n번 일치: {n,}

`{ n,}` 수량자는 적어도 *n*번 이전 요소와 일치하며 여기서 *n*은 정수입니다. `{ n,}`은 그게으른 형태가 `{n,}?`인 탐욕적 수량자입니다.

예를 들어, 정규식 `\b\d{2,}\b\D+`는 단어 경계와 그 뒤에 적어도 두 개의 숫자, 또 다른 단어 경계, 그리고 숫자가 아닌 문자가 있는 요소와 일치하려고 시도합니다. 다음 예제에서는 이 정규식을 설명합니다. 정규식은 `"7 days"`라는 구를 찾는 데 실패합니다. 하나의 10진수 숫자를 포함하지만 `"10 weeks"` 및 `"300 years"`라는 구와 일치하기 때문입니다.

C#

```
string pattern = @"\b\d{2,}\b\D+";
string input = "7 days, 10 weeks, 300 years";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        '10 weeks, ' found at position 8.
//        '300 years' found at position 18.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하십시오.
`\d{2,}`	최소 두 자리의 10진수와 일치합니다.
`\b`	단어 경계와 일치합니다.

패턴	설명
`\D+`	10진수가 아닌 숫자 하나 이상과 일치합니다.

n번에서 m번 사이에 일치: {n, m}

`{n,m}` 수량자는 이전 요소와 적어도 *n*번 일치하지만 *m*번보다 많지 않습니다. 여기서 *n* 및 *m*은 정수입니다. `{n,m}`은 탐욕적 수량자이며, 그 게으른 형태는 `{n,m}?`입니다.

다음 예제에서 정규식 `(00\s){2,4}`은 뒤에 공백이 있는 0 두 개가 2회에서 4회 발생하도록 일치시키려고 합니다. 입력 문자열의 마지막 부분에는 이 패턴이 5번이나 포함됩니다(최대 4번 아님). 그러나 공백 및 0의 다섯 번째 쌍까지의 하위 문자열의 처음 부분만이 정규식 패턴과 일치합니다.

C#

```
string pattern = @"(00\s){2,4}";
string input = "0x00 FF 00 00 18 17 FF 00 00 00 21 00 00 00 00 00";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        '00 00 ' found at position 8.
//        '00 00 00 ' found at position 23.
//        '00 00 00 00 ' found at position 35.
```

0번 이상 일치(게으른 일치 항목): *?

`*?` 수량자는 이전 요소를 0번 이상 찾지만 가능한 적은 횟수로 찾습니다. 탐욕적 수량자 `*`의 게으른 수량자입니다.

다음 예제에서 정규식 `\b\w*?oo\w*?\b`은 문자열 `oo`이 포함된 모든 단어를 찾습니다.

C#

```
 string pattern = @"\b\w*?oo\w*?\b";
 string input = "woof root root rob oof woo woe";
 foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
    Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

 //  The example displays the following output:
//        'woof' found at position 0.
//        'root' found at position 5.
//        'root' found at position 10.
```

```
//        'oof' found at position 19.
//        'woo' found at position 23.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하십시오.
`\w*?`	0개 이상의 단어 문자와 일치시키되 가능한 한 적은 문자를 포함하여 일치시킵니다.
`oo`	문자열 `oo`와 일치합니다.
`\w*?`	0개 이상의 단어 문자와 일치하지만, 가능한 한 적은 수의 문자와 일치합니다.
`\b`	단어 경계에서 끝내세요.

1번 이상 일치(게으른 일치 항목): +?

`+?` 수량자는 이전 요소를 1번 이상 찾지만 가능한 적은 횟수로 찾습니다. 이것은 탐욕적 수량자 `+`의 게으른 수량자입니다.

예를 들어, 정규식 `\b\w+?\b`은 단어 경계로 구분된 1개 이상의 문자를 찾습니다. 다음 예제에서는 이 정규식을 설명합니다.

C#

```
string pattern = @"\b\w+?\b";
string input = "Aa Bb Cc Dd Ee Ff";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        'Aa' found at position 0.
//        'Bb' found at position 3.
//        'Cc' found at position 6.
//        'Dd' found at position 9.
//        'Ee' found at position 12.
//        'Ff' found at position 15.
```

0번 또는 1번 일치(게으른 일치): ??

`??` 수량자는 이전 요소를 0번 또는 1번 찾지만 가능한 적은 횟수로 찾습니다. 탐욕적 수량자 `?`의 반대 개념인 게으른 수량자입니다.

예를 들어, 정규식 `^\s*(System.)??Console.Write(Line)??\(??`은 문자열 `Console.Write` 또는 `Console.WriteLine`과 일치시키려고 시도합니다. 문자열은 `System.` 앞에 `Console`을 포함할 수도 있고 그 뒤에 여는 괄호가 올 수도 있습니다. 문자열 앞에는 공백이 있을 수 있지만 줄의 시작 부분에 있어야 합니다. 다음 예제에서는 이 정규식을 설명합니다.

C#

```
string pattern = @"^\s*(System.)??Console.Write(Line)??\(??";
string input = "System.Console.WriteLine(\"Hello!\")\n" +
                      "Console.Write(\"Hello!\")\n" +
                      "Console.WriteLine(\"Hello!\")\n" +
                      "Console.ReadLine()\n" +
                      "   Console.WriteLine";
foreach (Match match in Regex.Matches(input, pattern,
                                      RegexOptions.IgnorePatternWhitespace |
                                      RegexOptions.IgnoreCase |
                                      RegexOptions.Multiline))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        'System.Console.Write' found at position 0.
//        'Console.Write' found at position 36.
//        'Console.Write' found at position 61.
//        '   Console.Write' found at position 110.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	입력 스트림의 시작과 일치합니다.
`\s*`	0개 이상의 공백 문자와 일치합니다.
`(System.)??`	문자열 `System.`이 0회 또는 1회 나타나는 것과 일치합니다.
`Console.Write`	문자열 `Console.Write`와 일치합니다.
`(Line)??`	문자열 `Line`이 0번 또는 1번 발생하는 경우와 일치합니다.
`\(??`	"여는 괄호의 0개 또는 1개 발생 패턴과 일치합니다."

정확히 n번 일치(게으른 일치): {n}?

`{n}?` 수량자는 정확하게 `n`번 이전 요소와 일치하며 여기서 *n*은 정수입니다. 탐욕적 수량자 `{n}`에 대응하는 게으른 수량자입니다.

다음 예제에서 정규식 `\b(\w{3,}?\.){2}?\w{3,}?\b`은 웹 사이트 주소를 식별하는 데 사용됩니다. 식은 `www.microsoft.com` 및 `msdn.microsoft.com`과 일치하지만 `mywebsite` 또는 `mycompany.com`과 일치하지 않습니다.

C#

```
string pattern = @"\b(\w{3,}?\.){2}?\w{3,}?\b";
string input = "www.microsoft.com msdn.microsoft.com mywebsite
mycompany.com";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        'www.microsoft.com' found at position 0.
//        'msdn.microsoft.com' found at position 18.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하세요.
`(\w{3,}?\.)`	3개 이상의 단어 문자와 일치하지만 가능한 한 적은 수의 문자와 점 또는 마침표 문자가 뒤따릅니다. 이 패턴은 첫 번째 캡처링 그룹입니다.
`(\w{3,}?\.){2}?`	첫 번째 그룹의 패턴과 두 번 매칭되지만 최대한 적은 횟수로 매칭됩니다.
`\b`	단어 경계에서 매치를 끝냅니다.

적어도 n번 일치(게으른 일치): {n,}?

`{n,}?` 수량자는 이전 요소와 적어도 `n`번 일치하지만, 가능한 적은 횟수만큼 일치합니다. 여기서 *n*은 정수입니다. 탐욕적 수량자 `{n,}`의 대조적 형태인 게으른 수량자입니다.

설명은 이전 섹션에서 `{n}?` 수량자에 대한 예제를 참조하세요. 해당 예제에서 정규식은 `{n,}` 수량자를 사용하여 뒤에 마침표가 있는 적어도 3개의 문자가 있는 문자열을 찾습니다.

n번에서 m번 사이에 일치(게으른 일치): {n,m}?

`{n,m}?` 수량자는 이전 요소를 `n`번에서 `m`번 사이에 찾지만 가능한 적은 횟수로 찾으며, 여기서 *n* 및 *m*은 정수입니다. 탐욕적 수량자 `{n,m}`의 게으른 상대입니다.

다음 예에서 정규식 `\b[A-Z](\w*?\s*?){1,10}[.!?]`은 1~10개의 단어가 포함된 문장과 일치합니다. 18개의 단어가 포함된 한 문장을 제외하고 입력 문자열에 있는 모든 문장을 찾습니다.

C#

```
string pattern = @"\b[A-Z](\w*?\s*?){1,10}[.!?]";
string input = "Hi. I am writing a short note. Its purpose is " +
                      "to test a regular expression that attempts to find "
+
                      "sentences with ten or fewer words. Most sentences " +
                      "in this note are short.";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine($"'{match.Value}' found at position {match.Index}.");

//  The example displays the following output:
//        'Hi.' found at position 0.
//        'I am writing a short note.' found at position 4.
//        'Most sentences in this note are short.' found at position 132.
```

정규식 패턴은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하세요.
`[A-Z]`	A부터 Z까지의 대문자와 일치합니다.
`(\w*?\s*?)`	0개 이상의 단어 문자가 있고, 뒤이어 가능한 한 적게 1개 이상의 공백 문자가 오는 경우를 찾습니다. 이 패턴은 첫 번째 캡처링 그룹입니다.
`{1,10}`	1회에서 10회 사이의 이전 패턴과 일치합니다.
`[.!?]`	문장 부호 문자 `.`, `!` 또는 `?` 중 하나와 일치합니다.

탐욕적 및 게으른 수량자

일부 수량자에는 두 가지 버전이 있습니다.

- 탐욕적 버전

  탐욕적 수량자는 대상을 최대한 많이 매칭하려고 합니다.

- 욕심 없는 (또는 게으른) 버전.

비탐욕적 수량자는 요소를 가능한 한 적게 매치하려고 합니다. `?`을 추가하여 탐욕적 수량자를 게으른 수량자로 바꿀 수 있습니다.

신용 카드 번호와 같은 숫자 문자열에서 마지막 4자리를 추출하기 위한 정규식을 고려해 보세요. `*` 탐욕적 수량자를 사용하는 정규식의 버전은 `\b.*([0-9]{4})\b`입니다. 그러나 문자열이 두 개의 숫자를 포함하는 경우 다음 예제와 같이 정규식은 두 번째 숫자의 마지막 4자리가 일치합니다.

C#

```
string greedyPattern = @"\b.*([0-9]{4})\b";
string input1 = "1112223333 3992991999";
foreach (Match match in Regex.Matches(input1, greedyPattern))
   Console.WriteLine($"Account ending in ******{match.Groups[1].Value}.");

// The example displays the following output:
//       Account ending in ******1999.
```

정규 표현식은 첫 번째 숫자와 일치하지 않는 이유는 `*` 수량자가 전체 문자열에서 이전 요소와 최대한 많이 일치시키려고 시도하여 결과적으로 문자열의 끝부분에서 일치하는 항목을 찾기 때문입니다.

이 동작은 원하는 동작이 아닙니다. 대신 다음 예제와 같이 `*?` 게으른 수량자를 사용하여 두 번호에서 숫자를 추출할 수 있습니다.

C#

```
string lazyPattern = @"\b.*?([0-9]{4})\b";
string input2 = "1112223333 3992991999";
foreach (Match match in Regex.Matches(input2, lazyPattern))
   Console.WriteLine($"Account ending in ******{match.Groups[1].Value}.");

// The example displays the following output:
//       Account ending in ******3333.
//       Account ending in ******1999.
```

대부분의 경우 탐욕적 및 게으른 수량자를 포함하는 정규식은 동일한 일치 항목을 반환합니다. 와일드카드(`.`) 메타 문자를 사용하면, 이들은 대개 서로 다른 결과를 반환합니다.

수량자 및 빈 일치 항목

`*`, `+` 및 `{n,m}` 수량자와 해당 게으른 수량자는 최소 캡처 수가 발견되면 빈 일치 후에 절대 반복되지 않습니다. 가능한 그룹 캡처의 최대 수가 무한 또는 거의 무한인 경우 이 규칙은 수량자가 빈 하위 식을 찾기 위해 무한 루프를 입력하지 않도록 방지합니다.

예를 들어 다음 코드는 0개 또는 1개의 Regex.Match 문자를 0번 이상 찾는 정규식 패턴 `(a?)*`를 포함한 `a` 메서드에 대한 호출의 결과를 표시합니다. 단일 캡처 그룹은 각 `a` 및 String.Empty를 캡처하지만 첫 번째 빈 일치로 인해 수량자가 반복을 중지하므로 두 번째 빈 일치가 없습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(a?)*";
      string input = "aaabbb";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine($"Match: '{match.Value}' at index {match.Index}");
      if (match.Groups.Count > 1) {
         GroupCollection groups = match.Groups;
         for (int grpCtr = 1; grpCtr <= groups.Count - 1; grpCtr++) {
            Console.WriteLine($"   Group {grpCtr}: '{groups[grpCtr].Value}'
at index {groups[grpCtr].Index}");
            int captureCtr = 0;
            foreach (Capture capture in groups[grpCtr].Captures) {
               captureCtr++;
               Console.WriteLine($"      Capture {captureCtr}:
'{capture.Value}' at index {capture.Index}");
            }
         }
      }
   }
}
// The example displays the following output:
//       Match: 'aaa' at index 0
//          Group 1: '' at index 3
//             Capture 1: 'a' at index 0
//             Capture 2: 'a' at index 1
//             Capture 3: 'a' at index 2
//             Capture 4: '' at index 3
```

캡처의 최소 및 최대 수를 정의하는 캡처링 그룹과 캡처의 고정 수를 정의하는 그룹 사이의 실질적인 차이를 확인하려면 정규식 패턴 `(a\1|(?(1)\1)){0,2}` 및 `(a\1|(?(1)\1)){2}`를 사용하는 것이 좋습니다. 두 정규식 모두 다음 표에 정의된 단일 캡처 그룹으로 구성됩니다.

테이블 확장

패턴	설명
`(a\1`	첫 번째 캡처된 그룹의 값과 함께 `a`와 일치합니다.
`|(?(1)`	... 또는 첫 번째 캡처된 그룹이 정의되었는지 테스트합니다. `(?(1)` 구문은 캡처링 그룹을 정의하지 않습니다.
`\1))`	첫 번째 캡처된 그룹이 있는 경우 해당 값을 찾습니다. 그룹이 존재하지 않으면 그룹은 String.Empty와 일치합니다.

첫 번째 정규식이 0번과 두 번 사이에 이 패턴을 찾으려 합니다. 두 번째 정규식은 정확히 두 번입니다. 첫 번째 패턴은 String.Empty의 첫 번째 캡처로 최소 캡처 수에 도달하므로 `a\1`과 일치시키려고 반복하지 않습니다. `{0,2}` 수량자는 마지막 반복에서 빈 일치 항목만 허용합니다. 대조적으로, 두 번째 정규식은 `a`을 두 번째로 평가하므로 `a\1`와 일치합니다. 최소 반복 횟수인 2는 빈 일치 후 엔진이 반복하도록 강제합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern, input;

      pattern = @"(a\1|(?(1)\1)){0,2}";
      input = "aaabbb";

      Console.WriteLine($"Regex pattern: {pattern}");
      Match match = Regex.Match(input, pattern);
      Console.WriteLine($"Match: '{match.Value}' at position
{match.Index}.");
      if (match.Groups.Count > 1) {
         for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1;
groupCtr++)
         {
            Group group = match.Groups[groupCtr];
            Console.WriteLine($"   Group: {groupCtr}: '{group.Value}' at
position {group.Index}.");
            int captureCtr = 0;
            foreach (Capture capture in group.Captures) {
               captureCtr++;
               Console.WriteLine($"      Capture: {captureCtr}:
'{capture.Value}' at position {capture.Index}.");
            }
         }
      }
      Console.WriteLine();
```

```
      pattern = @"(a\1|(?(1)\1)){2}";
      Console.WriteLine($"Regex pattern: {pattern}");
      match = Regex.Match(input, pattern);
         Console.WriteLine($"Matched '{match.Value}' at position
{match.Index}.");
      if (match.Groups.Count > 1) {
         for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1;
groupCtr++)
         {
            Group group = match.Groups[groupCtr];
            Console.WriteLine($"   Group: {groupCtr}: '{group.Value}' at
position {group.Index}.");
            int captureCtr = 0;
            foreach (Capture capture in group.Captures) {
               captureCtr++;
               Console.WriteLine($"      Capture: {captureCtr}:
'{capture.Value}' at position {capture.Index}.");
            }
         }
      }
   }
}
// The example displays the following output:
//       Regex pattern: (a\1|(?(1)\1)){0,2}
//       Match: '' at position 0.
//          Group: 1: '' at position 0.
//             Capture: 1: '' at position 0.
//
//       Regex pattern: (a\1|(?(1)\1)){2}
//       Matched 'a' at position 0.
//          Group: 1: 'a' at position 0.
//             Capture: 1: '' at position 0.
//             Capture: 2: 'a' at position 0.
```

참고 항목

- 정규식 언어 - 빠른 참조
- 백트래킹

정규 표현식에서의 역참조 문법

아티클 • 2025. 03. 26.

역참조는 문자열 내에서 반복된 문자 또는 부분 문자열을 식별하는 편리한 방법을 제공합니다. 예를 들어 입력 문자열에 임의의 부분 문자열이 여러 번 포함되어 있으면 첫 번째 발생을 캡처링 그룹과 일치시킨 다음 역참조를 사용하여 부분 문자열의 후속 발생을 일치시킬 수 있습니다.

> ⓘ **참고**
>
> 대체 문자열에서 명명된 캡처링 그룹과 번호가 매겨진 캡처링 그룹을 참조하기 위해 별도의 구문이 사용됩니다. 자세한 내용은 **대체**를 참조하세요.

.NET에서는 번호가 매겨진 캡처링 그룹과 명명된 캡처링 그룹을 참조하기 위해 별도의 언어 요소를 정의합니다. 캡처링 그룹에 대한 자세한 내용은 그룹화 구문을 참조하세요.

번호가 매겨진 역참조

번호가 매겨진 역참조에는 다음 구문이 사용됩니다.

`\` *번호*

여기서 *number*는 정규식에서 캡처링 그룹의 서수 위치입니다. 예를 들어 `\4`는 네 번째 캡처링 그룹의 내용과 일치합니다. *number*가 정규식 패턴에서 정의되지 않은 경우 구문 분석 오류가 발생하고 정규식 엔진이 ArgumentException을 throw합니다. 예를 들어 `\b(\w+)\s\1` 정규식은 `(\w+)`가 식의 첫 번째이자 유일한 캡처링 그룹이기 때문에 유효합니다. 반면에 `\b(\w+)\s\2`는 `\2`로 번호가 매겨진 캡처링 그룹이 없기 때문에 유효하지 않으며 인수 예외를 throw합니다. 또한 *번호*가 특정 서수 위치에서 캡처링 그룹을 식별하지만 해당 캡처링 그룹이 서수 위치가 아닌 다른 숫자 이름을 할당받은 경우 정규식 파서는 ArgumentException을 throw합니다.

8진수 이스케이프 코드(예: `\16`)와 동일한 표기법을 사용하는 `\`*number* 역참조 간의 모호성에 유의하세요. 이러한 모호성은 다음과 같이 해결됩니다.

- `\1`에서 `\9` 사이의 식은 항상 8진수 코드가 아니라 역참조로 해석됩니다.
- 다중 숫자 식의 첫 번째 숫자가 8 또는 9이면(예: `\80` 또는 `\91`) 식은 리터럴로 해석됩니다.
- `\10` 이상인 식은 이 숫자에 해당하는 역참조가 있을 경우 역참조로 간주되고, 없을 경우 8진수 코드로 해석됩니다.

- 정규식에 정의되지 않은 그룹 번호에 대한 역참조가 포함되어 있으면 구문 분석 오류가 발생하고 정규식 엔진이 ArgumentException을 throw합니다.

모호성 문제가 발생하는 경우 명확하고 8진수 문자 코드와 혼동되지 않는 `\k<`*name*`>` 표기법을 사용할 수 있습니다. 마찬가지로, `\xdd` 등의 16진수 코드는 명확하며 역참조와 혼동되지 않습니다.

다음 예제에서는 문자열에서 중복된 단어를 찾습니다. 다음과 같은 요소로 구성된 `(\w)\1` 정규식을 정의합니다.

테이블 확장

요소	설명
`(\w)`	단어 문자를 찾아 첫 번째 캡처링 그룹에 할당합니다.
`\1`	첫 번째 캡처링 그룹의 값과 일치하는 다음 문자를 찾습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w)\1";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"Found '{match.Value}' at position
{match.Index}.");
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

명명된 역참조

명명된 역참조는 다음 구문을 사용하여 정의됩니다.

`\k<` *name* `>`

또는:

`\k'` *name* `'`

여기서 *name*은 정규식 패턴에 정의된 캡처링 그룹의 이름입니다. *name*이 정규식 패턴에서 정의되지 않은 경우 구문 분석 오류가 발생하고 정규식 엔진이 ArgumentException을 throw합니다.

다음 예제에서는 문자열에서 중복된 단어 문자를 찾습니다. 다음과 같은 요소로 구성된 `(?<char>\w)\k<char>` 정규식을 정의합니다.

테이블 확장

요소	설명
`(?<char>\w)`	단어 문자를 찾아 이름이 `char`인 캡처링 그룹에 할당합니다.
`\k<char>`	`char` 캡처링 그룹의 값과 일치하는 다음 문자를 찾습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<char>\w)\k<char>";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"Found '{match.Value}' at position
{match.Index}.");
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

숫자로 명명된 역참조

`\k`를 사용하는 명명된 역참조에서 *이름*은 숫자의 문자열 표현일 수도 있습니다. 예를 들어 다음 예제에서는 `(?<2>\w)\k<2>` 정규식을 사용하여 문자열에서 2배 워드 문자를 찾습

니다. 이 경우에 예제에서는 명시적으로 "2"라고 명명된 캡처링 그룹을 정의하고 따라서 역참조는 "2"라고 명명됩니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<2>\w)\k<2>";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"Found '{match.Value}' at position
{match.Index}.");
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

*이름*이 숫자의 문자열 표현이고 해당 이름을 가 진 캡처링 그룹이 없는 경우 `\k<` *이름* `>`은 역참조 `\` *숫자*와 같습니다. 여기서 *번호*는 캡처의 서수 위치입니다. 다음 예제에는 `char` 라는 단일 캡처링 그룹이 있습니다. 역참조 구문은 `\k<1>`로 참조합니다. 예제의 출력이 표시한 대로 Regex.IsMatch가 첫 번째 캡처링 그룹이기 때문에 `char`에 대한 호출이 성공합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(Regex.IsMatch("aa", @"(?<char>\w)\k<1>"));
      // Displays "True".
   }
}
```

그러나 경우 *이름*이 숫자의 문자열 표현이고 해당 위치에 있는 캡처링 그룹이 명시적으로 숫자 이름을 할당받지 않은 경우 정규식 파서는 서수 위치를 기준으로 캡처링 그룹을

식별할 수 없습니다. 대신 ArgumentException을 throw합니다. 다음 예제의 캡처링 그룹만이 "2"라고 명명됩니다. `\k` 구문을 사용하여 "1"이라는 역참조를 정의하기 때문에 정규식 파서는 첫 번째 캡처링 그룹을 식별할 수 없고 예외를 throw합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(Regex.IsMatch("aa", @"(?<2>\w)\k<1>"));
      // Throws an ArgumentException.
   }
}
```

역참조가 매칭하는 것

역참조는 그룹의 가장 최근 정의(왼쪽에서 오른쪽으로 찾을 경우 가장 왼쪽에 있는 정의)를 가리킵니다. 그룹에서 여러 개의 캡처를 만드는 경우 역참조는 가장 최근 캡처를 가리킵니다.

다음 예제에는 이름이 \1인 그룹을 다시 정의하는 정규식 패턴 `(?<1>a)(?<1>\1b)*`가 포함되어 있습니다. 다음 표에서는 정규식의 각 패턴에 대해 설명합니다.

테이블 확장

패턴	설명
`(?<1>a)`	문자 "a"를 찾은 다음, 이름이 `1`인 캡처링 그룹에 결과를 할당합니다.
`(?<1>\1b)*`	이름이 `1`과 "b"로 지정된 그룹을 0번 이상 일치시키고, 이름이 `1`로 지정된 캡처링 그룹에 결과를 할당합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<1>a)(?<1>\1b)*";
      string input = "aababb";
```

```
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("Match: " + match.Value);
         foreach (Group group in match.Groups)
            Console.WriteLine("   Group: " + group.Value);
      }
   }
}
// The example displays the following output:
//          Group: aababb
//          Group: abb
```

입력 문자열("aababb")과 정규식을 비교할 때 정규식 엔진은 다음 작업을 수행합니다.

1. 문자열의 시작 부분에서 시작하여 `(?<1>a)` 표현과 "a"가 성공적으로 일치합니다. `1` 그룹의 값은 이제 “a”입니다.

2. 두 번째 문자로 진행하고, `\1b` 또는 'ab' 식과 'ab' 문자열을 성공적으로 일치시킵니다. 그런 다음 결과 "ab"를 `\1`에 할당합니다.

3. 네 번째 문자로 이동합니다. `(?<1>\1b)*` 식은 0번 이상 일치할 수 있으므로 `\1b` 식과 "abb" 문자열이 성공적으로 일치합니다. 결과 "abb"를 `\1`에 다시 할당합니다.

이 예제에서 `*`는 반복 수량자로, 정규식 엔진이 정의되는 패턴을 찾을 수 없을 때까지 반복해서 평가됩니다. 반복 수량자는 그룹 정의를 지우지 않습니다.

그룹에서 부분 문자열을 캡처하지 않은 경우 해당 그룹에 대한 역참조가 정의되지 않으며 일치되지 않습니다. 이 내용은 다음과 같이 정의된 정규식 패턴 `\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b`를 통해 확인할 수 있습니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(\p{Lu}{2})`	대문자 두 개를 일치시킵니다. 이 그룹은 첫 번째 캡처 그룹입니다.
`(\d{2})?`	두 개의 10진 숫자가 0번 또는 1번 나타나는 패턴과 일치시킵니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`(\p{Lu}{2})`	두 개의 대문자를 맞춥니다. 이 그룹은 세 번째 캡처링 그룹입니다.
`\b`	단어 경계에서 매칭을 종료합니다.

두 번째 캡처링 그룹에서 정의된 두 개의 10진수가 없는 경우에도 입력 문자열이 이 정규식과 일치할 수 있습니다. 다음 예제에서는 일치가 성공해도 성공적인 두 캡처링 그룹 사이에 빈 캡처링 그룹이 있음을 보여 줍니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b";
      string[] inputs = { "AA22ZZ", "AABB" };
      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
         {
            Console.WriteLine($"Match in {input}: {match.Value}");
            if (match.Groups.Count > 1)
            {
               for (int ctr = 1; ctr <= match.Groups.Count - 1; ctr++)
               {
                  if (match.Groups[ctr].Success)
                     Console.WriteLine($"Group {ctr}:
{match.Groups[ctr].Value}");
                  else
                     Console.WriteLine($"Group {ctr}: <no match>");
               }
            }
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match in AA22ZZ: AA22ZZ
//       Group 1: AA
//       Group 2: 22
//       Group 3: ZZ
//
//       Match in AABB: AABB
//       Group 1: AA
//       Group 2: <no match>
//       Group 3: BB
```

참고 항목

- 정규식 언어 - 빠른 참조

정규식의 교체 구문

아티클 • 2025. 04. 03.

교체 구문은 either/or 또는 조건부 일치를 허용하도록 정규식을 수정합니다. .NET에서는 다음 세 가지 교체 구문을 지원합니다.

- |를 사용한 패턴 일치
- (?(expression)yes|no)를 사용한 조건부 일치
- 유효한 캡처 그룹을 기준으로 조건부 일치

|을 이용한 패턴 매칭

세로 막대(|) 문자를 사용하여 일련의 패턴 중 하나를 찾을 수 있습니다. 여기서 | 문자는 각 패턴을 구분합니다.

긍정 문자 클래스처럼 | 문자는 여러 개의 단일 문자 중 하나와 매치할 수 있습니다. 다음 예제에서는 정수 문자 클래스와 둘 중 하나의 패턴 매칭을 | 문자를 사용하여 문자열에서 "gray" 또는 "grey"라는 단어를 찾습니다. 이 경우 | 문자는 더 자세한 정규식을 생성합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Regular expression using character class.
      string pattern1 = @"\bgr[ae]y\b";
      // Regular expression using either/or.
      string pattern2 = @"\bgr(a|e)y\b";

      string input = "The gray wolf blended in among the grey rocks.";
      foreach (Match match in Regex.Matches(input, pattern1))
         Console.WriteLine($"'{match.Value}' found at position
{match.Index}");
      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern2))
         Console.WriteLine($"'{match.Value}' found at position
{match.Index}");
   }
}
// The example displays the following output:
//       'gray' found at position 4
//       'grey' found at position 35
```

```
//
//       'gray' found at position 4
//       'grey' found at position 35
```

`|` 문자를 사용하는 정규식인 `\bgr(a|e)y\b`는 다음 테이블과 같이 해석됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작합니다.
`gr`	문자 "gr"과 일치시킵니다.
`(a|e)`	"a" 또는 "e" 중 하나를 선택합니다.
`y\b`	단어 경계에서 "y"를 찾습니다.

`|` 문자를 사용하여 문자 리터럴과 정규식 언어 요소의 조합을 포함할 수 있는 여러 문자나 하위 식을 통해 either/or 일치 항목을 찾을 수도 있습니다. 문자 클래스는 이 기능을 제공하지 않습니다. 다음 예제에서는 `|` 문자를 사용하여 미국 SSN(사회 보장 번호)(*ddd-dd-dddd* 형식의 9자리 숫자) 또는 미국 EIN(고용주 식별 번호)(*dd-ddddddd* 형식의 9자리 숫자)을 추출합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine($"Matches for {pattern}:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"   {match.Value} at position {match.Index}");
   }
}
// The example displays the following output:
//       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

정규식 `\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b`는 다음 테이블과 같이 해석됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작합니다.
`(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})`	10진수 2개, 하이픈, 10진수 7개 순의 일치 항목이나 10진수 3개, 하이픈, 10진수 2개, 또 다른 하이픈, 10진수 4개 순의 일치 항목을 찾습니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

식을 사용한 조건부 일치

이 언어 요소는 초기 패턴을 일치시킬 수 있는지 여부에 따라 두 가지 패턴 중 하나에 일치하려고 합니다. 사용되는 구문은 다음과 같습니다.

`(?(` *expression* `)` *yes* `)`

또는

`(?(` *expression* `)` *yes* `|` *no* `)`

여기서 *expression*은 일치 항목을 찾을 초기 패턴이고, *yes*는 *expression*과 일치하는 경우 일치 항목을 찾을 패턴이고, *no*는 *expression*이 일치하지 않을 경우 일치 항목을 찾을 선택적 패턴입니다(*no* 패턴이 제공되지 않으면 빈 *no*와 동일함). 정규식 엔진은 *expression*을 너비가 0인 어설션으로 처리합니다. 즉, 정규식 엔진은 *expression*을 계산한 후 입력 스트림에서 앞으로 이동하지 않습니다. 따라서 이 구문은 다음 구문과 같습니다.

`(?(?=` *expression* `)` *yes* `|` *no* `)`

여기서 `(?=`*expression*`)`은 너비가 0인 어설션 구문입니다. 자세한 내용은 그룹화 구문을 참조하세요. 정규식 엔진이 *expression*을 앵커(너비가 0인 어설션)로 해석하기 때문에 *expression*은 너비가 0인 어설션(자세한 내용은 앵커 참조) 또는 *yes*에도 포함되는 하위 식이어야 합니다. 그렇지 않으면 *yes* 패턴을 찾을 수 없습니다.

> **참고**
>
> *expression*이 이름이나 숫자가 지정된 캡처링 그룹인 경우 교체 구문은 캡처 테스트로 해석됩니다. 자세한 내용은 다음 섹션인 **유효한 캡처 그룹을 기준으로 조건부 일치**를 참조하세요. 즉, 정규식 엔진은 캡처된 하위 문자열을 찾으려고 하지 않지만, 대신 그룹의 존재 여부를 테스트합니다.

다음 예제는 |를 사용한 Either/Or 패턴 일치 섹션에 나타나는 예제의 변형입니다. 조건부 일치를 사용하여 단어 경계 뒤의 처음 문자 3개가 숫자 2개, 하이픈 순의 문자열인지를 확인합니다. 해당 문자열이라면 미국 고용주 식별 번호(EIN)와 일치시키려고 시도합니다. 그렇지 않으면 미국 사회 보장 번호(SSN)와 일치하는지를 확인하려고 시도합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine($"Matches for {pattern}:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"   {match.Value} at position {match.Index}");
   }
}
// The example displays the following output:
//       Matches for \b(?(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

정규식 패턴 `\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b`는 다음 테이블과 같이 해석됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작합니다.
`(?(\d{2}-)`	다음 문자 3개가 숫자 2개, 하이픈 순으로 구성되는지를 확인합니다.
`\d{2}-\d{7}`	이전 패턴이 일치하면 숫자 2개, 하이픈, 숫자 7개 순의 일치 항목을 찾습니다.
`\d{3}-\d{2}-\d{4}`	이전 패턴이 일치하지 않으면 10진수 3개, 하이픈, 10진수 2개, 또 다른 하이픈, 10진수 4개 순의 일치 항목을 찾습니다.
`\b`	단어 경계를 찾습니다.

유효한 캡처 그룹을 기준으로 조건부 일치

이 언어 요소는 지정된 캡처링 그룹에 일치시켰는지 여부에 따라 두 패턴 중 하나에 일치시키려고 시도합니다. 사용되는 구문은 다음과 같습니다.

`(?(` *name* `)` *yes* `)`

또는

`(?(` *name* `)` *yes* `|` *no* `)`

또는

`(?(` *number* `)` *yes* `)`

또는

`(?(` *number* `)` *yes* `|` *no* `)`

여기서 *name*은 이름이고, *number*는 캡처 그룹 수이고, *yes*는 *name* 또는 *number*에 일치 항목이 있을 경우 일치 항목을 찾을 식이고, *no*는 일치 항목이 없을 경우 일치 항목을 찾을 선택적 식입니다(*no* 패턴이 제공되지 않으면 빈 *no*와 동일함).

name 이 정규식 패턴에서 사용되는 캡처 그룹의 이름에 해당하지 않을 경우에는 교체 구문이 앞 섹션에서 설명한 대로 식 테스트로 해석됩니다. 일반적으로 이는 *expression*이 `false`로 계산됨을 의미합니다. *number* 가 정규식 패턴에 사용되는 번호 매기기 캡처 그룹에 해당하지 않는 경우 정규식 엔진이 ArgumentException을 throw합니다.

다음 예제는 |를 사용한 Either/Or 패턴 일치 섹션에 나타나는 예제의 변형입니다. 숫자 2개, 하이픈 순으로 구성된 `n2` 라는 캡처 그룹을 사용합니다. 교체 구문은 입력 문자열에서 이 캡처 그룹이 일치했는지를 테스트합니다. 일치한다면, 대체 구문은 9자리 미국 고용주 식별 번호(EIN)의 마지막 일곱 자리가 일치하는지 확인하려고 합니다. 일치하지 않는다면 9자리 미국 SSN(사회 보장 번호)을 찾으려고 합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine($"Matches for {pattern}:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"   {match.Value} at position {match.Index}");
   }
}
```

```
// The example displays the following output:
//       Matches for \b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

정규식 패턴 `\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b`는 다음 테이블과 같이 해석됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하십시오.
`(?<n2>\d{2}-)?`	두 자리 숫자가 하이픈 뒤에 나오는 경우를 0번 또는 1번 찾습니다. 이 캡처링 그룹의 이름을 `n2`로 지정합니다.
`(?(n2)`	`n2` 가 입력 문자열에서 일치하는지 여부를 테스트합니다.
`\d{7}`	`n2` 가 일치하는 경우 일곱 개의 10진수를 일치시킵니다.
`|\d{3}-\d{2}-\d{4}`	`n2` 가 일치하지 않는 경우 세 10진수, 하이픈, 두 10진수, 다른 하이픈 및 네 10진수를 일치시킵니다.
`\b`	단어 경계를 찾습니다.

다음 예제에서는 명명된 그룹 대신 번호가 있는 그룹을 사용하는 이 예제의 변형을 보여줍니다. 정규식 패턴은 `\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b`입니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine($"Matches for {pattern}:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"   {match.Value} at position {match.Index}");
   }
}
// The example display the following output:
//       Matches for \b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

참조

- 정규식 언어 - 빠른 참조

정규식의 대체

2025. 06. 17.

대체는 대체 패턴 내에서만 인식되는 언어 요소입니다. 정규식 패턴을 사용하여 입력 문자열에서 일치하는 텍스트를 대체할 텍스트의 전체 또는 일부를 정의합니다. 대체 패턴은 리터럴 문자와 함께 하나 이상의 대체로 구성됩니다. Regex.Replace 매개 변수가 있는 `replacement` 메서드의 오버로드와 Match.Result 메서드에 대체 패턴이 제공됩니다. 메서드는 일치하는 패턴을 매개 변수로 정의된 `replacement` 패턴으로 바꿉다.

.NET은 다음 표에 나열된 대체 요소를 정의합니다.

테이블 확장

대체	설명
$ *번호*	*숫자*로 식별된 캡처링 그룹의 마지막 부분 문자열을 대체 문자열에 포함합니다. 여기서 *숫자*는 십진수 값입니다. 자세한 내용은 번호 매기기된 그룹의 대체를 참조하세요.
${ *name* }	대체 문자열에서 `(?<`*이름*`>)`으로 지정된 명명된 그룹과 일치하는 마지막 부분 문자열을 포함합니다. 자세한 내용은 명명된 그룹 대체를 참조하세요.
$$	대체 문자열에 단일 "$" 리터럴을 포함합니다. 자세한 내용은 "$" 기호 대체를 참조하세요.
$&	전체 일치 항목의 복사본을 대체 문자열에 포함합니다. 자세한 내용은 전체 일치 항목 대체를 참조하세요.
$`	대체 문자열에서 일치하기 전에 입력 문자열의 모든 텍스트를 포함합니다. 자세한 내용은 일치되기 전에 텍스트를 대체하기 참조하세요.
$'	대체 문자열에서 일치 후 입력 문자열의 모든 텍스트를 포함합니다. 자세한 내용은 일치 후 텍스트 대체를 참조하세요.
$+	대체 문자열에 캡처된 마지막 그룹을 포함합니다. 자세한 내용은 마지막 캡처 그룹의 대체를 참고하십시오.
$_	대체 문자열에 전체 입력 문자열을 포함합니다. 자세한 내용은 전체 입력 문자열 대체를 참조하세요.

대체 요소 및 대체 패턴

대체는 대체 패턴에서 인식되는 유일한 특수 구문입니다. 문자 이스케이프 및 문자와 일치하는 마침표(`.`)를 포함한 다른 정규식 언어 요소는 지원되지 않습니다. 마찬가지로 대체 언어 요소는 대체 패턴에서만 인식되며 정규식 패턴에서는 유효하지 않습니다.

정규식 패턴 또는 대체 표현에서 나타날 수 있는 유일한 문자는 `$` 문자이며, 각 맥락에서 다른 의미로 사용됩니다. 정규식 패턴 `$` 에서 문자열의 끝과 일치하는 앵커입니다. 대체 패턴 `$` 에서 대체의 시작을 나타냅니다.

> ⓘ **참고**
>
> 정규식 내의 대체 패턴과 유사한 기능의 경우 역참조를 사용합니다. 역참조에 대한 자세한 내용은 **역참조 구문을 참조하세요**.

번호가 매겨진 그룹 교체

`$` *숫자* 언어 요소에는 대체 문자열의 *숫자* 캡처링 그룹과 일치하는 마지막 부분 문자열이 포함됩니다. 여기서 *숫자는* 캡처링 그룹의 인덱스입니다. 예를 들어 대체 패턴 `$1` 은 일치하는 부분 문자열을 캡처된 첫 번째 그룹으로 대체해야 임을 나타냅니다. 번호가 매겨진 캡처링 그룹에 대한 자세한 내용은 그룹화 구문을 참조하세요.

뒤에 있는 `$` 모든 숫자는 *숫자* 그룹에 속하는 것으로 해석됩니다. 사용자의 의도가 아닌 경우 명명된 그룹을 대신 대체할 수 있습니다. 예를 들어 대체 문자열 `${1}1` 을 "1"과 함께 캡처된 첫 번째 그룹의 값으로 정의하는 대신 `$11` 대체 문자열을 사용할 수 있습니다. 자세한 내용은 명명된 그룹 대체를 참조하세요.

`(?<` 구문을 사용하여 `>)` 명시적으로 할당되지 않은 그룹 캡처는 1부터 왼쪽에서 오른쪽으로 번호가 매겨집니다. 명명된 그룹은 마지막 이름 없는 그룹의 인덱스보다 큰 그룹에서 시작하여 왼쪽에서 오른쪽으로 번호가 매겨집니다. 예를 들어 정규식 `(\w)(?<digit>\d)`에서 명명된 그룹의 인덱 `digit` 스는 2입니다.

숫자가 정규식 패턴 `$` 에 정의된 유효한 캡처링 그룹을 지정하지 않으면 *숫자는* 각 일치 항목을 대체하는 데 사용되는 리터럴 문자 시퀀스로 해석됩니다.

다음 예제에서는 `$` 대체를 사용하여 10진수 값에서 통화 기호를 제거합니다. 통화 값의 시작 또는 끝에 있는 통화 기호를 제거하고 가장 일반적인 두 개의 소수 구분 기호("." 및 ",")를 인식합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*";
```

```
        string replacement = "$1";
        string input = "$16.32 12.19 £16.29 €18.29  €18,29";
        string result = Regex.Replace(input, pattern, replacement);
        Console.WriteLine(result);
    }
}
// The example displays the following output:
//       16.32 12.19 16.29 18.29  18,29
```

정규식 패턴 `\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*`는 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\p{Sc}*`	통화 기호 문자가 0개 이상일 경우 일치합니다.
`\s?`	공백 문자가 0개 또는 1개 있는지 일치시킵니다.
`\d+`	하나 이상의 10진수 숫자가 일치하는지 확인합니다.
`[.,]?`	0개 또는 1개의 마침표 또는 쉼표와 일치합니다.
`\d*`	0번 이상 나오는 10진수를 찾습니다.
`(\s?\d+[.,]?\d*)`	공백 뒤에 하나 이상의 소수 자릿수, 마침표 또는 쉼표가 있을 수도 있으며, 그 뒤에는 0개 이상의 소수 자릿수를 찾습니다. 이 그룹은 첫 번째 캡처 그룹입니다. 대체 패턴이므로 `$1`메서드에 대한 Regex.Replace 호출은 일치하는 전체 부분 문자열을 캡처된 그룹으로 바꿉니다.

명명된 그룹 대체하기

`${` *이름* `}` 언어 요소는 *이름* 캡처링 그룹과 일치하는 마지막 부분 문자열을 대체합니다. 여기서 *이름은* 이름 언어 요소로 `(?<` 정의된 캡처링 그룹의 *이름* `>)`입니다. 명명된 캡처링 그룹에 대한 자세한 내용은 그룹화 구문을 참조하세요.

이름이 정규식 패턴에 정의된 유효한 명명된 캡처링 그룹을 지정하지 않지만 숫자 `${` 로 구성된 경우 *이름은* `}` 번호가 매겨진 그룹으로 해석됩니다.

이름이 유효한 명명된 캡처링 그룹이나 정규식 패턴 `${` 에 정의된 유효한 번호 매기기 캡처 그룹을 지정하지 않는 경우 *이름은* `}` 각 일치 항목을 대체하는 데 사용되는 리터럴 문자 시퀀스로 해석됩니다.

다음 예제에서는 `${` 대체를 `}` 사용하여 10진수 값에서 통화 기호를 제거합니다. 통화 값의 시작 또는 끝에 있는 통화 기호를 제거하고 가장 일반적인 두 개의 소수 구분 기호("." 및 ",")를 인식합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\p{Sc}*(?<amount>\s?\d+[.,]?\d*)\p{Sc}*";
      string replacement = "${amount}";
      string input = "$16.32 12.19 £16.29 €18.29  €18,29";
      string result = Regex.Replace(input, pattern, replacement);
      Console.WriteLine(result);
   }
}
// The example displays the following output:
//       16.32 12.19 16.29 18.29  18,29
```

정규식 패턴 `\p{Sc}*(?<amount>\s?\d[.,]?\d*)\p{Sc}*`는 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\p{Sc}*`	통화 기호 문자가 0개 이상일 경우 일치합니다.
`\s?`	공백 문자가 0개 또는 1개 있는지 일치시킵니다.
`\d+`	하나 이상의 10진수 숫자가 일치하는지 확인합니다.
`[.,]?`	0개 또는 1개의 마침표 또는 쉼표와 일치합니다.
`\d*`	0번 이상 나오는 10진수를 찾습니다.
`(?<amount>\s?\d[.,]?\d*)`	공백을 찾은 다음 하나 이상의 소수 자릿수를 찾은 다음 0개 또는 1개의 마침표 또는 쉼표 뒤에 0개 이상의 소수 자릿수를 찾습니다. `amount`라는 이름이 붙은 캡처링 그룹입니다. 대체 패턴이므로 `${amount}`메서드에 대한 Regex.Replace 호출은 일치하는 전체 부분 문자열을 캡처된 그룹으로 바꿉니다.

"$" 문자 대체

대체는 `$$` 대체된 문자열에 리터럴 "$" 문자를 삽입합니다.

다음 예제에서는 개체를 NumberFormatInfo 사용하여 현재 문화권의 통화 기호와 통화 문자열의 배치를 확인합니다. 그런 다음 정규식 패턴과 대체 패턴을 동적으로 빌드합니다. 현재 문화권이 en-US컴퓨터에서 이 예제를 실행하면 정규식 패턴 `\b(\d+)(\.(\d+))?` 과 대체 패턴 `$$ $1$2`

이 생성됩니다. 대체 패턴은 일치하는 텍스트를 통화 기호와 공백, 첫 번째 및 두 번째 캡처된 그룹으로 바꿉니다.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define array of decimal values.
      string[] values= { "16.35", "19.72", "1234", "0.99"};
      // Determine whether currency precedes (True) or follows (False) number.
      bool precedes = NumberFormatInfo.CurrentInfo.CurrencyPositivePattern % 2 ==
0;
      // Get decimal separator.
      string cSeparator = NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator;
      // Get currency symbol.
      string symbol = NumberFormatInfo.CurrentInfo.CurrencySymbol;
      // If symbol is a "$", add an extra "$".
      if (symbol == "$") symbol = "$$";

      // Define regular expression pattern and replacement string.
      string pattern = @"\b(\d+)(" + cSeparator + @"(\d+))?";
      string replacement = "$1$2";
      replacement = precedes ? symbol + " " + replacement : replacement + " " +
symbol;
      foreach (string value in values)
         Console.WriteLine($"{value} --> {Regex.Replace(value, pattern,
replacement)}");
   }
}
// The example displays the following output:
//       16.35 --> $ 16.35
//       19.72 --> $ 19.72
//       1234 --> $ 1234
//       0.99 --> $ 0.99
```

정규식 패턴 `\b(\d+)(\.(\d+))?`는 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계의 시작 부분에서 맞추기를 시작하세요.
`(\d+)`	하나 이상의 10진수 숫자가 일치하는지 확인합니다. 이 그룹은 첫 번째 캡처 그룹입니다.

패턴	설명
`\.`	마침표(소수 구분 기호)를 인식합니다.
`(\d+)`	하나 이상의 10진수 숫자가 일치하는지 확인합니다. 이 그룹은 세 번째 캡처링 그룹입니다.
`(\.(\d+))?`	마침표가 0개 또는 1회 등장하고 뒤에 하나 이상의 소수 자릿수가 오는 경우와 일치합니다. 이것이 두 번째 캡처링 그룹입니다.

전체 일치 항목 대체

`$&` 대체는 대체 문자열에 일치하는 전체 항목을 포함합니다. 종종 일치하는 문자열의 시작 또는 끝에 부분 문자열을 추가하는 데 사용됩니다. 예를 들어, `($&)` 대체 패턴은 각 일치 항목의 시작과 끝에 괄호를 추가합니다. 일치하는 항목이 없으면 대체에 `$&` 영향을 주지 않습니다.

다음 예제에서는 대체를 `$&` 사용하여 문자열 배열에 저장된 책 제목의 시작과 끝에 따옴표를 추가합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^(\w+\s?)+$";
      string[] titles = { "A Tale of Two Cities",
                          "The Hound of the Baskervilles",
                          "The Protestant Ethic and the Spirit of Capitalism",
                          "The Origin of Species" };
      string replacement = "\"$&\"";
      foreach (string title in titles)
         Console.WriteLine(Regex.Replace(title, pattern, replacement));
   }
}
// The example displays the following output:
//       "A Tale of Two Cities"
//       "The Hound of the Baskervilles"
//       "The Protestant Ethic and the Spirit of Capitalism"
//       "The Origin of Species"
```

정규식 패턴 `^(\w+\s?)+$`는 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	입력 문자열의 시작 부분에서 일치를 확인합니다.
`(\w+\s?)+`	하나 이상의 단어 문자에 0개 또는 1개의 공백 문자가 뒤따르는 패턴이 한 번 이상 반복되는 것을 일치시킵니다.
`$`	입력 문자열의 끝 부분을 찾습니다.

대체 패턴 `"$&"`은 각 일치 항목의 시작과 끝에 문자 그대로의 따옴표를 추가합니다.

일치하기 전에 텍스트 대체

대체는 `$`` 일치 문자열을 일치하기 전에 전체 입력 문자열로 바꿉니다. 즉, 일치하는 텍스트를 제거하고, 일치할 때까지 입력 문자열을 복제합니다. 일치하는 텍스트 뒤에 있는 모든 텍스트는 결과 문자열에서 변경되지 않습니다. 입력 문자열에 일치하는 항목이 여러 개 있는 경우 대체 텍스트는 텍스트가 이전 일치 항목으로 대체된 문자열이 아닌 원래 입력 문자열에서 파생됩니다. (예제에서는 그림을 제공합니다.) 일치하는 항목이 없으면 대체에 `$`` 영향을 주지 않습니다.

다음 예제에서는 정규식 패턴을 `\d+` 사용하여 입력 문자열에서 하나 이상의 소수 자릿수 시퀀스를 일치시킵니다. 대체 문자열 `$`` 은 이러한 숫자를 일치 항목 앞에 오는 텍스트로 바꿉니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aa1bb2cc3dd4ee5";
      string pattern = @"\d+";
      string substitution = "$`";
      Console.WriteLine("Matches:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"   {match.Value} at position {match.Index}");

      Console.WriteLine($"Input string:  {input}");
      Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//    Matches:
//       1 at position 2
//       2 at position 5
//       3 at position 8
//       4 at position 11
```

```
//       5 at position 14
//    Input string:  aa1bb2cc3dd4ee5
//    Output string: aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddeeaa1bb2cc3dd4ee
```

이 예제에서 입력 문자열 `"aa1bb2cc3dd4ee5"` 은 5개의 일치 항목을 포함합니다. 다음 표에서는 `$`` 대체가 정규식 엔진이 입력 문자열의 각 일치 항목을 대체되는 방법을 보여 줍니다. 삽입된 텍스트는 결과 열에 굵게 표시됩니다.

테이블 확장

경기	위치	일치 이전의 문자열	결과 문자열
1	2	aa	aa**aa**bb2cc3dd4ee5
2	5	aa1bb	aaaabb**aa1bb**cc3dd4ee5
3	8 (여덟)	aa1bb2cc	aaaabbaa1bbcc**aa1bb2cc**dd4ee5
4	11	aa1bb2cc3ddd	aaaabbaa1bbccaa1bb2ccdd**aa1bb2cc3dd**ee5
5	14	aa1bb2cc3dd4ee	aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddee**aa1bb2cc3dd4ee**

일치 후 텍스트 대체

대체는 `$'` 일치하는 문자열을 일치 후 전체 입력 문자열로 바꿉니다. 즉, 일치하는 텍스트를 제거하는 동안 일치 후 입력 문자열을 복제합니다. 일치하는 텍스트 앞에 오는 모든 텍스트는 결과 문자열에서 변경되지 않습니다. 일치하는 항목이 없으면 대체에 `$'` 영향을 주지 않습니다.

다음 예제에서는 정규식 패턴을 `\d+` 사용하여 입력 문자열에서 하나 이상의 소수 자릿수 시퀀스를 일치시킵니다. 대체 문자열 `$'` 은 이러한 숫자를 일치 항목 뒤에 있는 텍스트로 바꿉니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aa1bb2cc3dd4ee5";
      string pattern = @"\d+";
      string substitution = "$'";
      Console.WriteLine("Matches:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine($"   {match.Value} at position {match.Index}");
      Console.WriteLine($"Input string:  {input}");
```

```
      Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//    Matches:
//       1 at position 2
//       2 at position 5
//       3 at position 8
//       4 at position 11
//       5 at position 14
//    Input string:  aa1bb2cc3dd4ee5
//    Output string: aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee
```

이 예제에서 입력 문자열 `"aa1bb2cc3dd4ee5"` 은 5개의 일치 항목을 포함합니다. 다음 표에서는 `$'` 대체가 정규식 엔진이 입력 문자열의 각 일치 항목을 대체되는 방법을 보여 줍니다. 삽입된 텍스트는 결과 열에 굵게 표시됩니다.

테이블 확장

경기	위치	일치된 후의 문자열	결과 문자열
1	2	bb2cc3dd4ee5	aa**bb2cc3dd4ee5**bb2cc3dd4ee5
2	5	cc3dd4ee5	aabb2cc3dd4ee5bb**cc3dd4ee5**cc3dd4ee5
3	8 (여덟)	dd4ee5	aabb2cc3dd4ee5bbcc3dd4ee5cc**dd4ee5**dd4ee5
4	11	ee5	aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5dd**ee5**ee5
5	14	String.Empty	aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee

마지막으로 캡처된 그룹을 대체하기

대체 `$+`는 일치하는 문자열을 마지막으로 캡처된 그룹으로 대체합니다. 캡처된 그룹이 없거나 마지막으로 캡처된 그룹의 값이 String.Empty `$+` 있는 경우 대체는 적용되지 않습니다.

다음 예제에서는 문자열에서 중복 단어를 식별하고 대체를 사용하여 `$+` 단어의 단일 항목으로 바꿉니다. 이 RegexOptions.IgnoreCase 옵션은 대/소문자만 다르고 그 외에는 동일한 단어가 중복된 것으로 간주되도록 하는 데 사용됩니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
```

```
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//       The dog jumped over the fence.
```

정규식 패턴 `\b(\w+)\s\1\b`는 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서부터 일치를 시작합니다.
`(\w+)`	하나 이상의 단어 문자를 일치시킵니다. 이 그룹은 첫 번째 캡처 그룹입니다.
`\s`	공백 문자를 일치시킵니다.
`\1`	첫 번째 캡처된 그룹을 매칭하십시오.
`\b`	단어 경계에서 경기를 종료합니다.

전체 입력 문자열 대체

대체는 `$_` 일치하는 문자열을 전체 입력 문자열로 바꿉니다. 즉, 일치하는 텍스트를 제거하고 일치하는 텍스트를 포함하여 전체 문자열로 바꿉니다.

다음 예제에서는 입력 문자열에서 하나 이상의 10진수를 찾습니다. 전체 입력 문자열로 교체하기 위해 `$_` 대체 기능을 사용합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "ABC123DEF456";
      string pattern = @"\d+";
      string substitution = "$_";
      Console.WriteLine($"Original string:          {input}");
```

```
      Console.WriteLine($"String with substitution: {Regex.Replace(input, pattern,
substitution)}");
   }
}
// The example displays the following output:
//       Original string:          ABC123DEF456
//       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

이 예제에서 입력 문자열 `"ABC123DEF456"` 에는 두 개의 일치 항목이 포함됩니다. 다음 표에서는 `$_` 대체가 정규식 엔진이 입력 문자열의 각 일치 항목을 대체되는 방법을 보여 줍니다. 삽입된 텍스트는 결과 열에 굵게 표시됩니다.

테이블 확장

경기	위치	경기	결과 문자열
1	3	123	ABC**ABC123DEF456**DEF456
2	5	456	ABCABC123DEF456DEF**ABC123DEF456**

참고하십시오

- 정규식 언어 - 빠른 참조

정규식 옵션

아티클 • 2023. 06. 27.

기본적으로 정규식 패턴의 리터럴 문자와 입력 문자열의 비교는 대/소문자를 구분하고, 정규식 패턴의 공백은 리터럴 공백 문자로 해석되며, 정규식의 캡처 그룹은 암시적으로 및 명시적으로 명명됩니다. 정규식 옵션을 지정하여 기본 정규식 동작의 이러한 측면과 몇 가지 다른 측면을 수정할 수 있습니다. 다음 표에 나열된 이러한 옵션 중 일부는 정규식 패턴의 일부로 인라인으로 포함되거나 클래스 생성자 또는 정적 패턴 일치 메서드에 System.Text.RegularExpressions.Regex 열거형 값으로 System.Text.RegularExpressions.RegexOptions 제공할 수 있습니다.

`RegexOptions` 멤버	인라인 문자	효과	추가 정보
None	사용할 수 없음	기본 동작을 사용합니다.	기본 옵션
IgnoreCase	`i`	대/소문자를 구분하지 않는 일치를 사용합니다.	대/소문자를 구분하지 않는 일치
Multiline	`m`	입력 문자열의 시작과 끝 대신 각 줄의 시작과 끝을 나타내는 `^` `$` 다중 줄 모드를 사용합니다.	여러 줄 모드
Singleline	`s`	한 줄 모드를 사용합니다. 여기서 마침표(.)는 모든 문자와 일치합니다(`\n`을 제외한 모든 문자 대신).	한 줄 모드
ExplicitCapture	`n`	명명되지 않은 그룹을 캡처하지 않습니다. 유효한 유일한 캡처는 양식 `(?<`*이름*`>`*하위*`)` 식의 명시적으로 명명되거나 번호가 매겨진 그룹입니다.	명시적 캡처만
Compiled	사용할 수 없음	정규식을 어셈블리로 컴파일합니다.	컴파일된 정규식

`RegexOptions` 멤버	인라인 문자	효과	추가 정보
IgnorePatternWhitespace	`x`	이스케이프되지 않은 공백을 패턴에서 제외하고 숫자 기호(`#`) 뒤에 주석을 사용하도록 설정합니다.	공백 무시
RightToLeft	사용할 수 없음	검색 방향을 변경합니다. 검색이 왼쪽에서 오른쪽으로 대신 오른쪽에서 왼쪽으로 이동합니다.	오른쪽에서 왼쪽 모드
ECMAScript	사용할 수 없음	식에 대해 ECMAScript와 호환되는 동작을 사용하도록 설정합니다.	ECMAScript 일치 동작
CultureInvariant	사용할 수 없음	언어의 문화권 차이를 무시합니다.	고정 문화권을 사용한 비교
NonBacktracking	사용할 수 없음	역추적을 방지하고 입력 길이의 선형 시간 처리를 보장하는 방법을 사용하여 일치합니다. (.NET 7 이상 버전에서 사용할 수 있습니다.)	역추적 모드

옵션 지정

정규식에 대한 옵션은 다음과 같은 세 가지 방법 중 하나로 지정할 수 있습니다.

- `options` 클래스 생성자 또는 정적(Visual Basic의 경우 System.Text.RegularExpressions.Regex) 패턴 일치 메서드(예: `Shared` 또는 Regex(String, RegexOptions))의 Regex.Match(String, String, RegexOptions) 매개 변수에 지정합니다. `options` 매개 변수는 System.Text.RegularExpressions.RegexOptions 열거형 값의 비트 OR 조합입니다.

클래스 생성자의 `options` 매개 변수를 사용하여 Regex 인스턴스에 옵션을 제공하면 해당 옵션이 System.Text.RegularExpressions.RegexOptions 속성에 할당됩니다. 그러나 System.Text.RegularExpressions.RegexOptions 속성은 정규식 패턴 자체에 인라인 옵션을 반영하지는 않습니다.

다음 예제에서 이에 대해 설명합니다. 이 예제에서는 `options` 메서드의 Regex.Match(String, String, RegexOptions) 매개 변수를 사용하여 대/소문자를 구분하지 않는 일치를 사용하도록 설정하고 문자 "d"로 시작하는 단어를 식별할 때 패턴 공백을 무시합니다.

```
C#
```

```
string pattern = @"d \w+ \s";
string input = "Dogs are decidedly good pets.";
RegexOptions options = RegexOptions.IgnoreCase |
RegexOptions.IgnorePatternWhitespace;

foreach (Match match in Regex.Matches(input, pattern, options))
    Console.WriteLine("'{0}// found at index {1}.", match.Value,
match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

- `(?imnsx-imnsx)` 구문을 사용하여 정규식 패턴에 인라인 옵션을 적용합니다. 이 옵션은 옵션이 정의된 지점에서 패턴의 끝 부분까지 또는 다른 인라인 옵션에 의해 옵션이 정의되지 않은 지점까지 패턴에 적용됩니다. Regex 인스턴스의 System.Text.RegularExpressions.RegexOptions 속성은 이러한 인라인 옵션을 반영하지 않습니다. 자세한 내용은 기타 구문 항목을 참조하세요.

  다음 예제에서 이에 대해 설명합니다. 이 예제에서는 인라인 옵션을 사용하여 대/소문자를 구분하지 않는 일치를 사용하도록 설정하고 문자 "d"로 시작하는 단어를 식별할 때 패턴 공백을 무시합니다.

  ```
  C#
  ```

  ```
  string pattern = @"(?ix) d \w+ \s";
  string input = "Dogs are decidedly good pets.";

  foreach (Match match in Regex.Matches(input, pattern))
      Console.WriteLine("'{0}// found at index {1}.", match.Value,
  match.Index);
  // The example displays the following output:
  //    'Dogs // found at index 0.
  //    'decidedly // found at index 9.
  ```

- `(?imnsx-imnsx:`*subexpression*`)` 구문을 사용하여 정규식 패턴의 특정 그룹화 구문에 인라인 옵션을 적용합니다. 옵션 집합 앞에 기호가 없으면 집합이 설정되고, 옵션 집합 앞에 빼기 기호가 있으면 집합이 해제됩니다. `?`는 옵션의 사용 여부에 따라 필요한 언어 구문의 고정 부분입니다. 이 옵션은 해당 그룹에만 적용됩니다. 자세한 내용은 그룹화 구문을 참조하세요.

  다음 예제에서 이에 대해 설명합니다. 이 예제에서는 그룹화 구문에 인라인 옵션을 사용하여 대/소문자를 구분하지 않는 일치를 사용하도록 설정하고 문자 "d"로 시작하는 단어를 식별할 때 패턴 공백을 무시합니다.

  C#

  ```
  string pattern = @"\b(?ix: d \w+)\s";
  string input = "Dogs are decidedly good pets.";

  foreach (Match match in Regex.Matches(input, pattern))
      Console.WriteLine("'{0}// found at index {1}.", match.Value,
  match.Index);
  // The example displays the following output:
  //    'Dogs // found at index 0.
  //    'decidedly // found at index 9.
  ```

옵션이 인라인으로 지정된 경우 옵션 또는 옵션 집합 앞에 빼기 기호(`-`)가 있으면 해당 옵션이 해제됩니다. 예를 들어, 인라인 구문 `(?ix-ms)`는 RegexOptions.IgnoreCase 및 RegexOptions.IgnorePatternWhitespace 옵션을 설정하고 RegexOptions.Multiline 및 RegexOptions.Singleline 옵션을 해제합니다. 모든 정규식 옵션은 기본적으로 해제되어 있습니다.

> **참고**
>
> 생성자 또는 메서드 호출의 `options` 매개 변수에 지정된 정규식 옵션이 정규식 패턴에 인라인으로 지정된 옵션과 충돌하는 경우 인라인 옵션이 사용됩니다.

다음 5개의 정규식 옵션은 옵션 매개 변수와 인라인으로 모두 설정할 수 있습니다.

- RegexOptions.IgnoreCase
- RegexOptions.Multiline
- RegexOptions.Singleline
- RegexOptions.ExplicitCapture
- RegexOptions.IgnorePatternWhitespace

다음 5개의 정규식 옵션은 `options` 매개 변수를 사용하여 설정할 수 있지만 인라인으로는 설정할 수 없습니다.

- RegexOptions.None
- RegexOptions.Compiled
- RegexOptions.RightToLeft
- RegexOptions.CultureInvariant
- RegexOptions.ECMAScript

옵션 결정

읽기 전용 Regex 속성 값을 검색하여 Regex.Options 개체를 인스턴스화했을 때 해당 개체에 제공된 옵션을 확인할 수 있습니다. 이 속성은 Regex.CompileToAssembly 메서드에 의해 만들어진 컴파일된 정규식에 대해 정의된 옵션을 확인하는 데 특히 유용합니다.

RegexOptions.None을 제외한 모든 옵션의 존재 유무를 테스트하려면 Regex.Options 속성 값 및 관심 있는 RegexOptions 값으로 AND 작업을 수행합니다. 그런 다음 결과가 해당 RegexOptions 값과 같은지 테스트합니다. 다음 예제에서는 RegexOptions.IgnoreCase 옵션이 설정되었는지 테스트합니다.

C#

```
if ((rgx.Options & RegexOptions.IgnoreCase) == RegexOptions.IgnoreCase)
    Console.WriteLine("Case-insensitive pattern comparison.");
else
    Console.WriteLine("Case-sensitive pattern comparison.");
```

RegexOptions.None에 대해 테스트하려면 다음 예제에 표시된 것처럼 Regex.Options 속성 값이 RegexOptions.None과 같은지 확인합니다.

C#

```
if (rgx.Options == RegexOptions.None)
    Console.WriteLine("No options have been set.");
```

다음 섹션에는 .NET의 정규식에서 지원하는 옵션이 나열되어 있습니다.

기본 옵션

RegexOptions.None 옵션은 지정된 옵션이 없고 정규식 엔진에서 해당 기본 동작을 사용함을 나타냅니다. 여기에는 다음과 같은 사항이 포함됩니다.

- 패턴이 ECMAScript 정규식이 아니라 정식으로 해석됩니다.
- 정규식 패턴이 입력 문자열에서 왼쪽에서 오른쪽으로 일치됩니다.
- 비교는 대/소문자를 구분합니다.
- `^` 및 `$` 언어 요소는 입력 문자열의 시작과 끝을 나타냅니다. 입력 문자열의 끝은 후행 줄 바 `\n` 꿈 문자일 수 있습니다.
- `.` 언어 요소는 `\n`을 제외한 모든 문자와 일치합니다.
- 정규식 패턴의 모든 공백은 리터럴 공백 문자로 해석됩니다.
- 패턴을 입력 문자열과 비교할 때 현재 문화권의 규칙이 사용됩니다.
- 정규식 패턴의 캡처링 그룹은 명시적 및 암시적입니다.

> ⓘ **참고**
>
> RegexOptions.None 옵션과 동일한 인라인 옵션은 없습니다. 정규식 옵션이 인라인으로 적용된 경우 기본 동작은 특정 옵션을 해제하여 옵션별로 복원됩니다. 예를 들어, `(?i)`는 대/소문자를 구분하지 않는 비교를 설정하고 `(?-i)`는 기본 대/소문자를 구분하는 비교를 복원합니다.

RegexOptions.None 옵션은 정규식 엔진의 기본 동작을 나타내므로 메서드 호출에 명시적으로 지정되는 경우가 드뭅니다. 대신 생성자 또는 정적 패턴 일치 메서드가 `options` 매개 변수 없이 호출됩니다.

대/소문자를 구분하지 않는 일치

IgnoreCase 옵션 또는 `i` 인라인 옵션은 대/소문자를 구분하지 않는 일치를 제공합니다. 기본적으로 현재 문화권의 대/소문자 사용 규칙이 사용됩니다.

다음 예제에서는 "the"로 시작하는 모든 단어와 일치하는 정규식 패턴 `\bthe\w*\b`를 정의합니다. Match 메서드에 대한 첫 번째 호출에서 기본 대/소문자를 구분하는 비교를 사용하므로 출력은 문장을 시작하는 문자열 "The"가 일치하지 않음을 나타냅니다. Match 메서드가 옵션이 IgnoreCase로 설정된 상태로 호출되면 이 문자열이 일치합니다.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bthe\w*\b";
      string input = "The man then told them about that event.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found {0} at index {1}.", match.Value,
match.Index);

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern,
                                            RegexOptions.IgnoreCase))
         Console.WriteLine("Found {0} at index {1}.", match.Value,
match.Index);
   }
}
// The example displays the following output:
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

다음 예제에서는 대/소문자를 구분하지 않는 비교를 제공하기 위해 `options` 매개 변수 대신 인라인 옵션을 사용하도록 이전 예제의 정규식 패턴을 수정합니다. 첫 번째 패턴은 문자열 "the"에서 문자 "t"에만 적용되는 그룹화 구문에 대/소문자를 구분하지 않는 옵션을 정의합니다. 옵션 구문이 패턴의 시작 부분에서 발생하므로 두 번째 패턴은 대/소문자를 구분하지 않는 옵션을 전체 정규식에 적용합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class CaseExample
{
    public static void Main()
    {
        string pattern = @"\b(?i:t)he\w*\b";
        string input = "The man then told them about that event.";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Found {0} at index {1}.", match.Value,
match.Index);

        Console.WriteLine();
        pattern = @"(?i)\bthe\w*\b";
```

```
        foreach (Match match in Regex.Matches(input, pattern,
                                              RegexOptions.IgnoreCase))
            Console.WriteLine("Found {0} at index {1}.", match.Value,
match.Index);
    }
}
// The example displays the following output:
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

여러 줄 모드

RegexOptions.Multiline 옵션 또는 `m` 인라인 옵션은 정규식 엔진이 여러 줄로 구성된 입력 문자열을 처리할 수 있게 해줍니다. 입력 문자열의 시작과 끝 대신 줄의 `^` 시작과 끝을 나타내도록 및 `$` 언어 요소의 해석을 변경합니다.

기본적으로 는 `$` 입력 문자열의 끝에만 충족됩니다. 옵션을 지정 RegexOptions.Multiline 하면 줄 바꿈 문자(`\n`) 또는 입력 문자열의 끝에 의해 충족됩니다.

두 경우 모두 캐리지 리턴/줄 바꿈 문자 조합(`\r\n`)을 인식하지 못합니다`$`. `$` 항상 캐리지 리턴(`\r`)을 무시합니다. 또는 `\n`로 `\r\n` 일치를 종료하려면 대신 `$`하위 식 `\r?$` 을 사용합니다. 이렇게 하면 일치 항목의 `\r` 일부가 됩니다.

다음 예제에서는 볼링하는 사람의 이름과 점수를 추출하여 이를 내림차순으로 정렬하는 SortedList<TKey,TValue> 컬렉션에 추가합니다. Matches 메서드가 두 번 호출됩니다. 첫 번째 메서드 호출에서 정규식은 `^(\w+)\s(\d+)$`이며 옵션이 설정되지 않았습니다. 출력에 표시된 것처럼, 정규식 엔진이 입력 패턴을 입력 문자열의 시작 부분 및 끝 부분과 함께 일치시킬 수 없어 찾은 일치 항목이 없습니다. 두 번째 메서드 호출에서 정규식은 `^(\w+)\s(\d+)\r?$`로 변경되었으며 옵션이 RegexOptions.Multiline으로 설정되었습니다. 출력에 표시된 것처럼, 이름 및 점수가 성공적으로 일치되었으며 점수가 내림차순으로 표시되었습니다.

C#

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Multiline1Example
{
    public static void Main()
```

```
    {
        SortedList<int, string> scores = new SortedList<int, string>(new
DescendingComparer1<int>());

        string input = "Joe 164\n" +
                       "Sam 208\n" +
                       "Allison 211\n" +
                       "Gwen 171\n";
        string pattern = @"^(\w+)\s(\d+)$";
        bool matched = false;

        Console.WriteLine("Without Multiline option:");
        foreach (Match match in Regex.Matches(input, pattern))
        {
            scores.Add(Int32.Parse(match.Groups[2].Value),
(string)match.Groups[1].Value);
            matched = true;
        }
        if (!matched)
            Console.WriteLine("   No matches.");
        Console.WriteLine();

        // Redefine pattern to handle multiple lines.
        pattern = @"^(\w+)\s(\d+)\r*$";
        Console.WriteLine("With multiline option:");
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Multiline))
            scores.Add(Int32.Parse(match.Groups[2].Value),
(string)match.Groups[1].Value);

        // List scores in descending order.
        foreach (KeyValuePair<int, string> score in scores)
            Console.WriteLine("{0}: {1}", score.Value, score.Key);
    }
}

public class DescendingComparer1<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return Comparer<T>.Default.Compare(x, y) * -1;
    }
}
// The example displays the following output:
//   Without Multiline option:
//      No matches.
//
//   With multiline option:
//   Allison: 211
//   Sam: 208
//   Gwen: 171
//   Joe: 164
```

정규식 패턴 `^(\w+)\s(\d+)\r*$`는 다음 테이블과 같이 정의됩니다.

무늬	설명
`^`	줄의 시작 부분에서 시작합니다.
`(\w+)`	하나 이상의 단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\s`	공백 문자를 찾습니다.
`(\d+)`	하나 이상의 10진수 숫자가 일치하는지 확인합니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`\r?`	0개 또는 1개의 캐리지 리턴 문자를 찾습니다.
`$`	줄의 끝 부분에서 끝납니다.

다음 예제는 인라인 옵션 `(?m)`를 사용하여 여러 줄 옵션을 설정한다는 점을 제외하고는 이전 예제와 동일합니다.

C#

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Multiline2Example
{
    public static void Main()
    {
        SortedList<int, string> scores = new SortedList<int, string>(new
DescendingComparer<int>());

        string input = "Joe 164\n" +
                       "Sam 208\n" +
                       "Allison 211\n" +
                       "Gwen 171\n";
        string pattern = @"(?m)^(\w+)\s(\d+)\r*$";

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Multiline))
            scores.Add(Convert.ToInt32(match.Groups[2].Value),
match.Groups[1].Value);

        // List scores in descending order.
        foreach (KeyValuePair<int, string> score in scores)
            Console.WriteLine("{0}: {1}", score.Value, score.Key);
    }
}

public class DescendingComparer<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
```

```
        return Comparer<T>.Default.Compare(x, y) * -1;
    }
}
// The example displays the following output:
//    Allison: 211
//    Sam: 208
//    Gwen: 171
//    Joe: 164
```

한 줄 모드

RegexOptions.Singleline 옵션 또는 `s` 인라인 옵션은 정규식 엔진이 입력 문자열이 한 줄로 구성된 것처럼 입력 문자열을 처리하도록 합니다. 줄 바꿈 문자를 `\n`제외한 모든 문자를 일치시키는 대신 모든 문자와 일치하도록 마침표(`.`) 언어 요소의 동작을 변경하여 이 작업을 수행합니다.

다음 예제에서는 `.` 옵션을 사용할 때 RegexOptions.Singleline 언어 요소의 동작이 변경되는 방법을 보여 줍니다. 정규식 `^.+`는 문자열의 시작 부분에서 시작하여 모든 문자를 찾습니다. 기본적으로 일치는 첫 번째 줄의 끝에서 끝납니다. 정규식 패턴은 캐리지 리턴 문자 `\r`와 일치하지만 는 일치하지 `\n`않습니다. RegexOptions.Singleline 옵션은 전체 입력 문자열을 한 줄로 해석하기 때문에 `\n`을 포함하여 입력 문자열의 모든 문자와 일치합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^.+";
      string input = "This is one line and" + Environment.NewLine + "this is
the second.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Singleline))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r
//
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

다음 예제는 인라인 옵션 `(?s)`를 사용하여 한 줄 모드를 사용하도록 설정한다는 점을 제외하고는 이전 예제와 동일합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class SingleLineExample
{
    public static void Main()
    {
        string pattern = "(?s)^.+";
        string input = "This is one line and" + Environment.NewLine + "this
is the second.";

        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(Regex.Escape(match.Value));
    }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

명시적 캡처만 해당

기본적으로 캡처링 그룹은 정규식 패턴에 괄호를 사용하여 정의됩니다. 명명된 그룹은 `(?<`*name*`>`*subexpression*`)` 언어 옵션에 의해 이름 또는 번호가 할당되는 반면, 명명되지 않은 그룹은 인덱스로 액세스할 수 있습니다. GroupCollection 개체에서 명명되지 않은 그룹은 명명된 그룹 앞에 있습니다.

그룹화 구문은 일반적으로 여러 언어 요소에 수량자를 적용하는 데만 사용되고, 캡처된 부분 문자열은 관심을 두는 부분이 아닙니다. 예를 들어, 다음 정규식

```
\b\(?((\w+),?\s?)+[\.!?]\)?
```

가 문서에서 마침표, 느낌표 또는 물음표로 끝나는 문장만 추출하려는 의도로 사용된 경우 결과 문장(Match 개체로 표현됨)만 관심을 두는 부분입니다. 컬렉션의 개별 단어는 관심을 두는 부분이 아닙니다.

정규식 엔진이 GroupCollection 및 CaptureCollection 컬렉션 개체를 둘 다 채워야 하므로 이후에 사용되지 않는 캡처링 그룹에는 비용이 많이 들 수 있습니다. 또는 옵션 또는 `n` 인라인 옵션을 사용하여 RegexOptions.ExplicitCapture 이름*하위*`)` 식 구문으로 지정된 유일한 유효한 캡처가 명시적으로 명명되거나 번호가 매겨진 `(?< >)` 그룹임을 지정할 수 있습니다.

다음 예제에서는 `\b\(?((\w+),?\s?)+[\.!?]\)?` 메서드가 Match 옵션을 사용하여 호출된 경우와 이 옵션을 사용하지 않고 호출된 경우에 RegexOptions.ExplicitCapture 정규식 패턴에서 반환하는 일치 항목에 대한 정보를 표시합니다. 첫 번째 메서드 호출의 출력이 보여 주는 것처럼, 정규식 엔진은 캡처된 부분 문자열에 대한 정보로 GroupCollection 및 CaptureCollection 컬렉션 개체를 완전히 채웁니다. 두 번째 메서드는 `options`가 RegexOptions.ExplicitCapture로 설정된 상태로 호출되었기 때문에 그룹에 대한 정보를 캡처하지 않습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Explicit1Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                       "of a literary masterpiece? I think not. Instead, " +
                       "it is a nonsensical paragraph.";
        string pattern = @"\b\(?((?>\w+),?\s?)+[\.!?]\)?";
        Console.WriteLine("With implicit captures:");
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("   Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("      Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
        Console.WriteLine();
        Console.WriteLine("With explicit captures only:");
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.ExplicitCapture))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("   Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
```

```
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("      Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
    }
}
// The example displays the following output:
//    With implicit captures:
//    The match: This is the first sentence.
//       Group 0: This is the first sentence.
//          Capture 0: This is the first sentence.
//       Group 1: sentence
//          Capture 0: This
//          Capture 1: is
//          Capture 2: the
//          Capture 3: first
//          Capture 4: sentence
//       Group 2: sentence
//          Capture 0: This
//          Capture 1: is
//          Capture 2: the
//          Capture 3: first
//          Capture 4: sentence
//    The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//          Capture 0: Is it the beginning of a literary masterpiece?
//       Group 1: masterpiece
//          Capture 0: Is
//          Capture 1: it
//          Capture 2: the
//          Capture 3: beginning
//          Capture 4: of
//          Capture 5: a
//          Capture 6: literary
//          Capture 7: masterpiece
//       Group 2: masterpiece
//          Capture 0: Is
//          Capture 1: it
//          Capture 2: the
//          Capture 3: beginning
//          Capture 4: of
//          Capture 5: a
//          Capture 6: literary
//          Capture 7: masterpiece
//    The match: I think not.
//       Group 0: I think not.
//          Capture 0: I think not.
//       Group 1: not
//          Capture 0: I
//          Capture 1: think
```

```
//          Capture 2: not
//       Group 2: not
//          Capture 0: I
//          Capture 1: think
//          Capture 2: not
//    The match: Instead, it is a nonsensical paragraph.
//       Group 0: Instead, it is a nonsensical paragraph.
//          Capture 0: Instead, it is a nonsensical paragraph.
//       Group 1: paragraph
//          Capture 0: Instead,
//          Capture 1: it
//          Capture 2: is
//          Capture 3: a
//          Capture 4: nonsensical
//          Capture 5: paragraph
//       Group 2: paragraph
//          Capture 0: Instead
//          Capture 1: it
//          Capture 2: is
//          Capture 3: a
//          Capture 4: nonsensical
//          Capture 5: paragraph
//
//    With explicit captures only:
//    The match: This is the first sentence.
//       Group 0: This is the first sentence.
//          Capture 0: This is the first sentence.
//    The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//          Capture 0: Is it the beginning of a literary masterpiece?
//    The match: I think not.
//       Group 0: I think not.
//          Capture 0: I think not.
//    The match: Instead, it is a nonsensical paragraph.
//       Group 0: Instead, it is a nonsensical paragraph.
//          Capture 0: Instead, it is a nonsensical paragraph.
```

정규식 패턴 `\b\(?((?>\w+),?\s?)+[\.!?]\)?`는 다음 표와 같이 정의됩니다.

무늬	설명
`\b`	단어 경계에서 시작합니다.
`\(?`	0개 또는 1개의 여는 괄호("(")를 찾습니다.
`(?>\w+),?`	하나 이상의 단어 문자 다음에 0개 또는 1개의 쉼표가 있는 일치 항목을 찾습니다. 단어 문자를 찾을 때 역추적하지 않습니다.
`\s?`	0회 이상 나오는 공백 문자를 찾습니다.
`((\w+),?\s?)+`	하나 이상의 단어 문자 다음에 0개 또는 1개의 쉼표 및 0개 또는 1개의 공백 문자가 한 번 이상 나타나는 조합을 찾습니다.

무늬	설명
`[\.!?]\)?`	세 가지 문장 부호 기호 다음에 0개 또는 1개의 닫는 괄호(")")가 있는 모든 일치 항목을 찾습니다.

또한 `(?n)` 인라인 요소를 사용하여 자동 캡처를 억제할 수 있습니다. 다음 예제에서는 `(?n)` 옵션 대신 RegexOptions.ExplicitCapture 인라인 요소를 사용하도록 이전 정규식 패턴을 수정합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Explicit2Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                       "of a literary masterpiece? I think not. Instead, " +
                       "it is a nonsensical paragraph.";
        string pattern = @"(?n)\b\(?((?>\w+),?\s?)+[\.!?]\)?";

        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("   Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("      Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
    }
}
// The example displays the following output:
//       The match: This is the first sentence.
//          Group 0: This is the first sentence.
//             Capture 0: This is the first sentence.
//       The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//             Capture 0: Is it the beginning of a literary masterpiece?
//       The match: I think not.
//          Group 0: I think not.
```

```
//             Capture 0: I think not.
//       The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//             Capture 0: Instead, it is a nonsensical paragraph.
```

마지막으로, 인라인 그룹 요소 `(?n:)`를 사용하여 그룹별로 자동 캡처를 억제할 수 있습니다. 다음 예제에서는 바깥쪽 그룹 `((?>\w+),?\s?)`에서 명명되지 않은 캡처를 억제하도록 이전 패턴을 수정합니다. 여기서는 안쪽 그룹에서도 명명되지 않은 캡처가 억제됩니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Explicit3Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                       "of a literary masterpiece? I think not. Instead, " +
                       "it is a nonsensical paragraph.";
        string pattern = @"\b\(?(?n:(?>\w+),?\s?)+[\.!?]\)?";

        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("   Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("      Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
    }
}
// The example displays the following output:
//       The match: This is the first sentence.
//          Group 0: This is the first sentence.
//             Capture 0: This is the first sentence.
//       The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//             Capture 0: Is it the beginning of a literary masterpiece?
//       The match: I think not.
//          Group 0: I think not.
```

```
//             Capture 0: I think not.
//       The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//             Capture 0: Instead, it is a nonsensical paragraph.
```

컴파일된 정규식

> ⓘ **참고**
>
> 가능한 경우 옵션을 사용하여 정규식을 컴파일하는 대신 **소스에서 생성된 정규** 식을 사용합니다 RegexOptions.Compiled . 원본 생성은 앱을 더 빠르게 시작하고, 더 빠르게 실행하고, 더 잘릴 수 있도록 도와줄 수 있습니다. 원본 생성이 가능한 경우를 알아보려면 **사용 시기를 참조하세요**.

기본적으로 .NET의 정규식은 해석됩니다. Regex 개체가 인스턴스화되거나 정적 Regex 메서드가 호출되면 정규식 패턴이 일련의 사용자 지정 opcode로 구문 분석되고 해석기는 이러한 opcode를 사용하여 정규식을 실행합니다. 여기에는 절충 사항이 수반됩니다. 즉, 정규식 엔진의 초기화 비용은 런타임 성능을 희생하여 최소화됩니다.

RegexOptions.Compiled 옵션을 사용하여, 해석된 정규식 대신 컴파일된 정규식을 사용할 수 있습니다. 이 경우 패턴이 정규식 엔진에 전달되면 일련의 opcode로 구문 분석된 다음, 공용 언어 런타임으로 직접 전달될 수 있는 MSIL(Microsoft Intermediate Language)로 변환됩니다. 컴파일된 정규식은 초기화 시간을 희생하여 런타임 성능을 최대화합니다.

> ⓘ **참고**
>
> 정규식은 RegexOptions.Compiled 값을 `options` 클래스 생성자 또는 정적 패턴 일치 메서드의 Regex 매개 변수에 제공해서만 컴파일할 수 있으며, 인라인 옵션으로는 사용할 수 없습니다.

컴파일된 정규식은 정적 정규식과 인스턴스 정규식 둘 다에 대한 호출에 사용할 수 있습니다. 정적 정규식에서 RegexOptions.Compiled 옵션은 정규식 패턴 일치 메서드의 `options` 매개 변수에 전달됩니다. 인스턴스 정규식에서 이 옵션은 `options` 클래스 생성자의 Regex 매개 변수에 전달됩니다. 따라서 두 가지 경우 모두에서 성능이 향상됩니다.

그러나 이 성능 향상은 다음 조건에서만 발생합니다.

- 특정 정규식을 나타내는 Regex 개체가 정규식 패턴 일치 메서드에 대한 여러 번의 호출에 사용됩니다.
- Regex 개체가 범위를 벗어날 수 없어 재사용될 수 있습니다.

- 정적 정규식이 정규식 패턴 일치 메서드에 대한 여러 번의 호출에 사용됩니다. (정적 메서드 호출에 사용된 정규식이 정규식 엔진에 의해 캐시되므로 성능 향상이 가능합니다.)

> ⓘ **참고**
>
> **RegexOptions.Compiled** 옵션은 미리 정의된 컴파일된 정규식을 포함하는 특수 목적 어셈블리를 만드는 **Regex.CompileToAssembly** 메서드와는 관련이 없습니다.

공백 무시

기본적으로 정규식 패턴에서 공백은 중요합니다. 공백은 정규식 엔진이 입력 문자열에서 공백 문자를 강제로 일치시키도록 합니다. 따라서 정규식 "`\b\w+\s`"와 "`\b\w+`"는 거의 동일한 정규식입니다. 또한 정규식 패턴에 숫자 기호(#)가 있으면 이는 일치시킬 리터럴 문자로 해석됩니다.

RegexOptions.IgnorePatternWhitespace 옵션 또는 `x` 인라인 옵션은 이 기본 동작을 다음과 같이 변경합니다.

- 정규식 패턴에서 이스케이프되지 않은 공백은 무시됩니다. 정규식 패턴의 일부가 되려면 공백 문자가 이스케이프되어야 합니다(예: `\s` 또는 "`\ `"로).

- 숫자 기호(#)는 리터럴 문자가 아니라 주석의 시작 부분으로 해석됩니다. 정규식 패턴의 모든 텍스트는 `#` 문자에서 다음 `\n` 문자 또는 문자열 끝까지 주석으로 해석됩니다.

그러나 다음 경우에 정규식의 공백 문자는 RegexOptions.IgnorePatternWhitespace 옵션을 사용하더라도 무시되지 않습니다.

- 문자 클래스 내의 공백은 항상 리터럴로 해석됩니다. 예를 들어, 정규식 패턴 `[ .,;:]`는 모든 단일 공백 문자, 마침표, 쉼표, 세미콜론 또는 콜론과 일치합니다.

- 중괄호로 묶은 수량자 내에는 공백이 허용되지 않습니다(예: `{`*n*`}`, `{`*n*`,}` 및 `{`*n*`,`*m*`}`). 예를 들어, 정규식 패턴 `\d{1, 3}`는 공백 문자를 포함하고 있기 때문에 1자리에서 3자리까지의 숫자로 이루어진 어떠한 숫자 시퀀스와도 일치하지 않습니다.

- 언어 요소를 도입하는 문자 시퀀스 내에는 공백이 허용되지 않습니다. 예를 들어:

  - 언어 요소 `(?:`*subexpression*`)`는 비 캡처링 그룹을 나타내고, 요소의 `(?:` 부분은 공백을 포함할 수 없습니다. `(? :`*subexpression*`)` 패턴은 정규식 엔진이 패턴을

구문 분석할 수 없고 `( ?:subexpression )` 패턴이 *subexpression*과 일치하지 않으므로 런타임에 ArgumentException을 throw합니다.

- 유니코드 범주 또는 명명된 블록을 나타내는 언어 요소 `\p{name}`는 요소의 `\p{` 부분에 공백을 포함할 수 없습니다. 공백을 포함하는 경우 이 요소가 런타임에 ArgumentException을 throw합니다.

이 옵션을 사용하면 일반적으로 구문 분석 및 이해가 어려운 정규식을 단순화하는 데 도움이 됩니다. 이 옵션은 가독성을 향상시키고 정규식을 문서화할 수 있게 해줍니다.

다음 예제에서는 다음과 같은 정규식 패턴을 정의합니다.

```
\b \(? ( (?>\w+) ,?\s? )+ [\.!?] \)? # Matches an entire sentence.
```

이 패턴은 RegexOptions.IgnorePatternWhitespace 옵션을 사용하여 패턴 공백을 무시하는 점을 제외하고는 명시적 캡처만 해당 섹션에 정의된 패턴과 유사합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Whitespace1Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                       "of a literary masterpiece? I think not. Instead, " +
                       "it is a nonsensical paragraph.";
        string pattern = @"\b \(? ( (?>\w+) ,?\s? )+ [\.!?] \)? # Matches an
entire sentence.";

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnorePatternWhitespace))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//       This is the first sentence.
//       Is it the beginning of a literary masterpiece?
//       I think not.
//       Instead, it is a nonsensical paragraph.
```

다음 예제에서는 인라인 옵션 `(?x)`를 사용하여 패턴 공백을 무시합니다.

C#

```
using System;
using System.Text.RegularExpressions;
```

```
public class Whitespace2Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                       "of a literary masterpiece? I think not. Instead, " +
                       "it is a nonsensical paragraph.";
        string pattern = @"(?x)\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? #
Matches an entire sentence.";

        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//       This is the first sentence.
//       Is it the beginning of a literary masterpiece?
//       I think not.
//       Instead, it is a nonsensical paragraph.
```

오른쪽에서 왼쪽 모드

기본적으로 정규식 엔진은 왼쪽에서 오른쪽으로 검색합니다. RegexOptions.RightToLeft 옵션을 사용하여 검색 방향을 반대로 할 수 있습니다. 오른쪽에서 왼쪽으로의 검색은 문자열의 마지막 문자 위치에서 자동으로 시작됩니다. 와 같은 Regex.Match(String, Int32)시작 위치 매개 변수를 포함하는 패턴 일치 메서드의 경우 지정된 시작 위치는 검색을 시작할 가장 오른쪽 문자 위치의 인덱스입니다.

> ⓘ **참고**
>
> 오른쪽에서 왼쪽 패턴 모드는 **RegexOptions.RightToLeft** 값을 `options` 클래스 생성자 또는 정적 패턴 일치 메서드의 **Regex** 매개 변수에 제공해서만 사용할 수 있으며, 인라인 옵션으로는 사용할 수 없습니다.

예제

정규식 `\bb\w+\s` 은 문자 "b"로 시작하고 뒤에 공백 문자가 있는 두 개 이상의 문자가 있는 단어와 일치합니다. 다음 예제에서 입력 문자열은 하나 이상의 "b" 문자를 포함하는 세 단어로 구성됩니다. 첫 번째 및 두 번째 단어는 "b"로 시작하고 세 번째 단어는 "b"로 끝납니다. 오른쪽에서 왼쪽 검색 예제의 출력에서 알 수 있듯이 첫 번째 단어와 두 번째 단어만 정규식 패턴과 일치하며 두 번째 단어는 먼저 일치합니다.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class RTL1Example
{
    public static void Main()
    {
        string pattern = @"\bb\w+\s";
        string input = "build band tab";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.RightToLeft))
            Console.WriteLine("'{0}' found at position {1}.", match.Value,
match.Index);
    }
}
// The example displays the following output:
//       'band ' found at position 6.
//       'build ' found at position 0.
```

평가 순서

옵션은 RegexOptions.RightToLeft 검색 방향을 변경하고 정규식 패턴이 평가되는 순서도 반대로 바꿉니다. 오른쪽에서 왼쪽 검색에서는 오른쪽에서 왼쪽으로 검색 패턴을 읽습니다. 이러한 구분은 캡처 그룹 및 역참조와 같은 항목에 영향을 줄 수 있기 때문에 중요합니다. 예를 들어 식 `Regex.Match("abcabc", @"\1(abc)", RegexOptions.RightToLeft)` 은 일치 `abcabc`항목을 찾지만 왼쪽에서 오른쪽 검색(`Regex.Match("abcabc", @"\1(abc)", RegexOptions.None)`)에서는 일치 항목을 찾을 수 없습니다. `(abc)` 일치 항목을 찾을 수 있도록 번호가 매겨진 캡처 그룹 요소(`\1`)전에 요소를 평가해야 하기 때문입니다.

Lookahead 및 lookbehind 어설션

lookahead() 또는 lookbehind`(?<=subexpression)`(`(?=subexpression)`) 어설션에 대한 일치 위치는 오른쪽에서 왼쪽 검색에서 변경되지 않습니다. lookahead 어설션은 현재 일치 위치의 오른쪽을 찾습니다. lookbehind 어설션은 현재 일치 위치의 왼쪽을 찾습니다.

> **팁**
>
> 검색이 오른쪽에서 왼쪽인지 여부에 관계없이 lookbehinds는 현재 일치 위치에서 시작하는 오른쪽에서 왼쪽 검색을 사용하여 구현됩니다.

예를 들어, 정규식 `(?<=\d{1,2}\s)\w+,\s\d{4}`에서는 lookbehind 어설션을 사용하여 월 이름 앞에 있는 날짜에 대해 테스트합니다. 그런 다음 정규식은 월 및 연도를 일치시킵니다. lookahead 및 lookbehind 어설션에 대한 자세한 내용은 그룹화 구문을 참조하세요.

C#

```
using System;
using System.Text.RegularExpressions;

public class RTL2Example
{
    public static void Main()
    {
        string[] inputs = { "1 May, 1917", "June 16, 2003" };
        string pattern = @"(?<=\d{1,2}\s)\w+,\s\d{4}";

        foreach (string input in inputs)
        {
            Match match = Regex.Match(input, pattern,
RegexOptions.RightToLeft);
            if (match.Success)
                Console.WriteLine("The date occurs in {0}.", match.Value);
            else
                Console.WriteLine("{0} does not match.", input);
        }
    }
}

// The example displays the following output:
//       The date occurs in May, 1917.
//       June 16, 2003 does not match.
```

이 정규식 패턴은 다음 테이블과 같이 정의됩니다.

무늬	설명
`(?<=\d{1,2}\s)`	일치 항목의 시작 부분 앞에는 한 개 또는 두 개의 10진수와 그 뒤에 공백이 하나 있어야 합니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`,`	하나의 쉼표 문자와 일치합니다.
`\s`	공백 문자를 찾습니다.
`\d{4}`	네 개의 10진수를 찾습니다.

ECMAScript 일치 동작

기본적으로 정규식 엔진은 정규식 패턴을 입력 텍스트에 일치시킬 때 정식 동작을 사용합니다. 그러나 RegexOptions.ECMAScript 옵션을 지정하여 ECMAScript 일치 동작을 사용하도록 정규식 엔진에 지시할 수 있습니다.

> ⓘ **참고**
>
> ECMAScript와 호환되는 동작은 **RegexOptions.ECMAScript** 값을 `options` 클래스 생성자 또는 정적 패턴 일치 메서드의 **Regex** 매개 변수에 제공해서만 사용할 수 있으며, 인라인 옵션으로는 사용할 수 없습니다.

RegexOptions.ECMAScript 옵션은 RegexOptions.IgnoreCase 및 RegexOptions.Multiline 옵션과만 조합할 수 있습니다. 정규식에 다른 옵션을 사용하면 ArgumentOutOfRangeException이 발생합니다.

ECMAScript의 동작과 정식 정규식의 동작은 문자 클래스 구문, 자신을 참조하는 캡처링 그룹 및 8진수 대 역참조 해석의 세 가지 영역에서 다릅니다.

- 문자 클래스 구문. 정식 정규식은 유니코드를 지원하는 반면 ECMAScript는 지원하지 않으므로, ECMAScript의 문자 클래스의 구문이 더 제한되어 있으며 일부 문자 클래스 언어 요소는 다른 의미를 지닙니다. 예를 들어, ECMAScript는 유니코드 범주 또는 블록 요소 `\p` 및 `\P`와 같은 언어 요소를 지원하지 않습니다. 마찬가지로, 단어 문자와 일치하는 `\w` 요소는 ECMAScript를 사용할 경우 `[a-zA-Z_0-9]` 문자 클래스와 동일하고 정식 동작을 사용할 경우 `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]`와 동일합니다. 자세한 내용은 문자 클래스를 참조하세요.

  다음 예제에서는 정식 패턴 일치와 ECMAScript 패턴 일치 간의 차이점을 보여 줍니다. 이 예제에서는 뒤에 공백 문자가 있는 단어와 일치하는 정규식 `\b(\w+\s*)+`를 정의합니다. 입력은 두 개의 문자열로 구성되어 있는데, 한 문자열은 라틴 문자 집합을 사용하고 다른 문자열은 키릴 자모 문자 집합을 사용합니다. 출력에 표시된 것처럼, ECMAScript 일치를 사용하는 Regex.IsMatch(String, String, RegexOptions) 메서드에 대한 호출은 키릴 자모 단어와 일치하지 않는 반면 정식 일치를 사용하는 메서드 호출은 이러한 단어와 일치합니다.

```
C#

using System;
using System.Text.RegularExpressions;

public class EcmaScriptExample
{
    public static void Main()
    {
        string[] values = { "целый мир", "the whole world" };
        string pattern = @"\b(\w+\s*)+";
        foreach (var value in values)
        {
            Console.Write("Canonical matching: ");
            if (Regex.IsMatch(value, pattern))
                Console.WriteLine("'{0}' matches the pattern.", value);
```

```
            else
                Console.WriteLine("{0} does not match the pattern.",
value);

            Console.Write("ECMAScript matching: ");
            if (Regex.IsMatch(value, pattern, RegexOptions.ECMAScript))
                Console.WriteLine("'{0}' matches the pattern.", value);
            else
                Console.WriteLine("{0} does not match the pattern.",
value);
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       Canonical matching: 'целый мир' matches the pattern.
//       ECMAScript matching: целый мир does not match the pattern.
//
//       Canonical matching: 'the whole world' matches the pattern.
//       ECMAScript matching: 'the whole world' matches the pattern.
```

- 자신을 참조하는 캡처링 그룹. 자신에 대한 역참조가 있는 정규식 캡처 클래스는 각 캡처 반복으로 업데이트되어야 합니다. 다음 예제에서 보여 주는 것처럼, 이 기능은 정규식 `((a+)(\1) ?)+`가 ECMAScript를 사용할 경우에는 입력 문자열 " aa aaaa aaaaaa "과 일치하고 정식 일치를 사용할 경우에는 일치하지 않도록 합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class EcmaScript2Example
{
    static string pattern;

    public static void Main()
    {
        string input = "aa aaaa aaaaaa ";
        pattern = @"((a+)(\1) ?)+";

        // Match input using canonical matching.
        AnalyzeMatch(Regex.Match(input, pattern));

        // Match input using ECMAScript.
        AnalyzeMatch(Regex.Match(input, pattern,
RegexOptions.ECMAScript));
    }

    private static void AnalyzeMatch(Match m)
    {
        if (m.Success)
        {
```

```
            Console.WriteLine("'{0}' matches {1} at position {2}.",
                              pattern, m.Value, m.Index);
            int grpCtr = 0;
            foreach (Group grp in m.Groups)
            {
                Console.WriteLine("   {0}: '{1}'", grpCtr, grp.Value);
                grpCtr++;
                int capCtr = 0;
                foreach (Capture cap in grp.Captures)
                {
                    Console.WriteLine("      {0}: '{1}'", capCtr,
cap.Value);
                    capCtr++;
                }
            }
        }
        else
        {
            Console.WriteLine("No match found.");
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    No match found.
//
//    '((a+)(\1) ?)+' matches aa aaaa aaaaaa  at position 0.
//       0: 'aa aaaa aaaaaa '
//          0: 'aa aaaa aaaaaa '
//       1: 'aaaaaa '
//          0: 'aa '
//          1: 'aaaa '
//          2: 'aaaaaa '
//       2: 'aa'
//          0: 'aa'
//          1: 'aa'
//          2: 'aa'
//       3: 'aaaa '
//          0: ''
//          1: 'aa '
//          2: 'aaaa '
```

정규식은 다음 테이블과 같이 정의됩니다.

무늬	설명
(a+)	문자 "a"를 1개 이상 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다.
(\1)	첫 번째 캡처링 그룹에 의해 캡처된 부분 문자열을 찾습니다. 이 그룹은 세 번째 캡처링 그룹입니다.
?	0개 또는 1개의 공백 문자를 찾습니다.

무늬	설명
((a+)(\1)?)+	하나 이상의 "a" 문자 다음에 첫 번째 캡처링 그룹과 일치하는 문자열이 있고 그 다음에 0개 또는 1개의 공백 문자가 한 번 이상 나타나는 패턴을 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.

- 8진수 이스케이프와 역참조 간의 모호성 해결. 다음 테이블에는 정식 정규식과 ECMAScript 정규식에서 8진수 대 역참조 해석의 차이점이 요약되어 있습니다.

정규식	정식 동작	ECMAScript 동작
`\0` 뒤에 0-2자리의 8진수	8진수로 해석됩니다. 예를 들어, `\044`는 항상 8진수 값으로 해석되며 "$"를 의미합니다.	동일한 동작입니다.
`\` 뒤에 1-9의 숫자 한 개, 그 뒤에 추가 10진수가 없음	역참조로 해석됩니다. 예를 들어, `\9`는 9번째 캡처링 그룹이 없더라도 항상 역참조 9를 의미합니다. 캡처링 그룹이 없는 경우 정규식 파서는 ArgumentException을 throw합니다.	한 자리 10진수 캡처링 그룹이 있는 경우 해당 숫자를 역참조합니다. 그러지 않으면 값이 리터럴로 해석됩니다.
`\` 뒤에 1-9의 숫자 한 개, 그 뒤에 추가 10진수가 있음	숫자가 10진수 값으로 해석됩니다. 해당 캡처링 그룹이 있는 경우 식이 역참조로 해석됩니다. 그러지 않으면 최대 8진수 377까지 선행 8진수가 해석됩니다. 즉, 값의 낮은 8비트만 고려합니다. 나머지 숫자는 리터럴로 해석됩니다. 예를 들어, `\3000` 식에서 캡처링 그룹 300이 있는 경우 역참조 300으로 해석됩니다. 캡처링 그룹 300이 없는 경우 8진수 300 뒤에 0이 있는 것으로 해석됩니다.	가능한 한 많은 숫자를 캡처를 참조할 수 있는 10진수 값으로 변환하여 역참조로 해석됩니다. 숫자를 변환할 수 없는 경우 최대 8진수 377까지 선행 8진수를 사용하여 8진수로 해석됩니다. 나머지 숫자는 리터럴로 해석됩니다.

고정 문화권 사용 비교

기본적으로 정규식 엔진은 대/소문자를 구분하지 않는 비교를 수행할 때 현재 문화권의 대/소문자 사용 규칙을 사용하여 동일한 대문자와 소문자를 확인합니다.

그러나 이 동작은 일부 비교 형식에는 바람직하지 않은데, 특히 사용자 입력을 시스템 리소스의 이름과 비교할 때 그렇습니다(예: 암호, 파일 또는 URL). 다음 예제에서는 이러한 시나리오를 보여 줍니다. 이 코드는 URL 앞에 **FILE://** 가 있는 모든 리소스에 대한 액세스를 차단하도록 작성되었습니다. 이 정규식에서는 정규식 `$FILE://`를 사용하여 문자열에 대해 대/소문자를 구분하지 않는 일치를 시도합니다. 그러나 현재 시스템 문화권이 tr-TR(터키-Türkiye)인 경우 "I"는 "i"와 같은 대문자입니다. 따라서 Regex.IsMatch 메서드에 대한 호출에서 `false`를 반환하고 파일에 대한 액세스가 허용됩니다.

C#

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-sensitive matching ({0} culture)...",
                  Thread.CurrentThread.CurrentCulture.Name);
if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
    Console.WriteLine("URLs that access files are not allowed.");
else
    Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//       Culture-sensitive matching (tr-TR culture)...
//       Access to file://c:/Documents.MyReport.doc is allowed.
```

> ⓘ **참고**
>
> 대/소문자를 구분하고 고정 문화권을 사용하는 문자열 비교에 대한 자세한 내용은 **문자열 사용에 대한 유용한 정보**를 참조하세요.

현재 문화권의 대/소문자를 구분하지 않는 비교를 사용하는 대신 RegexOptions.CultureInvariant 옵션을 지정하여 언어의 문화적 차이를 무시하고 고정 문화권의 규칙을 사용할 수 있습니다.

> ⓘ **참고**
>
> 고정 문화권을 사용하는 비교는 RegexOptions.CultureInvariant 값을 `options` 클래스 생성자 또는 정적 패턴 일치 메서드의 Regex 매개 변수에 제공해서만 사용할 수 있으며, 인라인 옵션으로는 사용할 수 없습니다.

다음 예제는 Regex.IsMatch(String, String, RegexOptions)를 포함하는 옵션을 사용하여 정적 RegexOptions.CultureInvariant 메서드를 호출한다는 점을 제외하고 이전 예제와 동일합니다. 현재 문화권이 터키어(Türkiye)로 설정된 경우에도 정규식 엔진은 "FILE" 및 "file"을 성공적으로 일치시키고 파일 리소스에 대한 액세스를 차단할 수 있습니다.

C#

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-insensitive matching...");
if (Regex.IsMatch(input, pattern,
                  RegexOptions.IgnoreCase | RegexOptions.CultureInvariant))
    Console.WriteLine("URLs that access files are not allowed.");
else
    Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//       Culture-insensitive matching...
//       URLs that access files are not allowed.
```

역추적 모드

기본적으로 입니다. NET의 regex 엔진은 *역추적을* 사용하여 패턴 일치 항목을 찾습니다. 역추적 엔진은 하나의 패턴을 일치시키려고 시도하는 엔진이며, 실패하면 뒤로 돌아가서 대체 패턴과 일치시키려고 시도합니다. 역추적 엔진은 일반적인 경우 매우 빠르지만 패턴 변경 수가 증가함에 따라 속도가 느려지며, 이로 인해 *치명적인 역추적*이 발생할 수 있습니다. 옵션은 RegexOptions.NonBacktracking 역추적을 사용하지 않으며 최악의 시나리오를 방지합니다. 검색되는 입력에 관계없이 일관되게 좋은 동작을 제공하는 것이 목표입니다.

옵션은 RegexOptions.NonBacktracking 다른 기본 제공 엔진에서 지원하는 모든 것을 지원하지 않습니다. 특히 옵션은 또는 RegexOptions.ECMAScript와 함께 RegexOptions.RightToLeft 사용할 수 없습니다. 또한 패턴에서 다음 구문을 허용하지 않습니다.

- 원자성 그룹
- 역참조
- 그룹 분산
- 조건
- 해결 방법

- 앵커 시작(`\G`)

RegexOptions.NonBacktracking 실행과 관련하여 미묘한 차이가 있습니다. 캡처 그룹이 루프에 있는 경우 대부분의(non-.NET) regex 엔진은 해당 캡처에 대해 마지막으로 일치하는 값만 제공합니다. 그러나. NET의 정규식 엔진은 루프 내에서 캡처되는 모든 값을 추적하고 액세스 권한을 제공합니다. 옵션은 RegexOptions.NonBacktracking 대부분의 다른 regex 구현과 유사하며 최종 캡처 제공만 지원합니다.

참조

- 정규식 언어 - 빠른 참조

정규 표현식의 기타 구성 요소

2025. 06. 17.

.NET의 정규식에는 세 가지 기타 언어 구문이 포함됩니다. 사용자는 정규 표현식의 중간에서 특정 매칭 옵션을 사용하거나 사용하지 않도록 설정할 수 있습니다. 남은 두 개의 기능은 정규 표현식에 주석을 포함할 수 있도록 합니다.

인라인 옵션

구문을 사용하여 정규식의 일부에 대해 특정 패턴 일치 옵션을 사용하거나 사용하지 않도록 설정할 수 있습니다.

`(?imnsx-imnsx)`

물음표 뒤에 사용할 옵션을 나열하고, 빼기 기호 뒤에 사용하지 않을 옵션을 나열합니다. 다음 표는 각 옵션에 대해 설명합니다. 각 옵션에 대한 자세한 내용은 정규식 옵션을 참조하세요.

테이블 확장

옵션	설명
`i`	대/소문자를 구분하지 않는 일치
`m`	여러 줄 모드입니다.
`n`	명시적 캡처만 허용됩니다. 괄호는 캡처링 그룹으로 동작하지 않습니다.
`s`	한 줄 모드입니다.
`x`	이스케이프되지 않은 공백을 무시하고 x-모드 주석을 허용합니다.

`(?imnsx-imnsx)` 구문에서 정의한 정규식 옵션의 변경 내용은 포함 그룹의 끝까지 계속 적용됩니다.

> **참고**
>
> `(?imnsx-imnsx:` *subexpression* `)` 그룹화 구문은 하위 식에 대해 동일한 기능을 제공합니다. 자세한 내용은 **그룹화 구문**을 참조하세요.

다음 예제에서는 `i`, `n` 및 `x` 옵션을 통해 대/소문자 구분 안 함 및 명시적 캡처를 사용하고 정규식 중간의 정규식 패턴에 있는 공백을 무시합니다.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern;
      string input = "double dare double Double a Drooling dog The Dreaded Deep";

      pattern = @"\b(D\w+)\s(d\w+)\b";
      // Match pattern using default options.
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
            for (int ctr = 1; ctr < match.Groups.Count; ctr++)
               Console.WriteLine($"   Group {ctr}: {match.Groups[ctr].Value}");
      }
      Console.WriteLine();

      // Change regular expression pattern to include options.
      pattern = @"\b(D\w+)(?ixn) \s (d\w+) \b";
      // Match new pattern with options.
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
            for (int ctr = 1; ctr < match.Groups.Count; ctr++)
               Console.WriteLine($"   Group {ctr}: '{match.Groups[ctr].Value}'");
      }
   }
}
// The example displays the following output:
//       Drooling dog
//          Group 1: Drooling
//          Group 2: dog
//
//       Drooling dog
//          Group 1: 'Drooling'
//       Dreaded Deep
//          Group 1: 'Dreaded'
```

이 예제에서는 두 개의 정규식을 정의합니다. 첫 번째 `\b(D\w+)\s(d\w+)\b` 정규식은 대문자 "D"와 소문자 "d"로 시작하는 두 개의 연속 단어와 일치합니다. 두 번째 `\b(D\w+)(?ixn) \s (d\w+) \b` 정규식은 다음 표에 설명된 대로 인라인 옵션을 사용하여 이 패턴을 수정합니다. 결과를 비교하면 `(?ixn)` 구문의 효과를 확인할 수 있습니다.

패턴	설명
`\b`	단어 경계에서 시작하십시오.
`(D\w+)`	대문자 "D" 뒤에 오는 하나 이상의 단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처 그룹입니다.
`(?ixn)`	이때부터 대/소문자를 구분하지 않고 비교하고, 명시적 캡처만 수행되며, 정규식 패턴의 공백이 무시됩니다.
`\s`	공백 문자를 일치시킵니다.
`(d\w+)`	대문자 또는 소문자 "d" 다음에 하나 이상의 단어 문자로 이루어진 부분을 찾습니다. `n` (명시적 캡처) 옵션이 사용되었으므로 이 그룹은 캡처되지 않습니다.
`\b`	단어 경계를 찾습니다.

인라인 주석

`(?#` *comment* `)` 구문을 통해 정규식에 인라인 주석을 포함할 수 있습니다. Regex.ToString 메서드에서 반환하는 문자열에 주석이 포함되어 있어도 정규식 엔진은 패턴 일치에 주석의 어떤 부분도 사용하지 않습니다. 주석이 첫 번째 닫는 괄호 문자에서 끝납니다.

다음 예제에서는 이전 섹션의 예제에서 사용된 첫 번째 정규식 패턴을 반복합니다. 정규 표현식에 두 개의 인라인 주석을 추가하여 문자열 비교가 대소문자 구분 여부를 나타냅니다. 정규식 패턴 `\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comparison)d\w+)\b` 는 다음과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 시작하십시오.
`(?# case-sensitive comparison)`	댓글. 패턴 일치 동작에 영향을 주지 않습니다.
`(D\w+)`	대문자 "D" 뒤에 오는 하나 이상의 단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처 그룹입니다.
`\s`	공백 문자를 일치시킵니다.
`(?ixn)`	이때부터 대/소문자를 구분하지 않고 비교하고, 명시적 캡처만 수행되며, 정규식 패턴의 공백이 무시됩니다.
`(?#case-insensitive comparison)`	댓글. 패턴 일치 동작에 영향을 주지 않습니다.

패턴	설명
`(d\w+)`	대문자 또는 소문자 "d" 다음에 하나 이상의 단어 문자로 이루어진 부분을 찾습니다. 두 번째 캡처 그룹입니다.
`\b`	단어 경계를 찾습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-
insensitive comparison)d\w+)\b";
      Regex rgx = new Regex(pattern);
      string input = "double dare double Double a Drooling dog The Dreaded Deep";

      Console.WriteLine("Pattern: " + pattern.ToString());
      // Match pattern using default options.
      foreach (Match match in rgx.Matches(input))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
         {
            for (int ctr = 1; ctr <match.Groups.Count; ctr++)
               Console.WriteLine($"   Group {ctr}: {match.Groups[ctr].Value}");
         }
      }
   }
}
// The example displays the following output:
//    Pattern: \b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive
comp
//    arison)d\w+)\b
//    Drooling dog
//       Group 1: Drooling
//    Dreaded Deep
//       Group 1: Dreaded
```

라인 끝 주석

숫자 기호(`#`)는 정규식 패턴의 끝에 있는 이스케이프되지 않은 # 문자에서 시작하고 줄의 끝까지 계속되는 x-mode 주석을 표시합니다. 이 구문을 사용하려면 인라인 옵션을 통해 `x` 옵션을 사용하도록 설정하거나, RegexOptions.IgnorePatternWhitespace 개체를 인스턴스화하거나 정적 `option` 메서드를 호출할 때 Regex 값을 Regex 매개 변수에 제공해야 합니다.

다음 예제에서는 줄의 끝 주석 구문을 보여 줍니다. 문자열이 하나 이상의 형식 항목을 포함하는 복합 형식 문자열인지 여부를 결정합니다. 다음 표에서는 정규식 패턴의 구문을 설명합니다.

`\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item.`

테이블 확장

패턴	설명
`\{`	여는 중괄호를 맞춥니다.
`\d+`	하나 이상의 10진수 숫자가 일치하는지 확인합니다.
`(,-*\d+)*`	쉼표가 0번 또는 한 번, 그 다음 선택적 빼기 기호, 그리고 하나 이상의 10진수를 찾습니다.
`(\:\w{1,4}?)*`	콜론이 0번 혹은 1번 나타난 후, 1개에서 4개의 공백 문자가 최대한 적게 포함되도록 일치시킵니다.
`\}`	닫는 중괄호와 일치시킵니다.
`(?x)`	줄의 끝 주석이 인식되도록 패턴 공백 무시 옵션을 사용하도록 설정합니다.
`# Looks for a composite format item.`	줄의 끝 주석입니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite
format item.";
      string input = "{0,-3:F}";
      Console.WriteLine($"'{input}':");
      if (Regex.IsMatch(input, pattern))
         Console.WriteLine("   contains a composite format item.");
      else
         Console.WriteLine("   does not contain a composite format item.");
   }
}
// The example displays the following output:
//       '{0,-3:F}':
//          contains a composite format item.
```

정규식에서 `(?x)` 구문을 제공하는 대신 Regex.IsMatch(String, String, RegexOptions) 메서드를 호출하여 RegexOptions.IgnorePatternWhitespace 열거형 값을 전달해도 주석이 인식될 수 있습니다.

더 보기

- 정규식 언어 - 빠른 참조

Regular Expression 개체 모델

아티클 • 2023. 04. 08.

이 항목에서는 .NET 정규식 작업을 수행하는 데 사용되는 개체 모델을 설명합니다. 여기에는 다음 단원이 포함되어 있습니다.

- 정규식 엔진
- MatchCollection 및 Match 개체
- 그룹 컬렉션
- 캡처된 그룹
- 캡처 컬렉션
- 개별 캡처

정규식 엔진

.NET의 정규식 엔진은 Regex 클래스로 표현됩니다. 정규식 엔진은 정규식을 구문 분석 및 컴파일하고, 정규식 패턴을 입력 문자열과 일치시키는 작업 수행을 담당합니다. 이 엔진은 .NET 정규식 개체 모델의 중심 구성 요소입니다.

정규식 엔진은 다음과 같은 두 가지 방법 중 하나로 사용할 수 있습니다.

- Regex 클래스의 정적 메서드를 호출합니다. 메서드 매개 변수에는 입력 문자열과 정규식 패턴이 포함됩니다. 정규식 엔진은 정적 메서드 호출에 사용되는 정규식을 캐시하므로 동일한 정규식을 사용하는 정적 정규식 메서드에 대한 반복 호출은 상대적으로 좋은 성능을 제공합니다.

- 정규식을 클래스 생성자에 전달하여 Regex 개체를 인스턴스화합니다. 이 경우 Regex 개체는 변경 불가능하며(읽기 전용), 단일 정규식과 강력하게 결합된 정규식 엔진을 나타냅니다. Regex 인스턴스에 사용되는 정규식은 캐시되지 않으므로 동일한 정규식으로 Regex 개체를 여러 번 인스턴스화해서는 안 됩니다.

Regex 클래스의 메서드를 호출하여 다음과 같은 작업을 수행할 수 있습니다.

- 문자열이 정규식 패턴과 일치하는지 확인합니다.
- 단일 일치 항목 또는 첫 번째 일치 항목을 추출합니다.
- 모든 일치 항목을 추출합니다.

- 일치하는 부분 문자열을 바꿉니다.
- 단일 문자열을 문자열 배열로 분할합니다.

이러한 작업은 다음 섹션에 설명되어 있습니다.

정규식 패턴 일치

Regex.IsMatch 메서드는 문자열이 패턴과 일치할 경우 `true`를 반환하고, 그렇지 않을 경우 `false`를 반환합니다. IsMatch 메서드는 일반적으로 문자열 입력의 유효성을 검사하는 데 사용됩니다. 예를 들어, 다음 코드는 문자열이 미국의 유효한 사회 보장 번호와 일치하도록 합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] values = { "111-22-3333", "111-2-3333"};
      string pattern = @"^\d{3}-\d{2}-\d{4}$";
      foreach (string value in values) {
         if (Regex.IsMatch(value, pattern))
            Console.WriteLine("{0} is a valid SSN.", value);
         else
            Console.WriteLine("{0}: Invalid", value);
      }
   }
}
// The example displays the following output:
//       111-22-3333 is a valid SSN.
//       111-2-3333: Invalid
```

정규식 패턴 `^\d{3}-\d{2}-\d{4}$` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`^`	입력 문자열의 시작 부분을 찾습니다.
`\d{3}`	세 개의 10진수를 찾습니다.
`-`	하이픈을 찾습니다.
`\d{2}`	두 개의 10진수를 찾습니다.

무늬	설명
`-`	하이픈을 찾습니다.
`\d{4}`	네 개의 10진수를 찾습니다.
`$`	입력 문자열의 끝 부분을 찾습니다.

단일 일치 항목 또는 첫 번째 일치 항목 추출

Regex.Match 메서드는 정규식 패턴과 일치하는 첫 번째 부분 문자열에 대한 정보가 포함된 Match 개체를 반환합니다. `Match.Success` 속성이 일치 항목을 찾았음을 나타내는 `true`를 반환하는 경우 Match.NextMatch 메서드를 호출하여 후속 일치 항목에 대한 정보를 검색할 수 있습니다. 이러한 메서드 호출은 `Match.Success` 속성이 `false`를 반환할 때까지 계속될 수 있습니다. 예를 들어, 다음 코드에서는 Regex.Match(String, String) 메서드를 사용하여 문자열에서 첫 번째 중복된 단어를 찾습니다. 그런 다음 Match.NextMatch 메서드를 호출하여 추가 중복된 단어를 찾습니다. 이 예제에서는 현재 찾기가 성공했는지와 `Match.Success` 메서드에 대한 호출이 뒤따라야 하는지를 확인하기 위해 메서드를 호출한 후마다 Match.NextMatch 속성을 검토합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is a a farm that that raises dairy cattle.";
      string pattern = @"\b(\w+)\W+(\1)\b";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine("Duplicate '{0}' found at position {1}.",
                           match.Groups[1].Value, match.Groups[2].Index);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       Duplicate 'a' found at position 10.
//       Duplicate 'that' found at position 22.
```

정규식 패턴 `\b(\w+)\W+(\1)\b` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(\w+)`	하나 이상의 단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\W+`	하나 이상의 단어가 아닌 문자를 찾습니다.
`(\1)`	캡처된 첫 번째 문자열을 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

모든 일치 항목 추출

Regex.Matches 메서드는 정규식 엔진이 입력 문자열에서 찾은 모든 일치 항목에 대한 정보가 포함된 MatchCollection 개체를 반환합니다. 예를 들어, 이전 예제를 Matches 및 Match 메서드 대신 NextMatch 메서드를 호출하도록 다시 작성할 수 있습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is a a farm that that raises dairy cattle.";
      string pattern = @"\b(\w+)\W+(\1)\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Duplicate '{0}' found at position {1}.",
                           match.Groups[1].Value, match.Groups[2].Index);
   }
}
// The example displays the following output:
//       Duplicate 'a' found at position 10.
//       Duplicate 'that' found at position 22.
```

일치하는 부분 문자열 바꾸기

Regex.Replace 메서드는 정규식 패턴과 일치하는 각 부분 문자열을 지정된 문자열 또는 정규식 패턴으로 바꾸고, 바뀐 내용이 포함된 전체 입력 문자열을 반환합니다. 예를 들어, 다음 코드는 문자열에서 10진수 앞에 미국 통화 기호를 추가합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\d+\.\d{2}\b";
      string replacement = "$$$&";
      string input = "Total Cost: 103.64";
      Console.WriteLine(Regex.Replace(input, pattern, replacement));
   }
}
// The example displays the following output:
//       Total Cost: $103.64
```

정규식 패턴 `\b\d+\.\d{2}\b` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`\d+`	하나 이상의 10진수 숫자가 일치하는지 확인합니다.
`\.`	마침표를 찾습니다.
`\d{2}`	두 개의 10진수를 찾습니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

바꾸기 패턴 `$$$&`는 다음 테이블과 같이 해석됩니다.

무늬	바꾸기 문자열
`$$`	달러 기호($) 문자
`$&`	일치하는 전제 부분 문자열

단일 문자열을 문자열 배열로 분할

Regex.Split 메서드는 정규식 일치에 의해 정의된 위치에서 입력 문자열을 분할합니다. 예를 들어, 다음 코드는 번호 매기기 목록의 항목을 문자열 배열에 배치합니다.

C#

```
using System;
using System.Text.RegularExpressions;
```

```
public class Example
{
   public static void Main()
   {
      string input = "1. Eggs 2. Bread 3. Milk 4. Coffee 5. Tea";
      string pattern = @"\b\d{1,2}\.\s";
      foreach (string item in Regex.Split(input, pattern))
      {
         if (! String.IsNullOrEmpty(item))
            Console.WriteLine(item);
      }
   }
}
// The example displays the following output:
//       Eggs
//       Bread
//       Milk
//       Coffee
//       Tea
```

정규식 패턴 `\b\d{1,2}\.\s` 는 다음 테이블과 같이 해석됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`\d{1,2}`	한 개 또는 두 개의 10진수를 찾습니다.
`\.`	마침표를 찾습니다.
`\s`	공백 문자를 찾습니다.

MatchCollection 및 Match 개체

정규식 메서드가 정규식 개체 모델의 일부인 두 개체 MatchCollection 개체 및 Match 개체를 반환합니다.

일치 컬렉션

Regex.Matches 메서드는 정규식 엔진이 입력 문자열에서 찾은 모든 일치 항목(입력 문자열에서 나타나는 순서대로)을 나타내는 MatchCollection 개체가 포함된 Match 개체를 반환합니다. 일치 항목이 없는 경우 이 메서드는 멤버 없이 MatchCollection 개체를 반환합니다. MatchCollection.Item[] 속성을 사용하여 인덱스(0에서 MatchCollection.Count 속성 값보다 1 작은 값까지)별로 컬렉션의 개별 멤버에 액세스할 수 있습니다. Item[]은 컬렉션의 인덱서(C#의 경우) 및 기본 속성(Visual Basic의 경우)입니다.

기본적으로 Regex.Matches 메서드에 대한 호출에서는 지연 평가를 사용하여 MatchCollection 개체를 채웁니다. 완전히 채워진 컬렉션이 필요한 속성(예: MatchCollection.Count 및 MatchCollection.Item[] 속성)에 액세스할 경우 성능이 저하될 수 있습니다. 따라서 IEnumerator 메서드에서 반환하는 MatchCollection.GetEnumerator 개체를 사용하여 컬렉션에 액세스하는 것이 좋습니다. 개별 언어는 컬렉션의 IEnumerator 인터페이스를 래핑하는 생성자(예: Visual Basic의 경우 `For Each` 및 C#의 경우 `foreach`)를 제공합니다.

다음 예제에서는 Regex.Matches(String) 메서드를 사용하여, 입력 문자열에서 찾은 모든 일치 항목으로 MatchCollection 개체를 채웁니다. 이 예제에서는 컬렉션을 열거하고 문자열 배열에 일치 항목을 복사하며 정수 배열에 문자 위치를 기록합니다.

C#

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
       MatchCollection matches;
       List<string> results = new List<string>();
       List<int> matchposition = new List<int>();

       // Create a new Regex object and define the regular expression.
       Regex r = new Regex("abc");
       // Use the Matches method to find all matches in the input string.
       matches = r.Matches("123abc4abcd");
       // Enumerate the collection to retrieve all matches and positions.
       foreach (Match match in matches)
       {
          // Add the match string to the string array.
           results.Add(match.Value);
           // Record the character position where the match was found.
           matchposition.Add(match.Index);
       }
       // List the results.
       for (int ctr = 0; ctr < results.Count; ctr++)
         Console.WriteLine("'{0}' found at position {1}.",
                           results[ctr], matchposition[ctr]);
   }
}
// The example displays the following output:
//       'abc' found at position 3.
//       'abc' found at position 7.
```

일치

Match 클래스는 단일 정규식 일치의 결과를 나타냅니다. 다음과 같은 두 가지 방법으로 Match 개체에 액세스할 수 있습니다.

- MatchCollection 메서드에서 반환하는 Regex.Matches 개체에서 해당 개체를 검색합니다. 개별 Match 개체를 검색하려면 `foreach`(C#의 경우) 또는 `For Each`...`Next`(Visual Basic의 경우) 생성자를 사용하여 컬렉션을 반복하거나, MatchCollection.Item[] 속성을 사용하여 인덱스 또는 이름으로 특정 Match 개체를 검색합니다. 또한 인덱스(0에서 컬렉션의 개체 수보다 1 작은 값까지)로 컬렉션을 반복하여 컬렉션에서 개별 Match 개체를 검색할 수 있습니다. 그러나 이 메서드는 MatchCollection.Count 속성에 액세스하므로 지연 평가를 사용하지 않습니다.

  다음 예제에서는 Match 또는 MatchCollection...`foreach` 생성자를 사용하여 컬렉션을 반복함으로써 `For Each` 개체에서 개별 `Next` 개체를 검색합니다. 정규식은 단순히 입력 문자열에서 문자열 "abc"를 찾습니다.

  C#

  ```
  using System;
  using System.Text.RegularExpressions;

  public class Example
  {
     public static void Main()
     {
        string pattern = "abc";
        string input = "abc123abc456abc789";
        foreach (Match match in Regex.Matches(input, pattern))
           Console.WriteLine("{0} found at position {1}.",
                             match.Value, match.Index);
     }
  }
  // The example displays the following output:
  //       abc found at position 0.
  //       abc found at position 6.
  //       abc found at position 12.
  ```

- 문자열 또는 문자열 일부에서 첫 번째 일치 항목을 나타내는 Regex.Match 개체를 반환하는 Match 메서드를 호출합니다. `Match.Success` 속성 값을 검색하여 일치 항목을 찾았는지를 확인할 수 있습니다. 후속 일치 항목을 나타내는 Match 개체를 검색하려면 반환된 Match.NextMatch 개체의 `Success` 속성이 Match일 때까지 `false` 메서드를 반복적으로 호출합니다.

  다음 예제에서는 Regex.Match(String, String) 및 Match.NextMatch 메서드를 사용하여 입력 문자열에서 문자열 "abc"를 찾습니다.

  C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "abc";
      string input = "abc123abc456abc789";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine("{0} found at position {1}.",
                           match.Value, match.Index);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       abc found at position 0.
//       abc found at position 6.
//       abc found at position 12.
```

Match 클래스의 두 속성은 컬렉션 개체를 반환합니다.

- Match.Groups 속성은 정규식 패턴에서 캡처링 그룹과 일치하는 부분 문자열에 대한 정보가 포함된 GroupCollection 개체를 반환합니다.

- `Match.Captures` 속성은 사용이 제한된 CaptureCollection 개체를 반환합니다. 컬렉션은 Match 속성이 `Success`인 `false` 개체에 대해서는 채워지지 않습니다. 그렇지 않은 경우 이 컬렉션은 Capture 개체와 동일한 정보를 가진 단일 Match 개체를 포함합니다.

이러한 개체에 대한 자세한 내용은 이 항목 뒷부분의 그룹 컬렉션 및 캡처 컬렉션 섹션을 참조하세요.

Match 클래스의 두 추가 속성은 일치 항목에 대한 정보를 제공합니다. `Match.Value` 속성은 입력 문자열에서 정규식 패턴과 일치하는 부분 문자열을 반환합니다. `Match.Index` 속성은 입력 문자열에서 일치하는 문자열의 0부터 시작하는 시작 위치를 반환합니다.

또한 Match 클래스에는 두 개의 패턴 일치 메서드가 포함되어 있습니다.

- Match.NextMatch 메서드는 현재 Match 개체가 나타내는 일치 항목 뒤에서 일치 항목을 찾아 해당 일치 항목을 나타내는 Match 개체를 반환합니다.

- Match.Result 메서드는 일치하는 문자열에 대해 지정된 바꾸기 작업을 수행하고 결과를 반환합니다.

다음 예제에서는 Match.Result 메서드를 사용하여, 두 소수 자릿수를 포함하는 모든 숫자 앞에 $ 기호와 공백 하나를 추가합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\d+(,\d{3})*\.\d{2}\b";
      string input = "16.32\n194.03\n1,903,672.08";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Result("$$ $&"));
   }
}
// The example displays the following output:
//       $ 16.32
//       $ 194.03
//       $ 1,903,672.08
```

정규식 패턴 `\b\d+(,\d{3})*\.\d{2}\b` 는 다음 테이블과 같이 정의됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`\d+`	하나 이상의 10진수 숫자가 일치하는지 확인합니다.
`(,\d{3})*`	쉼표 하나 다음에 세 개의 10진수가 있는 0개 이상의 일치 항목을 찾습니다.
`\.`	소수점 문자를 찾습니다.
`\d{2}`	두 개의 10진수를 찾습니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

바꾸기 패턴 `$$ $&`는 일치하는 부분 문자열이 달러 기호($)(`$$` 패턴), 공백 하나 및 일치 항목 값(`$&` 패턴)으로 바뀌어야 함을 나타냅니다.

맨 위로 이동

그룹 컬렉션

Match.Groups 속성은 단일 일치 항목에서 캡처된 그룹을 나타내는 GroupCollection 개체가 포함된 Group 개체를 반환합니다. 컬렉션의 첫 번째 Group 개체(인덱스 0에 있음)는 전체 일치를 나타냅니다. 뒤에 나오는 각 개체는 단일 캡처링 그룹의 결과를 나타냅니다.

Group 속성을 사용하여 컬렉션에서 개별 GroupCollection.Item[] 개체를 검색할 수 있습니다. 명명되지 않은 그룹은 컬렉션에서 위치로 검색하고, 명명된 그룹은 이름 또는 위치로 검색할 수 있습니다. 명명되지 않은 캡처는 컬렉션에서 맨 앞에 나타나고, 정규식 패턴에서 나타나는 순서대로 왼쪽에서 오른쪽으로 인덱싱됩니다. 명명된 캡처는 명명되지 않은 캡처 다음에, 정규식 패턴에서 나타나는 순서대로 왼쪽에서 오른쪽으로 인덱싱됩니다. 특정 정규식 일치 메서드에 대해 반환된 컬렉션에서 사용 가능한 번호가 매겨진 그룹을 확인하려면 인스턴스 Regex.GetGroupNumbers 메서드를 호출하면 됩니다. 컬렉션에서 사용 가능한 명명된 그룹을 확인하려면 인스턴스 Regex.GetGroupNames 메서드를 호출하면 됩니다. 두 메서드는 모두 정규식으로 찾은 일치 항목을 분석하는 범용 루틴에서 특히 유용합니다.

GroupCollection.Item[] 속성은 컬렉션의 인덱스(C#의 경우) 및 컬렉션 개체의 기본 속성(Visual Basic의 경우)입니다. 즉, 개별 Group 개체를 다음과 같이 인덱스로(또는 명명된 그룹의 경우 이름으로) 액세스할 수 있습니다.

C#

```
Group group = match.Groups[ctr];
```

다음 예제에서는 그룹화 구문을 사용하여 날짜의 월, 일 및 연도를 캡처하는 정규식을 정의합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s(\d{1,2}),\s(\d{4})\b";
      string input = "Born: July 28, 1989";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
         for (int ctr = 0; ctr <  match.Groups.Count; ctr++)
            Console.WriteLine("Group {0}: {1}", ctr,
match.Groups[ctr].Value);
    }
}
// The example displays the following output:
//       Group 0: July 28, 1989
//       Group 1: July
```

```
//       Group 2: 28
//       Group 3: 1989
```

정규식 패턴 `\b(\w+)\s(\d{1,2}),\s(\d{4})\b` 는 다음 테이블과 같이 정의됩니다.

무늬	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(\w+)`	하나 이상의 단어 문자를 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`\s`	공백 문자를 찾습니다.
`(\d{1,2})`	한 개 또는 두 개의 10진수를 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`,`	쉼표 하나를 찾습니다.
`\s`	공백 문자를 찾습니다.
`(\d{4})`	네 개의 10진수를 찾습니다. 이 그룹은 세 번째 캡처링 그룹입니다.
`\b`	단어 경계에서 일치 항목 찾기를 끝냅니다.

맨 위로 이동

캡처된 그룹

Group 클래스는 단일 캡처링 그룹의 결과를 나타냅니다. 정규식에 정의된 캡처링 그룹을 나타내는 그룹 개체는 Item[] 속성에서 반환하는 GroupCollection 개체의 Match.Groups 속성에서 반환합니다. Item[] 속성은 Group 클래스의 인덱스(C#의 경우) 및 기본 속성(Visual Basic의 경우)입니다. 또한 `foreach` 또는 `For Each` 구문을 사용하여 컬렉션을 반복함으로써 개별 멤버를 검색할 수 있습니다. 예제는 이전 섹션을 참조하세요.

다음 예제에서는 중첩된 그룹화 구문을 사용하여 부분 문자열을 그룹으로 캡처합니다. 정규식 패턴 `(a(b))c`는 문자열 "abc"와 일치합니다. 이 패턴은 부분 문자열 "ab"를 첫 번째 캡처링 그룹에 할당하고 부분 문자열 "b"를 두 번째 캡처링 그룹에 할당합니다.

C#

```
var matchposition = new List<int>();
var results = new List<string>();
// Define substrings abc, ab, b.
var r = new Regex("(a(b))c");
Match m = r.Match("abdabc");
for (int i = 0; m.Groups[i].Value != ""; i++)
{
    // Add groups to string array.
```

```
    results.Add(m.Groups[i].Value);
    // Record character position.
    matchposition.Add(m.Groups[i].Index);
}

// Display the capture groups.
for (int ctr = 0; ctr < results.Count; ctr++)
    Console.WriteLine("{0} at position {1}",
                      results[ctr], matchposition[ctr]);
// The example displays the following output:
//       abc at position 3
//       ab at position 3
//       b at position 4
```

다음 예제에서는 명명된 그룹화 구문을 사용하여 문자열에서 정규식을 콜론(:)에서 분할하는 "DATANAME:VALUE" 형식의 데이터가 포함된 부분 문자열을 캡처합니다.

C#

```
var r = new Regex(@"^(?<name>\w+):(?<value>\w+)");
Match m = r.Match("Section1:119900");
Console.WriteLine(m.Groups["name"].Value);
Console.WriteLine(m.Groups["value"].Value);
// The example displays the following output:
//       Section1
//       119900
```

정규식 패턴 `^(?<name>\w+):(?<value>\w+)` 는 다음 테이블과 같이 정의됩니다.

무늬	설명
`^`	입력 문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
`(?<name>\w+)`	하나 이상의 단어 문자를 찾습니다. 캡처링 그룹은 이름은 `name` 입니다.
`:`	콜론 하나를 찾습니다.
`(?<value>\w+)`	하나 이상의 단어 문자를 찾습니다. 캡처링 그룹은 이름은 `value` 입니다.

Group 클래스의 속성은 캡처된 그룹에 대한 정보를 제공합니다. `Group.Value` 속성은 캡처된 부분 문자열을 포함하고, `Group.Index` 속성은 입력 텍스트에서 캡처된 그룹의 시작 위치를 나타내며, `Group.Length` 속성은 캡처된 텍스트의 길이를 포함하고, `Group.Success` 속성은 부분 문자열이 캡처링 그룹에 의해 정의된 패턴과 일치하는지를 나타냅니다.

그룹에 수량자를 적용하면(자세한 내용은 수량자 참조) 캡처링 그룹별로 한 캡처의 관계가 다음과 같은 두 가지 방법으로 수정됩니다.

- 그룹에 `*` 또는 `*?` 수량자(0개 이상의 일치 항목을 지정함)가 적용된 경우 캡처링 그룹은 입력 문자열에 일치 항목이 없을 수도 있습니다. 캡처된 텍스트가 없는 경우 Group 개체의 속성은 다음 테이블과 같이 설정됩니다.

그룹 속성	값
`Success`	`false`
`Value`	String.Empty
`Length`	0

다음 예제에서 이에 대해 설명합니다. 정규식 패턴 `aaa(bbb)*ccc`에서 첫 번째 캡처링 그룹(부분 문자열 "bbb")은 0번 이상 일치할 수 있습니다. 입력 문자열 "aaaccc"가 패턴과 일치하므로 캡처링 그룹은 일치 항목이 없습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "aaa(bbb)*ccc";
      string input = "aaaccc";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match value: {0}", match.Value);
      if (match.Groups[1].Success)
         Console.WriteLine("Group 1 value: {0}",
match.Groups[1].Value);
      else
         Console.WriteLine("The first capturing group has no match.");
   }
}
// The example displays the following output:
//       Match value: aaaccc
//       The first capturing group has no match.
```

- 수량자는 캡처링 그룹에 의해 정의된 패턴과 여러 번 일치할 수 있습니다. 이 경우 `Value` 개체의 `Length` 및 Group 속성은 캡처된 마지막 부분 문자열에 대한 정보만 포함합니다. 예를 들어, 다음 정규식은 마침표로 끝나는 단일 문장과 일치합니다. 두 개의 그룹화 구문을 사용합니다. 첫 번째 그룹화 구문은 개별 단어를 공백 문자와 함께 캡처하고, 두 번째 그룹화 구문은 개별 단어를 캡처합니다. 예제의 출력이 보여주는 것처럼, 정규식이 전체 문장을 캡처하는 데 성공하더라도 두 번째 캡처링 그룹은 마지막 단어만 캡처합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b((\w+)\s?)+\.";
      string input = "This is a sentence. This is another sentence.";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Match: " + match.Value);
         Console.WriteLine("Group 2: " + match.Groups[2].Value);
      }
   }
}
// The example displays the following output:
//       Match: This is a sentence.
//       Group 2: sentence
```

맨 위로 이동

캡처 컬렉션

Group 개체는 마지막 캡처에 대한 정보만 포함합니다. 그러나 캡처링 그룹에 의해 만들어진 캡처의 전체 집합은 CaptureCollection 속성에서 반환하는 Group.Captures 개체에서 계속 제공합니다. 컬렉션의 각 멤버는 해당 캡처링 그룹에 의해 만들어진 캡처를 캡처된 순서(및 따라서 캡처된 문자열이 입력 문자열에서 왼쪽에서 오른쪽으로 검색된 순서)대로 나타내는 Capture 개체입니다. 다음과 같은 두 가지 방법 중 하나로 컬력션에서 개별 Capture 개체를 검색할 수 있습니다.

- `foreach`(C#의 경우) 또는 `For Each`(Visual Basic의 경우)와 같은 생성자를 사용하여 컬렉션을 반복합니다.
- CaptureCollection.Item[] 속성을 사용하여 인덱스로 특정 개체를 검색합니다. Item[] 속성은 CaptureCollection 개체의 기본 속성(Visual Basic의 경우) 또는 인덱서(C#의 경우)입니다.

캡처링 그룹에 수량자가 적용되지 않은 경우 CaptureCollection 개체는 별로 관심이 없는 단일 Capture 개체를 포함하는데, 해당 Group 개체와 동일한 일치 항목에 대한 정보를 제공하기 때문입니다. 캡처링 그룹에 수량자가 적용된 경우 CaptureCollection 개체는 캡처링 그룹에 의해 만들어진 모든 캡처를 포함하고, 컬렉션의 마지막 멤버는 Group 개체와 동일한 캡처를 나타냅니다.

예를 들어, 정규식 패턴 `((a(b))c)+`(여기서 + 수량자는 하나 이상의 일치 항목을 지정함)를 사용하여 문자열 "abcabcabc"에서 일치 항목을 캡처하는 경우 각 CaptureCollection 개체의 Group 개체는 세 개의 멤버를 포함합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "((a(b))c)+";
      string input = "abcabcabc";

      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Match: '{0}' at position {1}",
                           match.Value, match.Index);
         GroupCollection groups = match.Groups;
         for (int ctr = 0; ctr < groups.Count; ctr++) {
            Console.WriteLine("   Group {0}: '{1}' at position {2}",
                              ctr, groups[ctr].Value, groups[ctr].Index);
            CaptureCollection captures = groups[ctr].Captures;
            for (int ctr2 = 0; ctr2 < captures.Count; ctr2++) {
               Console.WriteLine("      Capture {0}: '{1}' at position {2}",
                                 ctr2, captures[ctr2].Value,
captures[ctr2].Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Match: 'abcabcabc' at position 0
//          Group 0: 'abcabcabc' at position 0
//             Capture 0: 'abcabcabc' at position 0
//          Group 1: 'abc' at position 6
//             Capture 0: 'abc' at position 0
//             Capture 1: 'abc' at position 3
//             Capture 2: 'abc' at position 6
//          Group 2: 'ab' at position 6
//             Capture 0: 'ab' at position 0
//             Capture 1: 'ab' at position 3
//             Capture 2: 'ab' at position 6
//          Group 3: 'b' at position 7
//             Capture 0: 'b' at position 1
//             Capture 1: 'b' at position 4
//             Capture 2: 'b' at position 7
```

다음 예제에서는 정규식 `(Abc)+`를 사용하여 문자열 "XYZAbcAbcAbcXYZAbcAb"에서 문자열 "Abc"의 하나 이상의 연속 발생을 찾습니다. 이 예제에서는 Group.Captures 속성을 사용하여, 캡처된 부분 문자열의 여러 그룹을 반환하는 방법을 보여 줍니다.

C#

```
int counter;
Match m;
CaptureCollection cc;
GroupCollection gc;

// Look for groupings of "Abc".
var r = new Regex("(Abc)+");
// Define the string to search.
m = r.Match("XYZAbcAbcAbcXYZAbcAb");
gc = m.Groups;

// Display the number of groups.
Console.WriteLine("Captured groups = " + gc.Count.ToString());

// Loop through each group.
for (int i = 0; i < gc.Count; i++)
{
    cc = gc[i].Captures;
    counter = cc.Count;

    // Display the number of captures in this group.
    Console.WriteLine("Captures count = " + counter.ToString());

    // Loop through each capture in the group.
    for (int ii = 0; ii < counter; ii++)
    {
        // Display the capture and its position.
        Console.WriteLine(cc[ii] + "   Starts at character " +
             cc[ii].Index);
    }
}
// The example displays the following output:
//       Captured groups = 2
//       Captures count = 1
//       AbcAbcAbc   Starts at character 3
//       Captures count = 3
//       Abc   Starts at character 3
//       Abc   Starts at character 6
//       Abc   Starts at character 9
```

맨 위로 이동

개별 캡처

Capture 클래스는 단일 하위 식 캡처의 결과를 포함합니다. Capture.Value 속성은 일치하는 텍스트를 포함하고, Capture.Index 속성은 입력 문자열에서 일치하는 하위 문장열이 시작하는 위치(0부터 시작함)를 나타냅니다.

다음 예제에서는 선택된 도시의 온도에 대한 입력 문자열을 구문 분석합니다. 도시와 해당 온도를 구분하는 데 쉼표(",")가 사용되었으며, 각 도시의 데이터를 구분하는 데 세미콜론(";")이 사용되었습니다. 전체 입력 문자열은 단일 일치 항목을 나타냅니다. 문자열을 구문 분석하는 데 사용되는 정규식 패턴 `((\w+(\s\w+)*),(\d+);)+`에서 도시 이름은 두 번째 캡처링 그룹에 할당되었고, 온도는 네 번째 캡처링 그룹에 할당되었습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "Miami,78;Chicago,62;New York,67;San
Francisco,59;Seattle,58;";
      string pattern = @"((\w+(\s\w+)*),(\d+);)+";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Current temperatures:");
         for (int ctr = 0; ctr < match.Groups[2].Captures.Count; ctr++)
            Console.WriteLine("{0,-20} {1,3}",
match.Groups[2].Captures[ctr].Value,
                              match.Groups[4].Captures[ctr].Value);
      }
   }
}
// The example displays the following output:
//       Current temperatures:
//       Miami                78
//       Chicago              62
//       New York             67
//       San Francisco        59
//       Seattle              58
```

정규식은 다음 테이블과 같이 정의됩니다.

무늬	설명
`\w+`	하나 이상의 단어 문자를 찾습니다.

무늬	설명
`(\s\w+)*`	공백 문자 다음에 하나 이상의 단어 문자가 있는 0개 이상의 일치 항목을 찾습니다. 이 패턴에서는 여러 단어로 된 도시 이름을 찾습니다. 이 그룹은 세 번째 캡처링 그룹입니다.
`(\w+ (\s\w+)*)`	하나 이상의 단어 문자 다음에 0개 이상의 공백 문자 및 하나 이상의 단어 문자가 있는 일치 항목을 찾습니다. 이 그룹은 두 번째 캡처링 그룹입니다.
`,`	쉼표 하나를 찾습니다.
`(\d+)`	하나 이상의 숫자를 찾습니다. 이 그룹은 네 번째 캡처링 그룹입니다.
`;`	세미콜론을 하나 찾습니다.
`((\w+ (\s\w+)*), (\d+);)+`	단어 하나 다음에 추가 단어가 있고 그 다음에 쉼표 하나, 하나 이상의 숫자 및 세미콜론 하나가 한 번 이상 나타나는 패턴을 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.

참조

- System.Text.RegularExpressions
- .NET 정규식
- 정규식 언어 - 빠른 참조

정규식 동작 정보

아티클 • 2025. 04. 01.

.NET 정규식 엔진은 Perl, Python, Emacs 및 Tcl에서 사용하는 것과 같은 기존의 NFA(Nondeterministic Finite Automaton) 엔진을 통합하는 역추적 정규식 일치 도구입니다. 이를 통해 해당 awk, egrep 또는 lex와 같은 빠르지만 제한적인 순수 정규식 DFA(Deterministic Finite Automaton) 엔진과 구분합니다. 또한 표준화되지만 느린 POSIX NFA과도 구분합니다. 다음 섹션에서는 세 가지 유형의 정규식 엔진을 설명하고 기존 NFA 엔진을 사용하여 .NET의 정규식을 구현하는 이유를 설명합니다.

NFA 엔진의 이점

DFA 엔진에서 패턴 일치를 수행하는 경우 해당 처리 순서는 입력 문자열에 의해 결정됩니다. 엔진은 입력 문자열의 처음부터 시작해서 순차적으로 다음 문자가 정규식 패턴과 일치하는지 여부를 확인합니다. 가능한 가장 긴 문자열을 일치시키도록 보장할 수 있습니다. 동일한 문자를 두 번 테스트하지 않기 때문에 DFA 엔진은 역추적을 지원하지 않습니다. 그러나 DFA 엔진에 유한 상태만 포함되기 때문에 역참조를 사용하는 패턴과 일치하지 않으며 명시적 확장을 생성하지 않기 때문에 하위 식을 캡처할 수 없습니다.

DFA 엔진과 달리 기존의 NFA 엔진이 패턴 일치를 수행할 경우 해당 처리 순서는 정규식 패턴에 의해 결정됩니다. 특정 언어 요소를 처리할 때, 엔진은 가능한 한 입력 문자열의 가장 많은 부분을 일치시키는 탐욕적 일치를 사용합니다. 하지만 성공적으로 하위 식을 찾은 후에 해당 상태를 저장하기도 합니다. 일치에 결국 실패하면 엔진이 추가 일치를 시도할 수 있도록 저장된 상태로 돌아갈 수 있습니다. 정규식에서 이후 언어 요소가 일치할 수 있도록 성공적인 하위 식 찾기를 포기하는 이 프로세스를 '역추적'이라고 합니다. NFA 엔진은 역추적을 사용하여 특정 순서에 따라 정규식에서 가능한 모든 확장을 테스트하고 첫 번째 일치 항목을 수락합니다. 기존의 NFA 엔진이 성공적으로 일치를 수행하기 위해 정규식의 특정 확장을 구성하기 때문에 하위 식 일치 및 일치 역참조를 캡처할 수 있습니다. 그러나 기존의 NFA는 역추적하기 때문에 다른 경로를 통해 도착할 경우 동일한 상태를 여러 번 방문할 수 있습니다. 결과적으로 최악의 경우 상당히 느리게 실행될 수 있습니다. 기존 NFA 엔진이 발견한 첫 번째 일치를 수용하기 때문에 다른 일치 항목(더 긴 일치일 가능성이 높음)이 발견되지 않을 수 있습니다.

POSIX NFA 엔진은 가장 긴 일치 항목을 발견했다고 보장할 수 있을 때까지 역추적을 계속한다는 점을 제외하면 기존의 NFA 엔진과 같습니다. 결과적으로 POSIX NFA 엔진은 기존 NFA 엔진보다 느립니다. 따라서 POSIX NFA 엔진을 사용하는 경우 역추적 검사의 순서를 변경하여 긴 일치 항목보다 짧은 일치 항목을 우선할 수 없습니다.

기존의 NFA 엔진은 DFA 또는 POSIX NFA 엔진보다 일치하는 문자열을 효율적으로 제어하기 때문에 프로그래머로부터 선호되었습니다. 최악의 경우 실행 속도가 느려질 수 있지

만 모호성을 줄이고 역추적을 제한하는 패턴을 사용하여 선형 또는 다항 시간에서 일치 항목을 찾도록 조정할 수 있습니다. 즉, NFA 엔진은 강력함과 유연성을 제공하는 대신 성능이 떨어지지만 대부분의 경우에 정규식을 잘 작성하여 역추적이 성능을 기하급수적으로 저하시키지 않도록 방지한다면 사용 가능한 적절한 성능을 제공합니다.

> ⓘ **참고**
>
> 과도한 역추적으로 인해 발생한 성능 저하 및 이를 해결하는 정규식을 만드는 방법에 대한 자세한 내용은 **역추적**을 참조하세요.

.NET 엔진 기능

기존 NFA 엔진의 이점을 누리려면 .NET 정규식 엔진에는 프로그래머가 역추적 엔진을 조정할 수 있는 일련의 구문이 포함되어 있습니다. 이러한 구문을 사용하여 일치 항목을 빠르게 찾거나 다른 일치 항목에 대한 특정 확장을 우선할 수 있습니다.

.NET 정규식 엔진의 다른 기능은 다음과 같습니다.

- 게으른 수량자: `??`, `*?`, `+?`, `{n,m}?` 이러한 구문은 역추적 엔진이 최소 반복 횟수를 먼저 검색하도록 지시합니다. 반면, 일반적인 탐욕적 수량자는 먼저 최대 반복 횟수를 찾으려고 합니다. 다음 예제에서는 둘 사이의 차이점을 보여 줍니다. 정규표현식은 숫자로 끝나는 문장을 매치하고 캡처링 그룹은 그 숫자를 추출하도록 설계되었습니다. 정규식 `.+(\d+)\.`은 탐욕적 수량자 `.+`를 포함하며 이로 인해 정규식 엔진은 번호의 마지막 숫자만 캡처하게 됩니다. 반대로 정규식 `.+?(\d+)\.`은 게으른 수량자 `.+?`를 포함하며 이로 인해 정규식 엔진은 전체 번호를 캡처하게 됩니다.

  C#

  ```
  using System;
  using System.Text.RegularExpressions;

  public class Example
  {
      public static void Main()
      {
          string greedyPattern = @".+(\d+)\.";
          string lazyPattern = @".+?(\d+)\.";
          string input = "This sentence ends with the number 107325.";
          Match match;

          // Match using greedy quantifier .+.
          match = Regex.Match(input, greedyPattern);
          if (match.Success)
              Console.WriteLine($"Number at end of sentence (greedy):
  {match.Groups[1].Value}");
  ```

```
        else
            Console.WriteLine($"{greedyPattern} finds no match.");

        // Match using lazy quantifier .+?.
        match = Regex.Match(input, lazyPattern);
        if (match.Success)
            Console.WriteLine($"Number at end of sentence (lazy):
{match.Groups[1].Value}");
        else
            Console.WriteLine($"{lazyPattern} finds no match.");
    }
}
// The example displays the following output:
//       Number at end of sentence (greedy): 5
//       Number at end of sentence (lazy): 107325
```

이 정규식의 탐욕적 버전 및 나태한 버전은 다음 표와 같이 정의됩니다.

테이블 확장

패턴	설명
`.+`(탐욕적 수량자)	적어도 한 번 이상 어떤 문자와 일치합니다. 그러면 정규식 엔진이 전체 문자열을 검색하고 필요한 패턴의 나머지 부분을 찾는 데 필요한 역추적을 수행합니다.
`.+?`(게으른 수량자)	적어도 한 번 문자를 찾지만 가능한 한 적게 일치시키도록 합니다.
`(\d+)`	적어도 하나의 숫자 문자를 찾아 첫 번째 캡처링 그룹에 할당합니다.
`\.`	마침표와 일치시킵니다.

수량자에 대한 더 많은 정보를 원하시면 수량자를 참조하세요.

- 긍정적 전방 탐색: `(?=`*subexpression*`)`. 이 기능을 사용하면 역추적 검사 엔진이 하위 식을 찾은 후에 텍스트의 동일한 지점으로 돌아오게 합니다. 동일한 위치에서 시작하는 여러 패턴을 확인하여 전체 텍스트를 검색하는 데 유용합니다. 이를 통해 엔진은 일치한 텍스트에 부분 문자열을 포함하지 않으면서 부분 문자열이 일치 항목의 끝에 존재하는지 확인할 수 있습니다. 다음 예제에서는 긍정적 전방 탐색을 사용하여 문장 부호가 뒤에 오지 않는 문장의 단어를 추출합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
```

```
    public static void Main()
    {
        string pattern = @"\b[A-Z]+\b(?=\P{P})";
        string input = "If so, what comes next?";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//       If
//       what
//       comes
```

`\b[A-Z]+\b(?=\P{P})` 정규식은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서부터 일치를 시작합니다.
`[A-Z]+`	알파벳 문자를 하나 이상 매칭합니다. Regex.Matches 메서드가 RegexOptions.IgnoreCase 옵션으로 호출되므로 이 비교는 대/소문자를 구분하지 않습니다.
`\b`	단어 경계에서 매치를 종료합니다.
`(?=\P{P})`	다음 문자가 문장 부호 기호인지 확인하기 위해 앞을 살펴봅니다. 그렇지 않은 경우 일치가 성공합니다.

긍정적 전방 탐색 어설션에 대한 자세한 내용은 그룹화 구문을 참조하세요.

- 부정 전방 탐색: `(?!`*subexpression*`)`. 이 기능은 하위 식이 일치하지 않을 때에만 표현식을 일치시키는 기능을 추가합니다. 이 기능은 검색 항목을 정리하는 데 유용합니다. 포함해야 하는 사례에 대한 식보다 제거해야 하는 경우에 대한 식을 제공하는 것이 더 간단하기 때문입니다. 예를 들어 "non"으로 시작하지 않는 단어에 대한 식을 작성하기가 어렵습니다. 다음 예제에서는 부정적 전방 탐색을 사용하여 그것들을 제외합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
```

```
        string pattern = @"\b(?!non)\w+\b";
        string input = "Nonsense is not always non-functional.";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//       is
//       not
//       always
//       functional
```

정규식 패턴 `\b(?!non)\w+\b`는 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 일치 항목 찾기를 시작합니다.
`(?!non)`	현재 문자열이 "non"으로 시작하지 않는지 확인하세요. 그렇게 되면 일치가 실패합니다.
`(\w+)`	하나 이상의 단어 문자를 매칭합니다.
`\b`	단어 경계에서 매치를 종료합니다.

부정 전방 단언에 대한 자세한 내용은 그룹화 구문을 참조하세요.

- 조건부 평가: `(?(`*expression*`)`*yes*`|`*no*`)` 및 `(?(`*name*`)`*yes*`|`*no*`)`. 여기서 *expression*은 일치시킬 하위 식이고, *name*은 캡처 그룹의 이름이고, *yes*는 *expression*이 일치하거나 *name*이 올바른 비어 있지 않은 캡처된 그룹인 경우 일치시킬 문자열이고, *no*는 *expression*이 일치하지 않거나 *name*이 유효하고 비어 있지 않은 캡처된 그룹이 아닌 경우 일치시킬 하위 식입니다. 엔진은 이 기능을 통해 이전 하위 식이 일치 결과 및 너비 0인 어설션 결과에 따라 둘 이상의 대체 패턴을 사용하여 검색할 수 있습니다. 따라서 이전 하위 식이 일치하는지 여부에 따라 하위 식을 찾도록 허용하는 더 강력한 형식의 역참조가 가능합니다. 다음 예제의 정규식은 공용으로 사용하고 내부적으로 사용하려는 단락을 모두 찾습니다. 내부적으로 사용하려는 단락은 `<PRIVATE>` 태그로 시작합니다. 정규식 패턴 `^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?$`은 조건부 평가를 사용하여 공용으로 사용하고 내부적으로 사용하려는 단락의 콘텐츠를 별도의 캡처링 그룹에 할당합니다. 이러한 단락은 다르게 처리될 수 있습니다.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "<PRIVATE> This is not for public consumption."
+ Environment.NewLine +
                       "But this is for public consumption." +
Environment.NewLine +
                       "<PRIVATE> Again, this is confidential.\n";
        string pattern = @"^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?
\s)+)|((\w+\p{P}?\s)+))\r?$";
        string publicDocument = null, privateDocument = null;

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Multiline))
        {
            if (match.Groups[1].Success)
            {
                privateDocument += match.Groups[1].Value + "\n";
            }
            else
            {
                publicDocument += match.Groups[3].Value + "\n";
                privateDocument += match.Groups[3].Value + "\n";
            }
        }

        Console.WriteLine("Private Document:");
        Console.WriteLine(privateDocument);
        Console.WriteLine("Public Document:");
        Console.WriteLine(publicDocument);
    }
}
// The example displays the following output:
//    Private Document:
//    This is not for public consumption.
//    But this is for public consumption.
//    Again, this is confidential.
//
//    Public Document:
//    But this is for public consumption.
```

정규식 패턴 는 다음 표와 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	줄의 시작 부분에서 매치를 시작합니다.
`(?<Pvt>\<PRIVATE\>\s)?`	공백 문자 다음의 0개 또는 1개의 `<PRIVATE>` 문자열을 일치시킵니다. `Pvt`로 명명된 캡처 그룹에 할당합니다.
`(?(Pvt)((\w+\p{P}?\s)+)`	`Pvt` 캡처링 그룹이 존재하면, 하나 이상의 단어 문자가 뒤따르고 0개 또는 1개의 문장 구분 기호 뒤에 공백 문자가 오는 경우를 한 번 이상 찾습니다. 첫 번째 캡처링 그룹에 부분 문자열을 할당합니다.
`|((\w+\p{P}?\s)+))`	`Pvt` 캡처링 그룹이 없는 경우, 하나 이상의 단어 문자가 0개 또는 1개의 문장 구분 기호와 그 뒤에 공백 문자와 함께 한 번 이상 등장하는 것을 찾습니다. 세 번째 캡처링 그룹에 부분 문자열을 할당합니다.
`\r?$`	줄의 끝 또는 문자열의 끝을 찾습니다.

조건부 평가에 대한 자세한 내용은 교체 구문을 참조하세요.

- 균형 조정 그룹 정의: `(?<`*name1*`-`*name2*`>`*subexpression*`)`. 정규식 엔진은 이 기능을 통해 괄호 또는 열고 닫는 대괄호와 같은 중첩된 구문을 추적할 수 있습니다. 예제는 그룹화 구문을 참조하세요.

- 원자성 그룹: `(?>`*subexpression*`)`. 이 기능은 역추적 엔진이 하위 식이 그 하위 식에 대해 처음 발견된 일치 항목만을 일치시키도록 보장해 주며, 이는 하위 식이 포함된 식과 독립적으로 실행되는 것과 같습니다. 이 생성자를 사용하지 않는 경우 더 큰 식의 역추적 검색은 하위 식의 동작을 변경할 수 있습니다. 예를 들어 정규식 `(a+)\w`는 "a" 문자의 시퀀스를 따르는 단어 문자와 함께 하나 이상의 "a" 문자를 찾고 "a" 문자의 시퀀스를 첫 번째 캡처링 그룹에 할당합니다. 하지만 입력 문자열의 마지막 문자가 "a"인 경우 `\w` 언어 요소에 의해 찾게 되며 캡처된 그룹에 포함되지 않습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "aaaaa", "aaaaab" };
        string backtrackingPattern = @"(a+)\w";
        Match match;

        foreach (string input in inputs)
        {
            Console.WriteLine($"Input: {input}");
            match = Regex.Match(input, backtrackingPattern);
            Console.WriteLine($"   Pattern: {backtrackingPattern}");
```

```
            if (match.Success)
            {
                Console.WriteLine($"      Match: {match.Value}");
                Console.WriteLine($"      Group 1:
{match.Groups[1].Value}");
            }
            else
            {
                Console.WriteLine("      Match failed.");
            }
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//       Input: aaaaa
//          Pattern: (a+)\w
//             Match: aaaaa
//             Group 1: aaaa
//       Input: aaaaab
//          Pattern: (a+)\w
//             Match: aaaaab
//             Group 1: aaaaa
```

정규식 `((?>a+))\w`은 이러한 동작을 방지합니다. 역추적 없이 연속된 모든 "a" 문자를 찾기 때문에 첫 번째 캡처링 그룹에는 이 연속된 모든 "a" 문자가 포함됩니다. "a" 문자의 뒤에 "a" 이외의 문자가 적어도 하나 이상 있지 않으면 검색에 실패합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "aaaaa", "aaaaab" };
        string nonbacktrackingPattern = @"((?>a+))\w";
        Match match;

        foreach (string input in inputs)
        {
            Console.WriteLine($"Input: {input}");
            match = Regex.Match(input, nonbacktrackingPattern);
            Console.WriteLine($"   Pattern: {nonbacktrackingPattern}");
            if (match.Success)
            {
                Console.WriteLine($"      Match: {match.Value}");
                Console.WriteLine($"      Group 1:
{match.Groups[1].Value}");
            }
```

```
            else
            {
                Console.WriteLine("      Match failed.");
            }
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//       Input: aaaaa
//          Pattern: ((?>a+))\w
//             Match failed.
//       Input: aaaaab
//          Pattern: ((?>a+))\w
//             Match: aaaaab
//             Group 1: aaaaa
```

원자성 그룹에 대한 자세한 내용은 그룹화 구문을 참조하세요.

- 오른쪽에서 왼쪽 찾기는 RegexOptions.RightToLeft 옵션을 Regex 클래스 생성자 또는 고정 인스턴스 일치 메서드에 제공하여 지정됩니다. 이 기능은 왼쪽에서 오른쪽이 아닌 오른쪽에서 왼쪽으로 찾는 경우에 유용하고 패턴의 왼쪽이 아닌 패턴의 오른쪽 부분에서 찾기를 시작하는 경우 효율적입니다. 다음 예제와 같이 오른쪽에서 왼쪽 찾기를 사용하면 탐욕적 수량자의 동작을 변경할 수 있습니다. 예제에서는 숫자로 끝나는 문장에 대해 두 개의 검색을 수행합니다. 탐욕적 수량자를 사용하는 왼쪽에서 오른쪽 검색 `+`은 문장의 6자리 숫자 중 하나와 일치하는 반면, 오른쪽에서 왼쪽 검색은 모든 6자리 숫자와 일치합니다. 정규식 패턴에 대한 설명은 이 섹션 앞부분의 게으른 수량자를 보여 주는 예제를 참조하세요.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string greedyPattern = @".+(\d+)\.";
        string input = "This sentence ends with the number 107325.";
        Match match;

        // Match from left-to-right using lazy quantifier .+?.
        match = Regex.Match(input, greedyPattern);
        if (match.Success)
            Console.WriteLine($"Number at end of sentence (left-to-
right): {match.Groups[1].Value}");
        else
            Console.WriteLine($"{greedyPattern} finds no match.");
```

```
        // Match from right-to-left using greedy quantifier .+.
        match = Regex.Match(input, greedyPattern,
RegexOptions.RightToLeft);
        if (match.Success)
            Console.WriteLine($"Number at end of sentence (right-to-
left): {match.Groups[1].Value}");
        else
            Console.WriteLine($"{greedyPattern} finds no match.");
    }
}
// The example displays the following output:
//       Number at end of sentence (left-to-right): 5
//       Number at end of sentence (right-to-left): 107325
```

오른쪽에서 왼쪽 찾기에 대한 자세한 내용은 정규식 옵션을 참조하세요.

- 긍정 및 부정 후방검사: 긍정 후방검사의 경우 `(?<=`*subexpression*`)`, 부정 후방검사의 경우 `(?<!`*subexpression*`)`. 이 기능은 이 항목의 앞부분에서 설명한 lookahead와 비슷합니다. 정규식 엔진은 완전한 오른쪽에서 왼쪽으로의 일치를 지원하기 때문에, 정규 표현식은 무제한적인 후방탐색을 허용합니다. 긍정 및 부정 lookbehind는 중첩된 하위 식이 외부 식의 상위 집합인 경우 중첩된 수량자를 방지하는 데도 사용될 수 있습니다. 이러한 중첩된 수량자가 있는 정규식은 종종 성능이 저하됩니다. 예를 들어 다음 예제에서는 문자열이 영숫자 문자로 시작하고 끝나고 문자열의 다른 문자가 큰 하위 집합 중 하나임을 확인합니다. 이 식은 전자 메일 주소의 유효성을 검사하는 데 사용되는 정규식의 일부를 형성합니다. 자세한 내용은 방법: 문자열이 올바른 전자 메일 형식인지 확인을 참조하세요.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "jack.sprat", "dog#", "dog#1", "me.myself",
                            "me.myself!" };
        string pattern = @"^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-
Z0-9])$";
        foreach (string input in inputs)
        {
            if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
                Console.WriteLine($"{input}: Valid");
            else
                Console.WriteLine($"{input}: Invalid");
        }
    }
}
// The example displays the following output:
```

```
//       jack.sprat: Valid
//       dog#: Invalid
//       dog#1: Valid
//       me.myself: Valid
//       me.myself!: Invalid
```

`^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-Z0-9])$` 정규식은 다음 테이블과 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	문자열의 시작 부분에서 일치를 시작합니다.
`[A-Z0-9]`	숫자 또는 영숫자 문자를 찾습니다. 비교는 대소문자를 구분하지 않습니다.
`([-!#$%&'.*+/=?^`{}|~\w])*`	단어 문자 또는 다음 문자 중 하나인 -, !, #, $, %, &, ', ., *, +, /, =, ?, ^, `, {, }, |, ~의 0개 이상의 일치 항목을 찾습니다.
`(?<=[A-Z0-9])`	숫자나 영숫자여야 하는 이전 글자를 살펴보십시오. 비교는 대소문자를 구분하지 않습니다.
`$`	문자열의 끝에서 매치를 종료합니다.

긍정 및 부정 후방 탐색에 대한 자세한 내용은 그룹화 구문을 참조하세요.

관련 문서

테이블 확장

제목	설명
백트래킹	정규 표현식의 역추적이 대체 일치를 찾기 위해 분기하는 방법에 대한 정보를 제공합니다.
컴파일 및 다시 사용	성능 향상을 위해 정규식을 컴파일하고 다시 사용하는 방법에 대한 정보를 제공합니다.
.NET 정규식 사용	정규식의 프로그래밍 언어 측면에 대한 개요를 제공합니다.
정규 표현식 객체 모델	정규식 클래스를 사용하는 방법을 보여 주는 코드 예제 및 정보를 제공합니다.
정규식 언어 - 빠른 참조	정규식을 정의하는 데 사용할 수 있는 문자, 연산자 및 생성자 집합에 대한 정보를 제공합니다.

참조

- System.Text.RegularExpressions

정규식의 역추적

아티클 • 2023. 08. 11.

역추적은 정규식 패턴에 선택적인 수량자 또는 교체 구문이 포함되어 있고 정규식 엔진이 일치 항목을 계속 검색하기 위해 이전에 저장한 상태로 되돌아갈 때 발생합니다. 역추적은 정규식 성능의 핵심입니다. 역추적을 사용하면 식의 성능과 유연성을 높일 수 있으며 매우 복잡한 패턴도 검색할 수 있습니다. 하지만 이러한 장점에는 단점이 수반됩니다. 역추적은 종종 정규식 엔진의 성능에 영향을 주는 가장 중요한 단일 요소입니다. 다행히도 개발자는 정규식 엔진의 동작과 역추적 사용 방식을 제어할 수 있습니다. 이 항목에서는 역추적의 작동 방식 및 역추적을 제어할 수 있는 방법에 대해 설명합니다.

> ⚠ **경고**
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions`에 대한 입력을 제공하여 **서비스 거부 공격**을 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

역추적을 사용하지 않는 선형 비교

정규식 패턴에 선택적인 수량자 또는 교체 구문이 없으면 정규식 엔진이 선형 시간으로 실행됩니다. 즉, 정규식 엔진은 패턴에서 입력 문자열의 텍스트와 일치하는 첫 번째 언어 요소를 검색한 후 다시 패턴에서 입력 문자열의 다음 문자 또는 문자 그룹과 일치하는 다음 언어 요소를 찾습니다. 이 작업은 검색이 성공할 때까지 계속되고, 그렇지 않으면 검색이 실패합니다. 어느 경우에든 정규식 엔진은 입력 문자열에서 한 번에 한 글자씩 검색을 진행합니다.

다음 예제에서 이에 대해 설명합니다. 정규식 `e{2}\w\b` 는 모든 단어 문자에서 단어 경계까지 "e"가 두 번 나오는 단어를 검색합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example1
{
    public static void Run()
    {
        string input = "needing a reed";
        string pattern = @"e{2}\w\b";
        foreach (Match match in Regex.Matches(input, pattern))
```

```
            Console.WriteLine("{0} found at position {1}",
                              match.Value, match.Index);
        }
    }
    // The example displays the following output:
    //       eed found at position 11
```

이 정규식은 수량자 `{2}`가 있더라도 선형 방식으로 평가됩니다. `{2}` 는 선택적인 수량자가 아니기 때문에 정규식 엔진이 역추적을 수행하지 않습니다. 이 수량자는 정확한 숫자를 지정하며 이전 하위 식이 검색해야 하는 가변 횟수가 아닙니다. 따라서 정규식 엔진은 다음 표에서와 같이 입력 문자열에서 정규식 패턴과 일치하는 항목을 검색하려고 시도합니다.

Operation	패턴 내 위치	문자열 내 위치	Result
6	e	"needing a reed"(인덱스 0)	일치하는 항목이 없습니다.
2	e	"eeding a reed"(인덱스 1)	일치 가능
3	e{2}	"eding a reed"(인덱스 2)	일치 가능
4	\w	"ding a reed"(인덱스 3)	일치 가능
5	\b	"ing a reed"(인덱스 4)	일치 가능 실패
6	e	"eding a reed"(인덱스 2)	일치 가능
7	e{2}	"ding a reed"(인덱스 3)	일치 가능 실패
8	e	"ding a reed"(인덱스 3)	검색이 실패합니다.
9	e	"ing a reed"(인덱스 4)	일치하는 항목이 없습니다.
10	e	"ng a reed"(인덱스 5)	일치하는 항목이 없습니다.
11	e	"g a reed"(인덱스 6)	일치하는 항목이 없습니다.
12	e	" a reed"(인덱스 7)	일치하는 항목이 없습니다.
13	e	"a reed"(인덱스 8)	일치하는 항목이 없습니다.
14	e	" reed"(인덱스 9)	일치하는 항목이 없습니다.
15	e	"reed"(인덱스 10)	일치 없음
16	e	"eed"(인덱스 11)	일치 가능
17	e{2}	"ed"(인덱스 12)	일치 가능
18	\w	"d"(인덱스 13)	일치 가능

Operation	패턴 내 위치	문자열 내 위치	Result
19	\b	""(인덱스 14)	일치

정규식 엔진에 선택적인 수량자가 없거나 교체 구문이 없는 경우 입력 문자열에서 정규식 패턴과 일치하는 항목을 찾기 위해 필요한 최대 비교 수는 입력 문자열에 있는 문자 수와 거의 동일합니다. 이 경우 정규식 엔진은 이 13자 길이의 문자열에서 가능한 일치 항목을 식별하기 위해 19가지를 비교합니다. 즉, 선택적인 수량자 또는 대체 생성 구문이 없는 경우 정규식 엔진이 선형에 가까운 시간으로 실행됩니다.

선택적인 수량자 또는 교체 구문을 사용한 역추적

정규식에 선택적인 수량자 또는 교체 구문이 포함된 경우 입력 문자열에 대한 평가는 더 이상 선형으로 수행되지 않습니다. NFA(Nondeterministic Finite Automaton) 엔진과 패턴 일치는 입력 문자열에서 일치시킬 문자가 아니라 정규식의 언어 요소에 의해 구동됩니다. 따라서 정규식 엔진은 선택적인 하위 식 또는 교체 하위 식에 대해 전체 검색을 수행합니다. 하위 식의 다음 언어 요소로 진행할 때 검색이 실패하면 정규식 엔진이 성공한 일치 부분을 버리고 입력 문자열 전체에 대한 정규식 검색을 수행하기 위해 이전에 저장된 상태로 돌아갈 수 있습니다. 일치하는 항목을 찾기 위해 이전에 저장된 상태로 돌아가는 프로세스를 역추적이라고 부릅니다.

예를 들어 임의의 문자로 시작해서 "es"가 포함된 항목을 검색하는 `.*(es)`라는 정규식 패턴이 있다고 가정해보십시오. 다음 예제에서와 같이 입력 문자열이 "Essential services are provided by regular expressions."인 경우 이 패턴은 "expressions"의 "es"를 포함하여 전체 문자열을 끝까지 검색합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example2
{
    public static void Run()
    {
        string input = "Essential services are provided by regular
expressions.";
        string pattern = ".*(es)";
        Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
        if (m.Success)
        {
            Console.WriteLine("'{0}' found at position {1}",
                              m.Value, m.Index);
            Console.WriteLine("'es' found at position {0}",
```

```
                                m.Groups[1].Index);
            }
        }
}
//    'Essential services are provided by regular expressions found at
position 0
//    'es' found at position 47
```

이를 위해 정규식 엔진은 다음과 같은 방식으로 역추적을 사용합니다.

- 전체 입력 문자열에서 `.*` (0개 이상의 임의 문자 검색)를 검색합니다.
- 정규식 패턴의 "e"와 일치하는 항목을 찾습니다. 하지만 입력 문자열에는 검색에 사용할 수 있는 남은 문자가 없습니다.
- 마지막으로 일치한 항목인 "Essential services are provided by regular expressions"로 역추적하고 문장의 끝에 있는 마침표에서 "e"와 일치하는 항목을 찾습니다. 그러면 검색이 실패합니다.
- 계속해서 일시적으로 일치하는 하위 문자열이 "Essential services are provided by regular expr"이 될 때까지 한 번에 한 글자씩 이전에 성공한 일치 항목으로 역추적합니다. 그런 다음 패턴에 있는 "e"와 "expressions"의 두 번째 "e"를 비교하여 일치하는 항목을 찾습니다.
- 패턴에 있는 "s"와 일치한 "e" 문자 다음의 "s"("expressions"의 첫 번째 "s")를 비교합니다. 그러면 검색이 성공합니다.

역추적을 사용할 경우 길이가 55자인 입력 문자열에서 정규식 패턴과 일치하는 항목을 검색하려면 67번의 비교 작업이 필요합니다. 일반적으로 정규식 엔진에 단일 교체 구문이 포함되었거나 선택적인 단일 수량자가 포함된 경우 패턴을 검색하는 데 필요한 비교 작업 수는 입력 문자열에 있는 문자 수의 두 배 이상입니다.

선택적인 중첩된 수량자를 사용한 역추적

패턴에 교체 구문이 많이 포함되었거나 중첩된 교체 구문이 포함되었거나, 선택적인 중첩된 수량자가 포함된 경우(가장 일반적인 경우) 정규식 패턴과 일치하는 항목을 찾기 위해 필요한 비교 작업 수가 기하급수적으로 증가할 수 있습니다. 예를 들어 정규식 패턴 `^(a+)+$` 는 하나 이상의 "a" 문자가 포함된 전체 문자열을 검색하도록 디자인되었습니다. 예제에는 동일한 길이의 두 입력 문자열이 제공되지만 첫 번째 문자열만 패턴과 일치합니다. System.Diagnostics.Stopwatch 클래스는 일치 항목을 검색하는 작업이 수행되는 시간을 확인하는 데 사용됩니다.

```
C#
```

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example3
{
    public static void Run()
    {
        string pattern = "^(a+)+$";
        string[] inputs = { "aaaaaaaaaaaaaaaaaaaaaaaaaa",
"aaaaaaaaaaaaaaaaaaaaaaaaa!" };
        Regex rgx = new Regex(pattern);
        Stopwatch sw;

        foreach (string input in inputs)
        {
            sw = Stopwatch.StartNew();
            Match match = rgx.Match(input);
            sw.Stop();
            if (match.Success)
                Console.WriteLine($"Matched {match.Value} in {sw.Elapsed}");
            else
                Console.WriteLine($"No match found in {sw.Elapsed}");
        }
    }
}
//    Matched aaaaaaaaaaaaaaaaaaaaaaaaaa in 00:00:00.0018281
//    No match found in 00:00:05.1882144
```

예제의 출력에서 알 수 있듯이 정규식 엔진은 일치하는 문자열을 식별하기 위해 입력 문자열 *이 패턴과 일치하지* 않는 것을 발견하는 데 훨씬 더 오래 걸렸습니다. 그 이유는 성공하지 못한 검색은 항상 최악의 경우를 나타내기 때문입니다. 정규식 엔진은 일치하는 항목이 없다는 결론을 내리기까지 정규식을 사용하여 데이터에서 가능한 모든 경로를 따라야 하며, 중첩된 괄호가 있으면 데이터에서 발생 가능한 경로 수가 추가로 늘어납니다. 정규식 엔진은 다음을 수행하여 두 번째 문자열이 패턴과 일치하지 않는다는 결론을 내립니다.

- 문자열의 시작 위치에 있는지 확인한 후 `a+`패턴을 사용하여 문자열에서 처음 5개 문자를 검색합니다. 그런 후 문자열에 "a" 문자의 추가 그룹이 없는지 확인합니다. 마지막으로 문자열의 끝인지 테스트합니다. 문자열에 하나의 추가 문자가 남아 있기 때문에 검색이 실패합니다. 이 실패한 검색에는 9번의 비교 작업이 필요합니다. 또한 정규식 엔진은 "a"(일치 1이라고 함), "aa"(일치 2), "aaa"(일치 3) 및 "aaaa"(일치 4)의 일치 항목에서 상태 정보를 저장합니다.

- 정규식 엔진이 이전에 저장된 매치 4로 돌아갑니다. 추가 캡처 그룹에 할당할 수 있도록 "a" 문자가 추가로 하나 더 있는지 확인합니다. 마지막으로 문자열의 끝인지 테스트합니다. 문자열에 하나의 추가 문자가 남아 있기 때문에 검색이 실패합니다. 이

실패한 검색에는 4번의 비교 작업이 필요합니다. 지금까지 총 13번의 비교 작업이 수행되었습니다.

- 정규식 엔진이 이전에 저장된 매치 3으로 돌아갑니다. 추가 캡처 그룹에 할당할 수 있도록 두 개의 추가 "a" 문자가 있는지 확인합니다. 하지만 문자열 끝 테스트가 실패합니다. 그런 다음, 3과 일치하도록 반환하고 캡처된 두 개의 추가 그룹에서 두 개의 추가 "a" 문자를 일치시키려고 시도합니다. 그래도 문자열 끝 테스트가 실패합니다. 이렇게 실패한 검색 작업에는 12번의 비교 작업이 필요합니다. 지금까지 총 25번의 비교 작업이 수행되었습니다.

입력 문자열을 정규식 엔진에서 비교하는 작업은 정규식이 검색 작업의 모든 가능한 조합을 시도하고 일치 항목이 없다는 결론을 내릴 때까지 이러한 방식으로 계속해서 수행됩니다. 중첩된 수량자로 인해 이러한 비교는 $O(2^n)$ 또는 지수 연산으로 수행되며, 여기서 n은 입력 문자열에 있는 문자 수입니다. 즉, 문자 수가 30개인 입력 문자열에서는 최악의 경우 약 1,073,741,824번의 비교 작업이 필요하고, 입력 문자열의 문자 수가 40개이면 약 1,099,511,627,776번의 비교 작업이 필요합니다. 이정도 또는 심지어 더 긴 문자열을 사용하면 정규식 메서드가 정규식 패턴과 일치하지 않는 입력을 처리할 때 완료 시간이 극단적으로 길어질 수 있습니다.

컨트롤 역추적

역추적을 사용하면 강력하고 유연한 정규식을 만들 수 있습니다. 하지만 이전 단원에 설명한 것처럼 이러한 장점 외에도 성능이 매우 크게 저하될 수 있음에 유의해야 합니다. 과도한 역추적을 방지하려면 Regex 개체를 인스턴스화하거나 정적 정규식 일치 메서드를 호출할 때 시간 제한 간격을 정의해야 합니다. 이에 대해서는 다음 섹션에서 설명합니다. 그 밖에도, .NET에서는 역추적을 제한하거나 억제하고, 성능상의 제약이 거의 없거나 전혀 없이 복잡한 정규식을 지원하는 세 가지 정규식 언어 요소인 원자성 그룹, lookbehind 어설션 및 lookahead 어설션을 지원합니다. 각 언어 요소에 대한 자세한 내용은 그룹화 구문을 참조 하세요.

역추적하지 않는 정규식 엔진

역추적이 필요한 구문(예: 조회, 역참조 또는 원자성 그룹)을 사용할 필요가 없는 경우 모드를 사용하는 RegexOptions.NonBacktracking 것이 좋습니다. 이 모드는 입력 길이에 비례하여 시간에 따라 실행되도록 설계되었습니다. 자세한 내용은 NonBacktracking 모드를 참조하세요. 제한 시간 값을 설정할 수도 있습니다.

입력 크기 제한

입력이 매우 크지 않는 한 일부 정규식은 허용 가능한 성능을 갖습니다. 시나리오의 모든 적절한 텍스트 입력이 특정 길이 미만인 것으로 알려진 경우 정규식을 적용하기 전에 더

긴 입력을 거부하는 것이 좋습니다.

제한 시간 간격 지정

정규식 엔진이 시도를 포기하고 예외를 throw하기 전에 단일 일치 항목을 검색하는 가장 긴 간격을 나타내는 제한 시간 값을 설정할 수 있습니다 RegexMatchTimeoutException . TimeSpan 값을 인스턴스 정규식을 위한 Regex(String, RegexOptions, TimeSpan) 생성자에 제공하여 시간 제한 간격을 지정합니다. 또한, 각각의 정적 패턴 일치 메서드에 시간 제한 값을 지정할 수 있게 해주는 TimeSpan 매개 변수의 오버로드가 있습니다.

제한 시간 값을 명시적으로 설정하지 않으면 기본 제한 시간 값이 다음과 같이 결정됩니다.

- 애플리케이션 수준 시간 제한을 사용 하 여 값 하나 있습니다. 이 애플리케이션 도메인에 적용 되는 제한 시간 값 수는 Regex 개체가 인스턴스화되거나 정적 메서드를 호출 합니다. 호출 하 여 애플리케이션 수준 시간 제한 값을 설정할 수 있습니다는 AppDomain.SetData 의 문자열 표현에 할당할 메서드를 TimeSpan "REGEX_DEFAULT_MATCH_TIMEOUT" 속성 값입니다.
- 값을 사용 하 여 InfiniteMatchTimeout없는 애플리케이션 수준 시간 제한 값이 설정된 경우.

기본적으로, 시간 제한 간격은 Regex.InfiniteMatchTimeout 으로 설정되고, 정규식 엔진의 시간이 초과되지 않습니다.

> **ⓘ 중요**
>
> 사용하지 **RegexOptions.NonBacktracking**않는 경우 정규식이 역추적을 사용하거나 신뢰할 수 없는 입력에서 작동하는 경우 항상 시간 제한 간격을 설정하는 것이 좋습니다.

RegexMatchTimeoutException 예외는 정규식 엔진이 지정된 시간 제한 간격 내에 일치 항목을 찾지 못했음을 나타내지만, 예외가 throw된 이유를 나타내지는 않습니다. 그 이유는 과도한 역추적일 수 있지만 예외가 throw될 때 시스템 부하를 감안할 때 제한 시간 간격이 너무 낮게 설정되었을 수도 있습니다. 예외를 처리할 때 입력 문자열을 포함한 다른 일치 항목을 버리거나 시간 제한 간격을 늘리고 일치 검사 작업을 재시도하는 방법 중에서 선택할 수 있습니다.

예를 들어 다음 코드는 생성자를 호출 Regex(String, RegexOptions, TimeSpan) 하여 시간 제한 값이 1초인 Regex 개체를 인스턴스화합니다. 줄의 끝에 하나 이상의 "a" 문자가 포함된 하나 이상의 시퀀스와 일치하는 정규식 패턴 `(a+)+$`는 과도한 역추적의 대상이 됩니다. throw RegexMatchTimeoutException 되는 경우 이 예제에서는 제한 시간 값을 최대 3초 간격으로 늘입니다. 그 후에는 패턴 일치를 찾는 시도를 취소합니다.

C#

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Security;
using System.Text.RegularExpressions;
using System.Threading;

public class Example
{
    const int MaxTimeoutInSeconds = 3;

    public static void Main()
    {
        string pattern = @"(a+)+$";    // DO NOT REUSE THIS PATTERN.
        Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase,
TimeSpan.FromSeconds(1));
        Stopwatch? sw = null;

        string[] inputs = { "aa", "aaaa>",
                            "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                            "aaaaaaaaaaaaaaaaaaaa>",
                            "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>" };

        foreach (var inputValue in inputs)
        {
            Console.WriteLine("Processing {0}", inputValue);
            bool timedOut = false;
            do
            {
                try
                {
                    sw = Stopwatch.StartNew();
                    // Display the result.
                    if (rgx.IsMatch(inputValue))
                    {
                        sw.Stop();
                        Console.WriteLine(@"Valid: '{0}' ({1:ss\.fffffff}
seconds)",
                                          inputValue, sw.Elapsed);
                    }
                    else
                    {
                        sw.Stop();
                        Console.WriteLine(@"'{0}' is not a valid string.
({1:ss\.fffff} seconds)",
                                          inputValue, sw.Elapsed);
                    }
                }
                catch (RegexMatchTimeoutException e)
                {
                    sw.Stop();
                    // Display the elapsed time until the exception.
```

```
                    Console.WriteLine(@"Timeout with '{0}' after
{1:ss\.fffff}",
                                      inputValue, sw.Elapsed);
                    Thread.Sleep(1500);       // Pause for 1.5 seconds.

                    // Increase the timeout interval and retry.
                    TimeSpan timeout =
e.MatchTimeout.Add(TimeSpan.FromSeconds(1));
                    if (timeout.TotalSeconds > MaxTimeoutInSeconds)
                    {
                        Console.WriteLine("Maximum timeout interval of {0}
seconds exceeded.",
                                          MaxTimeoutInSeconds);
                        timedOut = false;
                    }
                    else
                    {
                        Console.WriteLine("Changing the timeout interval to
{0}",
                                          timeout);
                        rgx = new Regex(pattern, RegexOptions.IgnoreCase,
timeout);
                        timedOut = true;
                    }
                }
            } while (timedOut);
            Console.WriteLine();
        }
    }
}
// The example displays output like the following :
//    Processing aa
//    Valid: 'aa' (00.0000779 seconds)
//
//    Processing aaaa>
//    'aaaa>' is not a valid string. (00.00005 seconds)
//
//    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
//    Valid: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' (00.0000043
seconds)
//
//    Processing aaaaaaaaaaaaaaaaaaaa>
//    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 01.00469
//    Changing the timeout interval to 00:00:02
//    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 02.01202
//    Changing the timeout interval to 00:00:03
//    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 03.01043
//    Maximum timeout interval of 3 seconds exceeded.
//
//    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
//    Timeout with 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>' after
03.01018
//    Maximum timeout interval of 3 seconds exceeded.
```

원자성 그룹

`(?>`*하위 식*`)` 언어 요소는 원자성 그룹화입니다. 하위 식으로 역추적을 방지합니다. 이 언어 요소가 성공적으로 일치하면 해당 일치 항목의 일부를 후속 역추적에 포기하지 않습니다. 예를 들어 `(?>\w*\d*)1` 패턴에서 `1`이 일치하지 않는 경우 `\d*`는 `1`을 성공적으로 일치시킬 수 있더라도 일치의 일부를 포기하지 않습니다. 원자성 그룹은 실패한 검색과 연관된 성능 문제를 방지하는 데 유용합니다.

다음 예제에서는 역추적을 억제하여 중첩된 수량자를 사용할 때 성능을 향상시키는 방법을 보여 줍니다. 이 예에서는 정규식 엔진이 입력 문자열이 두 개의 정규식과 일치하지 않는지 확인하기 위해 필요한 시간을 측정합니다. 첫 번째 정규식에서는 역추적을 사용하여 하나 이상의 16진수 숫자와 일치하는 하나 이상의 항목이 포함되고 콜론과 하나 이상의 16진수 숫자 그리고 두 개의 콜론이 이어지는 문자열을 검색하려고 시도합니다. 두 번째 정규식은 첫 번째와 동일하지만 역추적이 사용되지 않습니다. 예의 결과에서 보여 지듯이 역추적을 사용하지 않음으로써 얻게 되는 성능 향상 효과가 매우 큽니다.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example4
{
    public static void Run()
    {
        string input =
"b51:4:1DB:9EE1:5:27d60:f44:D4:cd:E:5:0A5:4a:D24:41Ad:";
        bool matched;
        Stopwatch sw;

        Console.WriteLine("With backtracking:");
        string backPattern = "^(([0-9a-fA-F]{1,4}:)*([0-9a-fA-F]{1,4}))*
(::)$";
        sw = Stopwatch.StartNew();
        matched = Regex.IsMatch(input, backPattern);
        sw.Stop();
        Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input,
backPattern), sw.Elapsed);
        Console.WriteLine();

        Console.WriteLine("Without backtracking:");
        string noBackPattern = "^((?>[0-9a-fA-F]{1,4}:)*(?>[0-9a-fA-F]
{1,4}))*(::)$";
        sw = Stopwatch.StartNew();
        matched = Regex.IsMatch(input, noBackPattern);
        sw.Stop();
        Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input,
noBackPattern), sw.Elapsed);
    }
```

```
}
// The example displays output like the following:
//       With backtracking:
//       Match: False in 00:00:27.4282019
//
//       Without backtracking:
//       Match: False in 00:00:00.0001391
```

Lookbehind 어설션

.NET에는 입력 문자열에서 이전 문자와 일치하는 두 가지 언어 요소인 `(?<=`*subexpression*`)` 및 `(?<!`*subexpression*`)`이 포함되어 있습니다. 두 언어 요소는 모두 너비가 0인 어설션입니다. 즉, 진행 또는 역추적 없이 현재 문자 바로 앞에 있는 문자를 *subexpression*과 일치시킬 수 있는지 여부를 확인합니다.

`(?<=`*subexpression*`)` 은 긍정 lookbehind 어설션입니다. 즉, 현재 위치 바로 전의 문자가 *subexpression*과 일치해야 합니다. `(?<!`*subexpression*`)` 은 부정 lookbehind 어설션입니다. 즉, 현재 위치 바로 전의 문자가 *subexpression*과 일치하면 안 됩니다. 긍정 및 부정 lookbehind 어설션 모두 *subexpression* 이 이전 하위 식의 하위 집합일 때 가장 유용합니다.

다음 예제에서는 전자 메일 주소에서 사용자 이름의 유효성을 검사하는 두 개의 동일한 정규식 패턴이 사용됩니다. 첫 번째 패턴은 과도한 역추적으로 인해 성능이 크게 저하됩니다. 두 번째 패턴은 중첩된 수량자를 긍정 lookbehind 어설션으로 바꿔서 첫 번째 정규식을 수정합니다. 이 예의 결과에는 Regex.IsMatch 메서드의 실행 시간이 표시됩니다.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example5
{
    public static void Run()
    {
        Stopwatch sw;
        string input = "test@contoso.com";
        bool result;

        string pattern = @"^[0-9A-Z]([-.\w]*[0-9A-Z])?@";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("Match: {0} in {1}", result, sw.Elapsed);

        string behindPattern = @"^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@";
        sw = Stopwatch.StartNew();
```

```
        result = Regex.IsMatch(input, behindPattern,
RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("Match with Lookbehind: {0} in {1}", result,
sw.Elapsed);
    }
}
// The example displays output similar to the following:
//       Match: True in 00:00:00.0017549
//       Match with Lookbehind: True in 00:00:00.0000659
```

첫 번째 정규식 패턴 `^[0-9A-Z]([-.\w]*[0-9A-Z])*@`은(는) 다음 표와 같이 정의됩니다.

패턴	설명
`^`	문자열의 시작 부분에서 검색을 시작합니다.
`[0-9A-Z]`	일치하는 영숫자 문자를 찾습니다. Regex.IsMatch 메서드가 RegexOptions.IgnoreCase 옵션으로 호출되므로 이 비교는 대/소문자를 구분하지 않습니다.
`[-.\w]*`	하이픈, 마침표 또는 단어 문자가 0개 이상 일치하는 항목을 찾습니다.
`[0-9A-Z]`	일치하는 영숫자 문자를 찾습니다.
`([-.\w]*[0-9A-Z])*`	영숫자 문자로 이어지는 하이픈, 마침표 또는 단어 문자가 0개 이상 조합된 일치하는 항목을 찾습니다. 이 그룹은 첫 번째 캡처링 그룹입니다.
`@`	"@" 기호를 찾습니다.

두 번째 정규식 패턴 `^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@`은 긍정 lookbehind 어설션을 사용합니다. 이 패턴은 다음 표에서와 같이 정의됩니다.

패턴	설명
`^`	문자열의 시작 부분에서 검색을 시작합니다.
`[0-9A-Z]`	일치하는 영숫자 문자를 찾습니다. Regex.IsMatch 메서드가 RegexOptions.IgnoreCase 옵션으로 호출되므로 이 비교는 대/소문자를 구분하지 않습니다.
`[-.\w]*`	하이픈, 마침표 또는 단어 문자가 0개 이상 포함된 일치하는 항목을 찾습니다.
`(?<=[0-9A-Z])`	마지막으로 일치한 문자를 다시 확인하고 영숫자인 경우 검색을 계속합니다. 영숫자 문자는 마침표, 하이픈 및 모든 단어 문자로 구성되는 집합의 하위 집합입니다.
`@`	"@" 기호를 찾습니다.

Lookahead 어설션

.NET에는 입력 문자열에서 다음 문자와 일치하는 두 가지 언어 요소인 `(?=subexpression)` 및 `(?!subexpression)`이 포함되어 있습니다. 두 언어 요소 모두 너비가 0인 어설션입니다. 즉, 진행 또는 역추적 없이 현재 문자 바로 뒤에 있는 문자를 *subexpression*과 일치시킬 수 있는지 여부를 확인합니다.

`(?=subexpression)` 은 긍정 lookahead 어설션입니다. 즉, 현재 위치 바로 뒤의 문자가 *subexpression*과 일치해야 합니다. `(?!subexpression)` 은 부정 lookahead 어설션입니다. 즉, 현재 위치 바로 뒤의 문자가 *subexpression*과 일치하면 안 됩니다. 긍정 및 부정 lookahead 어설션 모두 *subexpression* 이 다음 하위 식의 하위 집합인 경우 가장 유용합니다.

다음 예제에서는 정규화된 형식 이름의 유효성을 검사하는 두 개의 동일한 정규식 패턴이 사용됩니다. 첫 번째 패턴은 과도한 역추적으로 인해 성능이 크게 저하됩니다. 두 번째 패턴은 중첩된 수량자를 긍정 lookahead 어설션으로 바꿔서 첫 번째 정규식을 수정합니다. 이 예의 결과에는 Regex.IsMatch 메서드의 실행 시간이 표시됩니다.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example6
{
    public static void Run()
    {
        string input = "aaaaaaaaaaaaaaaaaaaaa.";
        bool result;
        Stopwatch sw;

        string pattern = @"^(([A-Z]\w*)+\.)*[A-Z]\w*$";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("{0} in {1}", result, sw.Elapsed);

        string aheadPattern = @"^((?=[A-Z])\w+\.)*[A-Z]\w*$";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, aheadPattern,
RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("{0} in {1}", result, sw.Elapsed);
    }
}
// The example displays the following output:
//       False in 00:00:03.8003793
//       False in 00:00:00.0000866
```

첫 번째 정규식 패턴 `^(([A-Z]\w*)+\.)*[A-Z]\w*$`은(는) 다음 표와 같이 정의됩니다.

패턴	설명
`^`	문자열의 시작 부분에서 검색을 시작합니다.
`([A-Z]\w*)+\.`	0개 이상의 단어 문자와 마침표가 이어지는 영문자(A-Z)를 찾습니다. Regex.IsMatch 메서드가 RegexOptions.IgnoreCase 옵션으로 호출되므로 이 비교는 대/소문자를 구분하지 않습니다.
`(([A-Z]\w*)+\.)*`	이전 패턴을 0번 이상 검색합니다.
`[A-Z]\w*`	0개 이상의 단어 문자로 이어지는 영문자를 찾습니다.
`$`	입력 문자열의 끝 부분에서 검색을 종료합니다.

두 번째 정규식 패턴 `^((?=[A-Z])\w+\.)*[A-Z]\w*$`에는 긍정 lookahead 어설션이 사용됩니다. 이 패턴은 다음 표에서와 같이 정의됩니다.

패턴	설명
`^`	문자열의 시작 부분에서 검색을 시작합니다.
`(?=[A-Z])`	첫 번째 문자를 검색하고 영문자(A-Z)인 경우 검색을 계속합니다. Regex.IsMatch 메서드가 RegexOptions.IgnoreCase 옵션으로 호출되므로 이 비교는 대/소문자를 구분하지 않습니다.
`\w+\.`	마침표로 이어지는 하나 이상의 문자를 검색합니다.
`((?=[A-Z])\w+\.)*`	마침표가 0번 이상 이어지는 하나 이상의 단어 문자의 패턴을 검색합니다. 초기 단어 문자는 영문자여야 합니다.
`[A-Z]\w*`	0개 이상의 단어 문자로 이어지는 영문자를 찾습니다.
`$`	입력 문자열의 끝 부분에서 검색을 종료합니다.

일반적인 성능 고려 사항

다음 제안은 과도한 역추적을 방지하기 위한 것이 아니라 정규식의 성능을 향상시키는 데 도움이 될 수 있습니다.

1. 많이 사용되는 패턴을 미리 컴파일합니다. 이 작업을 수행하는 가장 좋은 방법은 정규식 원본 생성기를 사용하여 미리 컴파일하는 것입니다. 원본 생성기를 앱에 사용할 수 없는 경우(예: .NET 7 이상을 대상으로 하지 않거나 컴파일 시간에 패턴을 모르는 경우) 옵션을 사용합니다 RegexOptions.Compiled .

2. 많이 사용되는 Regex 개체를 캐시합니다. 이는 소스 생성기를 사용할 때 암시적으로 발생합니다. 그렇지 않은 경우 정적 Regex 메서드를 사용하거나 Regex 개체를 만들고 버리지 않고 Regex 개체를 만들고 재사용하기 위해 저장합니다.

3. 오프셋에서 일치를 시작합니다. 일치 항목이 항상 특정 오프셋을 넘어 패턴으로 시작된다는 것을 알고 있는 경우 다음과 같은 Regex.Match(String, Int32)오버로드를 사용하여 오프셋을 전달합니다. 이렇게 하면 엔진에서 고려해야 하는 텍스트의 양이 줄어듭니다.

4. 필요한 정보만 수집합니다. 일치 항목이 발생하는지 여부만 알아야 하지만 일치하는 항목이 발생하는 위치는 알 필요가 없는 경우 다음을 사용하는 것이 좋습니다 Regex.IsMatch. 일치하는 횟수만 알아야 하는 경우 .Regex.Count 일치 항목의 범위만 알아야 하지만 일치 항목의 캡처에 대해서는 아무것도 모르는 경우 .Regex.EnumerateMatches 엔진이 제공해야 하는 정보가 적을수록 좋습니다.

5. 불필요한 캡처를 방지합니다. 패턴의 괄호는 기본적으로 캡처링 그룹을 형성합니다. 캡처가 필요하지 않은 경우 캡처하지 않는 그룹을 대신 지정 RegexOptions.ExplicitCapture 하거나 사용합니다. 이렇게 하면 엔진이 해당 캡처를 추적할 수 있습니다.

참고 항목

- .NET 정규식
- 정규식 언어 - 빠른 참조
- 수량자
- Alternation Constructs
- 정규식의 그룹화 구문

정규식의 컴파일 및 다시 사용

2025. 06. 17.

정규식 엔진이 식을 컴파일하는 방법과 정규식이 캐시되는 방법을 이해하면 정규식을 광범위하게 사용하는 애플리케이션의 성능을 최적화할 수 있습니다. 이 문서에서는 컴파일, 원본 생성 및 컴파일된 정규식 캐싱에 대해 설명합니다.

해석된 정규식

기본적으로 정규식 엔진은 정규식을 내부 명령 시퀀스(Common Intermediate Language(CIL)와 다른 고급 코드)로 컴파일합니다. 엔진은 정규식을 실행할 때 내부 코드를 해석합니다.

컴파일된 정규식

Regex 옵션으로 RegexOptions.Compiled 개체를 생성하는 경우 정규식이 고급 정규식 내부 명령 대신 명시적 CIL 코드로 컴파일됩니다. 이렇게 하면 .NET의 JIT(Just-In-Time) 컴파일러가 성능 향상을 위해 식을 네이티브 기계어 코드로 변환할 수 있습니다. Regex 개체를 구성하는 비용이 더 높을 수 있지만, 이를 사용하여 일치를 수행하는 비용이 훨씬 더 적을 수 있습니다.

소스 생성 정규식

정규식의 원본 생성은 .NET 7 이상 버전에서 사용할 수 있습니다. 원본 생성기는 IL에서 `Regex`가 내보내는 것과 유사한 논리를 사용하여 사용자 지정 `RegexOptions.Compiled` 파생 구현을 C# 코드로 내보냅니다. `RegexOptions.Compiled`의 처리량 성능 이점과 `Regex.CompileToAssembly`의 시작 이점을 모두 가져올 수 있지만 `CompileToAssembly`의 복잡성은 없습니다. 내보내는 원본은 프로젝트의 일부이므로 쉽게 보고 디버그할 수도 있습니다.

가능하다면 RegexOptions.Compiled 옵션을 사용하여 정규식을 컴파일하는 대신 소스에서 생성된 정규식을 사용하세요. 소스 생성 정규식에 대한 자세한 내용은 .NET 정규식 소스 생성기를 참조하세요.

정규식 캐시

성능 향상을 위해 정규식 엔진은 애플리케이션 수준의 컴파일된 정규식 캐시를 유지 관리합니다. 캐시는 정적 메서드 호출에만 사용되는 정규식 패턴을 저장합니다. (인스턴스 메서드에 제공된 정규식 패턴은 캐시되지 않습니다.) 캐싱을 사용하면 사용할 때마다 식을 개략적인 바이트 코드로 다시 분석할 필요가 없습니다.

캐시된 정규식의 최대 개수는 `static`(Visual Basic의 경우 `Shared`) Regex.CacheSize 속성 값에 의해 결정됩니다. 기본적으로 정규식 엔진은 최대 15개의 컴파일된 정규식을 캐시합니다. 컴파일된 정규식 개수가 캐시 크기를 초과하면 가장 오래 전에 사용한 정규식이 삭제되고 새 정규식이 캐시됩니다.

애플리케이션은 다음 두 가지 방법 중 하나를 사용하여 정규식을 재사용할 수 있습니다.

- Regex 개체의 정적 메서드를 사용하여 정규식을 정의합니다. 다른 정적 메서드 호출에서 이미 정의된 정규식 패턴을 사용하는 경우 정규식 엔진이 캐시에서 이를 검색합니다. 캐시에 없으면 엔진이 정규식을 컴파일하고 캐시에 추가합니다.
- 해당 정규식 패턴이 필요할 때까지 기존 Regex 개체를 다시 사용합니다.

개체 인스턴스화 및 정규식 컴파일의 오버헤드 때문에 수많은 Regex 개체를 만들고 금세 삭제할 경우 큰 비용이 듭니다. 다수의 정규식을 사용하는 애플리케이션의 경우 정적 `Regex` 메서드 호출을 사용하고 정규식 캐시의 크기를 늘려 성능을 최적화할 수 있습니다.

참고 항목

- .NET 정규식 원본 생성기
- .NET 정규식

.NET 정규식 소스 생성기

정규식은 개발자가 검색되는 패턴을 표현할 수 있는 문자열로, 검색된 문자열에서 텍스트를 검색하고 결과를 하위 집합으로 추출하는 일반적인 방법입니다. .NET에서 `System.Text.RegularExpressions` 네임스페이스는 Regex 인스턴스 및 정적 메서드를 정의하고 사용자 정의 패턴과 일치시키는 데 사용됩니다. 이 문서에서는 소스 생성을 사용하여 `Regex` 인스턴스를 생성해 성능을 최적화하는 방법을 알아봅니다.

> **참고 항목**
>
> 가능하다면 RegexOptions.Compiled 옵션을 사용하여 정규식을 컴파일하는 대신 소스에서 생성된 정규식을 사용하세요. 원본 생성을 통해 앱을 더 빠르게 시작하고, 더 빠르게 실행하고, 더 간편하게 다듬을 수 있습니다. 원본 생성이 가능한 시기를 알아보려면 **사용 시기**를 참조하세요.

컴파일된 정규식

`new Regex("somepattern")`를 작성하면 몇 가지 프로세스가 발생합니다. 지정된 패턴이 구문 분석됩니다. 이는 패턴의 유효성을 보장하고 구문 분석된 정규식을 나타내는 내부 트리로 변환하기 위한 것입니다. 그런 다음 트리가 다양한 방법으로 최적화되어 패턴을 보다 효율적으로 실행할 수 있는 기능적으로 동등한 변형으로 변환합니다. 트리는 정규식 인터프리터 엔진에 일치 방법에 대한 명령을 제공하는 일련의 opcode 및 피연산자로 해석될 수 있는 양식으로 작성됩니다. 일치가 수행되면 인터프리터는 입력 텍스트에 대해 이러한 명령을 처리합니다. 인터프리터는 새 `Regex` 인스턴스를 인스턴스화하거나 `Regex`에서 정적 메서드 중 하나를 호출할 때 사용하는 기본 엔진입니다.

RegexOptions.Compiled를 지정하면 동일한 구문 시간 작업이 모두 수행됩니다. 결과 명령은 반사 방출 기반 컴파일러에 의해 몇 가지 DynamicMethod 개체에 기록되는 IL 명령으로 추가로 변환됩니다. 일치가 수행되면 해당 `DynamicMethod` 메서드가 호출됩니다. 이 IL은 처리되는 정확한 패턴에 특화된 것을 제외하고 기본적으로 인터프리터가 수행하는 작업을 정확하게 수행합니다. 예를 들어, 패턴에 `[ac]`가 포함된 경우 인터프리터는 "이 집합 설명에 지정된 집합과 현재 위치의 입력 문자를 일치시킵니다"라는 opcode를 보게 됩니다. 반면 컴파일된 IL에는 "현재 위치의 입력 문자를 `'a'` 또는 `'c'`와 일치시킵니다"라고 효과적으로 말하는 코드가 포함됩니다. 이 특수 대/소문자 구분 및 패턴 지식을 기반으로 최적화를 수행하는 기능은 `RegexOptions.Compiled` 지정이 인터프리터보다 훨씬 빠르게 일치를 처리하는 주된 이유 중 일부입니다.

`RegexOptions.Compiled`에는 몇 가지 단점이 있습니다. 가장 큰 영향은 구성하는 데 비용이 많이 든다는 것입니다. 인터프리터와 동일한 비용은 물론이고 결과 `RegexNode` 트리 및 생성된 opcode/피연산자를 IL로 컴파일하기 위한 적지 않은 비용이 추가됩니다. 생성된 IL은 처음 사용할 때 JIT를 추가로 컴파일해야 하므로 시작 시 더 많은 비용이 발생합니다.

`RegexOptions.Compiled`는 처음 사용할 때의 오버헤드와 이후의 모든 사용에 대한 오버헤드 간의 근본적인 절충을 나타냅니다. System.Reflection.Emit의 사용은 또한 특정 환경에서 `RegexOptions.Compiled`의 사용을 억제합니다. 일부 운영 체제는 동적으로 생성된 코드를 실행할 수 없으므로 이러한 시스템에서는 `Compiled`가 작동하지 않습니다.

원본 생성

.NET 7에는 새 `RegexGenerator` 소스 생성기가 도입되었습니다. *원본 생성기*는 컴파일러에 연결되고 추가 소스 코드로 컴파일 단위를 확장하는 구성 요소입니다. .NET SDK(버전 7 이상)에는 GeneratedRegexAttribute를 반환하는 부분 메서드에서 `Regex` 특성을 인식하는 원본 생성기가 포함되어 있습니다. 원본 생성기는 `Regex`에 대한 모든 논리를 포함하는 해당 메서드의 구현을 제공합니다. 예를 들어, 이전에 다음과 같은 코드를 작성했을 수 있습니다.

C#

```
private static readonly Regex s_abcOrDefGeneratedRegex =
    new(pattern: "abc|def",
        options: RegexOptions.Compiled | RegexOptions.IgnoreCase);

private static void EvaluateText(string text)
{
    if (s_abcOrDefGeneratedRegex.IsMatch(text))
    {
        // Take action with matching text
    }
}
```

원본 생성기를 사용하려면 이전 코드를 다음과 같이 다시 작성합니다.

C#

```
[GeneratedRegex("abc|def", RegexOptions.IgnoreCase, "en-US")]
private static partial Regex AbcOrDefGeneratedRegex();

private static void EvaluateText(string text)
{
    if (AbcOrDefGeneratedRegex().IsMatch(text))
    {
        // Take action with matching text
    }
}
```

팁

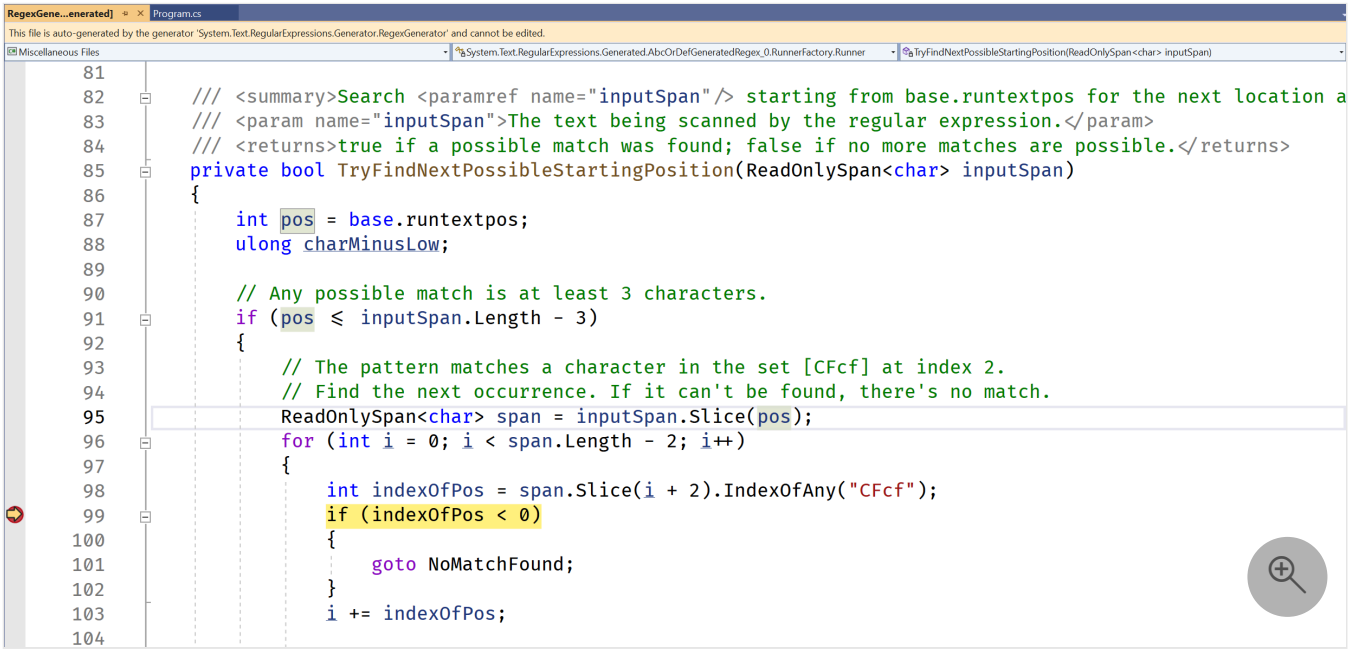
`RegexOptions.Compiled` 플래그는 원본 생성기에서 무시되므로 원본 생성 버전에서는 필요하지 않습니다.

생성된 `AbcOrDefGeneratedRegex()` 구현은 마찬가지로 싱글톤 `Regex` 인스턴스를 캐시하므로 코드를 사용하는 데 추가 캐싱이 필요하지 않습니다.

다음 이미지는 소스 생성 캐시 인스턴스, `internal`에서 소스 생성기가 방출하는 `Regex` 하위 클래스의 화면 캡처입니다.

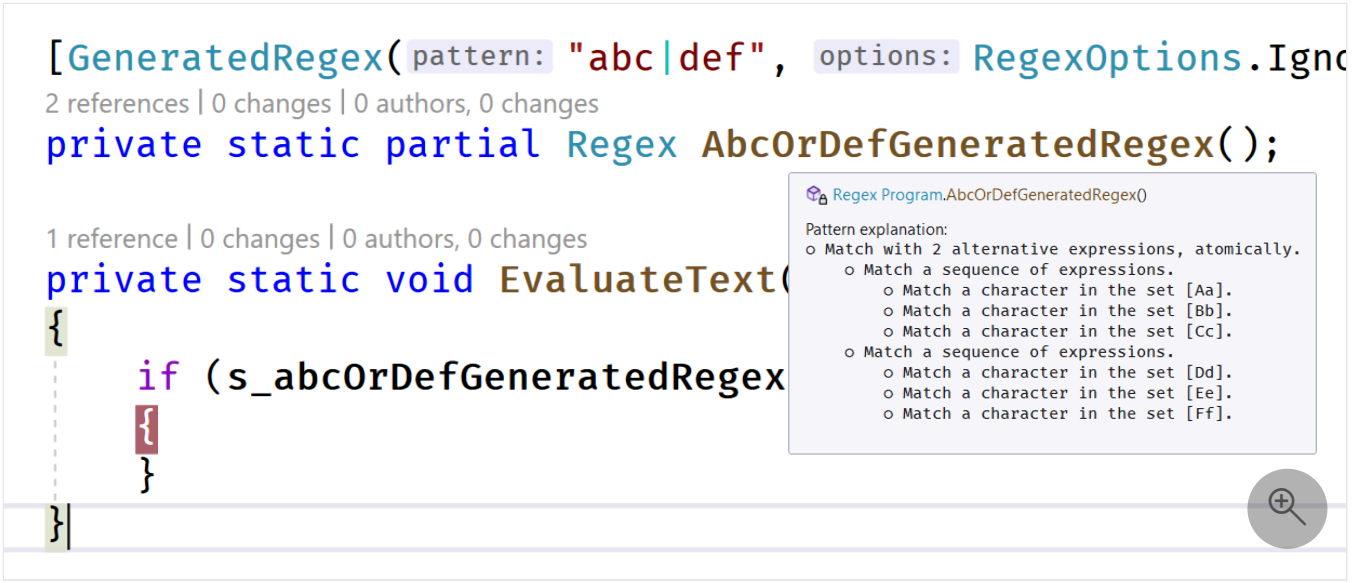
```
/// <summary>Cached, thread-safe singleton instance.</summary>
internal static readonly AbcOrDefGeneratedRegex_0 Instance = new AbcOrDefGeneratedRegex_0();
```

그러나 보다시피 `new Regex(...)`만 수행하는 것이 아닙니다. 대신 소스 생성기는 IL에서 `Regex`가 내보내는 것과 유사한 논리를 사용하여 C# 코드로 사용자 지정 `RegexOptions.Compiled` 파생 구현을 내보냅니다. `RegexOptions.Compiled`의 모든 처리량 성능 이점(실제로는 그 이상)과 `Regex.CompileToAssembly`의 시작 이점을 얻으면서도 `CompileToAssembly`의 복잡성은 없습니다. 내보내는 원본은 프로젝트의 일부이므로 쉽게 보고 디버그할 수도 있습니다.



> **팁**
>
> Visual Studio에서 부분 메서드 선언을 마우스 오른쪽 단추로 클릭하고 **정의로 이동**을 선택합니다. 또는 **솔루션 탐색기**에서 프로젝트 노드를 선택한 다음 **종속성**>**분석기**>**System.Text.RegularExpressions.Generator**>**System.Text.RegularExpressions.Generator.RegexGenerator**>*RegexGenerator.g.cs*를 확장하여 이 정규식 생성기에서 생성된 C# 코드를 확인합니다.

중단점을 설정하고, 한 단계씩 실행할 수 있으며, 학습 도구로 사용하여 정규식 엔진이 입력에서 패턴을 처리하는 방법을 정확하게 이해할 수 있습니다. 생성기는 식의 내용과 사용되는 위치를 한눈에 파악할 수 있도록 XML(트리플 슬래시) 주석을 생성합니다.



소스 생성 파일 내부

.NET 7에서는 소스 생성기 및 `RegexCompiler`가 모두 거의 완전히 다시 작성되어 생성되는 코드의 구조가 근본적으로 달라졌습니다. 이 접근 방식은 모든 구문을 처리하도록 확장되었으며(한 가지 주의 사항), 새 접근 방식에 따라 여전히 `RegexCompiler`와 소스 생성기가 대부분 1:1로 매핑됩니다. `abc|def` 식의 주 함수 중 하나에 대한 소스 생성기 출력을 예로 들어보겠습니다.

C#

```
private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextpos;
    int matchStart = pos;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);

    // Match with 2 alternative expressions, atomically.
    {
        if (slice.IsEmpty)
        {
            return false; // The input didn't match.
        }

        switch (slice[0])
        {
            case 'A' or 'a':
                if ((uint)slice.Length < 3 ||
                    !slice.Slice(1).StartsWith("bc",
StringComparison.OrdinalIgnoreCase)) // Match the string "bc" (ordinal case-
insensitive)
                {
```

```
                    return false; // The input didn't match.
                }

                pos += 3;
                slice = inputSpan.Slice(pos);
                break;

            case 'D' or 'd':
                if ((uint)slice.Length < 3 ||
                    !slice.Slice(1).StartsWith("ef",
StringComparison.OrdinalIgnoreCase)) // Match the string "ef" (ordinal case-
insensitive)
                {
                    return false; // The input didn't match.
                }

                pos += 3;
                slice = inputSpan.Slice(pos);
                break;

            default:
                return false; // The input didn't match.
        }
    }

    // The input matched.
    base.runtextpos = pos;
    base.Capture(0, matchStart, pos);
    return true;
}
```

소스 생성 코드의 목표는 따라가기 쉬운 구조, 각 단계에서 수행되는 작업을 설명하는 주석, 사람이 작성한 것과 같은 코드 등 이해하기 용이해야 한다는 것입니다. 역추적이 관련된 경우에도 역추적 구조는 스택을 사용하여 다음에 이동할 위치를 나타내는 대신 코드 구조의 일부가 됩니다. 예를 들어 식이 `[ab]*[bc]`일 때 생성된 것과 동일한 일치 함수의 코드는 다음과 같습니다.

C#

```
private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextpos;
    int matchStart = pos;
    int charloop_starting_pos = 0, charloop_ending_pos = 0;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);

    // Match a character in the set [ABab] greedily any number of times.
    //{
        charloop_starting_pos = pos;

        int iteration = slice.IndexOfAnyExcept(Utilities.s_ascii_600000006000000);
        if (iteration < 0)
        {
```

```
            iteration = slice.Length;
        }

        slice = slice.Slice(iteration);
        pos += iteration;

        charloop_ending_pos = pos;
        goto CharLoopEnd;

        CharLoopBacktrack:

        if (Utilities.s_hasTimeout)
        {
            base.CheckTimeout();
        }

        if (charloop_starting_pos >= charloop_ending_pos ||
            (charloop_ending_pos = inputSpan.Slice(charloop_starting_pos,
charloop_ending_pos -
charloop_starting_pos).LastIndexOfAny(Utilities.s_ascii_C0000000C000000)) < 0)
        {
            return false; // The input didn't match.
        }
        charloop_ending_pos += charloop_starting_pos;
        pos = charloop_ending_pos;
        slice = inputSpan.Slice(pos);

        CharLoopEnd:
    //}

    // Advance the next matching position.
    if (base.runtextpos < pos)
    {
        base.runtextpos = pos;
    }

    // Match a character in the set [BCbc].
    if (slice.IsEmpty || ((uint)((slice[0] | 0x20) - 'b') > (uint)('c' - 'b')))
    {
        goto CharLoopBacktrack;
    }

    // The input matched.
    pos++;
    base.runtextpos = pos;
    base.Capture(0, matchStart, pos);
    return true;
}
```

역추적 위치로 내보내는 `CharLoopBacktrack` 레이블과 정규식의 후속 부분이 실패할 때 해당 위치로 이동하는 데 사용되는 `goto`를 사용하여 코드에서 역추적 구조를 확인할 수 있습니다.

코드를 구현하는 `RegexCompiler`와 소스 생성기를 살펴보면 비슷한 이름의 메서드, 유사한 호출 구조, 구현 전체에서 유사한 주석 등 매우 비슷하게 표시됩니다. 대부분의 경우 IL 및 C#로 작성되었다는 차이를 제외하면 동일한 코드입니다. 물론 C# 컴파일러는 C#을 IL로 변환해야 하므로 두 경우 모두 결과 IL이 동일하지 않을 수 있습니다. 소스 생성기는 C# 컴파일러가 다양한 C# 구문을 추가로 최적화한다는 장점 때문에 다양한 경우에 C# 컴파일러를 사용합니다. 소스 생성기가 `RegexCompiler`보다 최적화된 일치 코드를 생성하는 몇 가지 구체적인 경우가 있습니다. 예를 들어 이전 예제 중 하나에서 switch 문을 내보내는 소스 생성기가 있습니다. `'a'`에 대한 분기 하나와 `'b'`에 대한 또 다른 분기가 있습니다. C# 컴파일러는 switch 문을 최적화하는 데 매우 능숙하여 여러 전략을 사용하여 처리할 수 있으므로 소스 생성기에는 `RegexCompiler`에 없는 특수 최적화가 있습니다. 변경의 경우 소스 생성기는 모든 분기를 살펴보고 모든 분기가 다른 시작 문자로 시작된다는 것을 증명할 수 있는 경우 첫 번째 문자에 대한 switch 문을 내보내고 해당 변경에 대한 역추적 코드를 출력하지 않습니다.

여기에 약간 더 복잡한 예가 있습니다. 역추적 엔진에서 보다 쉽게 최적화할 수 있고 소스 생성 코드가 더 간단해지는 방식으로 리팩터링할 수 있는지 여부를 확인하기 위해 변경을 더 집중적으로 분석합니다. 이러한 최적화 중 하나는 분기에서 공통 접두사를 추출하도록 지원하고 변경이 원자성이어서 순서가 중요하지 않은 경우 더 많은 추출을 허용하도록 분기를 다시 정렬합니다. 다음과 같은 일치 함수를 생성하는, 다음 요일 패턴 `Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday`대한 영향을 확인할 수 있습니다.

C#

```
private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextpos;
    int matchStart = pos;
    char ch;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);

    // Match with 6 alternative expressions, atomically.
    {
        int alternation_starting_pos = pos;

        // Branch 0
        {
            if ((uint)slice.Length < 6 ||
                !slice.StartsWith("monday", StringComparison.OrdinalIgnoreCase)) //
Match the string "monday" (ordinal case-insensitive)
            {
                goto AlternationBranch;
            }

            pos += 6;
            slice = inputSpan.Slice(pos);
            goto AlternationMatch;

            AlternationBranch:
            pos = alternation_starting_pos;
```

```
            slice = inputSpan.Slice(pos);
        }

        // Branch 1
        {
            if ((uint)slice.Length < 7 ||
                !slice.StartsWith("tuesday", StringComparison.OrdinalIgnoreCase)) //
Match the string "tuesday" (ordinal case-insensitive)
            {
                goto AlternationBranch1;
            }

            pos += 7;
            slice = inputSpan.Slice(pos);
            goto AlternationMatch;

            AlternationBranch1:
            pos = alternation_starting_pos;
            slice = inputSpan.Slice(pos);
        }

        // Branch 2
        {
            if ((uint)slice.Length < 9 ||
                !slice.StartsWith("wednesday", StringComparison.OrdinalIgnoreCase))
// Match the string "wednesday" (ordinal case-insensitive)
            {
                goto AlternationBranch2;
            }

            pos += 9;
            slice = inputSpan.Slice(pos);
            goto AlternationMatch;

            AlternationBranch2:
            pos = alternation_starting_pos;
            slice = inputSpan.Slice(pos);
        }

        // Branch 3
        {
            if ((uint)slice.Length < 8 ||
                !slice.StartsWith("thursday", StringComparison.OrdinalIgnoreCase))
// Match the string "thursday" (ordinal case-insensitive)
            {
                goto AlternationBranch3;
            }

            pos += 8;
            slice = inputSpan.Slice(pos);
            goto AlternationMatch;

            AlternationBranch3:
            pos = alternation_starting_pos;
            slice = inputSpan.Slice(pos);
```

```
        }

        // Branch 4
        {
            if ((uint)slice.Length < 6 ||
                !slice.StartsWith("fr", StringComparison.OrdinalIgnoreCase) || //
Match the string "fr" (ordinal case-insensitive)
                ((((ch = slice[2]) | 0x20) != 'i') & (ch != 'İ')) || // Match a
character in the set [Ii\u0130].
                !slice.Slice(3).StartsWith("day",
StringComparison.OrdinalIgnoreCase)) // Match the string "day" (ordinal case-
insensitive)
            {
                goto AlternationBranch4;
            }

            pos += 6;
            slice = inputSpan.Slice(pos);
            goto AlternationMatch;

            AlternationBranch4:
            pos = alternation_starting_pos;
            slice = inputSpan.Slice(pos);
        }

        // Branch 5
        {
            // Match a character in the set [Ss].
            if (slice.IsEmpty || ((slice[0] | 0x20) != 's'))
            {
                return false; // The input didn't match.
            }

            // Match with 2 alternative expressions, atomically.
            {
                if ((uint)slice.Length < 2)
                {
                    return false; // The input didn't match.
                }

                switch (slice[1])
                {
                    case 'A' or 'a':
                        if ((uint)slice.Length < 8 ||
                            !slice.Slice(2).StartsWith("turday",
StringComparison.OrdinalIgnoreCase)) // Match the string "turday" (ordinal case-
insensitive)
                        {
                            return false; // The input didn't match.
                        }

                        pos += 8;
                        slice = inputSpan.Slice(pos);
                        break;
```

```
                    case 'U' or 'u':
                        if ((uint)slice.Length < 6 ||
                            !slice.Slice(2).StartsWith("nday",
StringComparison.OrdinalIgnoreCase)) // Match the string "nday" (ordinal case-
insensitive)
                        {
                            return false; // The input didn't match.
                        }

                        pos += 6;
                        slice = inputSpan.Slice(pos);
                        break;

                    default:
                        return false; // The input didn't match.
                }
            }

        }

        AlternationMatch:;
    }

    // The input matched.
    base.runtextpos = pos;
    base.Capture(0, matchStart, pos);
    return true;
}
```

동시에 소스 생성기에는 IL로 직접 출력할 때는 존재하지 않는 다른 문제가 있습니다. 몇 가지 코드 예제를 다시 보면 일부 중괄호가 다소 이상하게 주석 처리된 것을 볼 수 있습니다. 이것은 실수가 아닙니다. 소스 생성기는 이러한 중괄호가 주석 처리되지 않은 경우 역추적 구조가 해당 범위 외부에서 해당 범위 내에 정의된 레이블로 이동하는 데 의존한다고 인식합니다. 이러한 레이블은 `goto`에 표시되지 않으며 코드가 컴파일되지 않습니다. 따라서 소스 생성기는 방해가 되는 범위가 없도록 해야 합니다. 경우에 따라 여기에서와 같이 범위를 주석 처리하기만 하면 됩니다. 이것이 가능하지 않은 다른 경우에는 문제가 될 수 있으면 범위가 필요한 구문(예: 다중 문 `if` 블록)을 사용하지 않습니다.

소스 생성기는 한 가지 예외를 제외하고 모든 `RegexCompiler` 핸들을 처리합니다. `RegexOptions.IgnoreCase` 처리와 마찬가지로 구현은 이제 대/소문자 테이블을 사용하여 생성 시 집합을 생성하고 `IgnoreCase` 역참조 일치가 해당 대/소문자 테이블을 참조하는 데 필요한 방법을 설명합니다. 해당 테이블은 `System.Text.RegularExpressions.dll` 내부에 있으며, 적어도 지금은 해당 어셈블리 외부의 코드(소스 생성기에서 내보낸 코드 포함)가 액세스할 수 없습니다. 따라서 `IgnoreCase` 역참조 처리는 소스 생성기에서 문제가 되며 지원되지 않습니다. 이것이 `RegexCompiler`는 지원하지만 소스 생성기는 지원하지 않는 한 구문입니다. 드물지만 이러한 패턴 중 하나를 사용하려고 하면 소스 생성기가 사용자 지정 구현을 내보내는 대신 일반 `Regex` 인스턴스 캐싱으로 대체됩니다.

```
/// <summary>Cached, thread-safe singleton instance.</summary>
internal static readonly Regex Instance =
    new Regex(pattern: "(\\w)\\1", options: RegexOptions.IgnoreCase);
```

또한 소스 생성기와 `RegexCompiler` 모두 새 `RegexOptions.NonBacktracking`을 지원하지 않습니다. `RegexOptions.Compiled | RegexOptions.NonBacktracking`을 지정하면 `Compiled` 플래그는 무시되고, 소스 생성기에 `NonBacktracking`을 지정하면 일반 `Regex` 인스턴스 캐싱으로 대체됩니다.

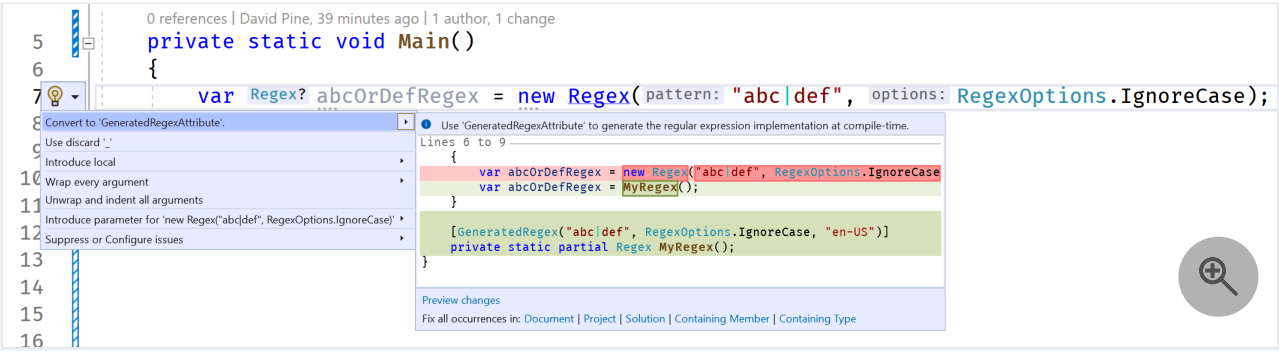
사용하는 경우

일반적인 참고 자료는 소스 생성기를 사용할 수 있는 경우라면 사용하라는 것입니다. 현재 C#에서 `Regex`를 컴파일 시간에 알려진 인수와 함께 사용 중이고, 특히 (정규식이 더 빠른 처리량의 이점을 얻을 수 있는 핫스팟으로 식별되었기 때문에) 이미 `RegexOptions.Compiled`를 사용하고 있는 경우 소스 생성기를 사용하는 것이 좋습니다. 소스 생성기는 정규식에 다음과 같은 이점을 제공합니다.

- `RegexOptions.Compiled`의 모든 처리량 이점.
- 런타임에 모든 정규식 구문 분석, 분석 및 컴파일을 수행할 필요가 없다는 시작의 이점입니다.
- 정규식에 대해 생성된 코드와 함께 미리 컴파일을 사용하는 옵션.
- 정규식에 대한 향상된 디버그 기능 및 이해.
- `RegexCompiler`와 연결된 코드를 잘라내어 앱의 크기를 줄일 수 있는 가능성(또한 리플렉션 내보내기 자체에서도 가능).

소스 생성기가 사용자 지정 구현을 생성할 수 없는, `RegexOptions.NonBacktracking`과 같은 옵션을 함께 사용하는 경우에도 구현을 설명하는 캐싱 및 XML 주석을 계속 내보내므로 유용합니다. 소스 생성기의 주요 단점은 어셈블리에 추가 코드를 내보내므로 크기가 증가할 가능성이 있다는 것입니다. 앱에 정규식이 더 많을수록 더 많은 코드가 내보내집니다. 어떤 상황에서는 `RegexOptions.Compiled`가 불필요할 수 있는 것처럼 원본 생성기도 불필요할 수 있습니다. 예를 들어 거의 필요하지 않고 처리량이 중요하지 않은 정규식이 있는 경우 이 산발적인 사용에 인터프리터만 사용하는 것이 더 유용할 수 있습니다.

> **ⓘ Important**
>
> .NET 7에는 소스 생성기로 변환할 수 있는 의 사용을 식별하는 `Regex`와 자동으로 변환을 수행하는 수정 도구가 포함되어 있습니다.



추가 정보

- 정규식 소스 생성을 위한 SYSLIB 진단
- .NET 정규식
- 정규식의 역추적
- 정규식의 컴파일 및 다시 사용
- 소스 생성기
- .NET 블로그: .NET 7의 정규식 개선 사항

Last updated on 2025. 10. 20.

.NET의 정규식에 대한 모범 사례

2025. 06. 17.

.NET의 정규식 엔진은 리터럴 텍스트를 비교하고 일치시키는 대신 패턴 일치를 기반으로 텍스트를 처리하는 강력한 전체 기능을 갖춘 도구입니다. 대부분의 경우 패턴 일치를 신속하고 효율적으로 수행합니다. 그러나 경우에 따라 정규식 엔진이 느린 것처럼 보일 수 있습니다. 극단적인 경우 몇 시간 또는 며칠 동안 비교적 작은 입력을 처리하므로 응답을 중지하는 것처럼 보일 수도 있습니다.

이 문서에서는 정규식이 최적의 성능을 달성할 수 있도록 개발자가 채택할 수 있는 몇 가지 모범 사례를 간략하게 설명합니다.

> ⚠ **경고**
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions` 입력을 제공하여 **서비스 거부 공격** 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

입력 원본 고려

일반적으로 정규식은 제한되거나 제한되지 않은 두 가지 유형의 입력을 허용할 수 있습니다. 제한된 입력은 알려진 소스 또는 신뢰할 수 있는 원본에서 시작되고 미리 정의된 형식을 따르는 텍스트입니다. 제한되지 않은 입력은 웹 사용자와 같은 신뢰할 수 없는 원본에서 시작되며 미리 정의되거나 예상된 형식을 따르지 않을 수 있는 텍스트입니다.

정규식 패턴은 유효한 입력과 일치하도록 작성되는 경우가 많습니다. 즉, 개발자는 일치시킬 텍스트를 검사한 다음 일치하는 정규식 패턴을 작성합니다. 그런 다음 개발자는 이 패턴을 여러 개의 유효한 입력 항목으로 테스트하여 수정 또는 추가 경과가 필요한지 여부를 결정합니다. 패턴이 유효한 것으로 추정되는 모든 입력과 일치하면 프로덕션 준비로 선언되며 릴리스된 애플리케이션에 포함될 수 있습니다. 이 방법을 사용하면 정규식 패턴이 제한된 입력을 일치시키는 데 적합합니다. 그러나 제한되지 않은 입력을 일치시키는 데 적합하지는 않습니다.

제한되지 않은 입력과 일치하려면 정규식이 세 종류의 텍스트를 효율적으로 처리해야 합니다.

- 정규식 패턴과 일치하는 텍스트입니다.
- 정규식 패턴과 일치하지 않는 텍스트입니다.
- 정규식 패턴과 거의 일치하는 텍스트입니다.

마지막 텍스트 형식은 제한된 입력을 처리하도록 작성된 정규식에 특히 문제가 됩니다. 해당 정규식이 광범위한 역추적에도 의존하는 경우 정규식 엔진은 겉보기에 무해한 텍스트를 처리하는

데 많은 시간(경우에 따라 여러 시간 또는 며칠)을 소비할 수 있습니다.

> **⚠ 경고**
>
> 다음 예제에서는 과도한 역추적이 발생하기 쉽고 유효한 전자 메일 주소를 거부할 가능성이 있는 정규식을 사용합니다. 전자 메일 유효성 검사 루틴에서는 사용하지 않아야 합니다. 전자 메일 주소의 유효성을 검사하는 정규식을 원하는 경우 **방법: 문자열이 유효한 전자 메일 형식인지 확인**합니다.

예를 들어 전자 메일 주소의 별칭의 유효성을 검사하기 위해 일반적으로 사용되지만 문제가 있는 정규식을 고려해 보세요. 정규식 `^[0-9A-Z]([-.\w]*[0-9A-Z])*$` 은 유효한 전자 메일 주소로 간주되는 것을 처리하기 위해 작성됩니다. 유효한 전자 메일 주소는 영숫자 문자로 시작하고, 그 뒤에 올 수 있는 영숫자, 마침표, 또는 하이픈으로 구성된 0개 이상의 문자로 구성됩니다. 정규식은 영숫자 문자로 끝나야 합니다. 그러나 다음 예제와 같이 이 정규식은 유효한 입력을 쉽게 처리하지만 거의 유효한 입력을 처리할 때 성능이 비효율적입니다.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class DesignExample
{
    public static void Main()
    {
        Stopwatch sw;
        string[] addresses = { "AAAAAAAAAA@contoso.com",
                               "AAAAAAAAAAaaaaaaaaaa!@contoso.com" };
        // The following regular expression should not actually be used to
        // validate an email address.
        string pattern = @"^[0-9A-Z]([-.\w]*[0-9A-Z])*$";
        string input;

        foreach (var address in addresses)
        {
            string mailBox = address.Substring(0, address.IndexOf("@"));
            int index = 0;
            for (int ctr = mailBox.Length - 1; ctr >= 0; ctr--)
            {
                index++;

                input = mailBox.Substring(ctr, index);
                sw = Stopwatch.StartNew();
                Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
                sw.Stop();
                if (m.Success)
                    Console.WriteLine("{0,2}. Matched '{1,25}' in {2}",
                                      index, m.Value, sw.Elapsed);
```

```
                else
                    Console.WriteLine("{0,2}. Failed  '{1,25}' in {2}",
                                      index, input, sw.Elapsed);
            }
            Console.WriteLine();
        }
    }
}

// The example displays output similar to the following:
//     1. Matched '                         A' in 00:00:00.0007122
//     2. Matched '                        AA' in 00:00:00.0000282
//     3. Matched '                       AAA' in 00:00:00.0000042
//     4. Matched '                      AAAA' in 00:00:00.0000038
//     5. Matched '                     AAAAA' in 00:00:00.0000042
//     6. Matched '                    AAAAAA' in 00:00:00.0000042
//     7. Matched '                   AAAAAAA' in 00:00:00.0000042
//     8. Matched '                  AAAAAAAA' in 00:00:00.0000087
//     9. Matched '                 AAAAAAAAA' in 00:00:00.0000045
//    10. Matched '                AAAAAAAAAA' in 00:00:00.0000045
//    11. Matched '               AAAAAAAAAAA' in 00:00:00.0000045
//
//     1. Failed  '                         !' in 00:00:00.0000447
//     2. Failed  '                        a!' in 00:00:00.0000071
//     3. Failed  '                       aa!' in 00:00:00.0000071
//     4. Failed  '                      aaa!' in 00:00:00.0000061
//     5. Failed  '                     aaaa!' in 00:00:00.0000081
//     6. Failed  '                    aaaaa!' in 00:00:00.0000126
//     7. Failed  '                   aaaaaa!' in 00:00:00.0000359
//     8. Failed  '                  aaaaaaa!' in 00:00:00.0000414
//     9. Failed  '                 aaaaaaaa!' in 00:00:00.0000758
//    10. Failed  '                aaaaaaaaa!' in 00:00:00.0001462
//    11. Failed  '               aaaaaaaaaa!' in 00:00:00.0002885
//    12. Failed  '              Aaaaaaaaaaa!' in 00:00:00.0005780
//    13. Failed  '             AAaaaaaaaaaa!' in 00:00:00.0011628
//    14. Failed  '            AAAaaaaaaaaaa!' in 00:00:00.0022851
//    15. Failed  '           AAAAaaaaaaaaaa!' in 00:00:00.0045864
//    16. Failed  '          AAAAAaaaaaaaaaa!' in 00:00:00.0093168
//    17. Failed  '         AAAAAAaaaaaaaaaa!' in 00:00:00.0185993
//    18. Failed  '        AAAAAAAaaaaaaaaaa!' in 00:00:00.0366723
//    19. Failed  '       AAAAAAAAaaaaaaaaaa!' in 00:00:00.1370108
//    20. Failed  '      AAAAAAAAAaaaaaaaaaa!' in 00:00:00.1553966
//    21. Failed  '     AAAAAAAAAAaaaaaaaaaa!' in 00:00:00.3223372
```

앞의 예제의 출력에서 알 수 있듯이 정규식 엔진은 길이에 관계없이 거의 동일한 시간 간격으로 유효한 이메일 별칭을 처리합니다. 반면에 거의 유효한 전자 메일 주소에 5자 이상이 있는 경우 처리 시간은 문자열의 각 추가 문자에 대해 약 두 배가 됩니다. 따라서 거의 유효한 28자 문자열은 처리하는 데 1시간이 걸리며 거의 유효한 33자 문자열은 처리하는 데 거의 하루가 걸립니다.

이 정규식은 일치시킬 입력 형식을 고려하여 개발되었으므로 패턴과 일치하지 않는 입력을 고려하지 못합니다. 이러한 간과는 정규식 패턴과 거의 일치하는 제한 없는 입력이 성능 저하로 이어질 수 있게 합니다.

이 문제를 해결하려면 다음을 수행할 수 있습니다.

- 패턴을 개발할 때 특히 정규식이 제한되지 않은 입력을 처리하도록 설계된 경우 역추적이 정규식 엔진의 성능에 미치는 영향을 고려해야 합니다. 자세한 내용은 역추적 관리 섹션을 참조하세요.
- 잘못되고 거의 유효하지 않으며 유효한 입력을 사용하여 정규식을 철저히 테스트합니다. Rex를 사용하여 특정 정규식에 대한 입력을 임의로 생성할 수 있습니다. Rex 는 Microsoft Research의 정규식 탐색 도구입니다.

개체 인스턴스화를 적절하게 처리

의 중심에 . NET의 정규식 개체 모델은 정규식 엔진을 나타내는 클래스입니다 System.Text.RegularExpressions.Regex . 정규식 성능에 영향을 주는 가장 큰 요인은 엔진이 사용되는 방식 Regex 인 경우가 많습니다. 정규식을 정의하려면 정규식 엔진을 정규식 패턴과 긴밀하게 결합해야 합니다. 이러한 결합 프로세스는 생성자에 정규식 패턴을 전달하여 개체를 Regex 인스턴스화하거나 분석할 정규식 패턴과 문자열을 전달하여 정적 메서드를 호출하는 작업이 포함되든 관계없이 비용이 많이 듭니다.

> **참고**
>
> 해석되고 컴파일된 정규식을 사용하는 성능에 대한 자세한 내용은 블로그 게시물 **정규식 성능 최적화, 파트 II: 역추적 담당**을 참조하세요.

정규식 엔진을 특정 정규식 패턴과 결합한 다음 엔진을 사용하여 여러 가지 방법으로 텍스트를 일치시킬 수 있습니다.

- 와 같은 Regex.Match(String, String)정적 패턴 일치 메서드를 호출할 수 있습니다. 이 메서드는 정규식 개체의 인스턴스화가 필요하지 않습니다.
- Regex 객체를 인스턴스화하고 정규식 엔진을 정규식 패턴에 바인딩하는 기본 메서드인 해석된 정규식의 인스턴스 패턴 매칭 메서드를 호출할 수 있습니다. Regex 개체가 `options` 인수를 포함하지 않고 Compiled 플래그와 함께 인스턴스화될 때 발생합니다.
- 개체를 Regex 인스턴스화하고 소스에서 생성된 정규식의 인스턴스 패턴 일치 메서드를 호출할 수 있습니다. 대부분의 경우 이 기술을 사용하는 것이 좋습니다. 이렇게 하려면 GeneratedRegexAttribute 특성을 `Regex`을 반환하는 partial 메서드에 배치합니다.
- 개체를 Regex 인스턴스화하고 컴파일된 정규식의 인스턴스 패턴 일치 메서드를 호출할 수 있습니다. 정규식 객체는 Regex 플래그가 포함된 `options` 인수를 사용하여 Compiled 객체를 인스턴스화할 때 컴파일된 패턴을 나타냅니다.

정규식 일치 메서드를 호출하는 특정 방법은 애플리케이션의 성능에 영향을 줄 수 있습니다. 다음 섹션에서는 정적 메서드 호출, 소스 생성 정규식, 해석된 정규식 및 컴파일된 정규식을 사용하여 애플리케이션의 성능을 향상시키는 경우에 대해 설명합니다.

> ⓘ **중요**
>
> 메서드 호출 형식(정적, 해석됨, 소스 생성, 컴파일됨)은 메서드 호출에서 동일한 정규식을 반복적으로 사용하거나 애플리케이션에서 정규식 개체를 광범위하게 사용하는 경우 성능에 영향을 줍니다.

정적 정규식

정규식 개체를 동일한 정규식으로 반복적으로 인스턴스화하는 대신 정적 정규식 메서드를 사용하는 것이 좋습니다. 정규식 개체에서 사용되는 정규식 패턴과 달리 정적 메서드 호출에 사용되는 패턴의 작업 코드(opcodes) 또는 컴파일된 CIL(공용 중간 언어)은 정규식 엔진에 의해 내부적으로 캐시됩니다.

예를 들어 이벤트 처리기는 사용자 입력의 유효성을 검사하기 위해 다른 메서드를 자주 호출합니다. 이 예제는 다음 코드에 반영되며 Button , 컨트롤의 Click 이벤트를 사용하여 사용자가 통화 기호를 입력한 후 소수 자릿수를 하나 이상 입력했는지 여부를 확인하는 메서드 `IsValidCurrency`를 호출합니다.

C#

```
public void OKButton_Click(object sender, EventArgs e)
{
   if (! String.IsNullOrEmpty(sourceCurrency.Text))
      if (RegexLib.IsValidCurrency(sourceCurrency.Text))
         PerformConversion();
      else
         status.Text = "The source currency value is invalid.";
}
```

메서드의 `IsValidCurrency` 비효율적인 구현은 다음 예제에 나와 있습니다.

> ⓘ **참고**
>
> 각 메서드 호출은 동일한 패턴으로 **Regex** 개체를 재인스턴스화합니다. 즉, 메서드가 호출될 때마다 정규식 패턴을 다시 컴파일해야 합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class RegexLib
{
   public static bool IsValidCurrency(string currencyValue)
   {
      string pattern = @"\p{Sc}+\s*\d+";
      Regex currencyRegex = new Regex(pattern);
      return currencyRegex.IsMatch(currencyValue);
   }
}
```

위의 비효율적인 코드를 정적 Regex.IsMatch(String, String) 메서드 호출로 바꿔야 합니다. 이 방법을 사용하면 패턴 일치 메서드를 호출할 때마다 개체를 인스턴스화 Regex 할 필요가 없으며 정규식 엔진이 캐시에서 컴파일된 버전의 정규식을 검색할 수 있습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class RegexLib2
{
   public static bool IsValidCurrency(string currencyValue)
   {
      string pattern = @"\p{Sc}+\s*\d+";
      return Regex.IsMatch(currencyValue, pattern);
   }
}
```

기본적으로 가장 최근에 사용한 최근 15개의 정적 정규식 패턴이 캐시됩니다. 더 많은 수의 캐시된 정적 정규식이 필요한 애플리케이션의 경우 속성을 설정 Regex.CacheSize 하여 캐시 크기를 조정할 수 있습니다.

이 예제에서 사용되는 정규식 `\p{Sc}+\s*\d+` 은 입력 문자열에 통화 기호와 소수 자릿수가 하나 이상 있음을 확인합니다. 패턴은 다음 표와 같이 정의됩니다.

테이블 확장

패턴	설명
`\p{Sc}+`	유니코드 기호, 통화 범주에 있는 하나 이상의 문자와 일치합니다.
`\s*`	0개 이상의 공백 문자와 일치합니다.
`\d+`	하나 이상의 10진수와 일치합니다.

해석된 정규식과 소스 생성 및 컴파일된 정규식 비교

옵션의 사양 Compiled 을 통해 정규식 엔진에 바인딩되지 않은 정규식 패턴이 *해석*됩니다. 정규식 개체가 인스턴스화되면 정규식 엔진은 정규식을 작업 코드 집합으로 변환합니다. 인스턴스 메서드가 호출되면 작업 코드가 CIL로 변환되고 JIT 컴파일러에서 실행됩니다. 마찬가지로 정적 정규식 메서드가 호출되고 정규식을 캐시에서 찾을 수 없는 경우 정규식 엔진은 정규식을 작업 코드 집합으로 변환하고 캐시에 저장합니다. 그런 다음 JIT 컴파일러가 실행할 수 있도록 이러한 작업 코드를 CIL로 변환합니다. 해석된 정규식은 실행 시간이 느려지는 대가로 시작 시간을 줄입니다. 이 프로세스 때문에 정규식을 적은 수의 메서드 호출에서 사용하거나 정규식 메서드에 대한 정확한 호출 수를 알 수 없지만 작을 것으로 예상되는 경우에 가장 적합합니다. 메서드 호출 수가 증가함에 따라 시작 시간 단축으로 인한 성능 이점이 실행 속도가 느려지면서 상쇄됩니다.

옵션의 사양 Compiled 을 통해 정규식 엔진에 바인딩된 정규식 패턴이 *컴파일됩니다*. 따라서 정규식 개체가 인스턴스화되거나 정적 정규식 메서드가 호출되고 캐시에서 정규식을 찾을 수 없는 경우 정규식 엔진은 정규식을 중간 작업 코드 집합으로 변환합니다. 그런 다음 이러한 코드는 CIL로 변환됩니다. 메서드가 호출되면 JIT 컴파일러는 CIL을 실행합니다. 해석된 정규식과 달리 컴파일된 정규식은 시작 시간을 늘리지만 개별 패턴 일치 메서드를 더 빠르게 실행합니다. 따라서 정규식 컴파일로 인한 성능 이점은 호출된 정규식 메서드 수에 비례하여 증가합니다.

`Regex` 반환 메서드에 GeneratedRegexAttribute 특성을 부여하여 정규식 엔진에 바인딩된 정규 표현식 패턴은 *소스 생성됩니다*. 컴파일러에 연결되는 소스 생성기는 CIL에서 `Regex`가 내보내는 것과 유사한 논리를 사용하여 C# 코드로 사용자 지정 `RegexOptions.Compiled` 파생 구현을 내보낸다. `RegexOptions.Compiled`의 모든 처리량 성능 이점(실제로는 그 이상)과 `Regex.CompileToAssembly`의 시작 이점을 얻으면서도 `CompileToAssembly`의 복잡성은 없습니다. 내보내는 원본은 프로젝트의 일부이므로 쉽게 보고 디버그할 수도 있습니다.

요약하면 다음을 수행하는 것이 좋습니다.

- 정규식 메서드를 특정 정규식과 함께 비교적 드물게 호출할 때 *해석된* 정규식을 사용합니다.
- C#에서 컴파일 시간에 알려진 인수와 함께 사용하고 특정 정규식을 비교적 자주 사용하는 경우 `Regex` 정규식을 사용합니다.
- 특정 정규식을 비교적 자주 사용하여 정규식 메서드를 호출하고 .NET 6 또는 이전 버전을 사용하는 경우 *컴파일된* 정규식을 사용합니다.

해석된 정규식의 느린 실행 속도가 시작 시간 단축으로 인한 이익보다 더 큰 정확한 임계값을 결정하기는 어렵습니다. 또한 소스 생성 또는 컴파일된 정규식의 느린 시작 시간이 더 빠른 실행 속도의 이득보다 더 큰 임계값을 결정하기도 어렵습니다. 임계값은 정규식의 복잡성 및 처리되는 특정 데이터를 포함하여 다양한 요인에 따라 달라집니다. 특정 애플리케이션 시나리오에 가장 적합한 성능을 제공하는 정규식을 결정하려면 클래스를 Stopwatch 사용하여 실행 시간을 비교할 수 있습니다.

다음 예제에서는 처음 10개의 문장을 읽을 때와 William D. Guthrie의 *Magna Carta 및 기타 주소* 텍스트에서 모든 문장을 읽을 때 컴파일, 소스 생성 및 해석된 정규식의 성능을 비교합니다. 예제의 출력에서 알 수 있듯이 정규식 일치 메서드에 대해 10개만 호출하는 경우 해석되거나 소스에서 생성된 정규식은 컴파일된 정규식보다 더 나은 성능을 제공합니다. 그러나 컴파일된 정규식은 많은 수의 호출(이 경우 13,000개 이상)이 수행될 때 더 나은 성능을 제공합니다.

C#

```
const string Pattern = @"\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]";

static readonly HttpClient s_client = new();

[GeneratedRegex(Pattern, RegexOptions.Singleline)]
private static partial Regex GeneratedRegex();

public async static Task RunIt()
{
    Stopwatch sw;
    Match match;
    int ctr;

    string text =
            await
s_client.GetStringAsync("https://www.gutenberg.org/cache/epub/64197/pg64197.txt");

    // Read first ten sentences with interpreted regex.
    Console.WriteLine("10 Sentences with Interpreted Regex:");
    sw = Stopwatch.StartNew();
    Regex int10 = new(Pattern, RegexOptions.Singleline);
    match = int10.Match(text);
    for (ctr = 0; ctr <= 9; ctr++)
    {
        if (match.Success)
            // Do nothing with the match except get the next match.
            match = match.NextMatch();
        else
            break;
    }
    sw.Stop();
    Console.WriteLine($"   {ctr} matches in {sw.Elapsed}");

    // Read first ten sentences with compiled regex.
    Console.WriteLine("10 Sentences with Compiled Regex:");
    sw = Stopwatch.StartNew();
    Regex comp10 = new Regex(Pattern,
                RegexOptions.Singleline | RegexOptions.Compiled);
    match = comp10.Match(text);
    for (ctr = 0; ctr <= 9; ctr++)
    {
        if (match.Success)
            // Do nothing with the match except get the next match.
            match = match.NextMatch();
        else
```

```
        break;
}
sw.Stop();
Console.WriteLine($"   {ctr} matches in {sw.Elapsed}");

// Read first ten sentences with source-generated regex.
Console.WriteLine("10 Sentences with Source-generated Regex:");
sw = Stopwatch.StartNew();

match = GeneratedRegex().Match(text);
for (ctr = 0; ctr <= 9; ctr++)
{
    if (match.Success)
        // Do nothing with the match except get the next match.
        match = match.NextMatch();
    else
        break;
}
sw.Stop();
Console.WriteLine($"   {ctr} matches in {sw.Elapsed}");

// Read all sentences with interpreted regex.
Console.WriteLine("All Sentences with Interpreted Regex:");
sw = Stopwatch.StartNew();
Regex intAll = new(Pattern, RegexOptions.Singleline);
match = intAll.Match(text);
int matches = 0;
while (match.Success)
{
    matches++;
    // Do nothing with the match except get the next match.
    match = match.NextMatch();
}
sw.Stop();
Console.WriteLine($"   {matches:N0} matches in {sw.Elapsed}");

// Read all sentences with compiled regex.
Console.WriteLine("All Sentences with Compiled Regex:");
sw = Stopwatch.StartNew();
Regex compAll = new(Pattern,
                RegexOptions.Singleline | RegexOptions.Compiled);
match = compAll.Match(text);
matches = 0;
while (match.Success)
{
    matches++;
    // Do nothing with the match except get the next match.
    match = match.NextMatch();
}
sw.Stop();
Console.WriteLine($"   {matches:N0} matches in {sw.Elapsed}");

// Read all sentences with source-generated regex.
Console.WriteLine("All Sentences with Source-generated Regex:");
sw = Stopwatch.StartNew();
```

```
    match = GeneratedRegex().Match(text);
    matches = 0;
    while (match.Success)
    {
        matches++;
        // Do nothing with the match except get the next match.
        match = match.NextMatch();
    }
    sw.Stop();
    Console.WriteLine($"   {matches:N0} matches in {sw.Elapsed}");

    return;
}
/* The example displays output similar to the following:

   10 Sentences with Interpreted Regex:
       10 matches in 00:00:00.0104920
   10 Sentences with Compiled Regex:
       10 matches in 00:00:00.0234604
   10 Sentences with Source-generated Regex:
       10 matches in 00:00:00.0060982
   All Sentences with Interpreted Regex:
       3,427 matches in 00:00:00.1745455
   All Sentences with Compiled Regex:
       3,427 matches in 00:00:00.0575488
   All Sentences with Source-generated Regex:
       3,427 matches in 00:00:00.2698670
*/
```

예제 `\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]`에서 사용되는 정규식 패턴은 다음 표와 같이 정의됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서부터 일치를 시작합니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`(\r?\n)|,?\s)`	줄바꿈 문자 앞에 0개 또는 1개의 캐리지 리턴을, 공백 문자 앞에 0개 또는 1개의 쉼표를 일치시킵니다.
`(\w+((\r?\n)|,?\s))*`	하나 이상의 단어 문자가 0개 이상 연속하여 일치하며, 이는 0개 또는 1개의 캐리지 리턴과 줄 바꿈 문자 뒤에 오거나, 0개 또는 1개의 쉼표 뒤에 공백 문자가 오는 경우입니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`[.?:;!]`	마침표, 물음표, 콜론, 세미콜론 또는 느낌표와 일치합니다.

역추적을 주도하십시오

일반적으로 정규식 엔진은 선형 진행을 사용하여 입력 문자열을 이동하고 정규식 패턴과 비교합니다. 그러나 정규식 패턴에 사용되는 미정량자(예: `*`, `+`, `?`)가 사용될 때, 정규식 엔진은 일부 성공적인 부분 일치를 포기하고 전체 패턴의 성공적인 일치를 찾기 위해 이전에 저장된 상태로 돌아갈 수 있습니다. 이 프로세스를 역추적이라고 합니다.

> **팁**
>
> 역추적에 대한 자세한 내용은 정규식 동작 및 **역추적에 대한 세부 정보를** 참조하세요. 역추적에 대한 자세한 내용은 **.NET 7의 정규식 개선 사항** 및 **정규식 성능 최적화** 블로그 게시물을 참조하세요.

역추적을 지원하면 정규식의 성능과 유연성이 지원됩니다. 또한 정규식 개발자의 손에 정규식 엔진의 작업을 제어할 책임이 있습니다. 개발자는 종종 이러한 책임을 인식하지 못하기 때문에 역추적을 오용하거나 과도한 역추적에 의존하는 것이 정규식 성능 저하에서 가장 중요한 역할을 하는 경우가 많습니다. 최악의 시나리오에서는 입력 문자열의 각 추가 문자에 대해 실행 시간이 두 배가 될 수 있습니다. 실제로 역추적을 과도하게 사용하면 입력이 정규식 패턴과 거의 일치하는 경우 무한 루프에 해당하는 프로그래밍 방식으로 쉽게 만들 수 있습니다. 정규식 엔진은 비교적 짧은 입력 문자열을 처리하는 데 몇 시간 또는 며칠이 걸릴 수 있습니다.

애플리케이션은 역추적이 필수적이지 않더라도 이를 사용함으로 인해 성능 저하를 겪는 경우가 많습니다. 예를 들어 정규식 `\b\p{Lu}\w*\b` 은 다음 표와 같이 대문자로 시작하는 모든 단어와 일치합니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서부터 일치를 시작합니다.
`\p{Lu}`	대문자와 일치합니다.
`\w*`	단어 구성 문자(공백 없는 글자) 0개 이상을 찾습니다.
`\b`	단어 경계에서 경기를 종료합니다.

단어 경계는 단어 문자의 하위 집합과 동일하지 않으므로 단어 문자를 일치시킬 때 정규식 엔진이 단어 경계를 넘을 가능성은 없습니다. 따라서 이 정규식의 경우 역추적은 일치 항목의 전체적인 성공에 영향을 줄 수 없습니다. 정규식 엔진이 단어 문자의 성공적인 예비 일치에 대해 상태를 저장해야 하기 때문에 성능이 저하되는 것일 뿐입니다.

역추적이 필요하지 않다고 판단되면 다음과 같은 몇 가지 방법으로 역추적을 사용하지 않도록 설정할 수 있습니다.

- RegexOptions.NonBacktracking 옵션을 설정하면 (.NET 7에서 도입된 기능) 자세한 내용은 비백트래킹 모드참조하세요.
- 원자성 그룹이라고 하는 언어 요소를 사용합니다 `(?>subexpression)` . 다음 예제에서는 두 정규식을 사용하여 입력 문자열을 구문 분석합니다. 첫 번째 `\b\p{Lu}\w*\b` 는 역추적에 의존합니다. 두 번째 `\b\p{Lu}(?>\w*)\b` 는 역추적을 사용하지 않도록 설정합니다. 예제의 출력과 같이 둘 다 동일한 결과를 생성합니다.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class BackTrack2Example
{
    public static void Main()
    {
        string input = "This this word Sentence name Capital";
        string pattern = @"\b\p{Lu}\w*\b";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);

        Console.WriteLine();

        pattern = @"\b\p{Lu}(?>\w*)\b";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//       This
//       Sentence
//       Capital
//
//       This
//       Sentence
//       Capital
```

대부분의 경우 정규식 패턴을 입력 텍스트와 일치시키는 데 역추적이 필요합니다. 그러나 과도한 역추적은 성능을 심각하게 저하시키고 애플리케이션이 응답을 중지했다는 인상을 줄 수 있습니다. 특히 이 문제는 수량자가 중첩되고 외부 하위 식과 일치하는 텍스트가 내부 하위 식과 일치하는 텍스트의 하위 집합일 때 발생합니다.

> ⚠ **경고**

과도한 역추적을 방지하는 것 외에도 시간 제한 기능을 사용하여 과도한 역추적이 정규식 성능을 심각하게 저하하지 않도록 해야 합니다. 자세한 내용은 **제한 시간 값 사용 섹션을 참조하세요** .

예를 들어 정규식 패턴 `^[0-9A-Z]([-.\w]*[0-9A-Z])*\$$` 은 하나 이상의 영숫자 문자로 구성된 부품 번호와 일치하기 위한 것입니다. 모든 추가 문자는 영숫자 문자, 하이픈, 밑줄 또는 마침표로 구성될 수 있지만 마지막 문자는 영숫자여야 합니다. 달러 기호는 부품 번호를 종료합니다. 경우에 따라 수량자가 중첩되고 하위 `[0-9A-Z]` 식이 하위 `[-.\w]*` 식의 하위 집합이기 때문에 이 정규식 패턴의 성능이 저하될 수 있습니다.

이러한 경우 중첩된 수량자를 제거하고 외부 하위 식이 너비가 0인 lookahead 또는 lookbehind 어설션으로 바꿔 정규식 성능을 최적화할 수 있습니다. Lookahead 및 lookbehind 어설션은 앵커입니다. 입력 문자열에서 포인터를 이동하지 않고 앞이나 뒤로 이동하여 지정된 조건이 충족되는지 확인합니다. 예를 들어 부품 번호 정규식을 .로 `^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$` 다시 작성할 수 있습니다. 이 정규식 패턴은 다음 표와 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	입력 문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
`[0-9A-Z]`	영숫자 문자를 매치하세요. 부품 번호는 적어도 이 문자로 구성되어야 합니다.
`[-.\w]*`	단어 문자, 하이픈 또는 마침표가 0개 이상 나타나는 것을 찾습니다.
`\$`	달러 기호를 일치시킵니다.
`(?<=[0-9A-Z])`	달러 기호의 뒤쪽을 찾아 이전 문자가 영숫자인지 확인합니다.
`$`	입력 문자열의 끝에서 일치 항목을 종료합니다.

다음 예제에서는 가능한 부품 번호가 포함된 배열과 일치하도록 이 정규식을 사용하는 방법을 보여 줍니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class BackTrack4Example
{
    public static void Main()
    {
        string pattern = @"^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$";
        string[] partNos = { "A1C$", "A4", "A4$", "A1603D$", "A1603D#" };
```

```
        foreach (var input in partNos)
        {
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine(match.Value);
            else
                Console.WriteLine("Match not found.");
        }
    }
}
// The example displays the following output:
//       A1C$
//       Match not found.
//       A4$
//       A1603D$
//       Match not found.
```

.NET의 정규식 언어에는 중첩된 수량자를 제거하는 데 사용할 수 있는 다음 언어 요소가 포함되어 있습니다. 자세한 내용은 그룹화 구문을 참조하세요.

테이블 확장

Language 요소	설명
`(?=` `subexpression` `)`	너비가 0인 긍정 전방 탐색. 현재 위치보다 앞을 내다보며 입력 문자열과 일치하는지 여부를 `subexpression` 확인합니다.
`(?!` `subexpression` `)`	너비가 0인 부정 전방 탐색입니다. 현재 위치 앞을 살펴보고 `subexpression`이(가) 입력 문자열과 일치하지 않는지 확인합니다.
`(?<=` `subexpression` `)`	너비가 0인 긍정적 후방 탐색입니다. 현재 위치 뒤를 확인하여 입력 문자열과 일치하는지 여부를 `subexpression` 확인합니다.
`(?<!` `subexpression` `)`	너비가 0인 역방향 부정 탐색입니다. 현재 위치 뒤를 확인하여 입력 문자열과 일치하지 않는지 여부를 `subexpression` 확인합니다.

제한 시간 값 사용

정규식이 정규식 패턴과 거의 일치하는 입력을 처리하는 경우 과도한 역추적에 의존하여 성능에 상당한 영향을 줄 수 있습니다. 거의 일치하는 입력에 대해 역추적 및 테스트 정규식을 사용하는 것을 신중하게 고려하는 것 외에도, 발생하는 경우 과도한 역추적의 영향을 최소화하기 위해 항상 시간 제한 값을 설정해야 합니다.

정규식 제한 시간 간격은 정규식 엔진이 시간 초과되기 전에 단일 일치 항목을 찾는 기간을 정의합니다. 정규식 패턴 및 입력 텍스트에 따라 실행 시간이 지정된 제한 시간 간격을 초과할 수 있지만 지정된 제한 시간 간격보다 역추적하는 데 더 많은 시간이 소요되지는 않습니다. 기본 시간

초과 간격은 Regex.InfiniteMatchTimeout로, 이는 정규식이 시간 초과되지 않음을 의미합니다. 이 기본값을 재정의하고 다음과 같이 시간 초과 간격을 설정할 수 있습니다.

- 개체를 Regex(String, RegexOptions, TimeSpan) 인스턴스화 Regex 할 때 시간 제한 값을 제공하려면 생성자를 호출합니다.
- 매개 변수를 포함하는 정적 패턴 일치 메서드(예: Regex.Match(String, String, RegexOptions, TimeSpan) 또는 Regex.Replace(String, String, String, RegexOptions, TimeSpan))를 호출합니다 `matchTimeout` .
- 와 같은 `AppDomain.CurrentDomain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT", TimeSpan.FromMilliseconds(100));` 코드를 사용하여 프로세스 전체 또는 앱 도메인 전체 값을 설정합니다.

제한 시간 간격을 정의했고 해당 간격의 끝에 일치하는 항목을 찾을 수 없는 경우 정규식 메서드는 예외를 RegexMatchTimeoutException throw합니다. 예외 처리기에서 더 긴 시간 제한 간격으로 일치를 다시 시도하거나, 일치 시도를 중단하거나, 일치 항목이 없다고 가정하거나, 일치 시도를 포기하고, 향후 분석을 위해 예외 정보를 기록하도록 선택할 수 있습니다.

다음 예제에서는 제한 시간 간격이 350밀리초인 정규식을 인스턴스화하여 텍스트 문서에서 단어의 단어 수와 평균 문자 수를 계산하는 메서드를 정의 `GetWordData` 합니다. 일치 작업이 타임아웃되면 시간 제한 간격이 350 밀리초 증가하고, Regex 개체가 재생성됩니다. 새 제한 시간 간격이 1초를 초과하는 경우, 메서드는 예외를 호출자에게 다시 던집니다.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.RegularExpressions;

public class TimeoutExample
{
    public static void Main()
    {
        RegexUtilities util = new RegexUtilities();
        string title = "Doyle - The Hound of the Baskervilles.txt";
        try
        {
            var info = util.GetWordData(title);
            Console.WriteLine($"Words:                {info.Item1:N0}");
            Console.WriteLine($"Average Word Length: {info.Item2:N2} characters");
        }
        catch (IOException e)
        {
            Console.WriteLine($"IOException reading file '{title}'");
            Console.WriteLine(e.Message);
        }
```

```
        catch (RegexMatchTimeoutException e)
        {
            Console.WriteLine($"The operation timed out after
{e.MatchTimeout.TotalMilliseconds:N0} milliseconds");
        }
    }
}

public class RegexUtilities
{
    public Tuple<int, double> GetWordData(string filename)
    {
        const int MAX_TIMEOUT = 1000;   // Maximum timeout interval in
milliseconds.
        const int INCREMENT = 350;      // Milliseconds increment of timeout.

        List<string> exclusions = new List<string>(new string[] { "a", "an", "the"
});
        int[] wordLengths = new int[29];        // Allocate an array of more than
ample size.
        string input = null;
        StreamReader sr = null;
        try
        {
            sr = new StreamReader(filename);
            input = sr.ReadToEnd();
        }
        catch (FileNotFoundException e)
        {
            string msg = String.Format("Unable to find the file '{0}'", filename);
            throw new IOException(msg, e);
        }
        catch (IOException e)
        {
            throw new IOException(e.Message, e);
        }
        finally
        {
            if (sr != null) sr.Close();
        }

        int timeoutInterval = INCREMENT;
        bool init = false;
        Regex rgx = null;
        Match m = null;
        int indexPos = 0;
        do
        {
            try
            {
                if (!init)
                {
                    rgx = new Regex(@"\b\w+\b", RegexOptions.None,
                                    TimeSpan.FromMilliseconds(timeoutInterval));
                    m = rgx.Match(input, indexPos);
```

```
                    init = true;
                }
                else
                {
                    m = m.NextMatch();
                }
                if (m.Success)
                {
                    if (!exclusions.Contains(m.Value.ToLower()))
                        wordLengths[m.Value.Length]++;

                    indexPos += m.Length + 1;
                }
            }
            catch (RegexMatchTimeoutException e)
            {
                if (e.MatchTimeout.TotalMilliseconds < MAX_TIMEOUT)
                {
                    timeoutInterval += INCREMENT;
                    init = false;
                }
                else
                {
                    // Rethrow the exception.
                    throw;
                }
            }
        } while (m.Success);

        // If regex completed successfully, calculate number of words and average
length.
        int nWords = 0;
        long totalLength = 0;

        for (int ctr = wordLengths.GetLowerBound(0); ctr <=
wordLengths.GetUpperBound(0); ctr++)
        {
            nWords += wordLengths[ctr];
            totalLength += ctr * wordLengths[ctr];
        }
        return new Tuple<int, double>(nWords, totalLength / nWords);
    }
}
```

필요한 경우에만 캡처

.NET의 정규식은 정규식 패턴을 하나 이상의 하위 식으로 그룹화할 수 있는 그룹화 구문을 지원합니다. .NET 정규식 언어에서 가장 일반적으로 사용되는 그룹화 구문은 `(` 번호가 매겨진 캡처링 그룹을 정의하는 *하위*`)` 식과 `(?<` 명명된 캡처링 그룹을 정의하는 *이름*`>`*하위*`)` 식입니다. 그룹화 구문은 역참조를 만들고 수량자가 적용되는 하위 식 정의에 필수적입니다.

그러나 이러한 언어 요소를 사용하는 데는 비용이 듭니다. 속성 GroupCollection이(가) 반환한 Match.Groups 개체는 가장 최근의 명명되지 않은 캡처 또는 명명된 캡처로 채워집니다. 단일 그룹화 구문이 입력 문자열에서 여러 부분 문자열을 캡처한 경우, 특정 캡처링 그룹의 CaptureCollection 속성에서 반환된 Group.Captures 객체에 여러 Capture 객체가 채워집니다.

종종 그룹화 구문은 정규식에서만 사용되므로 수량자를 적용할 수 있습니다. 이러한 하위 식에서 캡처한 그룹은 나중에 사용되지 않습니다. 예를 들어 정규식 `\b(\w+[;,]?\s?)+[.?!]` 은 전체 문장을 캡처하도록 설계되었습니다. 다음 표에서는 이 정규식 패턴의 언어 요소와 개체 Match 및 Match.Groups 컬렉션에 Group.Captures 미치는 영향에 대해 설명합니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서부터 일치를 시작합니다.
`\w+`	하나 이상의 단어 문자를 찾습니다.
`[;,]?`	0개 또는 1개의 쉼표 또는 세미콜론과 일치합니다.
`\s?`	0개 또는 1개의 공백 문자를 찾습니다.
`(\w+[;,]?\s?)+`	하나 이상의 반복되는 단어 문자 뒤에 선택적으로 쉼표나 세미콜론이 올 수 있으며, 그 뒤에 선택적으로 공백 문자가 올 수 있습니다. 이 패턴은 정규식 엔진이 문장 끝에 도달할 때까지 여러 단어 문자(즉, 단어)와 선택적 문장 부호 기호의 조합이 반복되도록 필요한 첫 번째 캡처 그룹을 정의합니다.
`[.?!]`	마침표, 물음표 또는 느낌표와 일치합니다.

다음 예제와 같이 일치 항목이 발견되면 GroupCollection 개체와 CaptureCollection 개체가 모두 일치 항목의 캡처로 채워집니다. 이 경우 캡처링 그룹이 `(\w+[;,]?\s?)` 존재하므로 수량자를 적용할 수 있으므로 `+` 정규식 패턴이 문장의 각 단어와 일치할 수 있습니다. 그렇지 않으면 문장의 마지막 단어와 일치합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Group1Example
{
    public static void Main()
    {
        string input = "This is one sentence. This is another.";
        string pattern = @"\b(\w+[;,]?\s?)+[.?!]";

        foreach (Match match in Regex.Matches(input, pattern))
        {
```

```
            Console.WriteLine($"Match: '{match.Value}' at index {match.Index}.");
            int grpCtr = 0;
            foreach (Group grp in match.Groups)
            {
                Console.WriteLine($"   Group {grpCtr}: '{grp.Value}' at index
{grp.Index}.");
                int capCtr = 0;
                foreach (Capture cap in grp.Captures)
                {
                    Console.WriteLine($"      Capture {capCtr}: '{cap.Value}' at
{cap.Index}.");
                    capCtr++;
                }
                grpCtr++;
            }
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//             Capture 0: 'This is one sentence.' at 0.
//          Group 1: 'sentence' at index 12.
//             Capture 0: 'This ' at 0.
//             Capture 1: 'is ' at 5.
//             Capture 2: 'one ' at 8.
//             Capture 3: 'sentence' at 12.
//
//       Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//             Capture 0: 'This is another.' at 22.
//          Group 1: 'another' at index 30.
//             Capture 0: 'This ' at 22.
//             Capture 1: 'is ' at 27.
//             Capture 2: 'another' at 30.
```

하위 식만 사용하여 수량자를 적용하고 캡처된 텍스트에 관심이 없는 경우 그룹 캡처를 사용하지 않도록 설정해야 합니다. 예를 들어 `(?:subexpression)` 언어 요소는 적용되는 그룹이 일치하는 부분 문자열을 캡처할 수 없도록 합니다. 다음 예제에서는 이전 예제의 정규식 패턴이 .로 `\b(?:\w+[;,]?\s?)+[.?!]`변경됩니다. 출력에서 보다시피, 정규식 엔진이 GroupCollection 및 CaptureCollection 컬렉션을 채우지 못하도록 방지합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Group2Example
{
    public static void Main()
    {
```

```
        string input = "This is one sentence. This is another.";
        string pattern = @"\b(?:\w+[;,]?\s?)+[.?!]";

        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine($"Match: '{match.Value}' at index {match.Index}.");
            int grpCtr = 0;
            foreach (Group grp in match.Groups)
            {
                Console.WriteLine($"   Group {grpCtr}: '{grp.Value}' at index
{grp.Index}.");
                int capCtr = 0;
                foreach (Capture cap in grp.Captures)
                {
                    Console.WriteLine($"      Capture {capCtr}: '{cap.Value}' at
{cap.Index}.");
                    capCtr++;
                }
                grpCtr++;
            }
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//             Capture 0: 'This is one sentence.' at 0.
//
//       Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//             Capture 0: 'This is another.' at 22.
```

다음 방법 중 하나로 캡처를 사용하지 않도록 설정할 수 있습니다.

- `(?:subexpression)` 언어 요소를 사용합니다. 이 요소는 해당 요소가 적용되는 그룹에서 일치하는 부분 문자열의 캡처를 방지합니다. 중첩된 그룹에서 부분 문자열 캡처를 사용하지 않도록 설정하지 않습니다.

- ExplicitCapture 옵션을 사용합니다. 정규식 패턴에서 명명되지 않은 또는 암시적 캡처를 모두 사용하지 않도록 설정합니다. 이 옵션을 사용하는 경우 언어 요소로 `(?<name>subexpression)` 정의된 명명된 그룹과 일치하는 부분 문자열만 캡처할 수 있습니다. ExplicitCapture 플래그는 `options` 클래스 생성자의 Regex 매개 변수나 `options` 정적 일치 메서드의 Regex 매개 변수에 전달할 수 있습니다.

- 언어 요소에서 `n (?imnsx)` 옵션을 사용합니다. 이 옵션은 요소가 나타나는 정규식 패턴의 지점에서 명명되지 않은 캡처 또는 암시적 캡처를 모두 사용하지 않도록 설정합니다. 캡처는 패턴의 끝까지 비활성화되거나 `(-n)` 옵션이 호출되지 않은 캡처 또는 암시적 캡처를 허용할 때까지 비활성화됩니다. 자세한 내용은 기타 구문을 참조하세요.

- 언어 요소에서 `n (?imnsx:subexpression)` 옵션을 사용합니다. 이 옵션은 `subexpression`에서 모든 명명되지 않은 또는 암시적 캡처를 비활성화합니다. 명명되지 않은 또는 암시적 중첩 캡처 그룹의 캡처도 사용하지 않도록 설정됩니다.

스레드 안전성

Regex 클래스 자체는 스레드로부터 안전하고 변경할 수 없습니다(읽기 전용). 즉, `Regex` 모든 스레드에서 개체를 만들고 스레드 간에 공유할 수 있습니다. 일치하는 메서드는 스레드에서 호출하고 전역 상태를 변경하지 않을 수 있습니다.

그러나 반환 `Match` 된 결과 개체(`MatchCollection` 및 `Regex`)는 단일 스레드에서 사용해야 합니다. 이러한 개체의 대부분은 논리적으로 변경할 수 없지만, 구현은 성능을 향상시키기 위해 일부 결과의 계산을 지연시킬 수 있으므로 호출자는 해당 개체에 대한 액세스를 직렬화해야 합니다.

여러 스레드에서 결과 개체를 공유 `Regex` 해야 하는 경우 동기화된 메서드를 호출하여 이러한 개체를 스레드로부터 안전한 인스턴스로 변환할 수 있습니다. 열거자를 제외하고 모든 정규식 클래스는 스레드로부터 안전하거나 동기화된 메서드를 통해 스레드로부터 안전한 개체로 변환할 수 있습니다.

열거자만 예외입니다. 컬렉션 열거자에 대한 호출을 직렬화해야 합니다. 규칙은 컬렉션을 둘 이상의 스레드에서 동시에 열거할 수 있는 경우 열거자가 트래버스하는 컬렉션의 루트 개체에 열거자 메서드를 동기화해야 한다는 것입니다.

관련 문서

테이블 확장

제목	설명
정규식 동작에 대한 의 세부 정보	.NET에서 정규식 엔진의 구현을 검사합니다. 이 문서에서는 정규식의 유연성에 중점을 두고 정규식 엔진의 효율적이고 강력한 작동을 보장하는 개발자의 책임에 대해 설명합니다.
백트래킹	역추적의 정의와 역추적이 정규식 성능에 미치는 영향을 설명하고 역추적에 대한 대안을 제공하는 언어 요소를 검사합니다.
정규식 언어 - 빠른 참조	.NET에서 정규식 언어의 요소를 설명하고 각 언어 요소에 대한 자세한 설명서에 대한 링크를 제공합니다.

정규식 예: HREF 스캐닝

아티클 • 2024. 03. 11.

다음 예제는 입력 문자열을 검색하고 모든 href="..." 값과 문자열에서의 해당 위치를 보여 줍니다.

> ⚠ 경고
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions`에 대한 입력을 제공하여 서비스 거부 공격을 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

Regex 개체

`DumpHRefs` 메서드는 사용자 코드에서 여러 번 호출할 수 있으므로 `static`(Visual Basic의 경우 `Shared`) Regex.Match(String, String, RegexOptions) 메서드를 사용합니다. 그러면 정규식 엔진이 정규식을 캐시할 수 있으며 메서드를 호출할 때마다 새 Regex 개체를 인스턴스화하는 오버헤드를 방지합니다. 그리고 Match 개체는 문자열의 모든 일치 항목을 반복하는 데 사용됩니다.

C#

```
private static void DumpHRefs(string inputString)
{
    string hrefPattern = @"href\s*=\s*(?:[""']?(?<1>[^""'>]*)[""']?|(?<1>
[^>\s]+))";

    try
    {
        Match regexMatch = Regex.Match(inputString, hrefPattern,
                                       RegexOptions.IgnoreCase |
RegexOptions.Compiled,
                                       TimeSpan.FromSeconds(1));
        while (regexMatch.Success)
        {
            Console.WriteLine($"Found href {regexMatch.Groups[1]} at
{regexMatch.Groups[1].Index}");
            regexMatch = regexMatch.NextMatch();
        }
    }
    catch (RegexMatchTimeoutException)
    {
        Console.WriteLine("The matching operation timed out.");
```

```
    }
}
```

다음 예제에서는 `DumpHRefs` 메서드를 호출하는 방법을 보여 줍니다.

C#

```
public static void Main()
{
    string inputString = "My favorite web sites include:</P>" +
                         "<A HREF=\"https://learn.microsoft.com/en-
us/dotnet/\">" +
                         ".NET Documentation</A></P>" +
                         "<A HREF=\"http://www.microsoft.com\">" +
                         "Microsoft Corporation Home Page</A></P>" +
                         "<A
HREF=\"https://devblogs.microsoft.com/dotnet/\">" +
                         ".NET Blog</A></P>";
    DumpHRefs(inputString);
}
// The example displays the following output:
//       Found href https://learn.microsoft.com/dotnet/ at 43
//       Found href http://www.microsoft.com at 114
//       Found href https://devblogs.microsoft.com/dotnet/ at 188
```

정규식 패턴 `href\s*=\s*(?:["'](?<1>[^"']*)["']|(?<1>[^>\s]+))` 는 다음 테이블과 같이 해석됩니다.

테이블 확장

패턴	설명
`href`	리터럴 문자열 "href"과 일치합니다. 일치 항목 찾기에서는 대/소문자를 구분하지 않습니다.
`\s*`	0개 이상의 공백 문자가 일치하는지 확인합니다.
`=`	등호와 일치합니다.
`\s*`	0개 이상의 공백 문자가 일치하는지 확인합니다.
`(?:`	캡처하지 않는 그룹을 시작합니다.
`["'](?<1>[^"']*)["']`	따옴표 또는 아포스트로피를 일치시키고 그 뒤에 따옴표나 아포스트로피 이외의 문자와 일치하는 캡처링 그룹이 오고 그 뒤에 따옴표나 아포스트로피가 옵니다. `1` 이라는 그룹이 이 패턴에 포함됩니다.
|	이전 식 또는 다음 식과 일치하는 부울 OR입니다.

패턴	설명
`(?<1>[^>\s]+)`	보다 큼 기호나 공백 문자 이외의 모든 문자와 일치시키기 위해 부정 집합을 사용하는 캡처링 그룹입니다. `1`이라는 그룹이 이 패턴에 포함됩니다.
`)`	캡처하지 않는 그룹을 종료합니다.

일치 결과 클래스

검색 결과는 Match 클래스에 저장되는데, 이 클래스는 검색에서 추출한 모든 부분 문자열에 대한 액세스를 제공합니다. 또한 검색 중인 문자열 및 사용 중인 정규식을 기억하므로 Match.NextMatch 메서드를 호출하여 마지막 검색이 종료된 위치부터 또 다른 검색을 수행할 수 있습니다.

명시적으로 명명된 캡처

기존의 정규식에서 캡처링 괄호는 자동으로 순서대로 번호가 매겨집니다. 이 경우에 두 가지 문제가 발생합니다. 첫째, 괄호를 삽입하거나 제거하여 정규식을 수정하면 번호가 매겨진 캡처를 참조하는 모든 코드를 다시 작성하여 새로 지정된 번호를 반영해야 합니다. 둘째, 괄호의 다른 집합을 사용하여 허용 가능한 일치 항목에 두 개의 대체 식을 제공하기 때문에 둘 중 어느 쪽이 결과를 반환했는지 확인하기가 어려울 수 있습니다.

이러한 문제를 해결하기 위해 Regex 클래스는 `(?<name>…)` 구문이 지정된 슬롯에 일치 항목을 캡처하도록 지원합니다(문자열 또는 정수를 사용하여 슬롯에 이름을 지정할 수 있음. 정수를 더 신속하게 다시 호출할 수 있음). 따라서 동일한 문자열의 대체 일치 항목은 모두 동일한 위치로 지정될 수 있습니다. 충돌이 발생할 경우 슬롯에 삭제된 마지막 일치 항목이 성공적인 일치입니다. (그러나 단일 슬롯에 대한 여러 일치 항목이 있는 경우 전체 목록을 사용할 수 있습니다. 자세한 내용은 Group.Captures 컬렉션을 참조하세요.

참고 항목

- .NET 정규식

수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

제품 사용자 의견 제공

정규식 예제: 날짜 형식 변경

아티클 • 2025. 03. 31.

다음 코드 예제에서는 Regex.Replace 메서드를 사용하여 *mm/dd/yy* 형식의 날짜를 dd-*mm-yy*날짜로 바꿉다.

> **⚠ 경고**
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions` 입력을 제공하여 서비스 거부 공격 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

예시

C#

```
static string MDYToDMY(string input)
{
   try {
      return Regex.Replace(input,
             @"\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b",
            "${day}-${month}-${year}", RegexOptions.None,
            TimeSpan.FromMilliseconds(150));
   }
   catch (RegexMatchTimeoutException) {
      return input;
   }
}
```

다음 코드는 애플리케이션에서 `MDYToDMY` 메서드를 호출할 수 있는 방법을 보여줍니다.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Class1
{
   public static void Main()
   {
      string dateString = DateTime.Today.ToString("d",
                                          DateTimeFormatInfo.InvariantInfo);
      string resultString = MDYToDMY(dateString);
```

```
        Console.WriteLine($"Converted {dateString} to {resultString}.");
    }

    static string MDYToDMY(string input)
    {
        try {
            return Regex.Replace(input,
                   @"\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b",
                  "${day}-${month}-${year}", RegexOptions.None,
                  TimeSpan.FromMilliseconds(150));
        }
        catch (RegexMatchTimeoutException) {
            return input;
        }
    }
}
// The example displays the following output to the console if run on
8/21/2007:
//       Converted 08/21/2007 to 21-08-2007.
```

코멘트

`\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b` 정규식 패턴은 다음 표와 같이 해석됩니다.

테이블 확장

패턴	설명
`\b`	단어 경계에서 일치를 시작합니다.
`(?<month>\d{1,2})`	하나 또는 두 개의 소수 자릿수를 찾습니다. 캡처된 `month` 그룹입니다.
`/`	슬래시 기호를 맞추세요.
`(?<day>\d{1,2})`	하나 또는 두 개의 소수 자릿수를 찾습니다. 캡처된 `day` 그룹입니다.
`/`	슬래시 표시에 맞추세요.
`(?<year>\d{2,4})`	2자리에서 4자리까지의 소수 자릿수를 일치시킵니다. 캡처된 `year` 그룹입니다.
`\b`	단어 경계에서 경기를 종료합니다.

패턴 `${day}-${month}-${year}` 다음 표와 같이 대체 문자열을 정의합니다.

테이블 확장

패턴	설명
`$(day)`	`day` 캡처링 그룹에서 캡처한 문자열을 추가합니다.
`-`	하이픈을 추가합니다.
`$(month)`	`month` 캡처링 그룹에서 캡처한 문자열을 추가합니다.
`-`	하이픈을 추가합니다.
`$(year)`	`year` 캡처링 그룹에서 캡처한 문자열을 추가합니다.

참고하십시오

- .NET 정규식

방법: URL에서 프로토콜 및 포트 번호 주출

다음 예제에서는 URL에서 프로토콜 및 포트 번호를 추출합니다.

> ⚠ **Warning**
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions` 입력을 제공하여 **서비스 거부 공격** 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

예시

이 예제에서는 메서드를 Match.Result 사용하여 프로토콜을 반환한 다음 콜론과 포트 번호를 반환합니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string url = "http://www.contoso.com:8080/letters/readme.html";

      Regex r = new Regex(@"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",
                          RegexOptions.None, TimeSpan.FromMilliseconds(150));
      Match m = r.Match(url);
      if (m.Success)
         Console.WriteLine(m.Result("${proto}${port}"));
   }
}
// The example displays the following output:
//       http:8080
```

정규식 패턴 `^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/` 은 다음 표와 같이 해석할 수 있습니다.

테이블 확장

패턴	설명
`^`	문자열의 시작 부분에서 일치를 시작합니다.

패턴	설명
`(?<proto>\w+)`	하나 이상의 단어 문자를 일치시킵니다. 이 그룹의 `proto`이름을 지정합니다.
`://`	콜론 뒤에 슬래시 기호 두 개가 있는 패턴을 일치시키십시오.
`[^/]+?`	여러 번 발생할 수 있는 슬래시 표시가 아닌 문자를 가능한 최소로 찾아냅니다.
`(?<port>:\d+)?`	콜론 뒤에 하나 이상의 숫자 문자가 있는 경우 0번 또는 1번 발생을 매칭합니다. 이 그룹의 `port`이름을 지정합니다.
`/`	슬래시 표시와 일치합니다.

메서드는 Match.Result 정규식 패턴에서 `${proto}${port}` 캡처된 두 명명된 그룹의 값을 연결하는 대체 시퀀스를 확장합니다. Match.Groups 속성에 의해 반환된 컬렉션 개체에서 검색된 문자열을 명시적으로 연결하는 것에 대한 편리한 대안입니다.

이 예제에서는 Match.Result 메서드를 두 개의 대체 `${proto}` 및 `${port}`와 함께 사용하여 캡처된 그룹을 출력 문자열에 포함합니다. 다음 코드와 같이 대신 일치 GroupCollection 개체에서 캡처된 그룹을 검색할 수 있습니다.

C#

```
Console.WriteLine(m.Groups["proto"].Value + m.Groups["port"].Value);
```

참고하십시오

- .NET 정규식 사용

Last updated on 2026. 03. 31.

방법: 문자열에서 유효하지 않은 문자 제거

아티클 • 2023. 05. 10.

다음 예제에서는 정적 Regex.Replace 메서드를 사용하여 문자열에서 잘못된 문자를 제거합니다.

> ⚠ **경고**
>
> System.Text.RegularExpressions를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions`에 대한 입력을 제공하여 **서비스 거부 공격**을 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

예제

이 예제에 정의된 `CleanInput` 메서드가 사용하여 사용자 입력을 허용하는 텍스트 필드에 입력한 문제가 될 수 있는 문자를 제거할 수 있습니다. 이 경우에 `CleanInput`은 마침표(.), 기호 (@), 하이픈(-)을 제외한 모든 영숫자가 아닌 문자를 제거하고 나머지 문자열을 반환합니다. 그러나 입력 문자열에 포함되어야 하는 모든 문자를 제거하도록 정규식 패턴을 수정할 수 있습니다.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    static string CleanInput(string strIn)
    {
        // Replace invalid characters with empty strings.
        try {
           return Regex.Replace(strIn, @"[^\w\.@-]", "",
                                RegexOptions.None,
TimeSpan.FromSeconds(1.5));
        }
        // If we timeout when replacing invalid characters,
        // we should return Empty.
        catch (RegexMatchTimeoutException) {
           return String.Empty;
        }
```

```
    }
}
```

정규식 패턴 `[^\w\.@-]`은 단어 문자, 마침표, @ 기호 또는 하이픈이 아닌 모든 문자를 찾습니다. 단어 문자는 문자, 숫자 또는 밑줄과 같은 문장 부호입니다. 이 패턴과 일치하는 모든 문자는 바꾸기 패턴에 정의된 String.Empty 문자열로 바뀝니다. 사용자 입력에서 추가 문자를 허용하려면 해당 문자를 정규식 패턴의 문자 클래스에 추가합니다. 예를 들어 정규식 패턴 `[^\w\.@-\%]`도 입력 문자열에 백분율 기호 및 백슬래시를 허용합니다.

참조

- .NET 정규식

방법: 문자열이 올바른 전자 메일 형식인지 확인

아티클 • 2023. 05. 10.

이 문서의 예제에서는 정규식을 사용하여 문자열이 올바른 전자 메일 형식인지 확인합니다.

이 정규식은 실제로 메일로 사용될 수 있는 것에 비해 비교적 단순합니다. 정규식을 사용하여 메일의 유효성을 검사하는 것은 메일의 구조가 올바른지 확인하는 데 유용합니다. 하지만 메일이 실제로 존재하는지 확인하는 작업을 대신하지는 않습니다.

✔ 작은 정규식을 사용하여 메일의 유효한 구조를 확인합니다.

✔ 앱 사용자가 제공한 주소로 테스트 메일을 보냅니다.

✘ 메일의 유효성을 검사하는 유일한 방법으로 정규식을 사용하지 마세요.

메일의 구조가 올바른지 확인하는 '완벽한' 정규식을 만들려는 경우 해당 식은 아주 복잡해져서 디버그하거나 개선하기가 매우 어려워집니다. 정규식은 메일의 구조가 올바르게 구성된 경우에도 메일이 존재하는지 확인할 수 없습니다. 메일의 유효성을 검사하는 가장 좋은 방법은 주소로 테스트 메일을 보내는 것입니다.

> ⚠ **경고**
>
> **System.Text.RegularExpressions**를 사용하여 신뢰할 수 없는 입력을 처리하는 경우 시간 제한을 전달합니다. 악의적인 사용자가 `RegularExpressions`에 대한 입력을 제공하여 **서비스 거부 공격**을 일으킬 수 있습니다. `RegularExpressions`를 사용하는 ASP.NET Core Framework API는 시간 제한을 전달합니다.

예

이 예제에서는 문자열에 유효한 메일 주소가 포함되어 있으면 `IsValidEmail`을 반환하고, 그러지 않으면 `true`를 반환하지만 다른 작업을 수행하지 않는 `false` 메서드를 정의합니다.

전자 메일 주소가 올바른지 확인하기 위해 `IsValidEmail` 메서드는 Regex.Replace(String, String, MatchEvaluator) 정규식 패턴으로 `(@)(.+)$` 메서드를 호출하여 전자 메일 주소에서 도메인 이름을 분리합니다. 세 번째 매개 변수는 일치하는 텍스트를 처리하고 대체하는 메서드를 나타내는 MatchEvaluator 대리자입니다. 정규식 패턴은 다음과 같이 해석됩니다.

무늬	설명
`(@)`	@ 문자를 찾습니다. 이 부분은 첫 번째 캡처 그룹입니다.
`(.+)`	하나 이상의 문자를 찾습니다. 이 부분은 두 번째 캡처 그룹입니다.
`$`	문자열의 끝 부분에서 일치 항목 찾기를 끝냅니다.

도메인 이름이 @ 문자와 함께 `DomainMapper` 메서드에 전달됩니다. 이 메서드는 IdnMapping 클래스를 사용하여 US-ASCII 문자 범위 외부에 있는 유니코드 문자를 Punycode로 변환합니다. 이 메서드는 `invalid` 메서드가 도메인 이름에서 잘못된 문자를 발견하는 경우 `True` 플래그를 IdnMapping.GetAscii 로 설정합니다. 메서드는 앞에 @ 기호가 있는 Punycode 도메인 이름을 `IsValidEmail` 메서드에 반환합니다.

> **팁**
>
> 단순한 `(@)(.+)$` 정규식 패턴을 사용하여 도메인을 정규화한 후에 성공 또는 실패를 나타내는 값을 반환하는 것이 좋습니다. 하지만 이 문서의 예제에서는 정규식을 추가로 사용하여 메일의 유효성을 검사하는 방법을 설명합니다. 메일의 유효성을 검사하는 방법과 관계없이, 항상 주소로 테스트 메일을 전송하여 메일이 존재하는지 확인해야 합니다.

그러면 `IsValidEmail` 메서드는 Regex.IsMatch(String, String) 메서드를 호출하여 주소가 정규식 패턴을 따르는지 확인합니다.

`IsValidEmail` 메서드는 메일 형식이 메일 주소에 유효한지 여부를 확인할 뿐이며 메일이 존재하는지 여부를 확인하지는 않습니다. 또한 `IsValidEmail` 메서드는 최상위 도메인 이름이 조회 작업 시 요구되는 IANA 루트 영역 데이터베이스 에 나열된 유효한 도메인 이름인지 확인하지 않습니다.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

namespace RegexExamples
{
    class RegexUtilities
    {
        public static bool IsValidEmail(string email)
        {
            if (string.IsNullOrWhiteSpace(email))
                return false;
```

```
            try
            {
                // Normalize the domain
                email = Regex.Replace(email, @"(@)(.+)$", DomainMapper,
                                      RegexOptions.None,
TimeSpan.FromMilliseconds(200));

                // Examines the domain part of the email and normalizes it.
                string DomainMapper(Match match)
                {
                    // Use IdnMapping class to convert Unicode domain names.
                    var idn = new IdnMapping();

                    // Pull out and process domain name (throws
ArgumentException on invalid)
                    string domainName = idn.GetAscii(match.Groups[2].Value);

                    return match.Groups[1].Value + domainName;
                }
            }
            catch (RegexMatchTimeoutException e)
            {
                return false;
            }
            catch (ArgumentException e)
            {
                return false;
            }

            try
            {
                return Regex.IsMatch(email,
                    @"^[^@\s]+@[^@\s]+\.[^@\s]+$",
                    RegexOptions.IgnoreCase,
TimeSpan.FromMilliseconds(250));
            }
            catch (RegexMatchTimeoutException)
            {
                return false;
            }
        }
    }
}
```

이 예제에서 정규식 패턴 `^[^@\s]+@[^@\s]+\.[^@\s]+$`는 다음 테이블과 같이 해석됩니다. 정규식은 RegexOptions.IgnoreCase 플래그를 사용하여 컴파일됩니다.

무늬	설명
`^`	문자열의 시작 부분에서 일치 항목 찾기를 시작합니다.
`[^@\s]+`	@ 문자나 공백 이외의 다른 문자를 하나 이상 찾습니다.

무늬	설명
`@`	@ 문자를 찾습니다.
`[^@\s]+`	@ 문자나 공백 이외의 다른 문자를 하나 이상 찾습니다.
`\.`	마침표 문자 하나를 찾습니다.
`[^@\s]+`	@ 문자나 공백 이외의 다른 문자를 하나 이상 찾습니다.
`$`	문자열의 끝 부분에서 일치 항목 찾기를 끝냅니다.

ⓘ 중요

이 정규식은 유효한 메일 주소의 모든 측면을 포괄하기 위한 것이 아닙니다. 필요에 따라 확장하는 예제로 제공됩니다.

참조

- .NET 정규식
- 어느 정도까지 메일 주소의 유효성을 검사해야 하나요?

.NET의 직렬화에 대해

2025. 06. 17.

serialization은 개체의 상태를 유지하거나 전송할 수 있는 폼으로 변환하는 프로세스입니다. serialization의 보완은 스트림을 개체로 변환하는 역직렬화입니다. 이러한 프로세스를 함께 사용하면 데이터를 저장하고 전송할 수 있습니다.

.NET에는 다음과 같은 직렬화 기술이 있습니다.

- JSON serialization 은 JSON(JavaScript Object Notation)과 .NET 개체를 매핑합니다. JSON은 일반적으로 웹에서 데이터를 공유하는 데 사용되는 개방형 표준입니다. JSON serializer는 기본적으로 공용 속성을 직렬화하며 프라이빗 및 내부 멤버도 직렬화하도록 구성할 수 있습니다.

- XML 및 SOAP serialization 은 속성 및 필드만 `public` 직렬화하며 형식 충실도를 유지하지 않습니다. 이 기능은 데이터를 사용하는 애플리케이션을 제한하지 않고 데이터를 제공하거나 사용하려는 경우에 유용합니다. XML은 개방형 표준이므로 웹에서 데이터를 공유하는 데 적합합니다. SOAP도 마찬가지로 개방형 표준이므로 매력적인 선택입니다.

- 이진 직렬화는 *형식 충실도*를 유지합니다. 즉, 개체의 전체 상태가 기록되며, 역직렬화할 때 정확히 복원됩니다. 이러한 유형의 serialization은 애플리케이션의 여러 호출 간에 개체의 상태를 유지하는 데 유용합니다. 예를 들어 개체를 클립보드로 직렬화하여 서로 다른 애플리케이션 간에 공유할 수 있습니다. 개체를 스트림, 디스크, 메모리, 네트워크를 통해 직렬화할 수 있습니다. Remoting은 serialization을 사용하여 한 컴퓨터 또는 애플리케이션 도메인에서 다른 컴퓨터 또는 애플리케이션 도메인으로 "값별" 개체를 전달합니다.

  > **⚠ 경고**
  >
  > `BinaryFormatter`를 사용한 이진 직렬화는 위험할 수 있습니다. 자세한 내용은 BinaryFormatter 보안 가이드 **및 BinaryFormatter 마이그레이션 가이드를 참조**하세요.

참고 문헌

System.Text.Json
개체를 JSON 형식 문서 또는 스트림으로 serialize하는 데 사용할 수 있는 클래스를 포함합니다.

System.Runtime.Serialization
개체를 직렬화 및 역직렬화하는 데 사용할 수 있는 클래스를 포함합니다.

System.Xml.Serialization

개체를 XML 형식 문서 또는 스트림으로 serialize하는 데 사용할 수 있는 클래스를 포함합니다.

.NET의 JSON 직렬화 및 역직렬화 - 개요

System.Text.Json 네임스페이스는 JSON(JavaScript Object Notation)에서 직렬화 및 역직렬화(또는 마샬링 및 경계 해제)하는 기능을 제공합니다. *Serialization*은 개체의 상태, 즉 속성 값을 저장하거나 전송할 수 있는 형식으로 변환하는 프로세스입니다. 직렬화된 양식에는 개체의 연결된 메서드에 대한 정보가 포함되지 않습니다. *Deserialization*은 직렬화된 형식에서 개체를 다시 생성합니다.

`System.Text.Json` 라이브러리 디자인은 광범위한 기능 집합에 비해 높은 성능과 낮은 메모리 할당을 강조합니다. 기본 제공 UTF-8 지원은 UTF-8로 인코딩된 JSON 텍스트를 읽고 쓰는 프로세스를 최적화합니다. 이는 웹의 데이터와 디스크의 파일에 가장 널리 사용되는 인코딩입니다.

라이브러리는 메모리 내 DOM(문서 개체 모델)을 사용하기 위한 클래스도 제공합니다. 이 기능을 사용하면 JSON 파일 또는 문자열의 요소에 대한 임의 액세스가 가능합니다.

Visual Basic의 경우 사용할 수 있는 라이브러리 부분에는 몇 가지 제한 사항이 있습니다. 자세한 내용은 Visual Studio 지원을 참조하세요.

라이브러리를 가져오는 방법

라이브러리는 .NET Core 3.0 이상 버전에서 공유 프레임워크의 일부로 기본 제공됩니다. 원본 생성 기능은 .NET 6 이상 버전에서 공유 프레임워크의 일부로 기본 제공됩니다.

.NET Core 3.0 이전 프레임워크 버전의 경우 System.Text.Json NuGet 패키지를 설치합니다. 패키지는 다음을 지원합니다.

- .NET Standard 2.0 이상
- .NET Framework 4.6.2 이상
- .NET 8 이상

네임스페이스 및 API

- System.Text.Json 네임스페이스에는 모든 진입점과 기본 형식이 포함되어 있습니다.
- System.Text.Json.Serialization 네임스페이스에는 직렬화 및 역직렬화와 관련된 고급 시나리오 및 사용자 지정을 위한 특성과 API가 포함되어 있습니다.
- System.Net.Http.Json 네임스페이스에는 네트워크 JSON 페이로드를 직렬화하고 역직렬화하는확장 메서드가 포함되어 있습니다.

> **Important**
>
> `System.Text.Json`에서는 이전에 사용했을 수도 있는 다음 serialization API를 지원하지 않습니다.

- System.Runtime.Serialization 네임스페이스의 모든 특성.
- System.SerializableAttribute 특성 및 ISerializable 인터페이스. 이러한 형식은 **이진 및 XML 직렬화**에만 사용됩니다.

리플렉션과 소스 생성

기본적으로 `System.Text.Json`을 사용하여 런타임에 직렬화 및 역직렬화를 위해 개체의 속성에 액세스하는 데 필요한 메타데이터를 수집합니다. 또는 `System.Text.Json`은 C# 원본 생성 기능을 사용하여 성능을 개선하고, 프라이빗 메모리 사용량을 줄이며, 어셈블리 트리밍을 용이하게 하여 앱 크기를 줄일 수 있습니다.

자세한 내용은 리플렉션 및 원본 생성을 참조하세요.

보안 정보

JsonSerializer를 디자인할 때 고려된 보안 위협 및 완화 방법에 대한 내용은 System.Text.Json 위협 모델 을 참조하세요.

스레드 안전성

`System.Text.Json` 직렬 변환기는 스레드 안전을 염두에 두고 설계되었습니다. 실제로 일단 잠그기만 하면 JsonSerializerOptions 인스턴스를 여러 스레드에서 안전하게 공유할 수 있습니다. JsonDocument는 JSON 값에 대해 변경 불가능하며 .NET 8 이상 버전에서는 스레드로부터 안전한 DOM 표현을 제공합니다.

추가 자료

- 라이브러리를 사용하는 방법

Last updated on 2025. 12. 05.

.NET 개체를 JSON으로 쓰는 방법(직렬화)

이 문서에서는 System.Text.Json 네임스페이스를 사용하여 JSON(JavaScript Object Notation)으로 직렬화하는 방법을 보여 줍니다. `Newtonsoft.Json`에서 기존 코드를 이식하는 경우 System.Text.Json으로 마이그레이션 방법을 참조하세요.

> **팁**
>
> AI 지원을 사용하여 **JSON으로 직렬화할** 수 있습니다.

문자열 또는 파일에 JSON을 쓰려면 JsonSerializer.Serialize 메서드를 호출합니다.

직렬화 예제

다음은 JSON 파일을 문자열로 만드는 예제입니다.

C#

```
using System.Text.Json;

namespace SerializeBasic
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string jsonString = JsonSerializer.Serialize(weatherForecast);

            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

JSON 출력은 *기본적으로 축소* 됩니다(공백, 들여쓰기 및 줄 바꿈 문자 제거).

다음은 동기 코드를 사용하여 JSON 파일을 만드는 예제입니다.

C#

```
using System.Text.Json;

namespace SerializeToFile
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string fileName = "WeatherForecast.json";
            string jsonString = JsonSerializer.Serialize(weatherForecast);
            File.WriteAllText(fileName, jsonString);

            Console.WriteLine(File.ReadAllText(fileName));
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

다음은 비동기 코드를 사용하여 JSON 파일을 만드는 예지입니다.

C#

```
using System.Text.Json;

namespace SerializeToFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }
```

```
    public class Program
    {
        public static async Task Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string fileName = "WeatherForecast.json";
            await using FileStream createStream = File.Create(fileName);
            await JsonSerializer.SerializeAsync(createStream, weatherForecast);
            Console.WriteLine(File.ReadAllText(fileName));
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

앞의 예제에서는 직렬화되는 형식에 형식 유추를 사용합니다. `Serialize()`의 오버로드는 제네릭 형식 매개 변수를 사용합니다.

C#

```
using System.Text.Json;

namespace SerializeWithGenericParameter
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string jsonString = JsonSerializer.Serialize<WeatherForecast>
(weatherForecast);

            Console.WriteLine(jsonString);
```

```
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

AI를 사용하여 직렬화 코드를 생성할 수도 있습니다. 지침은 이 문서의 AI 사용 섹션을 참조하세요.

Serialization 작동 방식

- 기본적으로 모든 public 속성은 직렬화됩니다. 무시할 속성을 지정할 수 있습니다. 프라이빗 멤버을 포함할 수도 있습니다.
- 기본 인코더는 ASCII가 아닌 문자, ASCII 범위 내의 HTML 구분 문자 및 RFC 8259 JSON 사양 에 따라 이스케이프되어야 하는 문자를 이스케이프합니다.
- 기본적으로 JSON은 축소됩니다. JSON을 보기 좋게 출력할 수 있습니다.
- 기본적으로 JSON 이름의 대소문자는 .NET 이름과 동일하게 설정됩니다. JSON 이름 대/소문자를 사용자 지정할 수 있습니다.
- 기본적으로 순환 참조가 검색되고 예외가 발생합니다. 참조를 보존하고 순환 참조를 처리할 수 있습니다.
- 기본적으로 필드는 무시됩니다. 필드를 포함할 수 있습니다.

ASP.NET Core 앱에서 System.Text.Json을 간접적으로 사용하는 경우 몇 가지 기본 동작이 다릅니다. 자세한 내용은 JsonSerializerOptions 웹 기본값을 참조하세요.

지원되는 형식은 다음과 같습니다.

- 숫자 형식, 문자열, 부울 등 JavaScript 기본 형식에 매핑되는 .NET 기본 형식
- 사용자 정의 일반 CLR 객체 (Plain Old CLR Object, POCO)
- 1차원 및 계단식 배열(`T[][]`)
- 다음 네임스페이스의 컬렉션 및 사전.
  - System.Collections
  - System.Collections.Generic
  - System.Collections.Immutable
  - System.Collections.Concurrent
  - System.Collections.Specialized
  - System.Collections.ObjectModel

  자세한 내용은 지원되는 형식에서 System.Text.Json를 참조하세요.

추가 형식을 처리하거나 기본 변환기에서 지원하지 않는 기능을 제공하는 사용자 지정 변환기를 구현할 수 있습니다.

다음은 컬렉션 속성 및 사용자 정의 형식을 포함하는 클래스가 직렬화되는 방법을 보여 주는 예제입니다.

C#

```
using System.Text.Json;

namespace SerializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get; set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot",
                SummaryField = "Hot",
                DatesAvailable = new List<DateTimeOffset>()
                    { DateTime.Parse("2019-08-01"), DateTime.Parse("2019-08-02") },
                TemperatureRanges = new Dictionary<string, HighLowTemps>
                    {
                        ["Cold"] = new HighLowTemps { High = 20, Low = -10 },
                        ["Hot"] = new HighLowTemps { High = 60 , Low = 20 }
                    },
                SummaryWords = new[] { "Cool", "Windy", "Humid" }
            };

            var options = new JsonSerializerOptions { WriteIndented = true };
            string jsonString = JsonSerializer.Serialize(weatherForecast, options);

            Console.WriteLine(jsonString);
```

```
        }
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "TemperatureRanges": {
//    "Cold": {
//      "High": 20,
//      "Low": -10
//    },
//    "Hot": {
//    "High": 60,
//      "Low": 20
//    }
//  },
//  "SummaryWords": [
//    "Cool",
//    "Windy",
//    "Humid"
//  ]
//}
```

UTF-8로 직렬화

문자열 기반 방법을 사용하는 것보다 UTF-8 바이트 배열로 직렬화하는 것이 5~10% 더 빠릅니다. 그 이유는 바이트(UTF-8)를 문자열(UTF-16)로 변환할 필요가 없기 때문입니다.

UTF-8 바이트 배열로 직렬화하려면 JsonSerializer.SerializeToUtf8Bytes 메서드를 호출합니다.

C#

```
byte[] jsonUtf8Bytes = JsonSerializer.SerializeToUtf8Bytes(weatherForecast);
```

Serialize를 사용하는 Utf8JsonWriter 오버로드도 사용할 수 있습니다.

형식이 지정된 JSON으로 직렬화

JSON 출력을 보기 좋게 출력하려면 JsonSerializerOptions.WriteIndented을 `true`로 설정합니다.

C#

```
using System.Text.Json;

namespace SerializeWriteIndented
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            var options = new JsonSerializerOptions { WriteIndented = true };
            string jsonString = JsonSerializer.Serialize(weatherForecast, options);

            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot"
//}
```

.NET 9부터 IndentCharacter 및 IndentSize를 사용하여 들여쓰기 문자와 크기를 사용자 지정할 수도 있습니다.

> **팁**
>
> 동일한 옵션으로 `JsonSerializerOptions`를 반복적으로 사용하는 경우 사용할 때마다 새 `JsonSerializerOptions` 인스턴스를 만들지 마세요. 모든 호출에 대해 동일한 인스턴스를 다시 사용하세요. 자세한 내용은 **JsonSerializerOptions 인스턴스 다시 사용**을 참조하세요.

AI를 사용하여 JSON으로 직렬화

GitHub Copilot와 같은 AI 도구를 사용하여 JSON으로 직렬화하는 데 사용하는 `System.Text.Json` 코드를 생성할 수 있습니다. 개체 필드 및 serialization 요구 사항에 맞게 프롬프트를 사용자 지정할 수 있습니다.

다음은 serialization 코드를 생성하는 데 사용할 수 있는 프롬프트 예제입니다.

Copilot 프롬프트

```
I have a variable named weatherForecast of type WeatherForecast.
Serialize the variable using System.Text.Json and write the result directly to a
file named "output.json" with the JSON indented for pretty formatting.
Ensure the code includes all necessary using directives and compiles without errors.
```

코필로트의 제안을 적용하기 전에 검토합니다.

GitHub Copilot에 대한 자세한 내용은 GitHub의 FAQ를 참조하세요 .

참고하십시오

- Visual Studio의 GitHub Copilot
- Visual Studio Code의 GitHub Copilot

Last updated on 2025. 11. 20.

System.Text.Json으로 속성 이름 및 값을 사용자 지정하는 방법

아티클 • 2025. 05. 07.

기본적으로 JSON 출력에서는 대소문자를 포함하여 속성 이름과 사전 키가 변경되지 않습니다. 열거형 값은 숫자로 표시됩니다. 그리고 속성은 정의된 순서대로 직렬화됩니다. 그러나 다음을 통해 이러한 동작을 사용자 지정할 수 있습니다.

- 직렬화된 특정 속성 및 열거형 멤버 이름을 지정합니다.
- 속성 이름 및 사전 키에 대한 기본 제공 명명 정책(예: camelCase, snake_case 또는 kebab-case)를 사용합니다.
- 속성 이름 및 사전 키에 대한 사용자 지정 명명 정책을 사용합니다.
- 명명 정책을 사용하거나 사용하지 않고 열거형 값을 문자열로 직렬화합니다.
- 직렬화된 속성의 순서를 구성합니다.

> **참고**
>
> **웹 기본** 명명 정책은 카멜 케이스입니다.

> **팁**
>
> AI 지원을 사용하여 **GitHub Copilot를 사용하여 사용자 지정 serialization 속성이 있는 개체를 만들** 수 있습니다.

JSON 속성 이름 및 값을 특수하게 처리해야 하는 다른 시나리오의 경우 사용자 지정 변환기를 구현하면 됩니다.

개별 속성 이름 사용자 지정

개별 속성 이름을 설정하려면 [JsonPropertyName] 특성을 사용합니다.

다음은 직렬화 형식과 그 결과 JSON의 예입니다.

C#

```
public class WeatherForecastWithPropertyName
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonPropertyName("Wind")]
```

```
    public int WindSpeed { get; set; }
}
```

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "Wind": 35
}
```

이 특성을 통해 설정된 속성 이름은 다음과 같은 특징이 있습니다.

- 직렬화 및 역직렬화 양방향으로 적용됩니다.
- 속성 명명 정책보다 우선합니다.
- 매개 변수가 있는 생성자의 매개 변수 이름 일치에는 영향을 주지 않습니다.

기본 제공 명명 정책 사용

다음 표에서는 기본 제공 명명 정책과 속성 이름에 미치는 영향을 보여줍니다.

테이블 확장

명명 정책	설명	원래 자산 이름	변환된 속성 이름
CamelCase	첫 번째 단어는 소문자로 시작합니다. 연속 단어는 대문자로 시작합니다.	`TempCelsius`	`tempCelsius`
KebabCaseLower*	단어는 하이픈으로 구분됩니다. 모든 문자는 소문자입니다.	`TempCelsius`	`temp-celsius`
KebabCaseUpper*	단어는 하이픈으로 구분됩니다. 모든 문자는 대문자입니다.	`TempCelsius`	`TEMP-CELSIUS`
SnakeCaseLower*	단어는 밑줄로 구분됩니다. 모든 문자는 소문자입니다.	`TempCelsius`	`temp_celsius`
SnakeCaseUpper*	단어는 밑줄로 구분됩니다. 모든 문자는 대문자입니다.	`TempCelsius`	`TEMP_CELSIUS`

* .NET 8 이상 버전에서 사용할 수 있습니다.

다음 예에서는 JsonSerializerOptions.PropertyNamingPolicy을(를) JsonNamingPolicy.CamelCase(으)로 설정하여 모든 JSON 속성 이름에 카멜 표기법을 사용하는 방법을 보여줍니다.

C#

```
var serializeOptions = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

다음은 직렬화 클래스 및 JSON 출력의 예입니다.

C#

```
public class WeatherForecastWithPropertyName
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON

```
{
  "date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "summary": "Hot",
  "Wind": 35
}
```

명명 정책:

- 직렬화 및 역직렬화에 적용됩니다.
- `[JsonPropertyName]` 특성에 의해 재정의됩니다. 이러한 이유로 이 예제의 JSON 속성 이름 `Wind`는 카멜 표기법이 아닙니다.

> **참고**
>
> 기본 제공 명명 정책은 서로게이트 쌍인 문자를 지원하지 않습니다. 자세한 내용은 dotnet/런타임 이슈 90352 를 참조하세요.

사용자 지정 JSON 속성 명명 정책 사용

사용자 지정 JSON 속성 명명 정책을 사용하려면 다음 예제와 같이 JsonNamingPolicy에서 파생되는 클래스를 만들고 ConvertName 메서드를 재정의합니다.

C#

```
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class UpperCaseNamingPolicy : JsonNamingPolicy
    {
        public override string ConvertName(string name) =>
            name.ToUpper();
    }
}
```

그리고 다음과 같이 JsonSerializerOptions.PropertyNamingPolicy 속성을 명명 정책 클래스의 인스턴스로 설정합니다.

C#

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = new UpperCaseNamingPolicy(),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

다음은 직렬화 클래스 및 JSON 출력의 예입니다.

C#

```
public class WeatherForecastWithPropertyName
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON

```
{
  "DATE": "2019-08-01T00:00:00-07:00",
  "TEMPERATURECELSIUS": 25,
  "SUMMARY": "Hot",
```

```
  "Wind": 35
}
```

JSON 속성 명명 정책은 다음과 같습니다.

- 직렬화 및 역직렬화에 적용됩니다.
- `[JsonPropertyName]` 특성에 의해 재정의됩니다. 이러한 이유로 이 예제의 JSON 속성 이름 `Wind`는 대문자가 아닙니다.

사전 키에 명명 정책 사용

직렬화할 객체의 속성이 `Dictionary<string,TValue>` 형식인 경우, 카멜 표기법 등의 명명 규칙을 사용하여 `string` 키를 변환할 수 있습니다. 이렇게 하려면 JsonSerializerOptions.DictionaryKeyPolicy을(를) 원하는 명명 정책으로 설정합니다. 다음 예제에서는 `CamelCase` 명명 정책을 사용합니다.

C#

```
var options = new JsonSerializerOptions
{
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

키-값 쌍 `TemperatureRanges` 및 `"ColdMinTemp", 20`가 있는 `"HotMinTemp", 40`라는 사전이 포함된 개체를 직렬화하면 다음 예제와 같은 JSON 출력이 생성됩니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "TemperatureRanges": {
    "coldMinTemp": 20,
    "hotMinTemp": 40
  }
}
```

사전 키에 대한 명명 정책은 serialization에만 적용됩니다. 사전을 역직렬화하는 경우 기본이 아닌 명명 정책으로 JsonSerializerOptions.DictionaryKeyPolicy을(를) 설정하더라도 키가 JSON 파일과 일치합니다.

열거형을 문자열로 표현

기본적으로 열거형은 숫자로 직렬화됩니다. 열거형 이름을 문자열로 직렬화하려면 JsonStringEnumConverter 또는 JsonStringEnumConverter<TEnum> 변환기를 사용합니다. JsonStringEnumConverter<TEnum>만 네이티브 AOT 런타임에서 지원됩니다.

예를 들어 열거형을 포함하는 다음 클래스를 직렬화해야 한다고 가정해 봅시다.

C#

```
public class WeatherForecastWithEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Summary? Summary { get; set; }
}

public enum Summary
{
    Cold, Cool, Warm, Hot
}
```

Summary가 `Hot`이면 기본적으로 직렬화된 JSON은 다음과 같이 숫자 값 3을 갖습니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": 3
}
```

다음 샘플 코드는 숫자형 값 대신 열거형 이름을 직렬화하고, 이름을 카멜 케이스로 변환합니다.

C#

```
options = new JsonSerializerOptions
{
    WriteIndented = true,
    Converters =
    {
        new JsonStringEnumConverter(JsonNamingPolicy.CamelCase)
    }
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

그 결과로 얻는 JSON은 다음 예제와 유사합니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "hot"
}
```

기본 제공 JsonStringEnumConverter은 문자열 값을 역직렬화할 수도 있습니다. 지정된 명명 정책의 사용 여부에 관계없이 작동합니다. 다음 예제에서는 `CamelCase`를 사용한 역직렬화를 보여줍니다.

C#

```
options = new JsonSerializerOptions
{
    Converters =
    {
        new JsonStringEnumConverter(JsonNamingPolicy.CamelCase)
    }
};
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithEnum>(jsonString,
options)!;
```

JsonConverterAttribute (JSON 변환기 속성)

열거형에 JsonConverterAttribute 주석을 추가하여 사용할 변환기를 지정할 수도 있습니다. 다음 예제에서는 JsonStringEnumConverter<TEnum> 특성을 사용하여 JsonConverterAttribute(.NET 8 이상 버전에서 사용 가능)를 지정하는 방법을 보여줍니다. 예를 들어 열거형을 포함하는 다음 클래스를 직렬화해야 한다고 가정해 봅시다.

C#

```
public class WeatherForecastWithPrecipEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Precipitation? Precipitation { get; set; }
}

[JsonConverter(typeof(JsonStringEnumConverter<Precipitation>))]
public enum Precipitation
{
    Drizzle, Rain, Sleet, Hail, Snow
}
```

다음 샘플 코드는 숫자 값 대신 열거형 이름을 직렬화합니다.

C#

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

결과 JSON은 다음과 같습니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Precipitation": "Sleet"
}
```

열거형 멤버 이름 사용자 지정

.NET 9부터 문자열로 직렬화된 형식에 대해 개별 열거형 멤버의 이름을 사용자 지정할 수 있습니다. 열거형 멤버 이름을 사용자 지정하려면 JsonStringEnumMemberName 특성으로 주석을 추가합니다.

예를 들어 사용자 지정 멤버 이름을 가진 열거형이 있는 다음 클래스를 serialize해야 한다고 가정합니다.

C#

```
public class WeatherForecastWithEnumCustomName
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public CloudCover? Sky { get; set; }
}

[JsonConverter(typeof(JsonStringEnumConverter))]
public enum CloudCover
{
    Clear,
    [JsonStringEnumMemberName("Partly cloudy")]
    Partial,
    Overcast
}
```

다음 샘플 코드는 숫자 값 대신 열거형 이름을 직렬화합니다.

C#

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

결과 JSON은 다음과 같습니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Sky": "Partly cloudy"
}
```

원본 생성

원본 생성과 함께 변환기를 사용하려면 열거형 필드를 문자열로 직렬화를 참조하세요.

직렬화된 속성의 순서 구성

기본적으로 속성은 클래스에서 정의된 순서대로 직렬화됩니다. [JsonPropertyOrder] 특성을 사용하면 직렬화에서 JSON 출력의 속성 순서를 지정할 수 있습니다. `Order` 속성의 기본값은 0입니다. 속성을 기본값이 설정된 속성 뒤에 배치하려면 `Order`를 양수로 설정합니다. 음수 `Order`는 속성을 기본값을 가진 속성 앞에 배치합니다. 속성은 가장 낮은 `Order` 값에서 가장 높은 값까지 순서대로 작성됩니다. 예를 들어 다음과 같습니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace PropertyOrder
{
    public class WeatherForecast
    {
        [JsonPropertyOrder(-5)]
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        [JsonPropertyOrder(-2)]
        public int TemperatureF { get; set; }
        [JsonPropertyOrder(5)]
        public string? Summary { get; set; }
```

```
        [JsonPropertyOrder(2)]
        public int WindSpeed { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureC = 25,
                TemperatureF = 25,
                Summary = "Hot",
                WindSpeed = 10
            };

            var options = new JsonSerializerOptions { WriteIndented = true };
            string jsonString = JsonSerializer.Serialize(weatherForecast,
options);
            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00",
//  "TemperatureF": 25,
//  "TemperatureC": 25,
//  "WindSpeed": 10,
//  "Summary": "Hot"
//}
```

GitHub Copilot를 사용하여 속성 이름 직렬화 방법을 맞춤설정하십시오.

코드가 serialize되는 방식에 변경 패턴을 적용하라는 메시지를 GitHub Copilot에 표시할 수 있습니다.

클래스 선언에 다음과 같은 `PascalCasing`속성이 있고 프로젝트에 대한 JSON 표준이 `snake_casing`있다고 가정합니다. AI를 사용하여 클래스의 모든 속성에 필요한 [JsonPropertyName] 특성을 추가할 수 있습니다. Copilot를 사용하여 다음과 같이 채팅 프롬프트를 사용하여 이러한 변경을 수행할 수 있습니다.

Copilot 프롬프트

```
Update #ClassName:
when the property name contains more than one word,
```

```
change the serialized property name to use underscores between words.
Use built-in serialization attributes.
```

다음은 간단한 클래스를 포함하는 예제의 보다 완전한 버전입니다.

Copilot 프롬프트

```
Take this C# class:
public class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF { get; set; }
    public string? Summary { get; set; }
    public int WindSpeed { get; set; }
}
When the property name contains more than one word,
change the serialized property name to use underscores between words.
Use built-in serialization attributes.
```

GitHub Copilot는 AI를 통해 구동되므로 예상치 못한 실수가 발생할 수 있습니다. 자세한 내용은 부조종사 FAQ 참조하세요.

Visual Studio에서 GitHub Copilot 및 VS Code에서 GitHub Copilot에 대해 자세히 알아보세요.

참고

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

를 사용하여 속성을 무시하는 방법 System.Text.Json

C# 개체를 JSON(JavaScript Object Notation)으로 serialize하는 경우 기본적으로 모든 공용 속성이 serialize됩니다. 그 중 일부를 결과 JSON에 표시하지 않으려면 몇 가지 옵션이 있습니다. 이 문서에서는 다양한 조건에 따라 속성을 무시하는 방법을 알아봅니다.

- 개별 속성
- 모든 읽기 전용 속성
- 모든 null-value 속성
- 모든 기본값 속성

개별 속성 무시

개별 속성을 무시하려면 [JsonIgnore] 특성을 사용합니다.

다음 예제에서는 serialize할 형식을 보여줍니다. 또한 JSON 출력도 표시합니다.

C#

```
public class WeatherForecastWithIgnoreAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    [JsonIgnore]
    public string? Summary { get; set; }
}
```

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
}
```

[JsonIgnore] 특성의 `Condition` 속성을 설정하여 조건부 제외를 지정할 수 있습니다. 열거형은 JsonIgnoreCondition 다음 옵션을 제공합니다.

- `Always` - 속성은 항상 무시됩니다. 지정되지 `Condition` 않은 경우 이 옵션을 가정합니다.
- `Never` - 속성은 `DefaultIgnoreCondition`, `IgnoreReadOnlyProperties`, `IgnoreReadOnlyFields`와 같은 글로벌 설정에 관계없이 항상 직렬화되고 역직렬화됩니다.
- `WhenWritingDefault` - 속성이 참조 형식일 때, nullable 값 형식`null`일 때, 또는 값 형식 `null`일 때 serialization에서 무시됩니다`default`.

- `WhenWritingNull` - 속성이 참조 형식이거나 nullable 값 형식 `null`인 경우 serialization에서 무시됩니다 `null`.

다음 예제에서는 [JsonIgnore] 특성의 `Condition` 속성을 사용하는 방법을 보여 줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace JsonIgnoreAttributeExample
{
    public class Forecast
    {
        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingDefault)]
        public DateTime Date { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.Never)]
        public int TemperatureC { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Run()
        {
            Forecast forecast = new()
            {
                Date = default,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast,options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
//
//{"TemperatureC":0}
```

모든 읽기 전용 속성 무시

공용 getter가 포함되지만 공용 setter가 없는 경우 속성은 읽기 전용입니다. serialize할 때 모든 읽기 전용 속성을 무시하려면 다음 예제와 같이 다음으로 설정합니다 JsonSerializerOptions.IgnoreReadOnlyProperties `true`.

C#

```
var options = new JsonSerializerOptions
{
    IgnoreReadOnlyProperties = true,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

다음 예제에서는 serialize할 형식을 보여줍니다. 또한 JSON 출력도 표시합니다.

C#

```
public class WeatherForecastWithROProperty
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    public int WindSpeedReadOnly { get; private set; } = 35;
}
```

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
}
```

이 옵션은 속성에만 적용됩니다. 필드를 직렬화할 때 읽기 전용 필드를 무시하려면 전역 설정을 사용합니다JsonSerializerOptions.IgnoreReadOnlyFields.

> **참고**
>
> 읽기 전용 컬렉션 형식의 속성은 JsonSerializerOptions.IgnoreReadOnlyProperties로 설정된 경우에도 `true`에 여전히 직렬화됩니다.

모든 null-value 속성 무시

모든 null 값 속성을 무시하려면 다음 예제와 같이 속성을 DefaultIgnoreCondition로 설정합니다 WhenWritingNull.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace IgnoreNullOnSerialize
{
    public class Forecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Run()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
//
//{"Date":"2020-10-30T10:11:40.2359135-07:00","TemperatureC":0}
```

모든 기본값 속성 무시

값 형식 속성에서 기본값이 serialization되지 않도록 하려면 다음 예제와 같이 속성을 DefaultIgnoreCondition다음으로 설정합니다WhenWritingDefault.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace IgnoreValueDefaultOnSerialize
{
    public class Forecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Run()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
//
//{ "Date":"2020-10-21T15:40:06.8920138-07:00"}
```

또한 이 설정은 WhenWritingDefault null-value 참조 형식 및 nullable 값 형식 속성의 serialization을 방지합니다.

참고하십시오

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

필드 포함

아티클 • 2025. 05. 08.

기본적으로 필드는 직렬화되지 않습니다. 다음 예제와 JsonSerializerOptions.IncludeFields 같이 직렬화 또는 역직렬화할 때 필드를 포함하려면 전역 설정 또는 [JsonInclude] 특성을 사용합니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Fields
{
    public class Forecast
    {
        public DateTime Date;
        public int TemperatureC;
        public string? Summary;
    }

    public class Forecast2
    {
        [JsonInclude]
        public DateTime Date;
        [JsonInclude]
        public int TemperatureC;
        [JsonInclude]
        public string? Summary;
    }

    public class Program
    {
        public static void Run()
        {
            string json = """
                {
                    "Date":"2020-09-06T11:31:01.923395",
                    "TemperatureC":-1,
                    "Summary":"Cold"
                }
                """;
            Console.WriteLine($"Input JSON: {json}");

            var options = new JsonSerializerOptions
            {
                IncludeFields = true,
            };
            Forecast forecast = JsonSerializer.Deserialize<Forecast>(json,
options)!;

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC: {forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");
```

```
            string roundTrippedJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");

            Forecast2 forecast2 = JsonSerializer.Deserialize<Forecast2>(json)!;

            Console.WriteLine($"forecast2.Date: {forecast2.Date}");
            Console.WriteLine($"forecast2.TemperatureC:
{forecast2.TemperatureC}");
            Console.WriteLine($"forecast2.Summary: {forecast2.Summary}");

            roundTrippedJson = JsonSerializer.Serialize<Forecast2>(forecast2);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");
        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast.Date: 9/6/2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast2.Date: 9/6/2020 11:31:01 AM
//forecast2.TemperatureC: -1
//forecast2.Summary: Cold
//Output JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
```

읽기 전용 필드를 무시하려면 전역 설정을 사용합니다 JsonSerializerOptions.IgnoreReadOnlyFields .

JSON을 .NET 개체로 읽는 방법(역직렬화)

이 문서에서는 System.Text.Json 네임스페이스를 사용하여 JSON(JavaScript Object Notation)으로부터 역직렬화하는 방법을 보여줍니다. `Newtonsoft.Json`에서 기존 코드를 이식하는 경우 System.Text.Json으로 마이그레이션 방법을 참조하세요.

JSON을 역직렬화하는 일반적인 방법은 하나 이상의 JSON 속성을 나타내는 속성 및 필드가 있는 .NET 클래스를 갖거나 만드는 것입니다. 그런 다음 문자열 또는 파일에서 역직렬화하려면 JsonSerializer.Deserialize 메서드를 호출합니다. 제네릭 오버로드의 경우 제네릭 형식 매개 변수는 .NET 클래스입니다. 제네릭이 아닌 오버로드의 경우 클래스의 형식을 메서드 매개 변수로 전달합니다. 동기적 또는 비동기적으로 역직렬화할 수 있습니다.

> **팁**
>
> AI 지원을 사용하여 **JSON 문자열을 역직렬화할** 수 있습니다.

클래스에 표시되지 않는 JSON 속성은 기본적으로 무시됩니다. 또한 형식의 필수 속성이 JSON 페이로드에 없는 경우 역직렬화가 실패합니다.

예제

다음 예제에서는 컬렉션 및 중첩된 개체를 포함하는 JSON 문자열을 역직렬화하는 방법을 보여줍니다.

C#

```
using System.Text.Json;

namespace DeserializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get; set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }
```

```
    public class Program
    {
        public static void Main()
        {
            string jsonString =
                """
                {
                  "Date": "2019-08-01T00:00:00-07:00",
                  "TemperatureCelsius": 25,
                  "Summary": "Hot",
                  "DatesAvailable": [
                    "2019-08-01T00:00:00-07:00",
                    "2019-08-02T00:00:00-07:00"
                  ],
                  "TemperatureRanges": {
                    "Cold": {
                      "High": 20,
                      "Low": -10
                    },
                    "Hot": {
                      "High": 60,
                      "Low": 20
                    }
                  },
                  "SummaryWords": [
                    "Cool",
                    "Windy",
                    "Humid"
                  ]
                }
                """;

            WeatherForecast? weatherForecast =
                JsonSerializer.Deserialize<WeatherForecast>(jsonString);

            Console.WriteLine($"Date: {weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast?.Summary}");

            if (weatherForecast?.DatesAvailable != null)
            {
                foreach (DateTimeOffset dateTimeOffset in
weatherForecast.DatesAvailable)
                {
                    Console.WriteLine($"DateAvailable: {dateTimeOffset}");
                }
            }

            if (weatherForecast?.TemperatureRanges != null)
            {
                foreach (KeyValuePair<string, HighLowTemps> temperatureRange in
weatherForecast.TemperatureRanges)
                {
```

```
                    Console.WriteLine($"TemperatureRange: {temperatureRange.Key} is
{temperatureRange.Value.Low} to {temperatureRange.Value.High}");
                }
            }

            if (weatherForecast?.SummaryWords != null)
            {
                foreach (string summaryWord in weatherForecast.SummaryWords)
                {
                    Console.WriteLine($"SummaryWord: {summaryWord}");
                }
            }
        }
    }
}

/* Output:
 *
 * Date: 8/1/2019 12:00:00 AM -07:00
 * TemperatureCelsius: 25
 * Summary: Hot
 * DateAvailable: 8/1/2019 12:00:00 AM -07:00
 * DateAvailable: 8/2/2019 12:00:00 AM -07:00
 * TemperatureRange: Cold is -10 to 20
 * TemperatureRange: Hot is 20 to 60
 * SummaryWord: Cool
 * SummaryWord: Windy
 * SummaryWord: Humid
 * */
```

동기 코드를 사용하여 파일에서 역직렬화하려면 다음 예제와 같이 파일을 문자열로 읽습니다.

C#

```
using System.Text.Json;

namespace DeserializeFromFile
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string fileName = "WeatherForecast.json";
            string jsonString = File.ReadAllText(fileName);
            WeatherForecast weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(jsonString)!;
```

```
            Console.WriteLine($"Date: {weatherForecast.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast.Summary}");
        }
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

비동기 코드를 사용하여 파일에서 역직렬화하려면 DeserializeAsync 메서드를 호출합니다.

C#

```
using System.Text.Json;

namespace DeserializeFromFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            string fileName = "WeatherForecast.json";
            using FileStream openStream = File.OpenRead(fileName);
            WeatherForecast? weatherForecast =
                await JsonSerializer.DeserializeAsync<WeatherForecast>(openStream);
            Console.WriteLine($"Date: {weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast?.Summary}");
        }
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

역직렬화 동작

JSON을 역직렬화할 때 다음 동작이 적용됩니다.

- 기본적으로, 속성 이름 일치 시에 대/소문자를 구분합니다. 대/소문자를 구분 하지 않도록 지정할 수 있습니다.
- public이 아닌 생성자는 직렬 변환기에서 무시됩니다.
- 변경 불가능한 개체 또는 퍼블릭 `set` 접근자가 없는 속성에 대한 Deserialization이 지원되지만 기본적으로 활성화되지는 않습니다. 변경 불가능한 형식 및 레코드를 참조하세요.
- 기본적으로 열거형은 숫자로 지원됩니다. 문자열 열거형 필드 역직렬화를 수행할 수 있습니다.
- 기본적으로 필드는 무시됩니다. 필드를 포함할 수 있습니다.
- 기본적으로 JSON의 주석이나 후행 쉼표는 예외를 발생시킵니다. 주석과 후행 쉼표를 허용할 수 있습니다.
- 기본 최댓값은 64입니다.

ASP.NET Core 앱에서 System.Text.Json을 간접적으로 사용하는 경우 몇 가지 기본 동작이 다릅니다. 자세한 내용은 JsonSerializerOptions 웹 기본값을 참조하세요.

기본 변환기에서 지원하지 않는 기능을 제공하는 사용자 지정 변환기를 구현할 수 있습니다.

.NET 클래스 없이 역직렬화

역직렬화하려는 JSON이 있고 역직렬화 대상 클래스가 없는 경우 필요한 클래스를 수동으로 만드는 것 외에 다른 옵션이 있습니다.

- Utf8JsonReader를 직접 사용합니다.
- JSON DOM(문서 개체 모델)으로 역직렬화하고 DOM에서 필요한 항목을 추출합니다.

  DOM을 사용하면 JSON 페이로드의 하위 섹션으로 이동하고 단일 값, 사용자 지정 형식 또는 배열을 역직렬화할 수 있습니다. JsonNode DOM에 대한 자세한 내용은 JSON 페이로드의 하위 섹션 역직렬화를 참조하세요. JsonDocument DOM에 대한 자세한 내용은 JsonDocument 및 JsonElement에서 하위 요소를 검색하는 방법을 참조하세요.

- Visual Studio 2022 이상을 사용하여 필요한 클래스를 자동으로 생성합니다.
  - 역직렬화해야 하는 JSON을 복사합니다.
  - 클래스 파일을 만들고 템플릿 코드를 삭제합니다.
  - **편집**>**특수 붙여넣기**>**JSON을 클래스로 붙여넣기**를 선택합니다.

  결과는 역직렬화 대상에 사용할 수 있는 클래스입니다.

UTF-8에서 역직렬화

UTF-8에서 역직렬화하려면 다음 예제와 같이 JsonSerializer.Deserialize 또는 `ReadOnlySpan<byte>`을 사용하는 `Utf8JsonReader` 오버로드를 호출합니다. 이 예제에서는 JSON

이 jsonUtf8Bytes라는 바이트 배열에 있다고 가정합니다.

C#

```
var readOnlySpan = new ReadOnlySpan<byte>(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(readOnlySpan)!;
```

C#

```
var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(ref utf8Reader)!;
```

AI를 사용하여 JSON 역직렬화

GitHub Copilot와 같은 AI 도구를 사용하여 JSON에서 역직렬화하는 데 사용하는 `System.Text.Json` 코드를 생성할 수 있습니다. 예를 들어 대상 클래스가 JSON 입력에서 누락된 속성을 정의할 때 역직렬화를 보여 주도록 프롬프트를 사용자 지정할 수 있습니다.

다음 텍스트는 Copilot 채팅의 프롬프트 예를 보여줍니다.

Copilot 프롬프트

```
Generate C# code to use System.Text.Json to deserialize a JSON string
{"FirstName":"John","LastName":"Doe"} to an equivalent .NET object, where the class
defines an Age property.
Show what happens when the JSON is missing a property defined in the class.
Provide example output.
```

코필로트의 제안을 적용하기 전에 검토합니다.

GitHub Copilot에 대한 자세한 내용은 GitHub의 FAQ를 참조하세요.

참고하십시오

- 인터페이스로 역직렬화하는 방법
- Visual Studio의 GitHub Copilot
- Visual Studio Code의 GitHub Copilot

Last updated on 2025. 12. 13.

필수 속성

역직렬화가 성공하려면 JSON 페이로드에 있어야 함을 나타내기 위해 특정 속성을 표시할 수 있습니다. 마찬가지로 선택 사항이 아닌 모든 생성자 매개 변수가 JSON 페이로드에 있음을 지정하는 옵션을 설정할 수 있습니다. 이러한 필수 속성 중 하나 이상이 없는 경우 JsonSerializer.Deserialize 메서드는 JsonException을 throw합니다.

JSON deserialization에 필요한 속성 또는 필드를 표시하는 방법에는 다음 세 가지가 있습니다.

- 한정자를 required 추가합니다.
- JsonRequiredAttribute를 사용하여 주석을 추가합니다.
- 계약 모델의 JsonPropertyInfo.IsRequired 속성을 수정함으로써.

JSON 역직렬화 시 모든 비선택적 생성자 매개변수가 필수임을 지정하려면, JsonSerializerOptions.RespectRequiredConstructorParameters 옵션(또는 소스 생성의 경우 RespectRequiredConstructorParameters 속성)을 `true`로 설정합니다. 자세한 내용은 선택 사항이 아닌 생성자 매개 변수 섹션을 참조하세요.

직렬 변환기의 관점에서 C# `required` 한정자와 [JsonRequired] 특성은 동일하며, 둘 다 동일한 메타데이터 조각(예:)에 매핑됩니다 JsonPropertyInfo.IsRequired. 대부분의 경우 기본 제공 C# 키워드만 사용합니다. 그러나 다음 경우에는 JsonRequiredAttribute를 대신 사용해야 합니다.

- C# 이외의 프로그래밍 언어 또는 C#의 하위 수준 버전을 사용하는 경우
- 요구 사항을 JSON deserialization에만 적용하려는 경우
- 소스 생성 모드에서 `System.Text.Json`를 사용하는 경우 여기서는 소스 생성이 컴파일 시간에 발생하므로 `required` 한정자를 사용하는 경우 코드가 컴파일되지 않습니다.

다음 코드 조각은 `required` 키워드로 수정된 속성의 예를 보여 줍니다. deserialization이 성공하려면 이 속성이 JSON 페이로드에 있어야 합니다.

C#

```
public static void RunIt()
{
    // The following line throws a JsonException at run time.
    Console.WriteLine(JsonSerializer.Deserialize<Person>("""{"Age": 42}"""));
}

public class Person
{
    public required string Name { get; set; }
    public int Age { get; set; }
}
```

또는 JsonRequiredAttribute를 사용할 수 있습니다.

C#

```
public static void RunIt()
{
    // The following line throws a JsonException at run time.
    Console.WriteLine(JsonSerializer.Deserialize<Person>("""{"Age": 42}"""));
}

public class Person
{
    [JsonRequired]
    public string Name { get; set; }
    public int Age { get; set; }
}
```

JsonPropertyInfo.IsRequired 속성을 사용하여 계약 모델을 통해 특정 속성이 필수인지 제어할 수도 있습니다.

C#

```
public static void RunIt()
{
    var options = new JsonSerializerOptions
    {
        TypeInfoResolver = new DefaultJsonTypeInfoResolver
        {
            Modifiers =
            {
                static typeInfo =>
                {
                    if (typeInfo.Kind != JsonTypeInfoKind.Object)
                        return;

                    foreach (JsonPropertyInfo propertyInfo in typeInfo.Properties)
                    {
                        // Strip IsRequired constraint from every property.
                        propertyInfo.IsRequired = false;
                    }
                }
            }
        }
    };

    // Deserialization succeeds even though
    // the Name property isn't in the JSON payload.
    JsonSerializer.Deserialize<Person>("""{"Age": 42}""", options);
}

public class Person
{
    public required string Name { get; set; }
```

```
    public int Age { get; set; }
}
```

선택 사항이 아닌 생성자 매개 변수

.NET 9 이전의 생성자 기반 역직렬화는 다음 예제와 같이 모든 생성자 매개 변수를 선택 사항으로 처리했습니다.

C#

```
var result = JsonSerializer.Deserialize<Person>("{}");
Console.WriteLine(result); // Person { Name = , Age = 0 }

record Person(string Name, int Age);
```

.NET 9부터 필요에 따라 선택적이 아닌 생성자 매개 변수를 처리하도록 플래그를 설정할 RespectRequiredConstructorParameters 수 있습니다.

C#

```
public static void RunIt()
{
    JsonSerializerOptions options = new()
    {
        RespectRequiredConstructorParameters = true
    };
    string json = """{"Age": 42}""";

    // The following line throws a JsonException at run time.
    JsonSerializer.Deserialize<Person>(json, options);
}

record Person(string Name, int? Age = null);
```

기능 전환

`RespectRequiredConstructorParameters` 설정을 전역적으로 켜려면 `System.Text.Json.Serialization.RespectRequiredConstructorParametersDefault` 기능 스위치를 사용하십시오. 프로젝트 파일(예 : *.csproj* 파일)에 다음 MSBuild 항목을 추가합니다.

XML

```
<ItemGroup>
  <RuntimeHostConfigurationOption
Include="System.Text.Json.Serialization.RespectRequiredConstructorParametersDefault
```

```
" Value="true" />
</ItemGroup>
```

API는 `RespectRequiredConstructorParametersDefault` 기존 애플리케이션을 중단하지 않도록 .NET 9에서 옵트인 플래그로 구현되었습니다. 새 애플리케이션을 작성하는 경우 코드에서 이 플래그를 사용하도록 설정하는 것이 좋습니다.

참고 항목

- 널 가능 주석을 존중하다

Last updated on 2026. 02. 24.

null 허용 주석을 준수하십시오

.NET 9부터 JsonSerializer는 직렬화 및 역직렬화 모두에서 비-널 참조 형식 강제 적용을 제한적으로 지원합니다. JsonSerializerOptions.RespectNullableAnnotations 플래그를 사용하여 이 지원을 전환할 수 있습니다.

예를 들어 다음 코드 조각은 직렬화 중에 다음과 같은 메시지와 함께 JsonException를 발생시킵니다.

> 'Person' 형식의 속성 또는 필드 'Name'은 null 값을 가져올 수 없습니다. Null 허용 여부 주석을 업데이트하는 것이 좋습니다.

C#

```
    public static void RunIt()
    {
#nullable enable
        JsonSerializerOptions options = new()
        {
            RespectNullableAnnotations = true
        };

        Person invalidValue = new(Name: null!);
        JsonSerializer.Serialize(invalidValue, options);
    }

    record Person(string Name);
```

마찬가지로 RespectNullableAnnotations 역직렬화에 null 허용 불가를 강제합니다. 다음 코드 조각은 직렬화 중에 JsonException를 다음과 같은 메시지로 throw합니다.

> 'Person' 형식의 생성자 매개 변수 'Name'은 null 값을 허용하지 않습니다. Null 허용 여부 주석을 업데이트하는 것이 좋습니다.

C#

```
    public static void RunIt()
    {
#nullable enable
        JsonSerializerOptions options = new()
        {
            RespectNullableAnnotations = true
        };

        string json = """{"Name":null}""";
        JsonSerializer.Deserialize<Person>(json, options);
    }
```

```
record Person(string Name);
```

> **팁**
>
> - 및 IsGetNullable 속성을 사용하여 IsSetNullable 개별 속성 수준에서 null 허용 성을 구성할 수 있습니다.
> - C# 컴파일러는 , [NotNull]및 [AllowNull]특성을 사용하여 [MaybeNull]getter 및 [DisallowNull] setter에서 주석을 미세 조정합니다. 이러한 특성은 이 System.Text.Json 기능에서도 인식됩니다. 속성에 대한 자세한 내용은 **null 상태 정적 분석 속성**을 참조하세요.

제한 사항

nullable이 아닌 참조 형식을 구현하는 방법 때문에 이 기능에는 몇 가지 중요한 제한 사항이 있습니다. 기능을 켜기 전에 이러한 제한 사항을 숙지하세요. 문제의 근본 원인은 참조 형식 nullability에 IL(중간 언어)의 첫 번째 클래스 표현이 없다는 것입니다. 따라서 런타임 리플렉션의 관점에서 보면, 식 `MyPoco` 과 `MyPoco?` 은 구별할 수 없습니다. 컴파일러가 특성 메타데이터를 내보내서 이를 만회하려고 하는 동안(예제 sharplab.io 참조), 이 메타데이터는 특정 형식 정의로 범위가 지정된 제네릭이 아닌 멤버 주석으로 제한됩니다. 이 제한 사항은 플래그가 제네릭이 아닌 속성, 필드 및 생성자 매개 변수에 있는 null 허용 여부 주석만 유효성을 검사하는 이유입니다. System.Text.Json 에서는 다음에서 Null 허용 여부 적용을 지원하지 않습니다.

- 최상위 형식 또는 첫 번째 `JsonSerializer.Deserialize()` 또는 `JsonSerializer.Serialize()` 호출을 할 때 전달되는 형식입니다.
- 컬렉션 요소 형식(예: `List<string>` 형식 및 `List<string?>` 형식은 구별할 수 없음)입니다.
- 제네릭인 모든 속성, 필드 또는 생성자 매개 변수입니다.

이러한 경우 null 허용 여부 적용을 추가하려면 형식을 구조체로 모델링하거나(null 값을 인정하지 않으므로) 해당 HandleNull 속성을 `true`재정의하는 사용자 지정 변환기를 작성합니다.

기능 전환

`RespectNullableAnnotations` 설정을 전역적으로 켜려면 `System.Text.Json.Serialization.RespectNullableAnnotationsDefault` 기능 스위치를 사용하십시오. 프로젝트 파일(예 : *.csproj* 파일)에 다음 MSBuild 항목을 추가합니다.

XML

```
<ItemGroup>
  <RuntimeHostConfigurationOption
Include="System.Text.Json.Serialization.RespectNullableAnnotationsDefault"
Value="true" />
</ItemGroup>
```

API는 `RespectNullableAnnotationsDefault` 기존 애플리케이션을 중단하지 않도록 .NET 9에서 옵트인 플래그로 구현되었습니다. 새 애플리케이션을 작성하는 경우 코드에서 이 플래그를 사용하도록 설정하는 것이 좋습니다.

nullable 매개 변수와 선택적 매개 변수 간의 관계

RespectNullableAnnotations는 필수 및 null 불가능한 속성을 독립적인 개념으로 처리하므로 System.Text.Json은 지정되지 않은 JSON 값에 대한 적용을 확장하지 않습니다. 예를 들어 다음 코드 조각은 역직렬화 중에 예외를 throw하지 않습니다.

C#

```
public static void RunIt()
{
    JsonSerializerOptions options = new()
    {
        RespectNullableAnnotations = true
    };
    var result = JsonSerializer.Deserialize<MyPoco>("{}", options);
    Console.WriteLine(result.Name is null); // True.
}

class MyPoco
{
    public string Name { get; set; }
}
```

이 동작은 nullable 필수 속성을 가질 수 있는 C# 언어 자체에서 비롯됩니다.

C#

```
MyPoco poco = new() { Value = null }; // No compiler warnings.

class MyPoco
{
    public required string? Value { get; set; }
}
```

또한 null을 허용하지 않는 선택적 속성을 가질 수도 있습니다.

C#

```
class MyPoco
{
    public string Value { get; set; } = "default";
}
```

생성자 매개 변수에 동일한 직교가 적용됩니다.

C#

```
record MyPoco(
    string RequiredNonNullable,
    string? RequiredNullable,
    string OptionalNonNullable = "default",
    string? OptionalNullable = "default"
    );
```

누락된 값과 null 값

설정할 때 *누락된 JSON 속성*과 *명시적 `null` 값이 있는 속성* 간의 차이점을 이해하는 것이 중요합니다. JavaScript는 `undefined` (누락된 속성) 및 `null` (명시적 null 값)을 구분합니다. 그러나 .NET에는 `undefined`라는 개념이 없으므로 두 경우 모두 `null`로 .NET에서 역직렬화됩니다.

역직렬화 중에 `RespectNullableAnnotations`가 `true`일 때,

- **명시적 null 값은 null을 허용하지 않는 속성에 대해 예외를 발생시킵니다.** 예를 들어 `{"Name":null}`는 null이 될 수 없는 `string Name` 속성으로 역직렬화할 때 예외를 throw합니다.
- **누락된 속성**은 nullable이 아닌 속성에 대해서도 예외를 throw하지 않습니다. 예를 들어, `{}`를 null이 될 수 없는 `string Name` 속성으로 역직렬화할 때 예외를 발생시키지 않습니다. serializer는 속성을 설정하지 않고 생성자에서 기본값으로 남습니다. 초기화되지 않은 null 허용 안 함 참조 형식의 경우 `null`가 발생하며, 컴파일러 경고가 트리거됩니다.

  다음 코드에서는 역직렬화 중에 누락된 속성이 예외를 throw하지 않는 방법을 보여 있습니다.

  C#

  ```
      public static void RunIt()
      {
  #nullable enable
          JsonSerializerOptions options = new()
          {
              RespectNullableAnnotations = true
  ```

```
        };

        // Missing property - does NOT throw an exception.
        string jsonMissing = """{}""";
        var resultMissing = JsonSerializer.Deserialize<Person>(jsonMissing,
options);
        Console.WriteLine(resultMissing.Name is null); // True.
    }

    record Person(string Name);
```

이 동작 차이는 누락된 속성이 선택 사항(제공되지 않음)으로 처리되고 명시적 `null` 값은 nullable이 아닌 제약 조건을 위반하는 제공된 값으로 처리되기 때문에 발생합니다. JSON에 속성이 있어야 하므로 적용해야 하는 경우 한정자를 사용 `required` 하거나 필요에 따라 속성을 구성하거나 계약 모델을 사용합니다 JsonRequiredAttribute .

참고 항목

- 선택 사항이 아닌 생성자 매개 변수

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 24.

다음과 같은 특정 종류의 잘못된 JSON을 허용하는 방법 System.Text.Json

2025. 06. 22.

이 문서에서는 JSON에서 주석, 후행 쉼표 및 따옴표 붙은 숫자를 허용하는 방법과 숫자를 문자열로 작성하는 방법을 알아봅니다.

메모 및 후행 쉼표 허용

기본적으로 주석 및 후행 쉼표는 JSON에서 허용되지 않습니다. JSON에서 주석을 허용하려면 속성을 JsonSerializerOptions.ReadCommentHandling.로 설정합니다`JsonCommentHandling.Skip`. 후행 쉼표가 허용되도록 하려면 속성을 JsonSerializerOptions.AllowTrailingCommas.로 설정합니다`true`. 다음 예제에서는 두 가지를 모두 허용하는 방법을 보여 있습니다.

C#

```
var options = new JsonSerializerOptions
{
    ReadCommentHandling = JsonCommentHandling.Skip,
    AllowTrailingCommas = true,
};
WeatherForecast weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
    jsonString,
    options
    )!;
```

다음은 주석과 후행 쉼표가 있는 JSON 예제입니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25, // Fahrenheit 77
  "Summary": "Hot", /* Zharko */
  // Comments on
  /* separate lines */
}
```

따옴표 안에 숫자를 허용하거나 씁니다.

일부 serializer는 숫자를 JSON 문자열로 인코딩합니다(따옴표로 묶음).

다음은 그 예입니다.

JSON

```
{
    "DegreesCelsius": "23"
}
```

대신에:

JSON

```
{
    "DegreesCelsius": 23
}
```

숫자를 따옴표로 직렬화하거나 전체 입력 개체 그래프에서 따옴표로 숫자를 허용하려면 다음 예제와 같이 설정합니다 JsonSerializerOptions.NumberHandling .

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace QuotedNumbers
{
    public class Forecast
    {
        public DateTime Date { get; init; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Run()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new()
            {
                NumberHandling =
                    JsonNumberHandling.AllowReadingFromString |
                    JsonNumberHandling.WriteAsString,
                WriteIndented = true
            };

            string forecastJson =
```

```
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON:\n{forecastJson}");

            Forecast forecastDeserialized =
                JsonSerializer.Deserialize<Forecast>(forecastJson, options)!;

            Console.WriteLine($"Date: {forecastDeserialized.Date}");
            Console.WriteLine($"TemperatureC:
{forecastDeserialized.TemperatureC}");
            Console.WriteLine($"Summary: {forecastDeserialized.Summary}");
        }
    }
}

// Produces output like the following example:
//
//Output JSON:
//{
//  "Date": "2020-10-23T12:27:06.4017385-07:00",
//  "TemperatureC": "40",
//  "Summary": "Hot"
//}
//Date: 10/23/2020 12:27:06 PM
//TemperatureC: 40
//Summary: Hot
```

ASP.NET Core를 통해 간접적으로 사용하는 `System.Text.Json` 경우 ASP.NET Core는 웹 기본 옵션을 지정하기 때문에 역직렬화할 때 따옴표 붙은 숫자가 허용됩니다.

특정 속성, 필드 또는 형식에 따옴표 붙은 숫자를 허용하거나 쓰려면 [JsonNumberHandling] 특성을 사용합니다.

참고하십시오

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

역직렬화 중에 매핑되지 않은 멤버 처리

기본적으로 역직렬화하는 JSON 페이로드에 일반 이전 CLR 개체(POCO) 형식에 없는 속성이 포함되어 있으면 그들은 무시됩니다. .NET 8부터 모든 페이로드 속성이 POCO에 존재해야 한다고 지정할 수 있습니다. 그렇지 않은 경우, 역직렬화 중에 JsonException 예외가 발생합니다. 다음 세 가지 방법 중 하나로 이 동작을 구성할 수 있습니다.

- POCO 형식에 JsonUnmappedMemberHandlingAttribute 특성으로 주석을 추가하고, 매핑되지 않은 멤버를 Skip 또는 Disallow로 지정합니다.

  C#

  ```
  [JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
  public class MyPoco
  {
      public int Id { get; set; }
  }

  JsonSerializer.Deserialize<MyPoco>("""{"Id" : 42, "AnotherId" : -1 }""");
  // JsonException : The JSON property 'AnotherId' could not be mapped to any
  .NET member contained in type 'MyPoco'.
  ```

- JsonSerializerOptions.UnmappedMemberHandling를 Skip 또는 Disallow로 설정합니다.
- 관련 유형의 JsonTypeInfo 계약을 사용자 지정합니다. (계약 사용자 지정에 대한 자세한 내용은 JSON 계약 사용자 지정을 참조하세요.)

Last updated on 2026. 02. 24.

System.Text.Json에서 오버플로 JSON을 처리하거나 JsonElement 또는 JsonNode를 사용하는 방법

아티클 • 2023. 05. 26.

이 문서에서는 System.Text.Json 네임스페이스를 사용하여 오버플로 JSON을 처리하는 방법을 보여줍니다. 또한 대상 형식이 역직렬화되는 모든 JSON과 완벽하게 일치하지 않을 수 있는 다른 시나리오의 대안으로 JsonElement 또는 JsonNode로 역직렬화하는 방법도 보여줍니다.

오버플로 JSON 처리

역직렬화할 때 대상 형식의 속성으로 표시되지 않는 데이터를 JSON에서 받을 수 있습니다. 예를 들어 대상 형식이 다음과 같다고 가정하겠습니다.

C#

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

그리고 역직렬화할 JSON은 다음과 같습니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

표시된 JSON을 표시된 형식으로 역직렬화하면 `DatesAvailable` 및 `SummaryWords` 속성은 이동할 곳이 없으므로 없어집니다. 이러한 속성과 같은 추가 데이터를 캡처하려면 다음과 같이 JsonExtensionData 특성을 `Dictionary<string,object>` 또는 `Dictionary<string,JsonElement>` 형식의 속성에 적용합니다.

C#

```
public class WeatherForecastWithExtensionData
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonExtensionData]
    public Dictionary<string, JsonElement>? ExtensionData { get; set; }
}
```

다음 표에서는 이전에 표시된 JSON을 이 샘플 형식으로 역직렬화한 결과를 보여줍니다. 추가 데이터는 `ExtensionData` 속성의 키-값 쌍이 됩니다.

속성	값	참고
`Date`	`"8/1/2019 12:00:00 AM -07:00"`	
`TemperatureCelsius`	`0`	대/소문자를 구분하는 불일치(JSON의 `temperatureCelsius`). 따라서 속성이 설정되지 않습니다.
`Summary`	`"Hot"`	
`ExtensionData`	`"temperatureCelsius": 25, "DatesAvailable": ["2019-08-01T00:00:00-07:00","2019-08-02T00:00:00-07:00"], "SummaryWords": ["Cool","Windy","Humid"]`	대/소문자가 일치하지 않으므로 `temperatureCelsius`는 추가 속성이며 사전에서 키-값 쌍이 됩니다. JSON의 추가 배열은 각각 키-값 쌍이 되고, 배열은 값 개체로 사용됩니다.

대상 개체가 직렬화되면 확장 데이터 키 값 쌍은 마치 수신 JSON에 있는 것처럼 JSON 속성이 됩니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 0,
  "Summary": "Hot",
  "temperatureCelsius": 25,
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
```

```
    "2019-08-02T00:00:00-07:00"
  ],
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

`ExtensionData` 속성 이름은 JSON에 표시되지 않습니다. 이 동작을 통해 JSON은 추가 데이터를 잃지 않고도 라운드트립을 수행할 수 있으며, 이 동작이 없으면 추가 데이터가 역직렬화되지 않습니다.

다음 예제에서는 JSON에서 역직렬화된 개체로 이동해 다시 JSON으로 복귀하는 왕복을 보여줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace RoundtripExtensionData
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        [JsonExtensionData]
        public Dictionary<string, JsonElement>? ExtensionData { get; set; }
    }
    public class Program
    {
        public static void Main()
        {
            string jsonString =
@"{
  ""Date"": ""2019-08-01T00:00:00-07:00"",
  ""temperatureCelsius"": 25,
  ""Summary"": ""Hot"",
  ""SummaryField"": ""Hot"",
  ""DatesAvailable"": [
    ""2019-08-01T00:00:00-07:00"",
    ""2019-08-02T00:00:00-07:00""
  ],
  ""SummaryWords"": [
    ""Cool"",
    ""Windy"",
    ""Humid""
  ]
}";
            WeatherForecast weatherForecast =
```

```
                JsonSerializer.Deserialize<WeatherForecast>(jsonString)!;

            var serializeOptions = new JsonSerializerOptions { WriteIndented
= true };
            jsonString = JsonSerializer.Serialize(weatherForecast,
serializeOptions);
            Console.WriteLine($"JSON output:\n{jsonString}\n");
        }
    }
}
// output:
//JSON output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 0,
//  "Summary": "Hot",
//  "temperatureCelsius": 25,
//  "SummaryField": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "SummaryWords": [
//    "Cool",
//    "Windy",
//    "Humid"
//  ]
//}
```

JsonElement 또는 JsonNode로 역직렬화

특정 속성에 허용되는 JSON을 유연하게 사용하려는 경우 또는 JsonNode로 JsonElement 역직렬화하는 것이 대안입니다. 유효한 JSON 속성은 `JsonElement` 또는 `JsonNode`로 역직렬화할 수 있습니다. 변경할 수 없는 개체를 만들려면 `JsonElement`를 선택하고 변경 가능한 개체를 만들려면 `JsonNode`를 선택합니다.

다음 예제에서는 형식 `JsonElement` 및 `JsonNode`의 속성을 포함하는 클래스에 대해 JSON에서 출발해 JSON으로 복귀하는 왕복을 보여줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Nodes;

namespace RoundtripJsonElementAndNode
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
```

```
        public string? Summary { get; set; }
        public JsonElement DatesAvailable { get; set; }
        public JsonNode? SummaryWords { get; set; }
    }
        public class Program
    {
        public static void Main()
        {
            string jsonString =
@"{
  ""Date"": ""2019-08-01T00:00:00-07:00"",
  ""TemperatureCelsius"": 25,
  ""Summary"": ""Hot"",
  ""DatesAvailable"": [
    ""2019-08-01T00:00:00-07:00"",
    ""2019-08-02T00:00:00-07:00""
  ],
  ""SummaryWords"": [
    ""Cool"",
    ""Windy"",
    ""Humid""
  ]
}";
            WeatherForecast? weatherForecast =
                JsonSerializer.Deserialize<WeatherForecast>(jsonString);

            var serializeOptions = new JsonSerializerOptions { WriteIndented
= true };
            jsonString = JsonSerializer.Serialize(weatherForecast,
serializeOptions);
            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "SummaryWords": [
//    "Cool",
//    "Windy",
//    "Humid"
//  ]
//}
```

참고 항목

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

변경할 수 없는 형식 및 속성 사용

아티클 • 2025. 02. 06.

변경할 수 없는 *형식*/ 인스턴스화된 후 개체의 속성 또는 필드 값을 변경하지 못하게 하는 형식입니다. 형식은 레코드일 수 있으며, 공용 속성이나 필드가 없을 수도 있고, 읽기 전용 속성이 있을 수도 있으며, private 또는 초기화 용도로만 설정할 수 있는 setter가 있는 속성이 있을 수 있습니다. System.String 변경할 수 없는 형식의 예입니다. System.Text.Json 변경할 수 없는 형식으로 JSON을 역직렬화할 수 있는 다양한 방법을 제공합니다.

매개 변수가 있는 생성자

기본적으로 `System.Text.Json` 기본 공용 매개 변수 없는 생성자를 사용합니다. 그러나 매개 변수가 있는 생성자를 사용하도록 지시할 수 있으므로 변경할 수 없는 클래스 또는 구조체를 역직렬화할 수 있습니다.

- 클래스의 경우 매개 변수가 있는 생성자만 있으면 해당 생성자가 사용됩니다.
- 구조체 또는 여러 생성자가 있는 클래스의 경우 [JsonConstructor] 특성을 적용하여 사용할 클래스를 지정합니다. 특성을 사용하지 않으면 공용 매개 변수가 없는 생성자가 항상 사용됩니다(있는 경우).

  다음 예제에서는 `[JsonConstructor]` 특성을 사용합니다.

```
C#

using System.Text.Json;
using System.Text.Json.Serialization;

namespace ImmutableTypes
{
    public struct Forecast
    {
        public DateTime Date { get; }
        public int TemperatureC { get; }
        public string Summary { get; }

        [JsonConstructor]
        public Forecast(DateTime date, int temperatureC, string
summary) =>
            (Date, TemperatureC, Summary) = (date, temperatureC,
summary);
    }

    public class Program
    {
        public static void Main()
```

```
        {
            string json = """
                {
                    "date":"2020-09-06T11:31:01.923395-07:00",
                    "temperatureC":-1,
                    "summary":"Cold"
                }
                """;
            Console.WriteLine($"Input JSON: {json}");

            var options = JsonSerializerOptions.Web;

            Forecast forecast = JsonSerializer.Deserialize<Forecast>
(json, options);

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");

            string roundTrippedJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");
        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","temperatureC":-1,"summary":"Cold"}
//forecast.Date: 9 / 6 / 2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","temperatureC":-1,"summary":"Cold"}
```

.NET 7 및 이전 버전에서 `[JsonConstructor]` 특성은 공용 생성자와만 사용할 수 있습니다.

매개 변수가 있는 생성자의 매개 변수 이름은 속성 이름 및 형식과 일치해야 합니다. 일치는 대/소문자를 구분하지 않으며 [JsonPropertyName] 사용하여 속성 이름을 바꾸는 경우에도 생성자 매개 변수가 실제 속성 이름과 일치해야 합니다. 다음 예제에서는 `TemperatureC` 속성의 이름이 JSON에서 `celsius` 변경되지만 생성자 매개 변수의 이름은 여전히 `temperatureC`.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
```

```
namespace ImmutableTypesCtorParms
{
    public readonly struct Forecast
    {
        public DateTime Date { get; }
        [JsonPropertyName("celsius")]
        public int TemperatureC { get; }
        public string Summary { get; }

        [JsonConstructor]
        public Forecast(DateTime date, int temperatureC, string summary) =>
            (Date, TemperatureC, Summary) = (date, temperatureC, summary);
    }

    public class Program
    {
        public static void Main()
        {
            string json = """
                {
                    "date":"2020-09-06T11:31:01.923395-07:00",
                    "celsius":-1,
                    "summary":"Cold"
                }
                """;
            Console.WriteLine($"Input JSON: {json}");

            var options = JsonSerializerOptions.Web;

            Forecast forecast = JsonSerializer.Deserialize<Forecast>(json,
options);

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");

            string roundTrippedJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");

        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","celsius":-1,"summary":"Cold"}
//forecast.Date: 9 / 6 / 2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
```

```
//Output JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","celsius":-1,"summary":"Cold"}
```

`[JsonPropertyName]`외에도 다음 특성은 매개 변수가 있는 생성자를 사용하여 역직렬화를 지원합니다.

- [JsonConverter]
- [JsonIgnore]
- [JsonInclude]
- [JsonNumberHandling]

레코드

다음 예제와 같이 직렬화 및 역직렬화 모두에 대해 레코드가 지원됩니다.

C#

```
using System.Text.Json;

namespace Records
{
    public record Forecast(DateTime Date, int TemperatureC)
    {
        public string? Summary { get; init; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new(DateTime.Now, 40)
            {
                Summary = "Hot!"
            };

            string forecastJson = JsonSerializer.Serialize<Forecast>
(forecast);
            Console.WriteLine(forecastJson);
            Forecast? forecastObj = JsonSerializer.Deserialize<Forecast>
(forecastJson);
            Console.WriteLine(forecastObj);
        }
    }
}

// Produces output like the following example:
//
//{ "Date":"2020-10-21T15:26:10.5044594-
07:00","TemperatureC":40,"Summary":"Hot!"}
```

```
//Forecast { Date = 10 / 21 / 2020 3:26:10 PM, TemperatureC = 40, Summary =
Hot! }
```

속성에 `property:` 대상을 사용하여 속성 이름에 어떤 특성이든 적용할 수 있습니다. 위치 레코드에 대한 자세한 내용은 C# 언어 참조의 레코드에 대한 문서를 참조하세요.

비공개 멤버 및 속성 접근자

다음 예제와 같이 [JsonInclude] 특성을 사용하여 속성에 대한 비공용 *접근자* 사용하도록 설정할 수 있습니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace NonPublicAccessors
{
    public class Forecast
    {
        public DateTime Date { get; init; }

        [JsonInclude]
        public int TemperatureC { get; private set; }

        [JsonInclude]
        public string? Summary { private get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            string json = """
                {
                    "Date":"2020-10-23T09:51:03.8702889-07:00",
                    "TemperatureC":40,
                    "Summary":"Hot"
                }
                """;
            Console.WriteLine($"Input JSON: {json}");

            Forecast forecastDeserialized =
JsonSerializer.Deserialize<Forecast>(json)!;
            Console.WriteLine($"Date: {forecastDeserialized.Date}");
            Console.WriteLine($"TemperatureC:
{forecastDeserialized.TemperatureC}");

            json = JsonSerializer.Serialize<Forecast>(forecastDeserialized);
            Console.WriteLine($"Output JSON: {json}");
        }
```

```
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-10-23T09:51:03.8702889-
07:00","TemperatureC":40,"Summary":"Hot"}
//Date: 10 / 23 / 2020 9:51:03 AM
//TemperatureC: 40
//Output JSON: { "Date":"2020-10-23T09:51:03.8702889-
07:00","TemperatureC":40,"Summary":"Hot"}
```

속성에 프라이빗 setter를 포함하면 해당 속성을 역직렬화할 수 있습니다.

.NET 8 이상 버전에서는 [JsonInclude] 특성을 사용하여 지정된 형식의 serialization 계약에 *공용이 아닌* 멤버를 선택할 수도 있습니다.

> **참고**
>
> 소스 생성 모드에서는 `private` 멤버를 직렬화하거나 [JsonInclude] 특성에 주석을 추가하여 `private` 접근자를 사용할 수 없습니다. 생성된 JsonSerializerContext와 동일한 어셈블리에 있는 경우에만 `internal` 멤버를 직렬화하거나 `internal` 접근자를 사용할 수 있습니다.

읽기 전용 속성

.NET 8 이상 버전에서는 읽기 전용 속성 또는 프라이빗 또는 퍼블릭 setter가 없는 속성도 역직렬화할 수 있습니다. 속성이 참조하는 인스턴스는 변경할 수 없지만 속성 형식이 변경 가능한 경우 수정할 수 있습니다. 예를 들어 목록에 요소를 추가할 수 있습니다. 읽기 전용 속성을 역직렬화하려면 개체 만들기 처리 동작을 *바꿀대신* 채우도록 설정해야 합니다. 예를 들어 JsonObjectCreationHandlingAttribute 특성으로 속성에 주석을 추가할 수 있습니다.

C#

```
class A
{
    [JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
    public List<int> Numbers1 { get; } = new List<int>() { 1, 2, 3 };
}
```

자세한 내용은 초기화된 속성 채우기참조하세요.

참고 항목

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화하는 방법

초기화된 속성 채우기

.NET 8부터 JSON이 역직렬화될 때 .NET 속성을 바꾸 거나 채우는 기본 설정을 지정할 수 있습니다. 이 열거형은 JsonObjectCreationHandling 개체 생성 처리 옵션을 제공합니다.

- JsonObjectCreationHandling.Replace
- JsonObjectCreationHandling.Populate

기본(바꾸기) 동작

System.Text.Json 역직렬 변환기는 항상 대상 형식의 새 인스턴스를 만듭니다. 그러나 새 인스턴스가 만들어지더라도 일부 속성 및 필드는 개체 생성의 일부로 이미 초기화되었을 수 있습니다. 다음 형식을 고려합니다.

C#

```
class A
{
    public List<int> Numbers1 { get; } = [1, 2, 3];
    public List<int> Numbers2 { get; set; } = [1, 2, 3];
}
```

이 클래스 `Numbers1` 의 인스턴스를 만들 때 (및 `Numbers2`) 속성의 값은 세 개의 요소(1, 2 및 3)가 있는 목록입니다. JSON을 이 형식으로 역직렬화하는 경우 *기본* 동작은 속성 값이 *대체*된다는 것입니다.

- 의 경우 `Numbers1`읽기 전용(setter 없음)이므로 목록에 값 1, 2 및 3이 있습니다.
- 읽기-쓰기인 경우 `Numbers2`새 목록이 할당되고 JSON의 값이 추가됩니다.

예를 들어, 다음 역직렬화 코드를 실행하면 `Numbers1`에는 값 1, 2, 3이, `Numbers2`에는 값 4, 5, 6이 포함됩니다.

C#

```
A? a = JsonSerializer.Deserialize<A>("""{"Numbers1": [4,5,6], "Numbers2":
[4,5,6]}""");
```

채우기 동작

.NET 8부터 역직렬화 동작을 변경하여 속성 및 필드를 바꾸는 대신 수정(*채우기*)할 수 있습니다.

- 컬렉션 형식 속성의 경우 개체를 지우지 않고 다시 사용합니다. 컬렉션이 요소로 미리 채워지면 JSON의 값과 함께 최종 역직렬화된 결과에 표시됩니다. 예를 들어 컬렉션 속성 예제

를 참조하세요.

- 속성이 있는 개체의 경우 변경 가능한 속성이 JSON 값으로 업데이트되지만 개체 참조 자체는 변경되지 않습니다.
- 구조체 형식 속성의 경우 변경 가능한 속성에 대해 기존 값이 유지되고 JSON의 새 값이 추가된다는 것이 효과적입니다. 그러나 참조 속성과 달리 개체 자체는 값 형식이므로 다시 사용되지 않습니다. 대신 구조체의 복사본을 수정한 다음 속성에 다시 할당합니다. 예제는 구조체 속성 예제를 참조하세요.

  구조체 속성에는 setter가 있어야 합니다. 그렇지 않으면 런타임에 InvalidOperationException throw됩니다.

> ⓘ **참고 항목**
>
> 채우기 동작은 현재 매개 변수가 있는 생성자가 있는 형식에 대해 작동하지 않습니다. 자세한 내용은 dotnet/runtime 문제 92877 을 참조하세요.

읽기 전용 속성

변경 가능한 참조 속성을 채우기 위해 속성 참조 인스턴스가 *대체*되지 않으므로 속성에 setter가 필요하지 않습니다. 이 동작은 역직렬화가 *읽기 전용* 속성을 채울 수도 있음을 의미합니다.

> ⓘ **참고 항목**
>
> 인스턴스가 수정된 복사본으로 대체되므로 구조체 속성에는 여전히 setter가 필요합니다.

컬렉션 속성 예제

동일한 클래스 `A`를 바꾸기 동작 예제의 일부로 고려하지만, 이번에는 속성을 바꾸기보다는 *속성을 채우기*로 선호된 설정으로 주석을 달았습니다.

C#

```
[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
class A
{
    public List<int> Numbers1 { get; } = [1, 2, 3];
    public List<int> Numbers2 { get; set; } = [1, 2, 3];
}
```

다음 역직렬화 코드를 실행하면, `Numbers1`와 `Numbers2` 모두 1, 2, 3, 4, 5 및 6의 값을 포함하게 됩니다.

C#

```
A? a = JsonSerializer.Deserialize<A>("""{"Numbers1": [4,5,6], "Numbers2":
[4,5,6]}""");
```

구조체 속성 예제

다음 클래스에는 구조체 속성 `S1`이 있으며, 그 역직렬화 동작은 Populate으로 설정되어 있습니다. 이 코드를 `c.S1.Value1` 실행한 후 값이 10(생성자에서)이고 `c.S1.Value2` 값이 5(JSON에서)인 경우

C#

```
C? c = JsonSerializer.Deserialize<C>("""{"S1": {"Value2": 5}}""");

class C
{
    public C()
    {
        _s1 = new S
        {
            Value1 = 10
        };
    }

    private S _s1;

    [JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
    public S S1
    {
        get { return _s1; }
        set { _s1 = value; }
    }
}

struct S
{
    public int Value1 { get; set; }
    public int Value2 { get; set; }
}
```

대신 Replace의 기본 동작을 사용한 경우, `c.S1.Value1`는 역직렬화 후에 기본값으로 0이 됩니다. 생성자 `C()`가 호출되어 `c.S1.Value1`이 10으로 설정되지만, 뒤에 S1 값이 새 인스턴스로 대체되기 때문입니다. (`c.S1.Value2` JSON이 기본값을 대체하므로 여전히 5입니다.)

지정하는 방법

바꾸기 또는 *채우기*에 대한 기본 설정을 지정하는 방법에는 여러 가지가 있습니다.

- JsonObjectCreationHandlingAttribute 특성을 사용하여 형식 또는 속성 수준에서 주석을 달 수 있습니다. 형식 수준에서 특성을 설정하고 해당 Handling 속성을 Populate으로 설정하면, 채우기가 가능한 속성에만 동작이 적용됩니다(예: 값 형식에는 setter가 있어야 합니다).

  형식 전체 기본 설정을 Populate원하는 경우 해당 동작에서 하나 이상의 속성을 제외하려면 형식 수준에서 특성을 추가하고 속성 수준에서 다시 추가하여 상속된 동작을 재정의할 수 있습니다. 해당 패턴은 다음 코드에 나와 있습니다.

  C#

  ```
  // Type-level preference is Populate.
  [JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
  class B
  {
      // For this property only, use Replace behavior.
      [JsonObjectCreationHandling(JsonObjectCreationHandling.Replace)]
      public List<int> Numbers1 { get; } = [1, 2, 3];
      public List<int> Numbers2 { get; set; } = [1, 2, 3];
  }
  ```

- 전역 기본 설정을 지정하려면 JsonSerializerOptions.PreferredObjectCreationHandling (또는 원본 생성의 경우 JsonSourceGenerationOptionsAttribute.PreferredObjectCreationHandling)을 설정합니다.

  C#

  ```
  var options = new JsonSerializerOptions
  {
      PreferredObjectCreationHandling = JsonObjectCreationHandling.Populate
  };
  ```

Last updated on 2025. 12. 05.

Newtonsoft.Json에서 System.Text.Json(으)로 마이그레이션

2025. 10. 03.

이 문서에서는 Newtonsoft.Json에서 System.Text.Json로 마이그레이션하는 방법을 보여줍니다.

`System.Text.Json` 네임스페이스는 JSON(JavaScript Object Notation)에서 직렬화 및 역직렬화하는 기능을 제공합니다. `System.Text.Json` 라이브러리는 .NET Core 3.1 이상 버전의 런타임에 포함되어 있습니다. 다른 대상 프레임워크의 경우 System.Text.Json NuGet 패키지를 설치합니다. 패키지는 다음을 지원합니다.

- .NET Standard 2.0 이상 버전
- .NET Framework 4.6.2 이전 버전
- .NET Core 2.0, 2.1, 2.2

> **팁**
>
> AI 지원을 사용하여 **Newtonsoft.Json에서 마이그레이션**할 수 있습니다.

`System.Text.Json`은 성능, 보안 및 표준 규정 준수에 중점을 둡니다. 기본 동작에서 몇 가지 큰 차이가 있으며 `Newtonsoft.Json`과의 기능 패리티를 목표로 하지 않습니다. 일부 시나리오에서는 현재 `System.Text.Json`에 기본 제공 기능이 없지만, 권장 해결 방법이 있습니다. 그 외의 시나리오에서는 해결 방법이 실용적이지 않습니다.

`System.Text.Json` 팀은 가장 자주 요청된 기능을 추가하는 데 투자하고 있습니다. 애플리케이션에서 사용하는 기능이 없는 경우 dotnet/runtime GitHub 리포지토리에서 이슈를 제출하여 해당 시나리오에 대한 지원을 추가할 수 있는지 알아보세요.

이 문서의 대부분은 JsonSerializer API 사용 방법에 대한 내용이지만, JsonDocument(DOM(문서 개체 모델)을 나타냄), Utf8JsonReader 및 Utf8JsonWriter 형식을 사용하는 방법에 대한 지침도 포함되어 있습니다.

Visual Basic에서는 Utf8JsonReader를 사용할 수 없습니다. 즉, 사용자 지정 변환기를 작성할 수 없습니다. 여기에 제시된 대부분의 해결 방법은 사용자 지정 변환기를 작성해야 합니다. C#에서 사용자 지정 변환기를 작성하고 Visual Basic 프로젝트에 등록할 수 있습니다. 자세한 내용은 Visual Basic 지원을 참조하세요.

차이점 표

다음 표에는 `Newtonsoft.Json` 기능과 그에 상응하는 `System.Text.Json` 기능이 나열되어 있습니다. 상응하는 기능은 다음 범주로 분류됩니다.

- ✔ 기본 제공 기능에서 지원됩니다. `System.Text.Json`에서 유사한 동작을 가져오려면 특성 또는 글로벌 옵션을 사용해야 할 수도 있습니다.
- ⚠ 지원되지 않지만 해결이 가능합니다. 해결 방법은 사용자 지정 변환기이며, 사용자 지정 변환기는 `Newtonsoft.Json` 기능과의 완전한 패리티를 제공하지 않을 수 있습니다. 그 중 일부는 샘플 코드가 예제로 제공됩니다. 이러한 `Newtonsoft.Json` 기능을 사용하는 경우 마이그레이션을 수행하려면 .NET 개체 모델 또는 기타 코드 변경 내용을 수정해야 합니다.
- ✘ 지원되지 않으며 해결 방법이 실용적이지 않거나 가능하지 않습니다. 이러한 `Newtonsoft.Json` 기능을 사용하는 경우 중요한 변경 없이는 마이그레이션을 수행할 수 없습니다.

⛶ **테이블 확장**

Newtonsoft.Json 기능	**System.Text.Json 해당 항목**
기본적으로 대/소문자를 구분하지 않는 역직렬화	✔ PropertyNameCaseInsensitive 글로벌 설정
카멜식 대/소문자 속성 이름	✔ PropertyNamingPolicy 글로벌 설정
스네이크 케이스 속성 이름	✔ 스네이크 표기법 이름 지정 정책
최소 문자 이스케이프	✔ 엄격한 문자 이스케이프, 구성 가능
`NullValueHandling.Ignore` 글로벌 설정	✔ DefaultIgnoreCondition 전역 옵션
주석 허용	✔ ReadCommentHandling 글로벌 설정
후행 쉼표 허용	✔ AllowTrailingCommas 글로벌 설정
사용자 지정 변환기 등록	✔ 우선 순위가 다름
기본 최대 깊이는 64, 구성 가능	✔ 기본 최대 깊이는 64, 구성 가능
`PreserveReferencesHandling` 글로벌 설정	✔ ReferenceHandling 전역 설정
따옴표 안의 숫자를 직렬화하거나 역직렬화	✔ NumberHandling 전역 설정, [JsonNumberHandling] 특성
변경할 수 없는 클래스 및 구조체로 역직렬화	✔ JsonConstructor, C# 9 레코드
필드에 대한 지원	✔ IncludeFields 전역 설정, [JsonInclude] 특성
`DefaultValueHandling` 글로벌 설정	✔ DefaultIgnoreCondition 전역 설정
`NullValueHandling`에서의 `[JsonProperty]` 설정	✔ JsonIgnore 특성
`DefaultValueHandling`에서의 `[JsonProperty]` 설정	✔ JsonIgnore 특성

Newtonsoft.Json 기능	System.Text.Json 해당 항목
문자열이 아닌 키로 `Dictionary` 역직렬화	✔ 지원됨
public이 아닌 속성 setter 및 getter 지원	✔ JsonInclude 특성
`[JsonConstructor]` 특성	✔ JsonConstructor 특성
`ReferenceLoopHandling` 글로벌 설정	✔ ReferenceHandling 전역 설정
콜백	✔ 콜백
NaN, Infinity, -Infinity	✔ 지원됨
`Required` 특성에 대한 `[JsonProperty]` 설정	✔ [JsonRequired] 특성 및 C# 필수 한정자
속성을 무시하는 `DefaultContractResolver`	✔ DefaultJsonTypeInfoResolver 클래스
다형 직렬화	✔ [JsonDerivedType] 특성
다형 역직렬화	✔ [JsonDerivedType] 특성에 대한 형식 판별자
문자열 열거형 값 역직렬화	✔ 문자열 열거형 값 역직렬화
`MissingMemberHandling` 글로벌 설정	✔ 누락된 멤버 처리
setter 없이 속성 채우기	✔ setter 없이 속성 채우기
`ObjectCreationHandling` 글로벌 설정	✔ 속성을 바꾸지 않고 재사용
광범위한 형식 지원	⚠ 일부 형식은 사용자 지정 변환기 필요
유추 형식을 `object` 속성으로 역직렬화	⚠ 지원되지 않음, 해결 가능, 샘플
JSON `null` 리터럴을 null을 허용하지 않는 값 형식으로 역직렬화	⚠ 지원되지 않음, 해결 가능, 샘플
`DateTimeZoneHandling`, `DateFormatString` 설정	⚠ 지원되지 않음, 해결 가능, 샘플
`JsonConvert.PopulateObject` 메서드	⚠ 지원되지 않음, 해결 가능
`System.Runtime.Serialization` 특성 지원	⚠ 지원되지 않음, 해결 가능, 샘플
`JsonObjectAttribute`	⚠ 지원되지 않음, 해결 가능
따옴표 없는 속성 이름 허용	✘ 디자인에서 지원되지 않음
문자열 값 주변에 작은따옴표 허용	✘ 디자인에서 지원되지 않음
문자열 속성에 문자열이 아닌 JSON 값 허용	✘ 디자인에서 지원되지 않음
`TypeNameHandling.All` 글로벌 설정	✘ 디자인에서 지원되지 않음

Newtonsoft.Json 기능	System.Text.Json 해당 항목
`JsonPath` 쿼리 지원	❌ 지원되지 않음
구성 가능한 제한	❌ 지원되지 않음

이 목록은 `Newtonsoft.Json` 기능의 전체 목록이 아닙니다. 이 목록에는 GitHub 이슈 또는 StackOverflow 게시물에 요청된 여러 시나리오가 포함되어 있습니다. 여기에 나열된 시나리오 중에서 현재 샘플 코드가 없는 시나리오에 대한 해결 방법을 구현하셨으며 그 방법을 공유하려는 분들은 이 페이지 하단의 **피드백** 섹션에서 **이 페이지**를 선택하세요. 그러면 이 설명서의 GitHub 리포지토리에 이슈가 작성되고 이 페이지의 **피드백** 섹션에도 이슈가 나열됩니다.

기본 동작의 차이점

System.Text.Json은 기본적으로 엄격하며, 호출자를 대신하여 추측하거나 해석하는 것을 금지하고 결정적 동작을 강조합니다. 이 라이브러리는 성능과 보안을 위해 의도적으로 이렇게 설계되었습니다. `Newtonsoft.Json`은 기본적으로 유연합니다. 이러한 기본적인 디자인의 차이로 인해 기본 동작에서 다음과 같은 여러 가지 차이가 있습니다.

대/소문자를 구분하지 않는 역직렬화

역직렬화를 수행하는 동안 `Newtonsoft.Json`은 기본적으로 대/소문자를 구분하지 않는 속성 이름을 매칭합니다. System.Text.Json은 기본적으로 대/소문자를 구분하며, 이 방법은 매칭을 정확히 수행하기 때문에 보다 나은 성능을 제공합니다. 대/소문자를 구분하지 않는 매칭 방법에 대한 자세한 내용은 대/소문자를 구분하지 않는 속성 매칭을 참조하세요.

ASP.NET Core를 사용하여 간접적으로 `System.Text.Json`을 사용하는 경우 `Newtonsoft.Json`과 같은 동작을 얻기 위해 아무것도 할 필요가 없습니다. ASP.NET Core는 `System.Text.Json`을(를) 사용할 때 카멜식 대/소문자 구분 속성 이름 및 대/소문자를 구분하지 않는 매칭에 대한 설정을 지정합니다.

ASP.NET Core에서는 기본적으로 따옴표 붙은 숫자의 역직렬화도 가능합니다.

최소 문자 이스케이프

직렬화할 때 `Newtonsoft.Json`은 문자를 이스케이프하지 않고 허용하는 것에 대해 비교적 관대합니다. 즉, 문자를 `\uxxxx`로 바꾸지 않으며, 여기서 `xxxx`는 문자의 코드 포인트입니다. 문자를 이스케이프할 때는 문자 앞에 `\`를 내보냅니다. 예를 들어 `"`는 `\"`가 됩니다. System.Text.Json은 XSS(교차 사이트 스크립팅) 또는 정보 공개 공격에 대한 심층 방어를 제공하기 위해 기본적으로 더 많은 문자를 이스케이프하며, 그러기 위해 6문자 시퀀스를 사용합니다. `System.Text.Json`은 기본적으로 ASCII가 아닌 모든 문자를 이스케이프하므로, `StringEscapeHandling.EscapeNonAscii`

에서 `Newtonsoft.Json`를 사용 중이라면 아무것도 할 필요가 없습니다. 또한 `System.Text.Json`은 기본적으로 HTML 구분 문자를 이스케이프합니다. 기본 `System.Text.Json` 동작을 재정의하는 방법에 대한 자세한 내용은 문자 인코딩 사용자 지정을 참조하세요.

주석

역직렬화할 때 `Newtonsoft.Json`은 기본적으로 JSON의 주석을 무시합니다. System.Text.Json 사양에 주석이 포함되지 않기 때문에 은 기본적으로 주석에 대해 예외를 throw합니다. 주석을 허용하는 방법에 대한 자세한 내용은 주석과 후행 쉼표 허용을 참조하세요.

후행 쉼표

역직렬화할 때 `Newtonsoft.Json`은 기본적으로 후행 쉼표를 무시합니다. 또한 여러 개의 후행 쉼표를 무시합니다(예: `[{"Color":"Red"},{"Color":"Green"},,]`). System.Text.Json 사양에서 후행 쉼표를 허용하지 않기 때문에 은 기본적으로 후행 쉼표에 대해 예외를 throw합니다. `System.Text.Json`에서 후행 쉼표를 허용하게 만드는 방법은 주석과 후행 쉼표 허용을 참조하세요. 후행 쉼표를 여러 개 허용하는 방법은 없습니다.

변환기 등록 우선 순위

사용자 지정 변환기에 대한 `Newtonsoft.Json` 등록 우선 순위는 다음과 같습니다.

- 속성의 특성
- 형식의 특성
- 변환기 컬렉션

이 순서는 형식 수준에서 특성을 적용하여 등록된 변환기가 `Converters` 컬렉션의 사용자 지정 변환기를 재정의한다는 뜻입니다. 두 등록 모두 속성 수준에서 특성에 의해 재정의됩니다.

사용자 지정 변환기에 대한 System.Text.Json 등록 우선 순위는 다음과 같이 다릅니다.

- 속성의 특성
- Converters 컬렉션
- 형식의 특성

여기서 `Converters` 컬렉션의 사용자 지정 변환기가 형식 수준에서 특성을 재정의한다는 차이가 있습니다. 이 우선 순위 순서의 숨은 의도는 런타임 변경이 디자인 타임 선택 항목을 재정의하도록 하는 것입니다. 우선 순위를 변경할 수 있는 방법은 없습니다.

사용자 지정 변환기 등록에 대한 자세한 내용은 사용자 지정 변환기 등록을 참조하세요.

최대 깊이

`Newtonsoft.Json`의 최신 버전은 기본적으로 최대 깊이 제한이 64입니다. System.Text.Json도 기본 제한은 64이며, JsonSerializerOptions.MaxDepth를 설정하여 구성할 수 있습니다.

ASP.NET Core를 사용하여 간접적으로 `System.Text.Json`를 사용하는 경우 기본 최대 깊이 제한은 32입니다. 기본값은 모델 바인딩과 동일하며 JsonOptions 클래스 에 설정됩니다.

JSON 문자열(속성 이름 및 문자열 값)

역직렬화할 때 `Newtonsoft.Json`은 큰따옴표/작은따옴표로 묶은 속성 이름 또는 따옴표 없는 속성 이름을 허용합니다. 큰따옴표 또는 작은따옴표로 묶은 문자열 값을 허용합니다. 예를 들어 `Newtonsoft.Json`은 다음 JSON을 허용합니다.

JSON

```
{
  "name1": "value",
  'name2': "value",
  name3: 'value'
}
```

`System.Text.Json`은 큰따옴표로 묶은 속성 이름과 문자열 값만 허용합니다. RFC 8259 사양에서 이 형식을 요구하며 유일하게 유효한 JSON으로 간주되는 형식이기 때문입니다.

작은따옴표로 묶은 값은 다음 메시지와 함께 JsonException을 반환합니다.

출력

```
''' is an invalid start of a value.
```

문자열 속성에 문자열이 아닌 값 허용

`Newtonsoft.Json`은 형식 문자열의 속성으로 역직렬화할 때 숫자 또는 리터럴 `true` 및 `false`처럼 문자열이 아닌 값을 허용합니다. 아래는 `Newtonsoft.Json`이 성공적으로 다음 클래스로 역직렬화하는 JSON 예제입니다.

JSON

```
{
  "String1": 1,
  "String2": true,
```

```
  "String3": false
}
```

C#

```
public class ExampleClass
{
    public string String1 { get; set; }
    public string String2 { get; set; }
    public string String3 { get; set; }
}
```

`System.Text.Json`은 문자열이 아닌 값을 문자열 속성으로 역직렬화하지 않습니다. 문자열 필드에 대해 문자열이 아닌 값을 받으면 다음 메시지와 함께 JsonException이 반환됩니다.

출력

```
The JSON value could not be converted to System.String.
```

JsonSerializer를 사용하는 시나리오

다음 시나리오는 기본 제공 기능에서 지원되지 않지만 해결이 가능합니다. 해결 방법은 사용자 지정 변환기이며, 사용자 지정 변환기는 `Newtonsoft.Json` 기능과의 완전한 패리티를 제공하지 않을 수 있습니다. 그 중 일부는 샘플 코드가 예제로 제공됩니다. 이러한 `Newtonsoft.Json` 기능을 사용하는 경우 마이그레이션을 수행하려면 .NET 개체 모델 또는 기타 코드 변경 내용을 수정해야 합니다.

다음 몇몇 시나리오는 해결 방법이 실용적이지 않거나 가능하지 않습니다. 이러한 `Newtonsoft.Json` 기능을 사용하는 경우 중요한 변경 없이는 마이그레이션을 수행할 수 없습니다.

따옴표 안에 숫자를 허용하거나 씁니다.

`Newtonsoft.Json`은 JSON 문자열로 표시되는(따옴표로 묶은) 숫자를 직렬화 또는 역직렬화할 수 있습니다. 예를 들어 `{"DegreesCelsius":"23"}` 대신 `{"DegreesCelsius":23}`을 허용할 수 있습니다. System.Text.Json에서 이 동작을 사용하도록 설정하려면 JsonSerializerOptions.NumberHandling을 WriteAsString 또는 AllowReadingFromString으로 설정하거나 JsonNumberHandling 특성을 사용합니다.

ASP.NET Core를 사용하여 간접적으로 `System.Text.Json`을 사용하는 경우 `Newtonsoft.Json`과 같은 동작을 얻기 위해 아무것도 할 필요가 없습니다. ASP.NET Core는 을 사용할 때 `System.Text.Json`을 지정하고, 웹 기본값은 따옴표 붙은 숫자를 허용합니다.

자세한 내용은 따옴표 안의 숫자 허용 또는 쓰기를 참조하세요.

역직렬화할 때 사용할 생성자 지정

`Newtonsoft.Json` `[JsonConstructor]` 특성을 사용하면 POCO로 역직렬화할 때 호출할 생성자를 지정할 수 있습니다.

`System.Text.Json`에는 JsonConstructor 특성도 있습니다. 자세한 내용은 변경 불가능한 형식 및 레코드를 참조하세요.

조건부로 속성 무시

`Newtonsoft.Json`은 직렬화 또는 역직렬화에서 속성을 조건부로 무시하는 여러 가지 방법이 있습니다.

- `DefaultContractResolver`를 사용하면 임의의 조건에 따라 포함하거나 무시할 속성을 선택할 수 있습니다.
- `NullValueHandling`의 `DefaultValueHandling` 및 `JsonSerializerSettings` 설정을 사용하면 모든 Null 값 또는 기본값 속성을 무시하도록 지정할 수 있습니다.
- `NullValueHandling` 특성의 `DefaultValueHandling` 및 `[JsonProperty]` 설정을 사용하면 Null 또는 기본값으로 설정된 경우에 무시할 개별 속성을 지정할 수 있습니다.

System.Text.Json은 직렬화하는 동안 다음과 같은 방법으로 속성이나 필드를 무시할 수 있습니다.

- 속성의 [JsonIgnore] 특성은 직렬화하는 동안 JSON에서 속성을 생략하게 합니다.
- IgnoreReadOnlyProperties 전역 옵션을 사용하면 모든 읽기 전용 속성을 무시할 수 있습니다.
- 필드를 포함하는 경우 JsonSerializerOptions.IgnoreReadOnlyFields 전역 옵션을 통해 모든 읽기 전용 필드를 무시할 수 있습니다.
- `DefaultIgnoreCondition` 전역 옵션을 사용하면 기본값이 있는 모든 값 형식 속성을 무시하거나 null 값이 있는 모든 참조 형식 속성을 무시할 수 있습니다.

또한 .NET 7 이상 버전에서는 임의 조건에 따라 속성을 무시하도록 JSON 계약을 사용자 지정할 수 있습니다. 자세한 내용은 사용자 지정 계약을 참조하세요.

public 및 비-public 필드

`Newtonsoft.Json`은 속성뿐 아니라 필드까지 직렬화 및 역직렬화할 수 있습니다.

System.Text.Json에서 JsonSerializerOptions.IncludeFields 전역 설정 또는 JsonInclude 특성을 사용하여 직렬화 또는 역직렬화할 때 public 필드를 포함합니다. 예제는 필드 포함을 참조하세요.

개체 참조 유지 및 루프 처리

기본적으로 `Newtonsoft.Json`은 값으로 직렬화합니다. 예를 들어 개체에 두 개의 속성이 있고 두 속성은 동일한 `Person` 개체에 대한 참조를 포함하는 경우 해당 `Person` 개체의 속성 값이 JSON에서 중복됩니다.

`Newtonsoft.Json`에는 참조로 직렬화할 수 있는 `PreserveReferencesHandling`에 대한 `JsonSerializerSettings` 설정이 있습니다.

- 첫 번째 `Person` 개체에 대해 만든 JSON에 식별자 메타데이터가 추가됩니다.
- 두 번째 `Person` 개체에 대해 만든 JSON에는 속성 값 대신 해당 식별자에 대한 참조가 포함됩니다.

`Newtonsoft.Json`에는 예외를 throw하는 대신 순환 참조를 무시할 수 있는 `ReferenceLoopHandling` 설정도 있습니다.

System.Text.Json에서 참조를 보존하고 순환 참조를 처리하려면 JsonSerializerOptions.ReferenceHandler을 Preserve로 설정합니다. `ReferenceHandler.Preserve` 설정은 `PreserveReferencesHandling`의 = `PreserveReferencesHandling.All` `Newtonsoft.Json`과 동일합니다.

`ReferenceHandler.IgnoreCycles` 옵션에는 Newtonsoft.Json `ReferenceLoopHandling.Ignore`와 유사한 동작이 있습니다. 한 가지 차이점은 System.Text.Json 구현이 개체 참조를 무시하는 대신 참조 루프를 `null` JSON 토큰으로 대체한다는 것입니다. 자세한 내용은 순환 참조 무시를 참조하세요.

Newtonsoft.Json ReferenceResolver와 마찬가지로 System.Text.Json.Serialization.ReferenceResolver 클래스는 직렬화 및 역직렬화 시 참조를 보존하는 동작을 정의합니다. 파생 클래스를 만들어 사용자 지정 동작을 지정합니다. 예제는 GuidReferenceResolver를 참조하세요.

일부 관련 `Newtonsoft.Json` 기능은 지원되지 않습니다.

- JsonPropertyAttribute.IsReference
- JsonPropertyAttribute.ReferenceLoopHandling

자세한 내용은 참조 보존 및 순환 참조 처리를 참조하세요.

키가 문자열이 아닌 사전

`Newtonsoft.Json` 및 `System.Text.Json`은 모두 `Dictionary<TKey, TValue>` 형식의 컬렉션을 지원합니다. 지원되는 키 형식에 대한 자세한 내용은 지원되는 키 형식참조하세요.

⊗ 주의

`Dictionary<TKey, TValue>`가 `TKey` 이외의 형식으로 지정되는 `string`로 역직렬화하면 소비 애플리케이션에 보안 취약성이 발생할 수 있습니다. 자세한 내용은 dotnet/runtime#4761 을 참조하세요.

기본적으로 지원되지 않는 형식

System.Text.Json은 기본적으로 다음 형식을 지원하지 않습니다.

- DataTable 및 관련 형식(자세한 내용은 지원되는 형식참조)
- ExpandoObject
- TimeZoneInfo
- BigInteger
- DBNull
- Type
- ValueTuple 및 관련 제네릭 형식

기본적으로 지원되지 않는 형식에 대한 사용자 지정 변환기를 구현할 수 있습니다.

다형 직렬화

`Newtonsoft.Json`은 다형 직렬화를 자동으로 수행합니다. .NET 7부터 System.Text.Json은 JsonDerivedTypeAttribute 특성을 통한 다형 직렬화를 지원합니다. 자세한 내용은 파생 클래스의 직렬화 속성을 참조하세요.

다형 역직렬화

`Newtonsoft.Json`에는 직렬화하는 동안 JSON에 형식 이름 메타데이터를 추가하는 `TypeNameHandling` 설정이 있습니다. 이 설정은 역직렬화하는 동안 메타데이터를 사용하여 다형 역직렬화를 수행합니다. .NET 7부터 System.Text.Json은 형식 판별자 정보를 사용하여 다형 역직렬화를 수행합니다. 이 메타데이터는 JSON으로 내보낸 다음 역직렬화하는 동안 기본 형식으로 역직렬화할지 또는 파생 형식으로 역직렬화할지 여부를 결정하는 데 사용됩니다. 자세한 내용은 파생 클래스의 직렬화 속성을 참조하세요.

이전 .NET 버전에서 다형 역직렬화를 지원하려면 사용자 지정 변환기를 작성하는 방법의 예제와 같은 변환기를 만듭니다.

문자열 열거형 값 역직렬화

기본적으로 System.Text.Json 문자열 열거형 값은 역직렬화하는 것을 지원하지 않지만, `Newtonsoft.Json`는 그렇지 않습니다. 예를 들어 다음 코드는 JsonException을(를) 발생시킵니다.

C#

```
string json = "{ \"Text\": \"Hello\", \"Enum\": \"Two\" }";
var _ = JsonSerializer.Deserialize<MyObj>(json); // Throws exception.

class MyObj
{
    public string Text { get; set; } = "";
    public MyEnum Enum { get; set; }
}

enum MyEnum
{
    One,
    Two,
    Three
}
```

그러나 JsonStringEnumConverter 변환기를 사용하여 문자열 열거형 값의 역직렬화를 사용하도록 설정할 수 있습니다. 자세한 내용은 열거형을 문자열로 참조하세요.

개체 속성 역직렬화

`Newtonsoft.Json`은 Object로 역직렬화할 때 다음을 수행합니다.

- JSON 페이로드의 기본 값 형식(`null` 제외)을 유추하고 저장된 `string`, `long`, `double`, `boolean` 또는 `DateTime`을 boxed 개체로 반환합니다. *기본 값*은 JSON 숫자, 문자열, `true`, `false`, `null` 등의 단일 JSON 값입니다.
- JSON 페이로드의 복합 값에 대한 `JObject` 또는 `JArray`를 반환합니다. *복합 값*은 중괄호(`{}`) 안에 있는 JSON 키-값 쌍 컬렉션 또는 대괄호(`[]`) 안에 있는 값 목록입니다. 중괄호 또는 대괄호 안에 있는 속성과 값이 추가 속성 또는 값을 가질 수 있습니다.
- 페이로드에 `null` JSON 리터럴이 있으면 null 참조를 반환합니다.

System.Text.Json은 `JsonElement`로 역직렬화할 때마다 기본 값과 복합 값 모두에 대한 boxed Object를 저장하며, 다음은 그 예입니다.

- `object` 속성
- `object` 사전 값
- `object` 배열 값
- 루트 `object`

그러나 페이로드에 `System.Text.Json` JSON 리터럴이 있으면 `null`은 `Newtonsoft.Json`을 `null`과 같은 방법으로 처리하고 null 참조를 반환합니다.

`object` 속성에 대한 형식 유추를 구현하려면 사용자 지정 변환기를 작성하는 방법의 예제와 같은 변환기를 만듭니다.

null을 허용하지 않는 형식으로 Null 역직렬화

`Newtonsoft.Json`은 는 다음과 같은 시나리오에서 예외를 throw하지 않습니다.

- `NullValueHandling`이 `Ignore`로 설정된 경우
- 역직렬화를 수행하는 동안 JSON은 null을 허용하지 않는 값 형식에 대해 Null 값을 포함합니다.

동일한 시나리오에서 System.Text.Json은 예외를 throw합니다. (`System.Text.Json`에서 해당하는 null 처리 설정은 JsonSerializerOptions.IgnoreNullValues = `true`입니다.)

대상 형식을 소유하고 있는 경우 가장 좋은 해결 방법은 문제가 되는 속성을 null 허용으로 만드는 것입니다(예를 들어 `int`를 `int?`로 변경).

또 다른 해결 방법은 `DateTimeOffset` 형식의 Null 값을 처리하는 다음 예제처럼 형식에 대한 변환기를 만드는 것입니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DateTimeOffsetNullHandlingConverter :
JsonConverter<DateTimeOffset>
    {
        public override DateTimeOffset Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            reader.TokenType == JsonTokenType.Null
                ? default
                : reader.GetDateTimeOffset();

        public override void Write(
            Utf8JsonWriter writer,
            DateTimeOffset dateTimeValue,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(dateTimeValue);
```

```
    }
}
```

속성에 대한 특성을 사용하거나 컬렉션에 Converters 이 사용자 지정 변환기를 등록합니다.

참고: 위의 변환기는 기본값을 지정하는 POCO를 이 `Newtonsoft.Json` Null 값을 처리합니다. 예를 들어 다음 코드가 대상 개체를 보여준다고 가정하겠습니다.

C#

```
public class WeatherForecastWithDefault
{
    public WeatherForecastWithDefault()
    {
        Date = DateTimeOffset.Parse("2001-01-01");
        Summary = "No summary";
    }
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

그리고 앞의 변환기를 사용하여 다음 JSON을 역직렬화한다고 가정합니다.

JSON

```
{
  "Date": null,
  "TemperatureCelsius": 25,
  "Summary": null
}
```

역직렬화 후 `Date` 속성에 1/1/0001(`default(DateTimeOffset)`)이 있습니다. 즉, 생성자에서 설정한 값이 덮어쓰기 되었습니다. POCO 및 JSON이 동일할 때 `Newtonsoft.Json` 역직렬화 시 `Date` 속성에 1/1/2001이 남습니다.

변경할 수 없는 클래스 및 구조체로 역직렬화

`Newtonsoft.Json`은 매개 변수가 있는 생성자를 사용할 수 있기 때문에 변경 불가능한 클래스 및 구조체로 역직렬화할 수 있습니다.

System.Text.Json에서 JsonConstructor 특성을 사용하여 매개 변수가 있는 생성자의 사용을 지정합니다. C# 9의 레코드도 변경할 수 없으며, 역직렬화 대상으로 지원됩니다. 자세한 내용은 변경 불가능한 형식 및 레코드를 참조하세요.

필수 속성

`Newtonsoft.Json`에서 `Required` 특성에 대한 `[JsonProperty]`를 설정하여 속성을 필수로 지정합니다. 필수로 지정된 속성에 대해 JSON에서 값을 받지 못하면 `Newtonsoft.Json`이 예외를 throw 합니다.

.NET 7부터 필수 속성에서 C# `required` 한정자 또는 JsonRequiredAttribute 특성을 사용할 수 있습니다. System.Text.Json은 JSON 페이로드에 표시된 속성에 대한 값이 포함되어 있지 않으면 예외를 throw합니다. 자세한 내용은 필수 속성을 참조하세요.

날짜 형식 지정

`Newtonsoft.Json`은 `DateTime` 및 `DateTimeOffset` 형식의 속성 직렬화 및 역직렬화 방식을 여러 가지 방법으로 제어할 수 있습니다.

- `DateTimeZoneHandling` 설정을 사용하여 모든 `DateTime` 값을 UTC 날짜로 직렬화할 수 있습니다.
- `DateFormatString` 설정 및 `DateTime` 변환기를 사용하여 날짜 문자열의 형식을 사용자 지정할 수 있습니다.

System.Text.Json은 RFC 3339 프로필을 포함하여 ISO 8601-1:2019를 지원합니다. 이 형식은 널리 채택되었으며, 명확하고, 정확하게 왕복합니다. 다른 형식을 사용하려면 사용자 지정 변환기를 만듭니다. 예를 들어 다음 변환기는 표준 시간대 오프셋(`/Date(1590863400000-0700)/` 또는 `/Date(1590863400000)/`와 같은 값)이 포함되거나 포함되지 않은 Unix epoch 형식을 사용하는 JSON을 직렬화 및 역직렬화합니다.

C#

```
sealed class UnixEpochDateTimeOffsetConverter :
System.Text.Json.Serialization.JsonConverter<DateTimeOffset>
{
    static readonly DateTimeOffset s_epoch = new(1970, 1, 1, 0, 0, 0,
TimeSpan.Zero);
    static readonly Regex s_regex = new(
        "^/Date\\(([+-]*\\d+)([+-])(\\d{2})(\\d{2})\\)/$",
        RegexOptions.CultureInvariant);

    public override DateTimeOffset Read(
        ref Utf8JsonReader reader,
        Type typeToConvert,
        JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
```

```
                !match.Success
                || !long.TryParse(match.Groups[1].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out long unixTime)
                || !int.TryParse(match.Groups[3].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out int hours)
                || !int.TryParse(match.Groups[4].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out int minutes))
        {
            throw new System.Text.Json.JsonException();
        }

        int sign = match.Groups[2].Value[0] == '+' ? 1 : -1;
        TimeSpan utcOffset = new(hours * sign, minutes * sign, 0);

        return s_epoch.AddMilliseconds(unixTime).ToOffset(utcOffset);
    }

    public override void Write(
        Utf8JsonWriter writer,
        DateTimeOffset value,
        JsonSerializerOptions options)
    {
        long unixTime = value.ToUnixTimeMilliseconds();

        TimeSpan utcOffset = value.Offset;

        string formatted = string.Create(
            CultureInfo.InvariantCulture,
            $"/Date({unixTime}{(utcOffset >= TimeSpan.Zero ? "+" : "-")}
{utcOffset:hhmm})/");

        writer.WriteStringValue(formatted);
    }
}
```

C#

```
sealed class UnixEpochDateTimeConverter :
System.Text.Json.Serialization.JsonConverter<DateTime>
{
    static readonly DateTime s_epoch = new(1970, 1, 1, 0, 0, 0);
    static readonly Regex s_regex = new(
        "^/Date\\(([+-]*\\d+)\\)/$",
        RegexOptions.CultureInvariant);

    public override DateTime Read(
        ref Utf8JsonReader reader,
        Type typeToConvert,
        JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);
```

```
        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value, NumberStyles.Integer,
CultureInfo.InvariantCulture, out long unixTime))
        {
            throw new System.Text.Json.JsonException();
        }

        return s_epoch.AddMilliseconds(unixTime);
    }

    public override void Write(
        Utf8JsonWriter writer,
        DateTime value,
        JsonSerializerOptions options)
    {
        long unixTime = (value - s_epoch).Ticks / TimeSpan.TicksPerMillisecond;

        string formatted = string.Create(CultureInfo.InvariantCulture,
$"/Date({unixTime})/");
        writer.WriteStringValue(formatted);
    }
}
```

자세한 내용은 System.Text.Json에서 DateTime 및 DateTimeOffset 지원을 참조하세요.

콜백

`Newtonsoft.Json`은 다음과 같이 직렬화 또는 역직렬화 프로세스의 여러 지점에서 사용자 지정 코드를 실행할 수 있습니다.

- OnDeserializing(개체 역직렬화를 시작할 때)
- OnDeserialized(개체 역직렬화가 완료될 때)
- OnSerializing(개체 직렬화를 시작할 때)
- OnSerialized(개체 직렬화가 완료될 때)

System.Text.Json은 직렬화 및 역직렬화 중에 동일한 알림을 노출합니다. 이를 사용하려면 System.Text.Json.Serialization 네임스페이스에서 다음 인터페이스 중 하나 이상을 구현합니다.

- IJsonOnDeserializing
- IJsonOnDeserialized
- IJsonOnSerializing
- IJsonOnSerialized

다음은 직렬화 및 역직렬화의 시작과 끝에서 null 속성을 확인하고 메시지를 쓰는 예제입니다.

```
C#
```

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Callbacks
{
    public class WeatherForecast :
        IJsonOnDeserializing, IJsonOnDeserialized,
        IJsonOnSerializing, IJsonOnSerialized
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }

        void IJsonOnDeserializing.OnDeserializing() => Console.WriteLine("\nBegin
deserializing");
        void IJsonOnDeserialized.OnDeserialized()
        {
            Validate();
            Console.WriteLine("Finished deserializing");
        }
        void IJsonOnSerializing.OnSerializing()
        {
            Console.WriteLine("Begin serializing");
            Validate();
        }
        void IJsonOnSerialized.OnSerialized() => Console.WriteLine("Finished
serializing");

        private void Validate()
        {
            if (Summary is null)
            {
                Console.WriteLine("The 'Summary' property is 'null'.");
            }
        }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
            };

            string jsonString = JsonSerializer.Serialize(weatherForecast);
            Console.WriteLine(jsonString);

            weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius=
```

```
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary={weatherForecast?.Summary}");
        }
    }
}
// output:
//Begin serializing
//The 'Summary' property is 'null'.
//Finished serializing
//{"Date":"2019-08-01T00:00:00","TemperatureCelsius":25,"Summary":null}

//Begin deserializing
//The 'Summary' property is 'null'.
//Finished deserializing
//Date=8/1/2019 12:00:00 AM
//TemperatureCelsius = 25
//Summary=
```

`OnDeserializing` 코드는 새 POCO 인스턴스에 액세스할 수 없습니다. 역직렬화를 시작할 때 새 POCO 인스턴스를 조작하려면 이 코드를 POCO 생성자에 배치하세요.

public이 아닌 속성 setter 및 getter

`Newtonsoft.Json`은 `JsonProperty` 특성을 통해 private/internal 속성 setter 및 getter를 사용할 수 있습니다.

System.Text.Json은 JsonInclude 특성을 통해 private 및 internal 속성 setter 및 getter를 지원합니다. 샘플 코드는 public이 아닌 속성 접근자를 참조하세요.

기존 개체 채우기

`JsonConvert.PopulateObject`의 `Newtonsoft.Json` 메서드는 새 인스턴스를 만드는 대신 JSON 문서를 클래스의 기존 인스턴스로 역직렬화합니다. System.Text.Json은 항상 매개 변수 없는 기본 public 생성자를 사용하여 대상 형식의 새 인스턴스를 만듭니다. 사용자 지정 변환기는 기존 인스턴스로 역직렬화할 수 있습니다.

속성을 바꾸지 않고 재사용

.NET 8 System.Text.Json 부터는 초기화된 속성을 바꾸는 대신 다시 사용할 수 있습니다. API 제안 ↗에서 읽을 수 있는 동작에는 몇 가지 차이점이 있습니다.

자세한 내용은 초기화된 속성 채우기를 참조하세요.

setter 없이 속성 채우기

.NET 8 System.Text.Json 부터 setter가 없는 속성을 포함하여 채우기 속성을 지원합니다. 자세한 내용은 초기화된 속성 채우기를 참조하세요.

스네이크 표기법 이름 지정 정책

System.Text.Json에는 뱀 케이스에 대한 기본 제공 명명 정책이 포함되어 있습니다. 그러나 일부 입력에는 `Newtonsoft.Json`와(과) 몇 가지 동작 차이점이 있습니다. 다음 표에서는 JsonNamingPolicy.SnakeCaseLower 정책을 사용하여 입력을 변환할 때의 몇 가지 차이점을 보여 줍니다.

테이블 확장

입력	Newtonsoft.Json 결과	System.Text.Json 결과
AB1	"a_b1"	"ab1"
SHA512Managed	"sh_a512_managed"	"sha512_관리됨"
"abc123DEF456"	"abc123_de_f456"	"abc123_def456"
KEBAB-CASE	keba_b-_case	"케밥 케이스"

System.Runtime.Serialization 특성

System.Runtime.Serialization, DataContractAttribute 및 DataMemberAttribute 같은 IgnoreDataMemberAttribute 특성을 *데이터 계약*으로 정의할 수 있습니다. 데이터 계약 은 서비스와 클라이언트 사이에서 교환할 데이터를 추상적으로 설명한 공식 계약입니다. 데이터 계약은 교환을 위해 직렬화되는 속성을 정확하게 정의합니다.

System.Text.Json에는 이러한 특성에 대한 기본 제공 지원이 없습니다. 그러나 .NET 7부터 사용자 지정 형식 확인자를 사용하여 지원을 추가할 수 있습니다. 샘플은 ZCS.DataContractResolver 를 참조하세요.

8진수

`Newtonsoft.Json`은 선행 0이 있는 숫자를 8진수로 처리합니다. System.Text.Json 사양에서 선행 0을 허용하지 않으므로 은 선행 0을 허용하지 않습니다.

누락된 멤버 처리

JSON이 대상 형식에 없는 속성을 포함하는 경우 역직렬화 중에 예외를 발생시키도록 `Newtonsoft.Json`(을)를 구성할 수 있습니다. 기본적으로, System.Text.Json은

[JsonExtensionData] 특성을 사용하는 경우를 제외하고 JSON의 추가 속성을 무시합니다.

.NET 8 이상 버전에서는 다음 방법 중 하나를 사용하여 매핑되지 않은 JSON 속성을 건너뛰거나 허용하지 않는지 여부에 대한 기본 설정을 지정할 수 있습니다.

- 역직렬화할 형식에 JsonUnmappedMemberHandlingAttribute 특성을 적용합니다.
- 기본 설정을 전역적으로 설정하려면 JsonSerializerOptions.UnmappedMemberHandling 속성을 설정합니다. 또는 원본 생성의 경우 JsonSourceGenerationOptionsAttribute.UnmappedMemberHandling 속성을 설정하고 JsonSerializerContext 클래스에 특성을 적용합니다.
- JsonTypeInfo.UnmappedMemberHandling 속성을 사용자 지정합니다.

JsonObjectAttribute (JSON 객체 속성)

`Newtonsoft.Json`에는 직렬화되는 멤버, `JsonObjectAttribute` 값 처리 방법 및 모든 멤버가 필요한지 여부를 제어하기 위해 *형식 수준*에서 적용할 수 있는 `null` 특성이 있습니다. System.Text.Json에는 형식에 적용할 수 있는 동일한 특성이 없습니다. 값 처리와 같은 `null` 일부 동작의 경우 전역 JsonSerializerOptions 또는 각 속성에서 동일한 동작을 사용하여 JsonIgnoreAttribute개별적으로 구성할 수 있습니다.

모든 `Newtonsoft.Json.JsonObjectAttribute` 속성을 무시하도록 지정하는 데 사용하는 `null`(으)로 다음 예제를 고려합니다.

C#

```
[JsonObject(ItemNullValueHandling = NullValueHandling.Ignore)]
public class Person { ... }
```

System.Text.Json에서 모든 형식 및 속성에 대한 동작을 설정할 수 있습니다.

C#

```
JsonSerializerOptions options = new()
{
    DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
};

string json = JsonSerializer.Serialize<Person>(person, options);
```

또는 각 속성에 대한 동작을 별도로 설정할 수 있습니다.

C#

```
public class Person
{
```

```
    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public string? Name { get; set; }

    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public int? Age { get; set; }
}
```

다음으로, 모든 멤버 속성이 JSON에 있어야 하도록 지정하는 데 `Newtonsoft.Json.JsonObjectAttribute`(을)를 사용하는 다음 예제를 고려합니다.

C#

```
[JsonObject(ItemRequired = Required.Always)]
public class Person { ... }
```

C# System.Text.Json 한정자 또는 `required` JsonRequiredAttribute에 추가하여 의 동일한 동작을 수행할 수 있습니다. 자세한 내용은 필수 속성을 참조하세요.

C#

```
public class Person
{
    [JsonRequired]
    public string? Name { get; set; }

    public required int? Age { get; set; }
}
```

마지막으로 JSON 스키마 생성에 대한 제목을 지정하는 데 사용하는 `Newtonsoft.Json.JsonObjectAttribute` 다음 예제를 고려합니다.

C#

```
[JsonObject(Title = "PersonTitle")]
public class Person { ... }
```

이 `Title` 속성은 JSON 스키마 메타데이터에 사용되며 System.Text.Json에는 직접적인 대응항목이 없습니다. .NET 9부터 JSON 스키마를 생성하고 대리자를 사용하여 JsonSchemaExporter 스키마 제목을 사용자 지정할 수 있습니다TransformSchemaNode. 예제는 생성된 스키마 변환을 참조하세요.

TraceWriter

`Newtonsoft.Json`은 직렬화 또는 역직렬화에서 생성된 로그를 살펴보는 `TraceWriter`를 사용하여 디버그할 수 있습니다. System.Text.Json은 로깅을 수행하지 않습니다.

JsonDocument 및 JsonElement과 JToken(예: JObject, JArray)의 비교

System.Text.Json.JsonDocument은 기존 JSON 페이로드의 **읽기 전용** DOM(문서 개체 모델)을 구문 분석하고 빌드하는 기능을 제공합니다. DOM은 JSON 페이로드의 데이터에 대한 임의 액세스를 제공합니다. 페이로드를 구성하는 JSON 요소는 JsonElement 형식을 통해 액세스할 수 있습니다. `JsonElement` 형식은 JSON 텍스트를 일반적인 .NET 형식으로 변환하는 API를 제공합니다. `JsonDocument`는 RootElement 속성을 노출합니다.

.NET 6부터 네임스페이스의 JsonNode 형식 및 기타 형식을 사용하여 기존 JSON 페이로드에서 System.Text.Json.Nodes DOM을 구문 분석하고 빌드할 수 있습니다. 자세한 내용은 JsonNode 사용을 참조하세요.

JsonDocument는 IDisposable

`JsonDocument`는 데이터의 메모리 내 보기를 풀링된 버퍼에 빌드합니다. 따라서 `JObject`의 `JArray` 또는 `Newtonsoft.Json`와 달리, `JsonDocument` 형식은 `IDisposable`을 구현하며 using 블록 내에서 사용해야 합니다. 자세한 내용은 JsonDocument는 IDisposable을 참조하세요.

JsonDocument는 읽기 전용

System.Text.Json DOM은 JSON 요소를 추가, 제거 또는 수정할 수 없습니다. 이렇게 설계한 이유는 성능을 향상하고 일반 JSON 페이로드 크기를 구문 분석할 때 할당을 줄이는(즉, < 1MB) 것입니다.

JsonElement는 공용 구조체

`JsonDocument`는 `RootElement`를 JSON 요소를 포함하는 공용 구조체 형식인 JsonElement 형식의 속성으로 노출합니다. `Newtonsoft.Json`은 `JObject`, `JArray`, `JToken` 등의 전용 계층형 형식을 사용합니다. `JsonElement`는 검색하고 열거할 수 있으며, `JsonElement`를 사용하여 JSON 요소를 .NET 형식으로 구체화할 수 있습니다.

.NET 6부터 JsonNode 형식 및 System.Text.Json.Nodes 네임스페이스에서 `JObject`, `JArray` 및 `JToken`에 해당하는 형식을 사용할 수 있습니다. 자세한 내용은 JsonNode 사용을 참조하세요.

하위 요소의 JsonDocument 및 JsonElement를 검색하는 방법

`JObject`에서 `JArray` 또는 `Newtonsoft.Json`를 사용하여 JSON 토큰을 검색하면 일부 사전을 조회하기 때문에 비교적 속도가 빠릅니다. 그에 비해, `JsonElement`에서 검색하려면 속성을 순차적

으로 검색해야 하므로 상대적으로 느립니다(예: `TryGetProperty`를 사용하는 경우).

System.Text.Json은 조회 시간이 아닌 초기 구문 분석 시간을 최소화하도록 설계되었습니다. 자세한 내용은 하위 요소의 JsonDocument 및 JsonElement를 검색하는 방법을 참조하세요.

Utf8JsonReader 대 JsonTextReader

System.Text.Json.Utf8JsonReader는 ReadOnlySpan<byte> 또는 ReadOnlySequence<byte>에서 읽어온 UTF-8 인코딩 JSON 텍스트를 위한 고성능, 저할당, 전달 전용 판독기입니다. `Utf8JsonReader`는 사용자 지정 구문 분석기 및 역직렬 변환기를 빌드하는 데 활용할 수 있는 하위 수준 형식입니다.

Utf8JsonReader는 ref struct

`JsonTextReader`의 `Newtonsoft.Json`는 클래스입니다. `Utf8JsonReader` 형식은 *ref 구조체*라는 점에서 다릅니다. 자세한 내용은 Utf8JsonReader에 대한 ref 구조체 제한을 참조하세요.

Null 값을 null 허용 값 형식으로 읽기

`Newtonsoft.Json`은 Nullable<T>을 반환하도록 하기 위해 `ReadAsBoolean` `Null`를 처리하는 `TokenType`과 같은 `bool?`를 반환하는 API를 제공합니다. 기본 제공 `System.Text.Json` API는 null을 허용하지 않는 값 형식만 반환합니다. 자세한 내용은 nullable 값 형식으로 null 값 읽기를 참조하세요.

JSON 읽기를 위한 다중 대상

특정 대상 프레임워크에 `Newtonsoft.Json`을 계속 사용해야 하는 경우 다중 대상을 지정하여 두 가지를 구현할 수 있습니다. 그러나 이 방법은 간단하지 않으며 `#ifdefs` 및 원본 복제가 필요합니다. 최대한 많은 코드를 공유하는 한 가지 방법은 `ref struct` 및 Utf8JsonReader 주위에 `Newtonsoft.Json.JsonTextReader` 래퍼를 만드는 것입니다. 이 래퍼는 동작의 차이를 격리하면서 공개 노출 영역을 통합합니다. 이렇게 하면 새 형식을 참조로 전달하는 것과 함께 변경 내용을 주로 형식 생성으로 격리할 수 있습니다. 다음은 Microsoft.Extensions.DependencyModel 라이브러리가 따르는 패턴입니다.

- UnifiedJsonReader.JsonTextReader.cs
- UnifiedJsonReader.Utf8JsonReader.cs

Utf8JsonWriter 대 JsonTextWriter

System.Text.Json.Utf8JsonWriter은 `String`, `Int32` 및 `DateTime`과 같은 일반적인 .NET 형식에서 UTF-8 인코딩 JSON 텍스트를 쓸 수 있는 고성능 방법을 제공합니다. writer는 사용자 지정 직렬 변환기를 빌드하는 데 사용할 수 있는 하위 수준 형식입니다.

원시 값 작성

`Newtonsoft.Json`에는 값이 필요한 원시 JSON을 작성하는 `WriteRawValue` 메서드가 있습니다. System.Text.Json에는 직접적으로 해당하는 항목 Utf8JsonWriter.WriteRawValue가 없습니다. 자세한 내용은 원시 JSON 작성을 참조하세요.

JSON 형식 사용자 지정

`JsonTextWriter`에는 다음과 같은 설정이 있으며, Utf8JsonWriter에는 해당 기능이 없습니다.

- QuoteChar - 문자열 값을 묶는 데 사용할 문자를 지정합니다. `Utf8JsonWriter`는 항상 큰 따옴표를 사용합니다.
- QuoteName - 속성 이름을 따옴표로 묶을지 여부를 지정합니다. `Utf8JsonWriter`는 항상 속성 이름을 따옴표로 묶습니다.

.NET 9부터는 Utf8JsonWriter 구조체에서 노출하는 옵션을 사용하기 위해 JsonWriterOptions의 들여쓰기 문자와 크기를 사용자 지정할 수 있습니다.

- JsonWriterOptions.IndentCharacter
- JsonWriterOptions.IndentSize

이러한 방법으로 `Utf8JsonWriter`에서 생성된 JSON을 사용자 지정할 수 있는 해결 방법은 없습니다.

Timespan, Uri 또는 char 값 쓰기

`JsonTextWriter`는 `WriteValue`, Uri 및 char 값에 대한 메서드를 제공합니다. `Utf8JsonWriter`에는 이에 해당하는 메서드가 없습니다. 그 대신 이러한 값을 문자열로 포맷하고 (예를 들어 `ToString()`을 호출) WriteStringValue를 호출할 수 있습니다.

JSON 작성을 위한 다중 대상

특정 대상 프레임워크에 `Newtonsoft.Json`을 계속 사용해야 하는 경우 다중 대상을 지정하여 두 가지를 구현할 수 있습니다. 그러나 이 방법은 간단하지 않으며 `#ifdefs` 및 원본 복제가 필요합니다. 최대한 많은 코드를 공유하는 한 가지 방법은 Utf8JsonWriter 및 `Newtonsoft.Json.JsonTextWriter` 주위에 래퍼를 만드는 것입니다. 이 래퍼는 동작의 차이를 격

리하면서 공개 노출 영역을 통합합니다. 이를 통해 변경 내용을 주로 형식 생성으로 격리할 수 있습니다. Microsoft.Extensions.DependencyModel 라이브러리는 다음 패턴을 따릅니다.

- UnifiedJsonWriter.JsonTextWriter.cs
- UnifiedJsonWriter.Utf8JsonWriter.cs

TypeNameHandling.All이 지원되지 않음

`TypeNameHandling.All`에서 `System.Text.Json` 동등 기능을 제외한 결정은 의도적인 것이었습니다. JSON 페이로드가 자체 형식 정보를 지정하도록 허용하는 것은 웹 애플리케이션에서 일반적인 취약성 소스입니다. 특히 `Newtonsoft.Json`을 `TypeNameHandling.All`로 구성하면 원격 클라이언트가 JSON 페이로드 자체에 전체 실행 파일 애플리케이션을 포함할 수 있으므로 역직렬화 중에 웹 애플리케이션이 포함된 코드를 추출하고 실행할 수 있습니다. 자세한 내용은 Friday the 13th JSON 공격 PowerPoint 및 Friday the 13th JSON 공격 세부 정보를 참조하세요.

JSON 경로 쿼리가 지원되지 않음

`JsonDocument` DOM은 JSON 경로를 사용한 쿼리를 지원하지 않습니다.

JsonNode DOM에서 각 `JsonNode` 인스턴스에는 해당 노드에 대한 경로를 반환하는 `GetPath` 메서드가 있습니다. 그러나 JSON 경로 쿼리 문자열을 기반으로 쿼리를 처리하는 기본 제공 API는 없습니다.

자세한 내용은 dotnet/runtime #31068 GitHub 이슈를 참조하세요.

일부 제한을 구성할 수 없음

System.Text.Json은 최대 토큰 크기 문자(166MB) 및 base 64(125MB)와 같이 일부 값에 대해 변경할 수 없는 제한을 설정합니다. 자세한 내용은 JsonConstants에서 소스 코드 및 GitHub 이슈 dotnet/runtime #39953을 참조하세요.

NaN, Infinity, -Infinity

Newtonsoft는 `NaN`, `Infinity` 및 `-Infinity` JSON 문자열 토큰을 구문 분석합니다. System.Text.Json에서는 JsonNumberHandling.AllowNamedFloatingPointLiterals(을)를 사용합니다. 이 설정을 사용하는 방법에 대한 자세한 내용은 따옴표 안에 숫자 허용 또는 쓰기를 참조하세요.

솔루션 전체 마이그레이션에 AI 사용

GitHub Copilot과 같은 AI 도구를 사용하여 `Newtonsoft.Json`에서 `System.Text.Json`로 솔루션 코드를 마이그레이션하는 데 도움을 받을 수 있습니다.

다음은 Visual Studio Copilot 채팅에서 솔루션을 마이그레이션하는 데 사용할 수 있는 프롬프트 예제입니다.

Copilot 프롬프트

```
Convert all serialization code in this #solution from Newtonsoft.Json to
System.Text.Json, using the recommended approach for my current .NET version.
- Update attributes and properties, including rules for skipping or renaming
during serialization
- Ensure polymorphic serialization continues to work correctly
- Respect existing custom converters and project-level settings (for example, from
the Utilities folder or appsettings.json)
- Update related unit tests and highlight any potential breaking changes
- Generate a migration summary
```

코필로트의 제안을 적용하기 전에 검토합니다.

GitHub Copilot에 대한 자세한 내용은 GitHub의 FAQ를 참조하세요.

추가 리소스

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법
- Visual Studio의 GitHub Copilot
- VS Code의 GitHub Copilot

System.Text.Json을 사용하여 JsonSerializerOptions 인스턴스를 인스턴스화하는 방법

아티클 • 2024. 11. 12.

이 문서에서는 JsonSerializerOptions를 사용하는 경우 성능 문제를 방지하는 방법을 설명합니다. 또한 사용할 수 있는 매개 변수가 있는 생성자를 사용하는 방법을 보여 줍니다.

JsonSerializerOptions 인스턴스 다시 사용

동일한 옵션으로 `JsonSerializerOptions`를 반복적으로 사용하는 경우 사용할 때마다 새 `JsonSerializerOptions` 인스턴스를 만들지 마세요. 모든 호출에 대해 동일한 인스턴스를 다시 사용하세요. 이 지침은 사용자 지정 변환기에 대해 작성하는 코드와 JsonSerializer.Serialize 또는 JsonSerializer.Deserialize을 호출할 때 적용됩니다. 여러 스레드에서 동일한 인스턴스를 사용하는 것이 안전합니다. 옵션 인스턴스의 메타데이터 캐시는 스레드로부터 안전하며, 첫 번째 serialization 또는 deserialization 후에는 인스턴스를 변경할 수 없습니다.

`JsonSerializerOptions.Default` 속성

사용해야 하는 `JsonSerializerOptions`의 인스턴스가 (모든 기본 설정 및 기본 변환기가 있는) 기본 인스턴스인 경우, 옵션 인스턴스를 만드는 대신 JsonSerializerOptions.Default 속성을 사용합니다. 자세한 내용은 기본 시스템 변환기 사용을 참조하세요.

JsonSerializerOptions 복사

다음 예제와 같이 기존 인스턴스와 동일한 옵션을 사용하여 새 인스턴스를 만들 수 있는 JsonSerializerOptions 생성자가 있습니다.

C#

```
using System.Text.Json;

namespace CopyOptions
{
    public class Forecast
    {
        public DateTime Date { get; init; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
```

```
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new()
            {
                WriteIndented = true
            };

            JsonSerializerOptions optionsCopy = new(options);
            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, optionsCopy);

            Console.WriteLine($"Output JSON:\n{forecastJson}");
        }
    }
}

// Produces output like the following example:
//
//Output JSON:
//{
//  "Date": "2020-10-21T15:40:06.8998502-07:00",
//  "TemperatureC": 40,
//  "Summary": "Hot"
//}
```

기존 `JsonSerializerOptions` 인스턴스의 메타데이터 캐시는 새 인스턴스에 복사되지 않습니다. 따라서 이 생성자를 사용하는 것은 `JsonSerializerOptions`의 기존 인스턴스를 다시 사용하는 것과 동일하지 않습니다.

JsonSerializerOptions의 웹 기본값

다음 옵션은 웹앱의 기본값이 다릅니다.

- PropertyNameCaseInsensitive = `true`
- PropertyNamingPolicy = CamelCase
- NumberHandling = AllowReadingFromString

NET 9 이상 버전에서는 JsonSerializerOptions.Web 싱글톤을 사용하여 웹앱에 ASP.NET Core가 사용하는 기본 옵션으로 직렬화할 수 있습니다. 이전 버전에서는 다음 예제와 같이 JsonSerializerOptions 생성자를 호출하여 웹 기본값으로 새 인스턴스를 만듭니다.

C#

```
using System.Text.Json;

namespace OptionsDefaults
{
    public class Forecast
    {
        public DateTime? Date { get; init; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new(JsonSerializerDefaults.Web)
            {
                WriteIndented = true
            };

            Console.WriteLine(
                $"PropertyNameCaseInsensitive:
{options.PropertyNameCaseInsensitive}");
            Console.WriteLine(
                $"JsonNamingPolicy: {options.PropertyNamingPolicy}");
            Console.WriteLine(
                $"NumberHandling: {options.NumberHandling}");

            string forecastJson = JsonSerializer.Serialize<Forecast>
(forecast, options);
            Console.WriteLine($"Output JSON:\n{forecastJson}");

            Forecast? forecastDeserialized =
                JsonSerializer.Deserialize<Forecast>(forecastJson, options);

            Console.WriteLine($"Date: {forecastDeserialized?.Date}");
            Console.WriteLine($"TemperatureC:
{forecastDeserialized?.TemperatureC}");
            Console.WriteLine($"Summary: {forecastDeserialized?.Summary}");
        }
```

```
    }
}

// Produces output like the following example:
//
//PropertyNameCaseInsensitive: True
//JsonNamingPolicy: System.Text.Json.JsonCamelCaseNamingPolicy
//NumberHandling: AllowReadingFromString
//Output JSON:
//{
//  "date": "2020-10-21T15:40:06.9040831-07:00",
//  "temperatureC": 40,
//  "summary": "Hot"
//}
//Date: 10 / 21 / 2020 3:40:06 PM
//TemperatureC: 40
//Summary: Hot
```

System.Text.Json으로 대/소문자를 구분하지 않는 이름 일치를 사용하는 방법

아티클 • 2023. 05. 26.

이 문서에서는 `System.Text.Json` 네임스페이스로 대/소문자를 구분하지 않는 속성 이름 일치를 사용하는 방법을 알아봅니다.

대/소문자를 구분하지 않는 속성 매칭

기본적으로 역직렬화는 JSON과 대상 개체 속성 간에 일치하는 대/소문자 구분 속성 이름을 찾습니다. 이 동작을 변경하려면 JsonSerializerOptions.PropertyNameCaseInsensitive을 `true`로 설정합니다.

> ⓘ **참고**
>
> **웹 기본값**은 대/소문자 구분 안 함입니다.

C#

```
var options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString, options);
```

다음은 카멜식 대/소문자 속성 이름을 사용하는 JSON 예제입니다. 파스칼식 대/소문자 속성 이름을 사용하는 다음 형식으로 역직렬화할 수 있습니다.

JSON

```
{
  "date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "summary": "Hot",
}
```

C#

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
```

```
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

참고 항목

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

System.Text.Json을 사용하여 참조를 유지하고 순환 참조를 처리 또는 무시하는 방법

아티클 • 2024. 10. 21.

이 문서에서는 System.Text.Json을 사용하여 .NET에서 JSON을 직렬화 및 역직렬화하는 동안 참조를 유지하고 순환 참조를 처리하거나 무시하는 방법을 보여 줍니다.

참조를 보존하고 순환 참조를 처리

참조를 보존하고 순환 참조를 처리하려면 ReferenceHandler를 Preserve로 설정합니다. 이렇게 설정하면 다음과 같은 동작이 발생합니다.

- 직렬화 시:

  복합 형식을 쓸 때 직렬 변환기는 메타데이터 속성(`$id`, `$values`, `$ref`)도 씁니다.

- 역직렬화 시:

  메타데이터가 필요하며(필수는 아님), 역직렬 변환기는 이해를 시도합니다.

다음 코드는 `Preserve` 설정을 사용하는 방법을 보여 줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace PreserveReferences
{
    public class Employee
    {
        public string? Name { get; set; }
        public Employee? Manager { get; set; }
        public List<Employee>? DirectReports { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            Employee tyler = new()
            {
                Name = "Tyler Stein"
            };
```

```
            Employee adrian = new()
            {
                Name = "Adrian King"
            };

            tyler.DirectReports = [adrian];
            adrian.Manager = tyler;

            JsonSerializerOptions options = new()
            {
                ReferenceHandler = ReferenceHandler.Preserve,
                WriteIndented = true
            };

            string tylerJson = JsonSerializer.Serialize(tyler, options);
            Console.WriteLine($"Tyler serialized:\n{tylerJson}");

            Employee? tylerDeserialized =
                JsonSerializer.Deserialize<Employee>(tylerJson, options);

            Console.WriteLine(
                "Tyler is manager of Tyler's first direct report: ");
            Console.WriteLine(
                tylerDeserialized?.DirectReports?[0].Manager ==
tylerDeserialized);
        }
    }
}

// Produces output like the following example:
//
//Tyler serialized:
//{
//  "$id": "1",
//  "Name": "Tyler Stein",
//  "Manager": null,
//  "DirectReports": {
//    "$id": "2",
//    "$values": [
//      {
//        "$id": "3",
//        "Name": "Adrian King",
//        "Manager": {
//          "$ref": "1"
//        },
//        "DirectReports": null
//      }
//    ]
//  }
//}
//Tyler is manager of Tyler's first direct report:
//True
```

이 기능은 값 형식 또는 변경 불가능한 형식을 유지하는 데 사용할 수 없습니다. 역직렬화 시 변경 불가능한 형식의 인스턴스는 전체 페이로드를 읽은 후에 생성됩니다. 따라서 JSON 페이로드 내에 해당 참조가 표시되는 경우 동일한 인스턴스를 역직렬화할 수 없습니다.

값 형식, 변경 불가능한 형식, 배열의 경우 참조 메타데이터가 직렬화되지 않습니다. 역직렬화 시 `$ref` 또는 `$id`가 발견되면 예외가 throw됩니다. 그러나 값 형식은 무시 `$id` (컬렉션 `$values` 의 경우)하여 이러한 형식에 대한 메타데이터를 serialize하는 를 사용하여 Newtonsoft.Jsonserialize된 페이로드를 역직렬화할 수 있도록 합니다.

개체가 같은지 확인하기 위해 System.Text.Json은 두 개체 인스턴스를 비교할 때 값 같음(Object.Equals(Object)) 대신 참조 같음(Object.ReferenceEquals(Object, Object))을 사용하는 ReferenceEqualityComparer.Instance을 사용합니다.

참조가 직렬화 및 역직렬화되는 방법에 대한 자세한 내용은 ReferenceHandler.Preserve을 참조하세요.

ReferenceResolver 클래스는 직렬화 및 역직렬화 시 참조를 보존하는 동작을 정의합니다. 파생 클래스를 만들어 사용자 지정 동작을 지정합니다. 예제는 GuidReferenceResolver 를 참조하세요.

여러 serialization 및 deserialization 호출에서 참조 메타데이터 유지

기본적으로 참조 데이터는 Serialize 또는 Deserialize에 대한 각 호출에 대해서만 캐시됩니다. 한 `Serialize` 참조에서 참조를 유지하거나 `Deserialize` 다른 참조를 호출하려면 호출 사이트의 `Serialize`/`Deserialize`인스턴스를 루트 ReferenceResolver 로 지정합니다. 다음 코드에서는 이 시나리오의 예제를 보여 줍니다.

- `Employee` 개체 목록이 있으며 각 개체를 개별적으로 직렬화해야 합니다.
- `ReferenceHandler`에 대한 확인자에서 저장된 참조를 활용하려고 합니다.

다음은 `Employee` 클래스입니다.

C#

```
public class Employee
{
    public string? Name { get; set; }
    public Employee? Manager { get; set; }
    public List<Employee>? DirectReports { get; set; }
}
```

ReferenceResolver에서 파생되는 클래스는 참조를 사전에 저장합니다.

C#

```
class MyReferenceResolver : ReferenceResolver
{
    private uint _referenceCount;
    private readonly Dictionary<string, object> _referenceIdToObjectMap =
[];
    private readonly Dictionary<object, string> _objectToReferenceIdMap =
new (ReferenceEqualityComparer.Instance);

    public override void AddReference(string referenceId, object value)
    {
        if (!_referenceIdToObjectMap.TryAdd(referenceId, value))
        {
            throw new JsonException();
        }
    }

    public override string GetReference(object value, out bool
alreadyExists)
    {
        if (_objectToReferenceIdMap.TryGetValue(value, out string?
referenceId))
        {
            alreadyExists = true;
        }
        else
        {
            _referenceCount++;
            referenceId = _referenceCount.ToString();
            _objectToReferenceIdMap.Add(value, referenceId);
            alreadyExists = false;
        }

        return referenceId;
    }

    public override object ResolveReference(string referenceId)
    {
        if (!_referenceIdToObjectMap.TryGetValue(referenceId, out object?
value))
        {
            throw new JsonException();
        }

        return value;
    }
}
```

ReferenceHandler에서 파생되는 클래스는 `MyReferenceResolver`의 인스턴스를 보유하며 (이 예제에서 `Reset`으로 명명된 메서드에서) 필요한 경우에만 새 인스턴스를 만듭니다.

C#

```
class MyReferenceHandler : ReferenceHandler
{
    public MyReferenceHandler() => Reset();
    private ReferenceResolver? _rootedResolver;
    public override ReferenceResolver CreateResolver() => _rootedResolver!;
    public void Reset() => _rootedResolver = new MyReferenceResolver();
}
```

샘플 코드는 직렬 변환기를 호출할 때 ReferenceHandler 속성이 `MyReferenceHandler`의 인스턴스로 설정된 JsonSerializerOptions 인스턴스를 사용합니다. 이 패턴을 따를 때는 직렬화를 마쳤을 때 `ReferenceResolver` 사전을 다시 설정하여 영원히 증가하지 않도록 해야 합니다.

C#

```
var options = new JsonSerializerOptions
{
    WriteIndented = true
};
var myReferenceHandler = new MyReferenceHandler();
options.ReferenceHandler = myReferenceHandler;

string json;
foreach (Employee emp in employees)
{
    json = JsonSerializer.Serialize(emp, options);
    DoSomething(json);
}

// Reset after serializing to avoid out of bounds memory growth in the
resolver.
myReferenceHandler.Reset();
```

순환 참조 무시

순환 참조를 처리하는 대신 무시할 수 있습니다. 순환 참조를 무시하려면 ReferenceHandler를 IgnoreCycles로 설정합니다. 직렬 변환기는 다음 예제와 같이 순환 참조 속성을 `null`로 설정합니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SerializeIgnoreCycles
```

```
{
    public class Employee
    {
        public string? Name { get; set; }
        public Employee? Manager { get; set; }
        public List<Employee>? DirectReports { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            Employee tyler = new()
            {
                Name = "Tyler Stein"
            };

            Employee adrian = new()
            {
                Name = "Adrian King"
            };

            tyler.DirectReports = new List<Employee> { adrian };
            adrian.Manager = tyler;

            JsonSerializerOptions options = new()
            {
                ReferenceHandler = ReferenceHandler.IgnoreCycles,
                WriteIndented = true
            };

            string tylerJson = JsonSerializer.Serialize(tyler, options);
            Console.WriteLine($"Tyler serialized:\n{tylerJson}");

            Employee? tylerDeserialized =
                JsonSerializer.Deserialize<Employee>(tylerJson, options);

            Console.WriteLine(
                "Tyler is manager of Tyler's first direct report: ");
            Console.WriteLine(
                tylerDeserialized?.DirectReports?[0]?.Manager ==
tylerDeserialized);
        }
    }
}

// Produces output like the following example:
//
//Tyler serialized:
//{
//  "Name": "Tyler Stein",
//  "Manager": null,
//  "DirectReports": [
//    {
//      "Name": "Adrian King",
```

```
//      "Manager": null,
//      "DirectReports": null
//    }
//  ]
//}
//Tyler is manager of Tyler's first direct report:
//False
```

앞의 예제에서 `Adrian King` 아래의 `Manager`는 순환 참조를 방지하기 위해 `null`로 serialize됩니다. 이 동작은 ReferenceHandler.Preserve에 비해 다음과 같은 이점이 있습니다.

- 페이로드 크기가 줄어듭니다.
- System.Text.Json 및 Newtonsoft.Json 이외의 직렬 변환기에 대해 적절한 JSON이 생성됩니다.

이 동작에는 다음과 같은 단점이 있습니다.

- 데이터가 자동으로 손실됩니다.
- 데이터는 JSON에서 원본 개체로 왕복할 수 없습니다.

참고 항목

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

System.Text.Json을 사용하여 파생 클래스의 속성을 직렬화하는 방법

아티클 • 2024. 11. 12.

이 문서에서는 네임스페이스를 사용하여 파생 클래스의 속성을 serialize하는 방법을 알아봅니다 `System.Text.Json` .

파생 클래스의 속성 직렬화

.NET 7부터 `System.Text.Json`은 특성 주석을 사용하여 다형 형식 계층 구조 직렬화 및 역직렬화를 지원합니다.

테이블 확장

attribute	설명
JsonDerivedTypeAttribute	형식 선언에 배치되는 경우 지정된 하위 형식을 다형 직렬화로 옵트인해야 함을 나타냅니다. 또한 형식 판별자를 지정하는 기능도 노출합니다.
JsonPolymorphicAttribute	형식 선언에 배치하면 형식이 다형적으로 직렬화되어야 함을 나타냅니다. 또한 해당 형식에 대해 다형 직렬화 및 역직렬화를 구성하는 다양한 옵션을 노출합니다.

예를 들어 `WeatherForecastBase` 클래스와 `WeatherForecastWithCity` 파생 클래스가 있다고 가정해 봅시다.

C#

```
[JsonDerivedType(typeof(WeatherForecastWithCity))]
public class WeatherForecastBase
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

C#

```
public class WeatherForecastWithCity : WeatherForecastBase
{
    public string? City { get; set; }
}
```

그리고 컴파일 시간에 `Serialize<TValue>` 메서드의 형식 인수가 `WeatherForecastBase`라고 가정하겠습니다.

C#

```
options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize<WeatherForecastBase>
(weatherForecastBase, options);
```

이 시나리오에서는 `City` 속성이 직렬화됩니다. `weatherForecastBase` 개체가 실제로 `WeatherForecastWithCity` 개체이기 때문입니다. 이 구성은 특히 런타임 형식이 `WeatherForecastWithCity`인 경우 `WeatherForecastBase`에 대해 다형 직렬화를 지원합니다.

JSON

```
{
  "City": "Milwaukee",
  "Date": "2022-09-26T00:00:00-05:00",
  "TemperatureCelsius": 15,
  "Summary": "Cool"
}
```

`WeatherForecastBase`로서 페이로드의 라운드트립은 지원되지만 런타임 형식 `WeatherForecastWithCity`로 구체화되지 않습니다. 대신 런타임 형식 `WeatherForecastBase`로 구체화됩니다.

C#

```
WeatherForecastBase value = JsonSerializer.Deserialize<WeatherForecastBase>
("""
    {
      "City": "Milwaukee",
      "Date": "2022-09-26T00:00:00-05:00",
      "TemperatureCelsius": 15,
      "Summary": "Cool"
    }
    """);

Console.WriteLine(value is WeatherForecastWithCity); // False
```

다음 섹션에서는 파생 형식의 라운드트립을 지원하도록 메타데이터를 추가하는 방법을 설명합니다.

다형 형식 판별자

다형 역직렬화를 사용하도록 설정하려면 파생 클래스에 대해 형식 판별자를 지정해야 합니다.

C#

```
[JsonDerivedType(typeof(WeatherForecastBase), typeDiscriminator: "base")]
[JsonDerivedType(typeof(WeatherForecastWithCity), typeDiscriminator:
"withCity")]
public class WeatherForecastBase
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}

public class WeatherForecastWithCity : WeatherForecastBase
{
    public string? City { get; set; }
}
```

추가된 메타데이터, 특히 형식 판별자를 사용하여 직렬 변환기는 페이로드를 기본 형식 `WeatherForecastBase`에서 `WeatherForecastWithCity` 형식으로 직렬화 및 역직렬화할 수 있습니다. Serialization은 형식 판별자 메타데이터와 함께 JSON을 내보냅니다.

C#

```
WeatherForecastBase weather = new WeatherForecastWithCity
{
    City = "Milwaukee",
    Date = new DateTimeOffset(2022, 9, 26, 0, 0, 0, TimeSpan.FromHours(-5)),
    TemperatureCelsius = 15,
    Summary = "Cool"
}
var json = JsonSerializer.Serialize<WeatherForecastBase>(weather, options);
Console.WriteLine(json);
// Sample output:
//   {
//     "$type" : "withCity",
//     "City": "Milwaukee",
//     "Date": "2022-09-26T00:00:00-05:00",
//     "TemperatureCelsius": 15,
//     "Summary": "Cool"
//   }
```

형식 판별자를 사용하면 직렬 변환기는 페이로드를 `WeatherForecastWithCity`로 다형 역직렬화할 수 있습니다.

C#

```
WeatherForecastBase value = JsonSerializer.Deserialize<WeatherForecastBase>
(json);
Console.WriteLine(value is WeatherForecastWithCity); // True
```

ⓘ 참고

기본적으로 판별자는 `$type` JSON 개체의 시작 부분에 배치되어야 하며, 다음과 같은 `$id` 다른 메타데이터 속성과 `$ref`함께 그룹화되어야 합니다. 판별자를 JSON 개체의 중간에 배치 `$type` 하는 외부 API에서 데이터를 읽는 경우 다음으로 `true`설정합니다JsonSerializerOptions.AllowOutOfOrderMetadataProperties.

C#

```
JsonSerializerOptions options = new() {
AllowOutOfOrderMetadataProperties = true };
JsonSerializer.Deserialize<Base>("""
{"Name":"Name","$type":"derived"}""", options);
```

이 플래그를 사용하도록 설정하면 매우 큰 JSON 개체의 스트리밍 역직렬화를 수행할 때 오버 버퍼링(및 메모리 부족 실패)이 발생할 수 있으므로 주의해야 합니다.

형식 판별자 형식 혼합 및 일치

형식 판별자 식별자는 `string` 또는 `int` 양식에서 유효하므로 다음은 유효합니다.

C#

```
[JsonDerivedType(typeof(WeatherForecastWithCity), 0)]
[JsonDerivedType(typeof(WeatherForecastWithTimeSeries), 1)]
[JsonDerivedType(typeof(WeatherForecastWithLocalNews), 2)]
public class WeatherForecastBase { }

var json = JsonSerializer.Serialize<WeatherForecastBase>(new
WeatherForecastWithTimeSeries());
Console.WriteLine(json);
// Sample output:
//   {
//    "$type" : 1,
//    Omitted for brevity...
//   }
```

API는 형식 판별자 구성 혼합 및 일치를 지원하지만 권장되지 않습니다. 일반적인 권장 사항은 모두 `string` 형식 판별자를 사용하거나, 모두 `int` 형식 판별자를 사용하거나, 판별자를 전혀 사용하지 않는 것입니다. 다음 예제에서는 형식 판별자 구성을 혼합 및 일치시키는 방법을 보여 줍니다.

C#

```
[JsonDerivedType(typeof(ThreeDimensionalPoint), typeDiscriminator: 3)]
[JsonDerivedType(typeof(FourDimensionalPoint), typeDiscriminator: "4d")]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}

public sealed class FourDimensionalPoint : ThreeDimensionalPoint
{
    public int W { get; set; }
}
```

앞의 예제에서 `BasePoint` 형식은 형식 판별자가 없지만 `ThreeDimensionalPoint` 형식은 `int` 형식 판별자를 가지고 `FourDimensionalPoint` 형식은 `string` 형식 판별자를 갖습니다.

> **ⓘ 중요**
>
> 다형 직렬화가 작동하려면 직렬화된 값의 형식이 다형 기본 형식이어야 합니다. 여기에는 기본 형식을 루트 수준 값을 직렬화할 때 제네릭 형식 매개 변수로 사용하거나, 직렬화된 속성의 선언된 형식으로 사용하거나, 직렬화된 컬렉션의 컬렉션 요소로 사용하는 것이 포함됩니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

PerformRoundTrip<BasePoint>();
PerformRoundTrip<ThreeDimensionalPoint>();
PerformRoundTrip<FourDimensionalPoint>();

static void PerformRoundTrip<T>() where T : BasePoint, new()
{
```

```
    var json = JsonSerializer.Serialize<BasePoint>(new T());
    Console.WriteLine(json);

    BasePoint? result = JsonSerializer.Deserialize<BasePoint>(json);
    Console.WriteLine($"result is {typeof(T)}; // {result is T}");
    Console.WriteLine();
}
// Sample output:
//   { "X": 541, "Y": 503 }
//   result is BasePoint; // True
//
//   { "$type": 3, "Z": 399, "X": 835, "Y": 78 }
//   result is ThreeDimensionalPoint; // True
//
//   { "$type": "4d", "W": 993, "Z": 427, "X": 508, "Y": 741 }
//   result is FourDimensionalPoint; // True
```

형식 판별자 이름 사용자 지정

형식 판별자의 기본 속성 이름은 `$type`입니다. 속성 이름을 사용자 지정하려면 다음 예제와 같이 JsonPolymorphicAttribute를 사용합니다.

C#

```
[JsonPolymorphic(TypeDiscriminatorPropertyName = "$discriminator")]
[JsonDerivedType(typeof(ThreeDimensionalPoint), typeDiscriminator: "3d")]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public sealed class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}
```

이전 코드에서 `JsonPolymorphic` 특성은 `"$discriminator"` 값에 `TypeDiscriminatorPropertyName`을 구성합니다. 형식 판별자 이름을 구성했으면 다음 예제에서는 JSON으로 직렬화된 `ThreeDimensionalPoint` 형식을 보여줍니다.

C#

```
BasePoint point = new ThreeDimensionalPoint { X = 1, Y = 2, Z = 3 };
var json = JsonSerializer.Serialize<BasePoint>(point);
Console.WriteLine(json);
// Sample output:
//  { "$discriminator": "3d", "X": 1, "Y": 2, "Z": 3 }
```

> **팁**
>
> 형식 계층 구조의 속성과 충돌하는 JsonPolymorphicAttribute.TypeDiscriminatorPropertyName을 사용하지 마세요.

알 수 없는 파생 형식 처리

알 수 없는 파생 형식을 처리하려면 기본 형식에 주석을 사용하여 이러한 지원을 옵트인해야 합니다. 다음 형식 계층 구조를 예로 들 수 있습니다.

C#

```
[JsonDerivedType(typeof(ThreeDimensionalPoint))]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}

public class FourDimensionalPoint : ThreeDimensionalPoint
{
    public int W { get; set; }
}
```

구성에서 `FourDimensionalPoint`에 대한 지원을 명시적으로 옵트인하지 않았으므로 `FourDimensionalPoint` 인스턴스를 `BasePoint`로 직렬화하려고 하면 런타임 예외가 발생합니다.

C#

```
JsonSerializer.Serialize<BasePoint>(new FourDimensionalPoint()); // throws
NotSupportedException
```

다음과 같이 지정할 수 있는 JsonUnknownDerivedTypeHandling 열거형을 사용하여 기본 동작을 변경할 수 있습니다.

C#

```
[JsonPolymorphic(
    UnknownDerivedTypeHandling =
```

```
JsonUnknownDerivedTypeHandling.FallBackToBaseType)]
[JsonDerivedType(typeof(ThreeDimensionalPoint))]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}

public class FourDimensionalPoint : ThreeDimensionalPoint
{
    public int W { get; set; }
}
```

기본 형식으로 대체하는 대신 `FallBackToNearestAncestor` 설정을 사용하여 가장 가까운 선언된 파생 형식의 계약으로 대체할 수 있습니다.

C#

```
[JsonPolymorphic(
    UnknownDerivedTypeHandling =
JsonUnknownDerivedTypeHandling.FallBackToNearestAncestor)]
[JsonDerivedType(typeof(BasePoint))]
public interface IPoint { }

public class BasePoint : IPoint { }

public class ThreeDimensionalPoint : BasePoint { }
```

앞의 예제와 같은 구성을 사용하면 `ThreeDimensionalPoint` 형식이 `BasePoint`로 직렬화됩니다.

C#

```
// Serializes using the contract for BasePoint
JsonSerializer.Serialize<IPoint>(new ThreeDimensionalPoint());
```

그러나, 가장 가까운 상위 항목으로 다시 대체하는 것은 "다이아몬드" 모호성의 가능성이 있습니다. 예를 들어 다음 형식 계층 구조를 생각해 보겠습니다.

C#

```
[JsonPolymorphic(
    UnknownDerivedTypeHandling =
JsonUnknownDerivedTypeHandling.FallBackToNearestAncestor)]
```

```
[JsonDerivedType(typeof(BasePoint))]
[JsonDerivedType(typeof(IPointWithTimeSeries))]
public interface IPoint { }

public interface IPointWithTimeSeries : IPoint { }

public class BasePoint : IPoint { }

public class BasePointWithTimeSeries : BasePoint, IPointWithTimeSeries { }
```

이 경우 `BasePointWithTimeSeries` 형식은 `BasePoint` 또는 `IPointWithTimeSeries`로 직렬화할 수 있습니다. 둘 다 직접 상위 항목이기 때문입니다. 이러한 모호성으로 인해 `BasePointWithTimeSeries` 인스턴스를 `IPoint`로 직렬화하려고 하면 NotSupportedException이 throw됩니다.

C#

```
// throws NotSupportedException
JsonSerializer.Serialize<IPoint>(new BasePointWithTimeSeries());
```

계약 모델을 사용하여 다형성 구성

특성 주석이 비실용적이거나 불가능한 사용 사례(예: 대규모 도메인 모델, 어셈블리 간 계층 구조 또는 타사 종속성의 계층 구조)의 경우 다형성을 구성하려면 계약 모델을 사용합니다. 계약 모델은 다음 예제와 같이 형식별로 다형 구성을 동적으로 제공하는 사용자 지정 DefaultJsonTypeInfoResolver 하위 클래스를 만들어 형식 계층 구조에서 다형성을 구성하는 데 사용할 수 있는 API 집합입니다.

C#

```
public class PolymorphicTypeResolver : DefaultJsonTypeInfoResolver
{
    public override JsonTypeInfo GetTypeInfo(Type type,
JsonSerializerOptions options)
    {
        JsonTypeInfo jsonTypeInfo = base.GetTypeInfo(type, options);

        Type basePointType = typeof(BasePoint);
        if (jsonTypeInfo.Type == basePointType)
        {
            jsonTypeInfo.PolymorphismOptions = new JsonPolymorphismOptions
            {
                TypeDiscriminatorPropertyName = "$point-type",
                IgnoreUnrecognizedTypeDiscriminators = true,
                UnknownDerivedTypeHandling =
JsonUnknownDerivedTypeHandling.FailSerialization,
                DerivedTypes =
```

```
                {
                    new JsonDerivedType(typeof(ThreeDimensionalPoint),
"3d"),
                    new JsonDerivedType(typeof(FourDimensionalPoint), "4d")
                }
            };
        }

        return jsonTypeInfo;
    }
}
```

추가 다형 직렬화 세부 정보

- 다형 직렬화는 JsonDerivedTypeAttribute를 통해 명시적으로 옵트인된 파생 형식을 지원합니다. 선언되지 않은 형식으로 인해 런타임 예외가 발생합니다. JsonPolymorphicAttribute.UnknownDerivedTypeHandling 속성을 구성하여 이 동작을 변경할 수 있습니다.
- 파생 형식에 지정된 다형 구성은 기본 형식의 다형 구성이 상속하지 않습니다. 기본 형식은 독립적으로 구성해야 합니다.
- 다형 계층 구조는 `interface` 및 `class` 형식 모두에 대해 지원됩니다.
- 형식 판별자를 사용하는 다형성은 개체, 컬렉션 및 사전 형식에 대한 기본 변환기를 사용하는 형식 계층 구조에서만 지원됩니다.
- 다형성은 메타데이터 기반 소스 생성에서 지원되지만 빠른 경로 소스 생성에서는 지원되지 않습니다.

참고 항목

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

HttpClient의 Serialization 확장 메서드

아티클 • 2025. 01. 29.

네트워크에서 JSON 페이로드를 직렬화하고 역직렬화하는 작업은 일반적인 작업입니다. HttpClient 및 HttpContent 확장 메서드를 사용하면 이러한 작업을 한 줄의 코드로 수행할 수 있습니다. 이러한 확장 메서드는 JsonSerializerOptions 웹 기본값을 사용합니다.

다음 예제에서는 HttpClientJsonExtensions.GetFromJsonAsync 및 HttpClientJsonExtensions.PostAsJsonAsync사용하는 방법을 보여 줍니다.

C#

```
using System.Net.Http.Json;

namespace HttpClientExtensionMethods
{
    public class User
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Username { get; set; }
        public string? Email { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            using HttpClient client = new()
            {
                BaseAddress = new
Uri("https://jsonplaceholder.typicode.com")
            };

            // Get the user information.
            User? user = await client.GetFromJsonAsync<User>("users/1");
            Console.WriteLine($"Id: {user?.Id}");
            Console.WriteLine($"Name: {user?.Name}");
            Console.WriteLine($"Username: {user?.Username}");
            Console.WriteLine($"Email: {user?.Email}");

            // Post a new user.
            HttpResponseMessage response = await
client.PostAsJsonAsync("users", user);
            Console.WriteLine(
                $"{(response.IsSuccessStatusCode ? "Success" : "Error")} -
{response.StatusCode}");
        }
    }
}
```

```
// Produces output like the following example but with different names:
//
//Id: 1
//Name: Tyler King
//Username: Tyler
//Email: Tyler@contoso.com
//Success - Created
```

System.Text.Json에서 JSON 문서 개체 모델을 사용하는 방법

아티클 • 2024. 10. 21.

이 문서에서는 JSON 페이로드의 데이터에 임의로 액세스하기 위해 JSON DOM(문서 개체 모델)을 사용하는 방법을 보여 줍니다.

JSON DOM 선택

DOM을 사용하는 것은 다음의 경우 JsonSerializer를 사용한 역직렬화의 대안입니다.

- 역직렬화할 형식이 없습니다.
- 수신한 JSON에는 고정된 스키마가 없으며 포함된 내용을 확인하려면 검사해야 합니다.

`System.Text.Json`은 JSON DOM을 빌드하는 두 가지 방법을 다음과 같이 제공합니다.

- JsonDocument는 `Utf8JsonReader`를 사용하여 읽기 전용 DOM을 빌드하는 기능을 제공합니다. 페이로드를 구성하는 JSON 요소는 JsonElement 형식을 통해 액세스할 수 있습니다. `JsonElement` 형식은 JSON 텍스트를 일반적인 .NET 형식으로 변환하는 API와 함께 배열 및 개체 열거자를 제공합니다. `JsonDocument`는 RootElement 속성을 노출합니다. 자세한 내용은 이 문서의 뒷부분에서 JsonDocument 사용을 참조하세요.

- System.Text.Json.Nodes 네임스페이스에서 파생되는 클래스와 JsonNode는 변경 가능한 DOM을 만드는 기능을 제공합니다. 페이로드를 구성하는 JSON 요소는 JsonNode, JsonObject, JsonArray, JsonValue 및 JsonElement 형식을 통해 액세스할 수 있습니다. 자세한 내용은 이 문서의 뒷부분에서 JsonNode 사용을 참조하세요.

`JsonDocument`와 `JsonNode` 중에서 하나를 선택할 때 다음 요소를 고려합니다.

- `JsonNode` DOM을 만든 후에 변경할 수 있습니다. `JsonDocument` DOM은 변경할 수 없습니다.
- `JsonDocument` DOM은 데이터에 더 빠르게 액세스할 수 있도록 합니다.

`JsonNode` 사용

다음 예제에서는 JsonNode를 사용하는 방법과 System.Text.Json.Nodes 네임스페이스의 다른 형식을 사용하여 다음을 수행하는 방법을 보여줍니다.

- JSON 문자열에서 DOM 만들기
- DOM에서 JSON을 작성합니다.
- DOM에서 값, 개체 또는 배열을 가져옵니다.

C#

```
using System.Text.Json;
using System.Text.Json.Nodes;

namespace JsonNodeFromStringExample;

public class Program
{
    public static void Main()
    {
        string jsonString = """
            {
              "Date": "2019-08-01T00:00:00",
              "Temperature": 25,
              "Summary": "Hot",
              "DatesAvailable": [
                "2019-08-01T00:00:00",
                "2019-08-02T00:00:00"
              ],
              "TemperatureRanges": {
                  "Cold": {
                      "High": 20,
                      "Low": -10
                  },
                  "Hot": {
                      "High": 60,
                      "Low": 20
                  }
              }
            }
            """;
        // Create a JsonNode DOM from a JSON string.
        JsonNode forecastNode = JsonNode.Parse(jsonString)!;

        // Write JSON from a JsonNode
        var options = new JsonSerializerOptions { WriteIndented = true };
        Console.WriteLine(forecastNode!.ToJsonString(options));
        // output:
        //{
        //  "Date": "2019-08-01T00:00:00",
        //  "Temperature": 25,
        //  "Summary": "Hot",
        //  "DatesAvailable": [
        //    "2019-08-01T00:00:00",
        //    "2019-08-02T00:00:00"
        //  ],
        //  "TemperatureRanges": {
        //    "Cold": {
```

```
        //      "High": 20,
        //      "Low": -10
        //    },
        //    "Hot": {
        //      "High": 60,
        //      "Low": 20
        //    }
        //  }
        //}

        // Get value from a JsonNode.
        JsonNode temperatureNode = forecastNode!["Temperature"]!;
        Console.WriteLine($"Type={temperatureNode.GetType()}");
        Console.WriteLine($"JSON={temperatureNode.ToJsonString()}");
        //output:
        //Type =
System.Text.Json.Nodes.JsonValue`1[System.Text.Json.JsonElement]
        //JSON = 25

        // Get a typed value from a JsonNode.
        int temperatureInt = (int)forecastNode!["Temperature"]!;
        Console.WriteLine($"Value={temperatureInt}");
        //output:
        //Value=25

        // Get a typed value from a JsonNode by using GetValue<T>.
        temperatureInt = forecastNode!["Temperature"]!.GetValue<int>();
        Console.WriteLine($"TemperatureInt={temperatureInt}");
        //output:
        //Value=25

        // Get a JSON object from a JsonNode.
        JsonNode temperatureRanges = forecastNode!["TemperatureRanges"]!;
        Console.WriteLine($"Type={temperatureRanges.GetType()}");
        Console.WriteLine($"JSON={temperatureRanges.ToJsonString()}");
        //output:
        //Type = System.Text.Json.Nodes.JsonObject
        //JSON = { "Cold":{ "High":20,"Low":-10},"Hot":{ "High":60,"Low":20}
}

        // Get a JSON array from a JsonNode.
        JsonNode datesAvailable = forecastNode!["DatesAvailable"]!;
        Console.WriteLine($"Type={datesAvailable.GetType()}");
        Console.WriteLine($"JSON={datesAvailable.ToJsonString()}");
        //output:
        //datesAvailable Type = System.Text.Json.Nodes.JsonArray
        //datesAvailable JSON =["2019-08-01T00:00:00", "2019-08-
02T00:00:00"]

        // Get an array element value from a JsonArray.
        JsonNode firstDateAvailable = datesAvailable[0]!;
        Console.WriteLine($"Type={firstDateAvailable.GetType()}");
        Console.WriteLine($"JSON={firstDateAvailable.ToJsonString()}");
        //output:
        //Type =
```

```
System.Text.Json.Nodes.JsonValue`1[System.Text.Json.JsonElement]
        //JSON = "2019-08-01T00:00:00"

        // Get a typed value by chaining references.
        int coldHighTemperature = (int)forecastNode["TemperatureRanges"]!
["Cold"]!["High"]!;
        Console.WriteLine($"TemperatureRanges.Cold.High=
{coldHighTemperature}");
        //output:
        //TemperatureRanges.Cold.High = 20

        // Parse a JSON array
        var datesNode = JsonNode.Parse(@"[""2019-08-01T00:00:00"",""2019-08-
02T00:00:00""]");
        JsonNode firstDate = datesNode![0]!.GetValue<DateTime>();
        Console.WriteLine($"firstDate={ firstDate}");
        //output:
        //firstDate = "2019-08-01T00:00:00"
    }
}
```

개체 이니셜라이저를 사용하여 JsonNode DOM 만들기 및 변경

아래 예제는 다음과 같은 작업의 방법을 보여 줍니다.

- 개체 이니셜라이저를 사용하여 DOM을 만듭니다.
- DOM을 변경합니다.

C#

```
using System.Text.Json;
using System.Text.Json.Nodes;

namespace JsonNodeFromObjectExample;

public class Program
{
    public static void Main()
    {
        // Create a new JsonObject using object initializers.
        var forecastObject = new JsonObject
        {
            ["Date"] = new DateTime(2019, 8, 1),
            ["Temperature"] = 25,
            ["Summary"] = "Hot",
            ["DatesAvailable"] = new JsonArray(
                new DateTime(2019, 8, 1), new DateTime(2019, 8, 2)),
            ["TemperatureRanges"] = new JsonObject
            {
                ["Cold"] = new JsonObject
```

```
                {
                    ["High"] = 20,
                    ["Low"] = -10
                }
            },
            ["SummaryWords"] = new JsonArray("Cool", "Windy", "Humid")
        };

        // Add an object.
        forecastObject!["TemperatureRanges"]!["Hot"] =
            new JsonObject { ["High"] = 60, ["Low"] = 20 };

        // Remove a property.
        forecastObject.Remove("SummaryWords");

        // Change the value of a property.
        forecastObject["Date"] = new DateTime(2019, 8, 3);

        var options = new JsonSerializerOptions { WriteIndented = true };
        Console.WriteLine(forecastObject.ToJsonString(options));
        //output:
        //{
        //  "Date": "2019-08-03T00:00:00",
        //  "Temperature": 25,
        //  "Summary": "Hot",
        //  "DatesAvailable": [
        //    "2019-08-01T00:00:00",
        //    "2019-08-02T00:00:00"
        //  ],
        //  "TemperatureRanges": {
        //    "Cold": {
        //      "High": 20,
        //      "Low": -10
        //    },
        //    "Hot": {
        //      "High": 60,
        //      "Low": 20
        //    }
        //  }
        //}
    }
}
```

JSON 페이로드의 하위 섹션 역직렬화

다음 예제에서는 JsonNode를 사용하여 JSON 트리의 하위 섹션으로 이동하고 해당 하위 섹션에서 단일 값, 사용자 지정 형식 또는 배열을 역직렬화하는 방법을 보여줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Nodes;
```

```
namespace JsonNodePOCOExample;

public class TemperatureRanges : Dictionary<string, HighLowTemps>
{
}

public class HighLowTemps
{
    public int High { get; set; }
    public int Low { get; set; }
}

public class Program
{
    public static DateTime[]? DatesAvailable { get; set; }

    public static void Main()
    {
        string jsonString = """
            {
              "Date": "2019-08-01T00:00:00",
              "Temperature": 25,
              "Summary": "Hot",
              "DatesAvailable": [
                "2019-08-01T00:00:00",
                "2019-08-02T00:00:00"
              ],
              "TemperatureRanges": {
                  "Cold": {
                      "High": 20,
                      "Low": -10
                  },
                  "Hot": {
                      "High": 60,
                      "Low": 20
                  }
              }
            }
            """;
        // Parse all of the JSON.
        JsonNode forecastNode = JsonNode.Parse(jsonString)!;

        // Get a single value
        int hotHigh = forecastNode["TemperatureRanges"]!["Hot"]!
["High"]!.GetValue<int>();
        Console.WriteLine($"Hot.High={hotHigh}");
        // output:
        //Hot.High=60

        // Get a subsection and deserialize it into a custom type.
        JsonObject temperatureRangesObject = forecastNode!
["TemperatureRanges"]!.AsObject();
        using var stream = new MemoryStream();
        using var writer = new Utf8JsonWriter(stream);
```

```
        temperatureRangesObject.WriteTo(writer);
        writer.Flush();
        TemperatureRanges? temperatureRanges =
            JsonSerializer.Deserialize<TemperatureRanges>(stream.ToArray());
        Console.WriteLine($"Cold.Low={temperatureRanges!["Cold"].Low},
Hot.High={temperatureRanges["Hot"].High}");
        // output:
        //Cold.Low=-10, Hot.High=60

        // Get a subsection and deserialize it into an array.
        JsonArray datesAvailable = forecastNode!
["DatesAvailable"]!.AsArray()!;
        Console.WriteLine($"DatesAvailable[0]={datesAvailable[0]}");
        // output:
        //DatesAvailable[0]=8/1/2019 12:00:00 AM
    }
}
```

JsonNode 평균 등급 예제

다음 예제에서는 정수 값이 있는 JSON 배열을 선택하고 평균 값을 계산합니다.

C#

```
using System.Text.Json.Nodes;

namespace JsonNodeAverageGradeExample;

public class Program
{
    public static void Main()
    {
        string jsonString = """
            {
              "Class Name": "Science",
              "Teacher\u0027s Name": "Jane",
              "Semester": "2019-01-01",
              "Students": [
                {
                  "Name": "John",
                  "Grade": 94.3
                },
                {
                  "Name": "James",
                  "Grade": 81.0
                },
                {
                  "Name": "Julia",
                  "Grade": 91.9
                },
                {
                  "Name": "Jessica",
```

```
                    "Grade": 72.4
                },
                {
                    "Name": "Johnathan"
                }
            ],
            "Final": true
        }
        """;
        double sum = 0;
        JsonNode document = JsonNode.Parse(jsonString)!;

        JsonNode root = document.Root;
        JsonArray studentsArray = root["Students"]!.AsArray();

        int count = studentsArray.Count;
        foreach (JsonNode? student in studentsArray)
        {
            if (student?["Grade"] is JsonNode gradeNode)
            {
                sum += (double)gradeNode;
            }
            else
            {
                sum += 70;
            }
        }

        double average = sum / count;
        Console.WriteLine($"Average grade : {average}");
    }
}
// output:
//Average grade : 81.92
```

위의 코드는

- `Grade` 속성이 있는 `Students` 배열의 개체에 대한 평균 등급을 계산합니다.
- 등급이 없는 학생에게 기본 등급 70을 할당합니다.
- `JsonArray`의 `Count` 속성에서 학생 수를 가져옵니다.

`JsonNode`(`JsonSerializerOptions` 사용)

`JsonSerializer`를 사용하여 `JsonNode` 인스턴스를 직렬화하고 역직렬화할 수 있습니다. 그러나 `JsonSerializerOptions`를 사용하는 오버로드를 사용하는 경우 옵션 인스턴스는 사용자 지정 변환기를 가져오는 데만 사용됩니다. 옵션 인스턴스의 나머지 기능은 사용되지 않습니다. 예를 들어 JsonSerializerOptions.DefaultIgnoreCondition을 WhenWritingNull로 설정하고 `JsonSerializerOptions`를 사용하는 오버로드를 사용하여 `JsonSerializer`를 호출하는 경우 null 속성은 무시되지 않습니다.

`JsonSerializerOptions` 매개 변수를 사용하는 `JsonNode` 메서드인 WriteTo(Utf8JsonWriter, JsonSerializerOptions) 및 ToJsonString(JsonSerializerOptions)에도 동일한 제한 사항이 적용됩니다. 이러한 API는 사용자 지정 변환기를 가져오는 데만 `JsonSerializerOptions`를 사용합니다.

다음 예제에서는 `JsonSerializerOptions` 매개 변수를 사용하고 `JsonNode` 인스턴스를 직렬화하는 메서드를 사용한 결과를 보여줍니다.

C#

```
using System.Text;
using System.Text.Json;
using System.Text.Json.Nodes;
using System.Text.Json.Serialization;

namespace JsonNodeWithJsonSerializerOptions;

public class Program
{
    public static void Main()
    {
        Person person = new() { Name = "Nancy" };

        // Default serialization - Address property included with null
token.
        // Output: {"Name":"Nancy","Address":null}
        string personJsonWithNull = JsonSerializer.Serialize(person);
        Console.WriteLine(personJsonWithNull);

        // Serialize and ignore null properties - null Address property is
omitted
        // Output: {"Name":"Nancy"}
        JsonSerializerOptions options = new()
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        };
        string personJsonWithoutNull = JsonSerializer.Serialize(person,
options);
        Console.WriteLine(personJsonWithoutNull);

        // Ignore null properties doesn't work when serializing JsonNode
instance
        // by using JsonSerializer.
        // Output: {"Name":"Nancy","Address":null}
        JsonNode? personJsonNode = JsonSerializer.Deserialize<JsonNode>
(personJsonWithNull);
        personJsonWithNull = JsonSerializer.Serialize(personJsonNode,
options);
        Console.WriteLine(personJsonWithNull);

        // Ignore null properties doesn't work when serializing JsonNode
instance
```

```
        // by using JsonNode.ToJsonString method.
        // Output: {"Name":"Nancy","Address":null}
        personJsonWithNull = personJsonNode!.ToJsonString(options);
        Console.WriteLine(personJsonWithNull);

        // Ignore null properties doesn't work when serializing JsonNode
instance
        // by using JsonNode.WriteTo method.
        // Output: {"Name":"Nancy","Address":null}
        using var stream = new MemoryStream();
        using var writer = new Utf8JsonWriter(stream);
        personJsonNode!.WriteTo(writer, options);
        writer.Flush();
        personJsonWithNull = Encoding.UTF8.GetString(stream.ToArray());
        Console.WriteLine(personJsonWithNull);
    }
}

public class Person
{
    public string? Name { get; set; }
    public string? Address { get; set; }
}
```

사용자 지정 변환기 이외에 `JsonSerializerOptions`의 기능이 필요한 경우 `JsonNode` 대신 강력한 형식의 대상(예: 이 예제의 `Person` 클래스)과 함께 `JsonSerializer`를 사용합니다.

속성 순서 조작

JsonObject 는 페이로드의 JsonNode요소 중 하나이며 변경 가능한 JSON 개체를 나타냅니다. 형식은 각 항목이 개체의 속성인 형식으로 `IDictionary<string, JsonNode>`모델링되더라도 암시적 속성 순서를 캡슐화합니다. 그러나 특정 인덱스에서 항목을 삽입하고 RemoveAt(Int32) 제거할 수 있도록 하여 정렬된 사전과 같은 Insert(Int32, String, JsonNode) API에서 형식을 효과적으로 모델링합니다. 이러한 API를 사용하면 속성 순서에 직접적인 영향을 줄 수 있는 개체 인스턴스를 수정할 수 있습니다.

다음 코드는 개체의 시작 부분에 특정 속성을 추가하거나 이동하는 예제를 보여줍니다.

C#

```
var schema =
(JsonObject)JsonSerializerOptions.Default.GetJsonSchemaAsNode(typeof(MyPoco)
);

JsonNode? idValue;
switch (schema.IndexOf("$id"))
{
    // $id property missing.
    case < 0:
```

```
            idValue = (JsonNode)"https://example.com/schema";
            schema.Insert(0, "$id", idValue);
            break;

        // $id property already at the start of the object.
        case 0:
            break;

        // $id exists but not at the start of the object.
        case int index:
            idValue = schema[index];
            schema.RemoveAt(index);
            schema.Insert(0, "$id", idValue);
            break;
    }
```

JsonNodes 비교

하위 요소를 포함하여 두 `JsonNode` 개체를 같음으로 비교하려면 이 메서드를 JsonNode.DeepEquals(JsonNode, JsonNode) 사용합니다.

`JsonDocument` 사용

다음 예제에서는 JsonDocument 클래스를 사용하여 JSON 문자열의 데이터에 임의로 액세스하는 방법을 보여줍니다.

C#

```
double sum = 0;
int count = 0;

using (JsonDocument document = JsonDocument.Parse(jsonString))
{
    JsonElement root = document.RootElement;
    JsonElement studentsElement = root.GetProperty("Students");
    foreach (JsonElement student in studentsElement.EnumerateArray())
    {
        if (student.TryGetProperty("Grade", out JsonElement gradeElement))
        {
            sum += gradeElement.GetDouble();
        }
        else
        {
            sum += 70;
        }
        count++;
    }
}
```

```
double average = sum / count;
Console.WriteLine($"Average grade : {average}");
```

위의 코드는

- 분석할 JSON은 `jsonString`이라는 문자열에 있다고 가정합니다.
- `Grade` 속성이 있는 `Students` 배열의 개체에 대한 평균 등급을 계산합니다.
- 등급이 없는 학생에게 기본 등급 70을 할당합니다.
- `JsonDocument`가 `IDisposable`을 구현하기 때문에 using 문에 `JsonDocument` 인스턴스를 만듭니다. `JsonDocument` 인스턴스가 삭제되면 모든 `JsonElement` 인스턴스에 대한 액세스 권한도 잃게 됩니다. `JsonElement` 인스턴스에 대한 액세스를 유지하려면 이 인스턴스의 복사본을 만든 후 부모 `JsonDocument` 인스턴스를 삭제하세요. 복사본을 만들려면 JsonElement.Clone을 호출합니다. 자세한 내용은 JsonDocument is IDisposable을 참조하세요.

앞의 예제 코드는 각 반복을 사용하여 `count` 변수를 증가시켜 학생 수를 계산합니다. 다음 예제처럼 GetArrayLength를 호출하는 방법도 있습니다.

C#

```
double sum = 0;
int count = 0;

using (JsonDocument document = JsonDocument.Parse(jsonString))
{
    JsonElement root = document.RootElement;
    JsonElement studentsElement = root.GetProperty("Students");

    count = studentsElement.GetArrayLength();

    foreach (JsonElement student in studentsElement.EnumerateArray())
    {
        if (student.TryGetProperty("Grade", out JsonElement gradeElement))
        {
            sum += gradeElement.GetDouble();
        }
        else
        {
            sum += 70;
        }
    }
}

double average = sum / count;
Console.WriteLine($"Average grade : {average}");
```

다음은 이 코드가 처리하는 JSON 예제입니다.

JSON

```
{
  "Class Name": "Science",
  "Teacher\u0027s Name": "Jane",
  "Semester": "2019-01-01",
  "Students": [
    {
      "Name": "John",
      "Grade": 94.3
    },
    {
      "Name": "James",
      "Grade": 81.0
    },
    {
      "Name": "Julia",
      "Grade": 91.9
    },
    {
      "Name": "Jessica",
      "Grade": 72.4
    },
    {
      "Name": "Johnathan"
    }
  ],
  "Final": true
}
```

`JsonDocument` 대신 `JsonNode`를 사용하는 유사한 예제는 JsonNode 평균 등급 예제를 참조하세요.

하위 요소의 JsonDocument 및 JsonElement를 검색하는 방법

`JsonElement`에서 검색하려면 속성을 순차적으로 검색해야 하므로 검색 속도가 비교적 느립니다(예: `TryGetProperty`를 사용하는 경우). System.Text.Json은 조회 시간이 아닌 초기 구문 분석 시간을 최소화하도록 설계되었습니다. 따라서 `JsonDocument` 개체를 검색할 때 성능을 최적화하려면 다음 방법을 사용하세요.

- 자체적으로 인덱싱 또는 루프를 수행하지 말고 기본 제공 열거자(EnumerateArray 및 EnumerateObject)를 사용합니다.
- `RootElement`를 사용하여 전체 `JsonDocument`의 모든 속성을 순차적으로 검색하지 마세요. 그 대신, 알려진 JSON 데이터 구조체를 기반으로 중첩된 JSON 개체를 검색합니다. 예를 들어 앞의 코드 예제에서는 `Grade` 속성을 찾는 모든 `JsonElement` 개체를 검색하는 대신 `Student` 개체를 반복하고 각각에 대한 `Grade` 값을 가져오면

`Student` 개체에서 `Grade` 속성을 찾습니다. 후자의 검색을 수행하면 동일한 데이터에 대해 불필요한 전달 과정이 발생합니다.

JsonElements 비교

하위 요소를 포함하여 두 `JsonElement` 개체를 같음으로 비교하려면 이 메서드를 JsonElement.DeepEquals(JsonElement, JsonElement) 사용합니다.

C#

```
JsonElement left = JsonDocument.Parse("10e-3").RootElement;
JsonElement right = JsonDocument.Parse("0.01").RootElement;
bool equal = JsonElement.DeepEquals(left, right);
Console.WriteLine(equal); // True.
```

`JsonDocument`를 사용하여 JSON 작성

다음 예제에서는 JsonDocument에서 JSON을 쓰는 방법을 보여줍니다.

C#

```
string jsonString = File.ReadAllText(inputFileName);

var writerOptions = new JsonWriterOptions
{
    Indented = true
};

var documentOptions = new JsonDocumentOptions
{
    CommentHandling = JsonCommentHandling.Skip
};

using FileStream fs = File.Create(outputFileName);
using var writer = new Utf8JsonWriter(fs, options: writerOptions);
using JsonDocument document = JsonDocument.Parse(jsonString,
documentOptions);

JsonElement root = document.RootElement;

if (root.ValueKind == JsonValueKind.Object)
{
    writer.WriteStartObject();
}
else
{
    return;
}
```

```
foreach (JsonProperty property in root.EnumerateObject())
{
    property.WriteTo(writer);
}

writer.WriteEndObject();

writer.Flush();
```

위의 코드는

- JSON 파일을 읽고, `JsonDocument`에 데이터를 로드하고, 서식 있는(보기 좋게 출력된) JSON을 파일에 씁니다.
- JsonDocumentOptions를 사용하여 입력 JSON의 주석을 허용하지만 무시하도록 지정합니다.
- 완료되면 writer에서 Flush를 호출합니다. 삭제될 때 writer가 자동으로 플러시하도록 설정하는 방법도 있습니다.

다음은 예제 코드에서 처리할 JSON 입력의 예입니다.

JSON

```
{"Class Name": "Science","Teacher's Name": "Jane","Semester": "2019-01-
01","Students": [{"Name": "John","Grade": 94.3},{"Name": "James","Grade":
81.0},{"Name": "Julia","Grade": 91.9},{"Name": "Jessica","Grade": 72.4},
{"Name": "Johnathan"}],"Final": true}
```

결과는 다음과 같이 보기 좋게 출력된 JSON 출력입니다.

JSON

```
{
  "Class Name": "Science",
  "Teacher\u0027s Name": "Jane",
  "Semester": "2019-01-01",
  "Students": [
    {
      "Name": "John",
      "Grade": 94.3
    },
    {
      "Name": "James",
      "Grade": 81.0
    },
    {
      "Name": "Julia",
      "Grade": 91.9
    },
    {
```

```
      "Name": "Jessica",
      "Grade": 72.4
    },
    {
      "Name": "Johnathan"
    }
  ],
  "Final": true
}
```

JsonDocument는 IDisposable

`JsonDocument`는 데이터의 메모리 내 보기를 풀링된 버퍼에 빌드합니다. 따라서 `JsonDocument` 형식은 `IDisposable`을 구현하며 `using` 블록 내에서 사용해야 합니다.

수명 소유권 및 폐기 책임을 호출자에 양도하려는 경우에만 API에서 `JsonDocument`를 반환하세요. 대부분의 시나리오에서는 이렇게 할 필요가 없습니다. 호출자가 전체 JSON 문서를 사용해야 하는 경우 JsonElement인 RootElement의 Clone을 반환하세요. 호출자가 JSON 문서 내의 특정 요소를 사용해야 하는 경우 해당 JsonElement의 Clone을 반환하세요. `Clone`을 만들지 않고 `RootElement` 또는 하위 요소를 직접 반환하면 해당 소유권을 가진 `JsonDocument`가 폐기된 후에 반환되는 `JsonElement`에 호출자가 액세스할 수 없습니다.

다음은 `Clone`을 만들어야 하는 예제입니다.

C#

```
public JsonElement LookAndLoad(JsonElement source)
{
    string json =
File.ReadAllText(source.GetProperty("fileName").GetString());

    using (JsonDocument doc = JsonDocument.Parse(json))
    {
        return doc.RootElement.Clone();
    }
}
```

위의 코드에는 `fileName` 속성을 포함하는 `JsonElement`가 필요합니다. 위의 코드는 JSON 파일을 열고 `JsonDocument`를 만듭니다. 이 메서드는 호출자가 전체 문서를 사용하려 한다고 가정하고 `RootElement`의 `Clone`을 반환합니다.

`JsonElement`를 수신하고 하위 요소를 반환하는 경우에는 하위 요소의 `Clone`을 반환할 필요가 없습니다. 호출자는 전달된 `JsonElement`가 속한 `JsonDocument`를 활성 상태로 유지해야 합니다. 예를 들어:

C#

```
public JsonElement ReturnFileName(JsonElement source)
{
   return source.GetProperty("fileName");
}
```

`JsonDocument`(`JsonSerializerOptions` 사용)

`JsonSerializer`를 사용하여 `JsonDocument` 인스턴스를 직렬화하고 역직렬화할 수 있습니다. 그러나 `JsonSerializer`를 사용하여 `JsonDocument` 인스턴스를 읽고 쓰기 위한 구현은 JsonDocument.ParseValue(Utf8JsonReader) 및 JsonDocument.WriteTo(Utf8JsonWriter)에 대한 래퍼입니다. 이 래퍼는 `Utf8JsonReader` 또는 `Utf8JsonWriter`에 `JsonSerializerOptions`(직렬 변환기 기능)를 전달하지 않습니다. 예를 들어 JsonSerializerOptions.DefaultIgnoreCondition을 WhenWritingNull로 설정하고 `JsonSerializerOptions`를 사용하는 오버로드를 사용하여 `JsonSerializer`를 호출하는 경우 null 속성은 무시되지 않습니다.

다음 예제에서는 `JsonSerializerOptions` 매개 변수를 사용하고 `JsonDocument` 인스턴스를 직렬화하는 메서드를 사용한 결과를 보여줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace JsonDocumentWithJsonSerializerOptions;

public class Program
{
    public static void Main()
    {
        Person person = new() { Name = "Nancy" };

        // Default serialization - Address property included with null
token.
        // Output: {"Name":"Nancy","Address":null}
        string personJsonWithNull = JsonSerializer.Serialize(person);
        Console.WriteLine(personJsonWithNull);

        // Serialize and ignore null properties - null Address property is
omitted
        // Output: {"Name":"Nancy"}
        JsonSerializerOptions options = new()
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        };
        string personJsonWithoutNull = JsonSerializer.Serialize(person,
```

```
options);
        Console.WriteLine(personJsonWithoutNull);

        // Ignore null properties doesn't work when serializing JsonDocument
instance
        // by using JsonSerializer.
        // Output: {"Name":"Nancy","Address":null}
        JsonDocument? personJsonDocument =
JsonSerializer.Deserialize<JsonDocument>(personJsonWithNull);
        personJsonWithNull = JsonSerializer.Serialize(personJsonDocument,
options);
        Console.WriteLine(personJsonWithNull);
    }
}
public class Person
{
    public string? Name { get; set; }
    public string? Address { get; set; }
}
```

`JsonSerializerOptions`의 기능이 필요한 경우 `JsonDocument` 대신 강력한 형식의 대상(예: 이 예제의 `Person` 클래스)과 함께 `JsonSerializer`를 사용합니다.

참고 항목

- System.Text.Json 개요
- System.Text.Json API 참조
- System.Text.Json.Serialization API 참조

에서 Utf8JsonWriter를 사용하는 방법 System.Text.Json

아티클 • 2023. 05. 26.

이 문서에서는 사용자 지정 직렬 변환기를 빌드하는 데 형식을 사용하는 Utf8JsonWriter 방법을 보여 줍니다.

Utf8JsonWriter은 `String`, `Int32` 및 `DateTime`과 같은 일반적인 .NET 형식에서 UTF-8 인코딩 JSON 텍스트를 쓸 수 있는 고성능 방법을 제공합니다. writer는 사용자 지정 직렬 변환기를 빌드하는 데 사용할 수 있는 하위 수준 형식입니다. JsonSerializer.Serialize 메서드는 내부적으로 `Utf8JsonWriter`를 사용합니다.

다음 예제에서는 Utf8JsonWriter 클래스 사용 방법을 보여줍니다.

C#

```
var options = new JsonWriterOptions
{
    Indented = true
};

using var stream = new MemoryStream();
using var writer = new Utf8JsonWriter(stream, options);

writer.WriteStartObject();
writer.WriteString("date", DateTimeOffset.UtcNow);
writer.WriteNumber("temp", 42);
writer.WriteEndObject();
writer.Flush();

string json = Encoding.UTF8.GetString(stream.ToArray());
Console.WriteLine(json);
```

UTF-8 텍스트를 사용하여 쓰기

`Utf8JsonWriter`를 사용할 때 가능한 최상의 성능을 얻으려면 UTF-16 문자열이 아닌 UTF-8 텍스트로 이미 인코드된 JSON 페이로드를 쓰세요. UTF-16 문자열 리터럴을 사용하는 대신 JsonEncodedText를 사용하여 알려진 문자열 속성 이름 및 값을 정적으로 캐시 및 미리 인코딩하여 작성기에 전달하세요. 이 방법이 UTF-8 바이트 배열을 캐시하여 사용하는 것보다 빠릅니다.

이 방법은 사용자 지정 이스케이프를 수행해야 하는 경우에도 통합니다.
`System.Text.Json`은 문자열을 작성하는 동안 이스케이프를 해제할 수 없습니다. 그러나

사용자 지정 JavaScriptEncoder를 작성기에 옵션으로 전달하거나, 자체 `JavascriptEncoder`를 사용하여 이스케이프를 수행하는 고유한 `JsonEncodedText`를 만든 후 문자열 대신 `JsonEncodedText`를 작성할 수 있습니다. 자세한 내용은 문자 인코딩 사용자 지정을 참조하세요.

원시 JSON 작성

일부 시나리오에서는 `Utf8JsonWriter`를 사용하여 만드는 JSON 페이로드에 '원시' JSON을 작성할 수 있습니다. Utf8JsonWriter.WriteRawValue를 사용하여 이 작업을 수행할 수 있습니다. 일반적인 시나리오는 다음과 같습니다.

- 새 JSON에 묶을 기존 JSON 페이로드가 있습니다.
- 값의 서식을 기본 `Utf8JsonWriter` 서식과 다르게 지정하려고 합니다.

  예를 들어, 숫자 서식을 필요에 따라 사용자 지정할 수 있습니다. 기본적으로 System.Text.Json은 정수에 대한 소수점을 생략합니다. 예를 들어 `1.0`이 아닌 `1`을 작성합니다. 그 이유는 더 적은 바이트를 쓰는 것이 성능에 좋다는 것입니다. 그러나 JSON의 소비자가 10진수가 있는 숫자를 double로 처리하고 소수점이 없는 숫자를 정수로 처리한다고 가정해 보겠습니다. 정수에 대해 소수점과 0을 작성하여 배열의 숫자가 모두 double로 인식되도록 할 수 있습니다. 다음 예제에서는 해당 작업을 수행하는 방법을 보여줍니다.

```C#
using System.Text;
using System.Text.Json;

namespace WriteRawJson;

public class Program
{
    public static void Main()
    {
        JsonWriterOptions writerOptions = new() { Indented = true, };

        using MemoryStream stream = new();
        using Utf8JsonWriter writer = new(stream, writerOptions);

        writer.WriteStartObject();

        writer.WriteStartArray("defaultJsonFormatting");
        foreach (double number in new double[] { 50.4, 51 })
        {
            writer.WriteStartObject();
            writer.WritePropertyName("value");
            writer.WriteNumberValue(number);
```

```
            writer.WriteEndObject();
        }
        writer.WriteEndArray();

        writer.WriteStartArray("customJsonFormatting");
        foreach (double result in new double[] { 50.4, 51 })
        {
            writer.WriteStartObject();
            writer.WritePropertyName("value");
            writer.WriteRawValue(
                FormatNumberValue(result), skipInputValidation: true);
            writer.WriteEndObject();
        }
        writer.WriteEndArray();

        writer.WriteEndObject();
        writer.Flush();

        string json = Encoding.UTF8.GetString(stream.ToArray());
        Console.WriteLine(json);
    }
    static string FormatNumberValue(double numberValue)
    {
        return numberValue == Convert.ToInt32(numberValue) ?
            numberValue.ToString() + ".0" : numberValue.ToString();
    }
}
// output:
//{
//  "defaultJsonFormatting": [
//    {
//      "value": 50.4
//    },
//    {
//      "value": 51
//    }
//  ],
//  "customJsonFormatting": [
//    {
//      "value": 50.4
//    },
//    {
//      "value": 51.0
//    }
//  ]
//}
```

문자 이스케이프 사용자 지정

`JsonTextWriter`의 StringEscapeHandling 설정은 모든 비 ASCII 문자 **또는** HTML 문자를 이스케이프하는 옵션을 제공합니다. 기본적으로 `Utf8JsonWriter`는 모든 비 ASCII 문자 **및**

HTML 문자를 이스케이프합니다. 이러한 이스케이프는 심층 방어 보안을 위해 수행됩니다. 다른 이스케이프 정책을 지정하려면 JavaScriptEncoder를 만들고 JsonWriterOptions.Encoder을 설정하세요. 자세한 내용은 문자 인코딩 사용자 지정을 참조하세요.

null 값 쓰기

`Utf8JsonWriter`를 사용하여 null 값을 쓰려면 다음을 호출합니다.

- WriteNull - Null을 사용하는 키-값 쌍을 값으로 씁니다.
- WriteNullValue - Null을 JSON 배열의 요소로 씁니다.

문자열 속성의 경우 문자열이 Null이면 WriteString 및 WriteStringValue는 `WriteNull` 및 `WriteNullValue`와 동일합니다.

Timespan, Uri 또는 char 값 쓰기

`Timespan`, `Uri` 또는 `char` 값을 작성하려면 (예를 들어 `ToString()`을 호출하여) 문자열로 그 값의 서식을 지정하고 WriteStringValue를 호출합니다.

참고 항목

- 에서 Utf8JsonReader를 사용하는 방법 System.Text.Json

System.Text.Json에서 Utf8JsonReader를 사용하는 방법

이 문서에서는 사용자 지정 파서 및 역직렬 변환기를 빌드하는 데 Utf8JsonReader 형식을 사용하는 방법을 보여 줍니다.

Utf8JsonReader 는 UTF-8로 인코딩된 JSON 텍스트에 대한 고성능, 낮은 할당, 전달 전용 판독기입니다. 텍스트는 a `ReadOnlySpan<byte>` 또는 `ReadOnlySequence<byte>`.에서 읽습니다. `Utf8JsonReader` 는 사용자 지정 파서 및 역직렬 변환기를 빌드하는 데 사용할 수 있는 하위 수준 형식입니다. (메서드는 JsonSerializer.Deserialize 커버 아래에 사용됩니다 `Utf8JsonReader` .)

다음 예제에서는 클래스를 사용하는 Utf8JsonReader 방법을 보여줍니다. 이 코드는 변수가 `jsonUtf8Bytes` UTF-8로 인코딩된 유효한 JSON을 포함하는 바이트 배열이라고 가정합니다.

C#

```
var options = new JsonReaderOptions
{
    AllowTrailingCommas = true,
    CommentHandling = JsonCommentHandling.Skip
};
var reader = new Utf8JsonReader(jsonUtf8Bytes, options);

while (reader.Read())
{
    Console.Write(reader.TokenType);

    switch (reader.TokenType)
    {
        case JsonTokenType.PropertyName:
        case JsonTokenType.String:
            {
                string? text = reader.GetString();
                Console.Write(" ");
                Console.Write(text);
                break;
            }

        case JsonTokenType.Number:
            {
                int intValue = reader.GetInt32();
                Console.Write(" ");
                Console.Write(intValue);
                break;
            }

            // Other token types elided for brevity
    }
```

```
    Console.WriteLine();
}
```

> ⓘ **참고 항목**
>
> `Utf8JsonReader`는 Visual Basic 코드에서 직접 사용할 수 없습니다. 자세한 내용은 Visual Basic 지원을 참조하세요.

`Utf8JsonReader`를 사용하여 데이터 필터링

다음 예제는 파일을 동기적으로 읽고 값을 검색하는 방법을 보여줍니다.

C#

```
using System.Text;
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class Utf8ReaderFromFile
    {
        private static readonly byte[] s_nameUtf8 = Encoding.UTF8.GetBytes("name");
        private static ReadOnlySpan<byte> Utf8Bom => new byte[] { 0xEF, 0xBB, 0xBF
};

        public static void Run()
        {
            // ReadAllBytes if the file encoding is UTF-8:
            string fileName = "UniversitiesUtf8.json";
            ReadOnlySpan<byte> jsonReadOnlySpan = File.ReadAllBytes(fileName);

            // Read past the UTF-8 BOM bytes if a BOM exists.
            if (jsonReadOnlySpan.StartsWith(Utf8Bom))
            {
                jsonReadOnlySpan = jsonReadOnlySpan.Slice(Utf8Bom.Length);
            }

            // Or read as UTF-16 and transcode to UTF-8 to convert to a
ReadOnlySpan<byte>
            //string fileName = "Universities.json";
            //string jsonString = File.ReadAllText(fileName);
            //ReadOnlySpan<byte> jsonReadOnlySpan =
Encoding.UTF8.GetBytes(jsonString);

            int count = 0;
            int total = 0;

            var reader = new Utf8JsonReader(jsonReadOnlySpan);
```

```
            while (reader.Read())
            {
                JsonTokenType tokenType = reader.TokenType;

                switch (tokenType)
                {
                    case JsonTokenType.StartObject:
                        total++;
                        break;
                    case JsonTokenType.PropertyName:
                        if (reader.ValueTextEquals(s_nameUtf8))
                        {
                            // Assume valid JSON, known schema
                            reader.Read();
                            if (reader.GetString()!.EndsWith("University"))
                            {
                                count++;
                            }
                        }
                        break;
                }
            }
            Console.WriteLine($"{count} out of {total} have names that end with
'University'");
        }
    }
}
```

앞의 코드가 하는 역할은 다음과 같습니다.

- JSON에 개체 배열이 포함되어 있고 각 개체에 문자열 형식의 "name" 속성이 포함될 수 있다고 가정합니다.
- "대학교"로 끝나는 개체 및 "이름" 속성 값의 개수를 계산합니다.
- 파일이 UTF-16으로 인코딩되고 UTF-8로 코드 변환된다고 가정합니다.

  UTF-8로 인코딩된 파일은 다음 코드를 사용하여 `ReadOnlySpan<byte>`로 직접 읽을 수 있습니다.

  C#

  ```
  ReadOnlySpan<byte> jsonReadOnlySpan = File.ReadAllBytes(fileName);
  ```

  파일에 UTF-8 BOM(바이트 순서 표시)이 포함된 경우 제거한 후 바이트를 `Utf8JsonReader`에 전달해야 합니다. 판독기에는 텍스트가 필요하기 때문입니다. 그렇지 않으면 BOM은 잘못된 JSON으로 간주되고, 판독기에서 예외를 throw합니다.

다음은 위의 코드에서 읽을 수 있는 JSON 샘플입니다. 다음과 같이 "4개 이름 중 2개가 '대학교'로 끝나는 이름"이라는 결과 요약 메시지가 표시됩니다.

JSON


```
[
  {
    "web_pages": [ "https://contoso.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "contoso.edu" ],
    "name": "Contoso Community College"
  },
  {
    "web_pages": [ "http://fabrikam.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "fabrikam.edu" ],
    "name": "Fabrikam Community College"
  },
  {
    "web_pages": [ "http://www.contosouniversity.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "contosouniversity.edu" ],
    "name": "Contoso University"
  },
  {
    "web_pages": [ "http://www.fabrikamuniversity.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "fabrikamuniversity.edu" ],
    "name": "Fabrikam University"
  }
]
```


> **팁**
>
> 이 예제의 비동기 버전은 .NET 샘플 JSON 프로젝트 를 참조하세요.

`Utf8JsonReader`를 사용하여 스트림에서 읽기

큰 파일(예: 크기가 기가바이트 이상)을 읽을 때 전체 파일을 한 번에 메모리에 로드하지 않도록 할 수 있습니다. 이 시나리오에서는 FileStream을 사용할 수 있습니다.

`Utf8JsonReader`를 사용하여 스트림에서 읽는 경우 다음 규칙이 적용됩니다.

- 부분 JSON 페이로드를 포함하는 버퍼는 판독기가 진행할 수 있도록 최소한 그 안에 있는 가장 큰 JSON 토큰만큼 커야 합니다.
- 버퍼는 최소한 JSON 내에서 가장 큰 공백 시퀀스만큼 커야 합니다.
- 판독기는 JSON 페이로드의 다음 TokenType을 완전히 읽을 때까지는 읽은 데이터를 추적하지 않습니다. 따라서 버퍼에 바이트가 남아 있으면 판독기에 다시 전달해야 합니다. BytesConsumed를 사용하여 남은 바이트 수를 확인할 수 있습니다.

다음 코드에서는 스트림에서 읽는 방법을 보여 줍니다. 이 예제에서는 MemoryStream을 보여 줍니다. FileStream이 시작 시 UTF-8 BOM을 포함하는 경우를 제외하고는 `FileStream`에서 비슷한 코드가 작동합니다. 이 경우 남은 바이트를 `Utf8JsonReader`에 전달하기 전에 버퍼에서 3바이트를 제거해야 합니다. 그렇지 않으면 BOM이 JSON의 유효한 부분으로 간주되지 않으므로 판독기가 예외를 throw합니다.

이 샘플 코드는 4KB 버퍼로 시작하며 크기가 작아 전체 JSON 토큰을 수용할 수 없을 때마다 버퍼 크기를 두 배로 늘립니다. 이는 판독기가 JSON 페이로드에 대한 작업을 진행하는 데 필요합니다. 이 코드 조각에 제공된 JSON 샘플은 10바이트와 같이 매우 작은 초기 버퍼 크기를 설정하는 경우에만 버퍼 크기 증가를 트리거합니다. 초기 버퍼 크기를 10으로 설정하는 경우 `Console.WriteLine` 문은 버퍼 크기 증가의 원인과 영향을 보여 줍니다. 4KB 초기 버퍼 크기에서 전체 샘플 JSON은 각 호출에 `Console.WriteLine`의해 표시되며 버퍼 크기를 늘릴 필요가 없습니다.

C#

```
using System.Text;
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class Utf8ReaderPartialRead
    {
        public static void Run()
        {
            var jsonString = @"{
                ""Date"": ""2019-08-01T00:00:00-07:00"",
                ""Temperature"": 25,
                ""TemperatureRanges"": {
                    ""Cold"": { ""High"": 20, ""Low"": -10 },
                    ""Hot"": { ""High"": 60, ""Low"": 20 }
                },
                ""Summary"": ""Hot"",
            }";

            byte[] bytes = Encoding.UTF8.GetBytes(jsonString);
            var stream = new MemoryStream(bytes);

            var buffer = new byte[4096];

            // Fill the buffer.
```

```
            // For this snippet, we're assuming the stream is open and has data.
            // If it might be closed or empty, check if the return value is 0.
            stream.Read(buffer);

            // We set isFinalBlock to false since we expect more data in a
subsequent read from the stream.
            var reader = new Utf8JsonReader(buffer, isFinalBlock: false, state:
default);
            Console.WriteLine($"String in buffer is:
{Encoding.UTF8.GetString(buffer)}");

            // Search for "Summary" property name
            while (reader.TokenType != JsonTokenType.PropertyName ||
!reader.ValueTextEquals("Summary"))
            {
                if (!reader.Read())
                {
                    // Not enough of the JSON is in the buffer to complete a read.
                    GetMoreBytesFromStream(stream, ref buffer, ref reader);
                }
            }

            // Found the "Summary" property name.
            Console.WriteLine($"String in buffer is:
{Encoding.UTF8.GetString(buffer)}");
            while (!reader.Read())
            {
                // Not enough of the JSON is in the buffer to complete a read.
                GetMoreBytesFromStream(stream, ref buffer, ref reader);
            }
            // Display value of Summary property, that is, "Hot".
            Console.WriteLine($"Got property value: {reader.GetString()}");
        }

        private static void GetMoreBytesFromStream(
            MemoryStream stream, ref byte[] buffer, ref Utf8JsonReader reader)
        {
            int bytesRead;
            if (reader.BytesConsumed < buffer.Length)
            {
                ReadOnlySpan<byte> leftover =
buffer.AsSpan((int)reader.BytesConsumed);

                if (leftover.Length == buffer.Length)
                {
                    Array.Resize(ref buffer, buffer.Length * 2);
                    Console.WriteLine($"Increased buffer size to {buffer.Length}");
                }

                leftover.CopyTo(buffer);
                bytesRead = stream.Read(buffer.AsSpan(leftover.Length));
            }
            else
            {
                bytesRead = stream.Read(buffer);
```

```
            }
            Console.WriteLine($"String in buffer is:
{Encoding.UTF8.GetString(buffer)}");
            reader = new Utf8JsonReader(buffer, isFinalBlock: bytesRead == 0,
reader.CurrentState);
        }
    }
}
```

앞의 예제에서는 버퍼 크기를 늘릴 수 있는 크기 제한을 설정하지 않습니다. 토큰 크기가 너무 클 경우 코드는 OutOfMemoryException 예외와 함께 실패할 수 있습니다. 이 문제는 JSON이 크기가 약 1GB 이상인 토큰을 포함하는 경우 발생할 수 있습니다. 1GB 크기가 두 배가 되면 `int32` 버퍼에 비해 너무 커지기 때문입니다.

ref 구조체 제한 사항

형식이 `Utf8JsonReader` `ref struct`이기 때문에 특정 한정이 있습니다. 다른 클래스 또는 구조체가 아닌 `ref struct`에 필드로 저장할 수 없습니다.

고성능 `Utf8JsonReader`을 달성하려면 `ref struct`이어야 합니다. 입력 ReadOnlySpan<byte>(그 자체로 `ref struct`)를 캐시해야 하기 때문입니다. 또한 이 `Utf8JsonReader` 형식은 상태를 유지하기 때문에 변경 가능합니다. 따라서 값이 아닌 **참조로 전달**해야 합니다. `Utf8JsonReader` by 값을 전달하면 구조체 복사본이 생성되고 상태 변경 내용이 호출자에게 표시되지 않습니다.

ref 구조체를 사용하는 방법에 대한 자세한 내용은 할당 방지를 참조하세요.

UTF-8 텍스트 읽기

`Utf8JsonReader`를 사용할 때 가능한 최상의 성능을 얻으려면 UTF-16 문자열이 아닌 UTF-8 텍스트로 이미 인코딩된 JSON 페이로드를 읽으세요. 코드 예제는 Utf8JsonReader를 사용하여 데이터 필터링을 참조하세요.

다중 세그먼트로 ReadOnlySequence 읽기

JSON 입력이 ReadOnlySpan<byte>이면 읽기 루프를 진행할 때 판독기의 `ValueSpan` 속성에서 각 JSON 요소에 액세스할 수 있습니다. 그러나 입력이 ReadOnlySequence<byte>(PipeReader에서 읽은 결과)인 경우에는 일부 JSON 요소가 `ReadOnlySequence<byte>` 개체의 여러 세그먼트에 걸쳐 있을 수 있습니다. 이러한 요소는 연속 메모리 블록의 ValueSpan에서 액세스할 수 없습니다. 그 대신, 다중 세그먼트 `ReadOnlySequence<byte>`가 입력으로 사용될 때마다 판독기에서 HasValueSequence 속성을 폴링하여 현재 JSON 요소에 액세스하는 방법을 확인하세요. 다음은 권장 패턴입니다.

C#

```
while (reader.Read())
{
    switch (reader.TokenType)
    {
        // ...
        ReadOnlySpan<byte> jsonElement = reader.HasValueSequence ?
            reader.ValueSequence.ToArray() :
            reader.ValueSpan;
        // ...
    }
}
```

여러 JSON 문서 읽기

.NET 9 이상 버전에서는 단일 버퍼 또는 스트림에서 공백으로 구분된 여러 JSON 문서를 읽을 수 있습니다. 기본적으로 `Utf8JsonReader`는 첫 번째 최상위 문서 뒤에 오는 공백이 아닌 문자를 감지할 경우 예외를 발생시킵니다. 그러나 JsonReaderOptions.AllowMultipleValues 플래그를 사용하여 해당 동작을 구성할 수 있습니다.

C#

```
JsonReaderOptions options = new() { AllowMultipleValues = true };
Utf8JsonReader reader = new("null {} 1 \r\n [1,2,3]"u8, options);

reader.Read();
Console.WriteLine(reader.TokenType); // Null

reader.Read();
Console.WriteLine(reader.TokenType); // StartObject
reader.Skip();

reader.Read();
Console.WriteLine(reader.TokenType); // Number

reader.Read();
Console.WriteLine(reader.TokenType); // StartArray
reader.Skip();

Console.WriteLine(reader.Read()); // False
```

AllowMultipleValues가 `true`로 설정되면, 잘못된 JSON인 후행 데이터를 포함하는 페이로드에서도 JSON을 읽을 수 있습니다.

C#

```
JsonReaderOptions options = new() { AllowMultipleValues = true };
Utf8JsonReader reader = new("[1,2,3]    <NotJson/>"u8, options);

reader.Read();
reader.Skip(); // Succeeds.
reader.Read(); // Throws JsonReaderException.
```

여러 최상위 값을 스트리밍하려면 DeserializeAsyncEnumerable<TValue>(Stream, Boolean, JsonSerializerOptions, CancellationToken) 오버로드 또는 DeserializeAsyncEnumerable<TValue>(Stream, JsonTypeInfo<TValue>, Boolean, CancellationToken) 오버로드를 사용하십시오. 기본적으로 `DeserializeAsyncEnumerable` 단일 최상위 JSON 배열에 포함된 요소를 스트리밍하려고 시도합니다. `topLevelValues` 매개 변수에 `true`를 사용하여 여러 최상위 값을 스트리밍합니다.

C#

```
ReadOnlySpan<byte> utf8Json = """[0] [0,1] [0,1,1] [0,1,1,2] [0,1,1,2,3]"""u8;
using var stream = new MemoryStream(utf8Json.ToArray());

var items = JsonSerializer.DeserializeAsyncEnumerable<int[]>(stream,
topLevelValues: true);
await foreach (int[] item in items)
{
    Console.WriteLine(item.Length);
}

/* This snippet produces the following output:
 *
 * 1
 * 2
 * 3
 * 4
 * 5
 */
```

속성 이름 조회

속성 이름을 조회할 때, 바이트별로 비교하기 위해 ValueSpan를 사용하지 마세요. SequenceEqual을 호출하지 마세요. 대신 ValueTextEquals을 호출하십시오. 이 메서드는 JSON에서 이스케이프된 문자를 모두 복원합니다. 다음은 “name”이라는 속성을 검색하는 방법을 보여주는 예제입니다.

C#

```
private static readonly byte[] s_nameUtf8 = Encoding.UTF8.GetBytes("name");
```

C#

```
while (reader.Read())
{
    switch (reader.TokenType)
    {
        case JsonTokenType.StartObject:
            total++;
            break;
        case JsonTokenType.PropertyName:
            if (reader.ValueTextEquals(s_nameUtf8))
            {
                count++;
            }
            break;
    }
}
```

Null 값을 null 허용 값 형식으로 읽기

기본 제공 `System.Text.Json` API는 null을 허용하지 않는 값 형식만 반환합니다. 예를 들어 Utf8JsonReader.GetBoolean은 `bool`을 반환합니다. JSON에서 `Null`을 발견하면 예외를 throw합니다. 다음 예제에서는 Null을 처리하는 두 가지 방법을 보여줍니다. 하나는 null 허용 값 형식을 반환하는 방법이고, 다른 하나는 기본값을 반환하는 방법입니다.

C#

```
public bool? ReadAsNullableBoolean()
{
    _reader.Read();
    if (_reader.TokenType == JsonTokenType.Null)
    {
        return null;
    }
    if (_reader.TokenType != JsonTokenType.True && _reader.TokenType !=
JsonTokenType.False)
    {
        throw new JsonException();
    }
    return _reader.GetBoolean();
}
```

C#

```
public bool ReadAsBoolean(bool defaultValue)
{
    _reader.Read();
    if (_reader.TokenType == JsonTokenType.Null)
    {
```

```
        return defaultValue;
    }
    if (_reader.TokenType != JsonTokenType.True && _reader.TokenType !=
JsonTokenType.False)
    {
        throw new JsonException();
    }
    return _reader.GetBoolean();
}
```

토큰의 자식 건너뛰기

현재 JSON 토큰의 자식을 건너뛰려고 Utf8JsonReader.Skip() 메서드를 사용합니다. 토큰이 JsonTokenType.PropertyName 형식이면 판독기는 속성 값으로 이동합니다. 다음 코드 조각에서는 판독기를 속성 값으로 이동하는 데 Utf8JsonReader.Skip()을 사용하는 예제를 보여 줍니다.

C#

```
var weatherForecast = new WeatherForecast
{
    Date = DateTime.Parse("2019-08-01"),
    TemperatureCelsius = 25,
    Summary = "Hot"
};

byte[] jsonUtf8Bytes = JsonSerializer.SerializeToUtf8Bytes(weatherForecast);

var reader = new Utf8JsonReader(jsonUtf8Bytes);

int temp;
while (reader.Read())
{
    switch (reader.TokenType)
    {
        case JsonTokenType.PropertyName:
            {
                if (reader.ValueTextEquals("TemperatureCelsius"))
                {
                    reader.Skip();
                    temp = reader.GetInt32();

                    Console.WriteLine($"Temperature is {temp} degrees.");
                }
                continue;
            }
        default:
            continue;
    }
}
```

디코딩된 JSON 문자열 사용

.NET 7부터는 디코딩된 JSON 문자열을 사용하기 위해 Utf8JsonReader.CopyString 대신 Utf8JsonReader.GetString() 메서드를 사용할 수 있습니다. 항상 새 문자열을 할당하는 GetString()과 달리, CopyString은 현재 소유한 버퍼에 이스케이프되지 않은 문자열을 복사할 수 있습니다. 다음 코드 조각은 CopyString을 사용하여 UTF-16 문자열을 사용하는 예제를 보여줍니다.

C#

```
var reader = new Utf8JsonReader( /* jsonReadOnlySpan */ );

int valueLength = reader.HasValueSequence
    ? checked((int)reader.ValueSequence.Length)
    : reader.ValueSpan.Length;

char[] buffer = ArrayPool<char>.Shared.Rent(valueLength);
int charsRead = reader.CopyString(buffer);
ReadOnlySpan<char> source = buffer.AsSpan(0, charsRead);

// Handle the unescaped JSON string.
ParseUnescapedString(source);
ArrayPool<char>.Shared.Return(buffer, clearArray: true);

void ParseUnescapedString(ReadOnlySpan<char> source)
{
    // ...
}
```

관련 API

- `Utf8JsonReader` 인스턴스에서 사용자 지정 형식을 역직렬화하려면 JsonSerializer.Deserialize<TValue>(Utf8JsonReader, JsonSerializerOptions)또는 JsonSerializer.Deserialize<TValue>(Utf8JsonReader, JsonTypeInfo<TValue>)를 호출합니다. 예를 들어 UTF-8에서 역직렬화를 참조하세요.

- JsonNode과 그로부터 파생되는 클래스는 변경 가능한 DOM을 만드는 기능을 제공합니다. `Utf8JsonReader`를 호출하여 `JsonNode` 인스턴스를 JsonNode.Parse(Utf8JsonReader, Nullable<JsonNodeOptions>)로 변환할 수 있습니다. 다음 코드 조각은 예제를 보여 줍니다.

  C#

  ```
  using System.Text.Json;
  using System.Text.Json.Nodes;
  ```

```
namespace Utf8ReaderToJsonNode
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            byte[] jsonUtf8Bytes =
JsonSerializer.SerializeToUtf8Bytes(weatherForecast);

            var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
            JsonNode? node = JsonNode.Parse(ref utf8Reader);
            Console.WriteLine(node);
        }
    }
}
```

- JsonDocument는 `Utf8JsonReader`를 사용하여 읽기 전용 DOM을 빌드하는 기능을 제공합니다. `Utf8JsonReader` 인스턴스에서 `JsonDocument`을(를) 구문 분석하기 위해 JsonDocument.ParseValue(Utf8JsonReader) 메서드를 호출합니다. 페이로드를 구성하는 JSON 요소는 JsonElement 형식을 통해 액세스할 수 있습니다. JsonDocument.ParseValue(Utf8JsonReader)를 사용하는 예제 코드는 RoundtripDataTable.cs 및 유추 형식을 개체 속성으로 역직렬화의 코드 조각을 참조하세요.

- JsonElement.ParseValue(Utf8JsonReader)를 호출하여 `Utf8JsonReader` 인스턴스를 특정 JSON 값을 나타내는 JsonElement로 구문 분석할 수도 있습니다.

참조 항목

- System.Text.Json에서 Utf8JsonWriter를 사용하는 방법

Last updated on 2026. 03. 26.

Visual Basic 지원

System.Text.Json의 일부는 Visual Basic에서 지원되지 않는 ref struct를 사용합니다. Visual Basic에서 ref 구조체 API를 사용하려고 하면 System.Text.Json 컴파일러 오류 BC40000이 발생합니다. 이 오류 메시지는 문제가 사용되지 않는 API임을 나타내지만 실제 문제는 컴파일러에서 ref struct 지원이 부족하다는 것입니다. System.Text.Json의 다음 부분은 Visual Basic에서 사용할 수 없습니다.

- Utf8JsonReader 구조체입니다. JsonConverter<T>.Read 메서드는 `Utf8JsonReader` 매개 변수를 사용하므로 이 제한은 Visual Basic을 사용하여 사용자 지정 변환기를 작성할 수 없다는 것을 의미합니다. 이에 대한 해결 방법은 C# 라이브러리 어셈블리에서 사용자 지정 변환기를 구현하고 VB 프로젝트에서 해당 어셈블리를 참조하는 것입니다. 여기서는 Visual Basic에서는 변환기를 직렬 변환기에 등록하기만 하면 된다고 가정합니다. Visual Basic 코드에서 변환기의 `Read` 메서드를 호출할 수 없습니다.
- ReadOnlySpan<T> 형식을 포함하는 다른 API의 오버로드입니다. 대부분의 메서드에는 `String` 대신 `ReadOnlySpan`을 사용하는 오버로드가 포함됩니다.

참조 구조체는 "데이터 전달"만 하는 경우에도 언어 지원 없이는 안전하게 사용할 수 없으므로 이러한 제한이 적용됩니다. 이 오류를 전복해서는 안 됩니다. 이렇게 하면 Visual Basic 코드가 메모리를 손상할 수 있습니다.

Last updated on 2026. 02. 24.

System.Text.Json 지원되는 형식

아티클 • 2024. 12. 19.

이 문서에서는 serialization 및 deserialization에 지원되는 형식에 대한 개요를 제공합니다.

JSON 개체로 직렬화하는 형식

다음 형식은 JSON 개체로 직렬화합니다.

- 클래스*
- 구조체
- 인터페이스
- 레코드 및 구조체 레코드

* JSON 배열로 직렬화할 IEnumerable<T> 구현하는 사전이 아닌 형식입니다. IEnumerable<T>구현하는 사전 형식은 JSON 개체로 직렬화됩니다.

다음 코드 조각은 간단한 구조체의 serialization을 보여줍니다.

C#

```
public static void Main()
{
    var coordinates = new Coords(1.0, 2.0);
    string json = JsonSerializer.Serialize(coordinates);
    Console.WriteLine(json);

    // Output:
    // {"X":1,"Y":2}
}

public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }
}
```

JSON 배열로 직렬화하는 형식

.NET 컬렉션 형식은 JSON 배열로 직렬화됩니다. System.Text.Json.JsonSerializer 다음과 같은 경우 serialization을 위한 컬렉션 형식을 지원합니다.

- IEnumerable 또는 IAsyncEnumerable<T>파생됩니다.
- serialize할 수 있는 요소를 포함합니다.

serializer는 GetEnumerator() 메서드를 호출하고 요소를 씁니다.

역직렬화는 더 복잡하며 일부 컬렉션 형식에서는 지원되지 않습니다.

다음 섹션은 네임스페이스별로 구성되며 직렬화 및 역직렬화에 지원되는 형식을 보여 있습니다.

- System.Array 네임스페이스
- System.Collections 네임스페이스
- System.Collections.Generic 네임스페이스
- system.Collections.Immutable 네임스페이스
- System.Collections.Specialized 네임스페이스
- system.Collections.Concurrent 네임스페이스
- System.Collections.ObjectModel 네임스페이스
- 사용자 지정 컬렉션

System.Array 네임스페이스

테이블 확장

형	직렬화	역직렬화
1차원 배열	✔	✔
다차원 배열	✘	✘
가변 배열	✔	✔

System.Collections 네임스페이스

테이블 확장

형	직렬화	역직렬화
ArrayList	✔	✔
BitArray	✔	✘
DictionaryEntry	✔	✔

형	직렬화	역직렬화
Hashtable	✔	✔
ICollection	✔	✔
IDictionary	✔	✔
IEnumerable	✔	✔
IList	✔	✔
Queue	✔	✔
SortedList	✔	✔
Stack *	✔	✔

* 유형에 대한 지원 왕복을 참조하세요.

System.Collections.Generic 네임스페이스

테이블 확장

형	직렬화	역직렬화
Dictionary<TKey,TValue> *	✔	✔
HashSet<T>	✔	✔
IAsyncEnumerable<T> †	✔	✔
ICollection<T>	✔	✔
IDictionary<TKey,TValue> *	✔	✔
IEnumerable<T>	✔	✔
IList<T>	✔	✔
IReadOnlyCollection<T>	✔	✔
IReadOnlyDictionary<TKey,TValue> *	✔	✔
IReadOnlyList<T>	✔	✔
ISet<T>	✔	✔
KeyValuePair<TKey,TValue>	✔	✔
LinkedList<T>	✔	✔

형	직렬화	역직렬화
LinkedListNode<T>	✔	✘
List<T>	✔	✔
Queue<T>	✔	✔
SortedDictionary<TKey,TValue> *	✔	✔
SortedList<TKey,TValue> *	✔	✔
SortedSet<T>	✔	✔
Stack<T>	✔	✔

* 지원되는 키 형식참조하세요.

† `IAsyncEnumerable<T>`다음 섹션을 참조하세요.

• 유형은 지원 왕복을 참조하세요.

IAsyncEnumerable<T>

다음 예제에서는 비동기 데이터 원본의 표현으로 스트림을 사용합니다. 원본은 로컬 컴퓨터의 파일이거나 데이터베이스 쿼리 또는 웹 서비스 API 호출의 결과일 수 있습니다.

스트림 serialization

`System.Text.Json` 다음 예제와 같이 IAsyncEnumerable<T> 값을 JSON 배열로 직렬화할 수 있습니다.

C#

```
using System.Text.Json;

namespace IAsyncEnumerableSerialize;

public class Program
{
    public static async Task Main()
    {
        using Stream stream = Console.OpenStandardOutput();
        var data = new { Data = PrintNumbers(3) };
        await JsonSerializer.SerializeAsync(stream, data);
    }

    static async IAsyncEnumerable<int> PrintNumbers(int n)
    {
```

```
        for (int i = 0; i < n; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
}
// output:
//  {"Data":[0,1,2]}
```

`IAsyncEnumerable<T>` 값은 JsonSerializer.SerializeAsync같은 비동기 serialization 메서드에서만 지원됩니다.

스트림 역직렬화

`DeserializeAsyncEnumerable` 메서드는 다음 예제와 같이 스트리밍 역직렬화를 지원합니다.

C#

```
using System.Text;
using System.Text.Json;

namespace IAsyncEnumerableDeserialize;

public class Program
{
    public static async Task Main()
    {
        using var stream = new MemoryStream(Encoding.UTF8.GetBytes("
[0,1,2,3,4]"));
        await foreach (int item in
JsonSerializer.DeserializeAsyncEnumerable<int>(stream))
        {
            Console.WriteLine(item);
        }
    }
}
// output:
//0
//1
//2
//3
//4
```

`DeserializeAsyncEnumerable` 메서드는 루트 수준 JSON 배열에서 읽기만 지원합니다.

DeserializeAsync 메서드는 `IAsyncEnumerable<T>`지원하지만 해당 서명은 스트리밍을 허용하지 않습니다. 다음 예제와 같이 최종 결과를 단일 값으로 반환합니다.

C#

```
using System.Text;
using System.Text.Json;

namespace IAsyncEnumerableDeserializeNonStreaming;

public class MyPoco
{
    public IAsyncEnumerable<int>? Data { get; set; }
}

public class Program
{
    public static async Task Main()
    {
        using var stream = new MemoryStream(Encoding.UTF8.GetBytes(@"
{""Data"":[0,1,2,3,4]}"));
        MyPoco? result = await JsonSerializer.DeserializeAsync<MyPoco>
(stream)!;
        await foreach (int item in result!.Data!)
        {
            Console.WriteLine(item);
        }
    }
}
// output:
//0
//1
//2
//3
//4
```

이 예제에서 역직렬 변환기는 역직렬화된 개체를 반환하기 전에 메모리의 모든 `IAsyncEnumerable<T>` 콘텐츠를 버퍼링합니다. 역직렬 변환기는 결과를 반환하기 전에 전체 JSON 페이로드를 읽어야 하기 때문에 이 동작이 필요합니다.

System.Collections.Immutable 네임스페이스

테이블 확장

형	직렬화	역직렬화
IImmutableDictionary<TKey,TValue> †	✔	✔
IImmutableList<T>	✔	✔
IImmutableQueue<T>	✔	✔
IImmutableSet<T>	✔	✔

형	직렬화	역직렬화
IImmutableStack<T> *	✔	✔
ImmutableArray<T>	✔	✔
ImmutableDictionary<TKey,TValue> †	✔	✔
ImmutableHashSet<T>	✔	✔
ImmutableQueue<T>	✔	✔
ImmutableSortedDictionary<TKey,TValue> †	✔	✔
ImmutableSortedSet<T>	✔	✔
ImmutableStack<T> *	✔	✔

* 유형에 대한 지원 왕복을 참조하세요.

† 지원되는 키 형식참조하세요.

System.Collections.Specialized 네임스페이스

테이블 확장

형	직렬화	역직렬화
BitVector32	✔	✘*
HybridDictionary	✔	✔
IOrderedDictionary	✔	✘
ListDictionary	✔	✔
NameValueCollection	✔	✘
StringCollection	✔	✘
StringDictionary	✔	✘

* BitVector32 역직렬화되면 공용 setter가 없으므로 Data 속성은 건너뜁니다. 예외가 throw되지 않습니다.

System.Collections.Concurrent 네임스페이스

테이블 확장

형	직렬화	역직렬화
BlockingCollection<T>	✔	✘
ConcurrentBag<T>	✔	✘
ConcurrentDictionary<TKey,TValue> †	✔	✔
ConcurrentQueue<T>	✔	✔
ConcurrentStack<T> *	✔	✔

* 유형에 대한 지원 왕복을 참조하세요.

† 지원되는 키 형식참조하세요.

System.Collections.ObjectModel 네임스페이스

테이블 확장

형	직렬화	역직렬화
Collection<T>	✔	✔
KeyedCollection<문자열, TValue> *	✔	✘
ObservableCollection<T>	✔	✔
ReadOnlyCollection<T>	✔	✘
ReadOnlyDictionary<TKey,TValue>	✔	✘
ReadOnlyObservableCollection<T>	✔	✘

*`string` 키가 지원되지 않습니다.

사용자 지정 컬렉션

이전 네임스페이스 중 하나에 없는 컬렉션 형식은 사용자 지정 컬렉션으로 간주됩니다. 이러한 형식에는 ASP.NET Core에서 정의한 사용자 정의 형식 및 형식이 포함되었습니다. 예를 들어 Microsoft.Extensions.Primitives 이 그룹에 있습니다.

요소 형식이 지원되는 한 모든 사용자 지정 컬렉션(`IEnumerable`파생)은 serialization에 대해 지원됩니다.

역직렬화 지원

사용자 지정 컬렉션은 다음과 같은 경우 역직렬화에 대해 지원됩니다.

- 인터페이스 또는 추상이 아닙니다.
- 매개 변수가 없는 생성자가 있습니다.
- JsonSerializer지원되는 요소 형식을 포함합니다.
- 다음 인터페이스 또는 클래스 중 하나 이상을 구현하거나 상속합니다.
  - ConcurrentQueue<T>
  - ConcurrentStack<T> *
  - ICollection<T>
  - IDictionary
  - IDictionary<TKey,TValue> †
  - IList
  - IList<T>
  - Queue
  - Queue<T>
  - Stack *
  - Stack<T> *

  * 유형에 대한 지원 왕복을 참조하세요.

  † 지원되는 키 형식참조하세요.

알려진 문제

다음 사용자 지정 컬렉션에는 알려진 문제가 있습니다.

- ExpandoObject: dotnet/runtime#29690 참조하세요.
- DynamicObject: dotnet/runtime#1808 참조하세요.
- DataTable: dotnet/docs#21366 참조하세요.
- Microsoft.AspNetCore.Http.FormFile: dotnet/runtime#1559 참조하세요.
- Microsoft.AspNetCore.Http.IFormCollection: dotnet/runtime#1559 참조하세요.

알려진 문제에 대한 자세한 내용은 열린 문제를 참조하세요.

지원되는 키 형식

`Dictionary` 및 `SortedList` 형식의 키로 사용되는 경우 다음 형식은 기본 제공 지원을 제공합니다.

- `Boolean`
- `Byte`

- `DateTime`
- `DateTimeOffset`
- `Decimal`
- `Double`
- `Enum`
- `Guid`
- `Int16`
- `Int32`
- `Int64`
- `Object`(serialization에만 해당하며 런타임 형식이 이 목록에서 지원되는 형식 중 하나인 경우에만)
- `SByte`
- `Single`
- `String`
- TimeSpan
- `UInt16`
- `UInt32`
- `UInt64`
- Uri
- Version

또한 JsonConverter<T>.WriteAsPropertyName(Utf8JsonWriter, T, JsonSerializerOptions) 및 JsonConverter<T>.ReadAsPropertyName(Utf8JsonReader, Type, JsonSerializerOptions) 메서드를 사용하면 선택한 모든 형식에 대한 사전 키 지원을 추가할 수 있습니다.

지원되지 않는 형식

serialization에는 다음 형식이 지원되지 않습니다.

- System.Type 및 System.Reflection.MemberInfo
- 일반적으로 ReadOnlySpan<T>, Span<T> 및 ref 구조체
- 대리자 형식
- IntPtr 및 UIntPtr

System.Data 네임스페이스

System.Data 네임스페이스에는 DataSet, DataTable 및 관련 형식에 대한 기본 제공 변환기가 없습니다. 보안 지침설명한 대로 신뢰할 수 없는 입력에서 이러한 형식을 역직렬화하는 것은 안전하지 않습니다. 그러나 이러한 형식을 지원하는 사용자 지정 변환기를 작성

할 수 있습니다. `DataTable` 직렬화하고 역직렬화하는 샘플 사용자 지정 변환기 코드는 RoundtripDataTable.cs 참조하세요.

참고 항목

- 초기화된 속성 채우기
- System.Text.Json 개요
- System.Text.Json API 참조
- System.Text.Json. serialization API 참조

System.Text.Json의 리플렉션과 소스 생성

이 문서에서는 `System.Text.Json` Serialization과 관련하여 리플렉션과 소스 생성 간의 차이점을 설명합니다. 시나리오에 가장 적합한 접근 방식을 선택하는 방법에 대한 지침도 제공합니다.

메타데이터 수집

형식을 직렬화하거나 역직렬화하려면 JsonSerializer에 형식의 멤버에 액세스하는 방법에 대한 정보가 필요합니다. `JsonSerializer`에는 다음 정보가 필요합니다.

- 직렬화를 위해 속성 getter와 필드에 접근하는 방법
- 역직렬화를 위해 생성자, 속성 설정자 및 필드에 접근하는 방법
- 직렬화 또는 역직렬화를 사용자 정의하는 데 사용된 속성에 대한 정보.
- JsonSerializerOptions에서의 런타임 구성

이 정보를 '메타데이터'라고 합니다.

성찰

기본적으로 JsonSerializer리플렉션을 사용하여 런타임 시 메타데이터를 수집합니다. `JsonSerializer`는 처음으로 형식을 직렬화하거나 역직렬화해야 할 때마다 이 메타데이터를 수집하고 캐시합니다. 메타데이터 수집 프로세스는 시간이 걸리고 메모리를 사용합니다.

원본 생성

또는 `System.Text.Json`은 C# 원본 생성 기능을 사용하여 성능을 개선하고, 프라이빗 메모리 사용량을 줄이며, 어셈블리 트리밍을 용이하게 하여 앱 크기를 줄일 수 있습니다. 또한 특정 리플렉션 API는 네이티브 AOT 애플리케이션에서 사용할 수 없으므로 해당 앱에 대한 소스 생성을 사용해야 합니다.

소스 생성은 다음 두 가지 모드로 사용할 수 있습니다.

- **메타데이터 기반 모드**

  컴파일 하는 동안 `System.Text.Json`은 Serialization에 필요한 정보를 수집하고 요청된 형식에 대한 JSON 계약 메타데이터를 채우는 소스 코드 파일을 생성합니다.

- **직렬화 최적화(고속 경로) 모드**

  이름 지정 정책 및 참조 보존과 같은 Serialization의 출력을 사용자 지정하는 JsonSerializer 기능은 성능 오버헤드를 발생시킵니다. Serialization 최적화 모드에서 System.Text.Json은

Utf8JsonWriter를 직접 사용하는 최적화된 Serialization 코드를 생성합니다. 이 최적화된 또는 *빠른 경로* 코드는 직렬화 처리량을 증가시킵니다.

고속 경로 *역직렬화*는 현재 사용할 수 없습니다. 자세한 내용은 dotnet/런타임 문제 55043 을 참조하세요.

`System.Text.Json`의 소스를 생성하려면 C# 9.0 이상 버전이 필요합니다.

기능 비교

각 모드에서 제공하는 다음과 같은 이점에 따라 리플렉션 또는 소스 생성 모드를 선택합니다.

테이블 확장

이점	성찰	원본 생성 (메타데이터 기반 모드)	원본 생성 (Serialization 최적화 모드)
더 간단하게 코딩할 수 있습니다.	✔	✘	✘
더 간단하게 디버그할 수 있습니다.	✘	✔	✔
비공개 멤버를 지원합니다.	✔	✔*	✔*
사용 가능한 모든 serialization 사용자 지정을 지원합니다.	✔	✘ †	✘ †
시작 시간을 줄입니다.	✘	✔	✔
프라이빗 메모리 사용량을 줄입니다.	✘	✔	✔
런타임 리플렉션을 제거합니다.	✘	✔	✔
트리밍 안전 앱 크기를 편리하게 줄일 수 있습니다.	✘	✔	✔
직렬화 처리량을 증가시킵니다.	✘	✘	✔

* 소스 생성기는 *일부* Public이 아닌 멤버(예: 동일한 어셈블리의 내부 형식)를 지원합니다. † 소스 생성 계약은 계약 사용자 지정 API를 사용하여 수정할 수 있습니다.

Last updated on 2025. 12. 05.

System.Text.Json의 원본 생성 모드

원본 생성은 *메타데이터 기반* 및 *serialization 최적화*의 두 가지 모드에서 사용할 수 있습니다. 이 문서에서는 다양한 모드에 대해 설명합니다.

원본 생성 모드를 사용하는 방법에 대한 내용은 System.Text.Json에서 원본 생성을 사용하는 방법을 참조하세요.

메타데이터 기반 모드

원본 생성을 사용하여 메타데이터 수집 프로세스를 런타임에서 컴파일 시간으로 이동할 수 있습니다. 컴파일하는 동안 메타데이터가 수집되고 소스 코드 파일이 생성됩니다. 생성된 소스 코드 파일은 애플리케이션의 필수 부분으로 자동으로 컴파일됩니다. 이 기술은 런타임 메타데이터 컬렉션을 제거하여 serialization 및 deserialization의 성능을 향상시킵니다.

원본 생성을 통해 상당한 성능 향상 효과를 얻을 수 있습니다. 예를 들어 테스트 결과에는 최대 40% 이상의 시작 시간 감소, 프라이빗 메모리 감소, 처리량 속도 증가(serialization 최적화 모드) 및 앱 크기 감소가 표시됩니다.

알려진 문제

기본적으로 `public` 속성 및 필드만 두 가지serialization 모드(리플렉션 또는 소스 생성)에서 지원됩니다. 그러나 리플렉션 모드는 `private` 멤버 사용을 지원하지만 소스 생성 모드에서는 이를 지원하지 않습니다. 예를 들어, JsonInclude 특성을 `private` 특성이나 `private` setter 또는 getter가 있는 속성에 적용하면 리플렉션 모드에서 직렬화됩니다. 원본 생성 모드는 `public` 속성의 `internal` 또는 `public` 멤버와 `internal` 또는 `public` 접근자만 지원합니다. 멤버나 접근자에 `[JsonInclude]`를 설정하고 `private` 소스 생성 모드를 선택하면, 런타임에 `NotSupportedException` 예외가 발생하게 됩니다.

원본 생성과 관련된 다른 알려진 문제에 대한 내용은 dotnet/runtime 리포지토리에서 *"source-generator" 레이블이 지정된 GitHub 이슈*를 참조하세요.

직렬화 최적화(빠른 경로) 모드

`JsonSerializer`에는 명명 정책 및 참조 보존과 같이 직렬화 출력을 사용자 지정하는 많은 기능이 있습니다. 이러한 모든 기능을 지원하면 성능 오버헤드가 발생합니다. 원본 생성은 Utf8JsonWriter를 직접 사용하는 최적화된 코드를 생성하여 serialization 성능을 향상시킬 수 있습니다.

Serialization 최적화 모드는 빠른 경로 serialization 메서드를 내보내지만 serialization 메타데이터는 내보내지 않습니다. 빠른 경로 직렬화는 수행할 수 있는 작업이 제한되어 있습니다. 비동기

직렬화나 역직렬화 모드를 지원하지 않습니다.

또한 최적화된 코드는 `JsonSerializer`이(가) 지원하는 모든 serialization 기능을 지원하지 않습니다. 직렬 변환기는 최적화된 코드를 사용할 수 있는지 여부를 검색하고 지원되지 않는 옵션이 지정된 경우 기본 serialization 코드로 돌아갑니다. 예를 들어, JsonNumberHandling.AllowReadingFromString은 쓰기에 적용할 수 없으므로 이 옵션을 지정해도 기본 코드로 대체되지 않습니다.

다음 표는 빠른 경로 직렬화에서 지원되는 `JsonSerializerOptions`의 옵션을 보여 줍니다.

테이블 확장

직렬화 옵션	빠른 경로 지원
AllowTrailingCommas	✔
Converters	✘
DefaultBufferSize	✔
DefaultIgnoreCondition	✔
DictionaryKeyPolicy	✘
Encoder	✘
IgnoreNullValues	✘
IgnoreReadOnlyFields	✔
IgnoreReadOnlyProperties	✔
IncludeFields	✔
MaxDepth	✔
NumberHandling	✘
PropertyNamingPolicy	✔
ReferenceHandler	✘
TypeInfoResolver	✔
WriteIndented	✔

(다음 옵션은 *역직렬화에만 적용되므로 지원되지 않습니다*: PropertyNameCaseInsensitive, ReadCommentHandling 및 UnknownTypeHandling.)

다음 표는 빠른 경로 직렬화에서 지원되는 특성을 보여 줍니다.

테이블 확장

특성	빠른 경로 지원
JsonConstructorAttribute	✗
JsonConverterAttribute	✗
JsonDerivedTypeAttribute	✓
JsonExtensionDataAttribute	✗
JsonIgnoreAttribute	✓
JsonIncludeAttribute	✓
JsonNumberHandlingAttribute	✗
JsonPolymorphicAttribute	✓
JsonPropertyNameAttribute	✓
JsonPropertyOrderAttribute	✓
JsonRequiredAttribute	✓

지원되지 않는 옵션 또는 특성이 형식에 대해 지정된 경우 serializer는 원본 생성기가 메타데이터를 생성하도록 구성되었다고 가정하여 메타데이터 모드로 돌아갑니다. 이 경우 최적화된 코드는 해당 형식을 직렬화할 때 사용되지 않지만 다른 형식에 사용될 수 있습니다. 따라서 옵션 및 워크로드로 성능 테스트를 수행하여 serialization 최적화 모드에서 실제로 얻을 수 있는 이점을 파악하는 것이 중요합니다. 또한 `JsonSerializer` 코드로 돌아가려면 메타데이터 모드가 필요합니다. serialization 최적화 모드만 선택하면 `JsonSerializer` 코드로 대체해야 하는 형식이나 옵션에 대해 serialization이 실패할 수 있습니다.

참고 항목

- .NET의 JSON serialization 및 deserialization - 개요
- 라이브러리를 사용하는 방법

Last updated on 2026. 02. 24.

System.Text.Json에서 원본 생성을 사용하는 방법

이 문서에서는 앱에서 원본 생성 기반 직렬화System.Text.Json를 사용하는 방법을 보여 줍니다.

다양한 원본 생성 모드에 대한 자세한 내용은 원본 생성 모드를 참조하세요.

원본 생성 기본값 사용

모든 기본값(두 가지 모드, 기본 옵션)에서 원본 생성을 사용하려면 다음을 수행합니다.

1. JsonSerializerContext에서 파생되는 partial 클래스를 만듭니다.
2. 컨텍스트 클래스에 JsonSerializableAttribute를 적용하여 직렬화하거나 역직렬화할 형식을 지정합니다.
3. 다음 중 하나를 수행하는 JsonSerializer 메서드를 호출합니다.
    - JsonTypeInfo<T> 인스턴스를 사용합니다. 또는
    - JsonSerializerContext 인스턴스를 사용합니다. 또는
    - JsonSerializerOptions 인스턴스를 받아 그 JsonSerializerOptions.TypeInfoResolver 속성을 컨텍스트 유형의 `Default` 속성에 설정합니다.

기본적으로 원본 생성 모드(*메타데이터 기반* 및 *serialization 최적화*)는 둘 다 지정하지 않으면 사용됩니다. 사용할 모드를 지정하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 원본 생성 모드 지정을 참조하세요.

다음 예제에서 사용되는 형식은 다음과 같습니다.

C#

```
public class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

다음은 이전 클래스에 대한 원본을 생성하도록 구성된 컨텍스트 클래스입니다 `WeatherForecast`

.

C#

```
[JsonSourceGenerationOptions(WriteIndented = true)]
[JsonSerializable(typeof(WeatherForecast))]
```

```
internal partial class SourceGenerationContext : JsonSerializerContext { }
```

`WeatherForecast` 멤버의 형식은 `[JsonSerializable]` 특성으로 명시적으로 지정할 필요가 없습니다. `object`으로 선언된 멤버는 이 규칙의 예외입니다. `object`로 선언된 멤버의 런타임 형식을 지정해야 합니다. 예를 들어 다음과 같은 클래스가 있다고 가정하겠습니다.

C#

```
public class WeatherForecast
{
    public object? Data { get; set; }
    public List<object>? DataList { get; set; }
}
```

그리고 런타임에 다음과 같은 개체가 있을 `boolean` `int` 수 있다는 것을 알고 있습니다.

C#

```
WeatherForecast wf = new() { Data = true, DataList = [true, 1] };
```

그러면 `boolean` 및 `int`를 `[JsonSerializable]`로 선언해야 합니다.

C#

```
[JsonSerializable(typeof(WeatherForecast))]
[JsonSerializable(typeof(bool))]
[JsonSerializable(typeof(int))]
public partial class WeatherForecastContext : JsonSerializerContext
{
}
```

컬렉션의 원본 생성을 지정하려면 컬렉션 형식과 함께 `[JsonSerializable]`을 사용합니다. 예: `[JsonSerializable(typeof(List<WeatherForecast>))]`

원본 생성을 사용하는 `JsonSerializer` 메서드

다음 예제에서 컨텍스트 형식의 static `Default` 속성은 기본 옵션을 사용하여 컨텍스트 형식의 인스턴스를 제공합니다. 컨텍스트 인스턴스는 `WeatherForecast` 인스턴스를 반환하는 `JsonTypeInfo<WeatherForecast>` 속성을 제공합니다. TypeInfoPropertyName 특성의 `[JsonSerializable]` 속성을 사용하여 이 속성에 대해 다른 이름을 지정할 수 있습니다.

직렬화 예제

JsonTypeInfo<T> 사용:

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast!, SourceGenerationContext.Default.WeatherForecast);
```

JsonSerializerContext 사용:

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast, typeof(WeatherForecast), SourceGenerationContext.Default);
```

JsonSerializerOptions 사용:

C#

```
sourceGenOptions = new JsonSerializerOptions
{
    TypeInfoResolver = SourceGenerationContext.Default
};

jsonString = JsonSerializer.Serialize<WeatherForecast>(weatherForecast,
sourceGenOptions);
```

역직렬화 예제

JsonTypeInfo<T> 사용:

C#

```
weatherForecast = JsonSerializer.Deserialize(
    jsonString, SourceGenerationContext.Default.WeatherForecast);
```

JsonSerializerContext 사용:

C#

```
weatherForecast = JsonSerializer.Deserialize(
    jsonString, typeof(WeatherForecast), SourceGenerationContext.Default)
    as WeatherForecast;
```

JsonSerializerOptions 사용:

C#

```
var sourceGenOptions = new JsonSerializerOptions
{
```

```
    TypeInfoResolver = SourceGenerationContext.Default
};
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString,
sourceGenOptions);
```

전체 프로그램 예제

다음은 전체 프로그램에서 앞에 나온 예제입니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace BothModesNoOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSourceGenerationOptions(WriteIndented = true)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class SourceGenerationContext : JsonSerializerContext { }

    public class Program
    {
        public static void Main()
        {
            string jsonString = """
                {
                    "Date": "2019-08-01T00:00:00",
                    "TemperatureCelsius": 25,
                    "Summary": "Hot"
                }
                """;
            WeatherForecast? weatherForecast;

            weatherForecast = JsonSerializer.Deserialize(
                jsonString, SourceGenerationContext.Default.WeatherForecast);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
            //Date=8/1/2019 12:00:00 AM

            weatherForecast = JsonSerializer.Deserialize(
                jsonString, typeof(WeatherForecast),
SourceGenerationContext.Default)
                as WeatherForecast;
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
```

```
            //Date=8/1/2019 12:00:00 AM

            var sourceGenOptions = new JsonSerializerOptions
            {
                TypeInfoResolver = SourceGenerationContext.Default
            };
            weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString, sourceGenOptions);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
            //Date=8/1/2019 12:00:00 AM

            jsonString = JsonSerializer.Serialize(
                weatherForecast!, SourceGenerationContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

            jsonString = JsonSerializer.Serialize(
                weatherForecast, typeof(WeatherForecast),
SourceGenerationContext.Default);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

            sourceGenOptions = new JsonSerializerOptions
            {
                TypeInfoResolver = SourceGenerationContext.Default
            };

            jsonString = JsonSerializer.Serialize<WeatherForecast>(weatherForecast,
sourceGenOptions);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}
        }
    }
}
```

원본 생성 모드 지정

여러 형식을 포함할 수 있는 전체 컨텍스트에 대해 메타데이터 기반 모드 또는 serialization 최적화 모드를 지정할 수 있습니다. 또는 개별 형식에 대한 모드를 지정할 수 있습니다. 둘 다 수행하면 형식에 대한 모드 지정이 우세합니다.

- 전체 컨텍스트의 경우 JsonSourceGenerationOptionsAttribute.GenerationMode 속성을 사용합니다.
- 개별 형식의 경우 JsonSerializableAttribute.GenerationMode 속성을 사용합니다.

직렬화 최적화(패스트 패스) 모드 예시

- 전체 컨텍스트의 경우:

```C#
[JsonSourceGenerationOptions(GenerationMode =
JsonSourceGenerationMode.Serialization)]
[JsonSerializable(typeof(WeatherForecast))]
internal partial class SerializeOnlyContext : JsonSerializerContext
{
}
```

- 개별 형식의 경우:

```C#
[JsonSerializable(typeof(WeatherForecast), GenerationMode =
JsonSourceGenerationMode.Serialization)]
internal partial class SerializeOnlyWeatherForecastOnlyContext :
JsonSerializerContext
{
}
```

- 전체 프로그램 예제

```C#
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SerializeOnlyNoOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSourceGenerationOptions(GenerationMode =
JsonSourceGenerationMode.Serialization)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class SerializeOnlyContext : JsonSerializerContext
    {
    }

    [JsonSerializable(typeof(WeatherForecast), GenerationMode =
JsonSourceGenerationMode.Serialization)]
    internal partial class SerializeOnlyWeatherForecastOnlyContext :
JsonSerializerContext
    {
    }
```

```
     public class Program
    {
        public static void Main()
        {
            string jsonString;
            WeatherForecast weatherForecast = new()
                { Date = DateTime.Parse("2019-08-01"), TemperatureCelsius = 25,
Summary = "Hot" };

            // Use context that selects Serialization mode only for
WeatherForecast.
            jsonString = JsonSerializer.Serialize(weatherForecast,

SerializeOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

            // Use a context that selects Serialization mode.
            jsonString = JsonSerializer.Serialize(weatherForecast,
                SerializeOnlyContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}
        }
    }
}
```

메타데이터 기반 모드 예제

- 전체 컨텍스트의 경우:

C#

```
[JsonSourceGenerationOptions(GenerationMode =
JsonSourceGenerationMode.Metadata)]
[JsonSerializable(typeof(WeatherForecast))]
internal partial class MetadataOnlyContext : JsonSerializerContext
{
}
```

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast, MetadataOnlyContext.Default.WeatherForecast);
```

C#

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
    jsonString, MetadataOnlyContext.Default.WeatherForecast);
```

- 개별 형식의 경우:

C#

```
[JsonSerializable(typeof(WeatherForecast), GenerationMode =
JsonSourceGenerationMode.Metadata)]
internal partial class MetadataOnlyWeatherForecastOnlyContext :
JsonSerializerContext
{
}
```

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast,
    MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
```

C#

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
    jsonString,
MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
```

- 전체 프로그램 예제

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace MetadataOnlyNoOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSerializable(typeof(WeatherForecast), GenerationMode =
JsonSourceGenerationMode.Metadata)]
    internal partial class MetadataOnlyWeatherForecastOnlyContext :
JsonSerializerContext
    {
    }

    [JsonSourceGenerationOptions(GenerationMode =
```

```
JsonSourceGenerationMode.Metadata)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class MetadataOnlyContext : JsonSerializerContext
    {
    }

    public class Program
    {
        public static void Main()
        {
            string jsonString = """
                {
                  "Date": "2019-08-01T00:00:00",
                  "TemperatureCelsius": 25,
                  "Summary": "Hot"
                }
                """;
            WeatherForecast? weatherForecast;

            // Deserialize with context that selects metadata mode only for
WeatherForecast only.
            weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
                jsonString,
MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
            //Date=8/1/2019 12:00:00 AM

            // Serialize with context that selects metadata mode only for
WeatherForecast only.
            jsonString = JsonSerializer.Serialize(
                weatherForecast,

MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

            // Deserialize with context that selects metadata mode only.
            weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
                jsonString, MetadataOnlyContext.Default.WeatherForecast);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
            //Date=8/1/2019 12:00:00 AM

            // Serialize with context that selects metadata mode only.
            jsonString = JsonSerializer.Serialize(
                weatherForecast, MetadataOnlyContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}
        }
```

```
    }
}
```

ASP.NET Core의 원본 생성 지원

Blazor 앱에서는 원본 생성 컨텍스트를 받는 HttpClientJsonExtensions.GetFromJsonAsync 및 HttpClientJsonExtensions.PostAsJsonAsync 확장 메서드의 오버로드를 사용하거나 `TypeInfo<TValue>`를 사용합니다.

.NET 8부터는 소스 생성 컨텍스트 또는 HttpClientJsonExtensions.GetFromJsonAsAsyncEnumerable를 허용하는 `TypeInfo<TValue>` 확장 메서드의 오버로드도 사용할 수 있습니다.

Razor Pages, MVC, SignalR, Web API 앱에서 JsonSerializerOptions.TypeInfoResolver 속성을 사용하여 컨텍스트를 지정합니다.

C#

```
[JsonSerializable(typeof(WeatherForecast[]))]
internal partial class MyJsonContext : JsonSerializerContext { }
```

C#

```
var serializerOptions = new JsonSerializerOptions
{
    TypeInfoResolver = MyJsonContext.Default
};

services.AddControllers().AddJsonOptions(
    static options =>

options.JsonSerializerOptions.TypeInfoResolverChain.Add(MyJsonContext.Default));
```

ⓘ 참고 항목

JsonSourceGenerationMode.Serialization 또는 *패스트 패스 직렬화*는 비동기 직렬화에 지원되지 않습니다. 스트리밍 직렬화에는 메타데이터 기반 모델이 필요하지만 페이로드가 미리 결정된 버퍼 크기에 맞게 충분히 작은 것으로 알려진 경우 빠른 경로로 돌아갑니다. 자세한 내용은 https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/#json 를 참조하세요.

리플렉션 기본값 사용 안 함

System.Text.Json은 기본적으로 리플렉션을 사용하므로 기본 Serialization 메서드를 호출하면 필수 리플렉션 API를 모두 지원하지 않는 네이티브 AOT 앱이 중단될 수 있습니다. 이러한 중단은 예측할 수 없기 때문에 진단하기 어려울 수 있으며 리플렉션이 작동하는 CoreCLR 런타임을 사용하여 앱이 디버그되는 경우가 많습니다. 대신 리플렉션 기반 Serialization을 명시적으로 사용하지 않도록 설정하면 중단을 진단하기가 더 쉽습니다. 리플렉션 기반 시리얼라이제이션을 사용하는 코드는 런타임 시에 설명 메시지와 함께 InvalidOperationException가 발생합니다.

앱에서 기본 리플렉션을 사용하지 않도록 설정하려면 프로젝트 파일에서 `JsonSerializerIsReflectionEnabledByDefault` MSBuild 속성을 `false`로 설정합니다.

XML

```
<PropertyGroup>

<JsonSerializerIsReflectionEnabledByDefault>false</JsonSerializerIsReflectionEnabled
ByDefault>
</PropertyGroup>
```

- 이 속성의 동작은 런타임(CoreCLR 또는 Native AOT)에 관계없이 일관됩니다.
- 이 속성을 지정하지 않고 PublishTrimmed가 활성화된 경우 리플렉션 기반 Serialization이 자동으로 비활성화됩니다.

JsonSerializer.IsReflectionEnabledByDefault 속성을 사용하여 리플렉션이 비활성화되었는지 프로그래밍 방식으로 확인할 수 있습니다. 다음 코드 조각은 리플렉션이 사용되는지 여부에 따라 직렬 변환기를 구성하는 방법을 보여 줍니다.

C#

```
static JsonSerializerOptions CreateDefaultOptions()
{
    return new()
    {
        TypeInfoResolver = JsonSerializer.IsReflectionEnabledByDefault
            ? new DefaultJsonTypeInfoResolver()
            : MyContext.Default
    };
}
```

속성이 링크 시점 상수로 처리되기 때문에, 이전 방법은 네이티브 AOT에서 실행되는 애플리케이션에서 리플렉션 기반 리졸버를 루트화하지 않습니다.

옵션 지정

.NET 8 이상 버전에서는 JsonSerializerOptions 특성을 사용하여 설정할 수 있는 대부분의 옵션을 JsonSourceGenerationOptionsAttribute 특성을 사용하여 설정할 수도 있습니다. 특성을 통해 옵

션을 설정하는 것의 이점은 구성이 컴파일 시간에 지정되어 생성된 `MyContext.Default` 속성이 모든 관련 옵션 집합으로 미리 구성되도록 한다는 것입니다.

다음 코드에서는 JsonSourceGenerationOptionsAttribute 특성을 사용하여 옵션을 설정하는 방법을 보여 줍니다.

C#

```
[JsonSourceGenerationOptions(
    WriteIndented = true,
    PropertyNamingPolicy = JsonKnownNamingPolicy.CamelCase,
    GenerationMode = JsonSourceGenerationMode.Serialization)]
[JsonSerializable(typeof(WeatherForecast))]
internal partial class SerializationModeOptionsContext : JsonSerializerContext
{
}
```

`JsonSourceGenerationOptionsAttribute`를 사용하여 serialization 옵션을 지정하는 경우 다음 serialization 메서드 중 하나를 호출합니다.

- `JsonSerializer.Serialize` 메서드는 `TypeInfo<TValue>`을(를) 받는다. 컨텍스트 클래스의 `Default.<TypeName>` 속성을 전달합니다.

  C#

  ```
  jsonString = JsonSerializer.Serialize(
      weatherForecast, SerializationModeOptionsContext.Default.WeatherForecast);
  ```

- 컨텍스트를 사용하는 `JsonSerializer.Serialize` 메서드. 컨텍스트 클래스의 `Default` 정적 속성을 전달합니다.

  C#

  ```
  jsonString = JsonSerializer.Serialize(
      weatherForecast, typeof(WeatherForecast),
  SerializationModeOptionsContext.Default);
  ```

고유한 `Utf8JsonWriter` 인스턴스를 전달할 수 있는 메서드를 호출하는 경우 작성기의 Indented 설정이 `JsonSourceGenerationOptionsAttribute.WriteIndented` 옵션 대신 적용됩니다.

`JsonSerializerOptions` 인스턴스를 사용하는 생성자를 호출하여 컨텍스트 인스턴스를 만들고 사용하는 경우 제공된 인스턴스는 `JsonSourceGenerationOptionsAttribute`에서 지정한 옵션 대신 사용됩니다.

다음 코드는 전체 프로그램의 이전 예제를 보여 줍니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SerializeOnlyWithOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSourceGenerationOptions(
        WriteIndented = true,
        PropertyNamingPolicy = JsonKnownNamingPolicy.CamelCase,
        GenerationMode = JsonSourceGenerationMode.Serialization)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class SerializationModeOptionsContext : JsonSerializerContext
    {
    }
    public class Program
    {
        public static void Main()
        {
            string jsonString;
            WeatherForecast weatherForecast = new()
                { Date = DateTime.Parse("2019-08-01"),
                  TemperatureCelsius = 25,
                  Summary = "Hot" };

            // Serialize using TypeInfo<TValue> provided by the context
            // and options specified by [JsonSourceGenerationOptions].
            jsonString = JsonSerializer.Serialize(
                weatherForecast,
SerializationModeOptionsContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{
            //  "date": "2019-08-01T00:00:00",
            //  "temperatureCelsius": 0,
            //  "summary": "Hot"
            //}

            // Serialize using Default context
            // and options specified by [JsonSourceGenerationOptions].
            jsonString = JsonSerializer.Serialize(
                weatherForecast, typeof(WeatherForecast),
SerializationModeOptionsContext.Default);
            Console.WriteLine(jsonString);
            // output:
            //{
            //  "date": "2019-08-01T00:00:00",
            //  "temperatureCelsius": 0,
```

```
            //  "summary": "Hot"
            //}
        }
    }
}
```

원본 생성기 결합

단일 JsonSerializerOptions 인스턴스 내에서 여러 원본 생성 컨텍스트의 계약을 결합할 수 있습니다. JsonSerializerOptions.TypeInfoResolver 메서드를 사용하여 결합된 여러 컨텍스트를 연결하려면 JsonTypeInfoResolver.Combine(IJsonTypeInfoResolver[]) 속성을 사용합니다.

C#

```
var options = new JsonSerializerOptions
{
    TypeInfoResolver = JsonTypeInfoResolver.Combine(ContextA.Default,
ContextB.Default, ContextC.Default),
};
```

.NET 8부터 나중에 다른 컨텍스트를 앞이나 뒤에 추가하려는 경우 JsonSerializerOptions.TypeInfoResolverChain 속성을 사용하면 됩니다. 체인의 순서는 중요합니다. JsonSerializerOptions는 각 확인자를 지정된 순서로 쿼리하고 null이 아닌 첫 번째 결과를 반환합니다.

C#

```
options.TypeInfoResolverChain.Add(ContextD.Default); // Append to the end of the
list.
options.TypeInfoResolverChain.Insert(0, ContextE.Default); // Insert at the
beginning of the list.
```

TypeInfoResolverChain 속성에 대한 모든 변경 내용은 TypeInfoResolver에 반영되며 그 반대의 경우도 마찬가지입니다.

열거형 필드를 문자열로 직렬화

기본적으로 열거형은 숫자로 직렬화됩니다. 원본 생성을 사용할 때 특정 열거형의 필드를 문자열로 직렬화하려면 JsonStringEnumConverter<TEnum> 변환기로 주석을 추가합니다. 또는 모든 열거형에 대해 포괄 정책을 설정하려면 JsonSourceGenerationOptionsAttribute 특성을 사용합니다.

`JsonStringEnumConverter<T>` 변환기

원본 생성을 사용하여 열거형 이름을 문자열로 직렬화하려면 JsonStringEnumConverter<TEnum> 변환기를 사용합니다. (제네릭이 아닌 JsonStringEnumConverter 형식은 네이티브 AOT 런타임에서 지원되지 않습니다.)

JsonStringEnumConverter<TEnum> 특성을 사용하여 JsonConverterAttribute 변환기로 열거형 형식에 주석을 추가합니다.

C#

```
public class WeatherForecastWithPrecipEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Precipitation? Precipitation { get; set; }
}

[JsonConverter(typeof(JsonStringEnumConverter<Precipitation>))]
public enum Precipitation
{
    Drizzle, Rain, Sleet, Hail, Snow
}
```

JsonSerializerContext 클래스를 만들고 JsonSerializableAttribute 특성으로 주석을 추가합니다.

C#

```
[JsonSerializable(typeof(WeatherForecastWithPrecipEnum))]
public partial class Context1 : JsonSerializerContext { }
```

다음 코드는 숫자 값 대신 열거형 이름을 직렬화합니다.

C#

```
var weatherForecast = new WeatherForecastWithPrecipEnum
{
    Date = DateTime.Parse("2019-08-01"),
    TemperatureCelsius = 25,
    Precipitation = Precipitation.Sleet
};

var options = new JsonSerializerOptions
{
    WriteIndented = true,
    TypeInfoResolver = Context1.Default,
};
string? jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

그 결과로 얻는 JSON은 다음 예제와 유사합니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Precipitation": "Sleet"
}
```

포괄 정책

JsonStringEnumConverter<TEnum> 형식을 사용하는 대신 JsonSourceGenerationOptionsAttribute를 사용하여 열거형을 문자열로 직렬화하는 포괄 정책을 적용할 수 있습니다. JsonSerializerContext 클래스를 만들고 JsonSerializableAttribute *및* *JsonSourceGenerationOptionsAttribute* 특성으로 주석을 추가합니다.

C#

```
[JsonSourceGenerationOptions(UseStringEnumConverter = true)]
[JsonSerializable(typeof(WeatherForecast2WithPrecipEnum))]
public partial class Context2 : JsonSerializerContext { }
```

열거형에는 JsonConverterAttribute가 없습니다.

C#

```
public class WeatherForecast2WithPrecipEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Precipitation2? Precipitation { get; set; }
}

public enum Precipitation2
{
    Drizzle, Rain, Sleet, Hail, Snow
}
```

사용자 지정 열거형 멤버 이름

.NET 9부터 JsonStringEnumMemberName 특성을 사용하여 열거형 멤버 이름을 사용자 지정할 수 있습니다. 자세한 내용은 사용자 지정 열거형 멤버 이름을 참조 하세요.

참조

- .NET에서의 JSON 직렬화 및 역직렬화 - 개요
- 라이브러리를 사용하는 방법

Last updated on 2025. 11. 15.

System.Text.Json을 사용하여 문자 인코딩을 사용자 지정하는 방법

아티클 • 2023. 05. 26.

기본적으로 직렬 변환기는 ASCII가 아닌 문자를 모두 이스케이프합니다. 즉, ASCII가 아닌 문자를 `\uxxxx`로 바꾸며, 여기서 `xxxx`는 문자의 유니코드 코드입니다. 예를 들어 다음 JSON의 `Summary` 속성이 키릴 자모 `жарко`로 설정된 경우 다음 예제와 같이 `WeatherForecast` 개체가 직렬화됩니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "\u0436\u0430\u0440\u043A\u043E"
}
```

언어 문자 세트 직렬화

하나 이상의 언어 문자 세트를 이스케이프하지 않고 직렬화하려면 다음 예제와 같이 System.Text.Encodings.Web.JavaScriptEncoder 인스턴스를 만들 때 유니코드 범위를 지정합니다.

C#

```
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

C#

```
var options1 = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.Create(UnicodeRanges.BasicLatin,
UnicodeRanges.Cyrillic),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options1);
```

이 코드는 키릴 자모 또는 그리스어 문자를 이스케이프하지 않습니다. `Summary` 속성이 키릴 자모 `жарко`로 설정된 경우 다음 예제와 같이 `WeatherForecast` 개체가 직렬화됩니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "жарко"
}
```

기본적으로 인코더는 BasicLatin 범위로 초기화됩니다.

모든 언어 세트를 이스케이프하지 않고 직렬화하려면 UnicodeRanges.All을 사용합니다.

특정 문자 직렬화

이스케이프하지 않고 허용하려는 개별 문자를 지정하는 방법도 있습니다. 다음 예제는 `жарко`의 처음 두 문자만 직렬화합니다.

C#

```
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

C#

```
var encoderSettings = new TextEncoderSettings();
encoderSettings.AllowCharacters('\u0436', '\u0430');
encoderSettings.AllowRange(UnicodeRanges.BasicLatin);
var options2 = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.Create(encoderSettings),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options2);
```

다음은 이전 코드에서 생성된 JSON 예제입니다.

JSON

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "жa\u0440\u043A\u043E"
}
```

차단 목록

앞의 섹션에서는 이스케이프하지 않으려는 코드 요소 또는 범위의 허용 목록을 지정하는 방법을 보여 줍니다. 그러나 허용 목록의 특정 코드 요소를 재정의할 수 있는 전역 및 인코더별 차단 목록이 있습니다. 차단 목록의 코드 지점은 허용 목록에 포함된 경우에도 항상 이스케이프됩니다.

전역 차단 목록

전역 차단 목록에는 프라이빗 사용 문자, 컨트롤 문자, 정의되지 않은 코드 포인트 및 Space_Separator 범주와 같은 특정 유니코드 범주(`U+0020 SPACE` 제외)가 포함됩니다. 예를 들어 `U+3000 IDEOGRAPHIC SPACE`는 유니코드 범위 CJK 기호 및 문장 부호(U+3000-U+303F)를 허용 목록으로 지정하더라도 이스케이프됩니다.

전역 블록 목록은 .NET의 모든 릴리스에서 변경된 구현 세부 정보입니다. 전역 차단 목록의 멤버이거나 멤버가 아닌 문자에 대한 종속성은 사용하지 마세요.

인코더별 차단 목록

인코더별 차단된 코드 포인트의 예로는 HTML 인코더에 대한 `'<'` 및 `'&'`, JSON 인코더에 대한 `'\'`, URL 인코더에 대한 `'%'`가 있습니다. 예를 들어 HTML 인코더는 앰퍼샌드가 `BasicLatin` 범위에 있고 모든 인코더가 기본적으로 `BasicLatin`으로 초기화되더라도 항상 앰퍼샌드(`'&'`)를 이스케이프합니다.

모든 문자 직렬화

이스케이프를 최소화하려면 다음 예제와 같이 JavaScriptEncoder.UnsafeRelaxedJsonEscaping을 사용합니다.

C#

```
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

C#

```
var options3 = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping,
    WriteIndented = true
```

```
};
jsonString = JsonSerializer.Serialize(weatherForecast, options3);
```

⊗ 주의

기본 인코더와 비교할 때, `UnsafeRelaxedJsonEscaping` 인코더는 문자를 이스케이프하지 않은 상태로 통과할 수 있도록 허용하는 기준이 더 관대합니다.

- `<`, `>`, `&` 및 `'`처럼 HTML 구분 문자를 이스케이프하지 않습니다.
- *문자 세트*에 대한 클라이언트와 서버의 내용이 서로 다를 때 발생할 수 있는 XSS 또는 정보 노출 공격에 대한 심층 방어를 추가로 제공하지 않습니다.

클라이언트가 결과 페이로드를 UTF-8로 인코딩된 JSON으로 해석한다는 사실을 알고 있는 경우에만 안전하지 않은 인코더를 사용해야 합니다. 예를 들어 서버에서 응답 헤더 `Content-Type: application/json; charset=utf-8`을 보내는 경우에 사용할 수 있습니다. 원시 `UnsafeRelaxedJsonEscaping` 출력을 HTML 페이지나 `<script>` 요소로 내보내도록 허용하면 절대 안 됩니다.

참고 항목

- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

.NET에서 JSON serialization(마샬링)용 사용자 지정 변환기를 작성하는 방법

이 문서에서는 System.Text.Json 네임스페이스에 제공된 JSON serialization 클래스에 대한 사용자 지정 변환기를 만드는 방법을 보여 줍니다. `System.Text.Json`에 대한 소개는 .NET에서 JSON을 직렬화 및 역직렬화하는 방법을 참조하세요.

*변환기*는 개체 또는 값을 JSON으로 변환하는 클래스입니다. `System.Text.Json` 네임스페이스에는 JavaScript 기본 형식에 매핑되는 기본 형식 대부분에 대한 기본 제공 변환기가 있습니다. 사용자 지정 변환기를 작성하여 기본 제공 변환기의 기본 동작을 재정의할 수 있습니다. 다음은 그 예입니다.

- 값을 mm/dd/yyyy 형식으로 나타낼 수 있습니다 `DateTime` . 기본적으로 RFC 3339 프로필을 포함하여 ISO 8601-1:2019가 지원됩니다. 자세한 내용은 System.Text.Json의 DateTime 및 DateTimeOffset 지원을 참조하세요.
- 예를 들어 `PhoneNumber` 형식을 사용하여 POCO를 JSON 문자열로 직렬화할 수 있습니다.

또한 새로운 기능으로 `System.Text.Json`을 사용자 지정하거나 확장하기 위해 사용자 지정 변환기를 작성할 수도 있습니다. 이 문서의 뒷부분에서 다음과 같은 시나리오에 대해 설명합니다.

- 유추된 형식을 개체 속성으로 역직렬화
- 다형 deserialization 지원
- Stack 형식에 대한 왕복을 지원합니다.
- 기본 시스템 변환기를 사용합니다.

Visual Basic은 사용자 지정 변환기를 작성하는 데 사용할 수 없지만 C# 라이브러리에 구현된 변환기를 호출할 수 있습니다. 자세한 내용은 Visual Basic 지원을 참조하세요.

사용자 지정 변환기 패턴

사용자 지정 변환기를 만드는 데는 기본 패턴과 팩터리 패턴의 두 가지 패턴이 있습니다. 팩터리 패턴은 `Enum` 형식 또는 개방형 제네릭을 처리하는 변환기를 위한 것입니다. 기본 패턴은 제네릭이 아닌 형식과 폐쇄형 제네릭 형식을 위한 것입니다. 예를 들어 다음 형식의 변환기에는 팩터리 패턴이 필요합니다.

- Dictionary<TKey,TValue>
- Enum
- List<T>

기본 패턴으로 처리할 수 있는 형식의 몇 가지 예는 다음과 같습니다.

- `Dictionary<int, string>`

- `WeekdaysEnum`
- `List<DateTimeOffset>`
- DateTime
- Int32

기본 패턴은 한 형식을 처리할 수 있는 클래스를 만듭니다. 팩터리 패턴은 런타임에 특정 형식이 필요한지 여부를 결정하는 클래스를 만들고 적절한 변환기를 동적으로 만듭니다.

샘플 기본 변환기

다음 샘플은 기존 데이터 형식에 대한 기본 serialization을 재정의하는 변환기입니다. 변환기는 `DateTimeOffset` 속성에 mm/dd/yyyy 형식을 사용합니다.

C#

```
using System.Globalization;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DateTimeOffsetJsonConverter : JsonConverter<DateTimeOffset>
    {
        public override DateTimeOffset Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
                DateTimeOffset.ParseExact(reader.GetString()!,
                    "MM/dd/yyyy", CultureInfo.InvariantCulture);

        public override void Write(
            Utf8JsonWriter writer,
            DateTimeOffset dateTimeValue,
            JsonSerializerOptions options) =>
                writer.WriteStringValue(dateTimeValue.ToString(
                    "MM/dd/yyyy", CultureInfo.InvariantCulture));
    }
}
```

샘플 팩터리 패턴 변환기

다음 코드에서는 `Dictionary<Enum,TValue>`와 함께 작동하는 사용자 지정 변환기를 보여 줍니다. 첫 번째 제네릭 형식 매개 변수가 `Enum`이고 두 번째는 개방형이기 때문에 이 코드는 팩터리 패턴을 따릅니다. `CanConvert` 메서드는 두 개의 제네릭 매개 변수(이 중 첫 번째는 `true` 형식)가 있는 `Dictionary`에 대해서만 `Enum`를 반환합니다. 내부 변환기는 런타임에 `TValue`에 제공되는 형식을 처리하기 위해 기존 변환기를 가져옵니다.

C#

```
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DictionaryTKeyEnumTValueConverter : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
        {
            if (!typeToConvert.IsGenericType)
            {
                return false;
            }

            if (typeToConvert.GetGenericTypeDefinition() != typeof(Dictionary<,>))
            {
                return false;
            }

            return typeToConvert.GetGenericArguments()[0].IsEnum;
        }

        public override JsonConverter CreateConverter(
            Type type,
            JsonSerializerOptions options)
        {
            Type[] typeArguments = type.GetGenericArguments();
            Type keyType = typeArguments[0];
            Type valueType = typeArguments[1];

            JsonConverter converter = (JsonConverter)Activator.CreateInstance(
                typeof(DictionaryEnumConverterInner<,>).MakeGenericType(
                    [keyType, valueType]),
                BindingFlags.Instance | BindingFlags.Public,
                binder: null,
                args: [options],
                culture: null)!;

            return converter;
        }

        private class DictionaryEnumConverterInner<TKey, TValue> :
            JsonConverter<Dictionary<TKey, TValue>> where TKey : struct, Enum
        {
            private readonly JsonConverter<TValue> _valueConverter;
            private readonly Type _keyType;
            private readonly Type _valueType;

            public DictionaryEnumConverterInner(JsonSerializerOptions options)
            {
                // For performance, use the existing converter.
                _valueConverter = (JsonConverter<TValue>)options
```

```
                .GetConverter(typeof(TValue));

            // Cache the key and value types.
            _keyType = typeof(TKey);
            _valueType = typeof(TValue);
        }

        public override Dictionary<TKey, TValue> Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
                throw new JsonException();
            }

            var dictionary = new Dictionary<TKey, TValue>();

            while (reader.Read())
            {
                if (reader.TokenType == JsonTokenType.EndObject)
                {
                    return dictionary;
                }

                // Get the key.
                if (reader.TokenType != JsonTokenType.PropertyName)
                {
                    throw new JsonException();
                }

                string? propertyName = reader.GetString();

                // For performance, parse with ignoreCase:false first.
                if (!Enum.TryParse(propertyName, ignoreCase: false, out TKey
key) &&
                    !Enum.TryParse(propertyName, ignoreCase: true, out key))
                {
                    throw new JsonException(
                        $"Unable to convert \"{propertyName}\" to Enum \"
{_keyType}\".");
                }

                // Get the value.
                reader.Read();
                TValue value = _valueConverter.Read(ref reader, _valueType,
options)!;

                // Add to dictionary.
                dictionary.Add(key, value);
            }

            throw new JsonException();
        }
```

```
            public override void Write(
                Utf8JsonWriter writer,
                Dictionary<TKey, TValue> dictionary,
                JsonSerializerOptions options)
            {
                writer.WriteStartObject();

                foreach ((TKey key, TValue value) in dictionary)
                {
                    string propertyName = key.ToString();
                    writer.WritePropertyName
                        (options.PropertyNamingPolicy?.ConvertName(propertyName) ??
propertyName);

                    _valueConverter.Write(writer, value, options);
                }

                writer.WriteEndObject();
            }
        }
    }
}
```

기본 패턴을 따르기 위한 단계

다음 단계에서는 기본 패턴을 따라 변환기를 만드는 방법을 설명합니다.

- JsonConverter<T>에서 파생되는 클래스를 만듭니다. 여기서 `T`는 직렬화 및 역직렬화할 형식입니다.
- 들어오는 JSON을 역직렬화하고 `Read` 형식으로 변환하도록 `T` 메서드를 재정의합니다. 메서드에 전달된 Utf8JsonReader를 사용하여 JSON을 읽습니다. 직렬 변환기가 현재 JSON 범위에 대한 모든 데이터를 전달하므로 부분 데이터 처리에 대해 신경을 쓸 필요가 없습니다. 따라서 Skip 또는 TrySkip을 호출하거나 Read를 반환하는 `true`의 유효성을 검사할 필요가 없습니다.
- 들어오는 `Write` 형식의 개체를 직렬화하도록 `T` 메서드를 재정의합니다. 메서드에 전달된 Utf8JsonWriter를 사용하여 JSON을 작성합니다.
- 필요한 경우에만 `CanConvert` 메서드를 재정의합니다. 기본 구현은 변환할 형식이 `true` 형식일 때 `T`를 반환합니다. 따라서 `T` 형식만 지원하는 변환기는 이 메서드를 재정의할 필요가 없습니다. 이 메서드를 재정의해야 하는 변환기의 예제는 이 문서의 뒷부분에 나오는 다형 deserialization 섹션을 참조하세요.

사용자 지정 변환기 작성을 위한 참조 구현으로 기본 제공 변환기 소스 코드를 참조할 수 있습니다.

팩터리 패턴을 따르기 위한 단계

다음 단계에서는 팩터리 패턴을 따라 변환기를 만드는 방법을 설명합니다.

- JsonConverterFactory에서 파생되는 클래스를 만듭니다.
- 변환할 형식이 변환기에서 처리할 수 있는 형식인 경우 `CanConvert`를 반환하도록 `true` 메서드를 재정의합니다. 예를 들어 `List<T>`에 대한 변환기인 경우 `List<int>`, `List<string>` 및 `List<DateTime>`만 처리할 수 있습니다.
- 런타임에 제공되는 변환할 형식을 처리할 변환기 클래스 인스턴스를 반환하도록 `CreateConverter` 메서드를 재정의합니다.
- `CreateConverter` 메서드가 인스턴스화하는 변환기 클래스를 만듭니다.

개체와 문자열 간에 변환하는 코드가 모든 형식에 대해 동일하지 않기 때문에 개방형 제네릭에는 팩터리 패턴이 필요합니다. 개방형 제네릭 형식(예: `List<T>`)에 대한 변환기는 백그라운드에서 폐쇄형 제네릭 형식(예: `List<DateTime>`)에 대한 변환기를 만들어야 합니다. 변환기가 처리할 수 있는 각 폐쇄형 제네릭 형식을 처리하기 위해 코드를 작성해야 합니다.

`Enum` 형식은 개방형 제네릭 형식과 유사합니다. `Enum`에 대한 변환기는 내부적으로 특정 `Enum`(예: `WeekdaysEnum`)에 대한 변환기를 만들어야 합니다.

`Utf8JsonReader` 메서드에서 `Read`의 사용

변환기가 JSON 개체 `Utf8JsonReader` 를 변환하는 경우 메서드가 시작될 때 시작 개체 토큰에 `Read` 배치됩니다. 그런 다음 해당 개체의 모든 토큰을 끝까지 읽고 **해당 end 개체 토큰**에 판독기가 배치된 메서드를 종료해야 합니다. 개체의 끝 부분을 건너뛰어 읽거나 해당 end 토큰에 도달하기도 전에 읽기를 중지하는 경우 다음을 나타내는 `JsonException` 예외가 발생합니다.

> 변환기 'ConverterName'이 해당 토큰을 너무 많이 읽거나 충분히 읽지 않았습니다.

관련 예제는 이전 팩터리 패턴 샘플 변환기를 참조하세요. `Read` 메서드는 판독기가 start 개체 토큰에 배치되었는지 확인하는 것으로 시작합니다. 판독기가 다음 end 개체 토큰에 배치되는 것을 발견할 때까지 읽기는 계속 진행됩니다. 개체 내의 개체를 나타내는 중간 start 개체 토큰이 없으므로 다음 end 개체 토큰에서 읽기가 중지됩니다. 배열을 변환하는 경우 begin 토큰 및 end 토큰에 대한 동일한 규칙이 적용됩니다. 관련 예제는 이 문서 뒷부분에 있는 Stack<T> 샘플 변환기를 참조하세요.

오류 처리

직렬 변환기는 JsonException 및 NotSupportedException 예외 형식을 특별히 처리합니다.

JsonException

메시지 없이 `JsonException`을 throw하는 경우 직렬 변환기는 JSON에서 오류를 발생시킨 부분의 경로를 포함하는 메시지를 만듭니다. 예를 들어 `throw new JsonException()` 문은 다음 예제와 같은 오류 메시지를 생성합니다.

출력

```
Unhandled exception. System.Text.Json.JsonException:
The JSON value could not be converted to System.Object.
Path: $.Date | LineNumber: 1 | BytePositionInLine: 37.
```

메시지를 제공하는 경우(예: `throw new JsonException("Error occurred")`) 직렬 변환기는 Path, LineNumber 및 BytePositionInLine 속성을 계속 설정합니다.

NotSupportedException

`NotSupportedException`을 throw하는 경우 메시지에 항상 경로 정보를 받습니다. 메시지를 제공하는 경우 경로 정보가 추가됩니다. 예를 들어 `throw new NotSupportedException("Error occurred.")` 문은 다음 예제와 같은 오류 메시지를 생성합니다.

출력

```
Error occurred. The unsupported member type is located on type
'System.Collections.Generic.Dictionary`2[Samples.SummaryWords,System.Int32]'.
Path: $.TemperatureRanges | LineNumber: 4 | BytePositionInLine: 24
```

예외 형식을 throw하는 경우

JSON 페이로드에 역직렬화할 형식에 유효하지 않은 토큰이 포함되어 있으면 `JsonException`을(를) 던지세요.

특정 형식을 허용하지 않으려면 `NotSupportedException`을 throw합니다. 이 예외는 직렬 변환기가 지원되지 않는 형식에 대해 자동으로 발생시키는 것입니다. 예를 들어, `System.Type`는 보안상의 이유로 지원되지 않기 때문에, 이를 역직렬화하려고 하면 `NotSupportedException`가 발생합니다.

필요에 따라 다른 예외를 throw할 수 있지만, JSON 경로 정보는 자동으로 포함되지 않습니다.

사용자 지정 변환기 등록

및 `Serialize` 메서드가 사용할 수 있도록 사용자 지정 변환기를 `Deserialize`합니다. 다음 방법 중 하나를 선택합니다.

- JsonSerializerOptions.Converters 컬렉션에 변환기 클래스의 인스턴스를 추가합니다.
- [JsonConverter] 특성을 사용자 지정 변환기가 필요한 속성에 적용합니다.
- [JsonConverter] 특성을 사용자 지정 값 형식을 나타내는 클래스 또는 구조체에 적용합니다.

등록 샘플 - 변환기 컬렉션

다음 예제에서는 DateTimeOffsetJsonConverter를 DateTimeOffset 형식의 속성에 대한 기본값으로 지정합니다.

C#

```
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true
};
serializeOptions.Converters.Add(new DateTimeOffsetJsonConverter());

jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

다음 형식의 인스턴스를 직렬화한다고 가정합니다.

C#

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

다음은 사용자 지정 변환기가 사용되었음을 보여 주는 JSON 출력의 예입니다.

JSON

```
{
  "Date": "08/01/2019",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

다음 코드에서는 사용자 지정 `DateTimeOffset` 변환기를 사용하여 역직렬화하기 위해 동일한 방법을 사용합니다.

C#

```
var deserializeOptions = new JsonSerializerOptions();
deserializeOptions.Converters.Add(new DateTimeOffsetJsonConverter());
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString,
deserializeOptions)!;
```

등록 샘플 - 속성에 적용되는 [JsonConverter]

다음 코드에서는 `Date` 속성에 대한 사용자 지정 변환기를 선택합니다.

C#

```
public class WeatherForecastWithConverterAttribute
{
    [JsonConverter(typeof(DateTimeOffsetJsonConverter))]
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

`WeatherForecastWithConverterAttribute`를 직렬화하는 코드는 `JsonSerializeOptions.Converters`를 사용할 필요가 없습니다.

C#

```
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

역직렬화하는 코드도 `Converters`를 사용할 필요가 없습니다.

C#

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithConverterAttribute>
(jsonString)!;
```

등록 샘플 - 형식에 적용되는 [JsonConverter]

다음은 구조체를 만들고 `[JsonConverter]` 특성을 적용하는 코드입니다.

C#

```
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    [JsonConverter(typeof(TemperatureConverter))]
    public struct Temperature
    {
        public Temperature(int degrees, bool celsius)
        {
            Degrees = degrees;
            IsCelsius = celsius;
        }

        public int Degrees { get; }
        public bool IsCelsius { get; }
        public bool IsFahrenheit => !IsCelsius;

        public override string ToString() =>
            $"{Degrees}{(IsCelsius ? "C" : "F")}";

        public static Temperature Parse(string input)
        {
            int degrees = int.Parse(input.Substring(0, input.Length - 1));
            bool celsius = input.Substring(input.Length - 1) == "C";

            return new Temperature(degrees, celsius);
        }
    }
}
```

앞의 구조체에 대한 사용자 지정 변환기는 다음과 같습니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class TemperatureConverter : JsonConverter<Temperature>
    {
        public override Temperature Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
                Temperature.Parse(reader.GetString()!);

        public override void Write(
            Utf8JsonWriter writer,
            Temperature temperature,
            JsonSerializerOptions options) =>
                writer.WriteStringValue(temperature.ToString());
```

```
    }
}
```

구조체의 `[JsonConverter]` 특성은 사용자 지정 변환기를 `Temperature` 형식의 속성에 대한 기본값으로 등록합니다. 이 변환기는 다음 형식의 `TemperatureCelsius` 속성을 직렬화 또는 역직렬화할 때 자동으로 사용됩니다.

C#

```
public class WeatherForecastWithTemperatureStruct
{
    public DateTimeOffset Date { get; set; }
    public Temperature TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

변환기 등록 우선 순위

serialization 또는 deserialization 동안 각 JSON 요소에 대해 가장 높은 우선 순위에서 가장 낮은 순서로 변환기가 선택됩니다.

- 속성에 적용된 `[JsonConverter]`.
- `Converters` 컬렉션에 추가된 변환기.
- 사용자 지정 값 형식 또는 POCO에 적용된 `[JsonConverter]`.

`Converters` 컬렉션에 특정 형식에 대한 여러 사용자 지정 변환기가 등록된 경우 `true`에 `CanConvert`를 반환하는 첫 번째 변환기가 사용됩니다.

기본 제공 변환기는 해당하는 사용자 지정 변환기가 등록되지 않은 경우에만 선택됩니다.

일반 시나리오용 변환기 샘플

다음 섹션에서는 기본 제공 기능이 처리하지 않는 몇 가지 일반적인 시나리오를 해결하는 변환기 샘플을 제공합니다.

- 유추된 형식을 개체 속성으로 역직렬화
- Stack 형식에 대한 왕복을 지원합니다.
- 기본 시스템 변환기를 사용합니다.

샘플 DataTable 변환기는 지원되는 형식를 참조하세요.

유추된 형식을 개체 속성으로 역직렬화

`object` 형식의 속성으로 역직렬화할 때 `JsonElement` 개체가 만들어집니다. 그 이유는 역직렬 변환기가 만들 CLR 형식을 알지 못하고 추측하려고 하지 않기 때문입니다. 예를 들어 JSON 속성에 "true"가 있는 경우 역직렬 변환기는 값이 `Boolean`임을 유추하지 않으며 요소에 "01/01/2019"가 있는 경우 역직렬 변환기가 `DateTime`를 유추하지 않습니다.

형식 유추는 정확하지 않을 수 있습니다. 역직렬 변환기가 소수점이 없는 JSON 번호를 `long`으로 구문 분석하는 경우 값이 원래 `ulong` 또는 `BigInteger`로 직렬화되었다면 범위를 벗어남 문제가 발생할 수 있습니다. 소수점이 있는 숫자를 `double`로 구문 분석하면 해당 숫자가 원래 `decimal`로 직렬화된 경우에는 전체 자릿수가 손실될 수 있습니다.

형식 유추가 필요한 시나리오의 경우 다음 코드는 `object` 속성에 대한 사용자 지정 변환기를 보여 줍니다. 이 코드는 다음과 같이 변환합니다.

- `true` 및 `false`를 `Boolean`으로
- 소수점이 없는 숫자를 `long`으로
- 소수점이 있는 숫자를 `double`로
- 날짜를 `DateTime`으로
- 문자열을 `string`으로
- 그 밖의 모든 항목으로 `JsonElement`로

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace CustomConverterInferredTypesToObject
{
    public class ObjectToInferredTypesConverter : JsonConverter<object>
    {
        public override object Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) => reader.TokenType switch
            {
                JsonTokenType.True => true,
                JsonTokenType.False => false,
                JsonTokenType.Number when reader.TryGetInt64(out long l) => l,
                JsonTokenType.Number => reader.GetDouble(),
                JsonTokenType.String when reader.TryGetDateTime(out DateTime
datetime) => datetime,
                JsonTokenType.String => reader.GetString()!,
                _ => JsonDocument.ParseValue(ref reader).RootElement.Clone()
            };

        public override void Write(
            Utf8JsonWriter writer,
            object objectToWrite,
            JsonSerializerOptions options)
```

```
        {
            var runtimeType = objectToWrite.GetType();
            if (runtimeType == typeof(object))
            {
                writer.WriteStartObject();
                writer.WriteEndObject();
                return;
            }

            JsonSerializer.Serialize(writer, objectToWrite, runtimeType, options);
        }
    }

    public class WeatherForecast
    {
        public object? Date { get; set; }
        public object? TemperatureCelsius { get; set; }
        public object? Summary { get; set; }
    }

    public class Program
    {
        public static void Run()
        {
            string jsonString = """
                {
                  "Date": "2019-08-01T00:00:00-07:00",
                  "TemperatureCelsius": 25,
                  "Summary": "Hot"
                }
                """;

            WeatherForecast weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(jsonString)!;
            Console.WriteLine($"Type of Date property   no converter =
{weatherForecast.Date!.GetType()}");

            var options = new JsonSerializerOptions();
            options.WriteIndented = true;
            options.Converters.Add(new ObjectToInferredTypesConverter());
            weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString, options)!;
            Console.WriteLine($"Type of Date property with converter =
{weatherForecast.Date!.GetType()}");

            Console.WriteLine(JsonSerializer.Serialize(weatherForecast, options));
        }
    }
}

// Produces output like the following example:
//
//Type of Date property   no converter = System.Text.Json.JsonElement
//Type of Date property with converter = System.DateTime
//{
```

```
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot"
//}
```

이 예제에서는 변환기 코드와 `WeatherForecast` 속성이 있는 `object` 클래스를 보여줍니다. `Main` 메서드는 변환기부터 먼저 사용하지 않고 JSON 문자열을 `WeatherForecast` 인스턴스로 역직렬화한 후 변환기를 사용합니다. 콘솔 출력은 변환기가 없으면 `Date` 속성의 런타임 형식이 `JsonElement`이고 변환기가 있는 런타임 형식은 `DateTime`임을 보여줍니다.

네임스페이스의 `System.Text.Json.Serialization`에는 `object` 속성에 대한 deserialization을 처리하는 사용자 지정 변환기의 더 많은 예제가 있습니다.

다형 deserialization 지원

.NET 7은 다형 직렬화와 역직렬화를 모두 지원합니다. 그러나 이전 .NET 버전에서는 다형 직렬화 지원이 제한되어 있고 역직렬화를 지원하지 않았습니다. .NET 6 또는 이전 버전을 사용하는 경우 역직렬화에는 사용자 지정 변환기가 필요합니다.

예를 들어 `Person` 및 `Employee` 파생 클래스를 사용하는 `Customer` 추상 기본 클래스가 있다고 가정합니다. 다형성 deserialization은 디자인 타임에 `Person`을 deserialization 대상으로 지정할 수 있고 런타임에 JSON의 `Customer` 및 `Employee` 개체가 올바르게 역직렬화됨을 의미합니다. deserialization 동안 JSON에서 필요한 형식을 식별하는 단서를 찾아야 합니다. 사용 가능한 단서의 종류는 각 시나리오마다 다릅니다. 예를 들어 판별자 속성을 사용할 수 있거나 특정 속성이 존재하는지 여부에 의존해야 할 수도 있습니다. 현재 릴리스의 `System.Text.Json`에서는 다형 deserialization 시나리오를 처리하는 방법을 지정하는 특성을 제공하지 않으므로 사용자 지정 변환기가 필요합니다.

다음 코드에서는 기본 클래스, 두 개의 파생 클래스 및 해당 클래스에 대한 사용자 지정 변환기를 보여 줍니다. 이 변환기는 판별자 속성을 사용하여 다형 deserialization을 수행합니다. 형식 판별자는 클래스 정의에 없지만 serialization 동안 만들어지고 deserialization 동안 읽힙니다.

> **중요**
>
> 예제 코드를 사용하려면 JSON 개체 이름/값 쌍이 순서대로 유지되어야 하며 이는 JSON의 표준 요구 사항이 아닙니다.

C#

```
public class Person
{
    public string? Name { get; set; }
}
```

```
public class Customer : Person
{
    public decimal CreditLimit { get; set; }
}

public class Employee : Person
{
    public string? OfficeNumber { get; set; }
}
```

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class PersonConverterWithTypeDiscriminator : JsonConverter<Person>
    {
        enum TypeDiscriminator
        {
            Customer = 1,
            Employee = 2
        }

        public override bool CanConvert(Type typeToConvert) =>
            typeof(Person).IsAssignableFrom(typeToConvert);

        public override Person Read(
            ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions
options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
                throw new JsonException();
            }

            reader.Read();
            if (reader.TokenType != JsonTokenType.PropertyName)
            {
                throw new JsonException();
            }

            string? propertyName = reader.GetString();
            if (propertyName != "TypeDiscriminator")
            {
                throw new JsonException();
            }

            reader.Read();
            if (reader.TokenType != JsonTokenType.Number)
            {
                throw new JsonException();
```

```
            }

            TypeDiscriminator typeDiscriminator =
(TypeDiscriminator)reader.GetInt32();
            Person person = typeDiscriminator switch
            {
                TypeDiscriminator.Customer => new Customer(),
                TypeDiscriminator.Employee => new Employee(),
                _ => throw new JsonException()
            };

            while (reader.Read())
            {
                if (reader.TokenType == JsonTokenType.EndObject)
                {
                    return person;
                }

                if (reader.TokenType == JsonTokenType.PropertyName)
                {
                    propertyName = reader.GetString();
                    reader.Read();
                    switch (propertyName)
                    {
                        case "CreditLimit":
                            decimal creditLimit = reader.GetDecimal();
                            ((Customer)person).CreditLimit = creditLimit;
                            break;
                        case "OfficeNumber":
                            string? officeNumber = reader.GetString();
                            ((Employee)person).OfficeNumber = officeNumber;
                            break;
                        case "Name":
                            string? name = reader.GetString();
                            person.Name = name;
                            break;
                    }
                }
            }

            throw new JsonException();
        }

        public override void Write(
            Utf8JsonWriter writer, Person person, JsonSerializerOptions options)
        {
            writer.WriteStartObject();

            if (person is Customer customer)
            {
                writer.WriteNumber("TypeDiscriminator",
(int)TypeDiscriminator.Customer);
                writer.WriteNumber("CreditLimit", customer.CreditLimit);
            }
            else if (person is Employee employee)
```

```
            {
                writer.WriteNumber("TypeDiscriminator",
(int)TypeDiscriminator.Employee);
                writer.WriteString("OfficeNumber", employee.OfficeNumber);
            }

            writer.WriteString("Name", person.Name);

            writer.WriteEndObject();
        }
    }
}
```

다음 코드는 변환기를 등록합니다.

C#

```
var serializeOptions = new JsonSerializerOptions();
serializeOptions.Converters.Add(new PersonConverterWithTypeDiscriminator());
```

변환기는 동일한 직렬화 변환기를 사용하여 만든 JSON을 역직렬화할 수 있습니다. 예를 들면 다음과 같습니다.

JSON

```
[
  {
    "TypeDiscriminator": 1,
    "CreditLimit": 10000,
    "Name": "John"
  },
  {
    "TypeDiscriminator": 2,
    "OfficeNumber": "555-1234",
    "Name": "Nancy"
  }
]
```

이전 예제의 변환기 코드는 각 속성을 수동으로 읽고 씁니다. 대신 `Deserialize` 또는 `Serialize`를 호출하여 일부 작업을 수행할 수 있습니다. 예제는 이 StackOverflow 게시물을 참조하세요.

다형 역직렬화를 수행하는 다른 방법

`Deserialize` 메서드에서 `Read`를 호출할 수 있습니다.

- `Utf8JsonReader` 인스턴스의 복제본을 만듭니다. `Utf8JsonReader`는 구조체이므로 대입문만 있으면 됩니다.
- 복제본을 사용하여 판별자 토큰을 끝까지 읽습니다.

- 필요한 형식을 알고 나면 원래 `Deserialize` 인스턴스를 사용하여 `Reader`를 호출합니다. 원래 `Deserialize` 인스턴스가 여전히 위치하여 begin 개체 토큰을 읽을 수 있으므로 `Reader`를 호출할 수 있습니다.

이 메서드의 단점은 변환기를 `Deserialize`에 등록하는 원래 옵션 인스턴스를 전달할 수 없다는 것입니다. 이렇게 하면 필수 속성에 설명된 대로 스택 오버플로가 발생합니다. 다음 예제에서는 이 대안을 사용하는 `Read` 메서드를 보여줍니다.

C#

```
public override Person Read(
    ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
{
    Utf8JsonReader readerClone = reader;

    if (readerClone.TokenType != JsonTokenType.StartObject)
    {
        throw new JsonException();
    }

    readerClone.Read();
    if (readerClone.TokenType != JsonTokenType.PropertyName)
    {
        throw new JsonException();
    }

    string? propertyName = readerClone.GetString();
    if (propertyName != "TypeDiscriminator")
    {
        throw new JsonException();
    }

    readerClone.Read();
    if (readerClone.TokenType != JsonTokenType.Number)
    {
        throw new JsonException();
    }

    TypeDiscriminator typeDiscriminator = (TypeDiscriminator)readerClone.GetInt32();
    Person person = typeDiscriminator switch
    {
        TypeDiscriminator.Customer => JsonSerializer.Deserialize<Customer>(ref
reader)!,
        TypeDiscriminator.Employee => JsonSerializer.Deserialize<Employee>(ref
reader)!,
        _ => throw new JsonException()
    };
    return person;
}
```

`Stack` 형식에 대한 왕복 지원

JSON 문자열을 `Stack` 개체로 역직렬화한 다음 해당 개체를 직렬화하는 경우 스택의 내용이 역순으로 표시됩니다. 이 동작은 다음 형식 및 인터페이스, 그리고 이러한 형식에서 파생되는 사용자 정의 형식에 적용됩니다.

- Stack
- Stack<T>
- ConcurrentStack<T>
- ImmutableStack<T>
- IImmutableStack<T>

스택에서 원래 순서를 유지하는 serialization 및 deserialization을 지원하려면 사용자 지정 변환기가 필요합니다.

다음 코드는 `Stack<T>` 개체에 대한 라운드트립을 가능하게 하는 사용자 지정 변환기를 보여 줍니다.

C#

```
using System.Diagnostics;
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class JsonConverterFactoryForStackOfT : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
            => typeToConvert.IsGenericType
            && typeToConvert.GetGenericTypeDefinition() == typeof(Stack<>);

        public override JsonConverter CreateConverter(
            Type typeToConvert, JsonSerializerOptions options)
        {
            Debug.Assert(typeToConvert.IsGenericType &&
                typeToConvert.GetGenericTypeDefinition() == typeof(Stack<>));

            Type elementType = typeToConvert.GetGenericArguments()[0];

            JsonConverter converter = (JsonConverter)Activator.CreateInstance(
                typeof(JsonConverterForStackOfT<>)
                    .MakeGenericType(new Type[] { elementType }),
                BindingFlags.Instance | BindingFlags.Public,
                binder: null,
                args: null,
                culture: null)!;

            return converter;
```

```
        }
    }

    public class JsonConverterForStackOfT<T> : JsonConverter<Stack<T>>
    {
        public override Stack<T> Read(
            ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions
options)
        {
            if (reader.TokenType != JsonTokenType.StartArray)
            {
                throw new JsonException();
            }
            reader.Read();

            var elements = new Stack<T>();

            while (reader.TokenType != JsonTokenType.EndArray)
            {
                elements.Push(JsonSerializer.Deserialize<T>(ref reader, options)!);

                reader.Read();
            }

            return elements;
        }

        public override void Write(
            Utf8JsonWriter writer, Stack<T> value, JsonSerializerOptions options)
        {
            writer.WriteStartArray();

            var reversed = new Stack<T>(value);

            foreach (T item in reversed)
            {
                JsonSerializer.Serialize(writer, item, options);
            }

            writer.WriteEndArray();
        }
    }
}
```

다음 코드는 변환기를 등록합니다.

C#

```
var options = new JsonSerializerOptions();
options.Converters.Add(new JsonConverterFactoryForStackOfT());
```

기본 시스템 변환기를 사용

일부 시나리오에서는 사용자 지정 변환기에서 필요에 따라 기본 시스템 변환기를 사용할 수 있습니다. 이렇게 하려면 다음 예제와 같이 JsonSerializerOptions.Default 속성에서 시스템 변환기를 가져옵니다.

C#

```
public class MyCustomConverter : JsonConverter<int>
{
    private readonly static JsonConverter<int> s_defaultConverter =
        (JsonConverter<int>)JsonSerializerOptions.Default.GetConverter(typeof(int));

    // Custom serialization logic
    public override void Write(
        Utf8JsonWriter writer, int value, JsonSerializerOptions options)
    {
        writer.WriteStringValue(value.ToString());
    }

    // Fall back to default deserialization logic
    public override int Read(
        ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions
options)
    {
        return s_defaultConverter.Read(ref reader, typeToConvert, options);
    }
}
```

Null 값 처리

기본적으로 직렬 변환기는 null 값을 다음과 같이 처리합니다.

- 참조 형식 및 Nullable<T> 형식:
  - serialization에서 `null`는 사용자 지정 변환기에 전달하지 않습니다.
  - 역직렬화 시 `JsonTokenType.Null`는 사용자 지정 변환기로 전달되지 않습니다.
  - 역직렬화 시 `null` 인스턴스를 반환합니다.
  - 직렬화 시 기록기로 직접 `null`을 씁니다.

- null을 허용하지 않는 값 형식:
  - 역직렬화 시 사용자 지정 변환기에 `JsonTokenType.Null`을 전달합니다. (사용자 지정 변환기를 사용할 수 없는 경우 형식의 내부 변환기에서 `JsonException` 예외가 throw됩니다.)

이 null 처리 동작은 주로 변환기에 대한 추가 호출을 건너뛰어 성능을 최적화하기 위한 것입니다. 또한 nullable 형식의 변환기가 모든 `null` 및 `Read` 메서드 재정의가 시작될 때 강제로 `Write`을 확인하지 않도록 합니다.

사용자 지정 변환기가 참조 또는 값 형식의 `null`을 처리할 수 있게 하려면 다음 예제와 같이 JsonConverter<T>.HandleNull를 반환하도록 `true`을 재정의하세요.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace CustomConverterHandleNull
{
    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        [JsonConverter(typeof(DescriptionConverter))]
        public string? Description { get; set; }
    }

    public class DescriptionConverter : JsonConverter<string>
    {
        public override bool HandleNull => true;

        public override string Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            reader.GetString() ?? "No description provided.";

        public override void Write(
            Utf8JsonWriter writer,
            string value,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(value);
    }

    public class Program
    {
        public static void Run()
        {
            string json = @"{""x"":1,""y"":2,""Description"":null}";

            Point point = JsonSerializer.Deserialize<Point>(json)!;
            Console.WriteLine($"Description: {point.Description}");
        }
    }
}

// Produces output like the following example:
//
//Description: No description provided.
```

참조 유지

기본적으로 참조 데이터는 Serialize 또는 Deserialize에 대한 각 호출에 대해서만 캐시됩니다. 한 `Serialize`/`Deserialize` 호출에서 다른 호출로의 참조를 유지하려면 ReferenceResolver `Serialize`/의 호출 사이트에 `Deserialize` 인스턴스를 루트합니다. 다음 코드에서는 이 시나리오의 예제를 보여줍니다.

- `Company` 형식의 사용자 지정 변환기를 작성합니다.
- `Supervisor` 속성을 수동으로 직렬화하지 않으려는데, 이 속성은 `Employee`입니다. 직렬 변환기에 위임하려고 하며 이미 저장한 참조도 유지하려고 합니다.

`Employee` 및 `Company` 클래스는 다음과 같습니다.

C#

```
public class Employee
{
    public string? Name { get; set; }
    public Employee? Manager { get; set; }
    public List<Employee>? DirectReports { get; set; }
    public Company? Company { get; set; }
}

public class Company
{
    public string? Name { get; set; }
    public Employee? Supervisor { get; set; }
}
```

변환기는 다음과 같이 표시됩니다.

C#

```
class CompanyConverter : JsonConverter<Company>
{
    public override Company Read(ref Utf8JsonReader reader, Type typeToConvert,
JsonSerializerOptions options)
    {
        throw new NotImplementedException();
    }

    public override void Write(Utf8JsonWriter writer, Company value,
JsonSerializerOptions options)
    {
        writer.WriteStartObject();

        writer.WriteString("Name", value.Name);

        writer.WritePropertyName("Supervisor");
```

```
        JsonSerializer.Serialize(writer, value.Supervisor, options);

        writer.WriteEndObject();
    }
}
```

ReferenceResolver에서 파생되는 클래스는 참조를 사전에 저장합니다.

C#

```
class MyReferenceResolver : ReferenceResolver
{
    private uint _referenceCount;
    private readonly Dictionary<string, object> _referenceIdToObjectMap = new ();
    private readonly Dictionary<object, string> _objectToReferenceIdMap = new
(ReferenceEqualityComparer.Instance);

    public override void AddReference(string referenceId, object value)
    {
        if (!_referenceIdToObjectMap.TryAdd(referenceId, value))
        {
            throw new JsonException();
        }
    }

    public override string GetReference(object value, out bool alreadyExists)
    {
        if (_objectToReferenceIdMap.TryGetValue(value, out string? referenceId))
        {
            alreadyExists = true;
        }
        else
        {
            _referenceCount++;
            referenceId = _referenceCount.ToString();
            _objectToReferenceIdMap.Add(value, referenceId);
            alreadyExists = false;
        }

        return referenceId;
    }

    public override object ResolveReference(string referenceId)
    {
        if (!_referenceIdToObjectMap.TryGetValue(referenceId, out object? value))
        {
            throw new JsonException();
        }

        return value;
    }
}
```

ReferenceHandler에서 파생되는 클래스는 `MyReferenceResolver`의 인스턴스를 보유하며 (이 예제에서 `Reset`으로 명명된 메서드에서) 필요한 경우에만 새 인스턴스를 만듭니다.

C#

```
class MyReferenceHandler : ReferenceHandler
{
    public MyReferenceHandler() => Reset();

    private ReferenceResolver? _rootedResolver;
    public override ReferenceResolver CreateResolver() => _rootedResolver!;
    public void Reset() => _rootedResolver = new MyReferenceResolver();
}
```

샘플 코드는 직렬 변환기를 호출할 때 JsonSerializerOptions 속성이 ReferenceHandler의 인스턴스로 설정된 `MyReferenceHandler` 인스턴스를 사용합니다. 이 패턴을 따를 때는 직렬화를 마쳤을 때 `ReferenceResolver` 사전을 다시 설정하여 영원히 증가하지 않도록 해야 합니다.

C#

```
var options = new JsonSerializerOptions();

options.Converters.Add(new CompanyConverter());
var myReferenceHandler = new MyReferenceHandler();
options.ReferenceHandler = myReferenceHandler;
options.DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull;
options.WriteIndented = true;

string str = JsonSerializer.Serialize(tyler, options);

// Reset after serializing to avoid out of bounds memory growth in the resolver.
myReferenceHandler.Reset();
```

앞의 예제에서는 직렬화만 수행하지만 역직렬화를 위해 유사한 방법을 채택할 수 있습니다.

사용자 지정 변환기의 ReferenceResolver 제한 사항

Preserve를 사용할 때 serializer가 사용자 정의 변환기를 호출하면 참조 처리 상태가 유지되지 않는다는 점에 유의하십시오. 즉, 참조 보존을 사용하도록 설정하여 직렬화 또는 역직렬화되는 개체 그래프의 일부인 형식에 대한 사용자 지정 변환기가 있는 경우 변환기와 중첩된 serialization 호출은 현재 ReferenceResolver 인스턴스에 액세스할 수 없습니다.

기타 사용자 지정 변환기 샘플

Newtonsoft.Json에서 System.Text.Json로 마이그레이션 문서에는 사용자 지정 변환기의 추가 샘플이 포함되어 있습니다.

소스 코드의 `System.Text.Json.Serialization`에는 다음과 같은 다른 사용자 지정 변환기 샘플이 포함되어 있습니다.

- 역직렬화할 때 null을 0으로 변환하는 Int32 변환기
- 역직렬화할 때 문자열 및 숫자 값을 모두 허용하는 Int32 변환기
- 열거형 변환기
- 외부 데이터를 허용하는 List<T>> 변환기
- 쉼표로 구분된 숫자 목록과 함께 작동하는 Long[] 변환기

기존 기본 제공 변환기의 동작을 수정하는 변환기를 만들어야 하는 경우 기존 변환기의 소스 코드를 가져와 사용자 지정을 위한 시작 지점으로 사용할 수 있습니다.

추가 리소스

- 기본 제공 변환기 소스 코드
- System.Text.Json 개요
- JSON 직렬화 및 역직렬화 방법

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

JSON 계약 사용자 지정

아티클 • 2023. 06. 16.

System.Text.Json 라이브러리는 해당 형식을 직렬화하고 역직렬화하는 방법을 정의하는 각 .NET 형식에 대해 JSON *계약*을 생성합니다. 계약은 해당 형식의 셰이프에서 파생됩니다. 여기에는 속성 및 필드와 같은 특성과 IEnumerable 또는 IDictionary 인터페이스를 구현하는지 여부가 포함됩니다. 형식은 리플렉션을 사용하여 런타임에 계약에 매핑되거나 또는 소스 생성기를 사용하여 컴파일 시간에 계약에 매핑됩니다.

.NET 7부터 이러한 JSON 계약을 사용자 지정하여 형식이 JSON으로 변환되는 방식을 보다 자세히 제어할 수 있으며 그 반대의 경우도 마찬가지입니다. 다음 목록에서는 직렬화 및 역직렬화에 수행할 수 있는 사용자 지정 유형의 몇 가지 예만 보여줍니다.

- 프라이빗 필드 및 속성을 직렬화합니다.
- 단일 속성에 대해 여러 이름을 지원합니다(예: 이전 라이브러리 버전에서 다른 이름을 사용한 경우).
- 특정 이름, 형식 또는 값이 있는 속성은 무시합니다.
- 명시적 `null` 값과 JSON 페이로드의 값 부족을 구분합니다.
- 와 같은 DataContractAttribute특성을 지원 System.Runtime.Serialization 합니다. 자세한 내용은 System.Runtime.Serialization 특성을 참조하세요.
- JSON에 대상 형식의 일부가 아닌 속성이 포함된 경우 예외를 throw합니다. 자세한 내용은 누락된 멤버 처리를 참조하세요.

옵트인하는 방법

사용자 지정에 연결하는 방법에는 두 가지가 있습니다. 두 가지 방법은 모두 직렬화해야 하는 각 형식에 대한 JsonTypeInfo 인스턴스를 제공하는 작업을 수행하는 확인자 가져오기를 포함합니다.

- DefaultJsonTypeInfoResolver() 생성자를 호출하여 JsonSerializerOptions.TypeInfoResolver를 가져오고 그 Modifiers 속성에 사용자 지정 작업을 추가합니다.

  다음은 그 예입니다.

  C#

  ```
  JsonSerializerOptions options = new()
  {
      TypeInfoResolver = new DefaultJsonTypeInfoResolver
      {
          Modifiers =
          {
  ```

```
            MyCustomModifier1,
            MyCustomModifier2
        }
    }
};
```

여러 한정자를 추가하면 순차적으로 호출됩니다.

- IJsonTypeInfoResolver를 구현하는 사용자 지정 확인자를 작성합니다.
  - 하나의 형식이 처리되지 않으면 IJsonTypeInfoResolver.GetTypeInfo는 해당 형식에 대해 `null`을 반환해야 합니다.
  - 사용자 지정 확인자(예: 기본 확인자)를 다른 사용자의 그것과 결합할 수도 있습니다. 확인자는 null이 아닌 JsonTypeInfo 값이 해당 형식에 대해 반환될 때까지 순서대로 쿼리됩니다.

구성 가능한 측면

JsonTypeInfo.Kind 속성은 변환기가 지정된 형식을 직렬화하는 방법(예: 개체 또는 배열로서 직렬화)과 그 속성이 직렬화되는지 여부를 나타냅니다. 이 속성을 쿼리하여 구성할 수 있는 형식의 JSON 계약의 측면을 확인할 수 있습니다. 이 속성은 다음의 네 가지 종류가 있습니다.

`JsonTypeInfo.Kind`	설명
JsonTypeInfoKind.Object	변환기는 형식을 JSON 개체로 직렬화하고 그 속성을 사용합니다. **이 종류는 대부분의 클래스 및 구조체 형식에 사용되며 가장 유연하게 사용할 수 있습니다.**
JsonTypeInfoKind.Enumerable	변환기는 형식을 JSON 배열로 직렬화합니다. 이 종류는 `List<T>` 및 배열과 같은 형식에 사용됩니다.
JsonTypeInfoKind.Dictionary	변환기는 해당 형식을 JSON 개체로 직렬화합니다. 이 종류는 `Dictionary<K, V>`과 같은 형식에 사용됩니다.
JsonTypeInfoKind.None	변환기는 해당 형식을 직렬화하는 방법 또는 그 형식이 사용할 `JsonTypeInfo` 속성을 지정하지 않습니다. 이 종류는 System.Object, `int` 및 `string`과 같은 형식과 사용자 지정 변환기를 사용하는 모든 형식에 사용됩니다.

한정자

한정자는 `Action<JsonTypeInfo>`이거나 또는 하나의 인수로 계약의 현재 상태를 가져오고 계약을 수정하는 JsonTypeInfo 매개 변수가 있는 메서드입니다. 예를 들어 지정한 JsonTypeInfo에서 미리 채워진 속성을 반복하여 관심 있는 속성을 찾은 다음, 해당

JsonPropertyInfo.Get 속성(직렬화용) 또는 JsonPropertyInfo.Set 속성(역직렬화용)을 수정할 수 있습니다. 또는 JsonTypeInfo.CreateJsonPropertyInfo(Type, String)을 사용하여 새 속성을 생성하고 JsonTypeInfo.Properties 컬렉션에 추가할 수 있습니다.

다음 표에서는 수행할 수 있는 수정 사항과 이를 수행하는 방법을 보여줍니다.

수정	적용 가능한 `JsonTypeInfo.Kind`	수행 방법	예
속성 값 사용자 지정	`JsonTypeInfoKind.Object`	해당 속성에 대한 JsonPropertyInfo.Get 대리자(직렬화용) 또는 JsonPropertyInfo.Set 대리자(역직렬화용)를 수정합니다.	속성 값 증가
속성 추가/제거	`JsonTypeInfoKind.Object`	JsonTypeInfo.Properties 목록에서 항목을 추가하거나 제거합니다.	프라이빗 필드 직렬화
조건부로 속성 직렬화	`JsonTypeInfoKind.Object`	해당 속성에 대한 JsonPropertyInfo.ShouldSerialize 조건자를 수정합니다.	특정 형식의 속성 무시
특정 형식에 대한 숫자 처리 사용자 지정	`JsonTypeInfoKind.None`	해당 형식의 JsonTypeInfo.NumberHandling 값을 수정합니다.	int 값을 문자열로 허용

예: 속성 값 증가

한정자가 JsonPropertyInfo.Set 대리자를 수정하여 역직렬화 시 특정 속성의 값을 증가시키는 다음 예제를 생각해 보세요. 이 예제에서는 한정자를 정의하는 것 외에도 값을 증분해야 하는 속성을 찾는 데 사용하는 새 특성도 소개합니다. *속성을 사용자 지정*하는 예제입니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization.Metadata;

namespace Serialization
{
    // Custom attribute to annotate the property
    // we want to be incremented.
    [AttributeUsage(AttributeTargets.Property)]
```

```
    class SerializationCountAttribute : Attribute
    {
    }

    // Example type to serialize and deserialize.
    class Product
    {
        public string Name { get; set; } = "";
        [SerializationCount]
        public int RoundTrips { get; set; }
    }

    public class SerializationCountExample
    {
        // Custom modifier that increments the value
        // of a specific property on deserialization.
        static void IncrementCounterModifier(JsonTypeInfo typeInfo)
        {
            foreach (JsonPropertyInfo propertyInfo in typeInfo.Properties)
            {
                if (propertyInfo.PropertyType != typeof(int))
                    continue;

                object[] serializationCountAttributes =
propertyInfo.AttributeProvider?.GetCustomAttributes(typeof(SerializationCoun
tAttribute), true) ?? Array.Empty<object>();
                SerializationCountAttribute? attribute =
serializationCountAttributes.Length == 1 ?
(SerializationCountAttribute)serializationCountAttributes[0] : null;

                if (attribute != null)
                {
                    Action<object, object?>? setProperty = propertyInfo.Set;
                    if (setProperty is not null)
                    {
                        propertyInfo.Set = (obj, value) =>
                        {
                            if (value != null)
                            {
                                // Increment the value by 1.
                                value = (int)value + 1;
                            }

                            setProperty (obj, value);
                        };
                    }
                }
            }
        }

        public static void RunIt()
        {
            var product = new Product
            {
                Name = "Aquafresh"
```

```
            };

            JsonSerializerOptions options = new()
            {
                TypeInfoResolver = new DefaultJsonTypeInfoResolver
                {
                    Modifiers = { IncrementCounterModifier }
                }
            };

            // First serialization and deserialization.
            string serialized = JsonSerializer.Serialize(product, options);
            Console.WriteLine(serialized);
            // {"Name":"Aquafresh","RoundTrips":0}

            Product deserialized = JsonSerializer.Deserialize<Product>
(serialized, options)!;
            Console.WriteLine($"{deserialized.RoundTrips}");
            // 1

            // Second serialization and deserialization.
            serialized = JsonSerializer.Serialize(deserialized, options);
            Console.WriteLine(serialized);
            // { "Name":"Aquafresh","RoundTrips":1}

            deserialized = JsonSerializer.Deserialize<Product>(serialized,
options)!;
            Console.WriteLine($"{deserialized.RoundTrips}");
            // 2
        }
    }
}
```

출력에서 `Product` 인스턴스가 역직렬화될 때마다 `RoundTrips`의 값이 증가합니다.

예제: 프라이빗 필드 직렬화

기본적으로 `System.Text.Json`은 프라이빗 필드와 속성을 무시합니다. 이 예제에서는 해당 기본값을 변경하기 위해 클래스 차원의 새 특성 `JsonIncludePrivateFieldsAttribute`를 추가합니다. 한정자가 하나의 형식에서 특성을 찾으면 해당 형식의 모든 프라이빗 필드를 JsonTypeInfo에 새 속성으로 추가합니다.

C#

```
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.Json.Serialization.Metadata;

namespace Serialization
```

```
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
    public class JsonIncludePrivateFieldsAttribute : Attribute { }

    [JsonIncludePrivateFields]
    public class Human
    {
        private string _name;
        private int _age;

        public Human()
        {
            // This constructor should be used only by deserializers.
            _name = null!;
            _age = 0;
        }

        public static Human Create(string name, int age)
        {
            Human h = new()
            {
                _name = name,
                _age = age
            };

            return h;
        }

        [JsonIgnore]
        public string Name
        {
            get => _name;
            set => throw new NotSupportedException();
        }

        [JsonIgnore]
        public int Age
        {
            get => _age;
            set => throw new NotSupportedException();
        }
    }

    public class PrivateFieldsExample
    {
        static void AddPrivateFieldsModifier(JsonTypeInfo jsonTypeInfo)
        {
            if (jsonTypeInfo.Kind != JsonTypeInfoKind.Object)
                return;

            if
(!jsonTypeInfo.Type.IsDefined(typeof(JsonIncludePrivateFieldsAttribute),
inherit: false))
                return;
```

```
            foreach (FieldInfo field in
jsonTypeInfo.Type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic))
            {
                JsonPropertyInfo jsonPropertyInfo =
jsonTypeInfo.CreateJsonPropertyInfo(field.FieldType, field.Name);
                jsonPropertyInfo.Get = field.GetValue;
                jsonPropertyInfo.Set = field.SetValue;

                jsonTypeInfo.Properties.Add(jsonPropertyInfo);
            }
        }

        public static void RunIt()
        {
            var options = new JsonSerializerOptions
            {
                TypeInfoResolver = new DefaultJsonTypeInfoResolver
                {
                    Modifiers = { AddPrivateFieldsModifier }
                }
            };

            var human = Human.Create("Julius", 37);
            string json = JsonSerializer.Serialize(human, options);
            Console.WriteLine(json);
            // {"_name":"Julius","_age":37}

            Human deserializedHuman = JsonSerializer.Deserialize<Human>
(json, options)!;
            Console.WriteLine($"[Name={deserializedHuman.Name}; Age=
{deserializedHuman.Age}]");
            // [Name=Julius; Age=37]
        }
    }
}
```

팁

프라이빗 필드 이름이 밑줄로 시작하는 경우 필드를 새 JSON 속성으로 추가할 때 이름에서 밑줄을 제거하는 것이 좋습니다.

예제: 특정 형식의 속성 무시

사용자에게 노출하지 않으려는 특정 이름 또는 형식의 속성이 모델에 있는 것일 수 있습니다. 예를 들어 자격 증명을 저장하는 속성이나 페이로드에 사용할 수 없는 일부 정보가 있을 수 있습니다.

다음 예제에서는 특정 형식인 `SecretHolder`의 속성을 필터링하는 방법을 보여줍니다. 이 작업에서는 IList<T> 확장 메서드를 사용하여 JsonTypeInfo.Properties 목록에서 지정된 형식이 있는 속성을 모두 제거합니다. 필터링된 속성은 계약에서 완전히 사라집니다. 즉, `System.Text.Json`은 직렬화 또는 역직렬화 중에는 이러한 속성을 보지 않습니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization.Metadata;

namespace Serialization
{
    class ExampleClass
    {
        public string Name { get; set; } = "";
        public SecretHolder? Secret { get; set; }
    }

    class SecretHolder
    {
        public string Value { get; set; } = "";
    }

    class IgnorePropertiesWithType
    {
        private readonly Type[] _ignoredTypes;

        public IgnorePropertiesWithType(params Type[] ignoredTypes)
            => _ignoredTypes = ignoredTypes;

        public void ModifyTypeInfo(JsonTypeInfo ti)
        {
            if (ti.Kind != JsonTypeInfoKind.Object)
                return;

            ti.Properties.RemoveAll(prop =>
_ignoredTypes.Contains(prop.PropertyType));
        }
    }

    public class IgnoreTypeExample
    {
        public static void RunIt()
        {
            var modifier = new
IgnorePropertiesWithType(typeof(SecretHolder));

            JsonSerializerOptions options = new()
            {
                TypeInfoResolver = new DefaultJsonTypeInfoResolver
                {
                    Modifiers = { modifier.ModifyTypeInfo }
                }
```

```
            };

            ExampleClass obj = new()
            {
                Name = "Password",
                Secret = new SecretHolder { Value = "MySecret" }
            };

            string output = JsonSerializer.Serialize(obj, options);
            Console.WriteLine(output);
            // {"Name":"Password"}
        }
    }

    public static class ListHelpers
    {
        // IList<T> implementation of List<T>.RemoveAll method.
        public static void RemoveAll<T>(this IList<T> list, Predicate<T>
predicate)
        {
            for (int i = 0; i < list.Count; i++)
            {
                if (predicate(list[i]))
                {
                    list.RemoveAt(i--);
                }
            }
        }
    }
}
```

예제: int 값을 문자열로 허용

아마도 입력 JSON은 숫자 형식 중 하나의 주위에 따옴표를 포함할 수 있지만 다른 형식에는 포함하지 않을 수 있습니다. 클래스를 제어할 수 있는 경우 이 문제를 해결하기 위해 해당 형식에 JsonNumberHandlingAttribute를 배치할 수 있지만 실제로는 배치하지 않습니다. .NET 7 이전에는 이 동작을 수정하기 위해 사용자 지정 변환기를 작성해야 하며, 이를 위해서는 상당한 코드를 작성해야 합니다. 계약 사용자 지정을 사용하여 모든 형식에 대한 숫자 처리 동작을 사용자 지정할 수 있습니다.

다음 예제에서는 모든 `int` 값에 대한 동작을 변경합니다. 이 예제는 모든 형식에 적용하거나 모든 형식의 특정 속성에 적용하도록 쉽게 조정할 수 있습니다.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.Json.Serialization.Metadata;
```

```
namespace Serialization
{
    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }
    }

    public class AllowIntsAsStringsExample
    {
        static void SetNumberHandlingModifier(JsonTypeInfo jsonTypeInfo)
        {
            if (jsonTypeInfo.Type == typeof(int))
            {
                jsonTypeInfo.NumberHandling =
JsonNumberHandling.AllowReadingFromString;
            }
        }

        public static void RunIt()
        {
            JsonSerializerOptions options = new()
            {
                TypeInfoResolver = new DefaultJsonTypeInfoResolver
                {
                    Modifiers = { SetNumberHandlingModifier }
                }
            };

            // Triple-quote syntax is a C# 11 feature.
            Point point = JsonSerializer.Deserialize<Point>("""
{"X":"12","Y":"3"}""", options)!;
            Console.WriteLine($"({point.X},{point.Y})");
            // (12,3)
        }
    }
}
```

문자열에서 `int` 값을 읽을 수 있도록 허용하는 한정자가 없다면 프로그램은 다음과 같이 예외로 끝났을 것입니다.

> 처리되지 않은 예외가 발생했습니다. System.Text.Json.JsonException: JSON 값을 System.Int32로 변환할 수 없습니다. 경로: $.X | LineNumber: 0 | BytePositionInLine: 9.

serialization을 사용자 지정하는 다른 방법

계약을 사용자 지정하는 것 외에도 다음을 포함하여 직렬화 및 역직렬화 동작에 영향을 미치는 다른 방법이 있습니다.

- JsonAttribute에서 파생된 특성(예: JsonIgnoreAttribute 및 JsonPropertyOrderAttribute)을 사용합니다.
- 예를 들어 JsonSerializerOptions를 수정하여 명명 정책을 설정하거나 열거형 값을 숫자 대신 문자열로 직렬화합니다.
- JSON을 작성하는 실제 작업을 수행하는 사용자 지정 변환기를 작성하고 역직렬화 중에 개체를 생성합니다.

계약 사용자 지정은 이러한 기존 사용자 지정에 비해 개선되었는데, 그 이유는 특성을 추가할 형식에 액세스할 수 없으며 사용자 지정 변환기를 작성하는 것은 복잡하고 오히려 성능이 저하되기 때문입니다.

추가 정보

- JSON 계약 사용자 지정(블로그 게시물)
- .NET 7의 System.Text.Json의 새로운 기능(블로그 게시물)

JSON 스키마 내보내기

JsonSchemaExporter.NET 9에 도입된 클래스를 사용하면 JsonSerializerOptions 또는 JsonTypeInfo 인스턴스를 사용하여 .NET 형식에서 JSON 스키마 문서를 추출할 수 있습니다. 결과 스키마는 .NET 형식에 대한 JSON serialization 계약의 사양을 제공합니다. 스키마는 직렬화할 개체와 역직렬화할 수 있는 요소의 형태를 설명합니다.

다음 코드 조각은 예제를 보여 줍니다.

C#

```
public static void SimpleExtraction()
{
    JsonSerializerOptions options = JsonSerializerOptions.Default;
    JsonNode schema = options.GetJsonSchemaAsNode(typeof(Person));
    Console.WriteLine(schema.ToString());
    //{
    //  "type": ["object", "null"],
    //  "properties": {
    //    "Name": { "type": "string" },
    //    "Age": { "type": "integer" },
    //    "Address": { "type": ["string", "null"], "default": null }
    //  },
    //  "required": ["Name", "Age"]
    //}
}

record Person(string Name, int Age, string? Address = null);
```

이 예제에서 볼 수 있듯이 내보내기는 null을 허용하는 속성과 null을 허용하지 않는 속성을 구분하며 생성자 매개 변수가 선택 사항인지 여부에 따라 `required` 키워드를 채웁니다.

스키마 출력 구성

JsonSerializerOptions 메서드를 호출하는 JsonTypeInfo 또는 GetJsonSchemaAsNode 인스턴스에 지정된 구성에 따라 스키마 출력에 영향을 줄 수 있습니다. 다음은 명명 정책을 KebabCaseUpper로 설정하고, 숫자를 문자열로 쓰고, 매핑되지 않은 속성을 허용하지 않는 예제입니다.

C#

```
public static void CustomExtraction()
{
    JsonSerializerOptions options = new(JsonSerializerOptions.Default)
    {
        PropertyNamingPolicy = JsonNamingPolicy.KebabCaseUpper,
        NumberHandling = JsonNumberHandling.WriteAsString,
```

```
        UnmappedMemberHandling = JsonUnmappedMemberHandling.Disallow,
    };

    JsonNode schema = options.GetJsonSchemaAsNode(typeof(MyPoco));
    Console.WriteLine(schema.ToString());
    //{
    //  "type": ["object", "null"],
    //  "properties": {
    //    "NUMERIC-VALUE": {
    //      "type": ["string", "integer"],
    //      "pattern": "^-?(?:0|[1-9]\\d*)$"
    //    }
    //  },
    //  "additionalProperties": false
    //}
}

class MyPoco
{
    public int NumericValue { get; init; }
}
```

JsonSchemaExporterOptions 구성 유형을 사용하여 생성된 스키마를 추가로 제어할 수 있습니다. 다음은 TreatNullObliviousAsNonNullable 속성을 `true`로 설정하여 루트 수준 형식을 null을 허용하지 않는 형식으로 표시하는 예제입니다.

C#

```
public static void CustomExtraction()
{
    JsonSerializerOptions options = JsonSerializerOptions.Default;
    JsonSchemaExporterOptions exporterOptions = new()
    {
        TreatNullObliviousAsNonNullable = true,
    };

    JsonNode schema = options.GetJsonSchemaAsNode(typeof(Person), exporterOptions);
    Console.WriteLine(schema.ToString());
    //{
    //  "type": "object",
    //  "properties": {
    //    "Name": { "type": "string" }
    //  },
    //  "required": ["Name"]
    //}
}

record Person(string Name);
```

생성된 스키마 변환

TransformSchemaNode 대리자를 지정하여 생성된 스키마 노드에 고유한 변환을 적용할 수 있습니다. 다음 예제에서는 DescriptionAttribute 주석의 텍스트를 생성된 스키마에 통합합니다.

C#

```
JsonSchemaExporterOptions exporterOptions = new()
{
    TransformSchemaNode = (context, schema) =>
    {
        // Determine if a type or property and extract the relevant attribute
provider.
        ICustomAttributeProvider? attributeProvider = context.PropertyInfo is not
null
            ? context.PropertyInfo.AttributeProvider
            : context.TypeInfo.Type;

        // Look up any description attributes.
        DescriptionAttribute? descriptionAttr = attributeProvider?
            .GetCustomAttributes(inherit: true)
            .Select(attr => attr as DescriptionAttribute)
            .FirstOrDefault(attr => attr is not null);

        // Apply description attribute to the generated schema.
        if (descriptionAttr != null)
        {
            if (schema is not JsonObject jObj)
            {
                // Handle the case where the schema is a Boolean.
                JsonValueKind valueKind = schema.GetValueKind();
                Debug.Assert(valueKind is JsonValueKind.True or
JsonValueKind.False);
                schema = jObj = new JsonObject();
                if (valueKind is JsonValueKind.False)
                {
                    jObj.Add("not", true);
                }
            }

            jObj.Insert(0, "description", descriptionAttr.Description);
        }

        return schema;
    }
};
```

다음 코드 예제에서는 `description` 주석에서 DescriptionAttribute 키워드 소스를 통합하는 스키마를 생성합니다.

C#

```
JsonSerializerOptions options = JsonSerializerOptions.Default;
JsonNode schema = options.GetJsonSchemaAsNode(typeof(Person), exporterOptions);
```

```
Console.WriteLine(schema.ToString());
//{
//  "description": "A person",
//  "type": ["object", "null"],
//  "properties": {
//    "Name": { "description": "The name of the person", "type": "string" }
//  },
//  "required": ["Name"]
//}
```

C#

```
[Description("A person")]
record Person([property: Description("The name of the person")] string Name);
```

Last updated on 2026. 02. 24.

XML 및 SOAP 직렬화

2025. 06. 17.

XML serialization은 개체의 공용 필드와 속성, 메서드의 매개 변수 및 반환 값을 특정 XSD(XML 스키마 정의 언어) 문서를 준수하는 XML 스트림으로 변환(serialize)합니다. XML serialization은 스토리지 또는 전송을 위해 직렬 형식(이 경우 XML)으로 변환되는 공용 속성 및 필드가 있는 강력한 형식의 클래스를 생성합니다.

XML은 개방형 표준이므로 플랫폼에 관계없이 필요에 따라 모든 애플리케이션에서 XML 스트림을 처리할 수 있습니다. 예를 들어 ASP.NET 사용하여 만든 XML 웹 서비스는 클래스를 사용하여 XmlSerializer 인터넷 또는 인트라넷에서 XML 웹 서비스 애플리케이션 간에 데이터를 전달하는 XML 스트림을 만듭니다. 반대로 역직렬화는 이러한 XML 스트림을 사용하고 개체를 다시 구성합니다.

XML serialization을 사용하여 SOAP 사양을 준수하는 XML 스트림으로 개체를 직렬화할 수도 있습니다. SOAP는 XML을 사용하여 프로시저 호출을 전송하도록 특별히 설계된 XML 기반 프로토콜입니다.

개체를 직렬화하거나 역직렬화하려면 클래스를 XmlSerializer 사용합니다. serialize할 클래스를 만들려면 XML 스키마 정의 도구를 사용합니다.

참고하십시오

- 이진 직렬화
- ASP.NET 및 XML Web Services 클라이언트를 사용하여 만든 XML Web Services

XML 직렬화

2025. 06. 17.

Serialization은 개체를 쉽게 전송할 수 있는 폼으로 변환하는 프로세스입니다. 예를 들어 클라이언트와 서버 간에 HTTP를 사용하여 개체를 직렬화하고 인터넷을 통해 전송할 수 있습니다. 한편, 역직렬화는 스트림에서 객체를 재구성합니다.

XML serialization은 개체의 공용 필드 및 속성 값만 XML 스트림으로 직렬화합니다. XML serialization에는 형식 정보가 포함되지 않습니다. 예를 들어 **Library** 네임스페이스에 **Book** 개체가 있는 경우 동일한 형식의 개체로 역직렬화된다는 보장은 없습니다.

> **참고**
>
> XML serialization은 메서드, 인덱서, 프라이빗 필드 또는 읽기 전용 속성(읽기 전용 컬렉션 제외)을 변환하지 않습니다. 공용 및 프라이빗 개체의 모든 필드와 속성을 직렬화하려면 XML serialization 대신 사용합니다 DataContractSerializer .

XML serialization의 중심 클래스는 XmlSerializer 클래스이며, 이 클래스에서 가장 중요한 메서드는 **Serialize** 및 **Deserialize** 메서드입니다. XmlSerializer는 C# 파일을 만들어서 .dll 파일로 컴파일하여 직렬화를 수행합니다. XML serializer 생성기 도구(Sgen.exe)는 애플리케이션과 함께 배포하기 위해 이러한 serialization 어셈블리를 미리 생성하고 시작 성능을 향상하도록 설계되었습니다. **XmlSerializer**에서 생성된 XML 스트림은 W3C(World Wide Web Consortium) XSD(XML 스키마 정의 언어) 1.0 권장 사항을 준수합니다. 또한 생성된 데이터 형식은 "XML 스키마 2부: 데이터 형식"이라는 문서를 준수합니다.

개체의 데이터는 클래스, 필드, 속성, 기본 형식, 배열 및 **XmlElement** 또는 **XmlAttribute** 개체 형식의 포함된 XML과 같은 프로그래밍 언어 구문을 사용하여 설명합니다. 고유한 클래스를 만들거나, 특성으로 주석을 추가하거나, XML 스키마 정의 도구를 사용하여 기존 XML 스키마를 기반으로 클래스를 생성할 수 있습니다.

XML 스키마가 있는 경우 XML 스키마 정의 도구를 실행하여 스키마에 강력하게 형식화되고 특성으로 주석이 추가된 클래스 집합을 생성할 수 있습니다. 이러한 클래스의 인스턴스가 serialize되면 생성된 XML은 XML 스키마를 준수합니다. 이러한 클래스를 사용하면 생성된 XML이 XML 스키마를 준수함을 보장하면서 쉽게 조작할 수 있는 개체 모델에 대해 프로그래밍할 수 있습니다. XmlReader 및 **XmlWriter** 클래스와 같은 .NET의 다른 클래스를 사용하여 **XML** 스트림을 구문 분석하고 작성하는 대신 사용할 수 있습니다. 자세한 내용은 XML 문서 및 데이터를 참조하세요. 이러한 클래스를 사용하면 XML 스트림을 구문 분석할 수 있습니다. 반면 XML 스트림이 알려진 XML 스키마를 준수해야 하는 경우 **XmlSerializer** 를 사용합니다.

특성은 XML 스트림의 XML 네임스페이스, 요소 이름, 특성 이름 등을 설정할 수 있도록 **XmlSerializer** 클래스에서 생성된 XML 스트림을 제어합니다. 이러한 특성 및 XML serialization을

제어하는 방법에 대한 자세한 내용은 특성을 사용하여 XML Serialization 제어를 참조하세요. 생성된 XML을 제어하는 데 사용되는 특성의 테이블은 XML Serialization을 제어하는 특성을 참조하세요.

XmlSerializer 클래스는 개체를 추가로 직렬화하고 인코딩된 SOAP XML 스트림을 생성할 수 있습니다. 생성된 XML은 "SOAP(Simple Object Access Protocol) 1.1"이라는 World Wide Web 컨소시엄 문서의 섹션 5를 준수합니다. 이 프로세스에 대한 자세한 내용은 방법: 개체를 SOAP-Encoded XML 스트림으로 serialize하는 방법을 참조하세요. 생성된 XML을 제어하는 특성의 테이블은 인코딩된 SOAP Serialization을 제어하는 특성을 참조하세요.

XmlSerializer 클래스는 XML 웹 서비스에 의해 생성되고 전달된 SOAP 메시지를 생성합니다. SOAP 메시지를 제어하려면 클래스에 특성을 적용하고 XML 웹 서비스 파일(.asmx)에 있는 값, 매개 변수 및 필드를 반환할 수 있습니다. XML 웹 서비스에서 리터럴 또는 인코딩된 SOAP 스타일을 사용할 수 있으므로 "XML Serialization을 제어하는 특성" 및 "인코딩된 SOAP Serialization을 제어하는 특성"에 나열된 특성을 모두 사용할 수 있습니다. 특성을 사용하여 XML 웹 서비스에서 생성된 XML을 제어하는 방법에 대한 자세한 내용은 XML Web Services를 사용한 XML Serialization을 참조하세요. SOAP 및 XML 웹 서비스에 대한 자세한 내용은 SOAP 메시지 서식 사용자 지정을 참조하세요.

XmlSerializer 애플리케이션에 대한 보안 고려 사항

XmlSerializer를 사용하는 애플리케이션을 만들 때는 다음 항목 및 해당 의미에 유의해야 합니다.

- **XmlSerializer**는 C#(.cs) 파일을 만들고 TEMP 환경 변수로 명명된 디렉터리의 .dll 파일로 컴파일합니다. serialization은 해당 DLL에서 발생합니다.

  > **참고**
  >
  > 이러한 serialization 어셈블리는 SGen.exe 도구를 사용하여 미리 생성하고 서명할 수 있습니다. 웹 서비스 서버에서는 작동하지 않습니다. 즉, 클라이언트 사용 및 수동 직렬화에만 해당합니다.

  코드와 DLL은 생성 및 컴파일 시 악의적인 프로세스에 취약합니다. 둘 이상의 사용자가 TEMP 디렉터리를 공유할 수 있습니다. 두 계정에 서로 다른 보안 권한이 있고 더 높은 권한의 계정이 **XmlSerializer**를 사용하여 애플리케이션을 실행하는 경우 TEMP 디렉터리를 공유하는 것은 위험합니다. 이 경우 한 사용자가 컴파일된 .cs 또는 .dll 파일을 대체하여 컴퓨터의 보안을 위반할 수 있습니다. 이 문제를 제거하려면 항상 컴퓨터의 각 계정에 고유한 프로필이 있는지 확인합니다. 기본적으로 TEMP 환경 변수는 각 계정에 대해 다른 디렉터리를 가리킵니다.

- 악의적인 사용자가 XML 데이터의 연속 스트림을 웹 서버(서비스 거부 공격)로 보내는 경우 **XmlSerializer** 는 컴퓨터가 리소스에서 부족할 때까지 데이터를 계속 처리합니다.

  IIS(인터넷 정보 서비스)를 실행하는 컴퓨터를 사용하고 애플리케이션이 IIS 내에서 실행되는 경우 이러한 종류의 공격은 제거됩니다. IIS에는 설정된 양보다 긴 스트림을 처리하지 않는 게이트가 있습니다(기본값은 4KB). IIS를 사용하지 않고 **XmlSerializer**를 사용하여 역직렬화하는 애플리케이션을 만드는 경우 서비스 거부 공격을 방지하는 유사한 게이트를 구현해야 합니다.

- **XmlSerializer**는 데이터를 직렬화하고 지정된 형식을 사용하여 모든 코드를 실행합니다.

  악의적인 개체가 위협을 표시하는 방법에는 두 가지가 있습니다. 악성 코드를 실행하거나 **XmlSerializer**에서 만든 C# 파일에 악성 코드를 삽입할 수 있습니다. 두 번째 경우 악의적인 개체가 **XmlSerializer**에서 만든 C# 파일에 코드를 삽입할 수 있다는 이론적인 가능성이 있습니다. 이 문제는 철저히 조사되었으며 이러한 공격은 가능성이 낮다고 간주되지만 알 수 없는 신뢰할 수 없는 형식으로 데이터를 직렬화하지 않도록 주의해야 합니다.

- 직렬화된 중요한 데이터는 취약할 수 있습니다.

  XmlSerializer가 데이터를 직렬화한 후에는 XML 파일 또는 다른 데이터 저장소로 저장할 수 있습니다. 데이터 저장소를 다른 프로세스에서 사용할 수 있거나 인트라넷 또는 인터넷에 표시되는 경우 데이터를 도난당하고 악의적으로 사용할 수 있습니다. 예를 들어 신용 카드 번호를 포함하는 주문을 직렬화하는 애플리케이션을 만드는 경우 데이터는 매우 중요합니다. 이를 방지하려면 항상 데이터에 대한 저장소를 보호하고 비공개로 유지하는 단계를 수행합니다.

단순 클래스의 직렬화

다음 코드 예제에서는 공용 필드가 있는 기본 클래스를 보여줍니다.

C#

```
public class OrderForm
{
    public DateTime OrderDate;
}
```

이 클래스의 인스턴스가 직렬화되면 다음과 유사할 수 있습니다.

XML

```
<OrderForm>
    <OrderDate>12/12/01</OrderDate>
</OrderForm>
```

serialization에 대한 자세한 예제는 XML Serialization 예제를 참조하세요.

Serialize할 수 있는 항목

XmlSerializer 클래스를 사용하여 다음 항목을 serialize할 수 있습니다.

- 공용 읽기/쓰기 속성 및 공용 클래스의 필드입니다.
- **ICollection** 또는 **IEnumerable**을 구현하는 클래스입니다.

  > **참고**
  >
  > 컬렉션만 직렬화되고 공용 속성은 serialize되지 않습니다.

- **XmlElement** 개체입니다.
- **XmlNode** 개체입니다.
- **DataSet** 개체입니다.

개체 직렬화 또는 역직렬화에 대한 자세한 내용은 방법: 개체 직렬화 및 방법: 개체 역직렬화를 참조하세요.

XML Serialization 사용의 이점

XmlSerializer 클래스는 개체를 XML로 직렬화할 때 완전하고 유연한 제어를 제공합니다. XML 웹 서비스를 만드는 경우 직렬화를 제어하는 특성을 클래스 및 멤버에 적용하여 XML 출력이 특정 스키마를 준수하도록 할 수 있습니다.

예를 들어 **XmlSerializer** 를 사용하면 다음을 수행할 수 있습니다.

- 필드 또는 속성을 특성 또는 요소로 인코딩할지 여부를 지정합니다.
- 사용할 XML 네임스페이스를 지정합니다.
- 필드 또는 속성 이름이 부적절한 경우 요소 또는 특성의 이름을 지정합니다.

XML serialization의 또 다른 이점은 생성된 XML 스트림이 지정된 스키마를 준수하는 한 개발하는 애플리케이션에 제약 조건이 없다는 것입니다. 책을 설명하는 데 사용되는 스키마를 상상해 보세요. 제목, 작성자, 게시자 및 ISBN 번호 요소가 있습니다. XML 데이터를 원하는 방식으로 처리하는 애플리케이션(예: 책 순서 또는 책 인벤토리)을 개발할 수 있습니다. 두 경우 모두 XML 스트림이 지정된 XSD(XML 스키마 정의 언어) 스키마를 준수하는 것이 유일한 요구 사항입니다.

XML Serialization 고려 사항

XmlSerializer 클래스를 사용하는 경우 다음 사항을 고려해야 합니다.

- Sgen.exe 도구는 최적의 성능을 위해 직렬화 어셈블리를 생성하도록 명시적으로 설계되었습니다.
- 직렬화된 데이터에는 데이터 자체와 클래스의 구조만 포함됩니다. 형식 ID 및 어셈블리 정보는 포함되지 않습니다.
- public 속성 및 필드만 직렬화할 수 있습니다. 속성에는 public 접근자(get 및 set 메서드)가 있어야 합니다. 공용이 아닌 데이터를 직렬화해야 하는 경우 XML serialization 대신 클래스를 DataContractSerializer 사용합니다.
- **XmlSerializer**에서 serialize할 매개 변수가 없는 생성자가 클래스에 있어야 합니다.
- 메서드를 serialize할 수 없습니다.
- **XmlSerializer** 는 다음과 같이 특정 요구 사항을 충족하는 경우 **IEnumerable** 또는 **ICollection** 을 다르게 구현하는 클래스를 처리할 수 있습니다.

  IEnumerable을 구현하는 클래스는 단일 매개 변수를 사용하는 public **Add** 메서드를 구현해야 합니다. **Add** 메서드의 매개 변수는 **GetEnumerator** 메서드에서 반환된 **IEnumerator.Current** 속성에서 반환된 형식과 일치해야 합니다(다형).

  IEnumerable(예: **CollectionBase**) 외에도 **ICollection**을 구현하는 클래스에는 정수를 사용하는 public **Item** 인덱싱된 속성(C#의 인덱서)이 있어야 하며 형식 **정**수의 public **Count 속성이** 있어야 합니다. **Add** 메서드에 전달된 매개 변수는 **Item** 속성 또는 해당 형식의 베이스 중 하나에서 반환된 형식과 동일해야 합니다.

  ICollection을 구현하는 클래스의 경우 직렬화할 값은 **GetEnumerator**를 호출하는 대신 인덱싱된 **Item** 속성에서 검색됩니다. 또한 다른 컬렉션 클래스(**ICollection**을 구현하는 컬렉션 클래스)를 반환하는 공용 필드를 제외하고 공용 필드와 속성은 직렬화되지 않습니다. 예제는 XML Serialization 예제를 참조하세요.

XSD 데이터 형식 매핑

XML 스키마 2부: 데이터 형식 이라는 W3C 문서는 XSD(XML 스키마 정의 언어) 스키마에서 허용되는 간단한 데이터 형식을 지정합니다. 이러한 많은 데이터 형식(예: **int** 및 **decimal**)의 경우 .NET에 해당 데이터 형식이 있습니다. 그러나 일부 XML 데이터 형식에는 해당 .NET 데이터 형식(예: **NMTOKEN** 데이터 형식)이 없습니다. 이러한 경우 XML 스키마 정의 도구(XML 스키마 정의 도구(Xsd.exe))를 사용하여 스키마에서 클래스를 생성하는 경우 적절한 특성이 문자열 형식의 멤버에 적용되고 해당 **DataType** 속성이 XML 데이터 형식 이름으로 설정됩니다. 예를 들어 스키마

에 XML 데이터 형식 **NMTOKEN**이 있는 "MyToken"이라는 요소가 포함된 경우 생성된 클래스는 다음 예제와 같이 멤버를 포함할 수 있습니다.

C#

```
[XmlElement(DataType = "NMTOKEN")]
public string MyToken;
```

마찬가지로 특정 XSD(XML 스키마)를 준수해야 하는 클래스를 만드는 경우 적절한 특성을 적용하고 **해당 DataType** 속성을 원하는 XML 데이터 형식 이름으로 설정해야 합니다.

형식 매핑의 전체 목록은 다음 특성 클래스에 대한 **DataType** 속성을 참조하세요.

- SoapAttributeAttribute
- SoapElementAttribute
- XmlArrayItemAttribute
- XmlAttributeAttribute
- XmlElementAttribute
- XmlRootAttribute

참고하십시오

- XmlSerializer
- DataContractSerializer
- FileStream
- XML 및 SOAP Serialization
- 이진 직렬화
- 직렬화
- XmlSerializer
- XML Serialization의 예
- 방법: 개체 직렬화
- 방법: 개체 역직렬화

XML Serialization의 예

2025. 06. 17.

XML serialization은 단순에서 복합 형식으로 둘 이상의 형태를 취할 수 있습니다. 예를 들어 XML Serialization 소개에 표시된 것처럼 단순히 공용 필드 및 속성으로 구성된 클래스 를 serialize할 수 있습니다. 다음 코드 예제에서는 XML serialization을 사용하여 특정 XSD(XML 스키마) 문서를 준수하는 XML 스트림을 생성하는 방법을 포함하여 다양한 고급 시나리오를 다룹니다.

데이터 세트 직렬화

공용 클래스의 인스턴스를 직렬화하는 것 외에도 다음 코드 예제와 같이 인스턴스를 DataSet직렬화할 수도 있습니다.

C#

```
private void SerializeDataSet(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(DataSet));

    // Creates a DataSet; adds a table, column, and ten rows.
    DataSet ds = new DataSet("myDataSet");
    DataTable t = new DataTable("table1");
    DataColumn c = new DataColumn("thing");
    t.Columns.Add(c);
    ds.Tables.Add(t);
    DataRow r;

    for (int i = 0; i < 10; i++) {
        r = t.NewRow();
        r[0] = "Thing " + i;
        t.Rows.Add(r);
    }

    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, ds);
    writer.Close();
}
```

XmlElement 및 XmlNode 직렬화하기

다음 코드 예제에서는 XmlElement 클래스 또는 XmlNode 클래스의 인스턴스를 직렬화할 수 있습니다.

C#

```
private void SerializeElement(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(XmlElement));
    XmlElement myElement = new XmlDocument().CreateElement("MyElement", "ns");
    myElement.InnerText = "Hello World";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myElement);
    writer.Close();
}

private void SerializeNode(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(XmlNode));
    XmlNode myNode = new XmlDocument().
    CreateNode(XmlNodeType.Element, "MyNode", "ns");
    myNode.InnerText = "Hello Node";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myNode);
    writer.Close();
}
```

복합 개체를 반환하는 필드가 포함된 클래스 직렬화

속성 또는 필드가 복합 개체(예: 배열 또는 클래스 인스턴스) XmlSerializer 를 반환하는 경우 기본 XML 문서 내에 중첩된 요소로 변환합니다. 예를 들어 다음 코드 예제의 첫 번째 클래스는 두 번째 클래스의 인스턴스를 반환합니다.

C#

```
public class PurchaseOrder
{
    public Address MyAddress;
}

public record Address
{
    public string FirstName;
}
```

직렬화된 XML 출력은 다음과 같습니다.

XML

```
<PurchaseOrder>
    <MyAddress>
        <FirstName>George</FirstName>
```

```
    </MyAddress>
</PurchaseOrder>
```

개체 배열 직렬화

다음 코드 예제와 같이 개체 배열을 반환하는 필드를 직렬화할 수도 있습니다.

C#

```
public class PurchaseOrder
{
    public Item [] ItemsOrders;
}

public class Item
{
    public string ItemID;
    public decimal ItemPrice;
}
```

두 항목이 정렬되면 serialize된 클래스 인스턴스는 다음 코드와 같이 표시될 수 있습니다.

XML

```
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <ItemsOrders>
        <Item>
            <ItemID>aaa111</ItemID>
            <ItemPrice>34.22</ItemPrice>
        </Item>
        <Item>
            <ItemID>bbb222</ItemID>
            <ItemPrice>2.89</ItemPrice>
        </Item>
    </ItemsOrders>
</PurchaseOrder>
```

ICollection 인터페이스를 구현하는 클래스 직렬화

인터페이스 ICollection를 구현하고 XmlSerializer를 사용하여 이러한 클래스의 인스턴스를 직렬화함으로써 고유한 컬렉션 클래스를 만들 수 있습니다.

> ⓘ **참고**

클래스가 ICollection 인터페이스를 구현하는 경우 클래스에 포함된 컬렉션만이 직렬화됩니다. 클래스에 추가된 모든 공용 속성 또는 필드는 serialize되지 않습니다. serialize하려면 클래스에 **Add** 메서드와 **Item** 속성(C# 인덱서)이 포함되어야 합니다.

C#

```
using System;
using System.Collections;
using System.IO;
using System.Xml.Serialization;

public class Test
{
    static void Main()
    {
        Test t = new Test();
        t.SerializeCollection("coll.xml");
    }

    private void SerializeCollection(string filename)
    {
        Employees Emps = new Employees();
        // Note that only the collection is serialized -- not the
        // CollectionName or any other public property of the class.
        Emps.CollectionName = "Employees";
        Employee John100 = new Employee("John", "100xxx");
        Emps.Add(John100);
        XmlSerializer x = new XmlSerializer(typeof(Employees));
        TextWriter writer = new StreamWriter(filename);
        x.Serialize(writer, Emps);
    }
}

public class Employees : ICollection
{
    public string CollectionName;
    private ArrayList empArray = new ArrayList();

    public Employee this[int index] => (Employee) empArray[index];

    public void CopyTo(Array a, int index)
    {
        empArray.CopyTo(a, index);
    }

    public int Count => empArray.Count;

    public object SyncRoot => this;

    public bool IsSynchronized => false;

    public IEnumerator GetEnumerator() => empArray.GetEnumerator();
```

```
    public void Add(Employee newEmployee)
    {
        empArray.Add(newEmployee);
    }
}

public class Employee
{
    public string EmpName;
    public string EmpID;

    public Employee() {}

    public Employee(string empName, string empID)
    {
        EmpName = empName;
        EmpID = empID;
    }
}
```

구매 주문 예제

다음 예제 코드를 잘라내어 텍스트 파일에 붙여넣고 .cs 또는 .vb 파일 이름 확장명을 사용하여 이름을 바꿀 수 있습니다. C# 또는 Visual Basic 컴파일러를 사용하여 파일을 컴파일합니다. 그런 다음 실행 파일의 이름을 사용하여 실행합니다.

이 예제에서는 간단한 시나리오를 사용하여 개체 인스턴스를 만들고 메서드를 사용하여 Serialize 파일 스트림으로 serialize하는 방법을 보여 줍니다. XML 스트림은 파일에 저장됩니다. 그런 다음 동일한 파일을 읽고 Deserialize 메서드를 사용하여 원래 개체의 복사본으로 다시 재구성합니다.

이 예제에서 명명 `PurchaseOrder` 된 클래스는 직렬화된 다음 역직렬화됩니다. `Address`라는 공용 필드를 `ShipTo`로 설정해야 하기 때문에 `Address`라는 두 번째 클래스도 포함됩니다. 마찬가지로 `OrderedItem` 클래스가 포함된 이유는 `OrderedItem` 필드에 `OrderedItems` 객체 배열을 설정해야 하기 때문입니다. 마지막으로 명명된 `Test` 클래스에는 클래스를 직렬화하고 역직렬화하는 코드가 포함됩니다.

메서드는 `CreatePO` , `PurchaseOrder`및 `Address` 클래스 개체를 만들고 `OrderedItem`공용 필드 값을 설정합니다. 또한 메서드는 직렬화 및 역직렬화하는 데 사용되는 클래스의 인스턴스 XmlSerializer 를 생성합니다 `PurchaseOrder`.

> ⓘ **참고**

코드는 생성자에 serialize할 클래스의 형식을 전달합니다. 이 코드는 XML 스트림을 XML 문서에 작성하기 위해 사용되는 `FileStream`를 생성합니다.

`ReadPo` 메서드는 좀 더 간단합니다. 개체를 만들어 역직렬화하고 해당 값을 읽습니다. 먼저 `CreatePo` 메서드를 사용할 때와 마찬가지로 역직렬화할 클래스의 형식을 생성자에 전달하여 XmlSerializer을 생성해야 합니다. FileStream 또한 XML 문서를 읽는 데 필요합니다. 개체를 역직렬화하려면 인수로 Deserialize 메서드를 FileStream 호출합니다. 역직렬화된 개체는 형식 `PurchaseOrder`의 개체 변수로 캐스팅되어야 합니다. 그런 다음, 이 코드는 역직렬화된 `PurchaseOrder`값의 값을 읽습니다.

> ⓘ **참고**
>
> 생성된 *PO.xml* 파일을 읽고 실제 XML 출력을 볼 수 있습니다.

C#

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

// The XmlRoot attribute allows you to set an alternate name
// (PurchaseOrder) for the XML element and its namespace. By
// default, the XmlSerializer uses the class name. The attribute
// also allows you to set the XML namespace for the element. Lastly,
// the attribute sets the IsNullable property, which specifies whether
// the xsi:null attribute appears if the class instance is set to
// a null reference.
[XmlRoot("PurchaseOrder", Namespace="http://www.cpandl.com",
IsNullable = false)]
public class PurchaseOrder
{
    public Address ShipTo;
    public string OrderDate;
    // The XmlArray attribute changes the XML element name
    // from the default of "OrderedItems" to "Items".
    [XmlArray("Items")]
    public OrderedItem[] OrderedItems;
    public decimal SubTotal;
    public decimal ShipCost;
    public decimal TotalCost;
}

public class Address
{
    // The XmlAttribute attribute instructs the XmlSerializer to serialize the
    // Name field as an XML attribute instead of an XML element (XML element is
    // the default behavior).
```

```
    [XmlAttribute]
    public string Name;
    public string Line1;

    // Setting the IsNullable property to false instructs the
    // XmlSerializer that the XML attribute will not appear if
    // the City field is set to a null reference.
    [XmlElement(IsNullable = false)]
    public string City;
    public string State;
    public string Zip;
}

public class OrderedItem
{
    public string ItemName;
    public string Description;
    public decimal UnitPrice;
    public int Quantity;
    public decimal LineTotal;

    // Calculate is a custom method that calculates the price per item
    // and stores the value in a field.
    public void Calculate()
    {
        LineTotal = UnitPrice * Quantity;
    }
}

public class Test
{
    public static void Main()
    {
        // Read and write purchase orders.
        Test t = new Test();
        t.CreatePO("po.xml");
        t.ReadPO("po.xml");
    }

    private void CreatePO(string filename)
    {
        // Creates an instance of the XmlSerializer class;
        // specifies the type of object to serialize.
        XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
        TextWriter writer = new StreamWriter(filename);
        PurchaseOrder po =new PurchaseOrder();

        // Creates an address to ship and bill to.
        Address billAddress = new Address();
        billAddress.Name = "Teresa Atkinson";
        billAddress.Line1 = "1 Main St.";
        billAddress.City = "AnyTown";
        billAddress.State = "WA";
        billAddress.Zip = "00000";
        // Sets ShipTo and BillTo to the same addressee.
```

```
        po.ShipTo = billAddress;
        po.OrderDate = System.DateTime.Now.ToLongDateString();

        // Creates an OrderedItem.
        OrderedItem i1 = new OrderedItem();
        i1.ItemName = "Widget S";
        i1.Description = "Small widget";
        i1.UnitPrice = (decimal) 5.23;
        i1.Quantity = 3;
        i1.Calculate();

        // Inserts the item into the array.
        OrderedItem [] items = {i1};
        po.OrderedItems = items;
        // Calculate the total cost.
        decimal subTotal = new decimal();
        foreach(OrderedItem oi in items)
        {
            subTotal += oi.LineTotal;
        }
        po.SubTotal = subTotal;
        po.ShipCost = (decimal) 12.51;
        po.TotalCost = po.SubTotal + po.ShipCost;
        // Serializes the purchase order, and closes the TextWriter.
        serializer.Serialize(writer, po);
        writer.Close();
    }

    protected void ReadPO(string filename)
    {
        // Creates an instance of the XmlSerializer class;
        // specifies the type of object to be deserialized.
        XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
        // If the XML document has been altered with unknown
        // nodes or attributes, handles them with the
        // UnknownNode and UnknownAttribute events.
        serializer.UnknownNode+= new
        XmlNodeEventHandler(serializer_UnknownNode);
        serializer.UnknownAttribute+= new
        XmlAttributeEventHandler(serializer_UnknownAttribute);

        // A FileStream is needed to read the XML document.
        FileStream fs = new FileStream(filename, FileMode.Open);
        // Declares an object variable of the type to be deserialized.
        PurchaseOrder po;
        // Uses the Deserialize method to restore the object's state
        // with data from the XML document. */
        po = (PurchaseOrder) serializer.Deserialize(fs);
        // Reads the order date.
        Console.WriteLine ("OrderDate: " + po.OrderDate);

        // Reads the shipping address.
        Address shipTo = po.ShipTo;
        ReadAddress(shipTo, "Ship To:");
        // Reads the list of ordered items.
```

```
        OrderedItem [] items = po.OrderedItems;
        Console.WriteLine("Items to be shipped:");
        foreach(OrderedItem oi in items)
        {
            Console.WriteLine("\t"+
            oi.ItemName + "\t" +
            oi.Description + "\t" +
            oi.UnitPrice + "\t" +
            oi.Quantity + "\t" +
            oi.LineTotal);
        }
        // Reads the subtotal, shipping cost, and total cost.
        Console.WriteLine(
        "\n\t\t\t\t\t Subtotal\t" + po.SubTotal +
        "\n\t\t\t\t\t Shipping\t" + po.ShipCost +
        "\n\t\t\t\t\t Total\t\t" + po.TotalCost
        );
    }

    protected void ReadAddress(Address a, string label)
    {
        // Reads the fields of the Address.
        Console.WriteLine(label);
        Console.Write("\t"+
        a.Name +"\n\t" +
        a.Line1 +"\n\t" +
        a.City +"\t" +
        a.State +"\n\t" +
        a.Zip +"\n");
    }

    protected void serializer_UnknownNode
    (object sender, XmlNodeEventArgs e)
    {
        Console.WriteLine("Unknown Node:" +   e.Name + "\t" + e.Text);
    }

    protected void serializer_UnknownAttribute
    (object sender, XmlAttributeEventArgs e)
    {
        System.Xml.XmlAttribute attr = e.Attr;
        Console.WriteLine("Unknown attribute " +
        attr.Name + "='" + attr.Value + "'");
    }
}
```

XML 출력은 다음과 같을 수 있습니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.cpandl.com">
```

```
    <ShipTo Name="Teresa Atkinson">
        <Line1>1 Main St.</Line1>
        <City>AnyTown</City>
        <State>WA</State>
        <Zip>00000</Zip>
    </ShipTo>
    <OrderDate>Wednesday, June 27, 2001</OrderDate>
    <Items>
        <OrderedItem>
            <ItemName>Widget S</ItemName>
            <Description>Small widget</Description>
            <UnitPrice>5.23</UnitPrice>
            <Quantity>3</Quantity>
            <LineTotal>15.69</LineTotal>
        </OrderedItem>
    </Items>
    <SubTotal>15.69</SubTotal>
    <ShipCost>12.51</ShipCost>
    <TotalCost>28.2</TotalCost>
</PurchaseOrder>
```

참고하십시오

- XML serialization 소개
- 특성을 사용하여 XML serialization 제어
- XML serialization을 제어하는 특성
- XmlSerializer 클래스
- 방법: 개체를 직렬화하는 방법
- 개체를 역직렬화하는 방법

XML 스키마 정의 도구 및 XML 직렬화

2025. 06. 17.

XML 스키마 정의 도구(XML 스키마 정의 도구(Xsd.exe)는 .NET Framework 도구와 함께 Windows® SDK(소프트웨어 개발 키트)의 일부로 설치됩니다. 이 도구는 주로 다음 두 가지 용도로 설계되었습니다.

- 특정 XSD(XML 스키마 정의 언어) 스키마를 준수하는 C# 또는 Visual Basic 클래스 파일을 생성하려면 이 도구는 XML 스키마를 인수로 사용하여 스키마를 직렬화 XmlSerializer할 때 스키마를 따르는 여러 클래스가 포함된 파일을 출력합니다. 도구를 사용하여 특정 스키마를 준수하는 클래스를 생성하는 방법에 대한 자세한 내용은 방법: XML 스키마 정의 도구를 사용하여 클래스 및 XML 스키마 문서를 생성합니다.
- .dll 파일 또는 .exe 파일에서 XML 스키마 문서를 생성하려면 만든 파일 집합 또는 특성으로 수정된 파일 집합의 스키마를 보려면 DLL 또는 EXE를 도구에 인수로 전달하여 XML 스키마를 생성합니다. 도구를 사용하여 클래스 집합에서 XML 스키마 문서를 생성하는 방법에 대한 자세한 내용은 방법: XML 스키마 정의 도구를 사용하여 클래스 및 XML 스키마 문서를 생성합니다.

도구 사용에 대한 자세한 내용은 XML 스키마 정의 도구(Xsd.exe)를 참조하세요.

참고하십시오

- DataSet
- XML Serialization 소개
- XML 스키마 정의 도구(Xsd.exe)
- XmlSerializer
- 방법: 개체 직렬화
- 방법: 개체 역직렬화
- 방법: XML 스키마 정의 도구를 사용하여 클래스 및 XML 스키마 문서 생성
- XML 스키마 바인딩 지원

특성을 사용하여 XML serialization 제어

특성을 사용하여 개체의 XML serialization을 제어하거나 동일한 클래스 집합에서 대체 XML 스트림을 만들 수 있습니다. 대체 XML 스트림을 만드는 방법에 대한 자세한 내용은 방법: XML 스트림에 대한 대체 요소 이름 지정을 참조하세요.

> ⓘ **참고 항목**
>
> 생성된 XML이 SOAP(Simple Object Access Protocol) 1.1 이라는 W3C(World Wide Web Consortium) 문서의 섹션 5를 준수해야 **하는 경우 인코딩된 SOAP Serialization을 제어하는 특성**에 나열된 특성을 사용합니다.

기본적으로 XML 요소 이름은 클래스 또는 멤버 이름으로 결정됩니다. 명명 `Book`된 클래스에서 **ISBN** 이라는 필드는 다음 예제와 같이 XML 요소 태그 `<ISBN>`를 생성합니다.

C#

```
public class Book
{
    public string ISBN;
}
// When an instance of the Book class is serialized, it might
// produce this XML:
// <ISBN>1234567890</ISBN>.
```

요소에 새 이름을 지정하려는 경우 기본 동작을 변경할 수 있습니다. 다음 코드는 특성을 설정하여 XmlElementAttribute의 ElementName 속성을 변경함으로써 이 기능을 활성화하는 방법을 보여 줍니다.

C#

```
public class TaxRates {
    [XmlElement(ElementName = "TaxRate")]
    public decimal ReturnTaxRate;
}
```

특성에 대한 자세한 내용은 특성을 참조 하세요. XML serialization을 제어하는 특성 목록은 XML Serialization을 제어하는 특성을 참조하세요.

배열 직렬화 제어

XmlArrayAttribute 및 XmlArrayItemAttribute 특성은 배열의 직렬화를 제어합니다. 이러한 특성을 사용하여 W3C 문서에 정의된 XML 스키마 파트 2: 데이터 형식에 정의된 대로 요소 이름,

네임스페이스 및 XSD(XML 스키마) 데이터 형식을 제어할 수 있습니다. 배열에 포함할 수 있는 형식을 지정할 수도 있습니다.

XmlArrayAttribute 배열이 serialize될 때 발생하는 바깥쪽 XML 요소의 속성을 결정합니다. 예를 들어 기본적으로 아래 배열을 serialize하면 `Employees`라는 이름의 XML 요소가 생성됩니다. 요소에는 `Employees` 배열 형식 `Employee`의 이름을 따서 명명된 일련의 요소가 포함됩니다.

C#

```
public class Group {
    public Employee[] Employees;
}
public class Employee {
    public string Name;
}
```

직렬화된 인스턴스는 다음 코드와 유사할 수 있습니다.

XML

```
<Group>
<Employees>
    <Employee>
        <Name>Haley</Name>
    </Employee>
</Employees>
</Group>
```

를 적용하면 XmlArrayAttribute다음과 같이 XML 요소의 이름을 변경할 수 있습니다.

C#

```
public class Group {
    [XmlArray("TeamMembers")]
    public Employee[] Employees;
}
```

결과 XML은 다음 코드와 유사할 수 있습니다.

XML

```
<Group>
<TeamMembers>
    <Employee>
        <Name>Haley</Name>
    </Employee>
</TeamMembers>
</Group>
```

XmlArrayItemAttribute는 반면에 배열에 포함된 항목을 직렬화하는 방법을 제어합니다.

> ⓘ **참고 항목**
>
> 배열을 반환하는 필드에 특성이 적용됩니다.

C#

```
public class Group {
    [XmlArrayItem("MemberName")]
    public Employee[] Employees;
}
```

결과 XML은 다음 코드와 유사할 수 있습니다.

XML

```
<Group>
<Employees>
    <MemberName>Haley</MemberName>
</Employees>
</Group>
```

파생 클래스 직렬화

파생 클래스의 XmlArrayItemAttribute serialization을 허용하는 것이 또 다른 용도입니다. 예를 들어, `Employee`에서 파생된 다른 `Manager` 클래스를 이전 예제에 추가할 수 있습니다. 적용 XmlArrayItemAttribute하지 않으면 파생 클래스 형식이 인식되지 않으므로 런타임에 코드가 실패합니다. 이 결과를 해결하려면 허용되는 각 형식(기본 및 파생)에 대한 속성을 설정할 Type 때마다 특성을 두 번 적용합니다.

C#

```
public class Group {
    [XmlArrayItem(Type = typeof(Employee)),
    XmlArrayItem(Type = typeof(Manager))]
    public Employee[] Employees;
}
public class Employee {
    public string Name;
}
public class Manager:Employee {
    public int Level;
}
```

직렬화된 인스턴스는 다음 코드와 유사할 수 있습니다.

XML

```
<Group>
<Employees>
    <Employee>
        <Name>Haley</Name>
    </Employee>
    <Employee xsi:type = "Manager">
        <Name>Ann</Name>
        <Level>3</Level>
    </Employee>
</Employees>
</Group>
```

배열을 요소 시퀀스로 직렬화

XmlElementAttribute를 배열을 반환하는 필드에 적용함으로써 배열을 XML 요소의 플랫 시퀀스로 직렬화할 수도 있습니다.

C#

```
public class Group {
    [XmlElement]
    public Employee[] Employees;
}
```

직렬화된 인스턴스는 다음 코드와 유사할 수 있습니다.

XML

```
<Group>
<Employees>
    <Name>Haley</Name>
</Employees>
<Employees>
    <Name>Noriko</Name>
</Employees>
<Employees>
    <Name>Marco</Name>
</Employees>
</Group>
```

두 XML 스트림을 구분하는 또 다른 방법은 XML 스키마 정의 도구를 사용하여 컴파일된 코드에서 XSD(XML 스키마) 문서 파일을 생성하는 것입니다. 도구 사용에 대한 자세한 내용은 XML 스키마 정의 도구 및 XML Serialization을 참조하세요. 필드에 특성이 적용되지 않으면 스키마는 다음과 같은 방식으로 요소를 설명합니다.

XML

```
<xs:element minOccurs="0" maxOccurs ="1" name="Employees" type="ArrayOfEmployee" />
```

XmlElementAttribute 필드에 적용되는 경우 결과 스키마는 다음과 같이 요소를 설명합니다.

XML

```
<xs:element minOccurs="0" maxOccurs="unbounded" name="Employees" type="Employee" />
```

ArrayList 직렬화

클래스에는 ArrayList 다양한 개체의 컬렉션이 포함될 수 있습니다. 따라서 ArrayList를 배열처럼 사용할 수 있습니다. 그러나 형식화된 개체의 배열을 반환하는 필드를 만드는 대신 단일 ArrayList개체를 반환하는 필드를 만들 수 있습니다. 배열과 마찬가지로 ArrayList 에 포함된 개체의 형식을 XmlSerializer에 알려야 합니다. 이렇게 하려면 다음 예제와 XmlElementAttribute 같이 필드에 여러 인스턴스를 할당합니다.

C#

```
public class Group {
    [XmlElement(Type = typeof(Employee)),
    XmlElement(Type = typeof(Manager))]
    public ArrayList Info;
}
```

XmlRootAttribute 및 XmlTypeAttribute를 사용하여 클래스의 Serialization 제어

클래스에만 XmlRootAttributeXmlTypeAttribute두 개의 특성을 적용할 수 있습니다. 이러한 특성은 비슷합니다. 직렬화할 때 XML 문서의 여는 XmlRootAttribute 요소와 닫는 요소, 즉 루트 요소를 나타내는 클래스인 한 클래스에만 적용할 수 있습니다. XmlTypeAttribute반면에 루트 클래스를 비롯한 모든 클래스에 적용할 수 있습니다.

예를 들어 이전 예제 `Group` 에서 클래스는 루트 클래스이며 모든 공용 필드와 속성은 XML 문서에 있는 XML 요소가 됩니다. 따라서 루트 클래스는 하나만 가질 수 있습니다. XmlRootAttribute 을 적용함으로써 XmlSerializer에 의해 생성된 XML 스트림을 제어할 수 있습니다. 예를 들어 요소 이름 및 네임스페이스를 변경할 수 있습니다.

생성된 XmlTypeAttribute XML의 스키마를 제어할 수 있습니다. 이 기능은 XML 웹 서비스를 통해 스키마를 게시해야 하는 경우에 유용합니다. 다음 예제에서는 동일한 클래스에 XmlTypeAttribute와 XmlRootAttribute 모두 적용합니다.

C#

```
[XmlRoot("NewGroupName")]
[XmlType("NewTypeName")]
public class Group {
    public Employee[] Employees;
}
```

이 클래스가 컴파일되고 XML 스키마 정의 도구를 사용하여 스키마를 생성하는 경우 다음 XML에서 다음을 설명합니다 `Group`.

XML

```
<xs:element name="NewGroupName" type="NewTypeName" />
```

반면, 클래스의 인스턴스를 직렬화하는 경우 XML 문서에서만 `NewGroupName`을 찾을 수 있습니다.

XML

```
<NewGroupName>
    . . .
</NewGroupName>
```

XmlIgnoreAttribute를 사용하여 Serialization 방지

공유 속성 또는 필드를 직렬화할 필요가 없는 상황에 직면할 수 있습니다. 예를 들어 필드 또는 속성을 사용하여 메타데이터를 포함할 수 있습니다. 이러한 경우 필드 또는 속성에 XmlIgnoreAttribute 적용하고 XmlSerializer 건너뜁니다.

참고하십시오

- XML Serialization을 제어하는 특성
- 인코딩된 SOAP Serialization을 제어하는 특성
- XML Serialization 소개
- XML Serialization의 예
- 방법: XML 스트림의 대체 요소 이름 지정
- 개체를 직렬화하는 방법
- 방법: 개체 디시리얼라이즈

Last updated on 2025. 12. 05.

XML Serialization을 제어하는 특성

2025. 06. 17.

클래스 및 클래스 멤버에 다음 표의 특성을 적용하여 클래스 인스턴스를 직렬화하거나 역직렬화하는 방식을 XmlSerializer 제어할 수 있습니다. 이러한 특성이 XML serialization을 제어하는 방법을 이해하려면 특성을 사용하여 XML Serialization 제어를 참조하세요.

이러한 특성을 사용하여 XML 웹 서비스에서 생성된 리터럴 스타일 SOAP 메시지를 제어할 수도 있습니다. 이러한 특성을 XML 웹 서비스 메서드에 적용하는 방법에 대한 자세한 내용은 XML Web Services를 사용한 XML Serialization을 참조하세요.

특성에 대한 자세한 내용은 특성을 참조 하세요.

테이블 확장

특성	적용 대상	명시
XmlAnyAttributeAttribute	개체 배열 XmlAttribute 을 반환하는 공용 필드, 속성, 매개 변수 또는 반환 값입니다.	역직렬화할 때 배열은 스키마에 XmlAttribute 알 수 없는 모든 XML 특성을 나타내는 개체로 채워집니다.
XmlAnyElementAttribute	개체 배열 XmlElement 을 반환하는 공용 필드, 속성, 매개 변수 또는 반환 값입니다.	역직렬화할 때 배열은 스키마에 XmlElement 알려지지 않은 모든 XML 요소를 나타내는 개체로 채워집니다.
XmlArrayAttribute	복합 개체의 배열을 반환하는 공용 필드, 속성, 매개 변수 또는 반환 값입니다.	배열의 멤버는 XML 배열의 멤버로 생성됩니다.
XmlArrayItemAttribute	복합 개체의 배열을 반환하는 공용 필드, 속성, 매개 변수 또는 반환 값입니다.	배열에 삽입할 수 있는 파생 형식입니다. 일반적으로 XmlArrayAttribute와 함께 적용됩니다.
XmlAttributeAttribute	공용 필드, 속성, 매개 변수 또는 반환 값입니다.	멤버는 XML 특성으로 직렬화됩니다.
XmlChoiceIdentifierAttribute	공용 필드, 속성, 매개 변수 또는 반환 값입니다.	열거형을 사용하여 멤버를 더 명확하게 구분할 수 있습니다.
XmlElementAttribute	공용 필드, 속성, 매개 변수 또는 반환 값입니다.	필드 또는 속성은 XML 요소로 serialize 됩니다.
XmlEnumAttribute	열거형 식별자인 공용 필드입니다.	열거형 멤버의 요소 이름입니다.
XmlIgnoreAttribute	공용 속성 및 필드입니다.	포함하는 클래스가 serialize될 때 속성 또는 필드를 무시해야 합니다.

특성	적용 대상	명시
XmlIncludeAttribute	공용 파생 클래스 선언 및 WSDL(Web Services Description Language) 문서에 대한 공용 메서드의 값을 반환합니다.	클래스는 스키마를 생성할 때 포함되어야 합니다(serialize할 때 인식할 수 있도록).
XmlRootAttribute	공개 클래스 선언	XML 루트 요소로 특성 대상의 XML 직렬화를 제어합니다. 특성을 사용하여 네임스페이스 및 요소 이름을 추가로 지정합니다.
XmlTextAttribute	공용 속성 및 필드입니다.	속성 또는 필드를 XML 텍스트로 serialize해야 합니다.
XmlTypeAttribute	공개 클래스 선언	XML 형식의 이름 및 네임스페이스입니다.
ObsoleteAttribute	공용 속성 및 필드입니다.	포함하는 클래스가 serialize되면 속성 또는 필드가 무시됩니다.

네임스페이스에 있는 System.Xml.Serialization 이러한 특성 외에도 필드에 특성을 적용 DefaultValueAttribute 할 수도 있습니다. **DefaultValueAttribute**는 값이 지정되지 않은 경우 멤버에 자동으로 할당되는 값을 설정합니다.

인코딩된 SOAP XML serialization을 제어하려면 인코딩된 SOAP Serialization을 제어하는 특성을 참조하세요.

참고하십시오

- XML 및 SOAP Serialization
- XmlSerializer
- 특성을 사용하여 XML Serialization 제어
- 방법: XML 스트림의 대체 요소 이름 지정
- 방법: 개체 직렬화
- 방법: 개체 역직렬화

XML 웹 서비스를 통한 XML 직렬화

2025. 06. 17.

XML serialization은 XmlSerializer 클래스에서 수행하는 XML 웹 서비스 아키텍처의 기본 전송 메커니즘입니다. XML 웹 서비스에서 생성된 XML을 제어하려면 XML Serialization을 제어하는 특성 과 인코딩된 SOAP Serialization을 제어하는 특성 모두에 나열된 특성을 적용하여 XML 웹 서비스(.asmx)를 만드는 데 사용되는 파일의 값, 매개 변수 및 필드를 반환합니다. XML 웹 서비스를 만드는 방법에 대한 자세한 내용은 ASP.NET 사용하여 XML Web Services를 참조하세요.

리터럴 및 인코딩된 스타일

XML 웹 서비스에서 생성된 XML은 SOAP 메시지 서식 사용자 지정에 설명된 대로 리터럴 또는 인코딩된 두 가지 방법 중 하나로 서식을 지정할 수 있습니다. 따라서 XML serialization을 제어하는 두 가지 특성 집합이 있습니다. XML Serialization을 제어하는 특성에 나열된 특성은 리터럴 스타일 XML을 제어하도록 설계되었습니다. 인코딩된 SOAP Serialization을 제어하는 특성에 나열된 특성은 인코딩된 스타일을 제어합니다. 이러한 특성을 선택적으로 적용하여 애플리케이션을 맞춤화하여 스타일을 반환하거나 둘 다 반환할 수 있습니다. 또한 이러한 특성은 값과 매개 변수를 반환하는 데 적절하게 적용할 수 있습니다.

두 스타일 사용 예제

XML 웹 서비스를 만들 때 메서드에서 두 특성 집합을 모두 사용할 수 있습니다. 다음 코드 예제에서 명명된 `MyService` 클래스에는 두 개의 XML 웹 서비스 메서드 `MyLiteralMethod` 및 `MyEncodedMethod`. 두 메서드 모두 클래스의 인스턴스를 반환하는 동일한 함수를 수행합니다 `Order` . `Order` 클래스에서 XmlTypeAttribute 및 SoapTypeAttribute 속성이 모두 `OrderID` 필드에 적용되고, 각 특성의 `ElementName` 속성이 서로 다른 값으로 설정됩니다.

예제를 실행하려면 코드를 .asmx 확장자를 사용하여 파일에 붙여넣고 IIS(인터넷 정보 서비스)에서 관리하는 가상 디렉터리에 파일을 배치합니다. 웹 브라우저에서 컴퓨터, 가상 디렉터리 및 파일의 이름을 입력합니다.

C#

```
<%@ WebService Language="C#" Class="MyService" %>
using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;
public class Order {
    // Both types of attributes can be applied. Depending on which type
    // the method used, either one will affect the call.
    [SoapElement(ElementName = "EncodedOrderID")]
    [XmlElement(ElementName = "LiteralOrderID")]
```

```
    public String OrderID;
}
public class MyService {
    [WebMethod][SoapDocumentMethod]
    public Order MyLiteralMethod(){
        Order myOrder = new Order();
        return myOrder;
    }
    [WebMethod][SoapRpcMethod]
    public Order MyEncodedMethod(){
        Order myOrder = new Order();
        return myOrder;
    }
}
```

다음 코드 예제에서는 .를 호출합니다 `MyLiteralMethod`. 요소 이름이 "LiteralOrderID"로 변경됩니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethodResponse xmlns="http://tempuri.org/">
            <MyLiteralMethodResult>
                <LiteralOrderID>string</LiteralOrderID>
            </MyLiteralMethodResult>
        </MyLiteralMethodResponse>
    </soap:Body>
</soap:Envelope>
```

다음 코드 예제에서는 .를 호출합니다 `MyEncodedMethod`. 요소 이름은 "EncodedOrderID"입니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://tempuri.org/" xmlns:types="http://tempuri.org/encodedTypes"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <tns:MyEncodedMethodResponse>
            <MyEncodedMethodResult href="#id1" />
        </tns:MyEncodedMethodResponse>
        <types:Order id="id1" xsi:type="types:Order">
            <EncodedOrderID xsi:type="xsd:string">string</EncodedOrderID>
        </types:Order>
```

```
    </soap:Body>
</soap:Envelope>
```

반환 값에 특성 적용

특성을 적용하여 값을 반환하여 네임스페이스, 요소 이름 등을 제어할 수도 있습니다. 다음 코드 예제에서는 `XmlElementAttribute` 메서드의 반환 값에 `MyLiteralMethod` 특성을 적용합니다. 이렇게 하면 네임스페이스 및 요소 이름을 제어할 수 있습니다.

C#

```
[return: XmlElement(Namespace = "http://www.cohowinery.com",
ElementName = "BookOrder")]
[WebMethod][SoapDocumentMethod]
public Order MyLiteralMethod(){
    Order myOrder = new Order();
    return myOrder;
}
```

호출될 때 코드는 다음과 유사한 XML을 반환합니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethodResponse xmlns="http://tempuri.org/">
            <BookOrder xmlns="http://www.cohowinery.com">
                <LiteralOrderID>string</LiteralOrderID>
            </BookOrder>
        </MyLiteralMethodResponse>
    </soap:Body>
</soap:Envelope>
```

매개 변수에 적용된 특성

매개 변수에 특성을 적용하여 네임스페이스, 요소 이름 등을 지정할 수도 있습니다. 다음 코드 예제에서는 메서드에 매개 변수를 `MyLiteralMethodResponse` 추가하고 매개 변수에 `XmlAttributeAttribute` 특성을 적용합니다. 요소 이름과 네임스페이스는 모두 매개 변수에 대해 설정됩니다.

C#

```
[return: XmlElement(Namespace = "http://www.cohowinery.com",
ElementName = "BookOrder")]
[WebMethod][SoapDocumentMethod]
public Order MyLiteralMethod([XmlElement("MyOrderID",
Namespace="http://www.microsoft.com")] string ID){
    Order myOrder = new Order();
    myOrder.OrderID = ID;
    return myOrder;
}
```

SOAP 요청은 다음과 유사합니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethod xmlns="http://tempuri.org/">
            <MyOrderID xmlns="http://www.microsoft.com">string</MyOrderID>
        </MyLiteralMethod>
    </soap:Body>
</soap:Envelope>
```

클래스에 특성 적용

클래스와 관련된 요소의 네임스페이스를 제어해야 할 경우, 적절하게 `XmlTypeAttribute`, `XmlRootAttribute`, 및 `SoapTypeAttribute`를 적용할 수 있습니다. 다음 코드 예제에서는 세 가지를 모두 클래스에 적용합니다 `Order` .

C#

```
[XmlType("BigBooksService", Namespace = "http://www.cpandl.com")]
[SoapType("SoapBookService")]
[XmlRoot("BookOrderForm")]
public class Order {
    // Both types of attributes can be applied. Depending on which
    // the method used, either one will affect the call.
    [SoapElement(ElementName = "EncodedOrderID")]
    [XmlElement(ElementName = "LiteralOrderID")]
    public String OrderID;
}
```

아래 코드 예제에서 볼 수 있듯이 서비스 설명을 검사할 때 `XmlTypeAttribute`과 `SoapTypeAttribute` 적용 결과를 확인할 수 있습니다.

XML

```
<s:element name="BookOrderForm" type="s0:BigBookService" />
<s:complexType name="BigBookService">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="LiteralOrderID" type="s:string"
/>
  </s:sequence>

  <s:schema targetNamespace="http://tempuri.org/encodedTypes">
    <s:complexType name="SoapBookService">
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="EncodedOrderID"
type="s:string" />
      </s:sequence>
    </s:complexType>
  </s:schema>
</s:complexType>
```

이 효과는 `XmlRootAttribute` 다음과 같이 HTTP GET 및 HTTP POST 결과에서도 확인할 수 있습니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<BookOrderForm xmlns="http://tempuri.org/">
    <LiteralOrderID>string</LiteralOrderID>
</BookOrderForm>
```

참고하십시오

- XML 및 SOAP Serialization
- 인코딩된 SOAP Serialization을 제어하는 특성
- 방법: 개체를 SOAP-Encoded XML 스트림으로 직렬화
- 방법: 인코딩된 SOAP XML Serialization을 재정의하는 방법
- XML Serialization 소개
- 방법: 개체 직렬화
- 방법: 개체 역직렬화

인코딩된 SOAP Serialization을 제어하는 특성

2025. 06. 17.

SOAP(Simple Object Access Protocol) 1.1 이라는 W3C(World Wide Web 컨소시엄) 문서에는 SOAP 매개 변수를 인코딩하는 방법을 설명하는 선택적 섹션(섹션 5)이 포함되어 있습니다. 사양의 섹션 5를 준수하려면 네임스페이스에 있는 특수 특성 집합을 System.Xml.Serialization 사용해야 합니다. 해당 특성을 클래스 및 클래스의 멤버에 적절하게 적용한 다음 XmlSerializer를 사용하여 클래스 또는 클래스의 인스턴스를 직렬화합니다.

다음 표에서는 특성, 적용할 수 있는 위치 및 수행할 작업을 보여줍니다. 이러한 특성을 사용하여 XML serialization을 제어하는 방법에 대한 자세한 내용은 방법: 개체를 SOAP-Encoded XML 스트림으로 직렬화 및 방법: 인코딩된 SOAP XML Serialization 재정의를 참조하세요.

특성에 대한 자세한 내용은 특성을 참조 하세요.

테이블 확장

특성	적용 대상	명시
SoapAttributeAttribute	공용 필드, 속성, 매개 변수 또는 반환 값입니다.	클래스 멤버는 XML 특성으로 serialize됩니다.
SoapElementAttribute	공용 필드, 속성, 매개 변수 또는 반환 값입니다.	클래스는 XML 요소로 serialize됩니다.
SoapEnumAttribute	열거형 식별자인 공용 필드입니다.	열거형 멤버의 요소 이름입니다.
SoapIgnoreAttribute	공용 속성 및 필드입니다.	포함하는 클래스가 serialize될 때 속성 또는 필드를 무시해야 합니다.
SoapIncludeAttribute	WSDL(Web Services Description Language) 문서에 대한 공용 파생 클래스 선언 및 공용 메서드입니다.	스키마를 생성할 때 형식을 포함해야 합니다(serialize할 때 인식할 수 있도록).
SoapTypeAttribute	공개 클래스 선언	클래스는 XML 형식으로 serialize되어야 합니다.

참고하십시오

- XML 및 SOAP Serialization
- 방법: 개체를 SOAP-Encoded XML 스트림으로 직렬화
- 방법: 인코딩된 SOAP XML Serialization을 재정의하는 방법

- 특성
- XmlSerializer
- 방법: 개체 직렬화
- 방법: 개체 역직렬화

방법: 개체 Serialize

아티클 • 2024. 03. 05.

개체를 serialize하려면 먼저 serialize될 개체를 만들고 해당 public 속성과 필드를 설정합니다. 이렇게 하려면 XML 스트림이 저장될 전송 형식을 스트림 또는 파일 중에서 결정합니다. 예를 들어 XML 스트림을 영구적 형태로 저장해야 하는 경우에는 FileStream 개체를 만듭니다.

> ⓘ **참고**
>
> XML serialization에 대한 다른 예제를 보려면 **XML Serialization 예제**를 참조하세요.

개체를 serialize하려면

1. 개체를 만들고 해당 public 필드 및 속성을 설정합니다.
2. 개체의 형식을 사용하여 XmlSerializer를 생성합니다. 자세한 내용은 XmlSerializer 클래스 생성자를 참조하십시오.
3. Serialize 메서드를 호출하여 개체의 public 속성 및 필드의 파일 표현 또는 XML 스트림을 생성합니다. 다음 예제에서는 파일을 만듭니다.

```
C#

MySerializableClass myObject = new MySerializableClass();
// Insert code to set properties and fields of the object.
XmlSerializer mySerializer = new
XmlSerializer(typeof(MySerializableClass));
// To write to a file, create a StreamWriter object.
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
myWriter.Close();
```

참고 항목

- XML serialization 소개
- 방법: 개체 역직렬화

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

XmlSerializer를 사용하여 개체를 역직렬화하는 방법

아티클 • 2024. 03. 05.

개체를 역직렬화할 때는 전송 형식에 따라 스트림을 만들지 파일 개체를 만들지 여부가 결정됩니다. 전송 형식이 결정된 뒤에는 필요에 따라 Serialize 또는 Deserialize 메서드를 호출할 수 있습니다.

개체를 역직렬화하려면

1. 역직렬화할 개체의 형식을 사용하여 XmlSerializer를 생성합니다.
2. 개체의 복제본을 생성할 Deserialize 메서드를 호출합니다. 역직렬화할 때는 개체에서 파일을 역직렬화(스트림에서 역직렬화할 수도 있음)하는 다음 예제와 같이 반환된 개체를 원래의 형식으로 캐스팅해야 합니다.

```
C#
```

```
// Construct an instance of the XmlSerializer with the type
// of object that is being deserialized.
var mySerializer = new XmlSerializer(typeof(MySerializableClass));
// To read the file, create a FileStream.
using var myFileStream = new FileStream("myFileName.xml",
FileMode.Open);
// Call the Deserialize method and cast to the object type.
var myObject =
(MySerializableClass)mySerializer.Deserialize(myFileStream);
```

참고 항목

- XML serialization 소개
- 방법: 개체 직렬화

수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

제품 사용자 의견 제공

방법: XML 스키마 정의 도구를 사용하여 클래스 및 XML 스키마 문서 생성

XML 스키마 정의 도구(Xsd.exe)를 사용하면 클래스를 설명하는 XML 스키마를 생성하거나 XML 스키마로 정의된 클래스를 생성할 수 있습니다. 다음 절차에서는 이러한 작업을 수행하는 방법을 보여 줍니다.

XML 스키마 정의 도구(Xsd.exe)는 일반적으로 다음 경로에서 찾을 수 있습니다.
C:\Program Files (x86)\Microsoft SDKs\Windows\{version}\bin\NETFX {version} Tools

특정 스키마를 따르는 클래스를 생성하려면

1. 명령 프롬프트를 열십시오.

2. XML 스키마를 XML 스키마 정의 도구에 인수로 전달합니다. 그러면 예를 들어 XML 스키마에 정확하게 일치하는 클래스 집합이 만들어 집니다.

   콘솔

   ```
   xsd mySchema.xsd /classes
   ```

   (이 `/classes` 명령의 옵션은 스키마에 대한 `mySchema` 클래스를 생성하는 데 사용됩니다.) 이 도구는 2001년 3월 16일의 World Wide Web 컨소시엄 XML 사양을 참조하는 스키마만 처리할 수 있습니다. 즉, XML 스키마 네임스페이스는 다음 예제와 같이 `"http://www.w3.org/2001/XMLSchema"`합니다.

   XML

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified"
   targetNamespace="" xmlns:xs="http://www.w3.org/2001/XMLSchema" />
   ```

3. 필요에 따라 메서드, 속성 또는 필드로 클래스를 수정합니다. 특성을 사용하여 클래스를 수정하는 방법에 대한 자세한 내용은 특성을 사용하여 XML Serialization 제어 및 인코딩된 SOAP Serialization을 제어하는 특성을 참조하세요.

클래스의 인스턴스가 serialize될 때 생성되는 XML 스트림의 스키마를 검사하는 것이 유용할 때가 있습니다. 예를 들어 스키마를 다른 사람이 사용할 수 있도록 게시하거나 준수를 위해 다른 스키마와 비교할 수 있습니다.

클래스 집합에서 XML 스키마 문서를 생성하려면

1. 클래스를 DLL로 컴파일합니다.

2. 명령 프롬프트를 열십시오.

3. DLL을 Xsd.exe에 인수로 전달합니다. 예를 들면 다음과 같습니다.

   콘솔

   ```
   xsd MyFile.dll
   ```

   스키마가 이름 "schema0.xsd"로 시작하여 기록됩니다.

참고 항목

- DataSet
- XML 스키마 정의 도구 및 XML serialization
- XML serialization 소개
- XML 스키마 정의 도구(Xsd.exe)
- XmlSerializer
- 개체를 직렬화하는 방법
- 방법: 개체 역직렬화

Last updated on 2026. 02. 24.

방법: 파생 클래스의 Serialization 제어

아티클 • 2023. 04. 07.

XmlElementAttribute 특성을 사용하여 XML 요소의 이름을 변경하는 것이 개체 serialization을 사용자 지정하는 유일한 방법은 아닙니다. 기존 클래스에서 파생하고 새 클래스를 serialize하는 방법을 XmlSerializer 인스턴스에 지시하여 XML 스트림을 사용자 지정할 수도 있습니다.

예를 들어 `Book` 클래스의 경우 이 클래스에서 파생하고 몇 개의 속성이 더 있는 `ExpandedBook` 클래스를 만들 수 있습니다. 하지만 직렬화 또는 역직렬화할 때 파생된 형식을 허용하도록 **XmlSerializer**에 지시해야 합니다. 이렇게 하려면 XmlElementAttribute 인스턴스를 만들고 이 인스턴스의 **Type** 속성을 파생 클래스 형식으로 설정합니다. **XmlElementAttribute**를 XmlAttributes 인스턴스에 추가합니다. 그런 다음 **XmlAttributes**를 XmlAttributeOverrides 인스턴스로 추가하고 재정의되는 형식과 파생 클래스를 허용하는 멤버의 이름을 지정합니다. 다음 예제에서 이를 확인할 수 있습니다.

예제

C#

```
public class Orders
{
    public Book[] Books;
}

public class Book
{
    public string ISBN;
}

public class ExpandedBook:Book
{
    public bool NewEdition;
}

public class Run
{
    public void SerializeObject(string filename)
    {
        // Each overridden field, property, or type requires
        // an XmlAttributes instance.
        XmlAttributes attrs = new XmlAttributes();

        // Creates an XmlElementAttribute instance to override the
        // field that returns Book objects. The overridden field
        // returns Expanded objects instead.
```

```
        XmlElementAttribute attr = new XmlElementAttribute();
        attr.ElementName = "NewBook";
        attr.Type = typeof(ExpandedBook);

        // Adds the element to the collection of elements.
        attrs.XmlElements.Add(attr);

        // Creates the XmlAttributeOverrides instance.
        XmlAttributeOverrides attrOverrides = new XmlAttributeOverrides();

        // Adds the type of the class that contains the overridden
        // member, as well as the XmlAttributes instance to override it
        // with, to the XmlAttributeOverrides.
        attrOverrides.Add(typeof(Orders), "Books", attrs);

        // Creates the XmlSerializer using the XmlAttributeOverrides.
        XmlSerializer s =
        new XmlSerializer(typeof(Orders), attrOverrides);

        // Writing the file requires a TextWriter instance.
        TextWriter writer = new StreamWriter(filename);

        // Creates the object to be serialized.
        Orders myOrders = new Orders();

        // Creates an object of the derived type.
        ExpandedBook b = new ExpandedBook();
        b.ISBN= "123456789";
        b.NewEdition = true;
        myOrders.Books = new ExpandedBook[]{b};

        // Serializes the object.
        s.Serialize(writer,myOrders);
        writer.Close();
    }

    public void DeserializeObject(string filename)
    {
        XmlAttributeOverrides attrOverrides =
            new XmlAttributeOverrides();
        XmlAttributes attrs = new XmlAttributes();

        // Creates an XmlElementAttribute to override the
        // field that returns Book objects. The overridden field
        // returns Expanded objects instead.
        XmlElementAttribute attr = new XmlElementAttribute();
        attr.ElementName = "NewBook";
        attr.Type = typeof(ExpandedBook);

        // Adds the XmlElementAttribute to the collection of objects.
        attrs.XmlElements.Add(attr);

        attrOverrides.Add(typeof(Orders), "Books", attrs);

        // Creates the XmlSerializer using the XmlAttributeOverrides.
```

```
        XmlSerializer s =
        new XmlSerializer(typeof(Orders), attrOverrides);

        FileStream fs = new FileStream(filename, FileMode.Open);
        Orders myOrders = (Orders) s.Deserialize(fs);
        Console.WriteLine("ExpandedBook:");

        // The difference between deserializing the overridden
        // XML document and serializing it is this: To read the derived
        // object values, you must declare an object of the derived type
        // and cast the returned object to it.
        ExpandedBook expanded;
        foreach(Book b in myOrders.Books)
        {
            expanded = (ExpandedBook)b;
            Console.WriteLine(
            expanded.ISBN + "\n" +
            expanded.NewEdition);
        }
    }
}
```

참조

- XmlSerializer
- XmlElementAttribute
- XmlAttributes
- XmlAttributeOverrides
- XML 및 SOAP serialization
- 방법: 개체 직렬화
- 방법: XML 스트림의 대체 요소 이름 지정

방법: XML 스트림의 대체 요소 이름 지정

이 클래스를 XmlSerializer사용하면 동일한 클래스 집합을 사용하여 둘 이상의 XML 스트림을 생성할 수 있습니다. 두 개의 서로 다른 XML 웹 서비스에는 약간의 차이만 있는 동일한 기본 정보가 필요하기 때문에 이 작업을 수행할 수 있습니다. 예를 들어 책에 대한 주문을 처리하므로 둘 다 ISBN 번호가 필요한 두 개의 XML 웹 서비스를 상상해 보세요. 한 서비스는 태그 `<ISBN>` 를 사용하고 두 번째 서비스는 태그 `<BookID>`를 사용합니다. 이름이 `Book`지정된 `ISBN` 필드가 포함된 클래스가 있습니다. 클래스 인스턴스가 `Book` serialize되면 기본적으로 ISBN(멤버 이름)을 태그 요소 이름으로 사용합니다. 첫 번째 XML 웹 서비스의 경우 예상대로 수행됩니다. 그러나 XML 스트림을 두 번째 XML 웹 서비스로 보내려면 태그의 요소 이름이 `BookID`되도록 serialization을 재정의해야 합니다.

대체 요소 이름을 사용하여 XML 스트림을 만들려면

1. 클래스의 인스턴스를 만듭니다 XmlElementAttribute .
2. ElementName의 XmlElementAttribute을 "BookID"로 설정합니다.
3. 클래스의 인스턴스를 만듭니다 XmlAttributes .
4. `XmlElementAttribute` 속성을 통해 액세스하는 컬렉션에 XmlElements 개체를 XmlAttributes에 추가합니다.
5. 클래스의 인스턴스를 만듭니다 XmlAttributeOverrides .
6. `XmlAttributes`를 XmlAttributeOverrides에 추가하고, 재정의할 개체의 형식과 재정의되는 멤버의 이름을 전달합니다.
7. `XmlSerializer`을(를) 사용하여 `XmlAttributeOverrides` 클래스의 인스턴스를 만듭니다.
8. 클래스의 인스턴스를 `Book` 만들고 직렬화하거나 역직렬화합니다.

예시

C#

```
public void SerializeOverride()
{
    // Creates an XmlElementAttribute with the alternate name.
    XmlElementAttribute myElementAttribute = new XmlElementAttribute();
    myElementAttribute.ElementName = "BookID";
    XmlAttributes myAttributes = new XmlAttributes();
    myAttributes.XmlElements.Add(myElementAttribute);
    XmlAttributeOverrides myOverrides = new XmlAttributeOverrides();
```

```
    myOverrides.Add(typeof(Book), "ISBN", myAttributes);
    XmlSerializer mySerializer =
    new XmlSerializer(typeof(Book), myOverrides);
    Book b = new Book();
    b.ISBN = "123456789";
    // Creates a StreamWriter to write the XML stream to.
    StreamWriter writer = new StreamWriter("Book.xml");
    mySerializer.Serialize(writer, b);
}
```

XML 스트림은 다음과 유사할 수 있습니다.

XML

```
<Book>
    <BookID>123456789</BookID>
</Book>
```

참고하십시오

- XmlElementAttribute
- XmlAttributes
- XmlAttributeOverrides
- XML 및 SOAP Serialization
- XmlSerializer
- 개체를 직렬화하는 방법
- 방법: 개체 디시리얼라이즈

Last updated on 2026. 03. 31.

XML 요소 및 XML 특성 이름을 한정하는 방법

아티클 • 2024. 03. 05.

클래스의 인스턴스에 의해 포함된 XML 네임스페이스는 XmlSerializerNamespacesNamespaces in XML 이라는 W3C(World Wide Web 컨소시엄) 사양을 따라야 합니다.

XML 네임스페이스는 XML 문서에서 XML 요소 및 XML 특성의 이름을 정규화하는 메서드를 제공합니다. 정규화된 이름은 콜론으로 구분된 접두사와 로컬 이름으로 구성됩니다. 접두사는 자리 표시자로만 기능하여 네임스페이스를 지정하는 URI에 매핑됩니다. 보편적으로 관리되는 URI 네임스페이스와 로컬 이름을 조합하면 보편적으로 고유한 이름이 만들어집니다.

`XmlSerializerNamespaces`의 인스턴스를 만들고 네임스페이스 쌍을 개체에 추가하면 XML 문서에 사용되는 접두사를 지정할 수 있습니다.

XML 문서에서 정규화된 이름을 만들려면

1. `XmlSerializerNamespaces` 클래스의 인스턴스를 만듭니다.
2. 모든 접두사와 네임스페이스 쌍을 `XmlSerializerNamespaces`에 추가합니다.
3. 적절한 `System.Xml.Serialization` 특성을 XmlSerializer가 XML 문서로 serialize할 각 멤버나 클래스에 적용합니다.

   사용할 수 있는 특성은 XmlAnyElementAttribute, XmlArrayAttribute, XmlArrayItemAttribute, XmlAttributeAttribute, XmlElementAttribute, XmlRootAttribute 및 XmlTypeAttribute입니다.
4. 각 특성의 `Namespace` 속성을 `XmlSerializerNamespaces`의 네임스페이스 값 중 하나로 설정합니다.
5. `XmlSerializerNamespaces`를 `XmlSerializer`의 `Serialize` 메서드에 전달합니다.

예시

다음 예제에서는 `XmlSerializerNamespaces`를 만들고 두 개의 접두사와 네임스페이스 쌍을 개체에 추가합니다. 코드에서는 `XmlSerializer` 클래스의 인스턴스를 serialize하는 데

사용되는 `Books`를 만듭니다. 코드는 `Serialize`를 사용하여 `XmlSerializerNamespaces` 메서드를 호출하여 XML이 접두사가 지정된 네임스페이스를 포함할 수 있게 됩니다.

C#

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class Program
{
    public static void Main()
    {
        SerializeObject("XmlNamespaces.xml");
    }

    public static void SerializeObject(string filename)
    {
        var mySerializer = new XmlSerializer(typeof(Books));
        // Writing a file requires a TextWriter.
        TextWriter myWriter = new StreamWriter(filename);

        // Creates an XmlSerializerNamespaces and adds two
        // prefix-namespace pairs.
        var myNamespaces = new XmlSerializerNamespaces();
        myNamespaces.Add("books", "http://www.cpandl.com");
        myNamespaces.Add("money", "http://www.cohowinery.com");

        // Creates a Book.
        var myBook = new Book();
        myBook.TITLE = "A Book Title";
        var myPrice = new Price();
        myPrice.price = (decimal) 9.95;
        myPrice.currency = "US Dollar";
        myBook.PRICE = myPrice;
        var myBooks = new Books();
        myBooks.Book = myBook;
        mySerializer.Serialize(myWriter, myBooks, myNamespaces);
        myWriter.Close();
    }
}

public class Books
{
    [XmlElement(Namespace = "http://www.cohowinery.com")]
    public Book Book;
}

[XmlType(Namespace ="http://www.cpandl.com")]
public class Book
{
    [XmlElement(Namespace = "http://www.cpandl.com")]
    public string TITLE;
```

```
    [XmlElement(Namespace ="http://www.cohowinery.com")]
    public Price PRICE;
}

public class Price
{
    [XmlAttribute(Namespace = "http://www.cpandl.com")]
    public string currency;
    [XmlElement(Namespace = "http://www.cohowinery.com")]
    public decimal price;
}
```

참고 항목

- XmlSerializer
- XML 스키마 정의 도구 및 XML serialization
- XML serialization 소개
- XmlSerializer 클래스
- XML serialization을 제어하는 특성
- 방법: XML 스트림의 대체 요소 이름 지정
- 방법: 개체 직렬화
- 방법: 개체 역직렬화

방법: 개체를 SOAP 인코딩된 XML 스트림으로 직렬화

아티클 • 2023. 06. 08.

SOAP 메시지는 XML을 사용하여 생성되므로 XmlSerializer 클래스를 사용하여 클래스를 직렬화하고 인코딩된 SOAP 메시지를 생성할 수 있습니다. 결과 XML은 World Wide Web 컨소시엄 문서의 5단원 "SOAP(Simple Object Access Protocol) 1.1"을 따릅니다. SOAP 메시지를 통해 통신하는 XML Web services를 만들 때는 특별한 SOAP 특성 집합을 클래스와 클래스 멤버에 적용하여 XML 스트림을 사용자 지정할 수 있습니다. 특성 목록을 보려면 인코딩된 SOAP serialization을 제어하는 특성을 참조하세요.

개체를 SOAP 인코딩된 XML 스트림으로 serialize하려면

1. XML 스키마 정의 도구(Xsd.exe)를 사용하여 클래스를 만듭니다.
2. `System.Xml.Serialization`에 있는 하나 이상의 특수 특성을 적용합니다. "인코딩된 SOAP serialization을 제어하는 특성"의 목록을 참조하십시오.
3. 새 `XmlTypeMapping`를 만들고 serialize된 클래스의 형식으로 `SoapReflectionImporter` 메서드를 호출하여 `ImportTypeMapping`을 만듭니다.

   다음 코드 예제에서는 `SoapReflectionImporter` 클래스의 `ImportTypeMapping` 메서드를 호출하여 `XmlTypeMapping`을 만듭니다.

   ```
   C#

   // Serializes a class named Group as a SOAP message.
   XmlTypeMapping myTypeMapping =
       new SoapReflectionImporter().ImportTypeMapping(typeof(Group));
   ```

4. `XmlSerializer`을 `XmlTypeMapping` 생성자로 전달하여 XmlSerializer(XmlTypeMapping) 클래스의 인스턴스를 만듭니다.

   ```
   C#

   XmlSerializer mySerializer = new XmlSerializer(myTypeMapping);
   ```

5. `Serialize` 또는 `Deserialize` 메서드를 호출합니다.

예제

C#

```
// Serializes a class named Group as a SOAP message.
XmlTypeMapping myTypeMapping =
    new SoapReflectionImporter().ImportTypeMapping(typeof(Group));
XmlSerializer mySerializer = new XmlSerializer(myTypeMapping);
```

참조

- XML 및 SOAP serialization
- 인코딩된 SOAP serialization을 제어하는 특성
- XML Web Services의 XML serialization
- 방법: 개체 직렬화
- 방법: 개체 역직렬화
- 방법: 인코딩된 SOAP XML Serialization 재정의

방법: 인코딩된 SOAP XML Serialization 재정의

아티클 • 2024. 03. 12.

개체의 XML serialization을 SOAP 메시지로 재정의하는 프로세스는 표준 XML serialization을 재정의하는 프로세스와 비슷합니다. 표준 XML serialization을 재정의하는 방법에 대한 자세한 내용은 방법: XML 스트림의 대체 요소 이름 지정을 참조하세요.

개체의 serialization을 SOAP 메시지로 재정의하려면

1. SoapAttributeOverrides 클래스의 인스턴스를 만듭니다.
2. serialize되는 각 클래스 멤버에 대해 `SoapAttributes`를 만듭니다.
3. serialize되는 멤버에 적절하게 XML serialization에 영향을 주는 하나 이상의 특성의 인스턴스를 만듭니다. 자세한 내용은 "인코딩된 SOAP serialization을 제어하는 특성"을 참조하십시오.
4. `SoapAttributes`의 적절한 속성을 3단계에서 만든 특성으로 설정합니다.
5. `SoapAttributes`를 `SoapAttributeOverrides`에 추가합니다.
6. `XmlTypeMapping`를 사용하여 `SoapAttributeOverrides`을 만듭니다. `SoapReflectionImporter.ImportTypeMapping` 메서드를 사용합니다.
7. `XmlSerializer`을 사용하여 `XmlTypeMapping`를 만듭니다.
8. 개체를 직렬화하거나 역직렬화합니다.

예시

다음 코드 예제에서는 파일을 두 가지 방법으로 serialize합니다. 우선 `XmlSerializer` 클래스의 동작을 재정의하지 않고 serialize하고, 두 번째로 동작을 재정의하여 serialize합니다. 예제에는 몇 개의 멤버가 있는 `Group`이라는 클래스가 포함됩니다. `SoapElementAttribute`와 같은 다양한 특성이 클래스 멤버에 적용되었습니다. 클래스가 `SerializeOriginal` 메서드로 serialize될 때 특성이 SOAP 메시지 내용을 제어합니다. `SerializeOverride` 메서드가 호출될 때 `XmlSerializer`의 동작은 다양한 특성을 만들고 `SoapAttributes`의 속성을 해당 특성으로 적절하게 설정하여 재정의됩니다.

C#

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;
using System.Xml.Schema;

public class Group
{
    [SoapAttribute (Namespace = "http://www.cpandl.com")]
    public string GroupName;

    [SoapAttribute(DataType = "base64Binary")]
    public Byte [] GroupNumber;

    [SoapAttribute(DataType = "date", AttributeName = "CreationDate")]
    public DateTime Today;
    [SoapElement(DataType = "nonNegativeInteger", ElementName = "PosInt")]
    public string PositiveInt;
    // This is ignored when serialized unless it is overridden.
    [SoapIgnore]
    public bool IgnoreThis;

    public GroupType Grouptype;

    [SoapInclude(typeof(Car))]
    public Vehicle myCar(string licNumber)
    {
        Vehicle v;
        if(licNumber == "")
            {
                v = new Car();
            v.licenseNumber = "!!!!!!";
        }
        else
        {
            v = new Car();
            v.licenseNumber = licNumber;
        }
        return v;
    }
}

public abstract class Vehicle
{
    public string licenseNumber;
    public DateTime makeDate;
}

public class Car: Vehicle
{
}
```

```
public enum GroupType
{
    // These enums can be overridden.
    small,
    large
}

public class Run
{
    public static void Main()
    {
        Run test = new Run();
        test.SerializeOriginal("SoapOriginal.xml");
        test.SerializeOverride("SoapOverrides.xml");
        test.DeserializeOriginal("SoapOriginal.xml");
        test.DeserializeOverride("SoapOverrides.xml");

    }
    public void SerializeOriginal(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlTypeMapping myMapping =
        (new SoapReflectionImporter().ImportTypeMapping(
        typeof(Group)));
        XmlSerializer mySerializer =
        new XmlSerializer(myMapping);

        // Writing the file requires a TextWriter.
        TextWriter writer = new StreamWriter(filename);

        // Creates an instance of the class that will be serialized.
        Group myGroup = new Group();

        // Sets the object properties.
        myGroup.GroupName = ".NET";

        Byte [] hexByte = new Byte[2]{Convert.ToByte(100),
        Convert.ToByte(50)};
        myGroup.GroupNumber = hexByte;

        DateTime myDate = new DateTime(2002,5,2);
        myGroup.Today = myDate;

        myGroup.PositiveInt= "10000";
        myGroup.IgnoreThis=true;
        myGroup.Grouptype= GroupType.small;
        Car thisCar =(Car)  myGroup.myCar("1234566");

        // Prints the license number just to prove the car was created.
        Console.WriteLine("License#: " + thisCar.licenseNumber + "\n");

        // Serializes the class and closes the TextWriter.
        mySerializer.Serialize(writer, myGroup);
        writer.Close();
    }
```

```
public void SerializeOverride(string filename)
{
    // Creates an instance of the XmlSerializer class
    // that overrides the serialization.
    XmlSerializer overRideSerializer = CreateOverrideSerializer();

    // Writing the file requires a TextWriter.
    TextWriter writer = new StreamWriter(filename);

    // Creates an instance of the class that will be serialized.
    Group myGroup = new Group();

    // Sets the object properties.
    myGroup.GroupName = ".NET";

    Byte [] hexByte = new Byte[2]{Convert.ToByte(100),
    Convert.ToByte(50)};
    myGroup.GroupNumber = hexByte;

    DateTime myDate = new DateTime(2002,5,2);
    myGroup.Today = myDate;

    myGroup.PositiveInt= "10000";
    myGroup.IgnoreThis=true;
    myGroup.Grouptype= GroupType.small;
    Car thisCar =(Car)  myGroup.myCar("1234566");

    // Serializes the class and closes the TextWriter.
    overRideSerializer.Serialize(writer, myGroup);
     writer.Close();
}

public void DeserializeOriginal(string filename)
{
    // Creates an instance of the XmlSerializer class.
    XmlTypeMapping myMapping =
    (new SoapReflectionImporter().ImportTypeMapping(
    typeof(Group)));
    XmlSerializer mySerializer =
    new XmlSerializer(myMapping);

    TextReader reader = new StreamReader(filename);

    // Deserializes and casts the object.
    Group myGroup;
    myGroup = (Group) mySerializer.Deserialize(reader);

    Console.WriteLine(myGroup.GroupName);
    Console.WriteLine(myGroup.GroupNumber[0]);
    Console.WriteLine(myGroup.GroupNumber[1]);
    Console.WriteLine(myGroup.Today);
    Console.WriteLine(myGroup.PositiveInt);
    Console.WriteLine(myGroup.IgnoreThis);
    Console.WriteLine();
```

```
    }

    public void DeserializeOverride(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlSerializer overRideSerializer = CreateOverrideSerializer();
        // Reading the file requires a TextReader.
        TextReader reader = new StreamReader(filename);

        // Deserializes and casts the object.
        Group myGroup;
        myGroup = (Group) overRideSerializer.Deserialize(reader);

        Console.WriteLine(myGroup.GroupName);
        Console.WriteLine(myGroup.GroupNumber[0]);
        Console.WriteLine(myGroup.GroupNumber[1]);
        Console.WriteLine(myGroup.Today);
        Console.WriteLine(myGroup.PositiveInt);
        Console.WriteLine(myGroup.IgnoreThis);
    }

    private XmlSerializer CreateOverrideSerializer()
    {
        SoapAttributeOverrides mySoapAttributeOverrides =
        new SoapAttributeOverrides();
        SoapAttributes soapAtts = new SoapAttributes();

        SoapElementAttribute mySoapElement = new SoapElementAttribute();
        mySoapElement.ElementName = "xxxx";
        soapAtts.SoapElement = mySoapElement;
        mySoapAttributeOverrides.Add(typeof(Group), "PositiveInt",
        soapAtts);

        // Overrides the IgnoreThis property.
        SoapIgnoreAttribute myIgnore = new SoapIgnoreAttribute();
        soapAtts = new SoapAttributes();
        soapAtts.SoapIgnore = false;
        mySoapAttributeOverrides.Add(typeof(Group), "IgnoreThis",
        soapAtts);

        // Overrides the GroupType enumeration.
        soapAtts = new SoapAttributes();
        SoapEnumAttribute xSoapEnum = new SoapEnumAttribute();
        xSoapEnum.Name = "Over1000";
        soapAtts.SoapEnum = xSoapEnum;

        // Adds the SoapAttributes to the
        // mySoapAttributeOverrides.
        mySoapAttributeOverrides.Add(typeof(GroupType), "large",
        soapAtts);

        // Creates a second enumeration and adds it.
        soapAtts = new SoapAttributes();
        xSoapEnum = new SoapEnumAttribute();
        xSoapEnum.Name = "ZeroTo1000";
```

```
        soapAtts.SoapEnum = xSoapEnum;
        mySoapAttributeOverrides.Add(typeof(GroupType), "small",
        soapAtts);

        // Overrides the Group type.
        soapAtts = new SoapAttributes();
        SoapTypeAttribute soapType = new SoapTypeAttribute();
        soapType.TypeName = "Team";
        soapAtts.SoapType = soapType;
        mySoapAttributeOverrides.Add(typeof(Group),soapAtts);

        // Creates an XmlTypeMapping that is used to create an instance
        // of the XmlSerializer class. Then returns the XmlSerializer.
        XmlTypeMapping myMapping = (new SoapReflectionImporter(
        mySoapAttributeOverrides)).ImportTypeMapping(typeof(Group));

        XmlSerializer ser = new XmlSerializer(myMapping);
        return ser;
    }
}
```

참고 항목

- XML 및 SOAP serialization
- 인코딩된 SOAP serialization을 제어하는 특성
- XML Web Services의 XML serialization
- 방법: 개체 직렬화
- 방법: 개체 역직렬화
- 방법: 개체를 SOAP 인코딩된 XML 스트림으로 직렬화

방법: 직렬화된 데이터 청크

아티클 • 2024. 08. 08.

> ⚠ **경고**
>
> `BinaryFormatter`를 사용한 이진 직렬화는 위험할 수 있습니다. 자세한 내용은 BinaryFormatter 보안 가이드 **및 BinaryFormatter 마이그레이션 가이드를 참조**하세요.

대용량 데이터 집합을 웹 서비스 메시지에서 전송할 때 발생하는 두 가지 문제는 다음과 같습니다.

1. serialization 엔진의 버퍼링으로 인한 대용량 작업 집합(메모리)
2. Base64 인코딩 후 33퍼센트 확장으로 인한 과도한 대역폭 소비

이러한 문제를 해결하려면 IXmlSerializable 인터페이스를 구현하여 serialization과 deserialization을 제어합니다. 특히 WriteXml 및 ReadXml 메서드를 구현하여 데이터를 청크합니다.

서버측 청크를 구현하려면

1. 서버 시스템에서 웹 메서드는 ASP.NET 버퍼링을 끄고 IXmlSerializable을 구현하는 형식을 반환해야 합니다.
2. IXmlSerializable을 구현하는 형식이 WriteXml 메서드의 데이터를 청크합니다.

클라이언트측 처리를 구현하려면

1. 클라이언트 프록시에서 웹 메서드를 변경하여 IXmlSerializable을 구현하는 형식을 반환합니다. SchemaImporterExtension을 사용하여 이 작업을 자동으로 수행할 수 있지만 여기에는 표시되지 않습니다.
2. ReadXml 메서드를 구현하여 청크된 데이터 스트림을 읽고 바이트를 디스크에 씁니다. 또한 이 구현은 진행률 표시줄 등과 같은 그래픽 컨트롤에서 사용할 수 있는 진행률 이벤트도 발생시킵니다.

예시

다음 코드 예제에서는 ASP.NET 버퍼링을 끄는 클라이언트의 웹 메서드를 보여 줍니다. 또한 IXmlSerializable 메서드의 데이터를 청크하는 WriteXml 인터페이스의 클라이언트측 구현도 보여 줍니다.

C#

```
[WebMethod]
[SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
public SongStream DownloadSong(DownloadAuthorization Authorization, string
filePath)
{
    // Turn off response buffering.
    System.Web.HttpContext.Current.Response.Buffer = false;
    // Return a song.
    SongStream song = new SongStream(filePath);
    return song;
}
```

C#

```
[XmlSchemaProvider("MySchema")]
public class SongStream : IXmlSerializable
{
    private const string ns = "http://demos.Contoso.com/webservices";
    private string filePath;

    public SongStream() { }

    public SongStream(string filePath)
    {
        this.filePath = filePath;
    }

    // This is the method named by the XmlSchemaProviderAttribute applied to
the type.
    public static XmlQualifiedName MySchema(XmlSchemaSet xs)
    {
        // This method is called by the framework to get the schema for this
type.
        // We return an existing schema from disk.

        XmlSerializer schemaSerializer = new
XmlSerializer(typeof(XmlSchema));
        string xsdPath = null;
        // NOTE: replace the string with your own path.
        xsdPath =
System.Web.HttpContext.Current.Server.MapPath("SongStream.xsd");
        XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
            new XmlTextReader(xsdPath), null);
        xs.XmlResolver = new XmlUrlResolver();
        xs.Add(s);
```

```
        return new XmlQualifiedName("songStream", ns);
    }

    void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
    {
        // This is the chunking code.
        // ASP.NET buffering must be turned off for this to work.

        int bufferSize = 4096;
        char[] songBytes = new char[bufferSize];
        FileStream inFile = File.Open(this.filePath, FileMode.Open,
FileAccess.Read);

        long length = inFile.Length;

        // Write the file name.
        writer.WriteElementString("fileName", ns,
Path.GetFileNameWithoutExtension(this.filePath));

        // Write the size.
        writer.WriteElementString("size", ns, length.ToString());

        // Write the song bytes.
        writer.WriteStartElement("song", ns);

        StreamReader sr = new StreamReader(inFile, true);
        int readLen = sr.Read(songBytes, 0, bufferSize);

        while (readLen > 0)
        {
            writer.WriteStartElement("chunk", ns);
            writer.WriteChars(songBytes, 0, readLen);
            writer.WriteEndElement();

            writer.Flush();
            readLen = sr.Read(songBytes, 0, bufferSize);
        }

        writer.WriteEndElement();
        inFile.Close();
    }

    XmlSchema IXmlSerializable.GetSchema()
    {
        throw new NotImplementedException();
    }

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
        throw new NotImplementedException();
    }
}
```

C#

```
public class SongFile : IXmlSerializable
{
    public static event ProgressMade OnProgress;

    public SongFile()
    { }

    private const string ns = "http://demos.teched2004.com/webservices";
    public static string MusicPath;
    private string filePath;
    private double size;

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
        reader.ReadStartElement("DownloadSongResult", ns);
        ReadFileName(reader);
        ReadSongSize(reader);
        ReadAndSaveSong(reader);
        reader.ReadEndElement();
    }

    void ReadFileName(XmlReader reader)
    {
        string fileName = reader.ReadElementString("fileName", ns);
        this.filePath =
            Path.Combine(MusicPath, Path.ChangeExtension(fileName, ".mp3"));
    }

    void ReadSongSize(XmlReader reader)
    {
        this.size = Convert.ToDouble(reader.ReadElementString("size", ns));
    }

    void ReadAndSaveSong(XmlReader reader)
    {
        FileStream outFile = File.Open(
            this.filePath, FileMode.Create, FileAccess.Write);

        string songBase64;
        byte[] songBytes;
        reader.ReadStartElement("song", ns);
        double totalRead = 0;
        while (true)
        {
            if (reader.IsStartElement("chunk", ns))
            {
                songBase64 = reader.ReadElementString();
                totalRead += songBase64.Length;
                songBytes = Convert.FromBase64String(songBase64);
                outFile.Write(songBytes, 0, songBytes.Length);
                outFile.Flush();

                if (OnProgress != null)
                {
```

```
                    OnProgress(100 * (totalRead / size));
                }
            }

            else
            {
                break;
            }
        }

        outFile.Close();
        reader.ReadEndElement();
    }

    public void Play()
    {
        System.Diagnostics.Process.Start(this.filePath);
    }

    XmlSchema IXmlSerializable.GetSchema()
    {
        throw new NotImplementedException();
    }

    public void WriteXml(XmlWriter writer)
    {
        throw new NotImplementedException();
    }
}
```

코드 컴파일

- 코드에서는 System, System.Runtime.Serialization, System.Web.Services, System.Web.Services.Protocols, System.Xml, System.Xml.Serialization 및 System.Xml.Schema 네임스페이스를 사용합니다.

참고 항목

- 사용자 지정 serialization

수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

설명서 문제 열기

제품 사용자 의견 제공

`<system.xml.serialization>` 요소

XML serialization을 제어하기 위한 최상위 요소입니다. 구성 파일에 대한 자세한 내용은 구성 파일 스키마를 참조하세요.

<구성>
< .serializationsystem.xml>

문법

XML

```
<system.xml.serialization>
</system.xml.serialization>
```

특성 및 요소

다음의 섹션은 특성, 자식 요소 및 부모 요소에 대해 설명합니다.

Attributes

없음.

자식 요소

테이블 확장

요소	Description
<dateTimeSerialization> 요소	개체의 serialization 모드를 DateTime 결정합니다.
<schemaImporterExtensions> 요소	XSD 형식을 .NET 형식에 XmlSchemaImporter 매핑하는 데 사용되는 형식을 포함합니다.

부모 요소

테이블 확장

요소	Description
<configuration> 요	공용 언어 런타임 및 .NET Framework 애플리케이션에서 사용하는 모든 구성 파일

요소	Description
소	의 루트 요소입니다.

예시

다음 코드 예제에서는 개체의 DateTime serialization 모드를 지정하는 방법과 XSD 형식을 .NET 형식에 XmlSchemaImporter 매핑할 때 사용되는 형식을 추가하는 방법을 보여 줍니다.

XML

```
<system.xml.serialization>
    <xmlSerializer checkDeserializeAdvances="false" />
    <dateTimeSerialization mode = "Local" />
    <schemaImporterExtensions>
        <add
        name = "MobileCapabilities"
        type = "System.Web.Mobile.MobileCapabilities,
        System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b03f5f6f11d40a3a" />
    </schemaImporterExtensions>
</system.xml.serialization>
```

참고하십시오

- XmlSchemaImporter
- DateTimeSerializationSection.DateTimeSerializationMode
- 구성 파일 스키마
- <dateTimeSerialization> 요소
- <schemaImporterExtensions> 요소
- <add> 에 대한 요소 <schemaImporterExtensions>

Last updated on 2025. 12. 02.

`<dateTimeSerialization>` 요소

개체의 serialization 모드를 DateTime 결정합니다.

`<configuration>` `<dateTimeSerialization>`

문법

XML

```
<dateTimeSerialization
    mode = "Roundtrip|Local"
/>
```

특성 및 요소

다음의 섹션은 특성, 자식 요소 및 부모 요소에 대해 설명합니다.

Attributes

테이블 확장

Attributes	Description
`mode`	Optional. serialization 모드를 지정합니다. 값 중 DateTimeSerializationSection.DateTimeSerializationMode 하나로 설정합니다. 기본값은 **RoundTrip**입니다.

자식 요소

없음.

부모 요소

테이블 확장

요소	Description
.serialization system.xml	XML serialization을 제어하기 위한 최상위 요소입니다.

비고

이 속성을 **로컬**DateTime로 설정 하면 개체는 항상 로컬 시간으로 형식이 지정됩니다. 즉, 현지 표준 시간대 정보는 항상 직렬화된 데이터에 포함됩니다.

이 속성이 **Roundtrip**으로 설정되면 개체가 로컬, DateTime UTC 또는 지정되지 않은 표준 시간대에 있는지 여부를 확인하기 위해 검사됩니다. 그런 다음 개체는 DateTime 이 정보가 유지되는 방식으로 직렬화됩니다. 기본 동작이며 이전 버전의 프레임워크와 통신하지 않는 모든 새 애플리케이션에 권장되는 동작입니다.

참고하십시오

- DateTime
- XmlSchemaImporter
- DateTimeSerializationSection.DateTimeSerializationMode
- 구성 파일 스키마
- <schemaImporterExtensions> 요소
- <add> 에 대한 요소 <schemaImporterExtensions>
- <system.xml.serialization> 요소

Last updated on 2025. 12. 02.

`<schemaImporterExtensions>` 요소

XSD 형식을 .NET 형식에 XmlSchemaImporter 매핑하는 데 사용되는 형식을 포함합니다. 구성 파일에 대한 자세한 내용은 구성 파일 스키마를 참조하세요.

문법

XML

```xml
<schemaImporterExtensions>
    <!-- Add types -->
</schemaImporterExtensions>
```

자식 요소

테이블 확장

요소	Description
<add> Element for <schemaImporterExtensions>	매핑을 만드는 데 사용되는 XmlSchemaImporter 형식을 추가합니다.

부모 요소

테이블 확장

요소	Description
<system.xml.serialization> 요소	XML serialization을 제어하기 위한 최상위 요소입니다.

예시

다음 코드 예제에서는 XSD 형식을 .NET 형식에 XmlSchemaImporter 매핑할 때 사용되는 형식을 추가하는 방법을 보여 줍니다.

XML

```xml
<system.xml.serialization>
    <schemaImporterExtensions>
        <add name = "MobileCapabilities" type =
        "System.Web.Mobile.MobileCapabilities,
        System.Web.Mobile, Version - 2.0.0.0, Culture = neutral,
```

```
        PublicKeyToken = b03f5f6f11d40a3a" />
    </schemaImporterExtensions>
</system.xml.serialization>
```

참고하십시오

- XmlSchemaImporter
- DateTimeSerializationSection.DateTimeSerializationMode
- 구성 파일 스키마
- <dateTimeSerialization> 요소
- <add> 에 대한 요소 <schemaImporterExtensions>
- <system.xml.serialization> 요소

Last updated on 2025. 12. 02.

`<add>` 에 대한 요소

`<schemaImporterExtensions>`

XSD 형식을 매핑하는 XmlSchemaImporter 데 사용되는 형식을 .NET 형식에 추가합니다. 구성 파일에 대한 자세한 내용은 구성 파일 스키마를 참조하세요.

<구성>system.xml.serialization<>schemaImporterExtensions><추가<>

문법

XML

```
<add name = "typeName" type="fully qualified type [,Version=version number]
[,Culture=culture] [,PublicKeyToken= token]"/>
```

특성 및 요소

다음의 섹션은 특성, 자식 요소 및 부모 요소에 대해 설명합니다.

Attributes

테이블 확장

특성	Description
name	인스턴스를 찾는 데 사용되는 간단한 이름입니다.
type	필수 사항입니다. 추가할 스키마 확장 클래스를 지정합니다. **형식** 특성 값은 한 줄에 있어야 하며 정규화된 형식 이름을 포함해야 합니다. 어셈블리가 GAC(전역 어셈블리 캐시)에 배치되면 서명된 어셈블리의 버전, 문화권 및 공개 키 토큰도 포함해야 합니다.

자식 요소

없음.

부모 요소

테이블 확장

요소	Description
<schemaImporterExtensions>	에 사용되는 형식을 XmlSchemaImporter포함합니다.

예시

다음 코드 예제에서는 XmlSchemaImporter가 형식을 매핑할 때 사용할 수 있는 확장 형식을 추가합니다.

XML

```
<configuration>
  <system.xml.serialization>
    <schemaImporterExtensions>
       <add name="contoso" type="System.Web.Mobile.MobileCapabilities,
       System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a" />
    </schemaImporterExtensions>
  </system.xml.serialization>
</configuration>
```

참고하십시오

- XmlSchemaImporter
- <system.xml.serialization> 요소
- <schemaImporterExtensions> 요소

Last updated on 2025. 12. 02.

`<xmlSerializer>` 요소

진행률 XmlSerializer 에 대한 추가 검사가 수행되는지 여부를 지정합니다.

`<configuration>` `<system.xml.serialization>`

문법

XML

```
<xmlSerializer checkDeserializerAdvance = "true|false" />
```

특성 및 요소

다음의 섹션은 특성, 자식 요소 및 부모 요소에 대해 설명합니다.

Attributes

테이블 확장

특성	Description
checkDeserializeAdvances	진행률을 XmlSerializer 확인할지 여부를 지정합니다. 특성을 "true" 또는 "false"로 설정합니다. 기본값은 "true"입니다.
useLegacySerializationGeneration	파일에 C# 코드를 작성한 다음 어셈블리로 컴파일하여 어셈블리를 생성하는 레거시 serialization 생성을 사용할지 여부를 XmlSerializer 지정합니다. 기본값은 **false**입니다.

자식 요소

없음.

부모 요소

테이블 확장

요소	Description
<system.xml.serialization> 요소	및 XmlSchemaImporter 클래스에 XmlSerializer 대한 구성 설정을 포함합니다.

비고

기본적으로 XmlSerializer 신뢰할 수 없는 데이터를 역직렬화할 때 잠재적인 서비스 거부 공격에 대한 추가 보안 계층을 제공합니다. 역직렬화 중에 무한 루프를 검색하려고 시도하여 이 작업을 수행합니다. 이러한 조건이 감지되면 "내부 오류: 역직렬화가 기본 스트림을 진행하지 못했습니다."라는 메시지와 함께 예외가 throw됩니다.

이 메시지를 수신한다고 해서 서비스 거부 공격이 진행 중임을 반드시 나타내는 것은 아닙니다. 드문 경우지만 무한 루프 검색 메커니즘은 가양성으로 생성되고 합법적인 들어오는 메시지에 대해 예외가 throw됩니다. 특정 애플리케이션에서 합법적인 메시지가 이 추가 보호 계층에 의해 거부되는 경우 **checkDeserializeAdvances** 특성을 "false"로 설정합니다.

예시

다음 코드 예제에서는 **checkDeserializeAdvances** 특성을 "false"로 설정합니다.

XML

```
<configuration>
  <system.xml.serialization>
    <xmlSerializer checkDeserializeAdvances="false" />
  </system.xml.serialization>
</configuration>
```

참고하십시오

- XmlSerializer
- <system.xml.serialization> 요소
- XML 및 SOAP Serialization

Last updated on 2025. 12. 02.

XML Serializer 생성기 도구(Sgen.exe)

XML Serializer 생성기는 지정된 어셈블리의 형식에 대한 XML serialization 어셈블리를 만듭니다. serialization 어셈블리는 지정된 형식의 개체를 직렬화하거나 역직렬화할 때 XmlSerializer의 시작 성능을 향상시킵니다.

> **참고 항목**
>
> 이 도구는 .NET Framework 어셈블리와 관련이 있습니다. .NET(Core) 어셈블리용 XML serializer를 생성하려면 **.NET Core에서 Microsoft XML Serializer 생성기 사용을** 참조하세요.

문법

명령줄에서 도구를 실행합니다.

콘솔

```
sgen [options]
```

> **팁**
>
> .NET Framework 도구가 제대로 작동하려면 **Visual Studio 개발자 명령 프롬프트 또는 Visual Studio 개발자 PowerShell**을 사용하거나 환경 `Path` 변수 및 `Include` 환경 변수를 `Lib` 올바르게 설정해야 합니다. 이러한 환경 변수를 설정하려면 <실행합니다.

매개 변수

테이블 확장

옵션	설명
/a[ssembly]:*filename*	어셈블리에 포함된 모든 형식 또는 *파일 이름으로* 지정된 실행 파일에 대한 serialization 코드를 생성합니다. 하나의 파일 이름만 제공할 수 있습니다. 이 인수가 반복되면 마지막 파일 이름이 사용됩니다.
/c[ompiler]:*options*	C# 컴파일러에 전달할 옵션을 지정합니다. 모든 csc.exe 옵션은 컴파일러에 전달되므로 지원됩니다. 어셈블리에 서명하도록 지정하고 키 파일을 지정하는 데 사용할 수 있습니다.
/d[ebug]	디버거와 함께 사용할 수 있는 이미지를 생성합니다.

옵션	설명
/f[orce]	동일한 이름의 기존 어셈블리를 강제로 덮어씁니다. 기본값은 **false**입니다.
/help 또는 **/?**	이 도구의 명령 구문 및 옵션을 표시합니다.
/k[eep]	serialization 어셈블리로 컴파일된 후 생성된 원본 파일 및 기타 임시 파일의 삭제를 방지합니다. 도구가 특정 형식에 대한 serialization 코드를 생성하는지 여부를 확인하는 데 사용할 수 있습니다.
/n[ologo]	Microsoft 시작 배너의 표시를 억제합니다.
/o[ut]:*path*	생성된 어셈블리를 저장할 디렉터리를 지정합니다. **메모:** 생성된 어셈블리의 이름은 입력 어셈블리의 이름과 "xmlSerializers.dll"로 구성됩니다.
/p[roxytypes]	XML 웹 서비스 프록시 형식에 대해서만 serialization 코드를 생성합니다.
/r[eference]:*assemblyfiles*	XML serialization이 필요한 형식에서 참조하는 어셈블리를 지정합니다. 여러 어셈블리 파일을 쉼표로 구분하여 허용합니다.
/s[ilent]	성공 메시지를 표시하지 않습니다.
/t[ype]:*type*	지정된 형식에 대해서만 serialization 코드를 생성합니다.
/v[erbose]	디버깅을 위한 자세한 출력을 표시합니다. XmlSerializer를 사용하여 serialize할 수 없는 대상 어셈블리의 형식을 나열합니다.
/?	이 도구의 명령 구문 및 옵션을 표시합니다.

비고

XML Serializer 생성기를 사용하지 않으면 XmlSerializer 애플리케이션을 실행할 때마다 각 형식에 대해 직렬화 코드와 직렬화 어셈블리를 생성합니다. XML serialization 시작의 성능을 향상시키려면 Sgen.exe 도구를 사용하여 해당 어셈블리를 미리 생성합니다. 그런 다음 애플리케이션을 사용하여 이러한 어셈블리를 배포할 수 있습니다.

XML Serializer 생성기는 XML 웹 서비스 프록시를 사용하여 서버와 통신하는 클라이언트의 성능을 향상시킬 수도 있습니다. 직렬화 프로세스는 형식이 처음 로드될 때 성능이 저하되지 않기 때문입니다.

생성된 어셈블리는 웹 서비스의 서버 쪽에서 사용할 수 없습니다. 이 도구는 웹 서비스 클라이언트 및 수동 serialization 시나리오에만 사용됩니다.

참고 항목

이 `sgen` 도구는 **init 전용** setter와 호환되지 않습니다. 대상 어셈블리에 이 기능을 사용하는 공용 속성이 포함되어 있으면 도구가 실패합니다.

이름 지정

serialize할 형식이 포함된 어셈블리의 이름이 *MyType.dll*경우 연결된 serialization 어셈블리의 이름은 *MyType.XmlSerializers.dll*.

예시

다음 명령은 *Data.dll* 어셈블리에 포함된 모든 형식을 serialize하기 위해 *Data.XmlSerializers.dll* 이라는 어셈블리를 만듭니다.

콘솔

```
sgen Data.dll
```

Data.XmlSerializers.dll어셈블리는 *Data.dll* 형식을 직렬화하고 역직렬화해야 하는 코드에서 참조될 수 있습니다.

참고하십시오

- 도구
- 개발자 명령줄 셸

Last updated on 2026. 01. 06.

XML 스키마 정의 도구(Xsd.exe)

XML 스키마 정의(Xsd.exe) 도구는 XDR, XML 및 XSD 파일 또는 런타임 어셈블리의 클래스에서 XML 스키마 또는 공용 언어 런타임 클래스를 생성합니다.

XML 스키마 정의 도구(Xsd.exe)는 일반적으로 다음 경로에서 찾을 수 있습니다.
C:\Program Files (x86)\Microsoft SDKs\Windows\{version}\bin\NETFX {version} Tools

문법

명령줄에서 도구를 실행합니다.

콘솔

```
xsd file.xdr [-outputdir:directory][/parameters:file.xml]
xsd file.xml [-outputdir:directory] [/parameters:file.xml]
xsd file.xsd {/classes | /dataset} [/element:element]
             [/enableLinqDataSet] [/language:language]
                          [/namespace:namespace] [-outputdir:directory] [URI:uri]
                          [/parameters:file.xml]
xsd {file.dll | file.exe} [-outputdir:directory] [/type:typename [...]]
[/parameters:file.xml]
```

> **팁**
>
> .NET Framework 도구가 제대로 작동하려면 사용자 `Path` `Include` 및 `Lib` 환경 변수를 올바르게 설정해야 합니다. SDK<\version>\<Bin 디렉터리에 있는 >SDKVars.bat실행하여 이러한 환경 변수를 설정합니다. SDKVars.bat 모든 명령 셸에서 실행되어야 합니다.

논쟁

테이블 확장

논쟁	설명
file.extension	변환할 입력 파일을 지정합니다. 확장을 .xdr, .xml, .xsd, .dll또는 .exe중 하나로 지정해야 합니다. XDR 스키마 파일(.xdr 확장명)을 지정하면 Xsd.exe XDR 스키마를 XSD 스키마로 변환합니다. 출력 파일의 이름은 XDR 스키마와 같지만 확장명은 .xsd입니다. XML 파일(.xml 확장명)을 지정하면 Xsd.exe 파일의 데이터에서 스키마를 유추하고 XSD 스키마를 생성합니다. 출력 파일의 이름은 XML 파일과 같지만 확장명은 .xsd입니다.

논쟁	설명
	XML 스키마 파일(.xsd 확장명)을 지정하면 Xsd.exe XML 스키마에 해당하는 런타임 개체에 대한 소스 코드를 생성합니다. 런타임 어셈블리 파일(.exe 또는 .dll 확장명)을 지정하면 Xsd.exe 해당 어셈블리에서 하나 이상의 형식에 대한 스키마를 생성합니다. 이 `/type` 옵션을 사용하여 스키마를 생성할 형식을 지정할 수 있습니다. 출력 스키마의 이름은 schema0.xsd, schema1.xsd 등입니다. Xsd.exe 지정된 형식이 사용자 지정 특성을 사용하여 네임스페이스를 지정하는 경우에만 여러 스키마를 `XMLRoot` 생성합니다.

일반 옵션

테이블 확장

옵션	설명
/h[elp]	이 도구의 명령 구문 및 옵션을 표시합니다.
/o[utputdir]: *디렉터리*	출력 파일의 디렉터리를 지정합니다. 이 인수는 한 번만 표시할 수 있습니다. 기본값은 현재 디렉터리입니다.
/?	이 도구의 명령 구문 및 옵션을 표시합니다.
/p[arameters]: *file.xml*	지정된 .xml 파일에서 다양한 작업 모드에 대한 옵션을 읽습니다. 단축형은 `/p:`. 자세한 내용은 설명 섹션을 참조하세요.

XSD 파일 옵션

.xsd 파일에 대해 다음 옵션 중 하나만 지정해야 합니다.

테이블 확장

옵션	설명
/c[lasses]	지정된 스키마에 해당하는 클래스를 생성합니다. 개체에 XML 데이터를 읽으려면 메서드를 XmlSerializer.Deserialize 사용합니다.
/d[ataset]	지정된 스키마에 해당하는 클래스에서 DataSet 파생된 클래스를 생성합니다. XML 데이터를 파생 클래스로 읽으려면 메서드를 DataSet.ReadXml 사용합니다.

.xsd 파일에 대해 다음 옵션 중 원하는 옵션을 지정할 수도 있습니다.

테이블 확장

옵션	설명
/e[lement]:*element*	코드를 생성할 스키마의 요소를 지정합니다. 기본적으로 모든 요소가 입력됩니다. 이 인수를 두 번 이상 지정할 수 있습니다.
/enableDataBinding	데이터 바인딩을 INotifyPropertyChanged 사용하도록 설정하기 위해 생성된 모든 형식에 인터페이스를 구현합니다. 단축형은 `/edb`.
/enableLinqDataSet	(약식: `/eld`.) LINQ to DataSet을 사용하여 생성된 DataSet을 쿼리할 수 있도록 지정합니다. 이 옵션은 /dataset 옵션도 지정할 때 사용됩니다. 자세한 내용은 LINQ to DataSet 개요 및 형식화된 데이터 세트 쿼리를 참조하세요. LINQ 사용에 대한 일반적인 내용은 LINQ(Language-Integrated Query) - C# 또는 LINQ(Language-Integrated Query) - Visual Basic을 참조하세요.
/f[ields]	필드만 생성합니다. 기본적으로 지원 필드가 있는 속성 이 생성됩니다.
/l[anguage]:*언어*	사용할 프로그래밍 언어를 지정합니다. (기본값인 C#), (Visual Basic), `CS` (JScript) `VB` 또는 `JS` (Visual J#) 중에서 `VJS` 선택합니다. 구현하는 클래스의 정규화된 이름을 지정할 수도 있습니다. System.CodeDom.Compiler.CodeDomProvider
/n[amespace]:*namespace*	생성된 형식의 런타임 네임스페이스를 지정합니다. 기본 네임스페이스는 `Schemas`.
/nologo	배너를 제거합니다.
/주문	모든 파티클 멤버에 대해 명시적 순서 식별자를 생성합니다.
/o[ut]:*directoryName*	파일을 배치할 출력 디렉터리를 지정합니다. 기본값은 현재 디렉터리입니다.
/u[ri]:*uri*	코드를 생성할 스키마의 요소에 대한 URI를 지정합니다. 이 URI(있는 경우)는 옵션으로 지정된 모든 요소에 `/element` 적용됩니다.

DLL 및 EXE 파일 옵션

테이블 확장

옵션	설명
/t[ype]:*typename*	스키마를 만들 형식의 이름을 지정합니다. 여러 형식 인수를 지정할 수 있습니다. *typename*이 네임스페이스를 지정하지 않으면, Xsd.exe는 어셈블리 내에서 지정된 형식과 모든 형식을 일치시킵니다. *typename*이 네임스페이스를 지정하는 경우 해당 형식만 일치합니다. *typename*이 별표 문자(*)로 끝나는 경우 도구는 *앞의 문자열로 시작하는 모든 형식과 일치합니다. 옵션을 생략 `/type` 하면 Xsd.exe 어셈블리의 모든 형식에 대한 스키마를 생성합니다.

비고

다음 표에서는 Xsd.exe 수행하는 작업을 보여 줍니다.

테이블 확장

수술	설명
XDR에서 XSD로	XMLData-Reduced 스키마 파일에서 XML 스키마를 생성합니다. XDR은 초기 XML 기반 스키마 형식입니다.
XML에서 XSD로	XML 파일에서 XML 스키마를 생성합니다.
XSD에서 DataSet으로	XSD 스키마 파일에서 공용 언어 런타임 DataSet 클래스를 생성합니다. 생성된 클래스는 일반 XML 데이터에 대한 풍부한 개체 모델을 제공합니다.
XSD에서 클래스로	XSD 스키마 파일에서 런타임 클래스를 생성합니다. 생성된 클래스를 스키마 뒤에 있는 XML 코드를 읽고 쓰는 데 함께 System.Xml.Serialization.XmlSerializer 사용할 수 있습니다.
XSD에 대한 클래스	런타임 어셈블리 파일의 형식 또는 형식에서 XML 스키마를 생성합니다. 생성된 스키마는 XmlSerializer에서 사용하는 XML 형식을 정의합니다.

Xsd.exe W3C(World Wide Web 컨소시엄)에서 제안한 XSD(XML 스키마 정의) 언어를 따르는 XML 스키마만 조작할 수 있습니다. XML 스키마 정의 제안 또는 XML 표준에 대한 자세한 내용은 다음을 참조하세요 https://w3.org .

XML 파일을 사용하여 옵션 설정

스위치를 `/parameters` 사용하여 다양한 옵션을 설정하는 단일 XML 파일을 지정할 수 있습니다. 설정할 수 있는 옵션은 XSD.exe 도구를 사용하는 방법에 따라 달라집니다. 선택 항목으로는 스키마 생성, 코드 파일 생성 또는 기능이 포함된 코드 파일 생성 등이 있습니다 `DataSet` . 예를 들어, 스키마를 생성할 때는 `<assembly>` 요소를 실행 파일(.exe) 또는 형식 라이브러리(.dll) 파일의 이름으로 설정할 수 있지만, 코드 파일을 생성할 때는 설정할 수 없습니다. 다음 XML에서는 지정된 실행 파일과 `<generateSchemas>` 함께 요소를 사용하는 방법을 보여 있습니다.

XML

```
<!-- This is in a file named GenerateSchemas.xml. -->
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateSchemas>
   <assembly>ConsoleApplication1.exe</assembly>
</generateSchemas>
</xsd>
```

위의 XML이 GenerateSchemas.xml파일에 포함된 경우 명령 프롬프트에 다음을 입력하고 `/parameters` 키를 눌러 스위치를 사용합니다.

콘솔

```
xsd /p:GenerateSchemas.xml
```

반면 어셈블리에 있는 단일 형식에 대한 스키마를 생성하는 경우 다음 XML을 사용할 수 있습니다.

XML

```
<!-- This is in a file named GenerateSchemaFromType.xml. -->
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateSchemas>
   <type>IDItems</type>
</generateSchemas>
</xsd>
```

그러나 이전 코드를 사용하려면 명령 프롬프트에서 어셈블리의 이름도 제공해야 합니다. 명령 프롬프트에서 다음을 입력합니다(XML 파일 이름은 GenerateSchemaFromType.xml가정).

콘솔

```
xsd /p:GenerateSchemaFromType.xml ConsoleApplication1.exe
```

요소에 대해 `<generateSchemas>` 다음 옵션 중 하나만 지정해야 합니다.

테이블 확장

요소	설명
<조립>	스키마를 생성할 어셈블리를 지정합니다.
<형>	스키마를 생성할 어셈블리에 있는 형식을 지정합니다.
<Xml>	스키마를 생성할 XML 파일을 지정합니다.
<Xdr>	스키마를 생성할 XDR 파일을 지정합니다.

코드 파일을 생성하려면 요소를 사용합니다 `<generateClasses>` . 다음 예제에서는 코드 파일을 생성합니다. 생성된 파일의 프로그래밍 언어와 네임스페이스를 설정할 수 있는 두 가지 특성도 표시됩니다.

XML

```
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateClasses language='VB' namespace='Microsoft.Serialization.Examples'/>
</xsd>
```

```
<!-- You must supply an .xsd file when typing in the command line.-->
<!-- For example: xsd /p:genClasses mySchema.xsd -->
```

요소에 대해 설정할 수 있는 `<generateClasses>` 옵션은 다음과 같습니다.

테이블 확장

요소	설명
<요소>	코드를 생성할 .xsd 파일의 요소를 지정합니다.
<schemaImporterExtensions>	클래스에서 파생된 형식을 지정합니다 SchemaImporterExtension .
<스키마>	코드를 생성할 XML 스키마 파일을 지정합니다. 여러 요소를 사용하여 여러 `<schema>` XML 스키마 파일을 지정할 수 있습니다.

다음 표에서는 요소와 함께 `<generateClasses>` 사용할 수 있는 특성을 보여줍니다.

테이블 확장

특성	설명
언어	사용할 프로그래밍 언어를 지정합니다. (C#, 기본값), (Visual Basic), `CS` `VB` (JScript) 또는 `JS` (Visual J#) 중에서 `VJS` 선택합니다. 구현하는 클래스의 정규화된 이름을 지정할 수도 있습니다 CodeDomProvider.
네임스페이스	생성된 코드의 네임스페이스를 지정합니다. 네임스페이스는 CLR 표준을 준수해야 합니다(예: 공백 또는 백슬래시 문자 없음).
옵션	다음 값 중 하나: `none`, `properties` (공용 필드 대신 속성을 생성), `order` 또는 `enableDataBinding` (이전 XSD 파일 옵션 섹션에 있는 `/order` 및 `/enableDataBinding` 전환을 참조하세요).

`DataSet` 요소를 사용하여 `<generateDataSet>` 코드가 생성되는 방식을 제어할 수도 있습니다. 다음 XML은 생성된 코드가 구조체(예: `DataSet` 클래스)를 사용하여 DataTable 지정된 요소에 대한 Visual Basic 코드를 만들도록 지정합니다. 생성된 DataSet 구조는 LINQ 쿼리를 지원합니다.

XML

```
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
    <generateDataSet language='VB' namespace='Microsoft.Serialization.Examples'
enableLinqDataSet='true'>
    </generateDataSet>
</xsd>
```

요소에 대해 설정할 수 있는 `<generateDataSet>` 옵션은 다음과 같습니다.

테이블 확장

요소	설명
<스키마>	코드를 생성할 XML 스키마 파일을 지정합니다. 여러 요소를 사용하여 여러 `<schema>` XML 스키마 파일을 지정할 수 있습니다.

다음 표에서는 요소와 함께 `<generateDataSet>` 사용할 수 있는 특성을 보여줍니다.

테이블 확장

특성	설명
enableLinqDataSet (LINQ 데이터 세트 활성화)	LINQ to DataSet을 사용하여 생성된 DataSet을 쿼리할 수 있도록 지정합니다. 기본값은 false입니다.
언어	사용할 프로그래밍 언어를 지정합니다. (C#, 기본값), (Visual Basic), `CS` `VB` (JScript) 또는 `JS` (Visual J#) 중에서 `VJS` 선택합니다. 구현하는 클래스의 정규화된 이름을 지정할 수도 있습니다 CodeDomProvider.
네임스페이스	생성된 코드의 네임스페이스를 지정합니다. 네임스페이스는 CLR 표준을 준수해야 합니다(예: 공백 또는 백슬래시 문자 없음).

최상위 `<xsd>` 요소에 설정할 수 있는 특성이 있습니다. 이러한 옵션은 자식 요소(`<generateSchemas>` `<generateClasses>` 또는 `<generateDataSet>`)와 함께 사용할 수 있습니다. 다음 XML 코드는 "MyOutputDirectory"라는 출력 디렉터리에 "IDItems"라는 요소에 대한 코드를 생성합니다.

XML

```
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/' output='MyOutputDirectory'>
<generateClasses>
    <element>IDItems</element>
</generateClasses>
</xsd>
```

다음 표에서는 요소와 함께 `<xsd>` 사용할 수 있는 특성을 보여줍니다.

테이블 확장

특성	설명
출력	생성된 스키마 또는 코드 파일을 배치할 디렉터리의 이름입니다.
로고 없음	배너를 제거합니다. `true` 또는 `false`로 설정합니다.
도움	이 도구의 명령 구문 및 옵션을 표시합니다. `true` 또는 `false`로 설정합니다.

예시

다음 명령은 XML 스키마 `myFile.xdr` 를 생성하여 현재 디렉터리에 저장합니다.

콘솔

```
xsd myFile.xdr
```

다음 명령은 XML 스키마 `myFile.xml` 를 생성하고 지정된 디렉터리에 저장합니다.

콘솔

```
xsd myFile.xml /outputdir:myOutputDir
```

다음 명령은 C# 언어로 지정된 스키마에 해당하는 데이터 집합을 생성하고 현재 디렉터리에 저장합니다 `XSDSchemaFile.cs` .

콘솔

```
xsd /dataset /language:CS XSDSchemaFile.xsd
```

다음 명령은 어셈블리 `myAssembly.dll` 의 모든 형식에 대한 XML 스키마를 생성하고 현재 디렉터리에 있는 것처럼 `schema0.xsd` 저장합니다.

콘솔

```
xsd myAssembly.dll
```

참고하십시오

- DataSet
- System.Xml.Serialization.XmlSerializer
- 도구
- 개발자 명령줄 셸
- LINQ to DataSet 개요
- 형식화된 데이터 세트 쿼리
- LINQ(Language-Integrated 쿼리)(C#)
- LINQ(Language-Integrated 쿼리)(Visual Basic)
- Xsd.exe 소스 코드(참조 소스)

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 06.

BinaryFormatter 마이그레이션 가이드

2025. 06. 17.

> ⊗ **주의**
>
> **관련된 보안 위험**으로 인해 사용하지 BinaryFormatter 않도록 강력히 권장합니다. 기존 사용자는 .**에서 BinaryFormatter멀리 마이그레이션해야 합니다**.

.NET 9부터는 더 이상 런타임에 구현 BinaryFormatter 을 포함하지 않습니다. API는 여전히 존재하지만, 해당 구현은 프로젝트 유형에 관계없이 항상 예외를 발생시킵니다 PlatformNotSupportedException. 따라서 기존 이전 버전 호환성 플래그를 설정하는 것만으로는 BinaryFormatter을(를) 사용하는 데 충분하지 않습니다.

이 문제를 해결하는 두 가지 옵션이 있습니다.

- BinaryFormatter를 **더 이상 사용하지 않습니다**. 관련 보안 위험으로 인해 사용을 BinaryFormatter 중지하는 옵션을 조사하는 것이 좋습니다. 아래에 몇 가지 옵션이 나열되어 있습니다.
- **를 계속 사용하세요 BinaryFormatter**. .NET 9에서 BinaryFormatter을 계속 사용해야 하는 경우, 지원되지 않는 System.Runtime.Serialization 패키지에 의존해야 합니다. 이 패키지가 Throw 구현을 대체하는 Formatters NuGet 패키지입니다.

사용 BinaryFormatter시의 위험은 무엇인가요?

개체를 생성할 정보가 포함될 수 있는 입력을 허용하는 역직렬화기는, 이진 또는 텍스트 형식에서 보안 문제가 발생할 가능성이 높은 취약점을 가지고 있습니다. CWE-502 "신뢰할 수 없는 데이터의 역직렬화" 문제를 설명하는 CWE (일반적인 약점 열거형)가 있습니다. BinaryFormatter는 2002년 .NET Framework의 초기 릴리스에 포함된 역직렬 변환기입니다. 이 내용은 BinaryFormater 보안 가이드에서도 다룹니다.

알려진 사용 BinaryFormatter위험으로 인해 기능이 .NET Core 1.0에서 제외되었습니다. 그러나 더 안전한 항목으로 마이그레이션할 명확한 경로가 없었기 때문에, 고객의 수요로 인해 BinaryFormatter이 .NET Core 2.0에 포함되었습니다. 그 이후로 .NET 팀은 여러 프로젝트 형식에서 기본적으로 BinaryFormatter를 점진적으로 비활성화하며 제거하기 위해 노력해 오고 있습니다. 이전 버전과의 호환성이 필요한 경우 소비자는 플래그를 통해 특정 프로젝트 형식에서 옵트인할 수 있습니다.

결정에 대한 자세한 내용은 .NET 9에서 제거됨에 대한 공지BinaryFormatter 를 참조하세요.

마이그레이션 가이드에서 BinaryFormatter의 제거와 관련된 이슈가 해결되지 않은 경우에는, github.com/dotnet/runtime 에 문제를 제출하고 해당 문제가 BinaryFormatter의 제거와 관련된 것임을 명시하세요.

마이그레이션 항목

BinaryFormatter에서 벗어난다는 것은 일반적으로 다른 직렬 변환기를 선택하는 것을 의미합니다. 그러나 일반적으로 인코딩된 데이터의 생산자와 소비자를 모두 제어하는 경우에만 수행할 수 있습니다. 생산자를 제어하지 않는 경우 인코딩된 형식을 인스턴스화하지 않고 페이로드를 읽기 BinaryFormatter 위해 새 API 로 이동할 수도 있습니다.

두 옵션 모두 아래에서 살펴보세요.

직렬 변환기를 선택합니다

마이그레이션 `BinaryFormatter` 의 첫 번째 단계는 해당 위치에서 사용할 serializer를 선택하는 것입니다. 특정 요구 사항에 따라 .NET 팀은 4개의 서로 다른 serializer로 마이그레이션할 것을 권장합니다.

- System.Text.Json(JSON)으로 마이그레이션
- DataContractSerializer로 마이그레이션(XML)
- MessagePack으로 마이그레이션(이진 형식)
- protobuf-net으로 이진 형식으로 마이그레이션

읽기 BinaryFormatter (NRBF) 페이로드

많은 애플리케이션이 스토리지에 유지된 페이로드를 로드하고 역직렬화하며 모든 지속형 페이로드를 미리 변환할 수 있는 것은 아닙니다. 다른 시나리오에는 이러한 시스템을 독립적으로 마이그레이션해야 하는 경우 생성된 BinaryFormatter데이터를 수신하는 시스템 또는 서비스가 포함될 수 있습니다.

이러한 시나리오 및 기타 시나리오에서는 제공된 페이로드를 읽고 시간이 지남에 따라 새 형식으로 전환하기 위한 지원을 유지해야 합니다. 이러한 요구 사항을 충족하기 위해 범용 및 취약한 역직렬화를 수행하지 않고 `BinaryFormatter`로 생성한 NRBF 페이로드를 안전하게 읽을 수 있습니다.

Windows Forms 및 WPF 애플리케이션 마이그레이션

Windows Forms 및 WPF 애플리케이션에는 추가 변경이 필요할 수 있습니다. 추가 마이그레이션 지침 은 Windows Forms 앱, WPF 앱 및 WinForms/WPF 클립보드 및 끌어서 놓기 지침을 참조하세요.

관리 리소스 마이그레이션 (ResX)

가장 일반적인 리소스 종류(예: 문자열 및 아이콘)는 없이 BinaryFormatter작동합니다. 사용자 지정 형식의 경우, BinaryFormatter을 가져와서 호환성 스위치를 설정해야 합니다. 런타임 중 리소스 로드하기를 참조하세요.

호환성 패키지 사용

.NET 9로 업그레이드할 때 BinaryFormatter에서 마이그레이션할 수 없는 경우, 지원되지 않는 호환성 패키지를 사용할 수 있습니다. System.Runtime.Formatters NuGet 패키지에는 취약성과 위험을 포함하여 BinaryFormatter의 기능적 구현이 포함되어 있습니다.

지원되지 않으며 권장되지는 않지만 호환성 패키지 사용에 대한 가이드에는 패키지를 설치하고 기능을 사용하도록 설정하기 위한 세부 정보가 포함되어 있습니다.

> **⊗ 주의**
>
> **관련된 보안 위험**으로 인해 사용하지 BinaryFormatter 않도록 강력히 권장합니다. 기존 사용자는 .**에서 BinaryFormatter멀리 마이그레이션해야 합니다**.

직렬 변환기를 선택합니다

2025. 06. 22.

BinaryFormatter에 대해 대체되는 드롭은 없지만, .NET 형식을 직렬화하기 위해 권장되는 다양한 직렬 변환기가 있습니다. 새 직렬 변환기와 통합하려면 변경이 필요하며, 이는 어떤 직렬 변환기를 선택하든 그렇습니다. 이러한 마이그레이션에서는 선택한 직렬화 도구를 사용하여 기존 형식을 가능한 한 적은 변경으로 처리하도록 강제하는 것과 선택한 직렬화 도구에 맞게 형식을 리팩터링하여 관용적인 직렬화를 가능하게 하는 것 사이에서의 절충을 고려해야 합니다. 직렬 변환기가 선택되면 모범 사례를 위해 해당 설명서를 연구해야 합니다.

이진 직렬화 형식이 필수가 아니라면, JSON 또는 XML 직렬화 형식을 고려해 볼 수 있습니다. 이러한 직렬 변환기는 공식적으로 지원되며, .NET에 포함됩니다.

1. System.Text.Json을 사용하는 JSON
2. System.Runtime.Serialization.DataContractSerializer을(를) 사용하는 XML

시나리오에서 압축 이진 표현이 중요한 경우 다음 serialization 형식 및 오픈 소스 serializer를 사용하는 것이 좋습니다.

1. C#용 MessagePack 을 사용하는 MessagePack
2. protobuf-net 을 사용하는 프로토콜 버퍼

직렬화된 형식의 API 셰이프를 변경할 수 있는지 여부는 직렬화에 대한 방향과 접근 방식에 영향을 줍니다. 새 특성으로 형식에 주석을 달고, 새 생성자를 추가하고, 형식/멤버를 공용으로 만들고, 필드를 속성으로 변경하는 기능을 사용하는 이러한 직렬 변환기로의 마이그레이션은 더 간단할 수 있습니다. 이러한 기능이 없는 경우에는 사용자 지정 변환기 또는 확인자를 구현해야 최신 직렬 변환기를 사용할 수 있습니다.

테이블 확장

기능	BinaryFormatter	System.Text.Json	DataContractSerializer	C#용 MessagePack	protobuf-net
Serialization 형식	이진(NRBF)	JSON (자바스크립트 객체 표기법)	XML	이진 (MessagePack)	이진(프로토콜 버퍼)
압축 표현	✔	✘	✘	✔	✔
사람이 읽을 수 있는	✘	✔	✔	✘	✘
성능	✘	✔	✘	✔	✔
`[Serializable]` 특성 지원.	✔	✘	✔	✘	✘

기능	BinaryFormatter	System.Text.Json	DataContractSerializer	C#용 MessagePack	protobuf-net
공용 형식 직렬화	✔	✔	✔	✔	✔
공용이 아닌 형식 직렬화	✔	✔	✔	✔(확인자 필요)	✔
필드 직렬화	✔	✔(옵트인)	✔	✔(필수 특성)	✔(필수 특성)
공용이 아닌 필드 직렬화	✔	✔(확인자 필요)	✔	✔(확인자 필요)	✔(필수 특성)
속성 직렬화	✔*	✔	✔	✔(필수 특성)	✔(필수 특성)
읽기 전용 멤버 역직렬화	✔	✔(필수 특성)	✔	✔	✔(매개 변수가 없는 ctor 필수)
다형 형식 계층 구조	✔	✔(필수 특성)	✔	✔(필수 특성)	✔(필수 특성)
AOT 지원	✘	✔	✘	✔	✘(계획됨)

System.Text.Json을 사용하는 JSON

JSON(JavaScript Object Notation) 형식에 대한 보안, 고성능 및 낮은 메모리 할당을 강조하는 최신 직렬 변환기가 System.Text.Json 라이브러리입니다. JSON은 광범위한 플랫폼 간 지원을 제공하며, 사람이 읽을 수 있습니다. 이진 형식만큼 압축되지는 않는 텍스트 기반 형식은 크기를 크게 줄이기 위해 압축을 활용할 수 있습니다.

Serialization은(는) 공용이 아니며 읽기 전용인 멤버를 제외하며, 이는 특성 및 생성자를 통해 특별히 처리되지 않는 한 해당합니다. 또한 System.Text.Json은 형식이 JSON으로 변환되는 방식을 보다 세세하게 제어할 수 있는 사용자 지정 직렬화 및 역직렬화를 지원하며 그 반대의 경우도 마찬가지입니다. `[Serializable]` 특성을 System.Text.Json이 지원하지 않습니다.

System.Text.Json(JSON)으로 마이그레이션합니다.

DataContractSerializer를 사용하는 XML

WCF(Windows Communication Foundation) 메시지에서 보낸 데이터를 직렬화하고 역직렬화하는 데 .NET Framework 3.0에서 도입된 DataContractSerializer이(가) 사용됩니다.

DataContractSerializer는 특성 및 구현 `BinaryFormatter`을 적용 한다는 의미인 ISerializable 변환기입니다. 그러므로 이러한 직렬 변환기는 마이그레이션하는 데 최소한의 노력이 필요합니다. 하지만 알려진 형식을 미리 지정해야 합니다(그러나 기본 허용 목록에 대부분의 .NET 컬렉션 및 기본 형식이 있으므로 지정할 필요가 없습니다).

`DataContractSerializer`으(로)부터 마이그레이션할 때 BinaryFormatter이(가) 이러한 기능적 이점을 수반하지만, 다른 선택 사항만큼 성능이 좋거나 현대적이지는 않습니다.

DataContractSerializer(XML)로 마이그레이션합니다.

> ⚠ **경고**
>
> DataContractSerializer를 NetDataContractSerializer와 혼동하지 마세요.
> NetDataContractSerializer는 위험한 직렬 변환기로 식별됩니다.

MessagePack을 사용하는 이진

MessagePack 간단한 이진 serialization 형식이므로 JSON 및 XML에 비해 메시지 크기가 더 작습니다. 오픈 소스 C#용 MessagePack 라이브러리는 성능이 뛰어나며 더 작은 데이터 크기를 위한 기본 제공 초고속 LZ4 압축을 제공합니다. 데이터 형식이 `DataContractSerializer` 또는 라이브러리의 고유한 특성으로 주석을 추가할 때 가장 적합합니다. AOT 환경, 읽기 전용 형식 및 멤버, 공용이 아닌 형식 및 멤버를 지원하도록 구성할 수 있습니다.

MessagePack(이진)으로 마이그레이션합니다.

protobuf-net을 사용하는 이진

protobuf-net 라이브러리는 .NET용 계약 기반 직렬화 도구로, 이진 프로토콜 버퍼 직렬화 형식을 사용합니다. API는 일반적인 .NET 패턴을 따르며 `XmlSerializer` 및 `DataContractSerializer`와(과) 대체로 유사합니다. 이 인기 있는 라이브러리는 공용이 아닌 형식 및 필드를 처리할 수 있도록 풍부한 기능을 가지고 있지만 많은 시나리오에서 멤버에 특성을 적용해야 합니다.

protobuf-net(이진)으로 마이그레이션합니다.

System.Text.Json (JSON)으로 마이그레이션

2025. 06. 17.

이 System.Text.Json 라이브러리는 기본적으로 문자 그대로의 결정론적 행동을 강조하지 않으며, 호출자를 대신하여 추측하거나 해석하는 것을 금지합니다. 이 라이브러리는 보안과 성능을 위해 의도적으로 이렇게 설계되었습니다. `System.Text.Json`는 매우 구성 가능하며, 그 기능을 사용하여 직렬화된 형식에 필요한 변경을 최소화할 수 있지만, 기존 형식을 가능한 한 적은 변경으로 처리하는 것과 형식을 리팩터링하여 관용적이고 안전한 직렬화를 가능하게 하는 것 사이의 장단점을 고려하는 것이 중요합니다.

BinaryFormatter 에서 `System.Text.Json`로 마이그레이션할 때는 다음과 같은 동작 및 옵션을 적어 두는 것이 중요합니다.

- 기본적으로 필드는 직렬화되거나 역직렬화되지 않지만 serialization대해 주석을 수 있습니다. `JsonSerializerOptions.IncludeFields` 또는 직렬화되는 형식에 대한 모든 공용 필드를 포함하도록 `true` 신중하게 설정할 수 있습니다.

  ```
  C#

  JsonSerializerOptions options = new()
  {
      IncludeFields = true
  };
  ```

- 기본적으로 System.Text.Json**은 프라이빗 필드와 속성을 무시합니다**. 이 `[JsonInclude]` 특성을 사용하여 속성에서 public이 아닌 접근자를 사용하도록 설정할 수 있습니다. 프라이빗 필드를 포함하려면 몇 가지 사소한 추가 작업이 필요합니다.

- System.Text.Json **는 읽기 전용 필드** 또는 속성을 역직렬화할 수 없지만 `[JsonConstructor]`, 지정된 생성자를 사용하여 역직렬화 시 형식의 인스턴스를 만들어야 함을 나타내는 데 특성을 사용할 수 있습니다. 생성자는 읽기 전용 필드와 속성을 설정할 수 있습니다.

- 특정 형식의 기본 serialization 동작을 재정의하기 위해서는 사용자 지정 변환기를작성할 수 있습니다.

- 많은 컬렉션의 직렬화 및 역직렬화를 지원하지만 제한 사항이 있습니다. serialization 및 deserialization에 대해 어떤 형식 및 컬렉션이 지원되는지에 대한 더 자세한 정보는 지원되는 형식 설명서를 참조하세요.

- 특정 조건 하에서는 사용자 지정 제네릭 컬렉션의 직렬화 및 역직렬화를 지원합니다.

- 기본 제공 지원이 없는 다른 형식은 다음과 같습니다: `DataSet`, `DataTable`, `DBNull`, `TimeZoneInfo`, `Type`, `ValueTuple`. 그러나 이러한 형식을 지원하는 사용자 지정 변환기를 작

성할 수 있습니다.

- 지원합니다. 개방형 상속 계층은 지원되지 않으며 다형성으로 왕복하려면 알려진 모든 파생 형식에 대한 형식 판별자 식별자가 필요합니다.
- 속성에 `[JsonIgnore]` 특성이 있을 경우 serialization 시 JSON에서 해당 속성이 생략됩니다.
- `System.Text.Json`에서 참조를 보존하고 순환 참조를 처리하려면 `JsonSerializerOptions.ReferenceHandler`을 `ReferenceHandler.Preserve`로 설정합니다.
- Serialization 는 사용자 지정 계약으로 광범위하게 사용자 지정할 수 있으며, 직렬화된 형식에 대한 변경 내용을 최소화하면서 많은 시나리오의 차단을 해제할 수 있습니다.

DataContractSerializer(XML)로 마이그레이션합니다.

아티클 • 2025. 01. 22.

.NET 기본 클래스 라이브러리는 XmlSerializer 및 DataContractSerializer라는 두 개의 XML serializer를 제공합니다. 이 두 항목 사이에는 몇 가지 미묘한 차이점이 있지만 마이그레이션을 위해 이 섹션에서는 `DataContractSerializer`에만 중점을 둡니다. 이유는 무엇입니까? **는** `BinaryFormatter`에서 사용된 직렬화 프로그래밍 모델을 완전히 지원하므로 이미 `[Serializable]` 로 표시되거나 `ISerializable` 구현된 모든 형식은 `DataContractSerializer`으로 직렬화 할 수 있습니다. 캐치는 어디에 있습니까? 알려진 형식을 앞에 지정해야 합니다. 당신은 그들을 알고 `Type`에 대한 을 얻을 수 있어야 합니다.

가장 인기 있는 컬렉션 또는 기본 형식을 `string` `DateTime` 지정하거나(직렬화에는 자체 기본 허용 목록이 있음) 필수는 아니지만 다음과 같은 `DateTimeOffset`예외가 있습니다. 지원되는 형식에 대한 자세한 내용은 데이터 계약 직렬화 에서 지원하는 형식을 참조하십시오.

부분 신뢰 는 .NET(Core)로 이식되지 않은 .NET Framework 기능입니다. 코드가 .NET Framework에서 실행되고 이 기능을 사용하는 경우 이러한 시나리오에 적용될 수 있는 제한 사항에 대해 읽어봅니다.

단계별 마이그레이션

1. `BinaryFormatter`의 모든 사용 현황을 찾습니다.
2. serialization 코드 경로에 테스트가 적용되어 있는지 확인하고, 변경 사항을 확인하며 버그 발생을 방지할 수 있도록 하세요.
3. 마치 `DataContractSerializer` 이 .NET Core 라이브러리의 일부인 것처럼 패키지를 설치할 필요가 없습니다.
4. `BinaryFormatter`을 사용하여 직렬화되는 모든 형식을 찾습니다. 수정할 필요는 없지만 이 `knownTypes` 생성자의 인수 `DataContractSerializer` 를 통해 나열해야 할 수도 있습니다.
5. `BinaryFormatter` 의 사용량을 `DataContractSerializer`(으)로 교체합니다.

C#

```
DataContractSerializer serializer = new(
    type: input.GetType(),
    knownTypes: new Type[]
    {
        typeof(MyType1),
```

```
        typeof(MyType2)
    });
```

MessagePack(이진)으로 마이그레이션합니다.

2025. 06. 17.

MessagePack 간단한 이진 serialization 형식이므로 JSON 및 XML에 비해 메시지 크기가 더 작습니다. 오픈 소스 C#용 MessagePack 라이브러리는 성능이 뛰어나며 더 작은 데이터 크기를 위한 기본 제공 초고속 LZ4 압축을 제공합니다. 데이터 형식이 `DataContractSerializer` 또는 라이브러리의 고유한 특성으로 주석을 추가할 때 가장 적합합니다. AOT 환경, 읽기 전용 형식 및 멤버, 공용이 아닌 형식 및 멤버를 지원하도록 구성할 수 있습니다.

C#용 MessagePack의 일부 동작 및 기능은 특히 직렬화된 형식의 API를 변경할 수 없거나 최소화할 필요가 없는 경우 마이그레이션 BinaryFormatter하는 동안 주목할 만합니다.

- 기본적으로 퍼블릭 형식만 직렬화 할 수 있습니다. 프라이빗 및 내부 구조체 및 클래스는 `StandardResolverAllowPrivate.Options` 이 `MessagePackSerializer.Serialize` 의 인수와 `MessagePackSerializer.Deserialize` 메서드로 제공된 경우에만 직렬화 할 수 있습니다.

- MessagePack을 사용하려면 직렬화할 수 있는 각 형식에 특성에 주석을 `[MessagePackObject]` 추가해야 합니다. 이 ContractlessStandardResolver를 사용하여 이를 방지할 수 있지만 나중에 버전 관리와 관련된 문제가 발생할 수 있습니다.

- 직렬화할 수 있는 모든 비정적 필드와 속성은 특성으로 이 `[Key]` 주석을 추가해야 합니다. 특성으로 형식에 주석을 `[MessagePackObject(keyAsPropertyName: true)]` 달면 멤버에 명시적 주석이 필요하지 않습니다. 이러한 경우 특정 공용 멤버를 무시하려면 특성을 사용합니다 `[IgnoreMember]` .

- 프라이빗 멤버를 직렬화하려면 StandardResolverAllowPrivate를 사용합니다.

- `System.Runtime.Serialization` 주석은 MessagePack 주석 대신 사용할 수 있습니다: `[DataContract]` 대신 `[MessagePackObject]`, `[DataMember]` 대신 `[Key]`, 및 `[IgnoreDataMember]` 대신 `[IgnoreMember]` 사용할 수 있습니다. 이러한 주석은 직렬화 가능한 형식을 정의하는 라이브러리의 MessagePack에 대한 종속성을 방지하려는 경우에 유용할 수 있습니다.

- 읽기 전용/변경할 수 없는 형식 및 멤버를 지원합니다. serializer는 가장 일치하는 인수 목록과 함께 공용 생성자를 사용하려고 합니다. 특성을 사용하여 `[SerializationConstructor]` 생성자를 명시적 방식으로 지정할 수 있습니다.

- Serialization 임의 형식은 작성하기 간단한 사용자 지정 포맷터를 통해 지원됩니다. 이렇게 하면 특성 및 특정 생성자 또는 멤버 패턴에 대한 모든 요구 사항이 제거됩니다.

- serializer는 .NET 기본 클래스 라이브러리에서 제공하는 가장 자주 사용되는 기본 제공 형식 및 컬렉션을 지원합니다. 공식 문서에서 전체 목록을 찾을 수 있습니다. 사용자 지정을 허용하는 확장 지점 이 있습니다.

> **⚠ 경고**
>
> MessagePack에는 형식 제한 없이 데이터를 역직렬화할 수 있는 API가 있습니다. 각 **MessagePack 보안 정보** 별로 이러한 API는 피해야 합니다.

> **⚠ 경고**
>
> 일부 MessagePack API에는 *변경 가능한 통계*를 통해 사용자 지정할 수 있는 동작이 있습니다. 즉, 동일한 프로세스, AssemblyLoadContext 또는 AppDomain의 다른 코드에 따라 코드가 성공하거나 실패할 수 있습니다. 이 MessagePackAnalyzer 패키지를 참조하고 변경 가능한 동작으로 API의 모든 사용을 호출하는 MsgPack001 및 MsgPack002 분석기를 사용하도록 설정하여 코드를 복원력을 유지할 수 있습니다.

protobuf-net(이진)으로 마이그레이션합니다.

2025. 06. 17.

protobuf-net 라이브러리는 .NET용 계약 기반 직렬화 도구로, 이진 프로토콜 버퍼 직렬화 형식을 사용합니다. API는 일반적인 .NET 패턴을 따르며 `XmlSerializer` 및 `DataContractSerializer`와(과) 대체로 유사합니다.

protobuf-net의 일부 동작 및 기능은 마이그레이션 중에 주목할 만한 것이며, 많은 시나리오에서는 BinaryFormatter멤버에 특성을 적용해야 합니다.

- 기본적으로 public 형식과 public이 아닌 형식은 모두 직렬화할 수 있으며 serializer에는 매개 변수가 없는 생성자가 예상됩니다.
- protobuf-net에는 특성으로 `[ProtoContract]` 주석을 달 수 있는 각 직렬화 가능한 형식이 필요합니다. 이 특성은 필요에 따라 `SkipConstructor = true` 속성을 지정 하여 특정 생성자의 필요성을 *제거할* 수 있습니다.
- 직렬화할 수 있는 모든 비정적 필드와 속성은 특성으로 `[ProtoMember(int identifier)]` 주석을 추가해야 합니다. 멤버 이름은 데이터에 인코딩되지 않습니다. 대신, 사용자는 해당 형식 내에서 고유해야 하는 각 멤버를 식별하기 위해 양의 정수를 선택해야 합니다.
- 상속 은 알려진 하위 형식이 있는 각 형식의 특성을 통해 `[ProtoInclude(...)]` 명시적으로 선언되어야 합니다.
- 읽기 전용 필드는 기본적으로 지원됩니다.
- 또는 특성이 없는 일부 튜플과 유사한 형식은 *생성자 패턴에서* 인식됩니다. 선언된 모든 공용 멤버와 일치하는 매개 변수가 있는 생성자가 있는 형식은 튜플로 해석되고 매개 변수 순서는 해당 멤버의 식별자를 유추하는 데 사용됩니다.
- 이 protobuf-net.BuildTools 디자인 타임 패키지를 사용하는 것이 좋습니다. 일반적인 오류에 대한 컴파일 시간 경고를 제공합니다.

단계별 마이그레이션

1. `BinaryFormatter`의 모든 사용 현황을 찾습니다.
2. serialization 코드 경로에 테스트가 적용되어 있는지 확인하고, 변경 사항을 확인하며 버그 발생을 방지할 수 있도록 하세요.
3. 패키지를 설치 `protobuf-net` 합니다(선택 사항 `protobuf-net.BuildTools`).
4. `BinaryFormatter`을 사용하여 직렬화되는 모든 형식을 찾습니다.
5. 수정할 수 있는 형식의 경우:

    - 인터페이스로 `[ProtoContract]` 표시되거나 구현되는 모든 형식의 특성에 `[Serializable]` 주석을 `ISerializable` 추가합니다. 이러한 형식이 다른 직렬 변환기

를 `DataContractSerializer`사용할 수 있는 다른 앱(예: 라이브러리 작성 중)에 노출되지 않는 경우 주석 및 `[Serializable]` 주석을 `ISerializable` 제거할 수 있습니다.
    - 파생 형식의 경우 기본 형식에 적용 `[ProtoInclude(...)]` 합니다(아래 예제 참조).
    - 매개 변수를 허용하는 모든 생성자를 선언하는 모든 형식에 대해 매개 변수가 없는 생성자를 추가하거나 특성에 `SkipConstructor = true` 지정 `[ProtoContract]` 합니다. protobuf-net 요구 사항을 설명하는 주석을 남겨 둡니다(실수로 제거되는 사람은 아무도 없습니다).
    - 직렬화할 모든 멤버(필드 및 속성)를 표시합니다 `[ProtoMember(int identifier)]`. 모든 식별자는 단일 형식 내에서 고유해야 하지만 상속을 사용하는 경우 하위 형식에서 동일한 숫자를 다시 사용할 수 있습니다.
6. 수정할 수 없는 형식의 경우:

    - .NET 자체에서 제공하는 형식의 경우 API를 사용하여 `ProtoBuf.Meta.RuntimeTypeModel.Default.CanSerialize(Type type)` protobuf-net에서 기본적으로 지원되는지 확인할 수 있습니다.
    - 전용 DTO(데이터 전송 개체)를 만들고 그에 따라 매핑할 수 있습니다(이에 대해 암시적 캐스트 연산자 사용 가능).
    - API를 `RuntimeTypeModel` 사용하여 특성이 허용하는 모든 것을 정의합니다.

7. `BinaryFormatter` 의 사용량을 `ProtoBuf.Serializer`(으)로 교체합니다.

diff

```
-[Serializable]
+[ProtoContract]
+[ProtoInclude(2, typeof(Point2D))]
public class Point1D
{
+   [ProtoMember(1)]
    public int X { get; set; }
}

-[Serializable]
+[ProtoContract]
public class Point2D : Point1D
{
+   [ProtoMember(2)]
    public int Y { get; set; }
}
```

읽기 BinaryFormatter (NRBF) 페이로드

아티클 • 2025. 02. 06.

BinaryFormatter은 serialization을 위해 .NET Remoting: 이진 형식를 사용했습니다. NRBF 또는 MS-NRBF의 약어로 이 형식이 알려져 있습니다. 이전에 BinaryFormatter이(가) 필요한 이러한 페이로드를 읽으면서 스토리지에 유지되는 페이로드를 처리하는 것이 BinaryFormatter(으)로부터의 마이그레이션과 관련된 일반적인 과제입니다. 일부 시스템은 이러한 페이로드를 읽는 기능을 유지함으로써 BinaryFormatter 자체에 대한 참조를 피하면서 새 직렬 변환기로 점진적으로 마이그레이션해야 합니다.

.NET 9의 일부로 페이로드의 역직렬화를 수행하지 않고 NRBF 페이로드를 디코딩하는 새로운 *NrbfDecoder* 클래스가 도입되었습니다. 이 API는 신뢰할 수 있거나 신뢰할 수 없는 페이로드를 BinaryFormatter 역직렬화에 수반되는 위험 없이 디코딩하는 데 안전하게 사용할 수 있습니다. 하지만 NrbfDecoder는 데이터를 애플리케이션이 추가로 처리할 수 있는 구조로 디코딩하기만 하면 됩니다. 데이터를 적절한 인스턴스에 안전하게 로드하기 위해 NrbfDecoder를 사용할 때는 주의해야 합니다.

> **⊗ 주의**
>
> NrbfDecoder*NRBF 판독기의* 구현이지만 해당 동작은 BinaryFormatter구현을 엄격하게 따르지 않습니다. 따라서 NrbfDecoder 출력을 사용하여 BinaryFormatter 호출이 안전한지 여부를 결정해서는 안 됩니다.

NrbfDecoder이(가) JSON/XML 판독기를 역직렬 변환기 없이 사용하는 것과 같다고 생각할 수 있습니다.

NrbfDecoder

NrbfDecoder는 새 System.Formats.Nrbf NuGet 패키지의 일부입니다. .NET 9뿐만 아니라 .NET Framework 및 .NET Standard 2.0과 같은 이전 모니커도 대상으로 합니다. 이러한 다중 대상 지정을 사용하면 지원되는 .NET 버전을 사용하는 모든 사용자가 BinaryFormatter에서 마이그레이션할 수 있습니다. NrbfDecoder는 BinaryFormatter(기본값)을(를) 사용하여 FormatterTypeStyle.TypesAlways(으)로 직렬화된 페이로드를 읽을 수 있습니다.

NrbfDecoder는 모든 입력을 신뢰할 수 없는 것으로 처리하도록 설계되었습니다. 그러므로 다음과 같은 원칙이 있습니다.

- (원격 코드 실행과 같은 위험을 방지하기 위해) 어떤 종류의 형식도 로드하지 않습니다.

- (언바운드 재귀, 스택 오버플로 및 서비스 거부를 방지하기 위해) 어떤 종류의 재귀도 없습니다.
- 페이로드가 너무 작아서 약속된 데이터를 포함할 수 없는 경우, (메모리 부족 및 서비스 거부를 방지하기 위해) 페이로드에 제공된 크기에 따라 버퍼 사전 할당이 수행되지 않습니다.
- (페이로드를 만든 잠재적인 공격자와 동일한 양의 작업을 수행하려면) 입력의 모든 부분을 한 번만 디코딩합니다.
- (해시 코드 충돌 수에 따라 크기가 달라지는 배열에서 지원되는 사전의 메모리 부족을 방지하기 위해) 충돌 방지 임의 해시를 사용하여 다른 레코드에서 참조하는 레코드를 저장합니다.
- 기본 형식만 암시적 방식으로 인스턴스화할 수 있습니다. 요청에 따라 배열을 인스턴스화할 수 있습니다. 다른 형식은 인스턴스화되지 않습니다.

> **⊗ 주의**
>
> NrbfDecoder 을 사용할 때, 이러한 안전 장치를 무력화할 수 있으므로, 범용 코드에 해당 기능을 다시 도입하지 않는 것이 중요합니다.

닫힌 형식 집합 역직렬화

직렬화된 형식 목록이 알려진 닫힌 집합인 경우에만 NrbfDecoder가 유용합니다. 다른 방법으로 설정하려면 읽을 내용을 미리 알고 있어야 하며, 이는 이러한 형식의 인스턴스를 만들고 페이로드에서 읽은 데이터로 채워야 하기 때문입니다. 반대 예제 두 가지를 고려합니다.

- 라이브러리 자체에서 유지할 수 있는 `[Serializable]`에서의 모든 형식은 `sealed` 입니다. 그러므로 페이로드에는 알려진 형식만 포함될 수 있으며, 사용자가 만들 수 있는 사용자 지정 형식은 없습니다. 또한 이러한 형식을 페이로드에서 읽은 정보에 따라 다시 만들 수 있으며, 이는 형식은 공용 생성자를 제공하기 때문입니다.
- SettingsPropertyValue 형식은 구성 파일에 저장된 개체를 직렬화 및 역직렬화하는 데 PropertyValue을(를) 내부적으로 사용할 수 있는 형식 `object`의 속성 BinaryFormatter을(를) 노출합니다. 문자 그대로 모든 항목을 저장하는 데 사용할 수 있으며, 정수, 사용자 지정 형식, 사전 등이 포함됩니다. 그렇기 때문에, **호환성이 손상되는 변경 내용을 API에 도입하지 않고는 이 라이브러리를 마이그레이션할 수 없습니다**.

NRBF 페이로드 식별

NrbfDecoder는 NRBF 헤더로 지정된 스트림 또는 버퍼가 시작하는지 여부를 확인할 수 있는 두 가지 StartsWithPayloadHeader 메서드를 제공합니다. BinaryFormatter로 (으)로

지속된 페이로드를 마이그레이션할 때는 이러한 메서드를 사용하는 것이 권장됩니다.

- 스토리지에서 읽은 페이로드가 을(를) 포함하는 NrbfDecoder.StartsWithPayloadHeader 페이로드인지 여부를 확인합니다.
- 그렇다면 이를 NrbfDecoder.Decode(으)로 읽고, 새 직렬 변환기를 사용하여 다시 직렬화하고, 스토리지의 데이터를 덮어씁니다.
- 그렇지 않다면, 데이터를 역직렬화하기 위해 새 직렬 변환기를 사용합니다.

C#

```
internal static T LoadFromFile<T>(string path)
{
    bool update = false;
    T value;

    using (FileStream stream = File.OpenRead(path))
    {
        if (NrbfDecoder.StartsWithPayloadHeader(stream))
        {
            value = LoadLegacyValue<T>(stream);
            update = true;
        }
        else
        {
            value = LoadNewValue<T>(stream);
        }
    }

    if (update)
    {
        File.WriteAllBytes(path, NewSerializer(value));
    }

    return value;
}
```

안전하게 NRBF 페이로드 읽기

NRBF 페이로드는 직렬화된 개체와 해당 메타데이터를 나타내는 직렬화 레코드로 구성됩니다. Decode 메서드를 호출해야 전체 페이로드를 읽고 루트 개체를 얻을 수 있습니다.

Decode 메서드는 SerializationRecord 인스턴스를 반환합니다. SerializationRecord serialization 레코드를 나타내는 추상 클래스이며 Id, RecordType및 TypeName세 가지 자체 설명 속성을 제공합니다.

> ⓘ **참고**

공격자는 주기를 사용하여 페이로드를 만들 수 있습니다(예: 클래스 또는 자체에 대한 참조가 있는 개체 배열). Id IEquatable<T> 구현하는 SerializationRecordId 인스턴스를 반환하며, 무엇보다도 디코딩된 레코드에서 주기를 검색하는 데 사용할 수 있습니다.

SerializationRecord 페이로드에서 읽은 형식 이름(및 TypeName 속성을 통해 노출됨)과 지정된 형식을 비교하는 메서드 TypeNameMatches하나를 노출합니다. 사용자는 형식 전달 및 어셈블리 버전 관리와 관련된 걱정을 할 필요가 없으며, 이는 이 메서드는 어셈블리 이름을 무시하기 때문입니다. 또한 (이러한 정보를 가져오려면 형식 로드가 필요하기 때문에) 멤버 이름 또는 해당 형식을 고려하지 않습니다.

C#

```
using System.Formats.Nrbf;

static Animal Pseudocode(Stream payload)
{
    SerializationRecord record = NrbfDecoder.Read(payload);
    if (record.TypeNameMatches(typeof(Cat)) && record is ClassRecord
catRecord)
    {
        return new Cat()
        {
            Name = catRecord.GetString("Name"),
            WorshippersCount = catRecord.GetInt32("WorshippersCount")
        };
    }
    else if (record.TypeNameMatches(typeof(Dog)) && record is ClassRecord
dogRecord)
    {
        return new Dog()
        {
            Name = dogRecord.GetString("Name"),
            FriendsCount = dogRecord.GetInt32("FriendsCount")
        };
    }
    else
    {
        throw new Exception($"Unexpected record:
`{record.TypeName.AssemblyQualifiedName}`.");
    }
}
```

서로 다른 serialization 레코드 형식이 12가지 이상있습니다. 이 라이브러리는 다음 중 몇 가지만 학습하면 되며, 라이브러리가 추상화 집합을 제공하기 때문입니다.

- PrimitiveTypeRecord<T>: NRBF에서 기본적으로 지원되는 모든 기본 형식을 설명합니다(`string`, `bool`, `byte`, `sbyte`, `char`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`,

`float`, `double`, `decimal`, `TimeSpan`및 `DateTime`).
    - `Value` 속성을 통해 값을 노출합니다.
    - PrimitiveTypeRecord<T>은(는) PrimitiveTypeRecord 속성을 노출하는 제네릭이 아닌 Value 항목에서 파생됩니다. 하지만 기본 클래스에서는 (값 형식에 boxing을 도입하여) 값이 `object`(으)로 반환됩니다.
- ClassRecord: 앞서 언급한 기본 형식 외에 모든 `class` 및 `struct` 형식을 설명합니다.
- ArrayRecord: 다차원 배열 및 가변 배열을 포함하는 모든 배열 레코드를 설명합니다.
- SZArrayRecord<T>: `T`이(가) 기본 형식이거나 SerializationRecord일 수 있는 인덱싱되지 않은 1차원 배열 레코드를 설명합니다.

C#

```
SerializationRecord rootObject = NrbfDecoder.Decode(payload); // payload is
a Stream

if (rootObject is PrimitiveTypeRecord primitiveRecord)
{
    Console.WriteLine($"It was a primitive value:
'{primitiveRecord.Value}'");
}
else if (rootObject is ClassRecord classRecord)
{
    Console.WriteLine($"It was a class record of
'{classRecord.TypeName.AssemblyQualifiedName}' type name.");
}
else if (rootObject is SZArrayRecord<byte> arrayOfBytes)
{
    Console.WriteLine($"It was an array of `{arrayOfBytes.Length}`-many
bytes.");
}
```

Decode 외에도, NrbfDecoder는 DecodeClassRecord을(를) 반환(또는 throw)하는 ClassRecord 메서드를 노출합니다.

ClassRecord

SerializationRecord은(는) ClassRecord(으)로부터 파생되는 가장 중요한 형식으로, 배열 및 고유하게 지원되는 기본 형식 외의 **모든 `class` 및 `struct` 인스턴스를 나타내는 형식**입니다. 모든 멤버 이름 및 값을 읽을 수 있습니다. *멤버*가 무엇인지 이해하려면 BinaryFormatter 기능 참조를 참조하세요.

제공하는 API:

- MemberNames 속성은 직렬화된 멤버의 이름을 가져옵니다.
- HasMember 메서드는 페이로드에 지정된 이름의 멤버가 있는지 여부를 확인합니다. 이는 지정된 멤버의 이름을 바꿀 수 있는 버전 관리 시나리오를 처리하도록 설계

되었습니다.

- 다음은 제공된 멤버 이름의 기본값을 검색하기 위한 전용 메서드 집합입니다. GetString, GetBoolean, GetByte, GetSByte, GetChar, GetInt16, GetUInt16, GetInt32, GetUInt32, GetInt64, GetUInt64, GetSingle, GetDouble, GetDecimal, GetTimeSpan 및 GetDateTime.
- GetClassRecord [ClassRecord]의 인스턴스를 검색합니다. 주기의 경우 동일한 Id있는 현재 [ClassRecord]의 동일한 인스턴스입니다.
- GetArrayRecord [ArrayRecord]의 인스턴스를 검색합니다.
- GetSerializationRecord은 serialization 레코드를 검색하고, GetRawValue은 serialization 레코드나 원시 기본 값을 검색합니다.

다음과 같은 코드 조각은 ClassRecord을(를) 작동 중으로 표시합니다.

C#

```
[Serializable]
public class Sample
{
    public int Integer;
    public string? Text;
    public byte[]? ArrayOfBytes;
    public Sample? ClassInstance;
}

ClassRecord rootRecord = NrbfDecoder.DecodeClassRecord(payload);
Sample output = new()
{
    // using the dedicated methods to read primitive values
    Integer = rootRecord.GetInt32(nameof(Sample.Integer)),
    Text = rootRecord.GetString(nameof(Sample.Text)),
    // using dedicated method to read an array of bytes
    ArrayOfBytes =
((SZArrayRecord<byte>)rootRecord.GetArrayRecord(nameof(Sample.ArrayOfBytes))
).GetArray(),
};

// using GetClassRecord to read a class record
ClassRecord? referenced =
rootRecord.GetClassRecord(nameof(Sample.ClassInstance));
if (referenced is not null)
{
    if (referenced.Id.Equals(rootRecord.Id))
    {
        throw new Exception("Unexpected cycle detected!");
    }

    output.ClassInstance = new()
    {
        Text = referenced.GetString(nameof(Sample.Text))
```

```
    };
}
```

ArrayRecord

ArrayRecord은(는) NRBF 배열 레코드의 핵심 동작을 정의하며 파생 클래스에 대한 기반을 제공합니다. 이는 다음과 같은 두 가지 속성을 제공합니다.

- Rank- 배열의 순위를 가져옵니다.
- Lengths는 모든 차원의 요소 수를 나타내는 정수 버퍼를 획득합니다. GetArray호출하기 전에 제공된 배열 레코드 **총 길이를** 확인하는 것이 좋습니다.

또한 다음의 한 가지 메서드를 제공합니다. GetArray. 처음으로 사용되는 경우 배열을 할당하고 직렬화된 레코드에 제공된 데이터(고유하게 지원되는 `string` 또는 `int` 등의 기본 형식의 경우) 또는 직렬화된 레코드 자체(복합 형식 배열의 경우)로 채웁니다.

GetArray은(는) 예상된 배열의 형식을 지정하는 필수 인수를 필요로 합니다. 예를 들어, 레코드가 정수의 2D 배열이어야 하는 경우 `expectedArrayType`와(과) 같이 `typeof(int[,])`이(가) 제공되어야 하며, 반환된 배열 또한 `int[,]`와(과) 같습니다.

C#

```
ArrayRecord arrayRecord = (ArrayRecord)NrbfDecoder.Decode(stream);
if (arrayRecord.Rank != 2 || arrayRecord.Lengths[0] * arrayRecord.Lengths[1]
> 10_000)
{
    throw new Exception("The array had unexpected rank or length!");
}
int[,] array2d = (int[,])arrayRecord.GetArray(typeof(int[,]));
```

형식이 일치하지 않는 경우(예를 들어, 공격자가 두 개의 십억 문자열 배열로 페이로드를 제공하는 경우) 메서드가 InvalidOperationException을(를) throw합니다.

> **⊗ 주의**
>
> 아쉽게도 NRBF 형식을 사용하면 공격자가 많은 수의 null 배열 항목을 쉽게 압축할 수 있습니다. 따라서 GetArray호출하기 전에 항상 배열의 총 길이를 확인하는 것이 좋습니다. 또한 GetArray은 선택적 `allowNulls` 부울 인수를 허용하며, 해당 값을 `false`로 설정하면 null에 대해 예외를 발생시킵니다.

NrbfDecoder는 복합 형식 배열의 경우 SerializationRecord 배열을 반환하며, 이는 사용자 지정 형식을 로드하거나 인스턴스화하지 않기 때문입니다.

C#

```
[Serializable]
public class ComplexType3D
{
    public int I, J, K;
}

ArrayRecord arrayRecord = (ArrayRecord)NrbfDecoder.Decode(payload);
if (arrayRecord.Rank != 1 || arrayRecord.Lengths[0] > 10_000)
{
    throw new Exception("The array had unexpected rank or length!");
}

SerializationRecord[] records =
(SerializationRecord[])arrayRecord.GetArray(expectedArrayType:
typeof(ComplexType3D[]), allowNulls: false);
ComplexType3D[] output = records.OfType<ClassRecord>().Select(classRecord =>
new ComplexType3D()
{
    I = classRecord.GetInt32(nameof(ComplexType3D.I)),
    J = classRecord.GetInt32(nameof(ComplexType3D.J)),
    K = classRecord.GetInt32(nameof(ComplexType3D.K)),
}).ToArray();
```

NRBF 페이로드 내에서 인덱싱되지 않은 배열을 .NET Framework가 지원했지만 .NET(Core)으로 이 지원이 이식되지 않았습니다. 그러므로 NrbfDecoder는 인덱싱되지 않은 배열의 디코딩을 지원하지 않습니다.

SZArrayRecord

`SZArrayRecord<T>`은(는) NRBF 단일 차원, 인덱싱되지 않은 배열 레코드의 핵심 동작을 정의하고 파생 클래스에 대한 기반을 제공합니다. `T`은(는) 기본적으로 지원되는 기본 형식 또는 SerializationRecord 중 하나일 수 있습니다.

이는 Length이(가) 반환하는 GetArray 속성 및 `T[]` 오버로드를 제공합니다.

C#

```
[Serializable]
public class PrimitiveArrayFields
{
    public byte[]? Bytes;
    public uint[]? UnsignedIntegers;
}

ClassRecord rootRecord = NrbfDecoder.DecodeClassRecord(payload);
SZArrayRecord<byte> bytes =
(SZArrayRecord<byte>)rootRecord.GetArrayRecord(nameof(PrimitiveArrayFields.B
```

```
ytes));
SZArrayRecord<uint> uints =
(SZArrayRecord<uint>)rootRecord.GetArrayRecord(nameof(PrimitiveArrayFields.U
nsignedIntegers));
if (bytes.Length > 100_000 || uints.Length > 100_000)
{
    throw new Exception("The array exceeded our limit");
}

PrimitiveArrayFields output = new()
{
    Bytes = bytes.GetArray(),
    UnsignedIntegers = uints.GetArray()
};
```

BinaryFormatter 호환성 패키지

아티클 • 2024. 08. 08.

> **⊗ 주의**
>
> 호환성 패키지는 지원되지 않으며 안전하지 않습니다. 이 패키지에 대한 종속성을 사용하고 대신 **.에서 BinaryFormatter**마이그레이션하지 않도록 하는 것이 좋습니다.

마이그레이션할 수 없는 `BinaryFormatter` .NET 9+ 사용자는 지원 되지 않는 System.RuntimeSerialization을 설치할 수 있습니다. NuGet 패키지를 포맷하고 AppContext 스위치를 `System.Runtime.Serialization.EnableUnsafeBinaryFormatterSerialization` .로 `true`설정합니다.

> **ⓘ 참고**
>
> 이 패키지는 .의 BinaryFormatter형식 ID를 변경하지 않습니다. 기존 라이브러리를 사용하기 위해 이 패키지에 의존하도록 업데이트할 필요가 없습니다. 이 패키지에 의존해야 하는 유일한 위치는 애플리케이션 프로젝트입니다.

패키지는 해당 취약성 및 위험을 포함하여 기본 제공 구현을 작동하는 구현 BinaryFormatter 으로 바꿉니다. 이는 .NET 9 이상으로 마이그레이션하는 동안 아직 사용량을 대체하지 않은 상태에서 기다릴 수 없는 경우 중지 간격을 의미합니다 BinaryFormatter . 에서 마이그레이션BinaryFormatter하는 것이 좋습니다.

XML

```
<PropertyGroup>
  <TargetFramework>net9.0</TargetFramework>

<EnableUnsafeBinaryFormatterSerialization>true</EnableUnsafeBinaryFormatterS
erialization>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="System.Runtime.Serialization.Formatters"
Version="9.0.0-*" />
</ItemGroup>
```

> **⊗ 주의**

호환성 패키지는 지원되지 않으며 안전하지 않습니다. 이 패키지에 대한 종속성을 사용하고 대신 **.에서 BinaryFormatter**마이그레이션하지 않도록 하는 것이 좋습니다.

BinaryFormatter 기능 참조

아티클 • 2025. 01. 22.

2002년 .NET Framework의 초기 릴리스와 함께 BinaryFormatter이(가) 처음 도입되었습니다. BinaryFormatter 사용법을 바꾸는 방법을 이해하려면 BinaryFormatter의 작동 방식을 아는 것이 도움이 됩니다.

BinaryFormatter은(는) `[Serializable]`(으)로 주석이 달리거나 ISerializable 인터페이스를 구현한 모든 유형의 인스턴스를 직렬화할 수 있습니다.

멤버 이름

가장 일반적인 시나리오에서는 `[Serializable]`(으)로 형식에 주석이 달리고 직렬 변환기는 리플렉션을 사용하여 주석이 추가된 필드를 제외한 **모든 필드**(공개 및 비공개 모두)를 `[NonSerialized]`(으)로 직렬화합니다. 기본적으로 형식의 필드 이름과 직렬화된 멤버 이름은 일치합니다. 이로 인해 `[Serializable]` 형식에서 프라이빗 필드의 이름이 바뀐 경우에도 비호환성이 발생했습니다. BinaryFormatter(으)로부터 마이그레이션하는 동안 직렬화된 필드 이름이 처리되고 재정의되는 방법을 이해해야 합니다.

C# 자동 속성

C#BinaryFormatter이 아닌 C# 컴파일러에서 생성되는 지원 필드를 직렬화합니다. 직렬화된 백업 필드의 이름은 제어할 수 없으며, 잘못된 C# 문자를 포함합니다. (https://sharplab.io/ 또는 ILSpy와 같은) C# 디컴파일러는 런타임에 C# 자동 속성이 표시되는 방법을 보여 줄 수 있습니다.

C#

```
[Serializable]
internal class PropertySample
{
    public string Name { get; set; }
}
```

이전 클래스는 C# 컴파일러에서 다음과 같이 변환됩니다.

C#

```
[Serializable]
internal class PropertySample
{
    private string <Name>k__BackingField;
```

```
    public string Name
    {
        get
        {
            return <Name>k__BackingField;
        }
        set
        {
            <Name>k__BackingField = value;
        }
    }
}
```

이 경우, `<Name>k__BackingField`은(는) **직렬화된 페이로드에서 `BinaryFormatter`이(가) 사용하는 멤버의 이름**입니다. `nameof`을(를) 사용하거나 다른 C# 연산자를 사용하여 이 이름을 가져올 수 없습니다.

ISerializable 인터페이스에는 사용자가 GetObjectData 메서드 중 하나를 사용하여 이름을 제어할 수 있는 AddValue 메서드가 함께 제공됩니다.

C#

```
// Note lack of any special attribute.
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("Name", this.Name);
}
```

이와 같은 사용자 지정이 적용된 경우, 역직렬화하는 동안에도 정보를 제공해야 합니다. 이는 제공하는 `Get` 메서드 중 하나를 사용하여 SerializationInfo에서 모든 값을 읽고, **serialization 생성자**을 사용함으로써 가능합니다.

C#

```
private PropertySample(SerializationInfo info, StreamingContext context)
{
    this.Name = info.GetString("Name");
}
```

ⓘ 참고

이 `nameof` 연산자는 여기서 의도적으로 사용되지 않았으며, 이는 페이로드를 유지할 수 있고 나중에 속성 이름을 바꿀 수 있기 때문입니다. 따라서 이름을 바꾸더라도 (`LastName` 속성도 도입하기로 결정했기 때문에 `FirstName`) 이전 버전과 호환되는 상

태를 유지하려면 serialization은 여전히 어딘가에 유지될 수 있는 이전 이름을 사용해야 합니다.

Serialization 바인더

SerializationBinder을(를) 클래스 로드를 제어하고 로드할 클래스를 위임하기 위해 사용하는 것이 권장됩니다. 이렇게 하면 보안 취약성이 최소화됩니다(그러므로 공격자가 페이로드를 수정하여 다른 항목을 역직렬화하고 로드하는 경우에도 허용된 형식만 로드됩니다).

이 형식에서 상속하고 BindToType 메서드를 재정의해야 이 형식을 사용할 수 있습니다.

직렬화 가능한 형식 목록은 닫혀 있는 것이 가장 권장되며, 이는 보안 취약성을 줄이는 데 도움이 되는 인스턴스화할 수 있는 형식을 알고 있기 때문입니다.

Windows Forms에 대한 BinaryFormatter 마이그레이션 가이드

2025. 06. 17.

BinaryFormatter 제거

.NET 9부터는 알려진 BinaryFormatter으로 인해 이(가) 더 이상 지원되지 않으며 PlatformNotSupportedException 앱을 포함하는 모든 프로젝트 형식에 대해 API가 Windows Forms을(를) 항상 throw합니다. BinaryFormatter를 참조하여 BinaryFormatter이(가) 발생시키는 위험 및 제거 이유에 대한 자세한 내용을 확인하세요.

BinaryFormatter의 제거로 인해 Windows Forms 애플리케이션이 영향을 받을 것으로 예상되며. .NET 9 이상 버전으로 마이그레이션을 완료하기 위한 조치를 취해야 합니다.

BinaryFormatter이(가) Windows Forms에게 미치는 영향

.NET 9 이전에는 리소스 저장 또는 로드와 같은 시나리오에 대한 데이터를 클립보드, 끌어서 놓기, 디자인 타임에 직렬화 및 역직렬화하는 데 Windows Forms이(가) BinaryFormatter을(를) 사용하였습니다. .NET 9부터, Windows Forms 및 WPF는 이러한 시나리오에 대해 내부적으로 BinaryFormatter 구현의 하위 집합을 사용합니다. 범용 직렬화/역직렬화에서는 BinaryFormatter에 대한 위험을 해결할 수 없지만 이러한 매우 구체적인 사용 사례의 위험을 완화하기 위한 조치가 알려진 형식 집합을 사용하여 취해졌습니다. 애플리케이션에서 마이그레이션 단계를 수행하지 않는 한 예외를 throw하는 지원되지 않거나 알 수 없는 형식에 대한 대체 BinaryFormatter 항목이 여전히 적용됩니다.

Windows Forms 및 WPF 앱은 다음과 같은 형식을 이러한 형식의 배열 및 목록과 함께 처리합니다. 마이그레이션 단계 없이 이러한 형식으로 클립보드, 끌어서 놓기 및 디자인 타임 리소스가 계속 작동합니다.

- `bool`
- `byte`
- `char`
- `decimal`
- `double`
- `int`
- `sbyte`
- `float`

- TimeSpan
- DateTime
- `uint`
- `string`
- `nint`
- `nuint`
- `long`
- `ulong`
- `short`
- `ushort`
- PointF
- RectangleF

다음과 같은 추가 형식도 Windows Forms이(가) 지원합니다.

- Bitmap
- ImageListStreamer

OLE 시나리오

BinaryFormatter을 참조하여 클립보드 및 끌어서 놓기 등의 OLE 시나리오에서 Windows Forms 제거가 미치는 영향과 마이그레이션 지침에 대한 자세한 정보를 확인하세요.

리소스(ResX)

Windows Forms 디자이너

Windows Forms Out-of-Process 디자이너는 ResX 직렬화 및 역직렬화에도 BinaryFormatter을(를) 내부적으로 사용합니다.

Windows Forms 디자이너의 표준 동작으로 인해 형식 및 속성은 실현하지 않고 직렬화에 참여 가능합니다. BinaryFormatter이(가) 사용되는 방법 중 인식되지 못했던 한 가지는 `public`에 IComponent 속성이 도입되고, 디자인 타임에 해당 속성이 채워지거나 편집되는 경우입니다. 다음과 같은 조건에서 해당 속성은 리소스 파일로 직렬화됩니다.

- 디자이너의 Form이(가) 저장될 때 공용 속성에는 데이터가 포함됩니다.
- 해당 속성은 읽기 전용이 아닙니다.
- 해당 속성은 `[DesignerSerializationVisibility(false)]`(으)로 특성화되지 않습니다.
- 해당 속성에는 DefaultValueAttribute이(가) 없습니다.
- 이 속성에는 CodeDOM serialization 프로세스 시 `bool ShouldSerialize[PropertyName]`(이)가 반환되는 각 `false` 메서드가 없습니다. (참고: 메서드에는 `private` 범위가 있을 수 있습

니다.)
- 해당 속성은 DesignerSerializer이(가) 없는 형식입니다.

이러한 문이 참이면 디자이너는 형식 변환기가 해당 속성의 형식에 있는지 여부를 확인합니다. 그러한 경우, 디자이너는 속성 콘텐츠를 직렬화하기 위해 형식 변환기를 사용합니다. 그렇지 않으면 BinaryFormatter을(를) 콘텐츠를 리소스 파일로 직렬화하기 위해 사용합니다. Windows Forms은(는) 분석기와 코드 수정을 추가함으로써 개발자가 인식하지 못한 상태에서 BinaryFormatter 직렬화가 발생할 수 있는 이러한 유형의 동작에 대한 인식을 높이고자 했습니다.

런타임 중에 리소스 로드

이전에 BinaryFormatter을(를) 통해 리소스 파일로 직렬화된 형식은 ResX 파일의 콘텐츠가 신뢰할 수 있는 데이터로 간주되므로 BinaryFormatter이(가) 필요하지 않으며 예상대로 역직렬화됩니다. 지원되지 않는 호환성 패키지를 사용하여 다시 추가할 수 있으며, 이는 BinaryFormatter 없이는 역직렬화가 발생할 수 없기 때문입니다. BinaryFormatter 마이그레이션 가이드: 호환성 패키지를 참조하여 자세한 내용을 확인하세요. `System.Resources.Extensions.UseBinaryFormatter`을(를) 리소스에 사용하려면 `true` 앱 컨텍스트 스위치를 BinaryFormatter(으)로 설정하는 추가 단계가 필요합니다.

MSBuild를 통해 리소스 파일 생성

MSBuild를 통해 리소스 파일을 생성하면 오류가 발생할 `MSB3825` 수 있습니다. 이 오류는 실행 시 BinaryFormatter을(를) 사용하여 이진 형식의 리소스를 역직렬화할 수 있음을 나타냅니다. 경고는 .NET 9 이상을 대상으로 하는 빌드에서 제거되지만 .NET 9의 모든 릴리스에서 제거가 아직 완료되지 않았습니다. 경고는 .NET 8 이하를 대상으로 하는 경우에만 고려해야 합니다. 앞서 설명한 대로, 이러한 리소스는 .NET 9 이상 버전의 런타임에서 BinaryFormatter를 사용하여 역직렬화되지 않습니다. 속성을 `GenerateResourceWarnOnBinaryFormatterUse` `false`.로 설정하여 경고를 끌 수 있습니다. 지원되지 않는 호환성 패키지를 사용하여 다시 추가할 수 있으며, 이는 BinaryFormatter없이는 역직렬화가 발생할 수 없기 때문입니다. 자세한 내용은 마이그레이션 가이드: 호환성 패키지를 참조BinaryFormatter하세요. 리소스를 위해 `System.Resources.Extensions.UseBinaryFormatter`를 사용하려면 앱 컨텍스트 스위치를 `true`로 설정하는 추가 단계가 필요합니다.

BinaryFormatter(으)로부터 멀리 마이그레이션

본질적으로 처리되지 않는 형식이 직렬화 및 역직렬화 중에 영향을 받는 시나리오에서 사용되는 경우 .NET 9 이상 버전으로 마이그레이션을 완료하기 위해 조치를 취해야 합니다.

OLE 시나리오

Windows Forms 및 Windows Presentation Foundation BinaryFormatter OLE 지침을 참조하여 클립보드 및 끌어서 놓기와 같은 시나리오에서 BinaryFormatter(으)로부터 멀리 마이그레이션하는 방법에 대한 자세한 내용을 확인하세요.

디자인 타임 동안 리소스 로드 및 저장

직렬화에 참여하는 유형 또는 속성에 대해 BinaryFormatter이(가) 등록되어 있는지 확인하는 것은 ResX 시나리오의 디자이너와 같이 리소스로 직렬화되는 동안 본질적으로 처리되지 않는 유형의 경우, `TypeConverter`(으)로부터 마이그레이션하는 권장 방법입니다. 이렇게 하면 직렬화 및 역직렬화를 진행하는 동안 한때 `TypeConverter`이(가) 사용되었던 위치 대신 BinaryFormatter이(가) 사용됩니다. TypeConverter 클래스를 참조하여 형식 변환기를 구현하는 방법에 대한 자세한 내용을 확인하세요.

호환성 해결 방법(권장되지 않음)

BinaryFormatter(으)로부터 마이그레이션할 수 없는 .NET 9 사용자는 지원되지 않는 호환성 패키지를 설치할 수 있습니다. BinaryFormatter마이그레이션 가이드: 호환성 패키지를 참조하여 자세한 내용을 확인하세요.

> **⊗ 주의**
>
> BinaryFormatter은(는) 위험하며 권장되지 않는데, 이는 DoS(서비스 거부), 정보 공개 또는 원격 코드 실행 등의 공격에 소비하는 앱을 위험에 빠뜨리기 때문입니다. **BinaryFormatter**을 참조하여 BinaryFormatter이(가) 발생시키는 위험에 대한 자세한 내용을 확인하세요.

문제

Windows Forms 앱에서 BinaryFormatter의 직렬화 또는 역직렬화와 관련하여 예기치 않은 동작을 경험하셨다면, github.com/dotnet/winforms에 이슈를 등록하시고, 그 문제가 BinaryFormatter제거와 관련이 있음을 명시해 주세요.

Windows Presentation Foundation(WPF) BinaryFormatter에 대한 마이그레이션 가이드

2025. 06. 17.

BinaryFormatter 제거

.NET 9부터는 알려진 BinaryFormatter으로 인해 이(가) 더 이상 지원되지 않으며 PlatformNotSupportedException 앱을 포함하는 모든 프로젝트 형식에 대해 API가 WPF을(를) 항상 throw합니다. BinaryFormatter를 참조하여 BinaryFormatter이(가) 발생시키는 위험 및 제거 이유에 대한 자세한 내용을 확인하세요.

BinaryFormatter의 제거로 인해 WPF 애플리케이션이 영향을 받을 것으로 예상되며. .NET 9 이상 버전으로 마이그레이션을 완료하기 위한 조치를 취해야 합니다.

BinaryFormatter이(가) WPF에게 미치는 영향

.NET 9 이전의 Windows Presentation Foundation(WPF)은 BinaryFormatter을(를) 사용하여 저널의 클립보드, 끌어서 놓기 및 로드/저장 상태 등의 시나리오에 대한 데이터를 직렬화 및 역직렬화하였습니다. .NET 9부터, 이러한 시나리오에 대해 WPF 및 Windows Forms는 BinaryFormatter 구현의 하위 집합을 내부적으로 사용합니다. 범용 직렬화/역직렬화에서는 BinaryFormatter에 대한 위험을 해결할 수 없지만 이러한 매우 구체적인 사용 사례의 위험을 완화하기 위한 조치가 알려진 형식 집합을 사용하여 취해졌습니다. 애플리케이션에서 마이그레이션 단계를 수행하지 않는 한 BinaryFormatter을(를) throw하는 지원되지 않거나 알 수 없는 형식에 대한 대체 PlatformNotSupportedException 항목이 여전히 적용됩니다.

WPF 및 WinForms 앱은 다음과 같은 형식을 이러한 형식의 배열 및 목록과 함께 처리합니다. 마이그레이션 단계 없이 이러한 형식으로 저널의 클립보드, 끌어서 놓기 및 Avalon 바인딩이 계속 작동합니다.

- `bool`
- `byte`
- `char`
- `decimal`
- `double`
- `int`
- `sbyte`
- `float`

- TimeSpan
- DateTime
- `uint`
- `string`
- `nint`
- `nuint`
- `long`
- `ulong`
- `short`
- `ushort`
- PointF
- RectangleF

OLE 시나리오

BinaryFormatter을 참조하여 클립보드 및 끌어서 놓기 등의 OLE 시나리오에서 BinaryFormatter 제거가 미치는 영향과 마이그레이션 지침에 대한 자세한 정보를 확인하세요.

다음과 같이 처리할 개체를 읽기/저장하기 위한 대체로 BinaryFormatter을(를) 사용한 함수를 참조할 수 있습니다. OLE 시나리오의 경우 SaveObjectToHandle 및 ReadObjectFromHandle

저널링

WPF에서 탐색 기록을 관리하는 동안 상태를 저장하거나 로드해야 하는 경우

클래스의 / `DataStream`를 호출하여 로드/저장합니다. 새 구현에서 처리하는 알 수 있는 형식의 일부가 아닌 요소에 사용되는 경우 BinaryFormatter을(를) 사용합니다.

개발자가 JournalEntry를 탐색`Navigate` `GoForward`할 때 또는 `GoBack`노드의 데이터가 로드되거나 스트림에 저장되어 상태를 저장합니다. 직렬화/역직렬화 중에 관련된 형식이 본질적으로 처리되지 않는 경우 BinaryFormatter이(가) 사용됩니다.

참조: DataStream.cs

호환성 해결 방법(권장되지 않음)

BinaryFormatter(으)로부터 마이그레이션할 수 없는 .NET 9 사용자는 지원되지 않는 호환성 패키지를 설치할 수 있습니다. BinaryFormatter마이그레이션 가이드: 호환성 패키지를 참조하여 자세한 내용을 확인하세요.

> ⊗ **주의**

BinaryFormatter은(는) 위험하며 권장되지 않는데, 이는 DoS(서비스 거부), 정보 공개 또는 원격 코드 실행 등의 공격에 소비하는 앱을 위험에 빠뜨리기 때문입니다. BinaryFormatter을 참조하여 BinaryFormatter이(가) 발생시키는 위험에 대한 자세한 내용을 확인하세요.

문제

WPF관련 BinaryFormatter 애플리케이션에서 예기치 않은 동작이 발생하는 경우 dotnet/wpf/issues 문제를 제출하고 문제가 BinaryFormatter제거와 관련이 있음을 표시하세요.

Windows Forms 및 Windows Presentation FoundationBinaryFormatterOLE 지침

아티클 • 2025. 01. 23.

이 문서에서는 BinaryFormatter 및 OLE(Windows Forms)에서의 Windows Presentation Foundation 시나리오에서 WPF 제거가 미치는 영향을 간략하게 설명합니다. Windows Forms에 대한 BinaryFormatter에서의 제거의 일반적인 효과에 대한 자세한 내용을 확인하세요. WPF에 대한 BinaryFormatter에서의 제거의 일반적인 효과에 대한 자세한 내용을 확인하세요.

BinaryFormatter 시나리오에서의 OLE

클립보드

OLE 및 System.Windows.Forms.DataFormats에서의 모든 표준 System.Windows.DataFormats DataFormats는 BinaryFormatter 및 사용자 지정 형식을 제외하고 `DataFormats.Serializable`을(를) 통과하지 않습니다. `DataFormats.Serializable` 또는 사용자 지정 형식을 사용하는 경우 BinaryFormatter및 WPF에서 설명한 대로 클립보드 시나리오에 본질적으로 처리되지 않는 유형이 포함되어 있다면 이(가) 사용됩니다. 특히, BinaryFormatter 또는 System.Windows.Forms.Clipboard.SetData이(가) 해당 유형과 함께 호출될 때, 그리고 System.Windows.Clipboard.SetData 또는 System.Windows.Forms.Clipboard.GetData이(가) 해당 유형을 가져오기 위해 호출될 때 System.Windows.Clipboard.GetData이(가) 사용됩니다. BinaryFormatter 또는 System.Windows.Forms.Clipboard.SetDataObject이(가) 호출되는 경우에도 System.Windows.Clipboard.SetDataObject이(가) 사용됩니다. BinaryFormatter 제거를 사용하면 BinaryFormatter이(가) 필요한 경우 데이터를 클립보드에서 설정할 때 예외가 표시되지 않습니다. 대신, 클립보드에서 본질적으로 처리되지 않는 형식을 가져오려고 할 때 BinaryFormatter이(가) 제거되었다는 문자열이 표시됩니다.

끌어서 놓기 기능

직렬화 및 역직렬화 중에 끌어서 놓기 시나리오에 본질적으로 처리되지 않는 형식이 포함된 경우, BinaryFormatter 또는 System.Windows.Forms.Control.DoDragDrop이(가) 호출되고 데이터가 프로세스 외부로 끌려갔을 때 System.Windows.DragDrop.DoDragDrop이(가) 사용됩니다. 형식이 본질적으로 처리되지 않는 경우 BinaryFormatter 또는 System.Windows.Forms.DataObject.GetData이 다른 프로세스에서 시작된 데이터를 검색

하기 위해 호출될 때 System.Windows.DataObject.GetData 또한 사용됩니다. BinaryFormatter 제거를 사용하면 본질적으로 처리되지 않는 형식에 대해 BinaryFormatter이(가) 제거되었다는 문자열이 다른 프로세스에서 시작된 데이터를 검색하려고 할 때 표시됩니다.

BinaryFormatter(으)로부터 마이그레이션

클립보드 및 끌어서 놓기

클립보드 및 끌어서 놓기 작업에 사용되는 본질적으로 처리되지 않는 형식의 경우 클립보드 또는 끌어서 놓기 API에 전달하기 전에 데이터를 해당 형식을 `byte[]` 또는 `string` 페이로드로 지정하는 것이 권장됩니다. 이렇게 하는 방법 중 하나는 JSON을 사용하는 것입니다. JSON 형식 형식 수신을 JSON 형식 형식을 클립보드 또는 끌어서 놓기 작업에 배치하기 위한 조정과 유사하게 처리하도록 조정해야 합니다. .NET 개체를 JSON(직렬화)으로 작성하는 방법을 참조하여 JSON을 사용하여 형식을 직렬화 및 역직렬화하는 방법에 대한 자세한 내용을 확인하세요.

문제

BinaryFormatter 직렬화 또는 역직렬화와 관련하여 Windows Forms 또는 WPF 앱에서 예기치 않은 동작이 발생하는 경우 각각 github.com/dotnet/winforms 또는 github.com/dotnet/wpf 문제를 제출하세요.

BinaryFormatter 및 관련 형식 사용 시 역직렬화 위험

이 문서는 다음 형식에 적용됩니다.

- BinaryFormatter
- SoapFormatter
- NetDataContractSerializer
- LosFormatter
- ObjectStateFormatter

이 문서는 다음과 같은 .NET 구현에 적용됩니다.

- .NET Framework 모든 버전
- .NET Core 2.1 - 3.1
- .NET 5 이상

> ⊗ **주의**
>
> BinaryFormatter 형식은 위험하므로 데이터 처리에 사용하지 않는 것이 ***좋습니다***. 애플리케이션은 처리 중인 데이터가 신뢰할 수 있는 것으로 간주하더라도 가능한 한 빨리 **사용을 BinaryFormatter** 중지해야 합니다. `BinaryFormatter`는 안전하지 않으며 안전하게 할 수 없습니다.

> ⓘ **참고 항목**
>
> .NET 9부터 기본 BinaryFormatter 구현은 설정이 변경됨에 따라 이전에 활성화되었던 경우에도 사용 시 예외를 발생시킵니다. 이러한 설정도 제거됩니다. **자세한 내용은 BinaryFormatter 마이그레이션 가이드**를 참조하세요.

역직렬화 취약성

요청 페이로드가 안전하지 않게 처리되는 위협 범주에 속하는 취약성은 역직렬화입니다. 앱에 대한 이러한 취약성을 성공적으로 악용하는 공격자는 대상 앱 내에서 서비스 거부 공격(DoS), 정보 공개, 또는 원격 코드 실행을 유발할 수 있습니다. 이 위험 범주는 일관되게 OWASP 상위 10 에 올랐습니다. 대상으로는 C/C++, Java, C#을 비롯한 다양한 언어 로 작성된 앱이 있습니다.

.NET에서 가장 위험한 대상은 BinaryFormatter 형식을 사용하여 데이터를 역직렬화하는 앱입니다. `BinaryFormatter`는 강력한 기능과 사용 편의성 때문에 .NET 에코시스템 전체에서 널리 사용

됩니다. 그러나 이 동일한 기능으로 인해 공격자가 대상 앱 내의 제어 흐름에 영향을 줄 수 있습니다. 공격이 성공하면 공격자가 대상 프로세스의 컨텍스트 내에서 코드를 실행할 수 있습니다.

더 간단한 비유로, 페이로드에 대해 `BinaryFormatter.Deserialize`를 호출하는 것은 해당 페이로드를 독립 실행형 실행 파일로 해석하고 시작하는 것과 같다고 가정합니다.

BinaryFormatter 보안 취약성

> ⚠ **Warning**
>
> `BinaryFormatter.Deserialize` 메서드는 신뢰할 수 없는 입력과 함께 사용할 경우 안전하지 **않습니다**. 이 문서의 뒷부분에서 설명하는 대체 방법 중 하나를 대신 사용하는 것이 좋습니다.

`BinaryFormatter`는 deserialization 취약성이 잘 이해된 위협 범주가 되기 전에 구현되었습니다. 따라서 코드가 최신 모범 사례를 따르지 않습니다. `Deserialize` 메서드는 공격자가 앱 사용에 대해 DoS 공격을 수행하기 위한 벡터로 사용될 수 있습니다. 해당 공격으로 인해 앱이 응답하지 않거나 예기치 않게 프로세스가 종료될 수 있습니다. 이 공격 범주는 `SerializationBinder` 또는 다른 `BinaryFormatter` 구성 스위치를 사용하여 완화할 수 없습니다. .NET에서는 이 동작을 ***'의도적'*** 인 것으로 간주하고 동작을 수정하는 코드 업데이트를 실행하지 않습니다.

`BinaryFormatter.Deserialize` 정보 공개 또는 원격 코드 실행과 같은 다른 공격 범주에 취약할 수 있습니다. 사용자 지정 SerializationBinder 과 같은 기능을 활용하면 이러한 위험을 적절하게 완화하기에 충분하지 않을 수 있습니다. 공격자가 기존 완화를 우회하는 새로운 악용을 발견할 가능성이 있습니다. .NET은 이러한 바이패스에 대한 응답으로 패치 게시를 커밋하지 않습니다. 또한 이러한 패치를 개발하거나 배포하는 것은 기술적으로 불가능할 수 있습니다. 시나리오를 평가하고 이러한 위험에 대한 잠재적 노출을 고려해야 합니다.

`BinaryFormatter` 소비자는 해당 앱에서 개별 위험 평가를 수행하는 것이 좋습니다. `BinaryFormatter`를 활용할지 여부를 결정하는 것은 전적으로 소비자의 책임입니다. 사용을 고려하고 있다면 보안, 기술, 평판, 법적 및 규정 결과에 대한 위험을 평가해야 합니다.

선호 대안

.NET에는 신뢰할 수 없는 데이터를 안전하게 처리할 수 있는 여러 가지 기본 제공 직렬 변환기가 있습니다.

- XmlSerializer 및 DataContractSerializer를 사용하여 개체 그래프를 XML로 직렬화하고 XML에서 역직렬화합니다. `DataContractSerializer`를 NetDataContractSerializer와 혼동하지 마세요.

- BinaryReader 및 BinaryWriter - XML 및 JSON에 사용됩니다.
- System.Text.Json API - 개체 그래프를 JSON으로 직렬화합니다.

위험한 대안

다음 직렬화 도구는 사용하지 마십시오.

- SoapFormatter
- LosFormatter
- NetDataContractSerializer
- ObjectStateFormatter

앞의 직렬 변환기는 모두 무제한 다형성 역직렬화를 수행하며 `BinaryFormatter`와 마찬가지로 위험합니다.

데이터를 신뢰할 수 있다고 가정할 경우의 위험

앱 개발자가 신뢰할 수 있는 입력만 처리하고 있다고 믿는 경우가 많습니다. 드물긴 하지만 안전한 입력 사례인 경우도 있습니다. 그러나 개발자 모르게 페이로드가 신뢰 경계를 넘어가는 경우가 훨씬 더 일반적입니다.

직원들이 워크스테이션의 데스크톱 클라이언트를 사용하여 서비스를 조작하는 경우 **온-프레미스 서버를 고려합니다**. 이 시나리오는 기본적으로 `BinaryFormatter` 활용이 허용되는 "안전한" 설치로 간주할 수도 있습니다. 그러나 이 시나리오는 단일 직원 머신에 대한 액세스 권한을 얻은 맬웨어가 기업 전체에 확산할 수 있는 벡터를 제공합니다. 해당 맬웨어는 기업의 `BinaryFormatter` 사용을 활용하여 직원 워크스테이션에서 백 엔드 서버로 수평 이동할 수 있습니다. 그런 다음, 회사의 중요한 데이터를 반출할 수 있습니다. 이러한 데이터는 거래 비밀 또는 고객 데이터를 포함할 수 있습니다.

`BinaryFormatter`를 사용하여 저장 상태를 유지하는 앱도 고려합니다. 처음에는 사용자 고유의 하드 드라이브에서 데이터를 읽고 쓰는 것이 사소한 위협을 나타내므로 안전한 시나리오처럼 보일 수 있습니다. 그러나 전자 메일 또는 인터넷을 통한 문서 공유가 일반적이며, 대부분의 최종 사용자는 다운로드된 파일을 여는 것을 위험한 동작으로 인식하지 않습니다.

이 시나리오는 불법적인 결과로 활용될 수 있습니다. 앱이 게임인 경우 저장 파일을 공유하는 사용자는 자신도 모르게 위험에 노출됩니다. 개발자 자신도 대상이 될 수 있습니다. 공격자는 개발자의 기술 지원팀에 전자 메일을 보내고 악성 데이터 파일을 첨부하여 지원 담당자에게 파일을 열도록 요청할 수 있습니다. 이러한 종류의 공격을 통해 공격자는 기업에 침투할 수 있습니다.

또 다른 시나리오는 데이터 파일이 클라우드 스토리지에 저장되고 사용자 머신 간에 자동으로 동기화되는 경우입니다. 클라우드 스토리지 계정에 대한 액세스 권한을 얻을 수 있는 공격자는 데이터 파일을 감염시킬 수 있습니다. 이 데이터 파일이 사용자 머신에 자동으로 동기화됩니다.

사용자가 다음에 데이터 파일을 열면 공격자의 페이로드가 실행됩니다. 따라서 공격자는 클라우드 스토리지 계정 손상을 활용하여 전체 코드 실행 권한을 얻을 수 있습니다.

데스크톱 설치 모델에서 클라우드 우선 모델로 이동하는 앱을 고려합니다. 이 시나리오에는 데스크톱 앱 또는 리치 클라이언트 모델에서 웹 기반 모델로 이동하는 앱이 포함됩니다. 데스크톱 앱에 대해 작성된 모든 위협 모델이 클라우드 기반 서비스에 반드시 적용되는 것은 아닙니다. 데스크톱 앱에 대한 위협 모델은 주어진 위협을 "클라이언트 자신이 공격하기에 흥미롭지 않다"고 무시할 수 있습니다. 그러나 클라우드 서비스를 공격하는 원격 사용자(클라이언트)를 고려할 경우, 동일한 위협이 흥미로워질 수 있습니다.

> ⓘ **참고 항목**
>
> 일반적으로, 직렬화의 의도는 개체를 앱으로 전송하거나 앱에서 전송하는 것입니다. 위협 모델링 연습에서 이러한 종류의 데이터 전송은 거의 항상 신뢰 경계를 넘는 것으로 표시됩니다.

참고 항목

- BinaryFormatter 마이그레이션 가이드
- 이진 직렬화
- YSoSerial.Net 을(를) 활용하여 적들이 앱을 공격하는 방법에 대한 연구
- deserialization 취약성에 대한 일반적인 배경 정보:
  - OWASP: 신뢰할 수 없는 데이터의 역직렬화
  - CWE-502: 신뢰할 수 없는 데이터의 역직렬화

Last updated on 2026. 02. 24.

BinaryFormatter 이벤트 원본

아티클 • 2023. 04. 19.

.NET 5부터 개체 BinaryFormatter 직렬화 또는 역직렬화가 발생하는 시기를 확인할 수 있는 기본 제공 EventSource 이 포함되어 있습니다. 앱은 EventListener 파생 형식을 사용하여 이러한 알림을 수신하고 기록할 수 있습니다.

이 기능은 SerializationBinder 또는 ISerializationSurrogate를 대체하지 않으며, 직렬화되거나 역직렬화되는 데이터를 수정하는 데 사용할 수 없습니다. 대신 이 이벤트 시스템은 직렬화 또는 역직렬화되는 형식에 대한 인사이트를 제공하기 위한 것입니다. 타사 라이브러리 코드에서 발생하는 호출 같이 `BinaryFormatter` 인프라에 대한 의도하지 않은 호출을 검색하는 데도 사용할 수 있습니다.

이벤트 설명

`BinaryFormatter` 이벤트 원본에는 잘 알려진 이름 `System.Runtime.Serialization.Formatters.Binary.BinaryFormatterEventSource`가 있습니다. 수신기는 6개의 이벤트를 구독할 수 있습니다.

SerializationStart 이벤트(id = `10`)

BinaryFormatter.Serialize가 호출되고 직렬화 프로세스를 시작한 경우 발생합니다. 이 이벤트는 `SerializationEnd` 이벤트와 쌍을 이룹니다. 개체가 자체 직렬화 루틴 내에서 `BinaryFormatter.Serialize`를 호출하는 경우 `SerializationStart` 이벤트를 재귀적으로 호출할 수 있습니다.

이 이벤트에는 페이로드가 포함되지 않습니다.

SerializationEnd 이벤트(id = `11`)

`BinaryFormatter.Serialize`가 작업을 완료했을 때 발생합니다. `SerializationEnd`의 각 항목은 짝이 없는 마지막 `SerializationStart` 이벤트의 완료를 나타냅니다.

이 이벤트에는 페이로드가 포함되지 않습니다.

SerializingObject 이벤트(id = `12`)

`BinaryFormatter.Serialize`가 기본 형식이 아닌 형식을 직렬화하는 동안 발생합니다. `BinaryFormatter` 인프라는 `string` 및 `int`와 같은 특정 형식을 특수 처리하며, 이러한 형

식이 발견될 때 이 이벤트를 발생시키지 않습니다. 이 이벤트는 사용자 정의 형식 및 `BinaryFormatter`가 고유하게 이해하지 못하는 다른 형식의 경우에 발생합니다.

이 이벤트는 `SerializationStart` 이벤트와 `SerializationEnd` 이벤트 사이에서 0번 이상 발생할 수 있습니다.

이 이벤트에는 인수가 하나 있는 페이로드가 포함되어 있습니다.

- `typeName`(`string`): 직렬화되는 형식의 정규화된 어셈블리 이름(Type.AssemblyQualifiedName 참조)입니다.

DeserializationStart 이벤트(id = `20`)

BinaryFormatter.Deserialize가 호출되고 역직렬화 프로세스를 시작한 경우 발생합니다. 이 이벤트는 `DeserializationEnd` 이벤트와 쌍을 이룹니다. 개체가 자체 역직렬화 루틴 내에서 `BinaryFormatter.Deserialize`를 호출하는 경우 `DeserializationStart` 이벤트를 재귀적으로 호출할 수 있습니다.

이 이벤트에는 페이로드가 포함되지 않습니다.

DeserializationEnd 이벤트(id = `21`)

`BinaryFormatter.Deserialize`가 작업을 완료했을 때 발생합니다. `DeserializationEnd`의 각 항목은 짝이 없는 마지막 `DeserializationStart` 이벤트의 완료를 나타냅니다.

이 이벤트에는 페이로드가 포함되지 않습니다.

DeserializingObject 이벤트(id = `22`)

`BinaryFormatter.Deserialize`가 기본 형식이 아닌 형식을 역직렬화하는 동안 발생합니다. `BinaryFormatter` 인프라는 `string` 및 `int`와 같은 특정 형식을 특수 처리하며, 이러한 형식이 발견될 때 이 이벤트를 발생시키지 않습니다. 이 이벤트는 사용자 정의 형식 및 `BinaryFormatter`가 고유하게 이해하지 못하는 다른 형식의 경우에 발생합니다.

이 이벤트는 `DeserializationStart` 이벤트와 `DeserializationEnd` 이벤트 사이에서 0번 이상 발생할 수 있습니다.

이 이벤트에는 인수가 하나 있는 페이로드가 포함되어 있습니다.

- `typeName`(`string`): 역직렬화되는 형식의 정규화된 어셈블리 이름(Type.AssemblyQualifiedName 참조)입니다.

[고급] 알림 하위 집합 구독

알림 하위 집합만 구독하려는 수신기는 사용할 키워드를 선택할 수 있습니다.

- `Serialization` = `(EventKeywords)1`: `SerializationStart`, `SerializationEnd` 및 `SerializingObject` 이벤트를 발생시킵니다.
- `Deserialization` = `(EventKeywords)2`: `DeserializationStart`, `DeserializationEnd` 및 `DeserializingObject` 이벤트를 발생시킵니다.

`EventListener` 등록 중에 키워드 필터를 제공하지 않으면 모든 이벤트가 발생합니다.

자세한 내용은 System.Diagnostics.Tracing.EventKeywords를 참조하세요.

예제 코드

코드는 다음과 같습니다.

- `System.Console`에 쓰는 `EventListener` 파생 형식을 만듭니다.
- 해당 수신기가 `BinaryFormatter`에서 생성된 알림을 구독합니다.
- `BinaryFormatter`를 사용하여 단순 개체 그래프를 직렬화 및 역직렬화합니다.
- 발생한 이벤트를 분석합니다.

C#

```
using System;
using System.Diagnostics.Tracing;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinaryFormatterEventSample
{
    class Program
    {
        static EventListener? _globalListener = null;

        static void Main(string[] args)
        {
            // First, set up the event listener.
            // Note: We assign it to a static field so that it doesn't get
GCed.
            // We also provide a callback that subscribes this listener to
all
            // events produced by the well-known BinaryFormatter source.

            _globalListener = new ConsoleEventListener();
            _globalListener.EventSourceCreated += (sender, args) =>
            {
                if (args.EventSource?.Name ==
```

```
"System.Runtime.Serialization.Formatters.Binary.BinaryFormatterEventSource")
                {
                    ((EventListener?)sender)?
                        .EnableEvents(args.EventSource,
EventLevel.LogAlways);
                }
            };

            // Next, create the Person object and serialize it.

            Person originalPerson = new Person()
            {
                FirstName = "Logan",
                LastName = "Edwards",
                FavoriteBook = new Book()
                {
                    Title = "A Tale of Two Cities",
                    Author = "Charles Dickens",
                    Price = 10.25m
                }
            };

            byte[] serializedPerson = SerializePerson(originalPerson);

            // Finally, deserialize the Person object.

            Person rehydratedPerson = DeserializePerson(serializedPerson);

            Console.WriteLine
                ($"Rehydrated person {rehydratedPerson.FirstName}
{rehydratedPerson.LastName}");
            Console.Write
                ($"Favorite book: {rehydratedPerson.FavoriteBook?.Title} ");
            Console.Write
                ($"by {rehydratedPerson.FavoriteBook?.Author}, ");
            Console.WriteLine
                ($"list price {rehydratedPerson.FavoriteBook?.Price}");
        }

        private static byte[] SerializePerson(Person p)
        {
            MemoryStream memStream = new MemoryStream();
            BinaryFormatter formatter = new BinaryFormatter();
#pragma warning disable SYSLIB0011 // BinaryFormatter.Serialize is obsolete
            formatter.Serialize(memStream, p);
#pragma warning restore SYSLIB0011

            return memStream.ToArray();
        }

        private static Person DeserializePerson(byte[] serializedData)
        {
            MemoryStream memStream = new MemoryStream(serializedData);
            BinaryFormatter formatter = new BinaryFormatter();
```

```
#pragma warning disable SYSLIB0011 // Danger: BinaryFormatter.Deserialize is
insecure for untrusted input
            return (Person)formatter.Deserialize(memStream);
#pragma warning restore SYSLIB0011
        }
    }

    [Serializable]
    public class Person
    {
        public string? FirstName;
        public string? LastName;
        public Book? FavoriteBook;
    }

    [Serializable]
    public class Book
    {
        public string? Title;
        public string? Author;
        public decimal? Price;
    }

    // A sample EventListener that writes data to System.Console.
    public class ConsoleEventListener : EventListener
    {
        protected override void OnEventWritten(EventWrittenEventArgs
eventData)
        {
            base.OnEventWritten(eventData);

            Console.WriteLine($"Event {eventData.EventName} (id=
{eventData.EventId}) received.");
            if (eventData.PayloadNames != null)
            {
                for (int i = 0; i < eventData.PayloadNames.Count; i++)
                {
                    Console.WriteLine($"{eventData.PayloadNames[i]} =
{eventData.Payload?[i]}");
                }
            }
        }
    }
}
```

위 코드의 출력은 다음 예제와 유사합니다.

출력

```
Event SerializationStart (id=10) received.
Event SerializingObject (id=12) received.
typeName = BinaryFormatterEventSample.Person, BinaryFormatterEventSample,
```

```
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Event SerializingObject (id=12) received.
typeName = BinaryFormatterEventSample.Book, BinaryFormatterEventSample,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Event SerializationStop (id=11) received.
Event DeserializationStart (id=20) received.
Event DeserializingObject (id=22) received.
typeName = BinaryFormatterEventSample.Person, BinaryFormatterEventSample,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Event DeserializingObject (id=22) received.
typeName = BinaryFormatterEventSample.Book, BinaryFormatterEventSample,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Event DeserializationStop (id=21) received.
Rehydrated person Logan Edwards
Favorite book: A Tale of Two Cities by Charles Dickens, list price 10.25
```

이 샘플에서 콘솔 기반 `EventListener`는 직렬화가 시작되고, `Person` 및 `Book`의 인스턴스가 직렬화된 다음 직렬화가 완료됨을 기록합니다. 마찬가지로 역직렬화가 시작되면 `Person` 및 `Book` 인스턴스가 역직렬화된 다음 역직렬화가 완료됩니다.

그런 다음 앱은 역직렬화된 `Person`에 포함된 값을 출력하여 실제로 개체가 올바로 직렬화 및 역직렬화되었음을 보여 줍니다.

참고 항목

`EventListener`를 사용하여 `EventSource` 기반 알림을 수신하는 방법에 대한 자세한 내용은 EventListener 클래스를 참조하세요.

System.Runtime.Serialization.DataContractAttribute 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

DataContractAttribute 특성을 DataContractSerializer에 의해 직렬화 및 역직렬화 작업에 사용되는 형식(클래스, 구조 또는 열거형)에 적용합니다. WCF(Windows Communication Foundation) 인프라를 사용하여 메시지를 보내거나 받는 경우 메시지에서 보낸 데이터를 보관하고 조작하는 모든 클래스에도 적용 DataContractAttribute 해야 합니다. 데이터 계약에 대한 자세한 내용은 데이터 계약 사용을 참조하세요.

또한 직렬화하려는 값을 보유하는 필드, 속성, 또는 이벤트에 DataMemberAttribute를 적용해야 합니다. 이를 적용하여 DataContractAttribute데이터를 직렬화하고 역직렬화할 수 있도록 명시적으로 설정합니다 DataContractSerializer .

> ⊗ **주의**
>
> 프라이빗 필드에 **DataMemberAttribute**를 적용할 수 있습니다. 필드에 의해 반환된 데이터(프라이빗인 경우에도)는 직렬화되고 역직렬화되므로 악의적인 사용자 또는 프로세스에서 보거나 가로챌 수 있습니다.

데이터 계약에 대한 자세한 내용은 데이터 계약 사용에 나열된 항목을 참조하세요.

데이터 계약

데이터 계약은 각 필드에 대한 이름 및 데이터 형식이 있는 필드 집합에 대한 추상 설명입니다. 데이터 계약은 서로 다른 플랫폼의 서비스가 상호 운용할 수 있도록 단일 구현 외부에 존재합니다. 서비스 간에 전달된 데이터가 동일한 계약을 준수하는 한 모든 서비스는 데이터를 처리할 수 있습니다. 이 처리를 *느슨하게 결합된 시스템*이라고도 합니다. 또한 데이터 계약은 애플리케이션에서 처리할 수 있도록 데이터를 배달하는 방법을 지정한다는 인터페이스와 비슷합니다. 예를 들어 데이터 계약은 "FirstName" 및 "LastName"이라는 두 개의 텍스트 필드가 있는 "Person"이라는 데이터 형식을 호출할 수 있습니다. 데이터 계약을 만들려면 클래스에 DataContractAttribute를 적용하고, serialize해야 하는 모든 필드나 속성에 DataMemberAttribute를 적용합니다. serialize될 때 데이터는 형식에 암시적으로 기본 제공되는 데이터 계약을 준수합니다.

> ⓘ **참고**

데이터 계약은 상속 동작의 실제 인터페이스와 크게 다릅니다. 인터페이스는 파생된 형식에 의해 상속됩니다. 기본 클래스에 DataContractAttribute 적용할 때 파생된 형식은 특성 또는 동작을 상속하지 않습니다. 그러나 파생 형식에 데이터 계약이 있는 경우 기본 클래스의 데이터 멤버가 serialize됩니다. 그러나 파생 클래스의 DataMemberAttribute 새 멤버에 적용하여 직렬화할 수 있도록 해야 합니다.

XML 스키마 문서 및 SvcUtil 도구

다른 서비스와 데이터를 교환하는 경우 데이터 계약을 설명해야 합니다. 현재 버전의 DataContractSerializer경우 XML 스키마를 사용하여 데이터 계약을 정의할 수 있습니다. (다른 형태의 메타데이터/설명은 동일한 용도로 사용할 수 있습니다.) 애플리케이션에서 XML 스키마를 만들려면 /dconly 명령줄 옵션과 함께 **ServiceModel 메타데이터 유틸리티 도구(Svcutil.exe)**를 사용합니다. 도구에 대한 입력이 어셈블리인 경우 기본적으로 도구는 해당 어셈블리에 있는 모든 데이터 계약 형식을 정의하는 XML 스키마 집합을 생성합니다. 반대로 Svcutil.exe 도구를 사용하여 데이터 계약으로 표현할 수 있는 구문을 사용하는 XML 스키마의 요구 사항을 준수하는 Visual Basic 또는 C# 클래스 정의를 만들 수도 있습니다. 이 경우 **/dconly** 명령줄 옵션은 필요하지 않습니다.

Svcutil.exe 도구에 대한 입력이 XML 스키마인 경우 기본적으로 도구는 클래스 집합을 만듭니다. 이러한 클래스를 검사하면 DataContractAttribute 해당 클래스가 적용된 것을 확인할 수 있습니다. 이러한 클래스를 사용하여 다른 서비스와 교환해야 하는 데이터를 처리하는 새 애플리케이션을 만들 수 있습니다.

WSDL(Web Services Description Language) 문서를 반환하는 엔드포인트에 대해 도구를 실행하여 코드 및 구성을 자동으로 생성하여 WCF(Windows Communication Foundation) 클라이언트를 만들 수도 있습니다. 생성된 코드에는 .로 표시된 형식이 DataContractAttribute포함됩니다.

기존 형식 다시 사용

데이터 계약에는 안정적인 이름과 멤버 목록이라는 두 가지 기본 요구 사항이 있습니다. 안정적인 이름은 네임스페이스 URI(Uniform Resource Identifier)와 계약의 로컬 이름으로 구성됩니다. 기본적으로 클래스에 DataContractAttribute 적용하면 클래스 이름을 로컬 이름으로 사용하고 클래스의 네임스페이스(접두사로 접두사 `"http://schemas.datacontract.org/2004/07/"`)를 네임스페이스 URI로 사용합니다. Name 및 Namespace 속성을 설정하여 기본값을 재정의할 수 있습니다. 네임스페이스를 변경하려면 ContractNamespaceAttribute을 네임스페이스에 적용할 수 있습니다. 필요한 대로 데이터를 처리하지만 데이터 계약과 다른 네임스페이스 및 클래스 이름을 갖는 기존 형식이 있는 경우 이 기능을 사용합니다. 기본값을 재정의하여 기존 형식을 다시 사용하고 직렬화된 데이터가 데이터 계약을 준수하도록 할 수 있습니다.

참고

모든 코드에서 더 긴 `DataContract` 단어 대신 단어를 **DataContractAttribute** 사용할 수 있습니다.

버전 관리

데이터 계약은 이후 버전의 자체도 수용할 수 있습니다. 즉, 계약의 이후 버전에 추가 데이터가 포함되어 있으면 해당 데이터가 저장되고 그대로 보낸 사람에게 반환됩니다. 이렇게 하려면 인터페이스를 구현합니다 IExtensibleDataObject .

버전 관리에 대한 자세한 내용은 데이터 계약 버전 관리를 참조하세요.

System.Runtime.Serialization.DataContractSerializer 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스를 DataContractSerializer 사용하여 형식의 인스턴스를 XML 스트림 또는 문서로 직렬화하고 역직렬화합니다. 예를 들어 이름 및 주소와 같은 필수 데이터가 포함된 속성으로 명명 `Person` 된 형식을 만들 수 있습니다. 그런 다음 클래스의 `Person` 인스턴스를 만들고 조작하고 나중에 검색할 수 있는 XML 문서 또는 즉시 전송을 위해 XML 스트림에서 모든 속성 값을 작성할 수 있습니다. 가장 중요한 DataContractSerializer 것은 WCF(Windows Communication Foundation) 메시지에서 보낸 데이터를 직렬화하고 역직렬화하는 데 사용됩니다. 클래스에 DataContractAttribute 특성을 적용하고 DataMemberAttribute 클래스 멤버에 특성을 적용하여 serialize되는 속성과 필드를 지정합니다.

serialize할 수 있는 형식 목록은 데이터 계약 직렬 변환기에서 지원하는 형식을 참조하세요.

DataContractSerializer를 사용하려면 먼저 형식에 맞는 클래스의 인스턴스와 개체(예: XmlDictionaryWriter의 인스턴스)를 만듭니다. 그런 다음, 메서드를 WriteObject 호출하여 데이터를 유지합니다. 데이터를 검색하려면 데이터 형식(예: XML 문서용)을 읽는 데 적합한 개체를 XmlDictionaryReader 만들고 메서드를 호출합니다 ReadObject .

사용 DataContractSerializer방법에 대한 자세한 내용은 Serialization 및 Deserialization을 참조하세요.

클라이언트 애플리케이션 구성 파일에서 dataContractSerializer 요소를 사용하여< 데이터 계약 직렬 변환기의> 형식을 설정할 수 있습니다.

직렬화 또는 역직렬화를 위한 클래스 준비

DataContractSerializer는 DataContractAttribute 및 DataMemberAttribute 클래스와 결합하여 사용됩니다. serialization을 위해 클래스를 준비하려면 클래스에 DataContractAttribute 적용합니다. 직렬화하고자 하는 데이터를 반환하는 클래스의 각 멤버에 대해 DataMemberAttribute를 적용합니다. 접근성에 관계없이 필드와 속성을 private, protected, internal, protected internal 또는 public으로 직렬화할 수 있습니다.

예를 들어, 여러분의 스키마는 `Customer`에 `ID` 속성을 지정하지만, 여러분은 이미 `Person` 속성을 가진 `Name`라는 형식을 사용하는 기존 애플리케이션을 가지고 있습니다. 계약을 준수하는 형식을 만들려면 먼저 클래스에 DataContractAttribute 적용합니다. 그런 다음 DataMemberAttribute를 직렬화하려는 모든 필드 또는 속성에 적용합니다.

> ⓘ **참고**
>
> 프라이빗 및 퍼블릭 멤버 모두에 DataMemberAttribute 적용할 수 있습니다.

XML의 최종 형식은 텍스트일 필요가 없습니다. 대신 DataContractSerializer에서 데이터를 XML 정보 세트로 작성하여 XmlReader 및 XmlWriter에서 인식하는 모든 형식으로 데이터를 쓸 수 있습니다. `XmlDictionaryReader 및 XmlDictionaryWriter 클래스를 사용하여 읽고 쓰는 것이 추천됩니다. 둘 다 DataContractSerializer와 함께 작동하도록 최적화되어 있기 때문입니다.`

직렬화 또는 역직렬화가 발생하기 전에 채워야 하는 필드 또는 속성이 있는 클래스를 만드는 경우 Version-Tolerant 직렬화 콜백에 설명된 대로 콜백 속성을 사용하십시오.

알려진 형식의 컬렉션에 추가

개체를 직렬화하거나 역직렬화할 때는 형식이 "알려진" 형식이 되도록 해야 합니다 DataContractSerializer. 먼저 구현 IEnumerable<T> 하는 클래스의 인스턴스(예: List<T>)를 만들고 알려진 형식을 컬렉션에 추가합니다. 그런 다음, DataContractSerializer를 사용하는 오버로드 중 하나로 IEnumerable<T>의 인스턴스를 생성하세요 (예: DataContractSerializer(Type, IEnumerable<Type>)).

> ⓘ **참고**
>
> 다른 기본 형식 DateTimeOffset 과 달리 구조체는 기본적으로 알려진 형식이 아니므로 알려진 형식 목록에 수동으로 추가해야 합니다(**데이터 계약 알려진 형식** 참조).

전방 호환성

이후 DataContractSerializer 버전의 계약과 호환되도록 설계된 데이터 계약을 이해합니다. 이러한 형식은 인터페이스를 구현합니다 IExtensibleDataObject . 인터페이스는 ExtensionData 속성을 가지고 있으며, ExtensionDataObject 개체를 반환합니다. 자세한 내용은 Forward-Compatible 데이터 계약을 참조하세요.

부분 신뢰로 실행

역직렬화하는 동안 대상 개체를 인스턴스화 DataContractSerializer 할 때 대상 개체의 생성자를 호출하지 않습니다. 부분 신뢰(즉, 공용 및 특성이 적용된 어셈블리)에서 액세스할 수 있고 일부 보안 관련 작업을 수행하는 `AllowPartiallyTrustedCallers` 형식을 작성하는 경우 생성자가 호출되지 않는다는 점에 유의해야 합니다. 특히 다음 기술은 작동하지 않습니다.

- 생성자를 내부 또는 프라이빗으로 만들거나 생성자에 추가하여 `LinkDemand` 부분 신뢰 액세스를 제한하려는 경우 부분 신뢰 하에서 역직렬화하는 동안 아무런 영향도 미치지 않습니다.
- 생성자가 실행된 것으로 가정하는 클래스를 코딩하면 클래스가 악용 가능한 잘못된 내부 상태가 될 수 있습니다.

System.Runtime.Serialization.IExtensibleDataObject 인터페이스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

인터페이스는 IExtensibleDataObject 데이터 계약 외부에 있는 데이터를 저장하는 데 사용되는 구조를 설정하거나 반환하는 단일 속성을 제공합니다. 추가 데이터는 클래스의 ExtensionDataObject 인스턴스에 저장되고 속성을 통해 ExtensionData 액세스됩니다. 데이터가 수신, 처리 및 다시 전송되는 왕복 작업에서 추가 데이터는 원래 보낸 사람에게 그대로 다시 전송됩니다. 이는 이후 버전의 계약에서 받은 데이터를 저장하는 데 유용합니다. 인터페이스를 구현하지 않으면 왕복 작업 중에 추가 데이터가 무시되고 삭제됩니다.

이 버전 관리 기능을 사용하려면

1. 클래스에서 IExtensibleDataObject 인터페이스를 구현합니다.
2. 형식에 ExtensionData 속성을 추가합니다.
3. 클래스에 형식 ExtensionDataObject 의 프라이빗 멤버를 추가합니다.
4. 새 프라이빗 멤버를 사용하여 속성에 대한 get 및 set 메서드를 구현합니다.
5. 클래스에 DataContractAttribute 특성을 적용합니다. Name 및 Namespace 속성을 필요한 경우 적절한 값으로 설정합니다.

형식의 버전 관리에 대한 자세한 내용은 데이터 계약 버전 관리를 참조하세요. 정방향 호환 데이터 계약을 만드는 방법에 대한 자세한 내용은 Forward-Compatible 데이터 계약을 참조하세요. 데이터 계약에 대한 자세한 내용은 데이터 계약 사용을 참조하세요.

System.Runtime.Serialization.XsdDataContractExporter 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

XsdDataContractExporter CLR(공용 언어 런타임) 형식으로 표현되는 데이터를 통합하는 웹 서비스를 만든 경우와 다른 웹 서비스에서 사용할 각 형식에 대한 XML 스키마를 내보내야 하는 경우 클래스를 사용합니다. 즉, XsdDataContractExporter CLR 형식 집합을 XML 스키마로 변환합니다. 사용할 수 있는 형식에 대한 자세한 내용은 데이터 계약 직렬 변환기에서 지원하는 형식을 참조하세요. 그런 다음 서비스와 상호 운용해야 하는 다른 사용자가 사용할 수 있도록 WSDL(Web Services Description Language) 문서를 통해 스키마를 노출할 수 있습니다.

반대로 기존 웹 서비스와 상호 운용해야 하는 웹 서비스를 XsdDataContractImporter 만드는 경우 XML 스키마를 변환하고 선택한 프로그래밍 언어로 데이터를 나타내는 CLR 형식을 만듭니다.

XsdDataContractExporter은 스키마의 컬렉션이 포함된 XmlSchemaSet 개체를 생성합니다. 속성을 통해 스키마 집합에 액세스합니다 Schemas() .

> **참고**
>
> 다른 웹 서비스에서 사용할 수 있는 XSD(XML 스키마 정의) 파일을 신속하게 생성하려면 다음을 XsdDataContractExporter사용합니다.

XmlSchemaSet으로 스키마 내보내기

XML 스키마 파일이 포함된 클래스의 XmlSchemaSet 인스턴스를 만들려면 다음 사항에 유의해야 합니다.

내보내는 형식 집합은 내부 데이터 계약 집합으로 기록됩니다. 따라서 새 형식만 집합에 CanExport 추가되므로 성능 저하 없이 메서드를 여러 번 호출하여 스키마 집합에 새 형식을 추가할 수 있습니다. 작업 중에 Export 기존 스키마는 추가되는 새 스키마와 비교됩니다. 충돌이 발생하면 예외가 발생합니다. 일반적으로 데이터 계약 이름이 같지만 서로 다른 계약(다른 멤버)을 가진 두 형식이 동일한 XsdDataContractExporter 인스턴스에서 내보내는 경우 충돌이 검색됩니다.

내보내기 도구 사용

이 클래스를 사용하는 권장 방법은 다음과 같습니다.

1. 오버로드 중 CanExport 하나를 사용하여 지정된 형식 또는 형식 집합을 내보낼 수 있는지 여부를 확인합니다. 요구 사항에 적합한 오버로드 중 하나를 사용합니다.

2. 해당 메서드를 호출합니다 Export .

3. 속성에서 스키마를 검색합니다 Schemas .

System.CommandLine 개요

라이브러리는 `System.CommandLine` 명령줄 입력 구문 분석 및 도움말 텍스트 표시와 같은 명령줄 앱에서 일반적으로 필요한 기능을 제공합니다.

사용하는 `System.CommandLine` 앱에는 .NET CLI, 추가 도구 및 많은 전역 및 로컬 도구가 포함됩니다.

앱 개발자의 경우 라이브러리:

- 명령줄 입력을 구문 분석하거나 도움말 페이지를 생성하는 코드를 작성할 필요가 없으므로 앱 코드 작성에 집중할 수 있습니다.
- 입력 구문 분석 코드와 독립적으로 앱 코드를 테스트할 수 있습니다.
- 트리밍 친화적이므로 빠르고 가벼운 AOT 지원 CLI 앱을 개발하는 데 적합합니다.

라이브러리를 사용하면 앱 사용자에게도 다음과 같은 이점이 있습니다.

- 명령줄 입력이 POSIX 또는 Windows 규칙에 따라 일관되게 구문 분석되도록 합니다.
- 탭 완성 및 응답 파일을 자동으로 지원합니다.

NuGet 패키지

라이브러리는 NuGet 패키지 System.CommandLine 로 사용할 수 있습니다.

다음 단계

System.CommandLine 시작하려면 다음 리소스를 참조하세요.

- 자습서: 시작하기 System.CommandLine
- 구문 개요: 명령, 옵션 및 인수

자세한 내용은 다음 리소스를 참조하세요.

- 결과를 구문 분석하고 호출하는 방법
- 구문 분석 및 유효성 검사를 사용자 지정하는 방법
- 파서 구성 방법
- 도움말을 사용자 지정하는 방법
- 탭 완성을 사용하도록 설정하고 사용자 지정하는 방법
- 명령줄 디자인 지침
- 2.0.0-beta5 마이그레이션 가이드
- System.CommandLine API 참조

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 12. 05.

사용 안내: System.CommandLine 시작하기

이 자습서에서는 System.CommandLine 라이브러리사용하는 .NET 명령줄 앱을 만드는 방법을 보여줍니다. 먼저 하나의 옵션이 있는 간단한 루트 명령을 만듭니다. 그런 다음, 해당 기반으로 빌드하여 여러 하위 명령과 각 명령에 대한 다양한 옵션을 포함하는 더 복잡한 앱을 만듭니다.

이 자습서에서는 다음 방법을 알아봅니다.

- ✓ 명령, 옵션 및 인수를 만듭니다.
- ✓ 옵션의 기본값을 지정합니다.
- ✓ 명령에 옵션 및 인수를 할당합니다.
- ✓ 명령의 모든 하위 명령에 옵션을 재귀적으로 할당합니다.
- ✓ 여러 수준의 중첩된 하위 명령을 사용합니다.
- ✓ 명령 및 옵션에 대한 별칭을 만듭니다.
- ✓ `string`, `string[]`, `int`, `bool`, `FileInfo` 및 열거형 옵션 형식과 작업합니다.
- ✓ 명령 작업 코드에서 옵션 값을 읽습니다.
- ✓ 옵션 구문 분석 및 유효성 검사에 사용자 지정 코드를 사용합니다.

필수 조건

- 최신 .NET SDK
- Visual Studio Code 편집기
- C# 개발 키트

또는

- .NET 데스크톱 개발 워크로드가 설치된 **Visual Studio**.

앱 만들기

"scl"이라는 .NET 9 콘솔 앱 프로젝트를 만듭니다.

1. 프로젝트에 대한 *scl* 폴더를 만든 다음 새 폴더에서 명령 프롬프트를 엽니다.
2. 다음 명령을 실행합니다.

.NET CLI

```
dotnet new console --framework net9.0
```

System.CommandLine 패키지 설치

- 다음 명령을 실행합니다.

  .NET CLI

  ```
  dotnet add package System.CommandLine
  ```

  또는 .NET 10 이상에서 다음을 수행합니다.

  .NET CLI

  ```
  dotnet package add System.CommandLine
  ```

인수를 파싱합니다

*Program.cs*의 내용을 다음 코드로 바꿉니다.

C#

```
using System.CommandLine;
using System.CommandLine.Parsing;

namespace scl;

class Program
{
    static int Main(string[] args)
    {
        Option<FileInfo> fileOption = new("--file")
        {
            Description = "The file to read and display on the console."
        };

        RootCommand rootCommand = new("Sample app for System.CommandLine");
        rootCommand.Options.Add(fileOption);

        ParseResult parseResult = rootCommand.Parse(args);
        if (parseResult.Errors.Count == 0 && parseResult.GetValue(fileOption) is
FileInfo parsedFile)
        {
            ReadFile(parsedFile);
            return 0;
        }
        foreach (ParseError parseError in parseResult.Errors)
        {
            Console.Error.WriteLine(parseError.Message);
        }
        return 1;
    }

    static void ReadFile(FileInfo file)
```

```
    {
        foreach (string line in File.ReadLines(file.FullName))
        {
            Console.WriteLine(line);
        }
    }
}
```

앞의 코드는 다음과 같습니다.

- 형식 이라는 `--file`FileInfo 만들고 루트 명령에 추가합니다.

C#

```
Option<FileInfo> fileOption = new("--file")
{
    Description = "The file to read and display on the console."
};

RootCommand rootCommand = new("Sample app for System.CommandLine");
rootCommand.Options.Add(fileOption);
```

- `args`을 구문 분석하고 `--file` 옵션에 값이 제공되었는지 확인합니다. 이 경우 구문 분석된 값을 사용하여 메서드를 호출 `ReadFile` 하고 종료 코드를 `0`반환합니다.

C#

```
ParseResult parseResult = rootCommand.Parse(args);
if (parseResult.Errors.Count == 0 && parseResult.GetValue(fileOption) is FileInfo
parsedFile)
{
    ReadFile(parsedFile);
    return 0;
}
```

- 값을 `--file`제공하지 않은 경우, 가능한 구문 분석 오류를 출력하고 종료 코드 `1`을 반환합니다.

C#

```
foreach (ParseError parseError in parseResult.Errors)
{
    Console.Error.WriteLine(parseError.Message);
}
return 1;
```

- 메서드는 `ReadFile` 지정된 파일을 읽고 콘솔에 해당 내용을 표시합니다.

C#

```
static void ReadFile(FileInfo file)
{
    foreach (string line in File.ReadLines(file.FullName))
    {
        Console.WriteLine(line);
    }
}
```

앱 테스트

명령줄 앱을 개발하는 동안 다음 방법 중 원하는 방법으로 테스트할 수 있습니다.

- 먼저 `dotnet build` 명령을 실행한 다음, 빌드 출력 폴더에서 명령 프롬프트를 열고 실행 파일을 실행하십시오.

  콘솔

  ```
  dotnet build
  cd bin/Debug/net9.0
  scl --file scl.runtimeconfig.json
  ```

- 다음 예제와 같이 `dotnet run` 사용하여 `run`후 `--` 명령 대신 앱에 옵션 값을 전달합니다.

  .NET CLI

  ```
  dotnet run -- --file bin/Debug/net9.0/scl.runtimeconfig.json
  ```

(이 자습서에서는 이러한 옵션 중 첫 번째 옵션을 사용 중이라고 가정합니다.)

앱을 실행하면 `--file` 옵션으로 지정된 파일의 내용이 표시됩니다.

출력

```
{
  "runtimeOptions": {
    "tfm": "net9.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "9.0.0"
    }
  }
}
```

`--help`을 제공하여 도움말을 출력하도록 요청하면 콘솔에 아무것도 출력되지 않습니다. 앱이 `--file`가 제공되지 않고 구문 분석 오류도 없는 상황을 아직 처리하지 못하기 때문입니다.

인수 구문 분석 및 ParseResult 호출

System.CommandLine 를 사용하면 지정된 기호(명령, 지시문 또는 옵션)가 성공적으로 구문 분석될 때 호출되는 작업을 지정할 수 있습니다. 작업은 ParseResult 매개 변수를 받아 `int` 종료 코드를 반환하는 대리자입니다. (비동기 작업 도 사용할 수 있습니다). 종료 코드는 메서드에서 ParseResult.Invoke(InvocationConfiguration) 반환되며 명령이 성공적으로 실행되었는지 여부를 나타내는 데 사용할 수 있습니다.

*Program.cs*의 내용을 다음 코드로 바꿉니다.

C#

```
using System.CommandLine;

namespace scl;

class Program
{
    static int Main(string[] args)
    {
        Option<FileInfo> fileOption = new("--file")
        {
            Description = "The file to read and display on the console."
        };

        RootCommand rootCommand = new("Sample app for System.CommandLine");
        rootCommand.Options.Add(fileOption);

        rootCommand.SetAction(parseResult =>
        {
            FileInfo parsedFile = parseResult.GetValue(fileOption);
            ReadFile(parsedFile);
            return 0;
        });

        ParseResult parseResult = rootCommand.Parse(args);
        return parseResult.Invoke();
    }

    static void ReadFile(FileInfo file)
    {
        foreach (string line in File.ReadLines(file.FullName))
        {
            Console.WriteLine(line);
        }
    }
}
```

앞의 코드는 다음과 같습니다.

- 루트 명령이 `ReadFile` 호출될 때 호출되는 메서드를 지정**합니다**.

  C#

  ```
  rootCommand.SetAction(parseResult =>
  {
      FileInfo parsedFile = parseResult.GetValue(fileOption);
      ReadFile(parsedFile);
      return 0;
  });
  ```

- `args` 결과를 구문 분석하고 **호출합니다**.

  C#

  ```
  ParseResult parseResult = rootCommand.Parse(args);
  return parseResult.Invoke();
  ```

앱을 실행하면 `--file` 옵션으로 지정된 파일의 내용이 표시됩니다.

지정 `scl --help`하여 도움말을 표시하도록 요청하면 다음 출력이 출력됩니다.

출력

```
Description:
  Sample app for System.CommandLine

Usage:
  scl [options]

Options:
  -?, -h, --help  Show help and usage information
  --version       Show version information
  --file          The file to read and display on the console
```

RootCommand 기본적으로 도움말 옵션, 버전 옵션 및 Suggest 지시문을 제공합니다. 이 ParseResult.Invoke(InvocationConfiguration) 메서드는 파싱된 기호의 동작을 호출합니다. 명령에 대해 명시적으로 정의된 작업일 수도 있고, `System.CommandLine`가 `System.CommandLine.Help.HelpOption`을 위해 정의한 도움말 작업일 수도 있습니다.

또한 `Invoke` 메서드가 구문 분석 오류를 감지하면, 이를 표준 오류에 출력하고, 표준 출력에 도움말을 출력하며, 종료 코드로 `1`을 반환합니다.

콘솔

```
scl --invalid bla
```

출력

```
Unrecognized command or argument '--invalid'.
Unrecognized command or argument 'bla'.
```

하위 명령 및 옵션 추가

이 섹션에서는 다음을 수행합니다.

- 추가 옵션을 만듭니다.
- 하위 명령을 만듭니다.
- 새 하위 명령에 새 옵션을 할당합니다.

새 옵션을 사용하면 최종 사용자가 포그라운드 및 배경 텍스트 색과 읽기 속도를 구성할 수 있습니다. 이러한 기능은 Teleprompter 콘솔 앱 자습서인용문 모음을 읽기 위해 사용됩니다.

1. dotnet 샘플 리포지토리의 sampleQuotes.txt 파일을 프로젝트 디렉터리로 복사합니다.

   (파일을 다운로드하는 방법에 대한 자세한 내용은 샘플 및 자습서의 지침을 참조하세요.)

2. 프로젝트 파일을 열고 닫는 `<ItemGroup>` 태그 바로 앞에 `</Project>` 요소를 추가합니다.

   XML

   ```
   <ItemGroup>
     <Content Include="sampleQuotes.txt">
       <CopyToOutputDirectory>Always</CopyToOutputDirectory>
     </Content>
   </ItemGroup>
   ```

   이 태그를 추가하면 앱을 빌드할 때 텍스트 파일이 출력 폴더에 복사됩니다. 따라서 해당 폴더에서 실행 파일을 실행할 때 폴더 경로를 지정하지 않고 이름으로 파일에 액세스할 수 있습니다.

3. *Program.cs* `--file` 옵션을 만드는 코드 후 읽기 속도 및 텍스트 색을 제어하는 옵션을 만듭니다.

   C#

   ```
   Option<int> delayOption = new("--delay")
   {
       Description = "Delay between lines, specified as milliseconds per character
   in a line.",
       DefaultValueFactory = parseResult => 42
   };
   Option<ConsoleColor> fgcolorOption = new("--fgcolor")
   {
   ```

```
    Description = "Foreground color of text displayed on the console.",
    DefaultValueFactory = parseResult => ConsoleColor.White
};
Option<bool> lightModeOption = new("--light-mode")
{
    Description = "Background color of text displayed on the console: default
is black, light mode is white."
};
```

4. 루트 명령을 만드는 줄 뒤 옵션을 추가하는 `--file` 코드를 삭제합니다. 새 하위 명령에 추가하므로 여기에서 제거합니다.

5. 루트 명령을 만드는 줄 뒤 `read` 하위 명령을 만듭니다. 속성 대신 `Options` 컬렉션 이니셜라이저 구문을 사용하여 이 하위 명령에 옵션을 추가하고 루트 명령에 하위 명령을 추가합니다.

   C#

   ```
   Command readCommand = new("read", "Read and display the file.")
   {
       fileOption,
       delayOption,
       fgcolorOption,
       lightModeOption
   };
   rootCommand.Subcommands.Add(readCommand);
   ```

6. `SetAction` 코드를 새 하위 명령에 대한 다음 `SetAction` 코드로 바꿉다.

   C#

   ```
   readCommand.SetAction(parseResult => ReadFile(
       parseResult.GetValue(fileOption),
       parseResult.GetValue(delayOption),
       parseResult.GetValue(fgcolorOption),
       parseResult.GetValue(lightModeOption)));
   ```

   루트 명령에 더 이상 작업이 필요하지 않으므로 더 이상 루트 명령을 호출 `SetAction` 하지 않습니다. 명령에 하위 명령이 있는 경우 일반적으로 명령줄 앱을 호출할 때 하위 명령 중 하나를 지정해야 합니다.

7. 작업 메서드를 `ReadFile` 다음 코드로 바꿉다.

   C#

   ```
   internal static void ReadFile(FileInfo file, int delay, ConsoleColor fgColor,
   bool lightMode)
   {
       Console.BackgroundColor = lightMode ? ConsoleColor.White :
   ```

```
ConsoleColor.Black;
    Console.ForegroundColor = fgColor;
    foreach (string line in File.ReadLines(file.FullName))
    {
        Console.WriteLine(line);
        Thread.Sleep(TimeSpan.FromMilliseconds(delay * line.Length));
    }
}
```

이제 앱은 다음과 같이 표시됩니다.

C#

```
using System.CommandLine;

namespace scl;

class Program
{
    static int Main(string[] args)
    {
        Option<FileInfo> fileOption = new("--file")
        {
            Description = "The file to read and display on the console."
        };

        Option<int> delayOption = new("--delay")
        {
            Description = "Delay between lines, specified as milliseconds per
character in a line.",
            DefaultValueFactory = parseResult => 42
        };
        Option<ConsoleColor> fgcolorOption = new("--fgcolor")
        {
            Description = "Foreground color of text displayed on the console.",
            DefaultValueFactory = parseResult => ConsoleColor.White
        };
        Option<bool> lightModeOption = new("--light-mode")
        {
            Description = "Background color of text displayed on the console:
default is black, light mode is white."
        };

        RootCommand rootCommand = new("Sample app for System.CommandLine");

        Command readCommand = new("read", "Read and display the file.")
        {
            fileOption,
            delayOption,
            fgcolorOption,
            lightModeOption
        };
        rootCommand.Subcommands.Add(readCommand);
```

```
        readCommand.SetAction(parseResult => ReadFile(
            parseResult.GetValue(fileOption),
            parseResult.GetValue(delayOption),
            parseResult.GetValue(fgcolorOption),
            parseResult.GetValue(lightModeOption)));

        return rootCommand.Parse(args).Invoke();
    }

    internal static void ReadFile(FileInfo file, int delay, ConsoleColor fgColor,
bool lightMode)
    {
        Console.BackgroundColor = lightMode ? ConsoleColor.White :
ConsoleColor.Black;
        Console.ForegroundColor = fgColor;
        foreach (string line in File.ReadLines(file.FullName))
        {
            Console.WriteLine(line);
            Thread.Sleep(TimeSpan.FromMilliseconds(delay * line.Length));
        }
    }
}
```

새 하위 명령 테스트

이제 하위 명령을 지정하지 않고 앱을 실행하려고 하면 오류 메시지와 사용 가능한 하위 명령을 지정하는 도움말 메시지가 표시됩니다.

콘솔

```
scl --file sampleQuotes.txt
```

출력

```
'--file' was not matched. Did you mean one of the following?
--help

Required command was not provided.
Unrecognized command or argument '--file'.
Unrecognized command or argument 'sampleQuotes.txt'.

Description:
  Sample app for System.CommandLine

Usage:
  scl [command] [options]

Options:
  -?, -h, --help  Show help and usage information
  --version       Show version information
```

```
Commands:
  read  Read and display the file.
```

하위 명령 `read` 대한 도움말 텍스트는 네 가지 옵션을 사용할 수 있음을 보여줍니다. 열거형에 유효한 값을 표시합니다.

콘솔

```
scl read -h
```

출력

```
Description:
  Read and display the file.

Usage:
  scl read [options]

Options:
  --file <file>                                                    The file to read and
display on the console.
  --delay <delay>                                                  Delay between lines,
specified as milliseconds per
                                                                   character in a line.
[default: 42]
  --fgcolor                                                        Foreground color of
text displayed on the console.
  <Black|Blue|Cyan|DarkBlue|DarkCyan|DarkGray|DarkGreen|Dark  [default: White]
  Magenta|DarkRed|DarkYellow|Gray|Green|Magenta|Red|White|Ye
  llow>
  --light-mode                                                     Background color of
text displayed on the console:
                                                                   default is black,
light mode is white.
  -?, -h, --help                                                   Show help and usage
information
```

`read` 옵션만 지정하는 하위 명령 `--file` 실행하고 다른 세 가지 옵션에 대한 기본값을 가져옵니다.

콘솔

```
scl read --file sampleQuotes.txt
```

문자당 42밀리초의 기본 지연으로 인해 읽기 속도가 느려집니다. `--delay` 낮은 숫자로 설정하여 속도를 높일 수 있습니다.

콘솔

```
scl read --file sampleQuotes.txt --delay 0
```

`--fgcolor` 및 `--light-mode` 사용하여 텍스트 색을 설정할 수 있습니다.

콘솔

```
scl read --file sampleQuotes.txt --fgcolor red --light-mode
```

잘못된 값을 `--delay`제공하면 다음과 같은 오류 메시지가 표시됩니다.

콘솔

```
scl read --file sampleQuotes.txt --delay forty-two
```

출력

```
Cannot parse argument 'forty-two' for option '--int' as expected type
'System.Int32'.
```

잘못된 값을 `--file`제공하면 예외가 발생합니다.

콘솔

```
scl read --file nofile
```

출력

```
Unhandled exception: System.IO.FileNotFoundException: Could not find file
'C:\bin\Debug\net9.0\nofile''.
File name: 'C:\bin\Debug\net9.0\nofile''
```

하위 명령 및 사용자 지정 유효성 검사 추가

이 섹션에서는 앱의 최종 버전을 만듭니다. 완료되면 앱에는 다음과 같은 명령과 옵션이 있습니다.

테이블 확장

Command	Options	Arguments
루트 명령	`--file` *(재귀)*	
`quotes`		

Command	Options	Arguments
`read`	`--delay`, `--fgcolor` `--light-mode`	
`add`		`quote` 및 `byline`
`delete`	`--search-terms`	

(*재귀* 옵션은 할당된 명령에서 사용할 수 있으며 모든 하위 명령에 재귀적으로 사용할 수 있습니다.)

다음은 옵션 및 인수를 사용하여 사용 가능한 각 명령을 호출하는 샘플 명령줄 입력입니다.

콘솔

```
scl quotes read --file sampleQuotes.txt --delay 40 --fgcolor red --light-mode
scl quotes add "Hello world!" "Nancy Davolio"
scl quotes delete --search-terms David "You can do" Antoine "Perfection is achieved"
```

1. *Program.cs* `--file` 옵션을 만드는 코드를 다음 코드로 바꿉니다.

   C#

   ```
   Option<FileInfo> fileOption = new("--file")
   {
       Description = "An option whose argument is parsed as a FileInfo",
       Required = true,
       DefaultValueFactory = result =>
       {
           if (result.Tokens.Count == 0)
           {
               return new FileInfo("sampleQuotes.txt");

           }
           string filePath = result.Tokens.Single().Value;
           if (!File.Exists(filePath))
           {
               result.AddError("File does not exist");
               return null;
           }
           else
           {
               return new FileInfo(filePath);
           }
       }
   };
   ```

   이 코드는 ArgumentResult 사용하여 사용자 지정 구문 분석, 유효성 검사 및 오류 처리를 제공합니다.

이 코드가 없으면 누락된 파일이 예외 및 스택 추적으로 보고됩니다. 이 코드를 사용하면 지정된 오류 메시지만 표시됩니다.

또한 이 코드는 기본값을 지정하므로 사용자 지정 구문 분석 메서드로 설정합니다 Option<T>.DefaultValueFactory .

2. `lightModeOption`만드는 코드 후에는 `add` 및 `delete` 명령에 대한 옵션과 인수를 추가합니다.

C#

```
Option<string[]> searchTermsOption = new("--search-terms")
{
    Description = "Strings to search for when deleting entries.",
    Required = true,
    AllowMultipleArgumentsPerToken = true
};
Argument<string> quoteArgument = new("quote")
{
    Description = "Text of quote."
};
Argument<string> bylineArgument = new("byline")
{
    Description = "Byline of quote."
};
```

AllowMultipleArgumentsPerToken 설정을 사용하면 첫 번째 항목 뒤의 목록에서 요소를 지정할 때 `--search-terms` 옵션 이름을 생략할 수 있습니다. 명령줄 입력에 해당하는 다음 예제를 만듭니다.

콘솔

```
scl quotes delete --search-terms David "You can do"
scl quotes delete --search-terms David --search-terms "You can do"
```

3. 루트 명령과 `read` 명령을 만드는 코드를 다음 코드로 바꿉다.

C#

```
RootCommand rootCommand = new("Sample app for System.CommandLine");
fileOption.Recursive = true;
rootCommand.Options.Add(fileOption);

Command quotesCommand = new("quotes", "Work with a file that contains
quotes.");
rootCommand.Subcommands.Add(quotesCommand);

Command readCommand = new("read", "Read and display the file.")
{
```

```
    delayOption,
    fgcolorOption,
    lightModeOption
};
quotesCommand.Subcommands.Add(readCommand);

Command deleteCommand = new("delete", "Delete lines from the file.");
deleteCommand.Options.Add(searchTermsOption);
quotesCommand.Subcommands.Add(deleteCommand);

Command addCommand = new("add", "Add an entry to the file.");
addCommand.Arguments.Add(quoteArgument);
addCommand.Arguments.Add(bylineArgument);
addCommand.Aliases.Add("insert");
quotesCommand.Subcommands.Add(addCommand);
```

이 코드는 다음과 같이 변경합니다.

- `--file` 명령에서 `read` 옵션을 제거합니다.
- `--file` 루트 명령에 이 옵션을 재귀 옵션으로 추가합니다.
- `quotes` 명령을 만들고 루트 명령에 추가합니다.
- 루트 명령 대신 `read` 명령을 `quotes` 명령에 추가합니다.
- `add` 및 `delete` 명령을 만들고 `quotes` 명령에 추가합니다.

결과는 다음 명령 계층 구조입니다.

루트 명령 └── `quotes` └── `read` └── `add` └── `delete`

이제 앱은 부모 명령()이 영역 또는 그룹을 지정하고 자식 명령(`quotes`, `read`, `add`)이 작업인 `delete` 패턴을 구현합니다.

재귀 옵션은 명령에 적용되고 하위 명령에 재귀적으로 적용됩니다. `--file` 루트 명령에 있으므로 앱의 모든 하위 명령에서 자동으로 사용할 수 있습니다.

4. `SetAction` 코드 후에 새 하위 명령에 대한 새 `SetAction` 코드를 추가합니다.

C#

```
deleteCommand.SetAction(parseResult => DeleteFromFile(
    parseResult.GetValue(fileOption),
    parseResult.GetValue(searchTermsOption)));

addCommand.SetAction(parseResult => AddToFile(
    parseResult.GetValue(fileOption),
    parseResult.GetValue(quoteArgument),
```

```
    parseResult.GetValue(bylineArgument))
    );
```

하위 명령은 `quotes` 리프 명령이 아니므로 작업이 없습니다. 하위 명령 `read`, `add` 및 `delete` `quotes` 아래의 리프 명령이며 각각에 대해 `SetAction` 호출됩니다.

5. `add` 및 `delete`에 대한 작업을 추가합니다.

C#

```
internal static void DeleteFromFile(FileInfo file, string[] searchTerms)
{
    Console.WriteLine("Deleting from file");

    var lines = File.ReadLines(file.FullName).Where(line => searchTerms.All(s
=> !line.Contains(s)));
    File.WriteAllLines(file.FullName, lines);
}
internal static void AddToFile(FileInfo file, string quote, string byline)
{
    Console.WriteLine("Adding to file");

    using StreamWriter writer = file.AppendText();
    writer.WriteLine($"{Environment.NewLine}{Environment.NewLine}{quote}");
    writer.WriteLine($"{Environment.NewLine}-{byline}");
}
```

완성된 앱은 다음과 같습니다.

C#

```
using System.CommandLine;

namespace scl;

class Program
{
    static int Main(string[] args)
    {
        Option<FileInfo> fileOption = new("--file")
        {
            Description = "An option whose argument is parsed as a FileInfo",
            Required = true,
            DefaultValueFactory = result =>
            {
                if (result.Tokens.Count == 0)
                {
                    return new FileInfo("sampleQuotes.txt");

                }
                string filePath = result.Tokens.Single().Value;
                if (!File.Exists(filePath))
```

```
                {
                    result.AddError("File does not exist");
                    return null;
                }
                else
                {
                    return new FileInfo(filePath);
                }
            }
        };

        Option<int> delayOption = new("--delay")
        {
            Description = "Delay between lines, specified as milliseconds per
character in a line.",
            DefaultValueFactory = parseResult => 42
        };
        Option<ConsoleColor> fgcolorOption = new("--fgcolor")
        {
            Description = "Foreground color of text displayed on the console.",
            DefaultValueFactory = parseResult => ConsoleColor.White
        };
        Option<bool> lightModeOption = new("--light-mode")
        {
            Description = "Background color of text displayed on the console:
default is black, light mode is white."
        };

        Option<string[]> searchTermsOption = new("--search-terms")
        {
            Description = "Strings to search for when deleting entries.",
            Required = true,
            AllowMultipleArgumentsPerToken = true
        };
        Argument<string> quoteArgument = new("quote")
        {
            Description = "Text of quote."
        };
        Argument<string> bylineArgument = new("byline")
        {
            Description = "Byline of quote."
        };

        RootCommand rootCommand = new("Sample app for System.CommandLine");
        fileOption.Recursive = true;
        rootCommand.Options.Add(fileOption);

        Command quotesCommand = new("quotes", "Work with a file that contains
quotes.");
        rootCommand.Subcommands.Add(quotesCommand);

        Command readCommand = new("read", "Read and display the file.")
        {
            delayOption,
            fgcolorOption,
```

```
            lightModeOption
        };
        quotesCommand.Subcommands.Add(readCommand);

        Command deleteCommand = new("delete", "Delete lines from the file.");
        deleteCommand.Options.Add(searchTermsOption);
        quotesCommand.Subcommands.Add(deleteCommand);

        Command addCommand = new("add", "Add an entry to the file.");
        addCommand.Arguments.Add(quoteArgument);
        addCommand.Arguments.Add(bylineArgument);
        addCommand.Aliases.Add("insert");
        quotesCommand.Subcommands.Add(addCommand);

        readCommand.SetAction(parseResult => ReadFile(
            parseResult.GetValue(fileOption),
            parseResult.GetValue(delayOption),
            parseResult.GetValue(fgcolorOption),
            parseResult.GetValue(lightModeOption)));

        deleteCommand.SetAction(parseResult => DeleteFromFile(
            parseResult.GetValue(fileOption),
            parseResult.GetValue(searchTermsOption)));

        addCommand.SetAction(parseResult => AddToFile(
            parseResult.GetValue(fileOption),
            parseResult.GetValue(quoteArgument),
            parseResult.GetValue(bylineArgument))
            );

        return rootCommand.Parse(args).Invoke();
    }

    internal static void ReadFile(FileInfo file, int delay, ConsoleColor fgColor,
bool lightMode)
    {
        Console.BackgroundColor = lightMode ? ConsoleColor.White :
ConsoleColor.Black;
        Console.ForegroundColor = fgColor;
        foreach (string line in File.ReadLines(file.FullName))
        {
            Console.WriteLine(line);
            Thread.Sleep(TimeSpan.FromMilliseconds(delay * line.Length));
        }
    }
    internal static void DeleteFromFile(FileInfo file, string[] searchTerms)
    {
        Console.WriteLine("Deleting from file");

        var lines = File.ReadLines(file.FullName).Where(line => searchTerms.All(s =>
!line.Contains(s)));
        File.WriteAllLines(file.FullName, lines);
    }
    internal static void AddToFile(FileInfo file, string quote, string byline)
    {
```

```
        Console.WriteLine("Adding to file");

        using StreamWriter writer = file.AppendText();
        writer.WriteLine($"{Environment.NewLine}{Environment.NewLine}{quote}");
        writer.WriteLine($"{Environment.NewLine}-{byline}");
    }
}
```

프로젝트를 빌드한 다음, 다음 명령을 시도합니다.

존재하지 않는 파일을 `--file` 명령을 사용하여 `read` 제출하면 예외 및 스택 추적 대신 오류 메시지가 표시됩니다.

콘솔

```
scl quotes read --file nofile
```

출력

```
File does not exist
```

하위 명령 `quotes`를 실행하려고 하면 `read`, `add`, 또는 `delete`을 사용하라는 메시지가 표시됩니다.

콘솔

```
scl quotes
```

출력

```
Required command was not provided.

Description:
  Work with a file that contains quotes.

Usage:
  scl quotes [command] [options]

Options:
  --file <file>   An option whose argument is parsed as a FileInfo [default:
sampleQuotes.txt]
  -?, -h, --help  Show help and usage information

Commands:
  read                          Read and display the file.
  delete                        Delete lines from the file.
  add, insert <quote> <byline>  Add an entry to the file.
```

하위 명령 `add`실행한 다음 텍스트 파일의 끝을 확인하여 추가된 텍스트를 확인합니다.

콘솔

```
scl quotes add "Hello world!" "Nancy Davolio"
```

파일의 시작 부분에서 검색 문자열을 사용하여 하위 명령 `delete` 실행한 다음 텍스트 파일의 시작을 확인하여 텍스트가 제거된 위치를 확인합니다.

콘솔

```
scl quotes delete --search-terms David "You can do" Antoine "Perfection is achieved"
```

> ⓘ **참고 항목**
>
> 출력 폴더의 파일은 `add` 및 `delete` 명령에서의 변경 사항을 반영합니다. 프로젝트 폴더에 있는 파일의 복사본은 변경되지 않은 상태로 유지됩니다.

다음 단계

이 자습서에서는 `System.CommandLine`사용하는 간단한 명령줄 앱을 만들었습니다. 라이브러리에 대한 자세한 내용은 System.CommandLine 개요참조하세요. 탭 완성 기능을 사용하려면 탭 완성을 System.CommandLine참조하세요.

Last updated on 2025. 12. 18.

구문 개요: 명령, 옵션 및 인수

`System.CommandLine`에서 인식하는 명령줄 구문을 이 문서에서 설명합니다. 이 정보는 .NET CLI를 포함하여 .NET 명령줄 앱의 사용자와 개발자 모두에게 유용합니다.

토큰

`System.CommandLine` 명령줄 입력을 공백으로 구분된 문자열인 *토큰*으로 구문 분석합니다. 예를 들어 다음 명령줄을 고려합니다.

.NET CLI

```
dotnet tool install dotnet-suggest --global --verbosity quiet
```

이 입력은 `dotnet` 애플리케이션에서 `tool`, `install`, `dotnet-suggest`, `--global`, `--verbosity`, `quiet` 토큰으로 구문 분석됩니다.

토큰은 명령, 옵션 또는 인수로 해석됩니다. 호출되는 명령줄 앱은 첫 번째 토큰 이후의 토큰이 해석되는 방법을 결정합니다. 다음 표에서는 앞의 예제를 해석하는 방법을 `System.CommandLine` 보여 줍니다.

테이블 확장

토큰	로 구문 분석됨
`tool`	하위 명령
`install`	하위 명령
`dotnet-suggest`	설치 명령에 대한 인수
`--global`	설치 명령에 대한 옵션
`--verbosity`	설치 명령에 대한 옵션
`quiet`	옵션에 대한 `--verbosity` 인수

토큰은 따옴표(`"`)로 묶인 경우 공백을 포함할 수 있습니다. 예제는 다음과 같습니다.

콘솔

```
dotnet tool search "ef migrations add"
```

기호 계층 구조(명령, 옵션, 인수)는 신뢰할 수 있는 입력으로 간주됩니다. 그러나 토큰 값은 신뢰할 수 없습니다.

명령어

명령줄 입력의 *명령은* 작업을 지정하거나 관련 작업 그룹을 정의하는 토큰입니다. 다음은 그 예입니다.

- `dotnet run`에서 `run`은(는) 작업을 지정하는 명령입니다.
- `dotnet tool install` 에서 `install`동작을 지정하는 명령이며 `tool` 관련 명령 그룹을 지정하는 명령입니다. 다른 도구 관련 명령(예: `tool uninstall`, `tool list`및)이 있습니다 `tool update`.

루트 명령

루트 명령은 앱 실행 파일의 이름을 지정하는 명령입니다. 예를 들어 `dotnet` 명령은 *dotnet.exe* 실행 파일을 지정합니다.

Command는 모든 명령이나 하위 명령에 대한 범용 클래스이고, RootCommand는 애플리케이션의 루트 진입점을 위한 특수 버전입니다. `RootCommand`는 `Command`, 버전 옵션 및 Suggest 지시문과 같은 루트별 동작 및 기본값을 추가하지만 모든 기능을 상속합니다.

하위 명령

대부분의 명령줄 앱은 *동사*라고도 하는 *하위 명령을 지원합니다*. 예를 들어 명령에는 `dotnet` `run` 입력 `dotnet run`하여 호출하는 하위 명령이 있습니다.

하위 명령에는 자체 하위 명령이 있을 수 있습니다. `dotnet tool install`는 `install`의 `tool` 명령어 하위 명령입니다.

다음 예제와 같이 하위 명령을 추가할 수 있습니다.

C#

```
RootCommand rootCommand = new();

Command sub1Command = new("sub1", "First-level subcommand");
rootCommand.Subcommands.Add(sub1Command);

Command sub1aCommand = new("sub1a", "Second level subcommand");
sub1Command.Subcommands.Add(sub1aCommand);
```

이 예제의 가장 안쪽 하위 명령은 다음과 같이 호출할 수 있습니다.

콘솔

```
myapp sub1 sub1a
```

옵션

옵션은 명령에 전달할 수 있는 명명된 매개 변수입니다. POSIX CLI는 일반적으로 옵션 이름 앞에 두 개의 하이픈(`--`)을 추가합니다. 다음 예제에서는 두 가지 옵션을 보여줍니다.

.NET CLI

```
dotnet tool update dotnet-suggest --verbosity quiet --global
                                  ^---------^       ^------^
```

이 예제에서 알 수 있듯이 옵션 값은 명시적(`quiet` for `--verbosity`) 또는 암시적(다음 항목은 없음)일 수 있습니다 `--global`. 지정된 값이 없는 옵션은 일반적으로 명령줄에 옵션이 지정된 경우 기본값 `true` 인 부울 매개 변수입니다.

일부 Windows 명령줄 앱의 경우 옵션 이름과 함께 선행 슬래시(`/`)를 사용하여 옵션을 식별합니다. 다음은 그 예입니다.

콘솔

```
msbuild /version
        ^------^
```

`System.CommandLine` 는 POSIX 및 Windows 접두사 규칙을 모두 지원합니다.

옵션을 구성할 때 접두사를 포함하여 옵션 이름을 지정합니다.

C#

```
Option<int> delayOption = new("--delay", "-d")
{
    Description = "An option whose argument is parsed as an int",
    DefaultValueFactory = parseResult => 42,
};
Option<string> messageOption = new("--message", "-m")
{
    Description = "An option whose argument is parsed as a string"
};

RootCommand rootCommand = new();
rootCommand.Options.Add(delayOption);
rootCommand.Options.Add(messageOption);
```

글로벌 옵션

명령에 옵션을 추가하고 모든 하위 명령에 재귀적으로 추가하려면 속성을 Recursive.로 설정합니다`true`. 자세한 정보 표시, 출력 형식 또는 구성 파일 경로와 같이 전체 애플리케이션에 적용

되는 옵션에 유용합니다.

다음 예제에서는 모든 명령에서 사용할 수 있는 전역 옵션을 만드는 방법을 보여 줍니다.

C#

```
static void GlobalOptionExample(string[] args)
{
    // Create a global option that applies to all commands.
    Option<bool> verboseOption = new("--verbose", "-v")
    {
        Description = "Show verbose output",
        Recursive = true
    };

    RootCommand rootCommand = new("Sample app demonstrating global options");
    rootCommand.Options.Add(verboseOption);

    Command buildCommand = new("build", "Build the project");
    Command testCommand = new("test", "Run tests");

    rootCommand.Subcommands.Add(buildCommand);
    rootCommand.Subcommands.Add(testCommand);

    buildCommand.SetAction(parseResult =>
    {
        bool isVerbose = parseResult.GetValue(verboseOption);
        Console.WriteLine($"Building project... (verbose: {isVerbose})");
    });

    testCommand.SetAction(parseResult =>
    {
        bool isVerbose = parseResult.GetValue(verboseOption);
        Console.WriteLine($"Running tests... (verbose: {isVerbose})");
    });

    rootCommand.Parse(args).Invoke();
}
```

이 예제에서는 `--verbose` 옵션을 각 명령에 개별적으로 추가하지 않고도 `build` 명령과 `test` 명령 모두에서 사용할 수 있습니다.

자세한 정보 표시 옵션

많은 명령줄 앱은 `--verbosity` 표시되는 출력의 양을 제어하는 옵션을 제공합니다. 디자인 지침은 5가지 표준 세부 수준 `Q[uiet]`(`M[inimal]` `N[ormal]` `D[etailed]` `Diag[nostic]`)을 권장합니다.

다음 예제에서는 전체 이름과 약어 이름을 모두 허용하고 별칭(`-v`)을 포함하는 자세한 정보 표시 옵션을 구현하는 방법을 보여 줍니다. 값 없이 지정된 경우 `-v` 기본적으로 디자인 지침에 따

라 진단 세부 정보 표시 수준으로 설정됩니다. 이 예제에는 `--verbosity quiet`에 대한 약식 옵션(`-q`)도 포함되어 있습니다.

C#

```
static void VerbosityOptionExample(string[] args)
{
    // Create verbosity option that accepts full and short names as strings.
    // -v without an argument defaults to diagnostic.
    Option<string> verbosityOption = new("--verbosity", "-v")
    {
        Description = "Output verbosity level. Allowed values are q[uiet],
m[inimal], n[ormal], d[etailed], and diag[nostic].",
        Recursive = true,
        Arity = ArgumentArity.ZeroOrOne,
        DefaultValueFactory = result =>
            {
                // This runs only when the option isn't specified at all.
                // If the option is specified without a value (for example, `-v`),
                // DefaultValueFactory isn't called and the value is an empty
string,
                // which is handled later when mapping to "diagnostic".
                return "normal";
            }
    };

    // Add -q as a separate option for quiet verbosity.
    Option<bool> quietOption = new("-q")
    {
        Description = "Set verbosity to quiet (shorthand for --verbosity quiet)",
        Recursive = true
    };

    // Handle both short and long forms.
    verbosityOption.Validators.Add(result =>
    {
        if (result.Tokens.Count == 0)
        {
            return; // Allow default value.
        }

        string value = result.Tokens.Single().Value.ToLowerInvariant();
        string[] validValues = new[] { "quiet", "q", "minimal", "m", "normal", "n",
"detailed", "d", "diagnostic", "diag" };

        if (!validValues.Contains(value))
        {
            result.AddError($"Argument '{value}' not recognized. Must be one of:
'q[uiet]', 'm[inimal]', 'n[ormal]', 'd[etailed]', 'diag[nostic]'");
        }
    });

    RootCommand rootCommand = new("Sample app with verbosity");
```

```
    rootCommand.Options.Add(verbosityOption);
    rootCommand.Options.Add(quietOption);

    Command processCommand = new("build", "Build the project");
    rootCommand.Subcommands.Add(processCommand);

    processCommand.SetAction(parseResult =>
    {
        string verbosityString;

        // Check if -q was specified.
        if (parseResult.GetValue(quietOption))
        {
            verbosityString = "quiet";
        }
        else
        {
            verbosityString = parseResult.GetValue(verbosityOption);

            // If the option was specified without an argument,
            // the value will be empty string.
            // Set it to diagnostic as per design guidance.
            if (string.IsNullOrEmpty(verbosityString))
            {
                verbosityString = "diagnostic";
            }
        }

        // Convert string to enum.
        VerbosityLevel verbosity = verbosityString switch
        {
            "quiet" or "q" => VerbosityLevel.Quiet,
            "minimal" or "m" => VerbosityLevel.Minimal,
            "normal" or "n" => VerbosityLevel.Normal,
            "detailed" or "d" => VerbosityLevel.Detailed,
            "diagnostic" or "diag" => VerbosityLevel.Diagnostic,
            _ => VerbosityLevel.Normal
        };

        Console.WriteLine($"Verbosity level: {verbosity}");
    });

    rootCommand.Parse(args).Invoke();
}

enum VerbosityLevel
{
    Quiet,
    Minimal,
    Normal,
    Detailed,
    Diagnostic
}
```

다음 명령줄을 사용하여 앱을 호출할 수 있습니다.

콘솔

```
myapp build -q
myapp build -v
myapp build --verbosity minimal
myapp build --verbosity m
myapp build
```

필수 옵션

일부 옵션에는 필수 인수가 있습니다. 예를 들어 .NET CLI `--output` 에서는 폴더 이름 인수가 필요합니다. 인수가 제공되지 않으면 명령이 실패합니다. 필요한 옵션을 만들려면 다음 예제와 같이 해당 Required 속성을 `true` 다음 예제와 같이 설정합니다.

C#

```
Option<FileInfo> fileOption = new("--output")
{
    Required = true
};
```

필수 옵션에 기본값(속성을 통해 DefaultValueFactory 지정됨)이 있는 경우 명령줄에 옵션을 지정할 필요가 없습니다. 이 경우 기본값은 필요한 옵션 값을 제공합니다.

주장들

인수는 명령에 전달할 수 있는 명명되지 않은 매개 변수입니다. 다음 예제는 `build` 명령에 대한 인수를 보여줍니다.

콘솔

```
dotnet build myapp.csproj
             ^----------^
```

인수를 구성할 때 인수 이름을 지정하고(구문 분석에는 사용되지 않지만 이름 또는 도움말 표시로 구문 분석된 값을 가져오는 데 사용할 수 있습니다.) 다음을 입력합니다.

C#

```
Argument<int> delayArgument = new("delay")
{
    Description = "An argument that is parsed as an int.",
    DefaultValueFactory = parseResult => 42
```

```
};
Argument<string> messageArgument = new("message")
{
    Description = "An argument that is parsed as a string."
};

RootCommand rootCommand = new();
rootCommand.Arguments.Add(delayArgument);
rootCommand.Arguments.Add(messageArgument);
```

기본값

인수와 옵션 모두 명시적으로 제공된 인수가 없는 경우 적용되는 기본값을 가질 수 있습니다. 예를 들어 많은 옵션은 명령줄에 옵션 이름이 나타날 때 암시적으로 `true` 기본값을 가지는 부울 매개 변수입니다. 다음 명령줄 예제는 동일합니다.

.NET CLI

```
dotnet tool update dotnet-suggest --global
                                  ^------^

dotnet tool update dotnet-suggest --global true
                                  ^-----------^
```

기본값 없이 정의된 인수는 필수 인수로 처리됩니다.

구문 분석 오류

옵션 및 인수에는 예상 형식이 있으며 값을 구문 분석할 수 없는 경우 오류가 발생합니다. 예를 들어, 다음 명령어는 `--verbosity`에 대해 `silent`가 유효한 값 중 하나가 아니기 때문에 오류가 발생합니다.

.NET CLI

```
dotnet build --verbosity silent
```

C#

```
Option<string> verbosityOption = new("--verbosity", "-v")
{
    Description = "Set the verbosity level",
};
verbosityOption.AcceptOnlyFromAmong("quiet", "minimal", "normal", "detailed",
"diagnostic");
RootCommand rootCommand = new() { verbosityOption };
```

```
ParseResult parseResult = rootCommand.Parse(args);
foreach (ParseError parseError in parseResult.Errors)
{
    Console.WriteLine(parseError.Message);
}
```

출력

```
Argument 'silent' not recognized. Must be one of:
        'quiet'
        'minimal'
        'normal'
        'detailed'
        'diagnostic'
```

인수에는 제공할 수 있는 값 수에 대한 기대도 있습니다. 인수 진도 섹션에 예제가 제공됩니다.

옵션 및 인수 순서

명령줄에서 인수 앞의 옵션 또는 옵션 앞의 인수를 제공할 수 있습니다. 다음 명령은 동일합니다.

.NET CLI

```
dotnet add package System.CommandLine --no-restore
dotnet add package --no-restore System.CommandLine
```

옵션은 순서대로 지정할 수 있습니다. 다음 명령은 동일합니다.

.NET CLI

```
dotnet add package System.CommandLine --no-restore --source
https://api.nuget.org/v3/index.json
dotnet add package System.CommandLine --source https://api.nuget.org/v3/index.json -
-no-restore
```

인수가 여러 대 있는 경우 순서가 중요합니다. 다음 명령은 동일하지 않습니다. 값의 순서가 다르므로 결과가 다를 수 있습니다.

콘솔

```
myapp argument1 argument2
myapp argument2 argument1
```

별칭

POSIX와 Windows 모두에서 일부 명령과 옵션에 별칭이 있는 것이 일반적입니다. 일반적으로 입력하기 쉬운 짧은 형식입니다. 별칭은 다른 목적으로도 사용할 수 있는데, 케이스 민감도를 시뮬레이션하거나 단어의 대체 철자를 지원하는 데 사용될 수 있습니다.

POSIX 짧은 폼에는 일반적으로 단일 선행 하이픈과 단일 문자가 있습니다. 다음 명령은 동일합니다.

.NET CLI

```
dotnet build --verbosity quiet
dotnet build -v quiet
```

GNU 표준 은 자동 별칭을 권장합니다. 즉, 긴 형식 명령 또는 옵션 이름의 일부를 입력할 수 있으며 허용됩니다. 이 동작은 다음 명령줄을 동일하게 만듭니다.

.NET CLI

```
dotnet publish --output ./publish
dotnet publish --outpu ./publish
dotnet publish --outp ./publish
dotnet publish --out ./publish
dotnet publish --ou ./publish
dotnet publish --o ./publish
```

`System.CommandLine` 는 자동 별칭을 지원하지 않습니다. 각 별칭은 명시적으로 지정해야 합니다. 명령과 옵션 모두 속성을 노출합니다 `Aliases` . `Option` 에는 별칭을 매개 변수로 허용하는 생성자가 있으므로 한 줄에 여러 별칭이 있는 옵션을 정의할 수 있습니다.

C#

```
Option<bool> helpOption = new("--help", ["-h", "/h", "-?", "/?"]);
Command command = new("serialize") { helpOption };
command.Aliases.Add("serialise");
```

정의하는 옵션 별칭의 수를 최소화하고 특정 별칭을 정의하지 않는 것이 좋습니다. 자세한 내용은 짧은 형식 별칭을 참조하세요.

대/소문자 구분

명령 및 옵션 이름 및 별칭은 POSIX 규칙에 따라 기본적으로 대/소문자를 구분하며 `System.CommandLine` 이 규칙을 따릅니다. CLI가 대/소문자를 구분하지 않도록 하려면 다양한 대/소문자 대체에 대한 별칭을 정의합니다. 예를 들어 `--additional-probing-path` 는 `--Additional-Probing-Path` 과 `--ADDITIONAL-PROBING-PATH` 의 별칭을 가질 수 있습니다.

일부 명령줄 도구에서 대/소문자 구분의 차이는 기능의 차이를 의미합니다. 예를 들어, git clean -X 은 git clean -x 과 다르게 작동합니다. .NET CLI는 모두 소문자로 작성해야 합니다.

열거형을 기반으로 하는 옵션의 인수 값에는 대/소문자 구분이 적용되지 않습니다. 열거형 이름은 대/소문자 구분에 관계없이 일치합니다.

`--` 토큰

POSIX 규칙은 이중 대시(`--`) 토큰을 이스케이프 메커니즘으로 해석합니다. 이중 대시 토큰 뒤에 있는 모든 항목은 명령에 대한 인수로 해석됩니다. 이 기능은 옵션으로 해석되지 않으므로 옵션처럼 보이는 인수를 제출하는 데 사용할 수 있습니다.

*myapp*이 `message`이라는 인수를 받는다고 가정하고, `message`의 값을 `--interactive`로 설정하려고 합니다. 다음 명령줄은 예기치 않은 결과를 제공할 수 있습니다.

콘솔

```
myapp --interactive
```

옵션이 없 `myapp` 으면 `--interactive` 토큰이 `--interactive` 인수로 해석됩니다. 그러나 앱에 옵션이 있는 `--interactive` 경우 이 입력은 해당 옵션을 참조하는 것으로 해석됩니다.

다음 명령줄에서는 이중 대시 토큰을 사용하여 인수 값을 `message` "--interactive"로 설정합니다.

콘솔

```
myapp -- --interactive
      ^^
```

`System.CommandLine` 는 이 이중 대시 기능을 지원합니다.

옵션-인수 구분 기호

`System.CommandLine` 에서는 옵션 이름과 해당 인수 사이의 구분 기호로 공백 '=', ':'를 사용할 수 있습니다. 예를 들어 다음 명령은 동일합니다.

.NET CLI

```
dotnet build -v quiet
dotnet build -v=quiet
dotnet build -v:quiet
```

POSIX 규칙을 사용하면 단일 문자 옵션 별칭을 지정할 때 구분 기호를 생략할 수 있습니다. 예를 들어 다음 명령은 동일합니다.

콘솔

```
myapp -vquiet
myapp -v quiet
```

System.CommandLine 는 기본적으로 이 구문을 지원합니다.

인수 개수

옵션 또는 명령 인수의 *경도* 는 해당 옵션 또는 명령이 지정된 경우 전달할 수 있는 값의 수입니다.

Arity는 다음 표와 같이 최소값과 최대값으로 표현됩니다.

테이블 확장

민	맥스	유효성의 예	예시
0	0	유효한:	--파일
		올바르지 않음:	--file a.json
		올바르지 않음:	--file a.json --file b.json
0	1	유효한:	--플래그
		유효한:	--flag true
		유효한:	--flag false
		올바르지 않음:	--flag 거짓 --flag 거짓
1	1	유효한:	--file a.json
		올바르지 않음:	--파일
		올바르지 않음:	--file a.json --file b.json
0	*n*	유효한:	--파일
		유효한:	--file a.json
		유효한:	--file a.json --file b.json
1	*n*	유효한:	--file a.json

민	맥스	유효성의 예	예시
		유효한:	--file a.json b.json
		올바르지 않음:	--파일

`System.CommandLine` ArgumentArity 에는 다음 값을 사용하여 arity를 정의하기 위한 구조체가 있습니다.

- Zero - 허용되는 값이 없습니다.
- ZeroOrOne - 값 하나 또는 값이 없을 수 있습니다.
- ExactlyOne - 값이 하나 있어야 합니다.
- ZeroOrMore - 값 하나, 여러 값 또는 값이 없을 수 있습니다.
- OneOrMore - 여러 값을 가질 수 있으며, 하나 이상의 값을 반드시 가져야 합니다.

`Arity` 속성을 사용하여 arity를 명시적으로 설정할 수 있지만, 대부분의 경우에는 그렇게 할 필요가 없습니다. `System.CommandLine` 는 인수 형식에 따라 인수 계수 값을 자동으로 결정합니다.

테이블 확장

인수 형식	기본 인수 개수
`Boolean`	`ArgumentArity.ZeroOrOne`
컬렉션 형식	`ArgumentArity.ZeroOrMore`
기타 등등	`ArgumentArity.ExactlyOne`

옵션 재정의

`System.CommandLine`의 수용 가능 최대값이 1일 때에도 옵션의 여러 인스턴스를 허용하도록 구성할 수 있습니다. 이 경우 반복된 옵션의 마지막 인스턴스는 이전 인스턴스를 덮어씁니다. 다음 예제에서는 값 2가 명령에 전달됩니다 `myapp` .

콘솔

```
myapp --delay 3 --message example --delay 2
```

여러 인수

기본적으로 명령을 호출할 때 옵션 이름을 반복하여 최대 진도가 1보다 큰 옵션에 대해 여러 인수를 지정할 수 있습니다.

콘솔

```
myapp --items one --items two --items three
```

옵션 이름을 반복하지 않고 여러 인수를 허용하려면 AllowMultipleArgumentsPerToken을 `true` 으로 설정합니다. 이 설정을 사용하면 다음 명령줄을 입력할 수 있습니다.

콘솔

```
myapp --items one two three
```

최대 인수 크기가 1인 경우 동일한 설정에 다른 효과가 있습니다. 옵션을 반복할 수 있지만 줄의 마지막 값만 사용합니다. 다음 예제에서는 값 `three` 이 앱에 전달됩니다.

콘솔

```
myapp --item one --item two --item three
```

옵션 번들링

POSIX는 *스택*이라고도 하는 단일 문자 옵션의 *묶음을* 지원하는 것이 좋습니다. 번들 옵션은 단일 하이픈 접두사 다음에 함께 지정된 단일 문자 옵션 별칭입니다. 마지막 옵션만 인수를 지정할 수 있습니다. 예를 들어 다음 명령줄은 동일합니다.

콘솔

```
git clean -f -d -x
git clean -fdx
```

옵션 번들 다음에 인수가 제공되면 번들의 마지막 옵션에 적용됩니다. 다음 명령줄은 동일합니다.

콘솔

```
myapp -a -b -c arg
myapp -abc arg
```

이 예제의 두 변형에서 인수 `arg` 는 옵션 `-c`에만 적용됩니다.

부울 옵션(플래그)

인수를 가진 `true` 옵션에 `false` 또는 `bool`가 전달되면, 예상대로 구문 분석됩니다. 그러나 인수 형식이 `bool`인 옵션은 일반적으로 인수를 지정할 필요가 없습니다. 부울 옵션("flags"라고도 함)

은 일반적으로 항수가 0입니다ZeroOrOne. 명령줄에 옵션 이름이 있고 그 뒤에 인수가 없으면 기본값은 `true`입니다. 명령줄 입력에 옵션 이름이 없을 경우 값은 `false`로 설정됩니다. 명령이 `myapp` 불리언 옵션 `--interactive`의 값을 출력하면, 다음 입력은 다음과 같은 결과를 만듭니다.

콘솔

```
myapp
myapp --interactive
myapp --interactive false
myapp --interactive true
```

출력

```
False
True
False
True
```

버전 옵션

빌드된 `System.CommandLine` 앱은 루트 명령과 함께 사용되는 옵션에 대한 응답으로 `--version` 버전 번호를 자동으로 제공합니다. 다음은 그 예입니다.

.NET CLI

```
dotnet --version
```

출력

```
6.0.100
```

응답 파일

*응답 파일*은 명령줄 앱에 대한 토큰 집합을 포함하는 파일입니다. 응답 파일은 다음 두 가지 시나리오에서 유용한 기능 `System.CommandLine` 입니다.

- 터미널의 문자 제한보다 긴 입력을 지정하여 명령줄 앱을 호출합니다.
- 전체 줄을 다시 지정하지 않고 동일한 명령을 반복적으로 호출합니다.

응답 파일을 사용하려면 줄 어디에 명령, 옵션 및 인수를 삽입할지에 관계없이 파일 이름 앞에 `@` 기호를 붙여 입력합니다. *.rsp* 파일 확장명은 일반적인 규칙이지만 파일 확장자를 사용할 수 있습니다.

다음 줄은 동일합니다.

.NET CLI

```
dotnet build --no-restore --output ./build-output/
dotnet @sample1.rsp
dotnet build @sample2.rsp --output ./build-output/
```

*sample1.rsp*의 내용:

콘솔

```
build
--no-restore
--output
./build-output/
```

*sample2.rsp*의 내용:

콘솔

```
--no-restore
```

응답 파일의 텍스트를 해석하는 방법을 결정하는 구문 규칙은 다음과 같습니다.

- 토큰은 공백으로 구분됩니다. *안녕하세요!*를 포함하는 줄은 두 개의 토큰인 *Good*과 *morning!*으로 처리됩니다.
- 따옴표로 묶인 여러 토큰은 단일 토큰으로 해석됩니다. "*안녕하세요!*"가 포함된 줄은 하나의 토큰인 *안녕하세요!*로 처리됩니다.
- 기호와 줄 끝 사이의 `#` 모든 텍스트는 주석으로 처리되고 무시됩니다.
- 접두사 `@`가 붙은 토큰은 추가 응답 파일을 참조할 수 있습니다.
- 응답 파일에는 여러 줄의 텍스트가 있을 수 있습니다. 줄은 연결되고 토큰 시퀀스로 해석됩니다.

지침

`System.CommandLine`는 형식으로 표현되는 Directive이라는 구문 요소를 소개합니다. 예를 들어 지시[diagram]문은 기본 제공 지시문입니다. 앱 이름 `[diagram]` 뒤를 포함하는 `System.CommandLine` 경우 명령줄 앱을 호출하는 대신 구문 분석 결과의 다이어그램을 표시합니다.

.NET CLI

```
dotnet [diagram] build --no-restore --output ./build-output/
       ^-----^
```

출력

```
[ dotnet [ build [ --no-restore <True> ] [ --output <./build-output/> ] ] ]
```

지시자의 목적은 명령줄 앱 전반에 적용할 수 있는 범용 기능을 제공하는 것입니다. 지시문은 앱의 구문과 구문적으로 구별되므로 앱에 적용되는 기능을 제공할 수 있습니다.

지시문은 다음 구문 규칙을 준수해야 합니다.

- 앱 이름 뒤의 하위 명령이나 옵션 앞에 오는 명령줄의 토큰입니다.
- 대괄호로 묶입니다.
- 공백을 포함하지 않습니다.

인식할 수 없는 지시문은 구문 분석 오류를 일으키지 않고 무시됩니다.

지시문에는 지시문 이름과 콜론으로 구분된 인수가 포함될 수 있습니다.

다음 지시문이 기본 제공됩니다.

- [diagram]
- [suggest]

`[diagram]` 지시문

사용자와 개발자 모두 앱이 지정된 입력을 해석하는 방법을 확인하는 것이 유용할 수 있습니다. `System.CommandLine` 앱의 기본 기능 중 하나는 명령 입력을 해석한 결과를 미리 볼 수 있게 하는 `[diagram]` 지시어입니다. 다음은 그 예입니다.

콘솔

```
myapp [diagram] --delay not-an-int --interactive --file filename.txt extra
```

출력

```
![ myapp [ --delay !<not-an-int> ] [ --interactive <True> ] [ --file <filename.txt>
] *[ --fgcolor <White> ] ]   ???--> extra
```

앞의 예에서:

- 명령(`myapp`), 해당 자식 옵션 및 해당 옵션에 대한 인수는 대괄호를 사용하여 그룹화됩니다.

- 옵션 결과의 `[ --delay !<not-an-int> ]!` 경우 구문 분석 오류를 나타냅니다. `not-an-int` 옵션의 값 `int`은 예상 형식으로 변환할 수 없습니다. 오류 옵션이 포함된 명령 앞에는 `!`플래그가 지정됩니다: `![ myapp...`.
- 옵션 결과의 `*[ --fgcolor <White> ]`경우 명령줄에 옵션이 지정되지 않았으므로 구성된 기본값이 사용되었습니다. `White` 는 이 옵션의 유효 값입니다. 별표는 값이 기본값임을 나타냅니다.
- `???-->` 는 앱의 명령 또는 옵션과 일치하지 않는 입력을 가리킵니다.

제안 지시문

지시 `[suggest]` 문을 사용하면 정확한 명령을 모르는 경우 명령을 검색할 수 있습니다.

.NET CLI

```
dotnet [suggest] buil
```

출력

```
build
build-server
msbuild
```

참고하십시오

- 디자인 지침
- 오픈 소스 CLI 디자인 지침
- GNU 표준
- System.CommandLine 개요
- 자습서: 시작하기 System.CommandLine

Last updated on 2025. 12. 23.

System.CommandLine에서 구문 분석 및 호출하기

System.CommandLine 는 명령줄 구문 분석과 작업 호출 간의 명확한 구분을 제공합니다. *구문 분석 프로세스*는 명령줄 입력을 구문 분석하고 구문 분석된 값(및 구문 분석 오류)이 포함된 개체를 만드는 ParseResult 작업을 담당합니다. *작업 호출 프로세스*는 구문 분석된 명령, 옵션 또는 지시문과 연결된 작업을 호출합니다(인수에는 작업이 있을 수 없음).

자습서 시작 System.CommandLine `ParseResult` 의 다음 예제에서는 명령줄 입력을 구문 분석하여 만듭니다. 어떤 작업도 정의되거나 호출되지 않습니다.

C#

```
using System.CommandLine;
using System.CommandLine.Parsing;

namespace scl;

class Program
{
    static int Main(string[] args)
    {
        Option<FileInfo> fileOption = new("--file")
        {
            Description = "The file to read and display on the console."
        };

        RootCommand rootCommand = new("Sample app for System.CommandLine");
        rootCommand.Options.Add(fileOption);

        ParseResult parseResult = rootCommand.Parse(args);
        if (parseResult.Errors.Count == 0 && parseResult.GetValue(fileOption) is
FileInfo parsedFile)
        {
            ReadFile(parsedFile);
            return 0;
        }
        foreach (ParseError parseError in parseResult.Errors)
        {
            Console.Error.WriteLine(parseError.Message);
        }
        return 1;
    }

    static void ReadFile(FileInfo file)
    {
        foreach (string line in File.ReadLines(file.FullName))
        {
            Console.WriteLine(line);
        }
```

```
    }
}
```

지정된 명령(또는 지시문 또는 옵션)이 성공적으로 구문 분석될 때 작업이 호출됩니다. 이 작업은 인수를 `ParseResult` 사용하고 종료 코드를 반환하는 `int` 대리자입니다(비동기 작업 도 사용할 수 있음). 종료 코드는 메서드에서 ParseResult.Invoke(InvocationConfiguration) 반환되며 명령이 성공적으로 실행되었는지 여부를 나타내는 데 사용할 수 있습니다.

자습서 시작 System.CommandLine 의 다음 예제에서 작업은 루트 명령에 대해 정의되고 명령줄 입력을 구문 분석한 후 호출됩니다.

C#

```
using System.CommandLine;

namespace scl;

class Program
{
    static int Main(string[] args)
    {
        Option<FileInfo> fileOption = new("--file")
        {
            Description = "The file to read and display on the console."
        };

        RootCommand rootCommand = new("Sample app for System.CommandLine");
        rootCommand.Options.Add(fileOption);

        rootCommand.SetAction(parseResult =>
        {
            FileInfo parsedFile = parseResult.GetValue(fileOption);
            ReadFile(parsedFile);
            return 0;
        });

        ParseResult parseResult = rootCommand.Parse(args);
        return parseResult.Invoke();
    }

    static void ReadFile(FileInfo file)
    {
        foreach (string line in File.ReadLines(file.FullName))
        {
            Console.WriteLine(line);
        }
    }
}
```

일부 기본 제공 기호(예: HelpOption, VersionOption및 SuggestDirective)는 미리 정의된 작업과 함께 제공됩니다. 이러한 기호는 만들 때 루트 명령에 자동으로 추가되고 호출할 ParseResult때

"작동"합니다. 작업을 사용하면 앱 논리에 집중할 수 있으며 라이브러리는 기본 제공 기호에 대한 작업 구문 분석 및 호출을 처리합니다. 원하는 경우 구문 분석 프로세스를 고수하고 작업을 정의하지 않을 수 있습니다(이 문서의 첫 번째 예제와 같이).

`ParseResult`

이 클래스는 ParseResult 명령줄 입력을 구문 분석한 결과를 나타냅니다. 옵션 및 인수에 대한 구문 분석된 값을 가져오는 데 사용해야 합니다(작업을 사용하는지 여부에 관계없이). 구문 분석 오류 또는 일치하지 않는 토큰이 있는지 확인할 수도 있습니다.

값 가져오기

이 ParseResult.GetValue 메서드를 사용하면 옵션 및 인수의 값을 검색할 수 있습니다.

C#

```
int integer = parseResult.GetValue(delayOption);
string? message = parseResult.GetValue(messageOption);
```

이름으로 값을 가져올 수도 있지만, 이를 위해서는 가져올 값의 형식을 지정해야 합니다.

다음 예제에서는 C# 컬렉션 이니셜라이저를 사용하여 루트 명령을 만듭니다.

C#

```
RootCommand rootCommand = new("Parameter binding example")
{
    new Option<int>("--delay")
    {
        Description = "An option whose argument is parsed as an int."
    },
    new Option<string>("--message")
    {
        Description = "An option whose argument is parsed as a string."
    }
};
```

그런 다음, 메서드를 `GetValue` 사용하여 이름으로 값을 가져옵니다.

C#

```
rootCommand.SetAction(parseResult =>
{
    int integer = parseResult.GetValue<int>("--delay");
    string? message = parseResult.GetValue<string>("--message");
```

```
    DisplayIntAndString(integer, message);
});
```

이 오버로드 기능은 구문 분석된 명령의 컨텍스트에서 `GetValue`를 사용하여 지정된 기호 이름의 구문 분석된 값 또는 기본값을 가져옵니다 (전체 기호 트리가 아님). 별칭이 아닌 기호 이름을 허용합니다.

구문 분석 오류

이 속성에는 ParseResult.Errors 구문 분석 프로세스 중에 발생한 구문 분석 오류 목록이 포함되어 있습니다. 각 오류는 오류 메시지 및 오류를 발생시킨 토큰과 같은 오류에 대한 정보를 포함하는 개체로 표시됩니다 ParseError .

메서드를 ParseResult.Invoke(InvocationConfiguration) 호출하면 구문 분석이 성공했는지 여부를 나타내는 종료 코드가 반환됩니다. 구문 분석 오류가 있는 경우 종료 코드는 0이 아니고 모든 구문 분석 오류가 표준 오류로 출력됩니다.

메서드를 `ParseResult.Invoke` 호출하지 않는 경우 오류를 인쇄하여 직접 처리해야 합니다.

C#

```
foreach (ParseError parseError in parseResult.Errors)
{
    Console.Error.WriteLine(parseError.Message);
}
return 1;
```

일치하지 않는 토큰

이 속성에는 UnmatchedTokens 구문 분석되었지만 구성된 명령, 옵션 또는 인수와 일치하지 않는 토큰 목록이 포함되어 있습니다.

일치하지 않는 토큰 목록은 래퍼와 같은 방식으로 작동하는 명령에 유용합니다. 래퍼 명령은 토큰 집합을 가져와서 다른 명령 또는 앱으로 전달합니다. `sudo` Linux의 명령이 예제입니다. 사용자의 이름을 입력하여 해당 사용자로 가장한 후 명령을 실행합니다. 예를 들어 다음 명령은 사용자`apt update`로 명령을 실행 `admin` 합니다.

콘솔

```
sudo -u admin apt update
```

이와 같은 래퍼 명령을 구현하려면 명령 속성을 `System.CommandLine.Command.TreatUnmatchedTokensAsErrors`에서 `false`로 설정합니다. `System.CommandLine.Parsing.ParseResult.UnmatchedTokens` 그런 다음 이 속성에는 명령에 명시

적으로 속하지 않는 모든 인수가 포함됩니다. 앞의 예제에서 `ParseResult.UnmatchedTokens`는 `apt` 및 `update` 토큰을 포함합니다.

활동

작업은 명령(또는 옵션 또는 지시문)이 성공적으로 파싱될 때 호출되는 대리자입니다. 인수를 사용하고 ParseResult 종료 코드를 반환 `int` `Task<int>`합니다. 종료 코드는 작업이 성공적으로 실행되었는지 여부를 나타내는 데 사용됩니다.

System.CommandLine는 추상 기본 클래스 CommandLineAction 와 두 개의 파생 클래스를 제공합니다.SynchronousCommandLineActionAsynchronousCommandLineAction 전자는 종료 코드를 반환 `int` 하는 동기 작업에 사용되고, 후자는 종료 코드를 반환 `Task<int>` 하는 비동기 작업에 사용됩니다.

동작을 정의하기 위해 파생 형식을 만들 필요가 없습니다. 이 메서드를 SetAction 사용하여 명령에 대한 작업을 설정할 수 있습니다. 동기 작업은 인수를 사용하고 `ParseResult` 종료 코드를 반환하는 대리자 `int` 일 수 있습니다. 비동기 작업은 인수를 `ParseResult` 사용하고 CancellationToken 반환`Task<int>`하는 대리자일 수 있습니다.

C#

```
rootCommand.SetAction(parseResult =>
{
    FileInfo parsedFile = parseResult.GetValue(fileOption);
    ReadFile(parsedFile);
    return 0;
});
```

비동기 작업

동기 및 비동기 작업은 동일한 애플리케이션에서 혼합해서는 안 됩니다. 비동기 작업을 사용하려면 애플리케이션이 전체에서 비동기적이어야 합니다. 즉, 모든 작업은 비동기이어야 하며, 종료 코드를 반환하는 대리자를 허용하는 `SetAction` 메서드를 `Task<int>` 사용해야 합니다. 또한 CancellationToken 작업 대리자로 전달되는 작업은 파일 I/O 작업 또는 네트워크 요청과 같이 취소할 수 있는 모든 메서드에 추가로 전달되어야 합니다.

또한 메서드가 ParseResult.InvokeAsync(InvocationConfiguration, CancellationToken) 대신 사용되는지 확인해야 합니다 `Invoke`. 이 메서드는 비동기이며 종료 코드를 반환합니다 `Task<int>` . 또한 작업을 취소하는 데 사용할 수 있는 선택적 CancellationToken 매개 변수도 허용합니다.

다음 코드는 `SetAction`를 가져오는 오버로드와 단순한 CancellationToken오버로드를 사용합니다`ParseResult`.

C#

```
static Task<int> Main(string[] args)
{
    Option<string> urlOption = new("--url")
    {
        Description = "A URL."
    };
    RootCommand rootCommand = new("Handle termination example") { urlOption };

    rootCommand.SetAction((ParseResult parseResult, CancellationToken
cancellationToken) =>
    {
        string? urlOptionValue = parseResult.GetValue(urlOption);
        return DoRootCommand(urlOptionValue, cancellationToken);
    });

    return rootCommand.Parse(args).InvokeAsync();
}

public static async Task<int> DoRootCommand(
    string? urlOptionValue, CancellationToken cancellationToken)
{
    using HttpClient httpClient = new();

    try
    {
        await httpClient.GetAsync(urlOptionValue, cancellationToken);
        return 0;
    }
    catch (OperationCanceledException)
    {
        await Console.Error.WriteLineAsync("The operation was aborted");
        return 1;
    }
}
```

프로세스 종료 시간 제한

ProcessTerminationTimeout는 호출 중에 모든 비동기 작업에 전달되는 프로세스를 통해 프로세스 종료(+ `SIGINT`, `SIGTERM`, CancellationToken)의 신호 및 처리를 가능하게 합니다. 기본적으로 사용하도록 설정되어 있으며(2초), 사용하지 않도록 설정하려면 `null`로 설정할 수 있습니다.

사용하도록 설정하면 지정된 시간 제한 내에 작업이 완료되지 않으면 프로세스가 종료됩니다. 예를 들어 프로세스가 종료되기 전에 상태를 저장하여 종료를 정상적으로 처리하는 데 유용합니다.

이전 단락의 샘플 코드를 테스트하려면 로드하는 데 시간이 걸리는 URL을 사용하여 명령을 실행하고 로드가 완료되기 전에 `Ctrl` + `C`를 누릅니다. macOS에서 `명령` + `기간(.)`을 누릅니다. 다음은 그 예입니다.

.NET CLI

```
testapp --url https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis
```

출력

```
The operation was aborted
```

종료 코드

종료 코드는 성공 또는 실패를 나타내는 작업에서 반환되는 정수 값입니다. 규칙에 따라 종료 코드는 `0` 성공을 의미하지만 0이 아닌 값은 오류를 나타냅니다. 명령 실행 상태를 명확하게 전달하려면 애플리케이션에서 의미 있는 종료 코드를 정의하는 것이 중요합니다.

모든 `SetAction` 메서드는 종료 코드를 명시적으로 제공해야 하는 대리자를 수락하는 오버로드와 `int`를 반환하는 오버로드를 제공합니다.

C#

```
static int Main(string[] args)
{
    Option<int> delayOption = new("--delay");
    Option<string> messageOption = new("--message");

    RootCommand rootCommand = new("Parameter binding example")
    {
        delayOption,
        messageOption
    };

    rootCommand.SetAction(parseResult =>
    {
        Console.WriteLine($"--delay = {parseResult.GetValue(delayOption)}");
        Console.WriteLine($"--message = {parseResult.GetValue(messageOption)}");
        // Value returned from the action delegate is the exit code.
        return 100;
    });

    return rootCommand.Parse(args).Invoke();
}
```

참고하십시오

- 에서 구문 분석 및 유효성 검사를 사용자 지정하는 방법 System.CommandLine
- System.CommandLine 개요

Last updated on 2025. 12. 05.

에서 구문 분석 및 유효성 검사를 사용자 지정하는 방법 System.CommandLine

기본적으로 System.CommandLine 다음과 같은 여러 일반적인 형식을 구문 분석할 수 있는 기본 제공 파서 집합을 제공합니다.

- `bool`
- `byte` 및 `sbyte`
- `short` 및 `ushort`
- `int` 및 `uint`
- `long` 및 `ulong`
- `float` 및 `double`
- `decimal`
- `DateTime` 및 `DateTimeOffset`
- `DateOnly`그리고 `TimeOnly`
- `Guid`
- FileSystemInfo, FileInfo 및 DirectoryInfo
- 열거형
- **나열된 형식의 배열 및 목록**

다른 형식은 지원되지 않지만 사용자 지정 파서를 만들 수 있습니다. 구문 분석된 값의 유효성을 검사할 수도 있습니다. 이 값은 입력이 특정 조건을 충족하는지 확인하려는 경우에 유용합니다.

유효성 검사기

모든 옵션, 인수 및 명령에는 하나 이상의 유효성 검사기가 있을 수 있습니다. 유효성 검사기는 구문 분석된 값이 특정 조건을 충족하는지 확인하는 데 사용됩니다. 예를 들어 숫자가 양수이거나 문자열이 비어 있지 않은지 확인할 수 있습니다. 여러 조건을 확인하는 복잡한 유효성 검사기를 만들 수도 있습니다.

모든 기호 형식 System.CommandLine 에는 Validators 유효성 검사기 목록이 포함된 속성이 있습니다. 유효성 검사기는 입력을 구문 분석한 후에 실행되며 유효성 검사에 실패하면 오류를 보고할 수 있습니다.

사용자 지정 유효성 검사 코드를 제공하려면 다음 예제와 같이 옵션 또는 인수(또는 명령)를 호출 `System.CommandLine.Option.Validators.Add` 합니다.

C#

```
Option<int> delayOption = new("--delay");
delayOption.Validators.Add(result =>
```

```
{
    if (result.GetValue(delayOption) < 1)
    {
        result.AddError("Must be greater than 0");
    }
});
```

System.CommandLine 에서는 공통 형식의 유효성을 검사하는 데 사용할 수 있는 기본 제공 유효성 검사기 집합을 제공합니다.

- `AcceptExistingOnly` - 기존 파일 또는 디렉터리에 해당하는 값만 허용하도록 지정된 옵션 또는 인수를 구성합니다.
- `AcceptLegalFileNamesOnly` - 유효한 파일 이름을 나타내는 값만 허용하도록 지정된 옵션 또는 인수를 구성합니다.
- `AcceptOnlyFromAmong` - 지정된 값 집합의 값만 허용하도록 지정된 옵션 또는 인수를 구성합니다.

사용자 지정 파서

복합 형식과 같이 기본 파서가 없는 형식을 구문 분석하려면 사용자 지정 파서가 필요합니다. 사용자 지정 파서를 사용하여 지원되는 형식을 기본 제공 파서와 다른 방식으로 구문 분석할 수도 있습니다.

다음과 같은 형식이 있다고 가정합니다.`Person`

C#

```
public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

값을 읽고 명령 작업에서 인스턴스 `Person` 를 만들 수 있습니다.

C#

```
rootCommand.SetAction(parseResult =>
{
    Person person = new()
    {
        FirstName = parseResult.GetValue(firstNameOption),
        LastName = parseResult.GetValue(lastNameOption)
    };
    DoRootCommand(parseResult.GetValue(fileOption), person);
});
```

사용자 지정 파서에서는 기본 값을 가져오는 것과 동일한 방식으로 사용자 지정 형식을 가져올 수 있습니다.

C#

```
Option<Person?> personOption = new("--person")
{
    Description = "An option whose argument is parsed as a Person",
    CustomParser = result =>
    {
        if (result.Tokens.Count != 2)
        {
            result.AddError("--person requires two arguments");
            return null;
        }
        return new Person
        {
            FirstName = result.Tokens.First().Value,
            LastName = result.Tokens.Last().Value
        };
    }
};
```

구문 분석하고 입력의 유효성을 검사하려면 다음 예제와 같이 대리자를 사용합니다 `CustomParser` .

C#

```
Option<int> delayOption = new("--delay")
{
    Description = "An option whose argument is parsed as an int.",
    CustomParser = result =>
    {
        if (!result.Tokens.Any())
        {
            return 42;
        }

        if (int.TryParse(result.Tokens.Single().Value, out var delay))
        {
            if (delay < 1)
            {
                result.AddError("Must be greater than 0");
            }
            return delay;
        }
        else
        {
            result.AddError("Not an int.");
            return 0; // Ignored.
        }
```

```
    }
};
```

다음은 유효성 검사기를 사용해서는 할 수 없는 작업의 예시와 `CustomParser`을 사용해 할 수 있는 작업의 몇 가지 예입니다.

- 다른 종류의 입력 문자열을 구문 분석합니다(예: "1,2,3"을 `int[]`로 변환).
- 동적 입력 수(arity) 예를 들어 문자열 배열로 정의된 두 인수가 있고 명령줄 입력 `System.CommandLine.Parsing.ArgumentResult.OnlyTake` 에서 문자열 시퀀스를 처리해야 하는 경우 이 메서드를 사용하면 입력 문자열을 인수 간에 동적으로 나눌 수 있습니다.

참고하십시오

- 결과를 구문 분석하고 호출하는 방법
- System.CommandLine 개요

Last updated on 2025. 12. 05.

System.CommandLine에서 파서를 구성하는 방법

구문 분석 및 호출은 별도의 두 단계이므로 각 단계마다 고유한 구성이 있습니다.

- ParserConfiguration 는 구문 분석을 구성하는 속성을 제공하는 클래스입니다. 이 인수는 다음과 같은 `Parse` 모든 Command.Parse 메서드에 대한 선택적 인수입니다 CommandLineParser.Parse.
- InvocationConfiguration 는 호출을 구성하는 속성을 제공하는 클래스입니다. 이는 및 ParseResult.Invoke 메서드의 ParseResult.InvokeAsync 선택적 인수입니다.

및 속성에 ParseResult.ConfigurationParseResult.InvocationConfiguration 의해 노출됩니다. 제공되지 않으면 기본 구성이 사용됩니다.

ParserConfiguration

EnablePosixBundling

단일 문자 옵션의 묶음은 기본적으로 사용하도록 설정되어 있지만 속성을 ParserConfiguration.EnablePosixBundling.로 설정 `false` 하여 사용하지 않도록 설정할 수 있습니다.

ResponseFileTokenReplacer

응답 파일은 기본적으로 사용하도록 설정되어 있지만 속성을 ResponseFileTokenReplacer.로 설정 `null` 하여 사용하지 않도록 설정할 수 있습니다. 응답 파일 처리 방법을 사용자 지정하는 사용자 지정 구현을 제공할 수도 있습니다.

응답 파일에는 다른 응답 파일 이름이 포함될 수 있으므로 구문 분석 시 다른 파일 열기가 포함될 수 있습니다. 라이브러리는 모든 응답 파일이 신뢰할 수 있는 에이전트에 의해 생성되고 저장될 것으로 예상합니다.

InvocationConfiguration

표준 출력 및 오류

InvocationConfiguration를 사용하면 `System.Console`를 사용하는 것보다 테스트와 여러 확장성 시나리오를 더 쉽게 수행할 수 있습니다. 두 가지 속성, 즉 `TextWriter` 및 Output을 노출합니다:

Error. 이러한 속성은 테스트용 출력을 캡처하는 데 사용할 수 있는 `TextWriter` 인스턴스, 예를 들어 `StringWriter` 인스턴스에 설정할 수 있습니다.

표준 출력에 쓰는 간단한 명령을 정의합니다.

C#

```
Option<FileInfo?> fileOption = new("--file")
{
    Description = "An option whose argument is parsed as a FileInfo"
};

RootCommand rootCommand = new("Configuration sample")
{
    fileOption
};

rootCommand.SetAction((parseResult) =>
{
    FileInfo? fileOptionValue = parseResult.GetValue(fileOption);
    parseResult.InvocationConfiguration.Output.WriteLine(
        $"File option value: {fileOptionValue?.FullName}"
        );
});
```

이제 출력을 캡처하는 데 사용합니다 InvocationConfiguration .

C#

```
StringWriter output = new();
rootCommand.Parse("-h").Invoke(new() { Output = output });
Debug.Assert(output.ToString().Contains("Configuration sample"));
```

프로세스 종료 시간 초과

프로세스 종료 시간 제한 은 속성을 통해 ProcessTerminationTimeout 구성할 수 있습니다. 기본값은 2초입니다.

기본 예외 처리기 설정

기본적으로 명령어를 호출할 때 발생한 모든 처리되지 않은 예외가 포착되어 사용자에게 보고됩니다. EnableDefaultExceptionHandler 속성을 `false`으로 설정하여 이 동작을 비활성화할 수 있습니다. 이 기능은 예외를 로깅하거나 다른 사용자 환경을 제공하는 등 사용자 지정 방식으로 예외를 처리하려는 경우에 유용합니다.

파생 클래스

InvocationConfiguration 가 봉인되지 않았으므로 사용자 지정 속성 또는 메서드를 추가하기 위해 파생할 수 있습니다. 이는 애플리케이션과 관련된 추가 구성 옵션을 제공하려는 경우에 유용합니다.

참고하십시오

- System.CommandLine 개요

Last updated on 2025. 12. 05.

System.CommandLine에 대한 탭 완성

사용하는 `System.CommandLine` 앱은 특정 셸에서 탭 완성을 기본적으로 지원합니다. 이를 사용하도록 설정하려면 최종 사용자가 셸당 한 번 몇 단계를 수행해야 합니다. 이 작업이 완료되면 열거형 값 또는 호출 AcceptOnlyFromAmong(String[])로 정의된 값과 같은 앱의 정적 값에 대해 탭 완성이 자동으로 수행됩니다. 런타임에 동적으로 값을 제공하여 탭 완성을 사용자 지정할 수도 있습니다.

탭 완성 활성화

사용자(최종 사용자)가 탭 완성을 사용하도록 설정하려는 컴퓨터에서 다음 단계를 수행합니다.

.NET CLI의 경우:

- 탭 완성을 사용하도록 설정하는 방법을 참조하세요.

`System.CommandLine`을(를) 기반으로 구축된 다른 명령줄 앱의 경우:

- 전역 도구 dotnet-suggest를 설치하십시오.

  .NET CLI

  ```
  dotnet tool install -g dotnet-suggest
  ```

- 셸 프로필에 적절한 shim 스크립트를 추가합니다. 셸 프로필 파일을 만들어야 할 수도 있습니다. shim 스크립트는 셸에서의 완료 요청을 `dotnet-suggest` 도구로 전달하고, 이를 적절한 `System.CommandLine` 기반 앱에 위임합니다.

  - 의 경우 `bash` *dotnet-suggest-shim.bash* 의 내용을 *~/.bash_profile* 추가합니다.
  - 의 경우 `zsh` *dotnet-suggest-shim.zsh* 의 내용을 *~/.zshrc*에 추가합니다.
  - PowerShell의 경우 powerShell 프로필에 *dotnet-suggest-shim.ps1* 콘텐츠를 추가한 다음 PowerShell을 다시 시작합니다. 다음 명령을 사용하여 PowerShell 프로필에 대한 예상 경로를 찾을 수 있습니다.

    콘솔

    ```
    echo $profile
    ```

- 앱의 실행 파일 경로는 어디에 있는지 `dotnet-suggest register --command-path $executableFilePath` 호출`$executableFilePath`하여 앱을 등록합니다.

사용자의 셸이 설정되고 실행 파일이 등록되면 `System.CommandLine`을 사용하여 빌드된 모든 앱에서 자동 완성이 작동합니다.

Windows의 *cmd.exe* (Windows 명령 프롬프트)에서는 확장 가능한 탭 완성 메커니즘이 없으므로 shim 스크립트를 사용할 수 없습니다. 다른 셸의 경우 레이블이 지정된 GitHub 문제를 찾습니다 Area-Completions. 문제를 찾지 못하면 새 문제를 열 수 있습니다.

런타임에 탭 완성 값 가져오기

다음 코드는 런타임에 탭 완성에 대한 값을 동적으로 검색하는 앱을 보여 줍니다. 이 코드는 현재 날짜 이후의 다음 주 날짜 목록을 가져옵니다. `--date`을(를) 호출하여 목록이 `CompletionSources.Add` 옵션에 제공됩니다.

C#

```
using System.CommandLine;
using System.CommandLine.Completions;
using System.CommandLine.Parsing;

new DateCommand().Parse(args).Invoke();

class DateCommand : Command
{
    private Argument<string> subjectArgument = new("subject")
    {
        Description = "The subject of the appointment."
    };
    private Option<DateTime> dateOption = new("--date")
    {
        Description = "The day of week to schedule. Should be within one week."
    };

    public DateCommand() : base("schedule", "Makes an appointment for sometime in
the next week.")
    {
        this.Arguments.Add(subjectArgument);
        this.Options.Add(dateOption);

        dateOption.CompletionSources.Add(ctx => {
            var today = System.DateTime.Today;
            List<CompletionItem> dates = new();
            foreach (var i in Enumerable.Range(1, 7))
            {
                var date = today.AddDays(i);
                dates.Add(new CompletionItem(
                    label: date.ToShortDateString(),
                    sortText: $"{i:2}"));
            }
            return dates;
        });
```

```
        this.SetAction(parseResult =>
        {
            Console.WriteLine($"Scheduled \"
{parseResult.GetValue(subjectArgument)}\" for {parseResult.GetValue(dateOption)}");
        });
    }
}
```

`Tab` 키를 누를 때 표시되는 값은 인스턴스로 CompletionItem 제공됩니다.

C#

```
dates.Add(new CompletionItem(
    label: date.ToShortDateString(),
    sortText: $"{i:2}"));
```

다음 `CompletionItem` 속성이 설정됩니다.

- Label 는 표시할 완료 값입니다.
- SortText 는 목록의 값이 올바른 순서로 표시되는지 확인합니다. `i`을 두 자리 문자열로 변환하여 설정되면, 정렬은 01, 02, 03, ... 14에 따라 이루어집니다. 이 매개 변수를 설정하지 않으면, 정렬은 기준인 `Label`를 기반으로 수행됩니다. 이 예제에서 `Label`는 짧은 날짜 형식이므로 올바르게 정렬되지 않습니다.

`CompletionItem` 속성들(예: `Documentation` 및 `Detail`)이 있지만, `System.CommandLine`에서는 아직 사용되지 않습니다.

이 코드에서 만든 동적 탭 완성 목록도 도움말 출력에 표시됩니다.

출력

```
Description:
  Makes an appointment for sometime in the next week.

Usage:
  schedule <subject> [options]

Arguments:
  <subject>  The subject of the appointment.

Options:
  --date
The day of week to schedule. Should be within one week.
  <12/4/2025|12/5/2025|12/6/2025|12/7/2025|12/8/2025|12/9/2025|12/10/2025>
  --version
Show version information
  -?, -h, --help
```

참고하십시오

- System.CommandLine 개요
- .NET CLI에 대해 탭 완성을 사용하도록 설정하는 방법

Last updated on 2025. 12. 18.

도움말 출력 사용자 지정

2025. 08. 13.

명령줄 앱은 일반적으로 사용 가능한 명령, 옵션 및 인수에 대한 간략한 설명을 표시하는 옵션을 제공합니다. 는 기본적으로 RootCommand 옵션에 포함된 a0>을 제공합니다. `HelpOption`는 에 의해 노출되는 정보 및 기본값 또는 완료 원본과 같은 기타 속성을 사용하여 정의된 기호에 NameHelpNameDescription대한 도움말 출력을 생성합니다.

C#

```
Option<FileInfo> fileOption = new("--file")
{
    Description = "The file to print out.",
};
Option<bool> lightModeOption = new("--light-mode")
{
    Description = "Determines whether the background color will be black or white"
};
Option<ConsoleColor> foregroundColorOption = new("--color")
{
    Description = "Specifies the foreground color of console output",
    DefaultValueFactory = _ => ConsoleColor.White
};

RootCommand rootCommand = new("Read a file")
{
    fileOption,
    lightModeOption,
    foregroundColorOption
};
rootCommand.Parse("-h").Invoke();
```

.NET CLI

```
Description:
  Read a file

Usage:
  scl [options]

Options:
  -?, -h, --help                                                  Show help and usage
information
  --version                                                       Show version
information
  --file                                                          The file to print out.
  --light-mode                                                    Determines whether the
background color will be black

                                                                  or white
  --color                                                         Specifies the
```

```
foreground color of console output
  <Black|Blue|Cyan|DarkBlue|DarkCyan|DarkGray|DarkGreen|Da  [default: White]
  rkMagenta|DarkRed|DarkYellow|Gray|Green|Magenta|Red|Whit
  e|Yellow>
```

앱 사용자는 다양한 플랫폼에서 도움을 요청하는 다양한 방법에 익숙할 수 있으므로 기본 제공되는 `System.CommandLine` 앱은 여러 가지 방법으로 도움을 요청합니다. 다음 명령은 모두 동일합니다.

.NET CLI

```
dotnet --help
dotnet -h
dotnet /h
dotnet -?
dotnet /?
```

도움말 출력에 사용 가능한 명령, 인수 및 옵션이 모두 표시되지는 않습니다. 그 중 일부는 속성을 통해 Hidden질 수 있습니다. 즉, 도움말 출력(및 완료)에 표시되지 않지만 명령줄에 지정할 수 있습니다.

도움말 사용자 지정

각 기호에 대한 특정 도움말 텍스트를 정의하여 명령에 대한 도움말 출력을 사용자 지정하여 사용량에 대해 사용자에게 더 명확하게 제공할 수 있습니다.

옵션 인수의 이름을 사용자 지정하려면 옵션의 `System.CommandLine.Option.HelpName` 속성을 사용합니다.

샘플 앱의 `--light-mode`은 적절하게 설명되지만, `--file` 및 `--color` 옵션 설명의 변경이 도움이 될 것입니다. `--file`의 경우, 인수는 `<FILEPATH>`로 식별할 수 있습니다. 이 옵션의 `--color` 경우 사용 가능한 색 목록을 줄일 수 있습니다.

이러한 변경을 수행하려면 다음 코드를 사용하여 이전 코드를 확장합니다.

C#

```
fileOption.HelpName = "FILEPATH";
foregroundColorOption.AcceptOnlyFromAmong(
    ConsoleColor.Black.ToString(),
    ConsoleColor.White.ToString(),
    ConsoleColor.Red.ToString(),
    ConsoleColor.Yellow.ToString()
);
```

이제 앱은 다음과 같은 도움말 출력을 생성합니다.

출력

```
Description:
  Read a file

Usage:
  scl [options]

Options:
  -?, -h, --help                    Show help and usage information
  --version                         Show version information
  --file <FILEPATH>                 The file to print out.
  --light-mode                      Determines whether the background color will
be black or white
  --color <Black|Red|White|Yellow>  Specifies the foreground color of console
output [default: White]
```

출력에 도움이 되는 섹션 추가

도움말 출력에 첫 번째 또는 마지막 섹션을 추가할 수 있습니다. 예를 들어 Spectre.Console NuGet 패키지를 사용하여 설명 섹션에 ASCII 아트를 추가하려는 경우를 가정해 보겠습니다.

기본값 `HelpAction`을 호출하기 전과 후에 몇 가지 추가 논리를 수행하는 사용자 지정 작업을 정의합니다.

C#

```
internal class CustomHelpAction : SynchronousCommandLineAction
{
    private readonly HelpAction _defaultHelp;

    public CustomHelpAction(HelpAction action) => _defaultHelp = action;

    public override int Invoke(ParseResult parseResult)
    {
        Spectre.Console.AnsiConsole.Write(new
FigletText(parseResult.RootCommandResult.Command.Description!));

        int result = _defaultHelp.Invoke(parseResult);

        Spectre.Console.AnsiConsole.WriteLine("Sample usage: --file input.txt");

        return result;

    }
}
```

`HelpAction`에 의해 정의된 `RootCommand`를 사용자 지정 작업을 사용하도록 업데이트합니다.

C#

```
for (int i = 0; i < rootCommand.Options.Count; i++)
{
    // RootCommand has a default HelpOption; update its Action.
    if (rootCommand.Options[i] is HelpOption defaultHelpOption)
    {
        defaultHelpOption.Action = new
CustomHelpAction((HelpAction)defaultHelpOption.Action!);
        break;
    }
}
```

이제 도움말 출력은 다음과 같습니다.

출력

```
  ____                        _                 __   _   _
 |  _ \    ___    __ _    __| |     __ _      / _| (_) | |   ___
 | |_) |  / _ \  / _` |  / _` |    / _` |    | |_  | | | |  / _ \
 |  _ <  |  __/ | (_| | | (_| |   | (_| |    |  _| | | | | |  __/
 |_| \_\  \___|  \__,_|  \__,_|    \__,_|    |_|   |_| |_|  \___|

Description:
  Read a file

Usage:
  scl [options]

Options:
  -?, -h, --help                    Show help and usage information
  --version                         Show version information
  --file <FILEPATH>                 The file to print out.
  --light-mode                      Determines whether the background color will
be black or white
  --color <Black|Red|White|Yellow>  Specifies the foreground color of console
output [default: White]

Sample usage: --file input.txt
```

참고하십시오

System.CommandLine 개요

디자인 지침

2025. 06. 19.

다음 섹션에서는 저희가 권장하는 CLI 디자인 지침을 제공합니다. 명령줄에서 앱이 기대하는 것은 REST API 서버가 URL에 기대하는 것과 비슷하다고 생각합니다. REST API에 대한 일관된 규칙은 클라이언트 앱 개발자에게 쉽게 사용할 수 있도록 하는 것입니다. 마찬가지로 CLI 디자인이 일반적인 패턴을 따르는 경우 명령줄 앱의 사용자는 더 나은 환경을 갖습니다.

CLI를 만든 후에는 특히 사용자가 계속 실행해야 하는 스크립트에서 CLI를 사용한 경우 변경하기가 어렵습니다. 이 지침은 .NET CLI 이후 개발되었으며 항상 이러한 지침을 따르지는 않습니다. 호환성이 손상되는 변경 사항을 도입하지 않고 수행할 수 있는 .NET CLI를 업데이트하고 있습니다. 이 작업의 예로 .NET 7의 새 디자인 `dotnet new` 이 있습니다.

기호

명령 및 하위 명령

명령에 하위 명령이 있는 경우 명령은 작업을 지정하는 대신 하위 명령의 영역 또는 그룹화 식별자로 작동해야 합니다. 앱을 호출할 때 그룹화 명령과 해당 하위 명령 중 하나를 지정합니다. 예를 들어, `dotnet tool`을(를) 실행하려고 하면 오류 메시지가 표시됩니다. 이는 `tool` 명령이 `install`, `list`와 같은 도구 관련 하위 명령 그룹만 식별하기 때문입니다. 실행할 `dotnet tool install`수 있지만 `dotnet tool` 그 자체로는 불완전합니다.

영역을 정의하는 것이 사용자에게 도움이 되는 방법 중 하나는 도움말 출력을 구성하는 것입니다.

CLI 내에는 암시적 영역이 있는 경우가 많습니다. 예를 들어 .NET CLI에서 암시적 영역은 프로젝트이며 Docker CLI에서는 이미지입니다. 따라서 영역을 포함하지 않고 사용할 `dotnet build` 수 있습니다. CLI에 암시적 영역이 있는지 여부를 고려합니다. 사용자가 `docker build` 및 `docker image build`를 선택적으로 포함하거나 생략할 수 있도록 허용할지 여부를 고려하십시오. 사용자가 암시적 영역을 입력하도록 선택적으로 허용하는 경우 이 명령 그룹화에 대한 도움말 및 탭 완성도 자동으로 설정됩니다. 동일한 작업을 수행하는 두 명령을 정의하여 암시적 그룹의 선택적 사용을 제공합니다.

매개 변수로서의 옵션

옵션은 작업 자체를 지정하는 대신 명령에 매개 변수를 제공해야 합니다. 항상 뒤에 오는 `System.CommandLine` 것은 아니지만 권장되는 디자인 원칙입니다(`--help` 도움말 정보를 표시).

이름 지정

짧은 형태 별칭

일반적으로 정의하는 짧은 형식 옵션 별칭의 수를 최소화하는 것이 좋습니다.

특히 .NET CLI 및 기타 .NET 명령줄 앱에서 일반적인 사용과 다르게 다음 별칭을 사용하지 않도록 합니다.

- `-i` 을 선택합니다 `--interactive`.

  이 옵션은 명령이 응답해야 하는 질문에 대한 입력을 묻는 메시지가 표시될 수 있음을 사용자에게 알릴 수 있습니다. 예를 들어 사용자 이름을 묻는 메시지가 표시됩니다. CLI는 스크립트에서 사용될 수 있으므로 이 스위치를 지정하지 않은 사용자에게 메시지를 표시할 때는 주의해야 합니다.

- `-o` 을 선택합니다 `--output`.

  일부 명령은 실행 결과로 파일을 생성합니다. 이 옵션은 해당 파일을 배치해야 하는 위치를 결정하는 데 사용해야 합니다. 단일 파일이 만들어지는 경우 이 옵션은 파일 경로여야 합니다. 많은 파일이 만들어지는 경우 이 옵션은 디렉터리 경로여야 합니다.

- `-v` 을 선택합니다 `--verbosity`.

  명령은 종종 콘솔에서 사용자에게 출력을 제공합니다. 이 옵션은 사용자가 요청하는 출력의 양을 지정하는 데 사용됩니다. 자세한 내용은 이 문서의 뒷부분에 있는 옵션을 --verbosity 참조하세요.

일반적인 사용량이 .NET CLI로 제한되는 일부 별칭도 있습니다. 이러한 별칭을 앱의 다른 옵션에 사용할 수 있지만 혼동의 가능성을 알고 있어야 합니다.

- `-c`에 대한 `--configuration`

  이 옵션은 명명된 빌드 구성(예 `Debug` : 또는 `Release`.)을 참조하는 경우가 많습니다. 구성에 원하는 이름을 사용할 수 있지만 대부분의 도구에는 이러한 이름 중 하나가 예상됩니다. 이 설정은 구성을 빌드할 때 `Debug` 코드 최적화를 줄이는 등 해당 구성에 적합한 방식으로 다른 속성을 구성하는 데 자주 사용됩니다. 명령에 다른 작업 모드가 있는 경우 이 옵션을 고려합니다.

- `-f`에 대한 `--framework`

  이 옵션은 실행할 단일 TFM(대상 프레임워크 모니커) 을 선택하는 데 사용되므로 CLI 애플리케이션이 선택한 TFM에 따라 다른 동작이 있는 경우 이 플래그를 지원해야 합니다.

- `-p`에 대한 `--property`

  애플리케이션이 결국 MSBuild를 호출하는 경우 사용자는 어떤 식으로든 해당 호출을 사용자 지정해야 하는 경우가 많습니다. 이 옵션을 사용하면 명령줄에서 MSBuild 속성을 제공하고 기본 MSBuild 호출에 전달할 수 있습니다. 앱에서 MSBuild를 사용하지 않지만 키-값 쌍 집합이 필요한 경우 동일한 옵션 이름을 사용하여 사용자의 기대치를 활용하는 것이 좋습니다.

- `-r`에 대한 `--runtime`

  애플리케이션이 다른 런타임에서 실행되거나 런타임 관련 논리가 있는 경우 런타임 식별자를 지정하는 방법으로 이 옵션을 지원하는 것이 좋습니다. 앱이 `--runtime`를 지원하는 경우, `--os`와 `--arch`도 지원하는 것을 고려하십시오. 이러한 옵션을 사용하면 RID의 OS 또는 아키텍처 부분만 지정할 수 있으며, 지정되지 않은 부분은 현재 플랫폼을 기준으로 자동으로 결정됩니다. 자세한 내용은 dotnet publish를 참조하세요.

짧은 이름

명령, 옵션 및 인수의 이름을 최대한 짧고 쉽게 철자할 수 있도록 합니다. 예를 들어 충분히 명확한 경우 `class` 명령을 `classification`만들지 마세요.

이름을 소문자로 표기

소문자만으로 이름을 정의하십시오. 명령이나 옵션을 대/소문자를 구분하지 않도록 하기 위해 대문자 별칭을 만들 수 있습니다.

케밥 케이스 이름

케밥 케이스를 사용하여 단어를 구분합니다. 예: `--additional-probing-path`.

복수화

앱 내에서 복수화에서 일관성을 유지합니다. 예를 들어 여러 값을 가질 수 있는 옵션에 복수 및 단수 이름을 혼합하지 마세요(최대 크기가 1보다 큼).

테이블 확장

옵션 이름	일관성
`--additional-probing-paths` 및 `--sources`	✔
`--additional-probing-path` 및 `--source`	✔

옵션 이름	일관성
`--additional-probing-paths` 및 `--source`	❌
`--additional-probing-path` 및 `--sources`	❌

동사 및 명사

명령어 중 하위 명령이 없는 동작을 참조하는 경우에는 명사가 아니라 동사를 사용합니다. 예를 들면: `dotnet workload remove`, `dotnet workload removal`. 그리고 옵션에는 동사가 아니라 명사를 사용해야 합니다. 예를 들어: `--configuration`, `--configure`이 아닙니다.

옵션 `--verbosity`

`System.CommandLine` 애플리케이션은 일반적으로 콘솔로 `--verbosity` 전송되는 출력의 양을 지정하는 옵션을 제공합니다. 다음은 표준 5가지 설정입니다.

- `Q[uiet]`
- `M[inimal]`
- `N[ormal]`
- `D[etailed]`
- `Diag[nostic]`

이들은 표준 이름이지만, 기존 앱은 때때로 `Silent`를 `Quiet` 대신 사용하고, `Trace`, `Debug`, 또는 `Verbose`를 `Diagnostic` 대신 사용합니다.

각 앱은 각 수준에서 표시되는 항목을 결정하는 자체 조건을 정의합니다. 일반적으로 앱에는 다음 세 가지 수준만 필요합니다.

- 조용한
- 정상
- 진단

앱에 5개의 다른 수준이 필요하지 않은 경우 이 옵션은 여전히 동일한 5개의 설정을 정의해야 합니다. 이 경우 `Minimal` `Normal` 동일한 출력을 생성하며 `Detailed` `Diagnostic` 마찬가지로 동일합니다. 이렇게 하면 사용자가 익숙한 내용을 입력할 수 있으며 가장 적합한 항목이 사용됩니다.

콘솔에 `Quiet` 출력이 표시되지 않을 것으로 예상합니다. 그러나 앱이 대화형 모드를 제공하는 경우 앱은 다음 중 하나를 수행해야 합니다.

- `--interactive`이 지정되면 `--verbosity`이 `Quiet`인 경우에도 입력 프롬프트를 표시합니다.

- 함께 사용하는 `--verbosity Quiet` `--interactive` 것을 허용하지 않습니다.

그렇지 않으면 앱은 대기 중인 내용을 사용자에게 알리지 않고 입력을 기다립니다. 애플리케이션이 중지된 것으로 나타나고 사용자는 애플리케이션이 입력을 기다리고 있다는 사실을 전혀 알지 못합니다.

별칭을 정의하는 경우, `-v`를 `--verbosity`에 사용하고, 인수가 없는 `-v`를 `--verbosity Diagnostic`의 별칭으로 만드십시오. `-q`에 `--verbosity Quiet`를 사용합니다.

.NET CLI 및 POSIX 규칙

.NET CLI는 모든 POSIX 규칙을 일관되게 따르지 않습니다.

더블 대시

.NET CLI의 여러 명령에는 이중 대시 토큰의 특수한 구현이 있습니다. `dotnet run`, `dotnet watch`, `dotnet tool run`의 경우에는, `--` 뒤에 오는 토큰들이 명령에 의해 실행 중인 앱에 전달됩니다. 다음은 그 예입니다.

.NET CLI

```
dotnet run --project ./myapp.csproj -- --message "Hello world!"
                                       ^^
```

이 예제에서는 `--project` 옵션이 `dotnet run` 명령에 전달되고, `--message` 옵션과 그 인수가 *myapp*이 실행될 때 명령줄 옵션으로 전달됩니다.

`--`를 사용하여 `dotnet run` 실행하는 앱에 옵션을 전달할 때, 토큰이 항상 필요한 것은 아닙니다. 이중 대시 `dotnet run`가 없으면 인식되지 않는 모든 옵션은 자동으로 실행되는 앱에 전달됩니다. 이 옵션은 `dotnet run` 자체나 MSBuild에 적용되지 않는 것으로 간주됩니다. 따라서 인수 및 옵션을 인식하지 못하므로 다음 명령줄은 동일합니다 `dotnet run` .

.NET CLI

```
dotnet run -- quotes read --delay 0 --fg-color red
dotnet run quotes read --delay 0 --fg-color red
```

옵션-인수 구분 기호 생략

.NET CLI는 단일 문자 옵션 별칭을 지정할 때 구분 기호를 생략할 수 있는 POSIX 규칙을 지원하지 않습니다.

옵션 이름을 반복하지 않고 여러 인수

.NET CLI는 옵션 이름을 반복하지 않고 한 옵션에 대해 여러 인수를 허용하지 않습니다.

불리언 옵션

.NET CLI에서 일부 부울 옵션은 `false`을(를) 전달하는 경우와 `true`을(를) 전달하는 경우 동일하게 작동합니다. 이 동작은 옵션을 구현하는 .NET CLI 코드가 값을 무시하고 옵션의 존재 여부만 확인하는 경우 발생합니다. `--no-restore`은/는 `dotnet build` 명령에 대한 예입니다. `--no-restore false`를 전달하면 `--no-restore true` 또는 `--no-restore`를 지정한 경우와 동일하게 복원 작업이 건너뜁니다.

케밥 케이스

경우에 따라 .NET CLI에서는 명령어, 옵션 또는 인수 이름에 케밥 표기법을 사용하지 않습니다. 예를 들어, --additionalprobingpath 대신 `--additional-probing-path`로 이름이 지정된 .NET CLI 옵션이 있습니다.

참고하십시오

- 오픈 소스 CLI 디자인 지침
- GNU 표준

System.CommandLine 2.0.0-beta5+ 마이그레이션 가이드

2025. 08. 15.

> **ⓘ 중요**
>
> `System.CommandLine` 현재 미리 보기로 제공됩니다. 이 설명서는 버전 2.0 베타 5용입니다. 일부 정보는 릴리스되기 전에 실질적으로 수정될 수 있는 시험판 제품과 관련이 있습니다. Microsoft는 여기에 제공된 정보에 대해 어떠한 명시적이거나 묵시적인 보증도 하지 않습니다.

2.0.0-beta5 릴리스의 주요 초점은 API를 개선하고 안정적인 버전의 System.CommandLine릴리스를 위한 단계를 수행하는 것이었습니다. API는 간소화되었으며 프레임워크 디자인 지침과 더 일관되고 일관되었습니다. 이 문서에서는 2.0.0-beta5 및 2.0.0-beta7에서 수행된 주요 변경 내용과 그 뒤에 있는 추론에 대해 설명합니다.

이름 바꾸기

2.0.0-beta4에서는 모든 형식과 멤버가 명명 지침을 따르지 않았습니다. 일부는 부울 속성에 접두사를 사용하는 `Is` 것과 같이 명명 규칙과 일치하지 않았습니다. 2.0.0-beta5에서는 일부 형식과 멤버의 이름이 바뀌었습니다. 다음 표에는 이전 이름과 새 이름이 표시됩니다.

테이블 확장

이전 이름	새 이름
`System.CommandLine.Parsing.Parser`	CommandLineParser
`System.CommandLine.Parsing.OptionResult.IsImplicit`	Implicit
`System.CommandLine.Option.IsRequired`	Required
`System.CommandLine.Symbol.IsHidden`	Hidden
`System.CommandLine.Option.ArgumentHelpName`	HelpName
`System.CommandLine.Parsing.OptionResult.Token`	IdentifierToken
`System.CommandLine.Parsing.ParseResult.FindResultFor`	GetResult
`System.CommandLine.Parsing.SymbolResult.ErrorMessage`	AddError(String)†

†같은 기호에 대해 여러 오류를 보고 `ErrorMessage` 할 수 있도록 하기 위해 속성이 메서드로 변환되어 이름이 바뀌었습니다 `AddError`.

옵션 및 유효성 검사기의 변경 가능한 컬렉션

버전 2.0.0-beta4에는 인수, 옵션, 하위 명령, 유효성 검사기 및 완성과 같은 컬렉션에 항목을 추가하는 데 사용된 다양한 `Add` 메서드가 있었습니다. 이러한 컬렉션 중 일부는 속성을 통해 읽기 전용 컬렉션으로 노출되었습니다. 따라서 해당 컬렉션에서 항목을 제거하는 것은 불가능했습니다.

2.0.0-beta5에서는 API가 메서드 및 읽기 전용 컬렉션 대신 `Add` 변경 가능한 컬렉션을 노출하도록 변경되었습니다. 이렇게 하면 항목을 추가하거나 열거할 뿐만 아니라 제거할 수도 있습니다. 다음 표에는 이전 메서드 및 새 속성 이름이 표시됩니다.

테이블 확장

이전 메서드 이름	새 속성
`Command.AddArgument`	`Command.Arguments.Add`
`Command.AddOption`	`Command.Options.Add`
`Command.AddCommand`	`Command.Subcommands.Add`
`Command.AddValidator`	`Command.Validators.Add`
`Option.AddValidator`	`Option.Validators.Add`
`Argument.AddValidator`	`Argument.Validators.Add`
`Command.AddCompletions`	`Command.CompletionSources.Add`
`Option.AddCompletions`	`Option.CompletionSources.Add`
`Argument.AddCompletions`	`Argument.CompletionSources.Add`
`Command.AddAlias`	`Command.Aliases.Add`
`Option.AddAlias`	`Option.Aliases.Add`

`RemoveAlias` 메서드와 `HasAlias` 메서드도 제거되었습니다. 이는 `Aliases` 속성이 이제 변경 가능한 컬렉션이 되었기 때문입니다. 이 메서드를 `Remove` 사용하여 컬렉션에서 별칭을 제거할 수 있습니다. 이 메서드를 `Contains` 사용하여 별칭이 있는지 확인합니다.

이름 및 별칭

2.0.0-beta5 이전에는 기호의 이름과 별칭 간에 명확한 구분이 없었습니다. `name`가 `Option<T>` 생성자에 제공되지 않은 경우, 기호는 `--`, `-`, `/`과 같은 접두사가 제거된 가장 긴 별칭으로 이름을 보고했습니다. 혼란스러웠습니다.

게다가, 구문 분석된 값을 얻으려면 옵션 또는 인수에 대한 참조를 저장해야 했으며, 그런 다음 이를 사용하여 `ParseResult`에서 값을 가져와야 했습니다.

단순성과 명시성을 높이기 위해 기호의 이름은 이제 모든 기호 생성자(포함 `Argument<T>`)에 대한 필수 매개 변수입니다. 이름 및 별칭의 개념은 이제 구분됩니다. 별칭은 별칭일 뿐이며 기호의 이름은 포함하지 않습니다. 물론 선택 사항입니다. 그 결과 다음과 같은 변경이 수행되었습니다.

- `name`는 이제 모든 공용 생성자에 Argument<T>Option<T>대한 필수 인수입니다 Command. 이 `Argument<T>`경우 구문 분석이 아니라 도움말을 생성하는 데 사용됩니다. `Option<T>` `Command`구문 분석 중에 기호를 식별하고 도움말 및 완성을 위해 사용됩니다.
- 속성은 Symbol.Name 더 이상 `virtual`없습니다. 이제 읽기 전용이며 기호를 만들 때 제공된 대로 이름을 반환합니다. 따라서 `Symbol.DefaultName` 제거되었으며 Option.Name 더 이상 가장 긴 별칭에서 또는 다른 접두사를 제거 `--` `-` `/`하지 않습니다.
- `Aliases` 속성은 Option 및 Command에 의해 노출되며, 이제 변경 가능한 컬렉션입니다. 이 컬렉션은 더 이상 기호의 이름을 포함하지 않습니다.
- `System.CommandLine.Parsing.IdentifierSymbol` 가 제거되었습니다(둘 다 `Command` 에 `Option`대한 기본 형식이었습니다).

이름이 항상 있으면 이름으로 구문 분석된 값을 가져올 수 있습니다.

C#

```
RootCommand command = new("The description.")
{
    new Option<int>("--number")
};

ParseResult parseResult = command.Parse(args);
int number = parseResult.GetValue<int>("--number");
```

별칭을 사용하여 옵션 만들기

과거에 Option<T> 는 많은 생성자를 노출했으며 그 중 일부는 이름을 수락했습니다. 이제 이름이 필수이며 별칭이 자주 제공 `Option<T>`되므로 하나의 생성자만 있습니다. 이름 및 별칭 배열을 `params` 허용합니다.

2.0.0-beta5 `Option<T>` 이전에는 이름과 설명을 사용하는 생성자가 있었습니다. 따라서 두 번째 인수는 이제 설명이 아닌 별칭으로 처리될 수 있습니다. 컴파일러 오류를 일으키지 않는 API의 유일한 알려진 호환성이 손상되는 변경입니다.

이름과 별칭을 사용하는 새 생성자를 사용하도록 생성자에 설명을 전달한 코드를 업데이트한 다음 속성을 별도로 설정합니다 `Description` . 다음은 그 예입니다.

C#

```
Option<bool> beta4 = new("--help", "An option with aliases.");
beta4b.Aliases.Add("-h");
beta4b.Aliases.Add("/h");

Option<bool> beta5 = new("--help", "-h", "/h")
{
    Description = "An option with aliases."
};
```

기본값 및 사용자 지정 구문 분석

2.0.0-beta4에서는 메서드를 사용하여 옵션 및 인수에 대한 기본값을 `SetDefaultValue` 설정할 수 있습니다. 이러한 메서드는 형식이 `object` 안전하지 않은 값을 허용했으며 값이 옵션 또는 인수 형식과 호환되지 않는 경우 런타임 오류가 발생할 수 있습니다.

C#

```
Option<int> option = new("--number");
// This is not type safe, as the value is a string, not an int:
option.SetDefaultValue("text");
```

또한 일부 `Option` 및 `Argument` 생성자는 구문 분석 대리자와 그 대리자가 사용자 지정 파서인지 기본값 제공자인지를 나타내는 부울 값을 수락했습니다. 이를 통해 혼란을 야기했습니다.

`Option<T>` 및 `Argument<T>` 클래스에는 옵션 또는 인수의 기본값을 가져오기 위해 호출할 수 있는 대리자를 설정하는 데 사용할 수 있는 DefaultValueFactory 속성이 있습니다. 구문 분석된 명령줄 입력에서 옵션 또는 인수를 찾을 수 없는 경우 이 대리자가 호출됩니다.

C#

```
Option<int> number = new("--number")
{
    DefaultValueFactory = _ => 42
};
```

`Argument<T>` 및 `Option<T>` 는 기호에 대한 사용자 지정 파서를 설정할 수 있는 CustomParser 속성도 함께 제공합니다.

C#

```
Argument<Uri> uri = new("arg")
{
    CustomParser = result =>
    {
        if (!Uri.TryCreate(result.Tokens.Single().Value,
UriKind.RelativeOrAbsolute, out var uriValue))
        {
            result.AddError("Invalid URI format.");
            return null;
        }

        return uriValue;
    }
};
```

또한 `CustomParser` 이전 `Func<ParseResult,T>` 대리자가 아닌 형식 `ParseArgument`의 대리자를 허용합니다. 이 대리자와 몇 가지 다른 사용자 지정 대리자는 API를 단순화하고 API에 의해 노출되는 형식 수를 줄이기 위해 제거되어 JIT 컴파일 중에 소요되는 시작 시간을 줄입니다.

사용 `DefaultValueFactory` 및 `CustomParser` 방법에 대한 추가 예제는 System.CommandLine에서 구문 분석 및 유효성 검사 사용자 지정 방법을 참조하세요.

구문 분석 및 호출 분리

2.0.0-beta4에서는 명령의 구문 분석 및 호출을 분리할 수 있었지만 어떻게 해야 할지 명확하지 않았습니다. `Command`는 `Parse` 메서드를 노출하지 않았지만, `CommandExtensions`는 `Command`를 위한 `Parse`, `Invoke`, 및 `InvokeAsync` 확장 메서드를 제공했습니다. 어떤 방법을 언제 사용할지 명확하지 않았기 때문에 혼란스러웠습니다. API를 간소화하기 위해 다음과 같이 변경되었습니다.

- Command 이제 `Parse` 객체를 반환하는 `ParseResult` 메서드를 노출합니다. 이 메서드는 명령줄 입력을 구문 분석하고 구문 분석 작업의 결과를 반환하는 데 사용됩니다. 또한 명령이 호출되지 않고 구문 분석되며 동기식으로만 수행됨을 분명히 합니다.
- `ParseResult`은 이제 명령을 호출하는 데 사용할 수 있는 `Invoke` 및 `InvokeAsync` 메서드를 모두 노출합니다. 이 패턴은 구문 분석 후 명령이 호출되고 동기 및 비동기 호출을 모두 허용한다는 것을 분명히 합니다.
- `CommandExtensions` 더 이상 필요하지 않으므로 클래스가 제거되었습니다.

구성 / 설정

2.0.0-beta5 이전에는 일부 공용 `Parse` 메서드만 사용하여 구문 분석을 사용자 지정할 수 있었습니다. 두 개의 공용 생성자를 제공하는 `Parser` 클래스가 있었습니다. 하나는 `Command`을 수락하고, 다른 하나는 `CommandLineConfiguration`을 수락합니다. `CommandLineConfiguration` 변경할 수

없는 경우 이를 만들려면 클래스에서 노출하는 `CommandLineBuilder` 작성기 패턴을 사용해야 했습니다. API를 간소화하기 위해 다음과 같이 변경되었습니다.

- `CommandLineConfiguration` 는 두 개의 *변경 가능한* 클래스(2.0.0-beta7) ParserConfiguration 로 분할되었습니다 InvocationConfiguration. 호출 구성을 만드는 것은 이제 인스턴스 `InvocationConfiguration` 를 만들고 사용자 지정하려는 속성을 설정하는 것만큼 간단합니다.
- 이제 모든 `Parse` 메서드는 구문 분석을 사용자 지정하는 데 사용할 수 있는 선택적 ParserConfiguration 매개 변수를 허용합니다. 제공되지 않으면 기본 구성이 사용됩니다.
- 이름 충돌을 피하기 위해 `Parser`가 다른 파서 형식과 혼동되지 않도록 CommandLineParser로 이름이 변경되었습니다. 상태 비정상이므로 이제 정적 메서드만 있는 정적 클래스입니다. 두 가지 `Parse` 구문 분석 메서드를 제공합니다. 하나는 `IReadOnlyList<string> args`를 수락하고, 다른 하나는 `string args`를 수락합니다. 후자는 명령줄 입력을 CommandLineParser.SplitCommandLine(String)으로 분할하여 구문 분석하기 전에 (공용)을 사용합니다.

`CommandLineBuilderExtensions` 도 제거되었습니다. 메서드를 새 API에 매핑하는 방법은 다음과 같습니다.

- `CancelOnProcessTermination` 는 이제 InvocationConfiguration의 속성으로, ProcessTerminationTimeout로 불립니다. 기본적으로 2초 시간 제한으로 사용하도록 설정됩니다. 사용하지 않도록 설정하려면 으로 `null`설정합니다. 자세한 내용은 프로세스 종료 시간 제한을 참조하세요.

- `EnableDirectives`, `UseEnvironmentVariableDirective`, `UseParseDirective`및 `UseSuggestDirective` 제거되었습니다. 새 지시문 형식이 도입되었고 RootCommand 가 Directives 이제 속성을 노출합니다. 이 컬렉션을 사용하여 지시문을 추가, 제거 및 반복할 수 있습니다. Suggest 지시문은 기본적으로 포함되며, DiagramDirective나 EnvironmentVariablesDirective와 같은 다른 지시문도 사용할 수 있습니다.

- `EnableLegacyDoubleDashBehavior` 가 제거되었습니다. 이제 일치하지 않는 모든 토큰이 속성에 ParseResult.UnmatchedTokens 의해 노출됩니다. 자세한 내용은 일치하지 않는 토큰을 참조하세요.

- `EnablePosixBundling` 가 제거되었습니다. 묶음이 이제 기본적으로 사용 설정되어 있으며, 비활성화하려면 ParserConfiguration.EnablePosixBundling 속성을 `false`로 설정하십시오. 자세한 내용은 EnablePosixBundling을 참조하세요.

- `RegisterWithDotnetSuggest` 는 일반적으로 애플리케이션을 시작하는 동안 비용이 많이 드는 작업을 수행할 때 제거되었습니다. 이제 `dotnet suggest` 명령을 등록해야 합니다.

- `UseExceptionHandler` 가 제거되었습니다. 이제 기본 예외 처리기가 기본적으로 사용하도록 설정됩니다. 속성을 InvocationConfiguration.EnableDefaultExceptionHandler.로 설정 `false` 하여 사용하지 않도록 설정할 수 있습니다. 이는 `Invoke` 메서드 또는 `InvokeAsync` 메서드를 try-catch 블록으로 래핑하여 사용자 지정 방식으로 예외를 처리하려는 경우에 유용합니다. 자세한 내용은 EnableDefaultExceptionHandler를 참조하세요.

- `UseHelp`와 `UseVersion`가 제거되었습니다. 이제 HelpOption 및 VersionOption 공용 형식에 도움말 및 버전이 노출되었습니다. 둘 다 RootCommand에서 정의한 옵션에 기본적으로 포함됩니다. 자세한 내용은 도움말 출력 및 버전 사용자 지정 옵션을 참조하세요.

- `UseHelpBuilder` 가 제거되었습니다. 도움말 출력을 사용자 지정하는 방법에 대한 자세한 내용은 도움말System.CommandLine을 사용자 지정하는 방법을 참조하세요.

- `AddMiddleware` 가 제거되었습니다. 애플리케이션 시작 속도가 느려졌으며, 기능 없이도 기능을 표현할 수 있습니다.

- `UseParseErrorReporting`와 `UseTypoCorrections`가 제거되었습니다. 이제 `ParseResult`를 호출할 때 구문 오류가 기본적으로 보고됩니다. ParseResult.Action 속성에 의해 노출된 ParseErrorAction 작업을 사용하여 구성할 수 있습니다.

  ```
  C#

  ParseResult result = rootCommand.Parse("myArgs", config);
  if (result.Action is ParseErrorAction parseError)
  {
      parseError.ShowTypoCorrections = true;
      parseError.ShowHelp = false;
  }
  ```

- `UseLocalizationResources`와 `LocalizationResources`가 제거되었습니다. 이 기능은 주로 `dotnet`에 누락된 번역을 추가하기 위해 `System.CommandLine` CLI에서 사용되었습니다. 이러한 모든 번역은 자체로 System.CommandLine 이동되었으므로 이 기능은 더 이상 필요하지 않습니다. 언어에 대한 지원이 누락된 경우 문제를 보고하세요.

- `UseTokenReplacer` 가 제거되었습니다. 응답 파일은 기본적으로 사용하도록 설정되어 있지만 속성을 ResponseFileTokenReplacer.로 설정 `null` 하여 사용하지 않도록 설정할 수 있습니다. 응답 파일 처리 방법을 사용자 지정하는 사용자 지정 구현을 제공할 수도 있습니다.

마지막으로 모든 `IConsole` 관련 인터페이스(`IStandardOut`, `IStandardError`, `IStandardIn`)가 제거되었습니다. InvocationConfiguration는 두 가지 `TextWriter` 속성을 노출합니다. OutputError 이러한 속성은 테스트용 출력을 캡처하는 데 사용할 수 있는 TextWriter 인스턴스, 예를 들어 `StringWriter` 인스턴스에 설정할 수 있습니다. 이 변경의 동기는 더 적은 형식을 노출하고 기존 추상화 다시 사용하는 것이었습니다.

호출

2.0.0-beta4 버전에서는 `ICommandHandler` 인터페이스가 노출되었고, 이 인터페이스에서 `Invoke` 및 `InvokeAsync` 메서드를 사용하여 구문 분석된 명령을 호출했습니다. 따라서 명령에 대한 동기 처리기를 정의한 다음 비동기적으로 호출하여(교착 상태가 발생할 수 있음) 동기 코드와 비동기 코드를 쉽게 혼합할 수 있습니다. 또한 명령에 대해서만 처리기를 정의할 수 있었지만 옵션(예: 도움말을 표시하는 도움말) 또는 지시문에는 정의할 수 없었습니다.

새 추상 기본 클래스 CommandLineAction 와 두 개의 파생 클래스 및 `SynchronousCommandLineAction` `AsynchronousCommandLineAction`이 클래스가 도입되었습니다. 전자는 종료 코드를 반환 `int` 하는 동기 작업에 사용되고, 후자는 종료 코드를 반환 `Task<int>` 하는 비동기 작업에 사용됩니다.

동작을 정의하기 위해 파생 형식을 만들 필요가 없습니다. 이 메서드를 Command.SetAction 사용하여 명령에 대한 작업을 설정할 수 있습니다. 동기 작업은 `System.CommandLine.ParseResult` 매개 변수를 사용하고 `int` 종료 코드를 반환하는 대리자일 수 있습니다. 또는 아무것도 반환하지 않으면 기본 `0` 종료 코드가 반환됩니다. 비동기 작업은 `System.CommandLine.ParseResult` 및 CancellationToken 매개 변수를 받고 `Task<int>`를 반환하는 대리자가 될 수 있습니다 (또는 기본 종료 코드 반환을 위해 `Task`를 사용할 수 있습니다).

C#

```
rootCommand.SetAction(ParseResult parseResult =>
{
    FileInfo parsedFile = parseResult.GetValue(fileOption);
    ReadFile(parsedFile);
});
```

과거에는 `CancellationToken`이 `InvokeAsync`의 메서드를 통해 `InvocationContext`에게 전달되어 처리기에 노출되었습니다.

C#

```
rootCommand.SetHandler(async (InvocationContext context) =>
{
    string? urlOptionValue = context.ParseResult.GetValueForOption(urlOption);
    var token = context.GetCancellationToken();
    returnCode = await DoRootCommand(urlOptionValue, token);
});
```

대부분의 사용자는 이 토큰을 가져오고 더 이상 전달하지 않았습니다. `CancellationToken` 는 이제 비동기 작업에 대한 필수 인수로, 컴파일러가 더 이상 전달되지 않을 때 경고를 생성합니다(CA2016 참조).

C#

```
rootCommand.SetAction((ParseResult parseResult, CancellationToken token) =>
{
    string? urlOptionValue = parseResult.GetValue(urlOption);
    return DoRootCommandAsync(urlOptionValue, token);
});
```

이러한 변경 내용과 앞서 언급한 기타 변경 내용 `InvocationContext` 으로 인해 클래스도 제거되었습니다. `ParseResult` 이제 작업에 직접 전달되므로 구문 분석된 값과 옵션에 직접 액세스할 수 있습니다.

이러한 변경 내용을 요약하려면 다음을 수행합니다.

- 인터페이스 `ICommandHandler`가 제거되었습니다. `SynchronousCommandLineAction` 및 `AsynchronousCommandLineAction`이 소개되었습니다.
- 메서드의 `Command.SetHandler` 이름이 .로 바뀌었습니다 SetAction.
- 속성 이름이 `Command.Handler` .로 바뀌었습니다 Command.Action. `Option` 을 사용하여 Option.Action확장되었습니다.
- `InvocationContext` 가 제거되었습니다. `ParseResult`가 이제 작업에 직접 전달됩니다.

작업을 사용하는 방법에 대한 자세한 내용은 다음에서 System.CommandLine명령을 구문 분석하고 호출하는 방법을 참조하세요.

간소화된 API의 이점

2.0.0-beta5에서 변경된 사항은 API를 보다 일관되고, 미래 지향적이고, 기존 및 신규 사용자에게 더 쉽게 사용할 수 있도록 합니다.

공용 인터페이스 수가 11개에서 0개로 감소하고 공용 클래스(및 구조체)가 56개에서 38개로 감소하므로 새 사용자는 더 적은 개념과 형식을 학습해야 합니다. 공용 메서드 수는 378개에서 235개, 공용 속성은 118개에서 99개로 감소했습니다.

참조되는 System.CommandLine 어셈블리 수는 11개에서 6개로 줄어듭니다.

diff

```
System.Collections
- System.Collections.Concurrent
- System.ComponentModel
System.Console
- System.Diagnostics.Process
System.Linq
System.Memory
- System.Net.Primitives
```

```
System.Runtime
- System.Runtime.Serialization.Formatters
+ System.Runtime.InteropServices
- System.Threading
```

라이브러리 크기가 줄어들고(%32개씩) 라이브러리를 사용하는 NativeAOT 앱의 크기도 줄어듭니다.

단순성도 라이브러리의 성능을 향상시켰습니다(주요 목표가 아니라 작업의 부작용). 벤치마크는 명령의 구문 분석 및 호출이 2.0.0-beta4보다 빠르며, 특히 많은 옵션과 인수가 있는 대형 명령의 경우 더 빠릅니다. 성능 향상은 동기 및 비동기 시나리오 모두에서 표시됩니다.

앞에서 제시한 가장 간단한 앱은 다음과 같은 결과를 생성합니다.

ini

```
BenchmarkDotNet v0.15.0, Windows 11 (10.0.26100.4061/24H2/2024Update/HudsonValley)
AMD Ryzen Threadripper PRO 3945WX 12-Cores 3.99GHz, 1 CPU, 24 logical and 12
physical cores
.NET SDK 9.0.300
  [Host]     : .NET 9.0.5 (9.0.525.21509), X64 RyuJIT AVX2
  Job-JJVAFK : .NET 9.0.5 (9.0.525.21509), X64 RyuJIT AVX2

EvaluateOverhead=False  OutlierMode=DontRemove  InvocationCount=1
IterationCount=100  UnrollFactor=1  WarmupCount=3

| Method                  | Args           | Mean      | StdDev    | Ratio |
|------------------------ |--------------- |----------:|----------:|------:|
| Empty                   | --bool -s test |  63.58 ms | 0.825 ms |  0.83 |
| EmptyAOT                | --bool -s test |  14.39 ms | 0.507 ms |  0.19 |
| SystemCommandLineBeta4  | --bool -s test |  85.80 ms | 1.007 ms |  1.12 |
| SystemCommandLineNow    | --bool -s test |  76.74 ms | 1.099 ms |  1.00 |
| SystemCommandLineNowR2R | --bool -s test |  69.35 ms | 1.127 ms |  0.90 |
| SystemCommandLineNowAOT | --bool -s test |  17.35 ms | 0.487 ms |  0.23 |
```

볼 수 있듯이 시작 시간(벤치마크는 지정된 실행 파일을 실행하는 데 필요한 시간을 보고함)이 2.0.0-beta4에 비해 12% 향상되었습니다. NativeAOT를 사용하여 앱을 컴파일하는 경우 인수를 전혀 구문 분석하지 않는 NativeAOT 앱보다 3ms 느립니다(위 표의 EmptyAOT). 또한 빈 앱(63.58ms)의 오버헤드를 제외하면 구문 분석이 2.0.0-beta4(22.22ms 대 13.66ms)보다 40% 빠릅니다.

참고하십시오

- System.CommandLine 개요

파일 및 스트림 입출력

2025. 06. 17.

파일 및 스트림 I/O(입력/출력)는 스토리지 매체 간에 데이터를 전송하는 것을 의미합니다. .NET `System.IO` 에서 네임스페이스에는 데이터 스트림 및 파일에 대해 동기적으로나 비동기적으로 읽기 및 쓰기를 가능하게 하는 형식이 포함되어 있습니다. 이러한 네임스페이스에는 파일에서 압축 및 압축 해제를 수행하는 형식과 파이프 및 직렬 포트를 통한 통신을 가능하게 하는 형식도 포함됩니다.

파일은 영구 스토리지가 있는 바이트의 순서가 지정되고 명명된 컬렉션입니다. 파일을 사용하는 경우 디렉터리 경로, 디스크 스토리지, 파일 및 디렉터리 이름을 사용합니다. 반면 스트림은 백업 저장소에서 읽고 쓰는 데 사용할 수 있는 바이트 시퀀스로, 여러 스토리지 매체(예: 디스크 또는 메모리) 중 하나일 수 있습니다. 디스크 이외의 여러 백업 저장소가 있는 것처럼 네트워크, 메모리 및 파이프 스트림과 같은 파일 스트림 이외의 여러 종류의 스트림이 있습니다.

파일 및 디렉터리

네임스페이스의 형식을 System.IO 사용하여 파일 및 디렉터리와 상호 작용할 수 있습니다. 예를 들어 파일 및 디렉터리에 대한 속성을 가져와서 설정하고 검색 조건에 따라 파일 및 디렉터리의 컬렉션을 검색할 수 있습니다.

경로 명명 규칙 및 .NET Core 1.1 이상 및 .NET Framework 4.6.2 이상에서 지원되는 DOS 디바이스 구문을 포함하여 Windows 시스템의 파일 경로를 표현하는 방법은 Windows 시스템의 파일 경로 형식을 참조하세요.

일반적으로 사용되는 몇 가지 파일 및 디렉터리 클래스는 다음과 같습니다.

- File - 파일을 만들고, 복사하고, 삭제하고, 이동하고, 여는 정적 메서드를 제공하며 개체를 FileStream 만드는 데 도움이 됩니다.
- FileInfo - 파일 만들기, 복사, 삭제, 이동 및 열기를 위한 인스턴스 메서드를 제공하고 개체를 FileStream 만드는 데 도움이 됩니다.
- Directory - 디렉터리 및 하위 디렉터리를 통해 생성, 이동 및 열거를 위한 정적 메서드를 제공합니다.
- DirectoryInfo - 디렉터리 및 하위 디렉터리를 통해 만들고, 이동하고, 열거하는 인스턴스 메서드를 제공합니다.
- Path - 플랫폼 간 방식으로 디렉터리 문자열을 처리하기 위한 메서드 및 속성을 제공합니다.

파일 시스템 메서드를 호출할 때 항상 강력한 예외 처리를 제공해야 합니다. 자세한 내용은 I/O 오류 처리를 참조하세요.

Visual Basic 사용자는 이러한 클래스를 사용하는 것 외에도 파일 I/O에 대해 클래스에서 Microsoft.VisualBasic.FileIO.FileSystem 제공하는 메서드와 속성을 사용할 수 있습니다.

방법 : 디렉터리 복사, 방법: 디렉터리 목록 만들기 및 방법: 디렉터리 및 파일 열거를 참조하세요.

스트림

추상 기본 클래스 Stream 는 바이트 읽기 및 쓰기를 지원합니다. 스트림을 나타내는 모든 클래스는 Stream 클래스로부터 상속받습니다. 클래스와 파생 클래스는 Stream 데이터 원본 및 리포지토리에 대한 공통 보기를 제공하고 프로그래머를 운영 체제 및 기본 디바이스의 특정 세부 정보로부터 격리합니다.

스트림에는 다음 세 가지 기본 작업이 포함됩니다.

- 읽기 - 스트림에서 바이트 배열과 같은 데이터 구조로 데이터 전송
- 쓰기 - 데이터 원본에서 스트림으로 데이터 전송
- 검색 - 스트림 내에서 현재 위치 쿼리 및 수정

기본 데이터 원본 또는 리포지토리에 따라 스트림은 이러한 기능 중 일부만 지원할 수 있습니다. 예를 들어 클래스는 PipeStream 검색을 지원하지 않습니다. 스트림의 , CanRead및 CanWrite 속성은 CanSeek스트림이 지원하는 작업을 지정합니다.

다음은 일반적으로 사용되는 몇 가지 스트림 클래스입니다.

- FileStream – 파일 읽기 및 쓰기용입니다.
- IsolatedStorageFileStream – 격리된 스토리지의 파일을 읽고 쓰기 위한 것입니다.
- MemoryStream – 메모리를 백업 저장소로 읽고 쓰기 위한 것입니다.
- BufferedStream – 읽기 및 쓰기 작업의 성능을 향상하기 위한 것입니다.
- NetworkStream – 네트워크 소켓을 통해 읽고 쓰는 데 사용됩니다.
- PipeStream – 익명 및 명명된 파이프를 통해 읽고 쓰는 데 사용합니다.
- CryptoStream – 암호화 변환에 데이터 스트림을 연결하는 데 사용합니다.

스트림을 비동기적으로 사용하는 예제는 비동기 파일 I/O를 참조하세요.

독자 및 작성자

또한 네임스페이 System.IO 스는 스트림에서 인코딩된 문자를 읽고 스트림에 쓰기 위한 형식을 제공합니다. 일반적으로 스트림은 바이트 입력 및 출력을 위해 설계되었습니다. 판독기 및 기록기 형식은 스트림이 작업을 완료할 수 있도록 인코딩된 문자의 바이트 변환을 처리합니다. 각 판독기 및 기록기 클래스는 클래스의 `BaseStream` 속성을 통해 검색할 수 있는 스트림과 연결됩니다.

다음은 일반적으로 사용되는 몇 가지 판독기 및 작성기 클래스입니다.

- BinaryReader 및 BinaryWriter – 기본 데이터 형식을 이진 값으로 읽고 쓰는 데 사용됩니다.
- StreamReader 및 StreamWriter – 인코딩 값을 사용하여 문자를 바이트 단위로 변환하여 문자를 읽고 쓰는 경우
- StringReader 및 StringWriter – 문자열에서 문자를 읽고 쓰는 데 사용됩니다.
- TextReader 및 TextWriter – 문자와 문자열을 읽고 쓰는 다른 판독기 및 작성기에 대한 추상 기본 클래스로 사용되지만 이진 데이터는 사용하지 않습니다.

방법: 파일에서 텍스트 읽기, 방법: 파일에 텍스트 쓰기, 방법: 문자열에서 문자 읽기 및 방법: 문자열에 문자 쓰기

비동기 I/O 작업

많은 양의 데이터를 읽거나 쓰는 것은 리소스를 많이 사용할 수 있습니다. 애플리케이션이 사용자에게 계속 응답해야 하는 경우 이러한 작업을 비동기적으로 수행해야 합니다. 동기 I/O 작업을 사용하면 리소스 집약적 작업이 완료될 때까지 UI 스레드가 차단됩니다. Windows 8.x 스토어 앱을 개발할 때 비동기 I/O 작업을 사용하여 앱 작동이 중지되었다는 인상을 표시하지 않도록 합니다.

비동기 멤버는 이름에 `Async`이 포함되어 있으며, CopyToAsync, FlushAsync, ReadAsync, 및 WriteAsync 메서드가 그 예입니다. `async` 및 `await` 키워드와 함께 이러한 메서드를 사용합니다.

자세한 내용은 비동기 파일 I/O를 참조하세요.

압축

압축은 스토리지에 대한 파일의 크기를 줄이는 프로세스를 나타냅니다. 압축 해제는 압축된 파일의 콘텐츠를 추출하여 사용 가능한 형식으로 만드는 프로세스입니다. 네임스페이스에는 System.IO.Compression 파일 및 스트림을 압축 및 압축 해제하기 위한 형식이 포함되어 있습니다.

다음 클래스는 파일 및 스트림을 압축 및 압축 해제할 때 자주 사용됩니다.

- ZipArchive – zip 보관 파일에서 항목을 만들고 검색하는 데 사용할 수 있습니다.
- ZipArchiveEntry – 압축된 파일을 나타내는 데 사용할 수 있습니다.
- ZipFile – 압축된 패키지를 만들고, 추출하고, 여는 데 사용할 수 있습니다.
- ZipFileExtensions – 압축된 패키지에서 항목을 만들고 추출하는 데 사용할 수 있습니다.
- DeflateStream – Deflate 알고리즘을 사용하여 스트림을 압축 및 압축 해제합니다.
- GZipStream – gzip 데이터 형식으로 스트림 압축 및 압축을 해제하는 경우

방법: 파일 압축 및 추출을 참조하세요.

격리된 스토리지

격리된 스토리지는 코드를 저장된 데이터와 연결하는 표준화된 방법을 정의하여 격리 및 안전을 제공하는 데이터 스토리지 메커니즘입니다. 스토리지는 사용자, 어셈블리 및(선택 사항) 도메인에 의해 격리된 가상 파일 시스템을 제공합니다. 격리된 스토리지는 애플리케이션에 사용자 파일에 액세스할 수 있는 권한이 없는 경우에 특히 유용합니다. 컴퓨터의 보안 정책에 의해 제어되는 방식으로 애플리케이션에 대한 설정 또는 파일을 저장할 수 있습니다.

Windows 8.x 스토어 앱에는 격리된 스토리지를 사용할 수 없습니다. 대신 네임스페이스에서 애플리케이션 데이터 클래스를 Windows.Storage 사용합니다. 자세한 내용은 애플리케이션 데이터를 참조하세요.

격리된 스토리지를 구현할 때 자주 사용되는 클래스는 다음과 같습니다.

- IsolatedStorage – 격리된 스토리지 구현에 대한 기본 클래스를 제공합니다.
- IsolatedStorageFile – 파일 및 디렉터리를 포함하는 격리된 스토리지 영역을 제공합니다.
- IsolatedStorageFileStream - 격리된 스토리지 내의 파일을 노출합니다.

격리된 스토리지를 참조하세요.

Windows 스토어 앱의 I/O 작업

Windows 8.x 스토어 앱용 .NET에는 스트림에서 읽고 쓰기 위한 다양한 형식이 포함되어 있습니다. 그러나 이 집합에는 모든 .NET I/O 형식이 포함되지 않습니다.

Windows 8.x 스토어 앱에서 I/O 작업을 사용할 때 유의해야 할 몇 가지 중요한 차이점:

- 파일 작업과 관련된 형식(예: File, FileInfoDirectory 및DirectoryInfo)은 Windows 8.x 스토어 앱용 .NET에 포함되지 않습니다. 대신 Windows 런타임의 네임스페이 Windows.Storage 스에 있는 형식(예: StorageFile 및 StorageFolder.)을 사용합니다.
- 격리된 스토리지를 사용할 수 없습니다. 대신 애플리케이션 데이터를 사용합니다.
- 와 같은 ReadAsyncWriteAsync비동기 메서드를 사용하여 UI 스레드 차단을 방지합니다.
- 경로 기반 압축 형식 ZipFile 이며 ZipFileExtensions 사용할 수 없습니다. 대신 네임스페이스의 형식을 Windows.Storage.Compression 사용합니다.

필요한 경우 .NET Framework 스트림과 Windows 런타임 스트림 간에 변환할 수 있습니다. 자세한 내용은 방법: .NET Framework 스트림과 Windows 런타임 스트림 간 변환 또는 WindowsRuntimeStreamExtensions.

Windows 8.x 스토어 앱의 I/O 작업에 대한 자세한 내용은 빠른 시작: 파일 읽기 및 쓰기를 참조하세요.

I/O 및 보안

네임스페이스에서 클래스를 System.IO 사용하는 경우 ACL(액세스 제어 목록)과 같은 운영 체제 보안 요구 사항을 따라 파일 및 디렉터리에 대한 액세스를 제어해야 합니다. 이 요구 사항은 모든 FileIOPermission 요구 사항에 추가됩니다. 프로그래밍 방식으로 ACL을 관리할 수 있습니다. 자세한 내용은 방법: 액세스 제어 목록 항목 추가 또는 제거를 참조하세요.

기본 보안 정책은 인터넷 또는 인트라넷 애플리케이션이 사용자 컴퓨터의 파일에 액세스하는 것을 방지합니다. 따라서 인터넷 또는 인트라넷을 통해 다운로드할 코드를 작성할 때 실제 파일에 대한 경로가 필요한 I/O 클래스를 사용하지 마세요. 대신 .NET 애플리케이션에 격리된 스토리지 를 사용합니다.

보안 검사는 스트림이 생성될 때만 수행됩니다. 따라서 스트림을 연 다음 신뢰도가 낮은 코드 또는 애플리케이션 도메인에 전달하지 마세요.

관련 항목

- 일반적인 I/O 작업
  파일, 디렉터리 및 스트림과 연결된 I/O 작업 목록과 각 작업에 대한 관련 콘텐츠 및 예제에 대한 링크를 제공합니다.
- 비동기 파일 I/O
  비동기 I/O의 성능 이점 및 기본 작업에 대해 설명합니다.

- 격리된 스토리지
  코드를 저장된 데이터와 연결하는 표준화된 방법을 정의하여 격리 및 안전을 제공하는 데이터 스토리지 메커니즘에 대해 설명합니다.

- 파이프
  .NET의 익명 및 명명된 파이프 작업에 대해 설명합니다.

- Memory-Mapped 파일
  가상 메모리의 디스크에 있는 파일의 내용을 포함하는 메모리 매핑된 파일에 대해 설명합니다. 메모리 매핑된 파일을 사용하여 매우 큰 파일을 편집하고 프로세스 간 통신을 위한 공유 메모리를 만들 수 있습니다.

Windows 시스템의 파일 경로 형식

네임스페이스에 있는 System.IO 많은 형식의 멤버에는 파일 시스템 리소스에 `path` 대한 절대 또는 상대 경로를 지정할 수 있는 매개 변수가 포함됩니다. 그런 다음 이 경로가 Windows 파일 시스템 API에 전달됩니다. 이 항목에서는 Windows 시스템에서 사용할 수 있는 파일 경로의 형식에 대해 설명합니다.

기존 DOS 경로

표준 DOS 경로는 다음 세 가지 구성 요소로 구성됩니다.

- 볼륨 또는 드라이브 문자 뒤에 볼륨 구분 기호(`:`)가 잇습니다.
- 디렉터리 이름입니다. 디렉터리 구분 기호 문자는 중첩된 디렉터리 계층 내의 하위 디렉터리를 구분합니다.
- 선택적 파일 이름입니다. 디렉터리 구분 기호 문자는 파일 경로와 파일 이름을 구분합니다.

세 구성 요소가 모두 있는 경우 경로는 절대 경로입니다. 볼륨이나 드라이브 문자를 지정하지 않고 디렉터리 이름이 디렉터리 구분 문자로 시작하는 경우 경로는 현재 드라이브의 루트에서 상대적입니다. 그렇지 않으면 경로가 현재 디렉터리를 기준으로 합니다. 다음 표에서는 몇 가지 가능한 디렉터리 및 파일 경로를 보여 줍니다.

테이블 확장

경로	설명
`C:\Documents\Newsletters\Summer2018.pdf`	드라이브 `C:` 루트의 절대 파일 경로입니다.
`\Program Files\Custom Utilities\StringFinder.exe`	현재 드라이브의 루트를 기준으로 한 상대 경로입니다.
`2018\January.xlsx`	현재 디렉터리의 하위 디렉터리에 있는 파일에 대한 상대 경로입니다.
`..\Publications\TravelBrochure.pdf`	현재 디렉터리에서 시작하여 특정 디렉터리의 파일로 가는 상대 경로입니다.
`C:\Projects\apilibrary\apilibrary.sln`	드라이브 `C:` 의 루트에서 파일에 대한 절대 경로입니다.
`C:Projects\apilibrary\apilibrary.sln`	드라이브의 `C:` 현재 디렉터리에서의 상대 경로입니다.

> **중요**

마지막 두 경로 간의 차이점을 확인합니다. 둘 다 선택적 볼륨 지정자(`C:` 두 경우 모두)를 지정하지만 첫 번째는 지정된 볼륨의 루트로 시작하는 반면 두 번째 볼륨은 지정하지 않습니다. 결과적으로 첫 번째는 드라이브 `C:`의 루트 디렉터리에서 절대 경로인 반면 두 번째 경로는 드라이브 `C:`의 현재 디렉터리에서 상대 경로입니다. 첫 번째 양식이 의도된 경우 두 번째 양식을 사용하는 것은 Windows 파일 경로를 포함하는 버그의 일반적인 소스입니다.

메서드를 호출 Path.IsPathFullyQualified 하여 파일 경로가 정규화되었는지(즉, 경로가 현재 디렉터리와 독립적이며 현재 디렉터리가 변경되면 변경되지 않는 경우) 여부를 확인할 수 있습니다. 이러한 경로는 상대 디렉터리 세그먼트(`.` 및 `..`)를 포함할 수 있으며, 확인된 경로가 항상 동일한 위치를 가리키는 경우에도 완전히 해석된 경로로 간주될 수 있습니다.

다음 예제에서는 절대 경로와 상대 경로의 차이를 보여 줍니다. 디렉터리가 `D:\FY2018\` 있고 예제를 실행하기 전에 명령 프롬프트에서 현재 디렉터리를 `D:\` 설정하지 않았다고 가정합니다.

C#

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;

public class Example2
{
    public static void Main(string[] args)
    {
        Console.WriteLine($"Current directory is '{Environment.CurrentDirectory}'");
        Console.WriteLine("Setting current directory to 'C:\\'");

        Directory.SetCurrentDirectory(@"C:\");
        string path = Path.GetFullPath(@"D:\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");
        path = Path.GetFullPath(@"D:FY2018");
        Console.WriteLine($"'D:FY2018' resolves to {path}");

        Console.WriteLine("Setting current directory to 'D:\\Docs'");
        Directory.SetCurrentDirectory(@"D:\Docs");

        path = Path.GetFullPath(@"D:\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");
        path = Path.GetFullPath(@"D:FY2018");

        // This will be "D:\Docs\FY2018" as it happens to match the drive of the
current directory
        Console.WriteLine($"'D:FY2018' resolves to {path}");

        Console.WriteLine("Setting current directory to 'C:\\'");
        Directory.SetCurrentDirectory(@"C:\");

        path = Path.GetFullPath(@"D:\FY2018");
```

```
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");

        // This will be either "D:\FY2018" or "D:\FY2018\FY2018" in the subprocess.
In the sub process,
        // the command prompt set the current directory before launch of our
application, which
        // sets a hidden environment variable that is considered.
        path = Path.GetFullPath(@"D:FY2018");
        Console.WriteLine($"'D:FY2018' resolves to {path}");

        if (args.Length < 1)
        {
            Console.WriteLine(@"Launching again, after setting current directory to
D:\FY2018");
            Uri currentExe = new(Assembly.GetExecutingAssembly().Location,
UriKind.Absolute);
            string commandLine = $"/C cd D:\\FY2018 & \"{currentExe.LocalPath}\"
stop";
            ProcessStartInfo psi = new("cmd", commandLine); ;
            Process.Start(psi).WaitForExit();

            Console.WriteLine("Sub process returned:");
            path = Path.GetFullPath(@"D:\FY2018");
            Console.WriteLine($"'D:\\FY2018' resolves to {path}");
            path = Path.GetFullPath(@"D:FY2018");
            Console.WriteLine($"'D:FY2018' resolves to {path}");
        }
        Console.WriteLine("Press any key to continue... ");
        Console.ReadKey();
    }
}
// The example displays the following output:
//      Current directory is 'C:\Programs\file-paths'
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to d:\FY2018
//      Setting current directory to 'D:\Docs'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\Docs\FY2018
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to d:\FY2018
//      Launching again, after setting current directory to D:\FY2018
//      Sub process returned:
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to d:\FY2018
// The subprocess displays the following output:
//      Current directory is 'C:\'
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\FY2018\FY2018
//      Setting current directory to 'D:\Docs'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\Docs\FY2018
//      Setting current directory to 'C:\'
```

```
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\FY2018\FY2018
```

UNC 경로

네트워크 리소스에 액세스하는 데 사용되는 UNC(범용 명명 규칙) 경로의 형식은 다음과 같습니다.

- `\\`로 시작하는 서버 또는 호스트 이름입니다. 서버 이름은 NetBIOS 컴퓨터 이름 또는 IP/FQDN 주소(IPv4 및 v6 지원됨)일 수 있습니다.
- 공유 이름입니다. 이 이름은 호스트 이름 `\`에서 .로 구분됩니다. 서버와 공유 이름이 함께 볼륨을 구성합니다.
- 디렉터리 이름입니다. 디렉터리 구분 기호 문자는 중첩된 디렉터리 계층 내의 하위 디렉터리를 구분합니다.
- 선택적 파일 이름입니다. 디렉터리 구분 기호 문자는 파일 경로와 파일 이름을 구분합니다.

다음은 UNC 경로의 몇 가지 예입니다.

테이블 확장

경로	설명
`\\system07\C$\`	`C:`의 `system07` 드라이브 루트 디렉터리입니다.
`\\Server2\Share\Test\Foo.txt`	`Foo.txt` 볼륨의 Test 디렉터리에 있는 `\\Server2\Share` 파일.

UNC 경로는 항상 완전히 지정되어야 합니다. 상대 디렉터리 세그먼트(`.` 및 `..`)를 포함할 수 있지만 정규화된 경로의 일부여야 합니다. UNC 경로를 드라이브 문자에 매핑해야만 상대 경로를 사용할 수 있습니다.

DOS 디바이스 경로

Windows 운영 체제에는 파일을 포함한 모든 리소스를 가리키는 통합 개체 모델이 있습니다. 이러한 개체 경로는 콘솔 창에서 액세스할 수 있으며 레거시 DOS 및 UNC 경로가 매핑되는 기호화된 링크의 특수 폴더를 통해 Win32 계층에 노출됩니다. 이 특수 폴더는 다음 중 하나인 DOS 디바이스 경로 구문을 통해 액세스됩니다.

`\\.\C:\Test\Foo.txt` `\\?\C:\Test\Foo.txt`

드라이브 문자로 드라이브를 식별하는 것 외에도 볼륨 GUID를 사용하여 볼륨을 식별할 수 있습니다. 이 형식은 다음과 같습니다.

`\\.\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt` `\\?\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt`

> ⓘ **참고**
>
> DOS 디바이스 경로 구문은 .NET Core 1.1 및 .NET Framework 4.6.2부터 Windows에서 실행되는 .NET 구현에서 지원됩니다.

DOS 디바이스 경로는 다음 구성 요소로 구성됩니다.

- 경로를 DOS 디바이스 경로로 식별하는 디바이스 경로 지정자(`\\.\` 또는 `\\?\`)입니다.

  > ⓘ **참고**
  >
  > 모든 `\\?\` 버전의 .NET Core 및 .NET 5 이상 및 버전 4.6.2부터 .NET Framework에서 지원됩니다.

- "실제" 디바이스 개체에 대한 기호 링크(C: 드라이브 이름의 경우 또는 볼륨 GUID의 경우 Volume{b75e2c83-0000-0000-0000-602f00000000}).

  디바이스 경로 지정자가 볼륨 또는 드라이브를 식별한 후 DOS 디바이스 경로의 첫 번째 세그먼트입니다. (예: `\\?\C:\` 및 `\\.\BootPartition\`.)

  특정 UNC에 대한 링크가 있으며, 그 이름은 당연히 `UNC`입니다. 다음은 그 예입니다.

  `\\.\UNC\Server\Share\Test\Foo.txt` `\\?\UNC\Server\Share\Test\Foo.txt`

  디바이스 UNC의 경우 서버/공유 부분이 볼륨을 형성합니다. `\\?\server1\utilities\\filecomparer\`에서 예를 들어, 서버/공유 부분은 `server1\utilities` 입니다. 이는 상대 디렉터리 세그먼트와 같은 Path.GetFullPath(String, String) 메서드를 호출할 때 중요합니다. 볼륨을 지나 탐색할 수는 없습니다.

DOS 디바이스 경로는 정의에 따라 정규화되며 상대 디렉터리 세그먼트(`.` 또는 `..`)로 시작할 수 없습니다. 현재 디렉터리는 사용 측정에 포함되지 않습니다.

예: 동일한 파일을 참조하는 방법

다음 예제에서는 네임스페이스에서 API를 사용할 때 파일을 참조할 수 있는 System.IO 몇 가지 방법을 보여 줍니다. 이 예제에서는 개체를 FileInfo 인스턴스화하고 해당 Name 개체와 Length 속성을 사용하여 파일 이름과 파일 길이를 표시합니다.

C#

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string[] filenames = {
            @"c:\temp\test-file.txt",
            @"\\127.0.0.1\c$\temp\test-file.txt",
            @"\\LOCALHOST\c$\temp\test-file.txt",
            @"\\.\c:\temp\test-file.txt",
            @"\\?\c:\temp\test-file.txt",
            @"\\.\UNC\LOCALHOST\c$\temp\test-file.txt" };

        foreach (string filename in filenames)
        {
            FileInfo fi = new(filename);
            Console.WriteLine($"file {fi.Name}: {fi.Length:N0} bytes");
        }
    }
}
// The example displays output like the following:
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
```

경로 표준화

Windows API에 전달된 거의 모든 경로가 정규화됩니다. 정규화 중에 Windows는 다음 단계를 수행합니다.

- 경로를 식별합니다.
- 현재 디렉터리를 부분적으로 정규화된(상대) 경로에 적용합니다.
- 구성 요소 및 디렉터리 구분 기호를 정식화합니다.
- 상대 디렉터리 구성 요소(`.` 현재 디렉터리 및 `..` 부모 디렉터리의 경우)를 평가합니다.
- 특정 문자를 잘라냅니다.

이 정규화는 암시적으로 수행되지만 Path.GetFullPath에 대한 호출을 래핑하는 메서드를 호출 하여 명시적으로 수행할 수 있습니다. P/Invoke를 사용하여 Windows GetFullPathName() 함수 를 직접 호출할 수도 있습니다.

경로 식별

경로 정규화의 첫 번째 단계는 경로 유형을 식별하는 것입니다. 경로는 다음과 같은 몇 가지 범주 중 하나로 분류됩니다.

- 디바이스 경로입니다. 즉, 두 구분 기호와 물음표 또는 마침표(`\\?` 또는 `\\.`)로 시작합니다.
- UNC 경로입니다. 즉, 물음표나 마침표 없이 두 구분 기호로 시작합니다.
- 정규화된 DOS 경로입니다. 즉, 드라이브 문자, 볼륨 구분 기호 및 구성 요소 구분 기호(`C:\`)로 시작합니다.
- 레거시 디바이스(`CON`, `LPT1`)를 지정합니다.
- 현재 드라이브의 루트를 기준으로 합니다. 즉, 단일 구성 요소 구분 기호(`\`)로 시작합니다.
- 지정된 드라이브의 현재 디렉터리에 상대적입니다. 즉, 드라이브 문자로 시작하고 볼륨 구분 기호가 있으며 구성 요소 구분 기호(`C:`)는 없습니다.
- 현재 디렉터리를 기준으로 합니다. 즉, 다른 항목(`temp\testfile.txt`)으로 시작합니다.

경로의 형식은 현재 디렉터리가 어떤 방식으로 적용되는지 여부를 결정합니다. 경로의 "루트"도 결정합니다.

레거시 디바이스 처리

경로가 레거시 DOS 디바이스(예: `CON`또는`COM1` `LPT1`)인 경우 앞에 추가하여 `\\.\` 디바이스 경로로 변환되고 반환됩니다.

Windows 11 이전에는 레거시 디바이스 이름으로 시작하는 경로가 항상 메서드에 의해 Path.GetFullPath(String) 레거시 디바이스로 해석됩니다. 예를 들어 DOS 디바이스 경로 `CON.TXT`는 `\\.\CON`와 같으며, DOS 디바이스 경로 `COM1.TXT\file1.txt`는 `\\.\COM1`와 같습니다. Windows 11에서는 더 이상 적용되지 않으므로 레거시 DOS 디바이스의 전체 경로(예: `\\.\CON`)를 지정합니다.

현재 디렉터리 적용

경로가 정규화되지 않은 경우 Windows는 현재 디렉터리를 적용합니다. UNC 및 디바이스 경로에는 현재 디렉터리가 적용되지 않습니다. 구분 기호 `C:\`가 있는 전체 드라이브도 수행하지 않습니다.

경로가 단일 구성 요소 구분 기호로 시작하는 경우 현재 디렉터리의 드라이브가 적용됩니다. 예를 들어 파일 경로가 `\utilities`이고 현재 디렉터리가 `C:\temp\`인 경우, 정규화를 통해 `C:\utilities`가 생성됩니다.

경로가 드라이브 문자와 볼륨 구분 기호로 시작하고 구성 요소 구분 기호가 없는 경우, 지정된 드라이브에 대해 명령 셸에서 설정된 마지막 현재 디렉터리가 적용됩니다. 마지막 현재 디렉터리가 설정되지 않은 경우 드라이브만 적용됩니다. 예를 들어 파일 경로가 `D:sources`현재 디렉터

리이고 `C:\Documents\`D 드라이브 `D:\sources\`의 마지막 현재 디렉터리가 있는 경우 결과는 다음과 같습니다 `D:\sources\sources`. 이러한 "드라이브 상대" 경로는 프로그램 및 스크립트 논리 오류의 일반적인 소스입니다. 문자와 콜론으로 시작하는 경로가 상대 경로가 아니라고 가정하면 분명히 올바르지 않습니다.

경로가 구분 기호가 아닌 다른 항목으로 시작하는 경우 현재 드라이브와 현재 디렉터리가 적용됩니다. 예를 들어 경로가 `filecompare` 현재 디렉터리 `C:\utilities\`인 경우 결과는 다음과 같습니다 `C:\utilities\filecompare\`.

> **ⓘ 중요**
>
> 상대 경로는 현재 디렉터리가 프로세스별 설정이므로 다중 스레드 애플리케이션(즉, 대부분의 애플리케이션)에서 위험합니다. 모든 스레드는 언제든지 현재 디렉터리를 변경할 수 있습니다. .NET Core 2.1부터 Path.GetFullPath(String, String) 메서드를 호출하면 상대 경로와 기준 경로(현재 디렉터리)를 기반으로 절대 경로를 가져올 수 있습니다.

구분 기호 정규화

모든 정방향 슬래시(`/`)는 표준 Windows 구분 기호인 백슬래시(`\`)로 변환됩니다. 첫 두 슬래시 뒤에 오는 일련의 슬래시가 있을 경우, 그것들은 단일 슬래시로 축소됩니다.

> **ⓘ 참고**
>
> Unix 기반 운영 체제의 .NET 8부터 런타임은 더 이상 백 슬래시(`\`) 문자를 디렉터리 구분 기호(슬래시 `/`)로 변환하지 않습니다. 자세한 내용은 **Unix 파일 경로의 백슬래시 매핑을 참조하세요**.

상대 구성 요소 평가

경로가 처리되면 단일 또는 이중 기간(`.` 또는)으로 구성된 모든 구성 요소 또는 `..`세그먼트가 평가됩니다.

- 단일 기간 동안 현재 세그먼트는 현재 디렉터리를 참조하므로 제거됩니다.
- 이중 기간의 경우 이중 기간이 부모 디렉터리를 참조하므로 현재 세그먼트와 부모 세그먼트가 제거됩니다.

  부모 디렉터리는 경로의 루트를 초과하지 않는 경우에만 제거됩니다. 경로의 루트는 경로의 형식에 따라 달라집니다. DOS 경로의 드라이브(`C:\`), UNC의 서버/공유(`\\Server\Share`), 그리고 디바이스 경로(`\\?\` 또는 `\\.\`)의 디바이스 경로 접두사입니다.

문자 자르기

앞서 제거된 구분 기호 및 상대 세그먼트의 실행과 함께 정규화 중에 일부 추가 문자가 제거됩니다.

- 세그먼트가 단일 마침표로 끝나면 해당 기간이 제거됩니다. 단일 또는 이중 마침표의 세그먼트는 이전 단계에서 정규화됩니다. 세 개 이상의 마침표 세그먼트는 정규화되지 않으며, 실제로 정상적인 파일 또는 디렉터리 이름으로 간주됩니다.
- 경로가 구분 기호로 끝나지 않으면 모든 후행 마침표와 공백(U+0020)이 제거됩니다. 마지막 세그먼트가 단순히 단일 또는 이중 기간인 경우 위의 상대 구성 요소 규칙에 속합니다.

  이 규칙은 공백 뒤에 후행 구분 기호를 추가하여 후행 공백이 있는 디렉터리 이름을 만들 수 있음을 의미합니다.

  > **ⓘ 중요**
  >
  > 후행 공백이 있는 디렉터리 또는 파일 **이름을 만들면** 안 됩니다. 후행 공백을 사용하면 디렉터리에 액세스하기가 어렵거나 불가능할 수 있으며, 이름에 후행 공백이 포함된 디렉터리 또는 파일을 처리하려고 할 때 애플리케이션이 일반적으로 실패합니다.

정규화 건너뛰기

일반적으로 Windows API에 전달된 모든 경로는 GetFullPathName 함수 에 (효과적으로) 전달되고 정규화됩니다. 한 가지 중요한 예외가 있습니다. 마침표 대신 물음표로 시작하는 디바이스 경로입니다. 경로가 정확히 `\\?\` 시작되지 않는 한(정식 백슬래시 사용 참고) 정규화됩니다.

정규화를 건너뛰려는 이유는 무엇인가요? 세 가지 주요 이유가 있습니다.

1. 일반적으로 사용할 수 없지만 합법적인 경로에 액세스하려면 예를 들어 다른 방법으로는 액세스할 수 없는 파일 또는 디렉터리입니다 `hidden.`.
2. 이미 정규화된 경우 정규화를 건너뛰어 성능을 향상시킵니다.
3. .NET Framework에서만, 경로 길이 확인`MAX_PATH`을 생략하여 259자보다 큰 경로를 허용합니다. 대부분의 API는 일부 예외를 제외하고 이를 허용합니다.

> **ⓘ 참고**
>
> .NET Core 및 .NET 5+는 긴 경로를 암시적으로 처리하며 `MAX_PATH` 검사를 수행하지 않습니다. 검사는 `MAX_PATH` .NET Framework에만 적용됩니다.

정규화 및 최대 경로 검사를 건너뛰는 것은 두 디바이스 경로 구문 간의 유일한 차이점입니다. 그렇지 않으면 동일합니다. "일반" 애플리케이션에서 처리하기 어려운 경로를 쉽게 만들 수 있으므로 정규화를 건너뛰는 데 주의해야 합니다.

`\\?\`로 시작하는 경로는 GetFullPathName 함수에 명시적으로 전달할 때 여전히 정규화됩니다.

을 사용하지 않고`MAX_PATH`도 여러 문자의 경로를 `\\?\`에 전달할 수 있습니다. Windows에서 처리할 수 있는 최대 문자열 크기까지 임의의 길이 경로를 지원합니다.

사례 및 Windows 파일 시스템

Windows가 아닌 사용자와 개발자가 혼동하는 Windows 파일 시스템의 특수성은 경로 및 디렉터리 이름이 대/소문자를 구분하지 않는다는 것입니다. 즉, 디렉터리 및 파일 이름은 작성할 때 사용되는 문자열의 대문자와 소문자를 반영합니다. 예를 들어 메서드 호출

C#

```
Directory.Create("TeStDiReCtOrY");
```

는 TeStDiReCtOrY라는 디렉터리를 만듭니다. 디렉터리 또는 파일의 이름을 변경하여 대/소문자를 바꾸면, 디렉터리 또는 파일 이름은 변경 시 사용한 문자열의 대/소문자를 반영합니다. 예를 들어 다음 코드는 test.txt 파일 이름을 Test.txt바꿉니다.

C#

```
using System.IO;

class Example3
{
    static void Main()
    {
        var fi = new FileInfo(@".\test.txt");
        fi.MoveTo(@".\Test.txt");
    }
}
```

그러나 디렉터리 및 파일 이름 비교는 대/소문자를 구분하지 않습니다. "test.txt"라는 파일을 검색할 때, .NET 파일 시스템 API들은 대소문자를 구분하지 않고 비교합니다. "Test.txt", "TEST.TXT", "test.TXT" 및 대문자와 소문자의 다른 조합은 "test.txt"와 일치합니다.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

일반적인 I/O 작업

2025. 06. 17.

네임스페이 System.IO 스는 파일, 디렉터리 및 스트림에서 읽기 및 쓰기와 같은 다양한 작업을 수행할 수 있는 여러 클래스를 제공합니다. 자세한 내용은 파일 및 스트림 I/O를 참조하세요.

일반적인 파일 작업

테이블 확장

이 작업을 수행하려면...	이 항목의 예제를 참조하세요.
텍스트 파일 만들기	File.CreateText 메서드 FileInfo.CreateText 메서드 File.Create 메서드 FileInfo.Create 메서드
텍스트 파일에 쓰기	방법: 파일에 텍스트 쓰기 방법: 텍스트 파일 작성(C++/CLI)
텍스트 파일에서 읽기	방법: 파일에서 텍스트 읽기
파일에 텍스트 추가	방법: 로그 파일 열기 및 추가 File.AppendText 메서드 FileInfo.AppendText 메서드
파일 이름 바꾸기 또는 이동	File.Move 메서드 FileInfo.MoveTo 메서드
파일 삭제	File.Delete 메서드 FileInfo.Delete 메서드
파일 복사	File.Copy 메서드 FileInfo.CopyTo 메서드
파일 크기 가져오기	FileInfo.Length 속성
파일의 특성 가져오기	File.GetAttributes 메서드

이 작업을 수행하려면...	이 항목의 예제를 참조하세요.
파일의 특성 설정	File.SetAttributes 메서드
파일이 있는지 확인	File.Exists 메서드
이진 파일에서 읽기	방법: 새로 만든 데이터 파일 읽기 및 쓰기
이진 파일에 쓰기	방법: 새로 만든 데이터 파일 읽기 및 쓰기
파일 이름 확장명 검색	Path.GetExtension 메서드
파일의 완전한 경로 검색	Path.GetFullPath 메서드
경로에서 파일 이름 및 확장명 검색	Path.GetFileName 메서드
파일의 확장 프로그램 변경	Path.ChangeExtension 메서드

공통 디렉터리 작업

테이블 확장

이 작업을 수행하려면...	이 항목의 예제를 참조하세요.
내 문서와 같은 특수 폴더의 파일에 액세스	방법: 파일에 텍스트 쓰기
디렉터리 만들기	Directory.CreateDirectory 메서드 FileInfo.Directory 속성
하위 디렉터리 만들기	DirectoryInfo.CreateSubdirectory 메서드
디렉터리 이름 바꾸기 또는 이동	Directory.Move 메서드 DirectoryInfo.MoveTo 메서드
디렉터리 복사	방법: 디렉터리 복사
디렉터리 삭제	Directory.Delete 메서드 DirectoryInfo.Delete 메서드
디렉터리의 파일 및 하위 디렉터리 보기	방법: 디렉터리 및 파일 열거
디렉터리 크기 찾기	System.IO.Directory 클래스
디렉터리가 있는지 확인	Directory.Exists 메서드

참고하십시오

- 파일 및 스트림 I/O
- 스트림 작성
- 비동기 파일 I/O

방법: 디렉터리 복사

아티클 • 2024. 03. 14.

이 문서에서는 I/O 클래스를 사용하여 디렉터리의 내용을 다른 위치로 동기적으로 복사하는 방법을 보여 줍니다.

비동기 파일 복사의 예제는 비동기 파일 I/O을 참조하세요.

이 예에서는 `CopyDirectory` 메서드의 `recursive` 매개 변수를 `true`로 설정하여 하위 디렉터리를 복사합니다. `CopyDirectory` 메서드는 더 이상 복사할 항목이 없을 때까지 각 하위 디렉터리에서 자신을 호출하여 하위 디렉터리를 재귀적으로 복사합니다.

예시

C#

```
using System.IO;

CopyDirectory(@".\", @".\copytest", true);

static void CopyDirectory(string sourceDir, string destinationDir, bool
recursive)
{
    // Get information about the source directory
    var dir = new DirectoryInfo(sourceDir);

    // Check if the source directory exists
    if (!dir.Exists)
        throw new DirectoryNotFoundException($"Source directory not found:
{dir.FullName}");

    // Cache directories before we start copying
    DirectoryInfo[] dirs = dir.GetDirectories();

    // Create the destination directory
    Directory.CreateDirectory(destinationDir);

    // Get the files in the source directory and copy to the destination
directory
    foreach (FileInfo file in dir.GetFiles())
    {
        string targetFilePath = Path.Combine(destinationDir, file.Name);
        file.CopyTo(targetFilePath);
    }

    // If recursive and copying subdirectories, recursively call this method
    if (recursive)
    {
```

```
        foreach (DirectoryInfo subDir in dirs)
        {
            string newDestinationDir = Path.Combine(destinationDir,
subDir.Name);
            CopyDirectory(subDir.FullName, newDestinationDir, true);
        }
    }
}
```

참고 항목

- FileInfo
- DirectoryInfo
- FileStream
- 파일 및 스트림 I/O
- 공통 I/O 작업
- 비동기 파일 I/O

방법: 디렉터리 및 파일 열거

아티클 • 2023. 04. 08.

열거 가능한 컬렉션은 대규모의 디렉터리 및 파일 컬렉션으로 작업할 때 배열보다 나은 성능을 제공합니다. 디렉터리 및 파일을 열거하려면 디렉터리, 파일 이름이나 해당 DirectoryInfo, FileInfo 또는 FileSystemInfo 개체의 열거할 수 있는 컬렉션을 반환하는 메서드를 사용합니다.

디렉터리 또는 파일의 이름만 검색하고 반환하려면 Directory 클래스의 열거 메서드를 사용합니다. 디렉터리 또는 파일의 다른 속성을 검색하고 반환하려면 DirectoryInfo 및 FileSystemInfo 클래스를 사용합니다.

이러한 메서드의 열거 가능한 컬렉션은 List<T>과 같은 컬렉션 클래스의 생성자에 대한 IEnumerable<T> 매개 변수로 사용할 수 있습니다.

다음 표에는 열거 가능한 파일 및 디렉터리 컬렉션을 반환하는 메서드가 요약되어 있습니다.

검색 및 반환하려면	메서드 사용
디렉터리 이름	Directory.EnumerateDirectories
디렉터리 정보(DirectoryInfo)	DirectoryInfo.EnumerateDirectories
파일 이름	Directory.EnumerateFiles
파일 정보(FileInfo)	DirectoryInfo.EnumerateFiles
파일 시스템 항목 이름	Directory.EnumerateFileSystemEntries
파일 시스템 항목 정보(FileSystemInfo)	DirectoryInfo.EnumerateFileSystemInfos
디렉터리 및 파일 이름	Directory.EnumerateFileSystemEntries

> ⓘ **참고**
>
> 선택적 SearchOption 열거형의 AllDirectories 옵션을 사용하면 부모 디렉터리의 하위 디렉터리에 있는 모든 파일을 즉시 열거할 수 있지만, UnauthorizedAccessException 오류로 인해 열거가 불완전할 수 있습니다. 먼저 디렉터리를 열거한 다음, 파일을 열거하면 이러한 예외를 catch할 수 있습니다.

예: 디렉터리 클래스 사용

다음 예제에서는 Directory.EnumerateDirectories(String) 메서드를 사용하여 지정된 경로의 최상위 디렉터리 이름 목록을 가져옵니다.

C#

```
using System;
using System.Collections.Generic;
using System.IO;

class Program
{
    private static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            List<string> dirs = new List<string>
(Directory.EnumerateDirectories(docPath));

            foreach (var dir in dirs)
            {
                Console.WriteLine($"
{dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)}");
            }
            Console.WriteLine($"{dirs.Count} directories found.");
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.WriteLine(ex.Message);
        }
        catch (PathTooLongException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

다음 예제에서는 Directory.EnumerateFiles(String, String, SearchOption) 메서드를 사용하여 특정 패턴과 일치하는 디렉터리 및 하위 디렉터리의 모든 파일 이름을 재귀적으로 열거합니다. 그런 다음, 각 파일의 각 줄을 읽고 해당 파일 이름 및 경로가 있는 지정된 문자열이 포함된 줄을 표시합니다.

C#

```
using System;
using System.IO;
using System.Linq;
```

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =

Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            var files = from file in Directory.EnumerateFiles(docPath,
"*.txt", SearchOption.AllDirectories)
                        from line in File.ReadLines(file)
                        where line.Contains("Microsoft")
                        select new
                        {
                            File = file,
                            Line = line
                        };

            foreach (var f in files)
            {
                Console.WriteLine($"{f.File}\t{f.Line}");
            }
            Console.WriteLine($"{files.Count().ToString()} files found.");
        }
        catch (UnauthorizedAccessException uAEx)
        {
            Console.WriteLine(uAEx.Message);
        }
        catch (PathTooLongException pathEx)
        {
            Console.WriteLine(pathEx.Message);
        }
    }
}
```

예: DirectoryInfo 클래스 사용

다음 예제에서는 DirectoryInfo.EnumerateDirectories 메서드를 사용하여 CreationTimeUtc가 특정 DateTime 값보다 이전인 최상위 디렉터리의 컬렉션을 나열합니다.

C#

```
using System;
using System.IO;

namespace EnumDir
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            // Set a variable to the Documents path.
            string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            DirectoryInfo dirPrograms = new DirectoryInfo(docPath);
            DateTime StartOf2009 = new DateTime(2009, 01, 01);

            var dirs = from dir in dirPrograms.EnumerateDirectories()
            where dir.CreationTimeUtc > StartOf2009
            select new
            {
                ProgDir = dir,
            };

            foreach (var di in dirs)
            {
                Console.WriteLine($"{di.ProgDir.Name}");
            }
        }
    }
}
// </Snippet1>
```

다음 예제에서는 DirectoryInfo.EnumerateFiles 메서드를 사용하여 Length가 10MB를 초과하는 모든 파일을 나열합니다. 이 예제는 먼저 최상위 디렉터리를 열거하여 가능한 권한 없는 액세스 예외를 catch하고 파일을 열거합니다.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        // Set a variable to the My Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        DirectoryInfo diTop = new DirectoryInfo(docPath);

        try
        {
            foreach (var fi in diTop.EnumerateFiles())
            {
                try
                {
```

```
                    // Display each file over 10 MB;
                    if (fi.Length > 10000000)
                    {
                        Console.WriteLine($"
{fi.FullName}\t\t{fi.Length.ToString("N0")}");
                    }
                }
                catch (UnauthorizedAccessException unAuthTop)
                {
                    Console.WriteLine($"{unAuthTop.Message}");
                }
            }

            foreach (var di in diTop.EnumerateDirectories("*"))
            {
                try
                {
                    foreach (var fi in di.EnumerateFiles("*",
SearchOption.AllDirectories))
                    {
                        try
                        {
                            // Display each file over 10 MB;
                            if (fi.Length > 10000000)
                            {
                                Console.WriteLine($"
{fi.FullName}\t\t{fi.Length.ToString("N0")}");
                            }
                        }
                        catch (UnauthorizedAccessException unAuthFile)
                        {
                            Console.WriteLine($"unAuthFile:
{unAuthFile.Message}");
                        }
                    }
                }
                catch (UnauthorizedAccessException unAuthSubDir)
                {
                    Console.WriteLine($"unAuthSubDir:
{unAuthSubDir.Message}");
                }
            }
        }
        catch (DirectoryNotFoundException dirNotFound)
        {
            Console.WriteLine($"{dirNotFound.Message}");
        }
        catch (UnauthorizedAccessException unAuthDir)
        {
            Console.WriteLine($"unAuthDir: {unAuthDir.Message}");
        }
        catch (PathTooLongException longPath)
        {
            Console.WriteLine($"{longPath.Message}");
        }
```

```
    }
}
```

참조

- 파일 및 스트림 I/O

방법: 새로 만든 데이터 파일 읽기 및 쓰기

아티클 • 2025. 04. 30.

및 System.IO.BinaryWriter 클래스는 System.IO.BinaryReader 문자열 이외의 데이터를 작성하고 읽는 데 사용됩니다. 다음 예제에서는 빈 파일 스트림을 만들고, 빈 파일 스트림에 데이터를 쓰고, 해당 스트림에서 데이터를 읽는 방법을 보여줍니다.

이 예제에서는 현재 디렉터리에 *Test.data* 라는 데이터 파일을 만들고, 연결된 BinaryWriter 개체와 BinaryReader 개체를 만들고, 개체를 사용하여 BinaryWriter 0에서 10까지의 정수를 *Test.data* 에 씁니다. 그러면 파일 포인터가 파일 끝에 남습니다. 그런 다음 개체는 BinaryReader 파일 포인터를 원본으로 다시 설정하고 지정된 콘텐츠를 읽습니다.

> ⓘ **참고**
>
> *Test.data*가 현재 디렉터리에 이미 있는 경우 예외가 IOException throw됩니다. FileMode.Create 파일 모드 옵션을 사용하여 항상 새 파일을 만들고, FileMode.CreateNew 를 사용하지 않으면 예외가 발생하지 않습니다.

예시

C#

```
using System;
using System.IO;

class MyStream
{
    private const string FILE_NAME = "Test.data";

    public static void Main()
    {
        if (File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} already exists!");
            return;
        }

        using (FileStream fs = new FileStream(FILE_NAME, FileMode.CreateNew))
        {
            using (BinaryWriter w = new BinaryWriter(fs))
            {
                for (int i = 0; i < 11; i++)
                {
                    w.Write(i);
                }
            }
```

```
        }

        using (FileStream fs = new FileStream(FILE_NAME, FileMode.Open,
FileAccess.Read))
        {
            using (BinaryReader r = new BinaryReader(fs))
            {
                for (int i = 0; i < 11; i++)
                {
                    Console.WriteLine(r.ReadInt32());
                }
            }
        }
    }
}


// The example creates a file named "Test.data" and writes the integers 0 through
10 to it in binary format.
// It then writes the contents of Test.data to the console with each integer on a
separate line.
```

참고하십시오

- BinaryReader
- BinaryWriter
- FileStream
- FileStream.Seek
- SeekOrigin
- 방법: 디렉터리 및 파일 열거
- 방법: 로그 파일 열기 및 추가
- 방법: 파일에서 텍스트 읽기
- 방법: 파일에 텍스트 쓰기
- 방법: 문자열에서 문자 읽기
- 방법: 문자열에 문자 쓰기
- 파일 및 스트림 I/O

방법: 로그 파일 열기 및 추가

아티클 • 2024. 03. 12.

StreamWriter 및 StreamReader는 스트림에서 문자를 쓰고 문자를 읽습니다. 다음 코드 예제는 입력을 위해 *log.txt* 파일을 열거나, 파일이 존재하지 않는 경우 파일을 만들고 파일의 끝에 로그 정보를 추가합니다. 이 예제는 파일의 콘텐츠를 디스플레이의 표준 출력에 씁니다.

이 예제의 대안으로, 정보를 단일 문자열 또는 문자열 배열로 저장하고, File.WriteAllText 또는 File.WriteAllLines 메서드를 사용하여 동일한 기능을 수행할 수도 있습니다.

> ⓘ **참고**
>
> Visual Basic 사용자는 로그 파일을 생성하거나 로그 파일에 쓰기 위해 Log 클래스 또는 FileSystem 클래스에서 제공하는 메서드 및 속성을 사용하도록 선택할 수 있습니다.

예시

C#

```
using System;
using System.IO;

class DirAppend
{
    public static void Main()
    {
        using (StreamWriter w = File.AppendText("log.txt"))
        {
            Log("Test1", w);
            Log("Test2", w);
        }

        using (StreamReader r = File.OpenText("log.txt"))
        {
            DumpLog(r);
        }
    }

    public static void Log(string logMessage, TextWriter w)
    {
        w.Write("\r\nLog Entry : ");
        w.WriteLine($"{DateTime.Now.ToLongTimeString()}
{DateTime.Now.ToLongDateString()}");
        w.WriteLine("  :");
```

```
        w.WriteLine($"  :{logMessage}");
        w.WriteLine ("-------------------------------");
    }

    public static void DumpLog(StreamReader r)
    {
        string line;
        while ((line = r.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
}
// The example creates a file named "log.txt" and writes the following lines
to it,
// or appends them to the existing "log.txt" file:

// Log Entry : <current long time string> <current long date string>
//  :
//  :Test1
// -------------------------------

// Log Entry : <current long time string> <current long date string>
//  :
//  :Test2
// -------------------------------

// It then writes the contents of "log.txt" to the console.
```

참고 항목

- StreamWriter
- StreamReader
- File.AppendText
- File.OpenText
- StreamReader.ReadLine
- 방법: 디렉터리 및 파일 열거
- 방법: 새로 만든 데이터 파일 읽기 및 쓰기
- 방법: 파일에서 텍스트 읽기
- 방법: 파일에 텍스트 쓰기
- 방법: 문자열에서 문자 읽기
- 방법: 문자열에 문자 쓰기
- 파일 및 스트림 I/O

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

방법: 파일에 텍스트 쓰기

이 문서에서는 .NET 앱의 파일에 텍스트를 작성하는 다양한 방법을 보여줍니다.

파일에 텍스트를 쓸 때는 일반적으로 다음 클래스 및 메서드가 사용됩니다.

- StreamWriter에는 동기식(Write 및 WriteLine) 또는 비동기식(WriteAsync 및 WriteLineAsync)으로 파일에 쓰는 메서드가 포함되어 있습니다.
- File은 WriteAllLines 및 WriteAllText와 같은 파일에 텍스트를 쓰거나 AppendAllLines, AppendAllText 및 AppendText와 같은 파일에 텍스트를 추가하는 정적 메서드를 제공합니다.
- Path는 파일 또는 디렉터리 경로 정보가 포함된 문자열에 대한 것입니다. Combine 메서드를 포함하며, .NET Core 2.1 이상에서는 Join 및 TryJoin 메서드를 포함합니다. 이러한 메서드를 사용하면 파일 또는 디렉터리 경로를 빌드하기 위한 문자열을 연결할 수 있습니다.

> **참고**
>
> 다음 예제에서는 필요한 최소 코드 양만 보여줍니다. 실제 앱은 일반적으로 더 강력한 오류 검사 및 예외 처리 기능을 제공합니다.

예: StreamWriter를 사용하여 텍스트를 동기식으로 쓰기

다음 예제에서는 StreamWriter 클래스를 사용하여 한 번에 한 줄씩 새 파일에 텍스트를 동기식으로 쓰는 방법을 보여줍니다. StreamWriter 개체는 `using` 문에서 선언되고 인스턴스화되므로 스트림을 자동으로 플러시하고 닫는 Dispose 메서드가 호출됩니다.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {

        // Create a string array with the lines of text
        string[] lines = { "First line", "Second line", "Third line" };

        // Set a variable to the Documents path.
        string docPath =
          Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

```
        // Write the string array to a new file named "WriteLines.txt".
        using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath,
"WriteLines.txt")))
        {
            foreach (string line in lines)
                outputFile.WriteLine(line);
        }
    }
}
// The example creates a file named "WriteLines.txt" with the following contents:
// First line
// Second line
// Third line
```

예: StreamWriter를 사용하여 텍스트를 동기식으로 추가

다음 예제에서는 StreamWriter 클래스를 사용하여 첫 번째 예제에서 만든 텍스트 파일에 텍스트를 동기식으로 추가하는 방법을 보여줍니다.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {

        // Set a variable to the Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Append text to an existing file named "WriteLines.txt".
        using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath,
"WriteLines.txt"), true))
        {
            outputFile.WriteLine("Fourth Line");
        }
    }
}
// The example adds the following line to the contents of "WriteLines.txt":
// Fourth Line
```

예: StreamWriter를 사용하여 텍스트를 비동기식으로 쓰기

다음 예제에서는 StreamWriter 클래스를 사용하여 새 파일에 비동기적으로 텍스트를 쓰는 방법을 보여 줍니다. WriteAsync 메서드를 호출하려면 메서드 호출이 `async` 메서드 내에 있어야 합니다.

C#

```
using System;
using System.IO;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        // Set a variable to the Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Write the specified text asynchronously to a new file named
"WriteTextAsync.txt".
        using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath,
"WriteTextAsync.txt")))
        {
            await outputFile.WriteAsync("This is a sentence.");
        }
    }
}
// The example creates a file named "WriteTextAsync.txt" with the following
contents:
// This is a sentence.
```

예: 파일 클래스로 텍스트 쓰기 및 추가

다음 예제에서는 File 클래스를 사용하여 새 파일에 텍스트를 쓰고 동일한 파일에 새 텍스트 줄을 추가하는 방법을 보여 줍니다. WriteAllText 및 AppendAllLines 메서드는 자동으로 파일을 열고 닫습니다. WriteAllText 메서드에 제공한 경로가 이미 있는 경우 파일을 덮어씁니다.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
```

```
    {
        // Create a string with a line of text.
        string text = "First line" + Environment.NewLine;

        // Set a variable to the Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Write the text to a new file named "WriteFile.txt".
        File.WriteAllText(Path.Combine(docPath, "WriteFile.txt"), text);

        // Create a string array with the additional lines of text
        string[] lines = { "New line 1", "New line 2" };

        // Append new lines of text to the file
        File.AppendAllLines(Path.Combine(docPath, "WriteFile.txt"), lines);
    }
}
// The example creates a file named "WriteFile.txt" with the contents:
// First line
// And then appends the following contents:
// New line 1
// New line 2
```

참조

- StreamWriter
- Path
- File.CreateText
- 방법: 디렉터리 및 파일 열거
- 방법: 새로 만든 데이터 파일 읽기 및 쓰기
- 방법: 로그 파일 열기 및 추가
- 방법: 파일에서 텍스트 읽기
- 파일 및 스트림 I/O

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 21.

파일의 텍스트 읽기

아티클 • 2024. 05. 12.

다음 예제에서는 데스크톱 응용 프로그램용 .NET을 사용하여 텍스트 파일에서 텍스트를 동기 또는 비동기적으로 읽는 방법을 보여줍니다. 두 예제에서는 StreamReader 클래스의 인스턴스를 만들 때 파일의 상대 또는 절대 경로를 제공합니다.

> ⓘ **참고**
>
> Windows 런타임에서는 파일을 읽고 파일에 쓰는 다양한 스트림 형식을 제공하기 때문에 관련 코드 예제는 UWP(유니버설 Windows 플랫폼) 앱에는 적용되지 않습니다. 자세한 내용은 **UWP 파일 작업**을 참조하세요. .NET Framework 스트림과 Windows 런타임 스트림 간의 변환 방법을 보여주는 예제는 **방법: .NET Framework 스트림과 Windows 런타임 스트림 간 변환**을 참조하세요.

필수 조건

- 앱과 동일한 폴더에 *TestFile.txt*라는 텍스트 파일을 만듭니다.

  텍스트 파일에 일부 콘텐츠를 추가합니다. 이 문서의 예에서는 텍스트 파일의 콘텐츠를 콘솔에 씁니다.

파일 읽기

다음 예제에서는 콘솔 앱 내에서 동기식 읽기 작업을 보여줍니다. 파일 콘텐츠를 읽고 문자열 변수에 저장한 다음 콘솔에 씁니다.

1. StreamReader 인스턴스를 만듭니다.
2. StreamReader.ReadToEnd() 메서드를 호출하고 결과를 문자열에 할당합니다.
3. 콘솔에 출력을 씁니다.

C#

```
try
{
    // Open the text file using a stream reader.
    using StreamReader reader = new("TestFile.txt");

    // Read the stream as a string.
    string text = reader.ReadToEnd();

    // Write the text to the console.
```

```
    Console.WriteLine(text);
}
catch (IOException e)
{
    Console.WriteLine("The file could not be read:");
    Console.WriteLine(e.Message);
}
```

비동기적으로 파일 읽기

다음 예에서는 콘솔 앱 내의 비동기 읽기 작업을 보여 줍니다. 파일 콘텐츠를 읽고 문자열 변수에 저장한 다음 콘솔에 씁니다.

1. StreamReader 인스턴스를 만듭니다.
2. StreamReader.ReadToEndAsync() 메서드를 기다리고 결과를 문자열에 할당합니다.
3. 콘솔에 출력을 씁니다.

C#

```
try
{
    // Open the text file using a stream reader.
    using StreamReader reader = new("TestFile.txt");

    // Read the stream as a string.
    string text = await reader.ReadToEndAsync();

    // Write the text to the console.
    Console.WriteLine(text);
}
catch (IOException e)
{
    Console.WriteLine("The file could not be read:");
    Console.WriteLine(e.Message);
}
```

관련 콘텐츠

- 공통적인 I/O 작업.
- 비동기 파일 I/O.
- 방법: 파일에 텍스트 쓰기.
- StreamReader
- File.OpenText
- StreamReader.ReadLine

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

방법: 문자열에서 문자 읽기

아티클 • 2025. 05. 04.

다음 코드 예제에서는 문자열에서 동기 또는 비동기적으로 문자를 읽는 방법을 보여 줍니다.

예: 동기식으로 문자 읽기

이 예제에서는 문자열에서 동기적으로 13자를 읽고, 배열에 저장하고, 표시합니다. 그런 다음, 문자열의 나머지 문자를 읽고, 여섯 번째 요소에서 시작하는 배열에 저장하고, 배열의 내용을 표시합니다.

C#

```
using System;
using System.IO;

public class CharsFromStr
{
    public static void Main()
    {
        string str = "Some number of characters";
        char[] b = new char[str.Length];

        using (StringReader sr = new StringReader(str))
        {
            // Read 13 characters from the string into the array.
            sr.Read(b, 0, 13);
            Console.WriteLine(b);

            // Read the rest of the string starting at the current string
position.
            // Put in the array starting at the 6th array member.
            sr.Read(b, 5, str.Length - 13);
            Console.WriteLine(b);
        }
    }
}
// The example has the following output:
//
// Some number o
// Some f characters
```

예: 문자를 비동기적으로 읽습니다.

다음 예제는 WPF 앱 뒤에 있는 코드입니다. 창 로드 시 예제에서는 컨트롤의 TextBox 모든 문자를 비동기적으로 읽고 배열에 저장합니다. 그런 다음 각 문자 또는 공백 문자를 컨트롤의 TextBlock 별도 줄에 비동기적으로 씁니다.

C#

```
using System;
using System.Text;
using System.Windows;
using System.IO;

namespace StringReaderWriter
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Window_Loaded(object sender, RoutedEventArgs e)
        {
            char[] charsRead = new char[UserInput.Text.Length];
            using (StringReader reader = new StringReader(UserInput.Text))
            {
                await reader.ReadAsync(charsRead, 0, UserInput.Text.Length);
            }

            StringBuilder reformattedText = new StringBuilder();
            using (StringWriter writer = new StringWriter(reformattedText))
            {
                foreach (char c in charsRead)
                {
                    if (char.IsLetter(c) || char.IsWhiteSpace(c))
                    {
                        await writer.WriteLineAsync(char.ToLower(c));
                    }
                }
            }
            Result.Text = reformattedText.ToString();
        }
    }
}
```

참고하십시오

- StringReader
- StringReader.Read
- 비동기 파일 I/O
- 방법: 디렉터리 목록 만들기
- 방법: 새로 만든 데이터 파일 읽기 및 쓰기

- 방법: 로그 파일 열기 및 추가
- 방법: 파일에서 텍스트 읽기
- 방법: 파일에 텍스트 쓰기
- 방법: 문자열에 문자 쓰기
- 파일 및 스트림 I/O

방법: 문자열에 문자 쓰기

아티클 • 2023. 04. 07.

다음 코드 예제는 동기식 또는 비동기식으로 문자 배열에서 문자열로 문자를 씁니다.

예: 콘솔 앱에서 동기적으로 문자 쓰기

다음 예제에서는 StringWriter를 사용하여 동기식으로 StringBuilder 개체에 문자 5개를 씁니다.

C#

```
using System;
using System.IO;
using System.Text;

public class CharsToStr
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("Start with a string and add
from ");
        char[] b = { 'c', 'h', 'a', 'r', '.', ' ', 'B', 'u', 't', ' ', 'n',
'o', 't', ' ', 'a', 'l', 'l' };

        using (StringWriter sw = new StringWriter(sb))
        {
            // Write five characters from the array into the StringBuilder.
            sw.Write(b, 0, 5);
            Console.WriteLine(sb);
        }
    }
}
// The example has the following output:
//
// Start with a string and add from char.
```

예: WPF 앱에서 비동기적으로 문자 쓰기

다음 예제는 WPF 앱의 코드입니다. 창 로드 시, 이 예제는 TextBox 컨트롤에서 모든 문자를 비동기식으로 읽고 배열에 저장합니다. 그런 다음, 각 문자 또는 공백 문자를 TextBlock 컨트롤의 별도 라인에 비동기식으로 씁니다.

C#

```
using System;
using System.Text;
using System.Windows;
using System.IO;

namespace StringReaderWriter
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Window_Loaded(object sender, RoutedEventArgs e)
        {
            char[] charsRead = new char[UserInput.Text.Length];
            using (StringReader reader = new StringReader(UserInput.Text))
            {
                await reader.ReadAsync(charsRead, 0, UserInput.Text.Length);
            }

            StringBuilder reformattedText = new StringBuilder();
            using (StringWriter writer = new StringWriter(reformattedText))
            {
                foreach (char c in charsRead)
                {
                    if (char.IsLetter(c) || char.IsWhiteSpace(c))
                    {
                        await writer.WriteLineAsync(char.ToLower(c));
                    }
                }
            }
            Result.Text = reformattedText.ToString();
        }
    }
}
```

참조

- StringWriter
- StringWriter.Write
- StringBuilder
- 파일 및 스트림 I/O
- 비동기 파일 I/O
- 방법: 디렉터리 및 파일 열거

- 방법: 새로 만든 데이터 파일 읽기 및 쓰기
- 방법: 로그 파일 열기 및 추가
- 방법: 파일에서 텍스트 읽기
- 방법: 파일에 텍스트 쓰기
- 방법: 문자열에서 문자 읽기

방법: 액세스 제어 목록 항목 추가 또는 제거

아티클 • 2024. 06. 11.

파일 또는 디렉터리에서 ACL(액세스 제어 목록) 항목을 추가하거나 제거하려면 파일 또는 디렉터리에서 개체를 DirectorySecurity 가져옵니다FileSecurity. 개체를 수정한 다음, 파일이나 디렉터리에 다시 적용합니다.

파일에서

1. FileSystemAclExtensions.GetAccessControl(FileInfo) (또는 .NET Framework 앱 FileInfo.GetAccessControl의 경우) 메서드를 호출하여 파일의 현재 ACL 항목을 포함하는 개체를 가져옵니다FileSecurity.
2. 1단계에서 가져온 개체에서 FileSecurity ACL 항목을 추가하거나 제거합니다.
3. 변경 내용을 적용하려면 개체를 (또는 .NET Framework 앱FileInfo.SetAccessControl의 FileSystemAclExtensions.SetAccessControl(FileInfo, FileSecurity) 경우) 메서드에 전달 FileSecurity 합니다.

디렉터리에서

1. 디렉터리의 FileSystemAclExtensions.GetAccessControl(DirectoryInfo) 현재 ACL 항목을 포함하는 개체를 가져오 DirectorySecurity 려면 (또는 .NET Framework 앱 DirectoryInfo.GetAccessControl의 경우) 메서드를 호출합니다.
2. 1단계에서 가져온 개체에서 DirectorySecurity ACL 항목을 추가하거나 제거합니다.
3. 변경 내용을 적용하려면 개체를 (또는 .NET Framework 앱 DirectoryInfo.SetAccessControl의 FileSystemAclExtensions.SetAccessControl(DirectoryInfo, DirectorySecurity) 경우) 메서드에 전달 DirectorySecurity 합니다.

예시

이 예제를 실행하려면 유효한 사용자 또는 그룹 계정을 지정해야 합니다. 예제에서는 FileInfo 개체를 사용합니다. 클래스에 대해 동일한 프로시저를 DirectoryInfo 사용합니다.

```
C#
```

```
using System;
using System.IO;
using System.Security.AccessControl;

    class FileExample
    {
        public static void Main()
        {
            try
            {
                string fileName = "test.xml";

                Console.WriteLine($"Adding access control entry for
{fileName}");

                // Add the access control entry to the file.
                AddFileSecurity(fileName, @"DomainName\AccountName",
                    FileSystemRights.ReadData, AccessControlType.Allow);

                Console.WriteLine($"Removing access control entry from
{fileName}");

                // Remove the access control entry from the file.
                RemoveFileSecurity(fileName, @"DomainName\AccountName",
                    FileSystemRights.ReadData, AccessControlType.Allow);

                Console.WriteLine("Done.");
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }

        // Adds an ACL entry on the specified file for the specified
account.
        public static void AddFileSecurity(string fileName, string account,
            FileSystemRights rights, AccessControlType controlType)
        {
            FileInfo fileInfo = new(fileName);
            FileSecurity fSecurity = fileInfo.GetAccessControl();

            // Add the FileSystemAccessRule to the security settings.
            fSecurity.AddAccessRule(new FileSystemAccessRule(account,
                rights, controlType));

            // Set the new access settings.
            fileInfo.SetAccessControl(fSecurity);
        }

        // Removes an ACL entry on the specified file for the specified
account.
        public static void RemoveFileSecurity(string fileName, string
account,
```

```
            FileSystemRights rights, AccessControlType controlType)
        {
            FileInfo fileInfo = new(fileName);
            FileSecurity fSecurity = fileInfo.GetAccessControl();

            // Remove the FileSystemAccessRule from the security settings.
            fSecurity.RemoveAccessRule(new FileSystemAccessRule(account,
                rights, controlType));

            // Set the new access settings.
            fileInfo.SetAccessControl(fSecurity);
        }
}
```

방법: 파일 압축 및 추출

아티클 • 2023. 04. 08.

System.IO.Compression 네임스페이스는 파일 및 스트림을 압축하고 압축을 푸는 다음 클래스를 포함합니다. 또한 이러한 형식을 사용하여 압축된 파일의 내용을 읽고 수정할 수 있습니다.

- ZipFile
- ZipArchive
- ZipArchiveEntry
- DeflateStream
- GZipStream

다음 예제에서는 압축된 파일로 수행할 수 있는 일부 작업을 보여줍니다. 이러한 예제를 수행하려면 다음 NuGet 패키지를 프로젝트에 추가해야 합니다.

- System.IO.Compression
- System.IO.Compression.ZipFile

.NET Framework를 사용하는 경우 다음 두 라이브러리에 대한 참조를 프로젝트에 추가합니다.

- `System.IO.Compression`
- `System.IO.Compression.FileSystem`

예제 1: .zip 파일 만들기 및 추출

다음 예제에서는 ZipFile 클래스를 사용하여 압축된 *.zip* 파일을 만들고 추출하는 방법을 보여줍니다. 이 예제는 폴더의 콘텐츠를 새로운 *.zip* 파일로 압축한 다음, 파일을 새 폴더에 추출합니다.

샘플을 실행하려면 프로그램 폴더에 *start* 폴더를 만들어서 zip 파일로 채웁니다.

C#

```
using System;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string startPath = @".\start";
        string zipPath = @".\result.zip";
```

```
        string extractPath = @".\extract";

        ZipFile.CreateFromDirectory(startPath, zipPath);

        ZipFile.ExtractToDirectory(zipPath, extractPath);
    }
}
```

예제 2: 특정 파일 확장명 추출

다음 예제는 기존 *.zip* 파일의 콘텐츠를 반복하고 확장명이 *.txt*인 파일을 추출합니다. ZipArchive 클래스를 사용하여 *zip* 파일에 액세스하고 ZipArchiveEntry 클래스를 사용하여 개별 항목을 검사합니다. ZipArchiveEntry 개체의 ExtractToFile 확장 메서드는 System.IO.Compression.ZipFileExtensions 클래스에서 사용할 수 있습니다.

샘플을 실행하려면 *result.zip*이라는 *.zip* 파일을 프로그램 폴더에 둡니다. 메시지가 표시되면 추출할 폴더 이름을 입력합니다.

> **ⓘ 중요**
>
> 파일 압축을 풀 때는 압축을 풀 디렉터리에서 이스케이프할 수 있는 악성 파일 경로를 찾아야 합니다. 이를 경로 통과 공격이라고 합니다. 다음 예제는 악성 파일 경로를 확인하는 방법과 안전하게 압축을 푸는 방법을 보여줍니다.

C#

```
using System;
using System.IO;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string zipPath = @".\result.zip";

        Console.WriteLine("Provide path where to extract the zip file:");
        string extractPath = Console.ReadLine();

        // Normalizes the path.
        extractPath = Path.GetFullPath(extractPath);

        // Ensures that the last character on the extraction path
        // is the directory separator char.
        // Without this, a malicious zip file could try to traverse outside
of the expected
        // extraction path.
```

```
        if (!extractPath.EndsWith(Path.DirectorySeparatorChar.ToString(),
StringComparison.Ordinal))
            extractPath += Path.DirectorySeparatorChar;

        using (ZipArchive archive = ZipFile.OpenRead(zipPath))
        {
            foreach (ZipArchiveEntry entry in archive.Entries)
            {
                if (entry.FullName.EndsWith(".txt",
StringComparison.OrdinalIgnoreCase))
                {
                    // Gets the full path to ensure that relative segments
are removed.
                    string destinationPath =
Path.GetFullPath(Path.Combine(extractPath, entry.FullName));

                    // Ordinal match is safest, case-sensitive volumes can
be mounted within volumes that
                    // are case-insensitive.
                    if (destinationPath.StartsWith(extractPath,
StringComparison.Ordinal))
                        entry.ExtractToFile(destinationPath);
                }
            }
        }
    }
}
```

예제 3: 기존 .zip 파일에 파일 추가

다음 예제는 ZipArchive 클래스를 사용하여 기존 *.zip* 파일에 액세스하고 파일을 추가합니다. 새 파일은 기존 *.zip* 파일에 추가할 때 압축됩니다.

C#

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            using (FileStream zipToOpen = new
FileStream(@"c:\users\exampleuser\release.zip", FileMode.Open))
            {
                using (ZipArchive archive = new ZipArchive(zipToOpen,
ZipArchiveMode.Update))
                {
```

```
                    ZipArchiveEntry readmeEntry =
archive.CreateEntry("Readme.txt");
                    using (StreamWriter writer = new
StreamWriter(readmeEntry.Open()))
                    {
                            writer.WriteLine("Information about this
package.");
                            writer.WriteLine("========================");
                    }
                }
            }
        }
    }
}
```

예제 4: .gz 파일 압축 및 압축 풀기

GZipStream 및 DeflateStream 클래스를 사용하여 데이터를 압축하거나 압축을 풀 수도 있습니다. 동일한 압축 알고리즘을 사용합니다. *.gz* 파일에 기록된 GZipStream 개체의 압축은 많은 일반적인 도구를 사용하여 풀 수 있습니다. 다음 예제에서는 GZipStream 클래스를 사용하여 파일의 디렉터리를 압축하고 압축을 푸는 방법을 보여 줍니다.

C#

```
using System;
using System.IO;
using System.IO.Compression;

public class Program
{
    private static string directoryPath = @".\temp";
    public static void Main()
    {
        DirectoryInfo directorySelected = new DirectoryInfo(directoryPath);
        Compress(directorySelected);

        foreach (FileInfo fileToDecompress in
directorySelected.GetFiles("*.gz"))
        {
            Decompress(fileToDecompress);
        }
    }

    public static void Compress(DirectoryInfo directorySelected)
    {
        foreach (FileInfo fileToCompress in directorySelected.GetFiles())
        {
            using (FileStream originalFileStream =
fileToCompress.OpenRead())
            {
```

```
                if ((File.GetAttributes(fileToCompress.FullName) &
                   FileAttributes.Hidden) != FileAttributes.Hidden &
fileToCompress.Extension != ".gz")
                {
                    using (FileStream compressedFileStream =
File.Create(fileToCompress.FullName + ".gz"))
                    {
                        using (GZipStream compressionStream = new
GZipStream(compressedFileStream,
                           CompressionMode.Compress))
                        {
                            originalFileStream.CopyTo(compressionStream);
                        }
                    }
                    FileInfo info = new FileInfo(directoryPath +
Path.DirectorySeparatorChar + fileToCompress.Name + ".gz");
                    Console.WriteLine($"Compressed {fileToCompress.Name}
from {fileToCompress.Length.ToString()} to {info.Length.ToString()}
bytes.");
                }
            }
        }
    }

    public static void Decompress(FileInfo fileToDecompress)
    {
        using (FileStream originalFileStream = fileToDecompress.OpenRead())
        {
            string currentFileName = fileToDecompress.FullName;
            string newFileName =
currentFileName.Remove(currentFileName.Length -
fileToDecompress.Extension.Length);

            using (FileStream decompressedFileStream =
File.Create(newFileName))
            {
                using (GZipStream decompressionStream = new
GZipStream(originalFileStream, CompressionMode.Decompress))
                {
                    decompressionStream.CopyTo(decompressedFileStream);
                    Console.WriteLine($"Decompressed:
{fileToDecompress.Name}");
                }
            }
        }
    }
}
```

참조

- ZipArchive
- ZipFile

- ZipArchiveEntry
- DeflateStream
- GZipStream
- 파일 및 스트림 I/O

스트림 작성

백업 저장소(*backing store*)는 디스크 또는 메모리와 같은 스토리지 매체입니다. 각 유형의 백업 저장소는 자체 스트림을 Stream 클래스의 구현으로 구현합니다.

각 스트림 유형은 지정된 백업 저장소에 유입 또는 유출되는 바이트를 읽고 씁니다. 백업 저장소에 연결되는 스트림을 기본 스트림(*base stream*)이라고 합니다. 기본 스트림에는 스트림을 백업 저장소에 연결하는 데 필요한 매개 변수가 포함된 생성자가 있습니다. 예를 들어 FileStream에는 파일을 읽을지, 쓸지, 아니면 둘 다 할지 결정하는 액세스 모드 매개 변수를 지정하는 생성자가 있습니다.

System.IO 클래스 디자인은 간소화된 스트림 컴퍼지션을 제공합니다. 원하는 기능을 제공하는 하나 이상의 통과 스트림에 기본 스트림을 연결할 수 있습니다. 체인의 끝에 reader 또는 writer를 연결하여 선호되는 형식을 쉽게 읽거나 쓸 수 있습니다.

필수 조건

이러한 예제는 *data.txt*라는 일반 텍스트 파일을 사용합니다. 이 파일에는 일부 텍스트가 포함되어야 합니다.

예: 스트림 데이터 암호화 및 암호 해독

다음 예제에서는 파일에서 데이터를 읽고 암호화한 다음 암호화된 데이터를 다른 파일에 씁니다. 스트림 구성은 기본 변환 암호화를 사용하여 데이터를 변환하는 데 사용됩니다. 스트림을 통과하는 각 바이트의 값은 **80**만큼 변경됩니다.

> ⚠ **Warning**
>
> 이 예제에서 사용되는 암호화는 기본적이고 안전하지 않습니다. 실제로 데이터를 암호화하여 사용하기 위한 것이 아니라, 스트림 구성을 통해 데이터를 변경하는 방법을 보여주기 위해 제공됩니다.

암호화를 위한 원본 데이터 읽기

다음 코드는 한 파일에서 텍스트를 읽고 변환한 다음 다른 파일에 씁니다.

> 💡 **팁**

이 코드를 검토하기 전에 `CipherStream`은(는) 사용자 정의 형식임을 알아두세요. 이 클래스의 코드는 **CipherStream 클래스** 섹션에서 제공됩니다.

C#

```
void WriteShiftedFile()
{
    // Create the base streams for the input and output files
    using FileStream inputBaseStream = File.OpenRead("data.txt");
    using CipherStream encryptStream = CipherStream.CreateForRead(inputBaseStream);
    using FileStream outputBaseStream = File.Open("shifted.txt", FileMode.Create,
FileAccess.Write);

    int intValue;

    // Read byte from inputBaseStream through the encryptStream (normal bytes into
shifted bytes)
    while ((intValue = encryptStream.ReadByte()) != -1)
    {
        outputBaseStream.WriteByte((byte)intValue);
    }

    // Process is:
    //  (inputBaseStream -> encryptStream) -> outputBaseStream
}
```

이전 코드에 대해 다음 측면을 고려하세요.

- 두 개의 FileStream 개체가 있습니다.
  - 첫 번째 `FileStream`(`inputBaseStream` 변수) 개체는 *data.txt* 파일의 내용을 읽습니다. 이것은 **입력** 데이터 스트림입니다.
  - 두 번째 `FileStream`(`outputBaseStream` 변수) 개체는 수신 데이터를 *shifted.txt* 파일에 씁니다. **출력** 데이터 스트림입니다.
- `CipherStream`(`encryptStream` 변수) 개체는 `inputBaseStream`을(를) 래핑하여 `inputBaseStream`을(를) `encryptStream`에 대한 기본 스트림으로 만듭니다.

입력 스트림을 직접 읽고 출력 스트림에 데이터를 쓸 수 있지만 데이터를 변환하지는 않습니다. 대신 `encryptStream` 입력 스트림 래퍼를 사용하여 데이터를 읽습니다. `encryptStream`에서 데이터를 읽으면 `inputBaseStream` 기본 스트림에서 가져와서 변환하고 반환합니다. 반환된 데이터는 `outputBaseStream`에 쓰여지고 이를 통해 *shifted.txt* 파일에 데이터가 쓰여집니다.

암호 해독을 위해 변환된 데이터 읽기

이 코드는 이전 코드에서 수행한 암호화를 되돌립니다.

C#

```
void ReadShiftedFile()
{
    int intValue;

    // Create the base streams for the input and output files
    using FileStream inputBaseStream = File.OpenRead("shifted.txt");
    using FileStream outputBaseStream = File.Open("unshifted.txt", FileMode.Create,
FileAccess.Write);
    using CipherStream unencryptStream =
CipherStream.CreateForWrite(outputBaseStream);

    // Read byte from inputBaseStream through the encryptStream (shifted bytes into
normal bytes)
    while ((intValue = inputBaseStream.ReadByte()) != -1)
    {
        unencryptStream.WriteByte((byte)intValue);
    }

    // Process is:
    //  inputBaseStream -> (encryptStream -> outputBaseStream)
}
```

이전 코드에 대해 다음 측면을 고려하세요.

- 두 개의 FileStream 개체가 있습니다.
  - 첫 번째 `FileStream`(변수 `inputBaseStream`) 개체는 *shifted.txt* 파일의 내용을 읽습니다. 이것은 **입력** 데이터 스트림입니다.
  - 두 번째 `FileStream` (`outputBaseStream` 변수) 개체는 들어오는 데이터를 *unshifted.txt* 파일에 씁니다. **출력** 데이터 스트림입니다.
- `CipherStream`(`unencryptStream` 변수) 개체는 `outputBaseStream`을(를) 래핑하여 `outputBaseStream`을(를) `unencryptStream`에 대한 기본 스트림으로 만듭니다.

여기서 코드는 이전 예제와 약간 다릅니다. 입력 스트림을 래핑하는 대신 `unencryptStream`에서 출력 스트림을 래핑합니다. `inputBaseStream` 입력 스트림에서 데이터를 읽으면 `unencryptStream` 출력 스트림 래퍼로 전송됩니다. `unencryptStream`에서 데이터를 받으면 데이터를 변환한 다음 `outputBaseStream` 기본 스트림에 씁니다. `outputBaseStream` 출력 스트림은 데이터를 *unshifted.txt* 파일에 씁니다.

변환된 데이터 유효성 검사

이전의 두 예제에서는 데이터에 대해 두 가지 작업을 수행했습니다. 첫째로 *data.txt* 파일의 내용이 암호화되어 *shifted.txt* 파일에 저장되었습니다. 둘째로 *shifted.txt* 파일의 암호화된 내용이 암호 해독되어 *unshifted.txt* 파일에 저장되었습니다. 따라서 *data.txt* 파일과 *unshifted.txt* 파일은 같아야 합니다. 다음 코드는 그 파일들이 동일한지 비교합니다.

C#

```
bool IsShiftedFileValid()
{
    // Read the shifted file
    string originalText = File.ReadAllText("data.txt");

    // Read the shifted file
    string shiftedText = File.ReadAllText("unshifted.txt");

    // Check if the decrypted file is valid
    return shiftedText == originalText;
}
```

다음 코드는 이 전체 암호화-암호 해독 프로세스를 실행합니다.

C#

```
// Read the contents of data.txt, encrypt it, and write it to shifted.txt
WriteShiftedFile();

// Read the contents of shifted.txt, decrypt it, and write it to unshifted.txt
ReadShiftedFile();

// Check if the decrypted file is valid
Console.WriteLine(IsShiftedFileValid()
                    ? "Decrypted file is valid"     // True
                    : "Decrypted file is invalid"   // False
                );

// Output:
//   Decrypted file is valid
```

CipherStream 클래스

다음 코드 조각은 기본 변환 암호화를 사용하여 바이트를 암호화하고 암호를 해독하는 `CipherStream` 클래스를 제공합니다. 이 클래스는 Stream에서 상속되며 데이터 읽기 또는 쓰기를 지원합니다.

> ⚠ **Warning**
>
> 이 예제에서 사용되는 암호화는 기본적이고 안전하지 않습니다. 실제로 데이터를 암호화하여 사용하기 위한 것이 아니라, 스트림 구성을 통해 데이터를 변경하는 방법을 보여주기 위해 제공됩니다.

C#

```
using System.IO;

public class CipherStream : Stream
{
    // WARNING: This is a simple encoding algorithm and should not be used in
production code

    const byte ENCODING_OFFSET = 80;

    private bool _readable;
    private bool _writable;

    private Stream _wrappedBaseStream;

    public override bool CanRead => _readable;
    public override bool CanSeek => false;
    public override bool CanWrite => _writable;
    public override long Length => _wrappedBaseStream.Length;
    public override long Position
    {
        get => _wrappedBaseStream.Position;
        set => _wrappedBaseStream.Position = value;
    }

    public static CipherStream CreateForRead(Stream baseStream)
    {
        return new CipherStream(baseStream)
        {
            _readable = true,
            _writable = false
        };
    }

    public static CipherStream CreateForWrite(Stream baseStream)
    {
        return new CipherStream(baseStream)
        {
            _readable = false,
            _writable = true
        };
    }

    private CipherStream(Stream baseStream) =>
        _wrappedBaseStream = baseStream;

    public override int Read(byte[] buffer, int offset, int count)
    {
        if (!_readable) throw new NotSupportedException();
        if (count == 0) return 0;

        int returnCounter = 0;

        for (int i = 0; i < count; i++)
        {
```

```
            int value = _wrappedBaseStream.ReadByte();

            if (value == -1)
                return returnCounter;

            value += ENCODING_OFFSET;
            if (value > byte.MaxValue)
                value -= byte.MaxValue;

            buffer[i + offset] = Convert.ToByte(value);
            returnCounter++;
        }

        return returnCounter;
    }

    public override void Write(byte[] buffer, int offset, int count)
    {
        if (!_writable) throw new NotSupportedException();

        byte[] newBuffer = new byte[count];
        buffer.CopyTo(newBuffer, offset);

        for (int i = 0; i < count; i++)
        {
            int value = newBuffer[i];

            value -= ENCODING_OFFSET;

            if (value < 0)
                value = byte.MaxValue - value;

            newBuffer[i] = Convert.ToByte(value);
        }

        _wrappedBaseStream.Write(newBuffer, 0, count);
    }

    public override void Flush() => _wrappedBaseStream.Flush();
    public override long Seek(long offset, SeekOrigin origin) => throw new
NotSupportedException();
    public override void SetLength(long value) => throw new
NotSupportedException();
}
```

참고 항목

- StreamReader
- StreamReader.ReadLine
- StreamReader.Peek
- FileStream

- BinaryReader
- BinaryReader.ReadByte
- BinaryReader.PeekChar

Last updated on 2026. 02. 24.

방법: .NET Framework와 Windows 런타임 스트림 간 변환(Windows만 해당)

UWP 앱용 .NET Framework는 전체 .NET Framework의 하위 집합입니다. UWP 앱에 대한 보안 및 기타 요구 사항 때문에 전체 .NET Framework API 집합을 사용하여 파일을 열고 읽을 수 없습니다. 자세한 내용은 UWP 앱용 .NET 개요를 참조하세요. 그러나 다른 스트림 조작 작업에 .NET Framework API를 사용할 수 있습니다. 이러한 스트림을 조작하려면 .NET Framework 스트림 형식(예: MemoryStream 또는 FileStream)과 Windows 런타임 스트림(예: IInputStream, IOutputStream또는 IRandomAccessStream)을 변환할 수 있습니다.

클래스에는 System.IO.WindowsRuntimeStreamExtensions 이러한 변환을 쉽게 만드는 메서드가 포함되어 있습니다. 그러나 .NET Framework와 Windows 런타임 스트림 간의 기본 차이점은 다음 섹션에 설명된 대로 이러한 메서드를 사용한 결과에 영향을 줍니다.

Windows 런타임에서 .NET Framework 스트림으로 변환

Windows 런타임 스트림에서 .NET Framework 스트림으로 변환하려면 다음 System.IO.WindowsRuntimeStreamExtensions 방법 중 하나를 사용합니다.

- WindowsRuntimeStreamExtensions.AsStream 는 Windows 런타임의 임의 액세스 스트림을 UWP 앱용 .NET의 관리되는 스트림으로 변환합니다.
- WindowsRuntimeStreamExtensions.AsStreamForWrite 는 Windows 런타임의 출력 스트림을 UWP 앱용 .NET의 관리되는 스트림으로 변환합니다.
- WindowsRuntimeStreamExtensions.AsStreamForRead 는 Windows 런타임의 입력 스트림을 UWP 앱용 .NET의 관리되는 스트림으로 변환합니다.

Windows 런타임은 읽기 전용, 쓰기 전용 또는 읽기 및 쓰기를 지원하는 스트림 형식을 제공합니다. 이러한 기능은 Windows 런타임 스트림을 .NET Framework 스트림으로 변환할 때 유지 관리됩니다. 또한 Windows 런타임 스트림을 .NET Framework 스트림으로 변환하고 다시 변환하는 경우 원래 Windows 런타임 인스턴스를 다시 가져옵니다.

변환하려는 Windows 런타임 스트림의 기능과 일치하는 변환 방법을 사용하는 것이 가장 좋습니다. 그러나 IRandomAccessStream 읽기 가능하고 쓰기 가능하므로(둘 다 IOutputStream 구현하고 IInputStream) 변환 메서드는 원래 스트림의 기능을 유지합니다. 예를 들어, WindowsRuntimeStreamExtensions.AsStreamForRead를 사용하여 IRandomAccessStream을 변환할 때, 변환된 .NET Framework 스트림이 단지 읽을 수 있는 것으로 제한되지 않습니다. 또한 쓰기가 가능합니다.

예: Windows 런타임 임의 액세스를 .NET Framework 스트림으로 변환

Windows 런타임 임의 액세스 스트림에서 .NET Framework 스트림으로 변환하려면 이 메서드를 WindowsRuntimeStreamExtensions.AsStream 사용합니다.

다음 코드 예제에서는 파일을 선택하고 Windows 런타임 API를 사용하여 파일을 연 다음 .NET Framework 스트림으로 변환하라는 메시지를 표시합니다. 스트림을 읽고 텍스트 블록에 출력합니다. 일반적으로 결과를 출력하기 전에 .NET Framework API를 사용하여 스트림을 조작합니다.

C#

```
// Create a file picker.
FileOpenPicker picker = new FileOpenPicker();
picker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
picker.ViewMode = PickerViewMode.List;
picker.FileTypeFilter.Add(".txt");

// Show picker, enabling user to pick one file.
StorageFile result = await picker.PickSingleFileAsync();
if (result != null)
{
    try
    {
        // Retrieve the stream. This method returns a
IRandomAccessStreamWithContentType.
        var stream = await result.OpenReadAsync();

        // Convert the stream to a .NET stream using AsStream, pass to a
        // StreamReader and read the stream.
        using (StreamReader sr = new StreamReader(stream.AsStream()))
        {
            TextBlock1.Text = sr.ReadToEnd();
        }
    }
    catch (Exception ex)
    {
        // ...
    }
}
```

.NET Framework에서 Windows 런타임 스트림으로 변환

.NET Framework 스트림에서 Windows 런타임 스트림으로 변환하려면 다음 System.IO.WindowsRuntimeStreamExtensions 방법 중 하나를 사용합니다.

- WindowsRuntimeStreamExtensions.AsInputStream 는 UWP 앱용 .NET의 관리되는 스트림을 Windows 런타임의 입력 스트림으로 변환합니다.
- WindowsRuntimeStreamExtensions.AsOutputStream 는 UWP 앱용 .NET의 관리되는 스트림을 Windows 런타임의 출력 스트림으로 변환합니다.
- WindowsRuntimeStreamExtensions.AsRandomAccessStream 는 UWP 앱용 .NET의 관리되는 스트림을 Windows 런타임이 읽기 또는 쓰기에 사용할 수 있는 임의 액세스 스트림으로 변환합니다.

.NET Framework 스트림을 Windows 런타임 스트림으로 변환하는 경우 변환된 스트림의 기능은 원래 스트림에 따라 달라집니다. 예를 들어 원래 스트림이 읽기 및 쓰기를 모두 지원하고 스트림을 변환하기 위해 호출 WindowsRuntimeStreamExtensions.AsInputStream 하는 경우 반환되는 형식은 `IRandomAccessStream`입니다. `IRandomAccessStream` 는 `IInputStream` 읽기 및 `IOutputStream`쓰기를 구현하고 지원합니다.

.NET Framework 스트림은 변환 후에도 복제를 지원하지 않습니다. .NET Framework 스트림을 Windows 런타임 스트림으로 변환한 후 GetInputStreamAt 또는 GetOutputStreamAt을 호출하고, CloneStream를 호출하거나 CloneStream를 직접 호출하면 예외가 발생합니다.

예: .NET Framework를 Windows 런타임 임의 액세스 스트림으로 변환

.NET Framework 스트림에서 Windows 런타임 임의 액세스 스트림으로 변환하려면 다음 예제와 같이 메서드를 사용합니다 AsRandomAccessStream .

> **ⓘ Important**
>
> 사용 중인 .NET Framework 스트림이 검색을 지원하는지 확인하거나 해당 스트림에 복사합니다. 속성 Stream.CanSeek을 사용하여 이를 확인할 수 있습니다.

C#

```
// Create an HttpClient and access an image as a stream.
var client = new HttpClient();
Stream stream = await client.GetStreamAsync("https://learn.microsoft.com/en-
us/dotnet/images/hub/featured-1.png");
// Create a .NET memory stream.
var memStream = new MemoryStream();
// Convert the stream to the memory stream, because a memory stream supports
seeking.
await stream.CopyToAsync(memStream);
// Set the start position.
memStream.Position = 0;
```

```
// Create a new bitmap image.
var bitmap = new BitmapImage();
// Set the bitmap source to the stream, which is converted to a
IRandomAccessStream.
bitmap.SetSource(memStream.AsRandomAccessStream());
// Set the image control source to the bitmap.
Image1.Source = bitmap;
```

참고하십시오

- 빠른 시작: 파일 읽기 및 쓰기(Windows)
- Windows 스토어 앱용 .NET 개요
- .NET 용 Windows 스토어 앱 API

Last updated on 2026. 03. 26.

비동기 파일 입출력

2025. 06. 17.

비동기 작업을 사용하면 주 스레드를 차단하지 않고 리소스 집약적 I/O 작업을 수행할 수 있습니다. 이 성능 고려 사항은 시간이 많이 걸리는 스트림 작업이 UI 스레드를 차단하고 앱이 작동하지 않는 것처럼 표시되도록 할 수 있는 Windows 8.x 스토어 앱 또는 데스크톱 앱에서 특히 중요합니다.

.NET Framework 4.5부터 I/O 형식에는 비동기 작업을 간소화하는 비동기 메서드가 포함됩니다. 비동기 메서드는 `Async` 이름(예: ReadAsync, WriteAsync, CopyToAsyncFlushAsync, ReadLineAsync및 ReadToEndAsync)을 포함합니다. 이러한 비동기 메서드는 스트림 클래스인 Stream, FileStream, MemoryStream 및 스트림에서 읽거나 스트림에 쓰는 데 사용되는 클래스인 TextReader, TextWriter에 구현됩니다.

.NET Framework 4 및 이전 버전에서는 비동기 I/O 작업과 BeginRead 같은 EndRead 메서드를 사용하고 구현해야 합니다. 이러한 메서드는 레거시 코드를 지원하기 위해 현재 .NET 버전에서 계속 사용할 수 있습니다. 그러나 비동기 메서드를 사용하면 비동기 I/O 작업을 보다 쉽게 구현할 수 있습니다.

C# 및 Visual Basic에는 각각 비동기 프로그래밍을 위한 두 개의 키워드가 있습니다.

- `Async` (Visual Basic) 또는 `async` (C#) 한정자는 비동기 작업이 포함된 메서드를 표시하는 데 사용됩니다.
- `Await` 비동기 메서드의 결과에 적용되는 (Visual Basic) 또는 `await` (C#) 연산자입니다.

비동기 I/O 작업을 구현하려면 다음 예제와 같이 비동기 메서드와 함께 이러한 키워드를 사용합니다. 자세한 내용은 비동기 및 await를 사용한 비동기 프로그래밍(C#) 또는 Async 및 Await를 사용한 비동기 프로그래밍(Visual Basic)을 참조하세요.

다음 예제에서는 두 FileStream 개체를 사용하여 한 디렉터리에서 다른 디렉터리로 파일을 비동기적으로 복사하는 방법을 보여 줍니다. Click 이벤트 처리기는 Button 컨트롤의 `async` 한정자로 표시된 이유는 비동기 메서드를 호출하기 때문입니다.

C#

```
using System;
using System.Threading.Tasks;
using System.Windows;
using System.IO;

namespace WpfApplication
{
    public partial class MainWindow : Window
    {
```

```
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Button_Click(object sender, RoutedEventArgs e)
        {
            string startDirectory = @"c:\Users\exampleuser\start";
            string endDirectory = @"c:\Users\exampleuser\end";

            foreach (string filename in Directory.EnumerateFiles(startDirectory))
            {
                using (FileStream sourceStream = File.Open(filename,
FileMode.Open))
                {
                    using (FileStream destinationStream =
File.Create(Path.Combine(endDirectory, Path.GetFileName(filename))))
                    {
                        await sourceStream.CopyToAsync(destinationStream);
                    }
                }
            }
        }
    }
}
```

다음 예제는 이전 예제와 비슷하지만 StreamReader와 StreamWriter 개체를 사용하여 텍스트 파일의 내용을 비동기적으로 읽고 씁니다.

C#

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    string UserDirectory = @"c:\Users\exampleuser\";

    using (StreamReader SourceReader = File.OpenText(UserDirectory +
"BigFile.txt"))
    {
        using (StreamWriter DestinationWriter = File.CreateText(UserDirectory +
"CopiedFile.txt"))
        {
            await CopyFilesAsync(SourceReader, DestinationWriter);
        }
    }
}

public async Task CopyFilesAsync(StreamReader Source, StreamWriter Destination)
{
    char[] buffer = new char[0x1000];
    int numRead;
    while ((numRead = await Source.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        await Destination.WriteAsync(buffer, 0, numRead);
```

```
    }
}
```

다음 예제에서는 Windows 8.x 스토어 앱에서 Stream로 파일을 열고 StreamReader 클래스의 인스턴스를 사용하여 콘텐츠를 읽는 데 사용되는 코드 숨김 파일과 XAML 파일을 보여 줍니다. 비동기 메서드를 사용하여 파일을 스트림으로 열고 해당 내용을 읽습니다.

C#

```
using System;
using System.IO;
using System.Text;
using Windows.Storage.Pickers;
using Windows.Storage;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace ExampleApplication
{
    public sealed partial class BlankPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();
        }

        private async void Button_Click_1(object sender, RoutedEventArgs e)
        {
            StringBuilder contents = new StringBuilder();
            string nextLine;
            int lineCounter = 1;

            var openPicker = new FileOpenPicker();
            openPicker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
            openPicker.FileTypeFilter.Add(".txt");
            StorageFile selectedFile = await openPicker.PickSingleFileAsync();

            using (StreamReader reader = new StreamReader(await
selectedFile.OpenStreamForReadAsync()))
            {
                while ((nextLine = await reader.ReadLineAsync()) != null)
                {
                    contents.AppendFormat("{0}. ", lineCounter);
                    contents.Append(nextLine);
                    contents.AppendLine();
                    lineCounter++;
                    if (lineCounter > 3)
                    {
                        contents.AppendLine("Only first 3 lines shown.");
                        break;
                    }
                }
            }
```

```
            DisplayContentsBlock.Text = contents.ToString();
        }
    }
}
```

XAML

```
<Page
    x:Class="ExampleApplication.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ExampleApplication"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel Background="{StaticResource ApplicationPageBackgroundBrush}"
VerticalAlignment="Center" HorizontalAlignment="Center">
        <TextBlock Text="Display lines from a file."></TextBlock>
        <Button Content="Load File" Click="Button_Click_1"></Button>
        <TextBlock Name="DisplayContentsBlock"></TextBlock>
    </StackPanel>
</Page>
```

참고하십시오

- Stream
- 파일 및 스트림 I/O
- async 및 await과 함께하는 비동기 프로그래밍 (C#)
- Async 및 Await를 사용한 비동기 프로그래밍(Visual Basic)

.NET에서 I/O 오류 처리

2025. 06. 17.

메서드 호출에서 발생할 수 있는 예외(시스템에 과부하가 걸릴 때 발생하는 OutOfMemoryException 또는 프로그래머 오류로 인한 NullReferenceException 등) 외에도, .NET 파일 시스템 메서드는 다음과 같은 예외를 던질 수 있습니다.

- System.IO.IOException는 모든 System.IO 예외 형식의 기본 클래스입니다. 운영 체제로부터 반환된 코드가 다른 예외 유형에 직접 매핑되지 않는 오류에 대해 throw됩니다.
- System.IO.FileNotFoundException;
- System.IO.DirectoryNotFoundException;
- DriveNotFoundException;
- System.IO.PathTooLongException;
- System.OperationCanceledException;
- System.UnauthorizedAccessException;
- System.ArgumentException는 .NET Framework 및 .NET Core 2.0 및 이전 버전에서 잘못된 경로 문자 때문에 throw되는 예외입니다.
- System.NotSupportedException는 .NET Framework에서 잘못된 콜론에 대해 throw됩니다.
- System.Security.SecurityException.NET Framework에 필요한 권한만 없는 제한된 신뢰로 실행되는 애플리케이션에 대해 throw됩니다. (완전 신뢰는 .NET Framework의 기본값입니다.)

오류 코드를 예외에 매핑

파일 시스템은 운영 체제 리소스이므로 .NET Core 및 .NET Framework의 I/O 메서드는 기본 운영 체제에 대한 호출을 래핑합니다. 운영 체제에서 실행하는 코드에서 I/O 오류가 발생하면 운영 체제는 .NET I/O 메서드에 오류 정보를 반환합니다. 그런 다음, 메서드는 오류 정보(일반적으로 오류 코드 형식)를 .NET 예외 형식으로 변환합니다. 대부분의 경우 오류 코드를 해당 예외 형식으로 직접 변환하여 이 작업을 수행합니다. 메서드 호출의 컨텍스트에 따라 오류의 특별한 매핑을 수행하지 않습니다.

예를 들어 Windows 운영 체제에서 오류 코드 `ERROR_FILE_NOT_FOUND` (또는 0x02)를 반환하는 메서드 호출은 FileNotFoundException에 매핑되고, 오류 코드 `ERROR_PATH_NOT_FOUND` (또는 0x03)는 DirectoryNotFoundException에 매핑됩니다.

그러나 운영 체제에서 특정 오류 코드를 반환하는 정확한 조건은 문서화되지 않았거나 제대로 문서화되지 않은 경우가 많습니다. 따라서 예기치 않은 예외가 발생할 수 있습니다. 예를 들어, 파일이 아닌 디렉터리로 작업하는 경우 DirectoryInfo 생성자에 잘못된 디렉터리 경로를 제공하면 DirectoryNotFoundException 예외가 발생합니다. 그러나 FileNotFoundException을(를) 던질 수도 있습니다.

I/O 작업의 예외 처리

운영 체제에 대한 이러한 의존으로 인해 동일한 예외 조건(예: 예제에서 디렉터리를 찾을 수 없음 오류)으로 인해 I/O 메서드가 I/O 예외의 전체 클래스 중 하나를 throw할 수 있습니다. 즉, I/O API를 호출할 때 다음 표와 같이 코드에서 이러한 예외의 대부분 또는 전부를 처리하도록 준비해야 합니다.

테이블 확장

예외 유형	.NET Core/.NET 5 이상	.NET Framework
IOException	예	예
FileNotFoundException	예	예
DirectoryNotFoundException	예	예
DriveNotFoundException	예	예
PathTooLongException	예	예
OperationCanceledException	예	예
UnauthorizedAccessException	예	예
ArgumentException	.NET Core 2.0 이하	예
NotSupportedException	아니오	예
SecurityException	아니오	제한된 신뢰만

IOException 처리

System.IO 네임스페이스에서 예외의 기본 클래스로서, 미리 정의된 예외 형식에 매핑되지 않는 오류 코드에 대해서도 IOException가 발생됩니다. 즉, 모든 I/O 작업에서 발생할 수 있습니다.

> ⓘ **중요**
>
> IOException 네임스페이스에 있는 System.IO 다른 예외 형식의 기본 클래스이므로 다른 I/O 관련 예외를 `catch` 처리한 후 블록에서 처리해야 합니다.

또한 .NET Core 2.1부터 유효성 검사에서 경로 정확성(예: 경로에 잘못된 문자가 없는지 확인)이 제거되었으며 런타임은 자체 유효성 검사 코드가 아닌 운영 체제 오류 코드에서 매핑된 예외를 throw합니다. 이 경우 throw될 가능성이 가장 큰 예외는 IOException이며, 다른 예외 형식도 throw될 수 있습니다.

예외 처리 코드에서는 항상 마지막을 IOException 처리해야 합니다. 그렇지 않으면 다른 모든 IO 예외의 기본 클래스이므로 파생 클래스의 catch 블록은 평가되지 않습니다.

이 IOException경우 IOException.HResult 속성에서 추가 오류 정보를 가져올 수 있습니다. HResult 값을 Win32 오류 코드로 변환하려면 32비트 값의 상위 16비트를 제거합니다. 다음 표에서는 IOException에 래핑될 수 있는 오류 코드를 나열합니다.

테이블 확장

HResult	변하지 않는 것	설명
공유 위반 오류	32	파일 이름이 없거나 파일 또는 디렉터리가 사용 중입니다.
오류_파일_존재함	80	파일이 이미 있습니다.
ERROR_INVALID_PARAMETER (잘못된 매개 변수)	87	메서드에 제공된 인수가 잘못되었습니다.
오류_이미_존재함	183	파일 또는 디렉터리가 이미 있습니다.

catch 문에서 `When` 절을 사용하여 다음 예제와 같이 항목을 처리할 수 있습니다.

C#

```
using System;
using System.IO;
using System.Text;

class Program
{
    static void Main()
    {
        var sw = OpenStream(@".\textfile.txt");
        if (sw is null)
            return;
        sw.WriteLine("This is the first line.");
        sw.WriteLine("This is the second line.");
        sw.Close();
    }

    static StreamWriter? OpenStream(string path)
    {
        if (path is null)
        {
            Console.WriteLine("You did not supply a file path.");
            return null;
        }

        try
```

```
        {
            var fs = new FileStream(path, FileMode.CreateNew);
            return new StreamWriter(fs);
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("The file or directory cannot be found.");
        }
        catch (DirectoryNotFoundException)
        {
            Console.WriteLine("The file or directory cannot be found.");
        }
        catch (DriveNotFoundException)
        {
            Console.WriteLine("The drive specified in 'path' is invalid.");
        }
        catch (PathTooLongException)
        {
            Console.WriteLine("'path' exceeds the maximum supported path
length.");
        }
        catch (UnauthorizedAccessException)
        {
            Console.WriteLine("You do not have permission to create this file.");
        }
        catch (IOException e) when ((e.HResult & 0x0000FFFF) == 32)
        {
            Console.WriteLine("There is a sharing violation.");
        }
        catch (IOException e) when ((e.HResult & 0x0000FFFF) == 80)
        {
            Console.WriteLine("The file already exists.");
        }
        catch (IOException e)
        {
            Console.WriteLine($"An exception occurred:\nError code: " +
                              $"{e.HResult & 0x0000FFFF}\nMessage: {e.Message}");
        }
        return null;
    }
}
```

참고하십시오

- .NET의 예외 처리 및 Throw
- 예외 처리(작업 병렬 라이브러리)
- 예외에 대한 모범 사례
- catch 블록에서 특정 예외를 사용하는 방법

격리된 스토리지

아티클 • 2023. 04. 08.

데스크톱 앱에서 격리된 스토리지는 코드와 저장된 데이터를 연결하는 표준화된 방법을 정의하여 격리와 안전을 제공하는 데이터 스토리지 메커니즘입니다. 표준화를 통해 다음과 같은 여러 가지 이점도 활용할 수 있습니다. 관리자는 파일 스토리지 구성, 보안 정책 설정, 사용하지 않은 데이터 삭제를 위해 격리된 스토리지를 조작하는 도구를 사용할 수 있습니다. 격리된 스토리지를 사용하면 더 이상 파일 시스템에서 안전한 위치를 지정하기 위해 코드에 고유 경로를 포함할 필요가 없으며 격리된 스토리지에만 액세스할 수 있는 다른 애플리케이션으로부터 데이터가 보호됩니다. 애플리케이션의 스토리지 영역 위치를 나타내는 하드 코드된 정보는 필요하지 않습니다.

> ⓘ **중요**
>
> Windows 8.x 스토어 앱에는 격리된 스토리지를 사용할 수 없습니다. 대신에 Windows Runtime API에 포함된 `Windows.Storage` 네임스페이스의 애플리케이션 데이터 클래스를 사용하여 로컬 데이터 및 파일을 저장합니다. 자세한 내용은 Windows 개발자 센터에서 **애플리케이션 데이터** 를 참조하세요.

데이터 구획 및 저장소

애플리케이션이 데이터를 파일에 저장할 때 스토리지 위치가 다른 애플리케이션에 알려져 손상될 가능성을 최소화할 수 있도록 파일 이름과 스토리지 위치를 신중하게 선택해야 합니다. 이러한 문제를 관리할 적절한 표준 시스템 없이 스토리지 충돌을 최소화하는 기법을 즉흥적으로 만드는 것은 복잡할 수 있으며 결과를 신뢰할 수도 없습니다.

격리된 스토리지를 사용하면 데이터는 항상 사용자와 어셈블리별로 격리됩니다. 어셈블리의 원본 또는 강력한 이름과 같은 자격 증명은 어셈블리 ID를 결정합니다. 또한 유사한 자격 증명을 사용하여 데이터가 애플리케이션 도메인별로 격리될 수도 있습니다.

격리된 스토리지를 사용하는 경우, 애플리케이션은 게시자 또는 서명 등과 같은 코드 ID의 몇 가지 측면과 관련된 고유 데이터 구획에 데이터를 저장합니다. 데이터 컴파트먼트는 특정 스토리지 위치가 아니라 추상적인 개념이며 스토리지라고 하는 하나 이상의 격리된 스토리지 파일로 구성됩니다. 이 스토리지는 데이터가 저장되는 실제 디렉터리 위치를 포함합니다. 예를 들어, 애플리케이션은 관련된 데이터 구획을 가질 수 있고 파일 시스템의 디렉터리는 이 애플리케이션의 데이터를 실제로 유지하는 저장소를 구현할 수 있습니다. 저장소에 저장된 데이터는 사용자 기본 설정 정보에서 애플리케이션 상태에 이르기까지 모든 종류의 데이터가 될 수 있습니다. 개발자의 경우, 데이터 구획의 위치는 투명합니다. 저장소는 보통 클라이언트에 있지만, 서버 애플리케이션은 사용자를 대신하여 관련

기능을 수행하면서 그 사용자를 가장하여 정보를 저장하는 격리된 저장소를 사용할 수 있습니다. 또한 격리된 스토리지는 로밍 사용자와 함께 정보가 이동되도록 사용자의 로밍 프로필과 함께 서버의 정보를 저장할 수 있습니다.

격리된 스토리지의 할당량

할당 한도는 사용 가능한 격리된 스토리지 크기의 제한 값입니다. 디렉터리 및 저장소의 다른 정보와 관련된 오버헤드는 물론 파일 공간(바이트)을 포함합니다. 격리된 스토리지는 IsolatedStoragePermission 개체를 사용하여 설정된 스토리지의 제한에 해당하는 사용 권한 할당을 사용합니다. 할당량을 초과하는 데이터를 쓰려고 하면 IsolatedStorageException 예외가 throw됩니다. .NET Framework 구성 도구(Mscorcfg.msc)를 사용하여 수정할 수 있는 보안 정책에 따라 코드에 부여될 권한이 결정됩니다. IsolatedStoragePermission 권한이 부여된 코드는 UserQuota 속성이 허용하는 수준의 스토리지만 사용하도록 제한됩니다. 그러나 코드에서 다른 사용자 ID를 제공하여 사용 권한 할당 한도를 무시할 수 있으므로 사용 권한 할당 한도는 코드 동작에 대해 고정된 제한이 아니라 코드 동작 방식에 대한 지침으로 사용됩니다.

로밍 저장소에는 할당 한도가 적용되지 않기 때문에 코드에 약간 높은 수준의 사용 권한이 있어야 이를 사용할 수 있습니다. 열거형 값 AssemblyIsolationByRoamingUser 및 DomainIsolationByRoamingUser는 로밍 사용자를 위한 격리된 스토리지를 사용하여 권한을 지정합니다.

액세스 보안

격리된 스토리지를 사용하면 부분적으로 신뢰할 수 있는 애플리케이션은 컴퓨터의 보안 정책에 의해 제어되는 방식으로 데이터를 저장할 수 있습니다. 이 방법은 특히 사용자가 주의하여 실행해야 하는 다운로드된 구성 요소에 유용합니다. 표준 I/O 메커니즘을 사용하여 파일 시스템에 액세스하는 경우, 사용 권한을 이 유형의 코드에 부여하는 경우는 거의 없습니다. 그러나 격리된 스토리지를 사용할 권한은 로컬 컴퓨터, 로컬 네트워크 또는 인터넷에서 실행되는 코드에 기본적으로 부여됩니다.

관리자는 해당 신뢰 수준에 따라 애플리케이션 또는 사용자가 가질 수 있는 격리된 스토리지 양을 제한할 수 있습니다. 또한 사용자의 지속된 데이터를 모두 제거할 수도 있습니다. 격리된 스토리지를 만들거나 액세스하려면 코드에 적절한 IsolatedStorageFilePermission 권한이 부여되어야 합니다.

격리된 스토리지에 액세스하려면 필요한 네이티브 플랫폼 운영 체제 권한이 모두 코드에 있어야 합니다. 파일 시스템을 사용할 수 있는 권한을 가진 사용자를 제어하는 ACL(액세스 제어 목록)이 충족되어야 합니다. .NET 애플리케이션은 특정 플랫폼 관련 가장을 수행하는 경우를 제외하고는 격리된 스토리지에 액세스할 수 있는 운영 체제 권한을 이미 가지고 있습니다. 이런 경우 애플리케이션은 가장된 사용자 ID가 격리된 스토리지에 액세스

할 수 있는 적절한 운영 체제 권한을 가지고 있는지 확인해야 합니다. 이 액세스 권한은 웹에서 실행되거나 다운로드된 코드에 특정 사용자와 관련된 스토리지 영역에서 읽고 쓸 수 있는 편리한 방법을 제공합니다.

격리된 스토리지에 대한 액세스를 제어하기 위해 공용 언어 런타임은 IsolatedStorageFilePermission 개체를 사용합니다. 각 개체에는 다음과 같은 값을 지정하는 속성이 있습니다.

- 허용 수준 - 허용된 액세스 형식을 나타내며 값은 IsolatedStorageContainment 열거형의 멤버입니다. 이러한 값에 대한 자세한 내용은 다음 섹션의 표를 참조하세요.
- 이전 섹션에서 설명한 스토리지 할당량입니다.

런타임은 코드에서 처음으로 저장소를 열려고 할 때 이 IsolatedStorageFilePermission 권한을 요청합니다. 코드의 신뢰 정도에 따라 이 권한을 부여할지 여부를 결정합니다. 이 사용 권한이 부여되면 보안 정책 및 코드의 IsolatedStorageFilePermission 요청에 의해 허용되는 사용법과 스토리지 할당량 값이 결정됩니다. 보안 정책은 .NET Framework 구성 도구(Mscorcfg.msc)를 사용하여 설정됩니다. 호출 스택의 모든 호출자를 검사하여 각 호출자가 적절한 최소 허용 수준 값을 가지고 있는지 확인합니다. 또한 런타임은 파일을 저장할 저장소를 열거나 만든 코드에 부과된 할당 한도도 검사합니다. 이러한 조건을 충족하면 사용 권한이 부여됩니다. 할당 한도는 파일을 저장소에 쓸 때마다 다시 검사됩니다.

공용 언어 런타임은 보안 정책을 기반으로 적절한 모든 IsolatedStorageFilePermission 을 부여하므로 권한을 요청하는 데 애플리케이션 코드가 필요하지 않습니다. 그러나 IsolatedStorageFilePermission을 포함하여 애플리케이션에서 필요로 하는 특정 사용 권한을 요청해야 하는 경우가 있습니다.

허용 수준과 보안 위험

IsolatedStorageFilePermission에 의해 지정되는 허용 사용법에 따라 코드에서 격리된 스토리지를 만들고 사용할 수 있는 정도가 결정됩니다. 다음 표에서는 사용 권한에 지정된 허용 수준이 어떤 방식으로 격리 유형에 부합하는지를 보여 주고 각 허용 수준과 관련된 보안 위험을 요약하여 설명합니다.

허용 수준	격리 유형	보안 효과
None	격리된 스토리지를 사용할 수 없습니다.	보안 효과가 없습니다.

허용 수준	격리 유형	보안 효과
DomainIsolationByUser	사용자, 도메인 및 어셈블리별 격리. 각 어셈블리는 도메인 내에 별도의 하위 저장소를 가지고 있습니다. 이 권한을 사용하는 저장소는 암시적으로 컴퓨터와 별로도 격리됩니다.	이 권한을 사용하면 비록 할당 한도가 적용되어 어느 정도까지는 허가되지 않은 수준의 리소스 남용을 방지하지만 그래도 이러한 리소스 남용이 발생할 수 있습니다. 이를 서비스 거부 공격이라고 합니다.
DomainIsolationByRoamingUser	`DomainIsolationByUser`와 동일하지만, 로밍 사용자 프로필을 사용할 수 있고 할당량이 적용되지 않은 경우 로밍되는 위치에 저장소가 저장됩니다.	할당 한도를 사용할 수 없으므로 스토리지 리소스가 서비스 거부 공격에 노출되기 쉽습니다.
AssemblyIsolationByUser	사용자 및 어셈블리별 격리. 이 권한을 사용하는 저장소는 암시적으로 컴퓨터와 별로도 격리됩니다.	서비스 거부 공격 문제를 방지하기 위해 이 수준에서 할당 한도가 적용됩니다. 다른 도메인의 동일한 어셈블리가 이 저장소에 액세스할 수 있으므로 애플리케이션 간에 정보가 누출될 가능성이 있습니다.
AssemblyIsolationByRoamingUser	`AssemblyIsolationByUser`와 동일하지만, 로밍 사용자 프로필을 사용할 수 있고 할당량이 적용되지 않은 경우 로밍되는 위치에 저장소가 저장됩니다.	`AssemblyIsolationByUser`의 경우와 동일하지만, 할당량이 없으므로 서비스 거부 공격 위험이 증가합니다.
AdministerIsolatedStorageByUser	사용자별 격리. 일반적으로 관리 또는 디버깅 도구에서 이 권한 수준을 사용합니다.	이 권한으로 액세스하면 코드가 어셈블리 격리와 관계없이 사용자의 격리된 스토리지 파일 또는 디렉터리를 보거나 삭제할 수 있습니다. 정보 누출 및 데이터 손실 등의 위험이 있지만 이에 제한되지는 않습니다.
UnrestrictedIsolatedStorage	모든 사용자, 도메인 및 어셈블리별 격리. 일반적으로 관리 또는 디버깅 도구에서 이 권한 수준을 사용합니다.	이 권한을 사용하면 모든 사용자에 대한 모든 격리된 저장소 전체가 손상될 가능성이 있습니다.

신뢰할 수 없는 데이터와 관련하여 격리된 스토리지 구성 요소의 안전

이 섹션은 다음과 같은 프레임워크에 적용됩니다.

- .NET Framework(모든 버전)
- .NET Core 2.1 이상
- .NET 5 이상

.NET Framework 및 .NET Core는 사용자, 애플리케이션 또는 구성 요소의 데이터를 유지하는 메커니즘으로 격리된 스토리지를 제공합니다. 기본적으로 이 레거시 구성 요소는 지금은 사용되지 않는 코드 액세스 보안 시나리오를 위해 설계되었습니다.

다양한 격리된 스토리지 API 및 도구를 사용하여 신뢰 경계 전반에서 데이터를 읽을 수 있습니다. 예를 들어 머신 전체 범위에서 데이터를 읽으면 머신에서 신뢰할 수 없는 다른 사용자 계정의 데이터를 집계할 수 있습니다. 머신 전체의 격리된 스토리지 범위에서 읽는 구성 요소 또는 애플리케이션은 이 데이터를 읽을 때의 결과를 알고 있어야 합니다.

머신 전체 범위에서 읽을 수 있는 보안 관련 API

다음 API를 호출하는 구성 요소 또는 애플리케이션은 머신 전체 범위에서 읽습니다.

- IsolatedStorageFile.GetEnumerator - IsolatedStorageScope.Machine 플래그를 포함하는 범위를 전달합니다.
- IsolatedStorageFile.GetMachineStoreForApplication
- IsolatedStorageFile.GetMachineStoreForAssembly
- IsolatedStorageFile.GetMachineStoreForDomain
- IsolatedStorageFile.GetStore - IsolatedStorageScope.Machine 플래그를 포함하는 범위를 전달합니다.
- IsolatedStorageFile.Remove - `IsolatedStorageScope.Machine` 플래그를 포함하는 범위를 전달합니다.

격리된 스토리지 도구`storeadm.exe`는 다음 코드와 같이 스위치를 사용하여 `/machine` 호출되는 경우 영향을 줍니다.

```txt
storeadm.exe /machine [any-other-switches]
```

격리된 스토리지 도구는 Visual Studio 및 .NET Framework SDK의 일부로 제공됩니다.

애플리케이션에서 위의 API를 호출하지 않거나 워크플로에서 이러한 방식으로 `storeadm.exe`를 호출하지 않는 경우에는 이 문서의 내용이 적용되지 않습니다.

다중 사용자 환경에 미치는 영향

앞에서 언급했듯이 이러한 API가 보안에 미치는 영향은 한 신뢰 환경에서 기록된 데이터를 다른 신뢰 환경에서 읽기 때문에 발생합니다. 격리된 스토리지는 일반적으로 다음 세 위치 중 하나를 사용하여 데이터를 읽고 씁니다.

1. `%LOCALAPPDATA%\IsolatedStorage\`: 예를 들어 `User` 범위의 경우 `C:\Users\<username>\AppData\Local\IsolatedStorage\`입니다.
2. `%APPDATA%\IsolatedStorage\`: 예를 들어 `User|Roaming` 범위의 경우 `C:\Users\<username>\AppData\Roaming\IsolatedStorage\`입니다.
3. `%PROGRAMDATA%\IsolatedStorage\`: 예를 들어 `Machine` 범위의 경우 `C:\ProgramData\IsolatedStorage\`입니다.

처음 두 위치는 사용자별로 격리됩니다. Windows에서는 같은 머신의 여러 사용자 계정이 서로의 사용자 프로필 폴더에 액세스할 수 없도록 합니다. `User` 또는 `User|Roaming` 저장소를 사용하는 서로 다른 두 사용자 계정은 서로의 데이터를 볼 수 없고 조작할 수 없습니다.

세 번째 위치는 머신의 모든 사용자 계정 간에 공유됩니다. 다른 계정이 이 위치에서 읽고 위치에 쓸 수 있으며 서로의 데이터를 볼 수 있습니다.

앞의 경로는 사용 중인 Windows 버전에 따라 다를 수 있습니다.

이제 두 명의 등록된 사용자 *Mallory* 및 *Bob*이 있는 다중 사용자 시스템을 살펴봅니다. Mallory는 자신의 사용자 프로필 디렉터리 `C:\Users\Mallory\`에 액세스할 수 있고 공유 머신 전체 스토리지 위치 `C:\ProgramData\IsolatedStorage\`에 액세스할 수 있습니다. Bob의 사용자 프로필 디렉터리 `C:\Users\Bob\`에는 액세스할 수 없습니다.

Mallory가 Bob을 공격하려는 경우 머신 전체 스토리지 위치에 데이터를 쓴 다음 Bob이 머신 전체 저장소에서 읽도록 영향을 주려고 할 수 있습니다. Bob이 이 저장소에서 읽는 앱을 실행하면 해당 앱은 Mallory가 여기에 저장한 데이터를 사용하지만 Bob의 사용자 계정 컨텍스트 내에서 작동합니다. 이 문서의 나머지 부분에서는 다양한 공격 벡터와 이러한 공격의 위험을 최소화하기 위해 앱에서 수행할 수 있는 단계를 고려합니다.

> ⓘ **참고**
>
> 이러한 공격을 수행하기 위해 Mallory는 다음이 필요합니다.
>
> - 머신의 사용자 계정
> - 파일 시스템의 알려진 위치에 파일을 저장할 수 있는 기능
> - Bob이 어느 시점에 이 데이터를 읽으려고 시도하는 앱을 실행할 것이라는 정보
>
> 이러한 위협 벡터는 가정용 PC 또는 직원이 한 명인 기업 워크스테이션과 같은 표준 단일 사용자 데스크톱 환경에는 적용되지 않습니다.

권한 상승

권한 상승 공격은 Bob의 앱이 Mallory의 파일을 읽고 자동으로 해당 페이로드의 콘텐츠를 기반으로 작업을 수행하려고 할 때 발생합니다. 머신 전체 저장소에서 시작 스크립트의 콘텐츠를 읽고 해당 콘텐츠를 `Process.Start`에 전달하는 앱이 있다고 가정해 보세요. Mallory가 머신 전체 저장소 내에 악성 스크립트를 저장할 수 있는 경우 Bob이 앱을 시작하면 다음이 발생합니다.

- 앱이 'Bob의 사용자 프로필 컨텍스트에서' Mallory의 악성 스크립트를 구문 분석하고 시작합니다.
- Mallory가 로컬 머신에서 Bob의 계정에 액세스합니다.

서비스 거부

서비스 거부 공격은 Bob의 앱이 Mallory의 파일을 읽고 크래시되거나 제대로 작동하지 않을 때 발생합니다. 앞에서 언급한 앱이 머신 전체 저장소에서 시작 스크립트를 구문 분석하려고 시도한다고 가정해 보세요. Mallory가 머신 전체 저장소 내에서 잘 구성되지 않은 콘텐츠가 포함된 파일을 저장할 수 있는 경우 다음을 수행할 수 있습니다.

- Bob의 앱이 시작 경로의 초기에 예외를 throw하도록 합니다.
- 앱이 예외 때문에 제대로 시작되지 않게 합니다.

그런 다음 자신의 사용자 계정에서 Bob이 앱을 시작할 수 있는 기능을 거부했습니다.

정보 공개

정보 공개 공격은 Mallory가 Bob을 속여 정상적으로는 액세스할 수 없는 파일의 콘텐츠를 공개하도록 할 수 있는 경우 발생합니다. Bob의 비밀 파일 *C:\Users\Bob\secret.txt*를 Mallory가 읽고 싶어한다고 가정해 보겠습니다. Mallory는 파일의 경로는 알지만 Windows에서 Bob의 사용자 프로필 디렉터리에 액세스하지 못하게 하므로 파일을 읽을 수 없습니다.

대신, Mallory는 하드 링크를 머신 전체 저장소에 배치합니다. 이 특수한 종류의 파일은 자체에 콘텐츠를 포함하지는 않고 디스크의 다른 파일을 가리킵니다. 하드 링크 파일을 읽으려고 하면 대신 링크의 대상으로 지정된 파일의 콘텐츠를 읽습니다. 하드 링크를 만든 후에도 Mallory는 링크의 대상(`C:\Users\Bob\secret.txt`)에 액세스할 수 없으므로 파일 콘텐츠를 읽을 수 없습니다. 그러나 Bob은 이 파일에 액세스할 수 '있습니다'.

Bob의 앱이 머신 전체 저장소에서 읽을 때 이제 `secret.txt` 파일 자체가 머신 전체 저장소에 있었던 것처럼 파일 콘텐츠를 실수로 읽습니다. Bob의 앱이 종료되면 파일을 컴퓨터 전체 저장소에 다시 저장하려고 하면 파일의 실제 복사본이

C:\ProgramData\IsolatedStorage 디렉터리에 배치됩니다. 이 디렉터리는 머신의 모든 사용자가 읽을 수 있으므로 이제 Mallory는 파일의 콘텐츠를 읽을 수 있습니다.

이러한 공격으로부터 방어하기 위한 모범 사례

중요: 환경에 상호 신뢰할 수 없는 사용자가 여러 명 있는 경우 API `IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.Machine)`를 호출하거나 `storeadm.exe /machine /list` 도구를 호출하지 **마세요**. 둘은 모두 신뢰할 수 있는 데이터에서 작동하고 있다고 가정합니다. 공격자가 머신 전체 저장소에 악성 페이로드를 시드할 수 있는 경우 해당 페이로드로 인해 이러한 명령을 실행하는 사용자의 컨텍스트에서 권한 상승 공격이 발생할 수 있습니다.

다중 사용자 환경에서 운영하는 경우 머신 범위를 대상으로 하는 격리된 스토리지 기능을 사용하는 것을 다시 고려하세요. 앱이 머신 전체 위치에서 데이터를 읽어야 하는 경우 관리자 계정만 쓸 수 있는 위치에서 데이터를 읽는 것이 좋습니다. `%PROGRAMFILES%` 디렉터리 및 `HKLM` 레지스트리 하이브는 관리자만 쓸 수 있고 모든 사용자가 읽을 수 있는 위치의 예입니다. 따라서 해당 위치에서 읽은 데이터는 신뢰할 수 있다고 간주합니다.

앱이 다중 사용자 환경에서 '머신' 범위를 사용해야 하는 경우에는 머신 전체 저장소에서 읽은 모든 파일의 콘텐츠를 유효성 검사하세요. 앱이 이러한 파일에서 개체 그래프를 역직렬화하는 경우 `BinaryFormatter` 또는 `NetDataContractSerializer`와 같이 위험한 직렬 변환기 대신 `XmlSerializer`와 같은 더 안전한 직렬 변환기를 사용하세요. 파일 콘텐츠에 따라 리소스 할당을 수행하는 개체 그래프나 많이 중첩된 개체 그래프는 주의해서 사용하세요.

격리된 스토리지 위치

때때로 운영 체제의 파일 시스템을 사용하여 격리된 스토리지에 대한 변경 내용을 확인하면 도움이 됩니다. 또한 개발자는 격리된 스토리지 파일의 위치를 알고 싶을 때가 있습니다. 이 위치는 운영 체제에 따라 다릅니다. 다음 표에서는 일반적으로 사용되는 몇 가지 운영 체제에서 격리된 스토리지가 만들어지는 루트 위치를 보여 줍니다. 이 루트 위치 아래에 있는 Microsoft\IsolatedStorage 디렉터리를 찾으세요. 파일 시스템에서 격리된 스토리지를 보려면 숨김 파일과 폴더를 표시하도록 폴더 설정을 변경해야 합니다.

운영 체제	파일 시스템에서의 위치

운영 체제	파일 시스템에서의 위치
Windows 2000, Windows XP, Windows Server 2003(Windows NT 4.0에서 업그레이드)	로밍 가능 저장소 = <SYSTEMROOT>\Profiles\ <user>\Application Data 비로밍 저장소 = <SYSTEMROOT>\Profiles\<user>\Local Settings\Application Data
Windows 2000 - 클린 설치(및 Windows 98 및 Windows NT 3.51에서 업그레이드)	로밍 가능 저장소 = <SYSTEMDRIVE>\Documents and Settings\ <user>\Application Data 비로밍 저장소 = <SYSTEMDRIVE>\Documents and Settings\ <user>\Local Settings\Application Data
Windows XP, Windows Server 2003 - 새로 설치 및 Windows 2000, Windows 98에서 업그레이드	로밍 가능 저장소 = <SYSTEMDRIVE>\Documents and Settings\ <user>\Application Data 비로밍 저장소 = <SYSTEMDRIVE>\Documents and Settings\ <user>\Local Settings\Application Data
Windows 8, Windows 7, Windows Server 2008, Windows Vista	로밍 가능 저장소 = <SYSTEMDRIVE>\Users\ <user>\AppData\Roaming 비로밍 저장소 = <SYSTEMDRIVE>\Users\ <user>\AppData\Local

격리된 스토리지 만들기, 열거 및 삭제

.NET에서는 System.IO.IsolatedStorage 네임스페이스의 세 가지 클래스를 제공해 격리된 스토리지와 관련된 작업을 수행할 수 있도록 해줍니다.

- IsolatedStorageFile에서 파생되는 System.IO.IsolatedStorage.IsolatedStorage 은 저장된 어셈블리 및 애플리케이션 파일의 기본 관리를 제공합니다. IsolatedStorageFile 클래스 인스턴스는 파일 시스템에 있는 단일 저장소를 나타냅니다.

- IsolatedStorageFileStream 에서 파생되는 System.IO.FileStream 은 저장소에 있는 파일에 대한 액세스를 제공합니다.

- IsolatedStorageScope 은 적절한 격리 유형을 사용하여 저장소를 만들고 선택할 수 있도록 하는 열거형입니다.

격리된 스토리지 클래스를 사용하여 격리된 스토리지를 만들고 열거하고 삭제할 수 있습니다. 이러한 작업을 수행하는 데 필요한 메서드는 IsolatedStorageFile 개체를 통해 사용할 수 있습니다. 일부 작업을 수행하려면 격리된 스토리지를 관리할 수 있는 권한을 나타내는 IsolatedStorageFilePermission 권한을 가져야 하며 파일이나 디렉터리에 액세스할 수 있는 운영 체제 권한도 가지고 있어야 합니다.

일반적인 격리된 스토리지 작업을 보여 주는 일련의 예제는 관련 항목에 나열되어 있는 방법 항목을 참조하세요.

격리된 스토리지 시나리오

격리된 스토리지는 다음 네 가지 시나리오를 비롯하여 다양한 상황에서 유용합니다.

- 다운로드된 컨트롤. 인터넷에서 다운로드된 관리 코드 컨트롤은 일반 I/O 클래스를 통해 하드 드라이브에 쓸 수 없지만 격리된 스토리지를 사용하여 사용자 설정 및 애플리케이션 상태를 유지할 수 있습니다.

- 공유 구성 요소 스토리지. 애플리케이션 간에 공유되는 구성 요소는 격리된 스토리지를 사용하여 데이터 스토리지에 대한 제어된 액세스를 제공할 수 있습니다.

- 서버 스토리지. 서버 애플리케이션은 격리된 스토리지를 사용하여 애플리케이션을 요청하는 다수의 사용자에게 개별 스토리지를 제공할 수 있습니다. 격리된 스토리지는 항상 사용자별로 분리되어 있으므로 서버는 요청하는 사용자를 가장해야 합니다. 이런 경우 데이터는 사용자를 구분하기 위해 애플리케이션에서 사용하는 동일한 ID인 보안 주체 ID를 기반으로 격리됩니다.

- 로밍. 또한 애플리케이션에서는 격리된 스토리지를 사용하여 로밍 사용자 프로필을 저장할 수 있습니다. 따라서 사용자의 격리된 저장소는 프로필을 로밍하는 데 사용됩니다.

다음과 같은 경우 격리된 스토리지를 사용하지 마세요.

- 격리된 스토리지는 충분히 신뢰할 수 있는 코드, 비관리 코드 또는 컴퓨터에서 신뢰할 수 있는 사용자로부터 보호되지 않으므로 암호화되지 않은 키 또는 암호 등의 상위 값 비밀을 저장하는 데 사용하지 마세요.
- 코드를 저장하는 데 사용하지 마세요.
- 관리자가 컨트롤하는 구성 및 배포 설정을 저장하는 데 사용하지 마세요. 사용자 기본 설정은 관리자가 제어하지 않으므로 구성 설정으로 간주되지 않습니다.

대부분의 애플리케이션은 데이터베이스를 사용하여 데이터를 저장하고 격리합니다. 이때 데이터베이스에 있는 하나 이상의 행은 특정 사용자에 대한 스토리지를 나타낼 수 있습니다. 사용자 수가 적은 경우, 데이터베이스 사용에 따른 오버헤드가 의미가 있는 경우 또는 데이터베이스 기능이 없는 경우 데이터베이스 대신 격리된 스토리지를 사용하도록 선택할 수 있습니다. 또한 데이터베이스에서 제공하는 행보다 더 융통성 있고 복잡한 스토리지가 애플리케이션에 필요한 경우에도 격리된 스토리지를 사용할 수 있습니다.

관련 문서

제목	설명
격리 유형	다양한 유형의 격리에 대해 설명합니다.
방법: 격리된 스토리지의 저장소 가져오기	IsolatedStorageFile 클래스를 사용하여 사용자 및 어셈블리별로 격리된 저장소를 가져오는 예제를 제공합니다.
방법: 격리된 스토리지의 저장소 열거	IsolatedStorageFile.GetEnumerator 메서드를 사용하여 사용자에 대한 모든 격리된 스토리지 크기를 계산하는 방법을 보여 줍니다.
방법: 격리된 스토리지에서 저장소 삭제	IsolatedStorageFile.Remove 메서드를 두 가지 방법으로 사용하여 격리된 저장소를 삭제하는 방법을 보여 줍니다.
방법: 격리된 스토리지의 공간 부족 상태 예상	격리된 저장소에서 남은 공간을 측정하는 방법을 보여 줍니다.
방법: 격리된 스토리지의 파일 및 디렉터리 만들기	격리된 저장소에서 파일 및 디렉터리를 만드는 몇 가지 예제를 제공합니다.
방법: 격리된 스토리지의 기존 파일 및 디렉터리 찾기	격리된 스토리지에서 디렉터리 구조 및 파일을 읽는 방법을 보여 줍니다.
방법: 격리된 스토리지의 파일 읽기 및 쓰기	격리된 스토리지 파일에 문자열을 쓰고 다시 문자열을 읽는 예제를 제공합니다.
방법: 격리된 스토리지의 파일 및 디렉터리 삭제	격리된 스토리지 파일 및 디렉터리를 삭제하는 방법을 보여 줍니다.

제목	설명
파일 및 스트림 I/O	동기 및 비동기 파일과 데이터 스트림 액세스를 수행할 수 있는 방법에 대해 설명합니다.

참고

- System.IO.IsolatedStorage.IsolatedStorage
- System.IO.IsolatedStorage.IsolatedStorageFile
- System.IO.IsolatedStorage.IsolatedStorageFileStream
- System.IO.IsolatedStorage.IsolatedStorageScope

격리 유형

아티클 • 2025. 04. 30.

격리된 스토리지에 대한 액세스는 항상 스토리지를 만든 사용자로 제한됩니다. 이러한 유형의 격리를 구현하기 위해 공용 언어 런타임은 운영 체제에서 인식하는 것과 동일한 사용자 ID 개념을 사용합니다. 이는 저장소를 열 때 코드가 실행되는 프로세스와 연결된 ID입니다. 이 ID는 인증된 사용자 ID이지만 가장하면 현재 사용자의 ID가 동적으로 변경될 수 있습니다.

격리된 스토리지에 대한 액세스는 애플리케이션의 도메인 및 어셈블리와 연결된 ID 또는 어셈블리에만 따라 제한됩니다. 런타임은 다음과 같은 방법으로 이러한 ID를 가져옵니다.

- 도메인 ID는 애플리케이션의 증거를 나타내며, 웹 애플리케이션의 경우 전체 URL일 수 있습니다. 셸 호스팅 코드의 경우 도메인 ID는 애플리케이션 디렉터리 경로를 기반으로 할 수 있습니다. 예를 들어 실행 파일이 C:\Office\MyApp.exe경로에서 실행되는 경우 도메인 ID는 C:\Office\MyApp.exe.

- 어셈블리 ID는 어셈블리의 증거입니다. 이는 어셈블리의 강력한 이름, 어셈블리의 소프트웨어 게시자 또는 해당 URL ID일 수 있는 암호화 디지털 서명에서 비롯될 수 있습니다. 어셈블리에 강력한 이름과 소프트웨어 게시자 ID가 모두 있는 경우 소프트웨어 게시자 ID가 사용됩니다. 어셈블리가 인터넷에서 제공되고 서명되지 않은 경우 URL ID가 사용됩니다. 어셈블리 및 강력한 이름에 대한 자세한 내용은 어셈블리를 사용한 프로그래밍을 참조하세요.

- 로밍 저장소는 로밍 사용자 프로필이 있는 사용자와 함께 이동합니다. 파일은 네트워크 디렉터리에 기록되며 사용자가 로그인하는 모든 컴퓨터에 다운로드됩니다. 로밍 사용자 프로필에 대한 자세한 내용은 다음을 참조하세요 IsolatedStorageScope.Roaming.

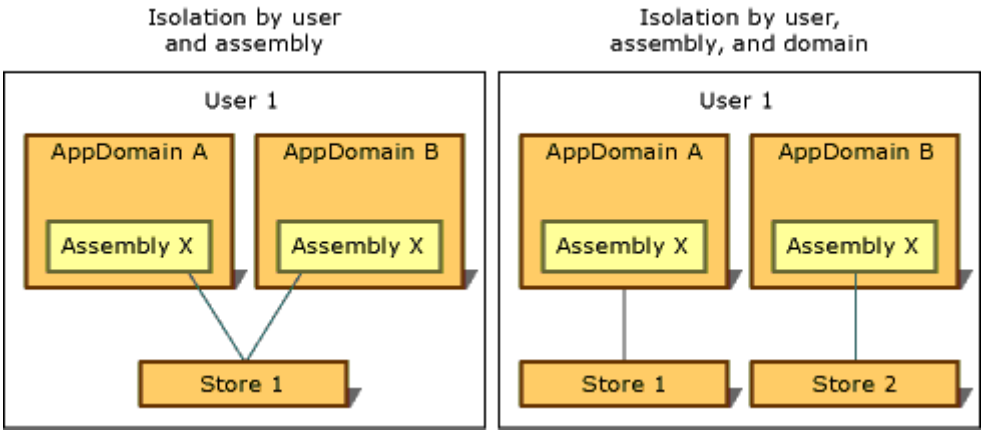
격리된 스토리지는 사용자, 도메인 및 어셈블리 ID의 개념을 결합하여 다음과 같은 방식으로 데이터를 격리할 수 있으며, 각각에는 고유한 사용 시나리오가 있습니다.

- 사용자 및 어셈블리별 격리

- 사용자, 도메인 및 어셈블리별 격리

이러한 격리 중 하나를 로밍 사용자 프로필과 결합할 수 있습니다. 자세한 내용은 격리된 스토리지 및 로밍 섹션을 참조하세요.

다음 그림에서는 여러 범위에서 저장소를 격리하는 방법을 보여 줍니다.



로밍 저장소를 제외하고 격리된 스토리지는 지정된 컴퓨터에 로컬인 스토리지 기능을 사용하기 때문에 항상 컴퓨터에 의해 암시적으로 격리됩니다.

> ⓘ **중요**
>
> Windows 8.x 스토어 앱에는 격리된 스토리지를 사용할 수 없습니다. 대신 Windows 런타임 API에 포함된 네임스페이 `Windows.Storage` 스의 애플리케이션 데이터 클래스를 사용하여 로컬 데이터 및 파일을 저장합니다. 자세한 내용은 Windows 개발자 센터의 **애플리케이션 데이터를** 참조하세요.

사용자 및 어셈블리별 격리

데이터 저장소를 사용하는 어셈블리가 애플리케이션의 도메인에서 액세스할 수 있어야 하는 경우 사용자 및 어셈블리에 의한 격리가 적절합니다. 일반적으로 이 경우 격리된 스토리지는 여러 애플리케이션에 적용되고 사용자의 이름 또는 라이선스 정보와 같은 특정 애플리케이션에 연결되지 않은 데이터를 저장하는 데 사용됩니다. 사용자 및 어셈블리에 의해 격리된 스토리지에 액세스하려면 애플리케이션 간에 정보를 전송하려면 코드를 신뢰할 수 있어야 합니다. 일반적으로 사용자 및 어셈블리에 의한 격리는 인터넷이 아닌 인트라넷에서 허용됩니다. 정적 IsolatedStorageFile.GetStore 메서드를 호출하고 사용자 및 어셈블리 IsolatedStorageScope 를 전달하면 이러한 종류의 격리를 사용하여 스토리지가 반환됩니다.

다음 코드 예제에서는 사용자 및 어셈블리에 의해 격리된 저장소를 검색합니다. 개체를 통해 `isoFile` 저장소에 액세스할 수 있습니다.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);
```

증거 매개 변수를 사용하는 예제는 다음을 참조하세요 GetStore(IsolatedStorageScope, Evidence, Type, Evidence, Type).

이 GetUserStoreForAssembly 메서드는 다음 코드 예제와 같이 바로 가기로 사용할 수 있습니다. 이 바로 가기는 로밍이 가능한 스토어를 여는 데 사용할 수 없습니다. 그러한 경우에는 GetStore를 사용하십시오.

C#

```
IsolatedStorageFile isoFile = IsolatedStorageFile.GetUserStoreForAssembly();
```

사용자, 도메인 및 어셈블리별 격리

애플리케이션에서 프라이빗 데이터 저장소가 필요한 타사 어셈블리를 사용하는 경우 격리된 스토리지를 사용하여 프라이빗 데이터를 저장할 수 있습니다. 사용자, 도메인 및 어셈블리에 의한 격리는 지정된 어셈블리의 코드만 데이터에 액세스할 수 있도록 하고, 어셈블리가 저장소를 만들 때 실행 중인 애플리케이션에서 어셈블리를 사용하는 경우에만, 그리고 저장소를 만든 사용자가 애플리케이션을 실행하는 경우에만 가능합니다. 사용자, 도메인 및 어셈블리에 의한 격리는 타사 어셈블리가 데이터를 다른 애플리케이션으로 유출하지 못하도록 합니다. 격리된 스토리지를 사용하려고 하지만 어떤 유형의 격리를 사용할지 확실하지 않은 경우 이 격리 유형이 기본 선택 항목이어야 합니다. GetStore의 정적 IsolatedStorageFile 메서드를 호출하고 사용자, 도메인 및 어셈블리 IsolatedStorageScope을(를) 전달하면 이러한 종류의 격리를 통해 스토리지가 반환됩니다.

다음 코드 예제에서는 사용자, 도메인 및 어셈블리에 의해 격리된 저장소를 검색합니다. 개체를 통해 `isoFile` 저장소에 액세스할 수 있습니다.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Domain |
        IsolatedStorageScope.Assembly, null, null);
```

다음 코드 예제와 같이 다른 메서드를 바로 가기로 사용할 수 있습니다. 이 바로 가기는 로밍이 가능한 스토어를 여는 데 사용할 수 없습니다. 그러한 경우에는 GetStore를 사용하십시오.

C#

```
IsolatedStorageFile isoFile = IsolatedStorageFile.GetUserStoreForDomain();
```

격리된 스토리지 및 로밍

로밍 사용자 프로필은 사용자가 네트워크에서 ID를 설정하고 해당 ID를 사용하여 모든 개인 설정으로 모든 네트워크 컴퓨터에 로그인할 수 있게 해주는 Windows 기능입니다. 격리된 스토리지를 사용하는 어셈블리는 사용자의 격리된 스토리지가 로밍 사용자 프로필과 함께 이동되도록 지정할 수 있습니다. 로밍은 사용자 및 어셈블리에 의한 격리 또는 사용자, 도메인 및 어셈블리별 격리와 함께 사용할 수 있습니다. 로밍 범위를 사용하지 않으면 로밍 사용자 프로필을 사용하는 경우에도 저장소가 로밍되지 않습니다.

다음 코드 예제에서는 사용자 및 어셈블리에 의해 격리된 로밍 저장소를 검색합니다. 개체를 통해 `isoFile` 저장소에 액세스할 수 있습니다.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly |
        IsolatedStorageScope.Roaming, null, null);
```

도메인 범위를 추가하여 사용자, 도메인 및 애플리케이션으로 격리된 로밍 저장소를 만들 수 있습니다. 다음 코드 예제에서는 이를 보여 줍니다.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain |
        IsolatedStorageScope.Roaming, null, null);
```

참고하십시오

- IsolatedStorageScope
- 격리된 스토리지

방법: 격리된 스토리지에 대한 저장소 가져오기

아티클 • 2025. 04. 30.

격리된 저장소는 데이터 구획 내에서 가상 파일 시스템을 노출합니다. 이 클래스는 IsolatedStorageFile 격리된 저장소와 상호 작용하기 위한 여러 메서드를 제공합니다. 저장소 IsolatedStorageFile 를 만들고 검색하려면 다음 세 가지 정적 메서드를 제공합니다.

- GetUserStoreForAssembly 는 사용자 및 어셈블리에 의해 격리된 스토리지를 반환합니다.
- GetUserStoreForDomain 는 도메인 및 어셈블리에 의해 격리된 스토리지를 반환합니다.

  두 메서드 모두 호출되는 코드에 속하는 저장소를 검색합니다.
- 정적 메서드 GetStore 는 범위 매개 변수 조합을 전달하여 지정된 격리된 저장소를 반환합니다.

다음 코드는 사용자, 어셈블리 및 도메인에 의해 격리된 저장소를 반환합니다.

C#

```
IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain, null, null);
```

로밍 사용자 프로필에 따라 저장소가 로밍하도록 이 GetStore 메서드를 사용할 수 있습니다. 이를 설정하는 방법에 대한 자세한 내용은 격리 유형을 참조하세요.

다른 어셈블리 내에서 가져온 격리된 저장소는 기본적으로 서로 다른 저장소입니다. 메서드의 매개 변수에서 어셈블리 또는 도메인 증명 정보를 전달하여 다른 어셈블리 또는 도메인의 GetStore 저장소에 액세스할 수 있습니다. 이렇게 하려면 애플리케이션 도메인 ID로 격리된 스토리지에 액세스할 수 있는 권한이 필요합니다. 자세한 내용은 GetStore 메서드 오버로드를 참조하세요.

GetUserStoreForAssembly, GetUserStoreForDomain, 및 GetStore 메서드는 IsolatedStorageFile 개체를 반환합니다. 상황에 가장 적합한 격리 유형을 결정하는 데 도움이 되도록 격리 유형을 참조하세요. 격리된 스토리지 파일 개체가 있는 경우 격리된 스토리지 메서드를 사용하여 파일 및 디렉터리를 읽고, 쓰고, 만들고, 삭제할 수 있습니다.

코드가 저장소 자체를 가져오기에 충분한 액세스 권한이 없는 코드에 개체를 전달하는 IsolatedStorageFile 것을 방지하는 메커니즘은 없습니다. 도메인 및 어셈블리 ID와 격리된 스토리지 권한은 IsolatedStorage 개체에 대한 참조를 얻을 때만 검사되며, 이는 일반적으로 GetUserStoreForAssembly, GetUserStoreForDomain 또는 GetStore 메서드에서 발생합니다. 따

라서 개체에 IsolatedStorageFile 대한 참조를 보호하는 것은 이러한 참조를 사용하는 코드의 책임입니다.

예시

다음 코드는 사용자 및 어셈블리에 의해 격리된 저장소를 가져오는 클래스의 간단한 예제를 제공합니다. 메서드가 전달하는 인수 IsolatedStorageScope.Domain 에 추가하여 GetStore 사용자, 도메인 및 어셈블리에 의해 격리된 저장소를 검색하도록 코드를 변경할 수 있습니다.

코드를 실행한 후에는 명령줄에 **StoreAdm /LIST** 를 입력하여 저장소가 생성되었는지 확인할 수 있습니다. 그러면 격리된 스토리지 도구(Storeadm.exe) 가 실행되고 사용자의 현재 격리된 저장소가 모두 나열됩니다.

C#

```
using System;
using System.IO.IsolatedStorage;

public class ObtainingAStore
{
    public static void Main()
    {
        // Get a new isolated store for this assembly and put it into an
        // isolated store object.

        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
    }
}
```

참고하십시오

- IsolatedStorageFile
- IsolatedStorageScope
- 격리된 스토리지
- 격리 유형
- .NET의 어셈블리

방법: 격리된 스토리지의 저장소 열거

아티클 • 2024. 03. 12.

IsolatedStorageFile.GetEnumerator 정적 메서드를 사용하여 현재 사용자의 모든 격리된 저장소를 열거할 수 있습니다. 이 메서드는 IsolatedStorageScope 값을 사용하여 IsolatedStorageFile 열거자를 반환합니다. 저장소를 열거하려면 IsolatedStorageFilePermission 값을 지정하는 AdministerIsolatedStorageByUser 권한이 있어야 합니다. GetEnumerator 값을 사용하여 User 메서드를 호출하면 현재 사용자에 대해 정의된 IsolatedStorageFile 개체의 배열을 반환합니다.

예시

다음 코드 예제에서는 사용자 및 어셈블리별로 격리된 저장소를 가져오고, 몇 개의 파일을 만든 다음, 이러한 파일을 GetEnumerator 메서드를 사용하여 검색합니다.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;
using System.Collections;

public class EnumeratingStores
{
    public static void Main()
    {
        using (IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null))
        {
            isoStore.CreateFile("TestFileA.Txt");
            isoStore.CreateFile("TestFileB.Txt");
            isoStore.CreateFile("TestFileC.Txt");
            isoStore.CreateFile("TestFileD.Txt");
        }

        IEnumerator allFiles =
IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User);
        long totalsize = 0;

        while (allFiles.MoveNext())
        {
            IsolatedStorageFile storeFile =
(IsolatedStorageFile)allFiles.Current;
            totalsize += (long)storeFile.UsedSize;
        }

        Console.WriteLine("The total size = " + totalsize);
```

```
    }
}
```

참고 항목

- IsolatedStorageFile
- 격리된 스토리지

방법: 격리된 스토리지의 저장소 삭제

2025. 06. 22.

이 클래스는 IsolatedStorageFile 격리된 스토리지 파일을 삭제하는 두 가지 메서드를 제공합니다.

- 인스턴스 메서드 Remove() 는 인수를 사용하지 않고 이를 호출하는 저장소를 삭제합니다. 이 작업에는 권한이 필요하지 않습니다. 저장소에 액세스할 수 있는 모든 코드는 저장소 내의 데이터를 모두 삭제할 수 있습니다.
- 정적 메서드 Remove(IsolatedStorageScope) 는 열거형 값을 사용하고 User 코드를 실행하는 사용자의 모든 저장소를 삭제합니다. 이 작업에는 IsolatedStorageFilePermission 값에 대한 AdministerIsolatedStorageByUser 권한이 필요합니다.

예시

다음 코드 예제에서는 정적 및 인스턴스 Remove 메서드의 사용을 보여 줍니다. 클래스는 두 개의 저장소를 가져옵니다. 하나는 사용자 및 어셈블리에 대해 격리되고 다른 하나는 사용자, 도메인 및 어셈블리에 대해 격리됩니다. 그러면 격리된 스토리지 파일Remove()의 메서드를 `isoStore1` 호출하여 사용자, 도메인 및 어셈블리 저장소가 삭제됩니다. 그런 다음, 정적 메서드 Remove(IsolatedStorageScope)를 호출하여 사용자의 나머지 모든 저장소가 삭제됩니다.

C#

```
using System;
using System.IO.IsolatedStorage;

public class DeletingStores
{
    public static void Main()
    {
        // Get a new isolated store for this user, domain, and assembly.
        // Put the store into an IsolatedStorageFile object.

        IsolatedStorageFile isoStore1 =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null,
null);
        Console.WriteLine("A store isolated by user, assembly, and domain has been
obtained.");

        // Get a new isolated store for user and assembly.
        // Put that store into a different IsolatedStorageFile object.

        IsolatedStorageFile isoStore2 =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
```

```
        Console.WriteLine("A store isolated by user and assembly has been
obtained.");

        // The Remove method deletes a specific store, in this case the
        // isoStore1 file.

        isoStore1.Remove();
        Console.WriteLine("The user, domain, and assembly isolated store has been
deleted.");

        // This static method deletes all the isolated stores for this user.

        IsolatedStorageFile.Remove(IsolatedStorageScope.User);
        Console.WriteLine("All isolated stores for this user have been deleted.");
    } // End of Main.
}
```

참고하십시오

- IsolatedStorageFile
- 격리된 스토리지

방법: 격리된 스토리지의 공간 부족 상태 예상

아티클 • 2025. 04. 30.

격리된 스토리지를 사용하는 코드는 격리된 스토리지 파일 및 디렉터리를 포함하는 데이터 구획의 최대 크기를 지정하는 할당량 으로 제한됩니다. 할당량은 보안 정책에 의해 정의되며 관리자가 구성할 수 있습니다. 데이터를 쓰려고 할 때 허용되는 최대 크기를 초과하면 예외가 IsolatedStorageException throw되고 작업이 실패합니다. 이렇게 하면 데이터 스토리지가 채워져 애플리케이션이 요청을 거부할 수 있는 악의적인 서비스 거부 공격을 방지할 수 있습니다.

해당 쓰기 시도가 이러한 이유로 실패할 가능성이 있는지를 판단하는 데 도움이 되도록, IsolatedStorage 클래스는 AvailableFreeSpace, UsedSize, Quota이라는 세 가지 읽기 전용 속성을 제공합니다. 이러한 속성을 사용하여 저장소에 쓰기로 인해 허용되는 최대 저장소 크기가 초과되는지 여부를 결정할 수 있습니다. 격리된 스토리지는 동시에 액세스할 수 있습니다. 따라서 남은 스토리지의 양을 계산할 때, 저장소에 쓰는 시점까지 스토리지 공간이 이미 사용될 수 있습니다. 그러나 저장소의 최대 크기를 사용하여 사용 가능한 스토리지의 상한에 도달할지 여부를 확인할 수 있습니다.

이 속성은 어셈블리에서 증거를 기반으로 하여 올바르게 작동합니다. 이러한 이유로, IsolatedStorageFile, GetUserStoreForAssembly, 또는 GetUserStoreForDomain 메서드를 사용하여 생성된 GetStore 객체에서만 이 속성을 검색해야 합니다. IsolatedStorageFile 다른 방법으로 만든 개체(예: 메서드에서 GetEnumerator 반환된 개체)는 정확한 최대 크기를 반환하지 않습니다.

예시

다음 코드 예제에서는 격리된 저장소를 가져오고, 몇 개의 파일을 만들고, 속성을 검색합니다 AvailableFreeSpace . 나머지 공간은 바이트 단위로 보고됩니다.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class CheckingSpace
{
    public static void Main()
    {
        // Get an isolated store for this assembly and put it into an
        // IsolatedStoreFile object.
        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
```

```
        // Create a few placeholder files in the isolated store.
        new IsolatedStorageFileStream("InTheRoot.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("Another.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("AThird.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("AFourth.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("AFifth.txt", FileMode.Create, isoStore);

        Console.WriteLine(isoStore.AvailableFreeSpace + " bytes of space remain in
this isolated store.");
    } // End of Main.
}
```

참고하십시오

- IsolatedStorageFile
- 격리된 스토리지
- 방법: 격리된 스토리지에 대한 저장소 가져오기

방법: 격리된 스토리지의 파일 및 디렉터리 만들기

아티클 • 2023. 04. 07.

격리된 저장소를 가져온 후에는 데이터를 저장할 디렉터리와 파일을 만들 수 있습니다. 저장소 내에서 파일 및 디렉터리 이름은 가상 파일 시스템의 루트와 관련하여 지정됩니다.

디렉터리를 만들려면 IsolatedStorageFile.CreateDirectory 인스턴스 메서드를 사용하십시오. 존재하지 않는 디렉터리의 하위 디렉터리를 지정하면 두 디렉터리가 모두 만들어지고 이미 존재하는 디렉터리를 지정하면 메서드가 디렉터리를 만들지 않은 채 반환되고 예외가 throw되지 않습니다. 그러나 잘못된 문자를 포함하는 디렉터리 이름을 지정하면 IsolatedStorageException 예외가 throw됩니다.

파일을 만들려면 메서드를 IsolatedStorageFile.CreateFile 사용합니다.

Windows 운영 체제에서 격리된 스토리지 파일 및 디렉터리 이름은 대/소문자를 구분하지 않습니다. 즉, `ThisFile.txt`라는 파일을 만든 다음 `THISFILE.TXT`라는 다른 파일을 만들면 한 파일만 만들어집니다. 파일 이름은 표시를 위해 원래 대소문자를 유지합니다.

격리된 스토리지 파일을 만들 때 존재하지 않는 디렉터리가 경로에 포함된 경우 IsolatedStorageException을 throw합니다.

예제

다음 코드 예제는 분리된 저장소에서 파일 및 디렉터리를 만드는 방법을 보여줍니다.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class CreatingFilesDirectories
{
    public static void Main()
    {
        using (IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null, null))
        {
            isoStore.CreateDirectory("TopLevelDirectory");
            isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
```

```
isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
            Console.WriteLine("Created directories.");

            isoStore.CreateFile("InTheRoot.txt");
            Console.WriteLine("Created a new file in the root.");


isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
            Console.WriteLine("Created a new file in the InsideDirectory.");
        }
    }
}
```

참조

- IsolatedStorageFile
- IsolatedStorageFileStream
- 격리된 스토리지

방법: 격리된 스토리지에서 기존 파일 및 디렉터리 찾기

아티클 • 2025. 05. 08.

격리된 스토리지에서 디렉터리를 검색하려면 이 메서드를 IsolatedStorageFile.GetDirectoryNames 사용합니다. 이 메서드는 검색 패턴을 나타내는 문자열을 사용합니다. 검색 패턴에서 단일 문자(?) 및 다중 문자(*) 와일드카드 문자를 모두 사용할 수 있지만 와일드카드 문자는 이름의 마지막 부분에 나타나야 합니다. 예를 들어 `directory1/*ect*` 유효한 검색 문자열이지만 `*ect*/directory2` 그렇지 않습니다.

파일을 검색하려면 메서드를 IsolatedStorageFile.GetFileNames 사용합니다. 에 적용되는 GetDirectoryNames 검색 문자열의 와일드카드 문자에 대한 제한 사항도 적용됩니다 GetFileNames.

이러한 메서드 중 어느 것도 재귀적이지 않습니다. 클래스는 저장소의 IsolatedStorageFile 모든 디렉터리 또는 파일을 나열하는 메서드를 제공하지 않습니다. 그러나 재귀 메서드는 다음 코드 예제에 나와 있습니다.

예시

다음 코드 예제에서는 격리된 저장소에서 파일 및 디렉터리를 만드는 방법을 보여 줍니다. 먼저 사용자, 도메인 및 어셈블리에 대해 격리된 저장소가 검색되어 변수에 `isoStore` 배치됩니다. 이 CreateDirectory 메서드는 몇 가지 다른 디렉터리를 설정하는 데 사용되며 IsolatedStorageFileStream(String, FileMode, IsolatedStorageFile) 생성자는 이러한 디렉터리에 일부 파일을 만듭니다. 그런 다음 코드는 메서드의 결과를 반복합니다 `GetAllDirectories` . 이 메서드는 현재 디렉터리에서 모든 디렉터리 이름을 찾는 데 사용합니다 GetDirectoryNames . 이러한 이름은 배열에 저장된 다음 `GetAllDirectories` , 자신을 호출하여 찾은 각 디렉터리를 전달합니다. 따라서 모든 디렉터리 이름이 배열에 반환됩니다. 다음으로, 코드는 메서드를 호출합니다 `GetAllFiles` . 이 메서드는 모든 디렉터리의 이름을 찾기 위해 호출 `GetAllDirectories` 한 다음 메서드를 사용하여 GetFileNames 각 디렉터리에서 파일을 확인합니다. 결과는 표시할 배열에 반환됩니다.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;
using System.Collections;
using System.Collections.Generic;

public class FindingExistingFilesAndDirectories
{
```

```
    // Retrieves an array of all directories in the store, and
    // displays the results.
    public static void Main()
    {
        // This part of the code sets up a few directories and files in the
        // store.
        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
        isoStore.CreateDirectory("TopLevelDirectory");
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        isoStore.CreateFile("InTheRoot.txt");

isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        // End of setup.

        Console.WriteLine('\r');
        Console.WriteLine("Here is a list of all directories in this isolated
store:");

        foreach (string directory in GetAllDirectories("*", isoStore))
        {
            Console.WriteLine(directory);
        }
        Console.WriteLine('\r');

        // Retrieve all the files in the directory by calling the GetFiles
        // method.

        Console.WriteLine("Here is a list of all the files in this isolated
store:");
        foreach (string file in GetAllFiles("*", isoStore)){
            Console.WriteLine(file);
        }
    } // End of Main.

    // Method to retrieve all directories, recursively, within a store.
    public static List<String> GetAllDirectories(string pattern,
IsolatedStorageFile storeFile)
    {
        // Get the root of the search string.
        string root = Path.GetDirectoryName(pattern);

        if (root != "")
        {
            root += "/";
        }

        // Retrieve directories.
        List<String> directoryList = new List<String>
(storeFile.GetDirectoryNames(pattern));

        // Retrieve subdirectories of matches.
        for (int i = 0, max = directoryList.Count; i < max; i++)
```

```
        {
            string directory = directoryList[i] + "/";
            List<String> more = GetAllDirectories(root + directory + "*",
storeFile);

            // For each subdirectory found, add in the base path.
            for (int j = 0; j < more.Count; j++)
            {
                more[j] = directory + more[j];
            }

            // Insert the subdirectories into the list and
            // update the counter and upper bound.
            directoryList.InsertRange(i + 1, more);
            i += more.Count;
            max += more.Count;
        }

        return directoryList;
    }

    public static List<String> GetAllFiles(string pattern, IsolatedStorageFile
storeFile)
    {
        // Get the root and file portions of the search string.
        string fileString = Path.GetFileName(pattern);

        List<String> fileList = new List<String>(storeFile.GetFileNames(pattern));

        // Loop through the subdirectories, collect matches,
        // and make separators consistent.
        foreach (string directory in GetAllDirectories("*", storeFile))
        {
            foreach (string file in storeFile.GetFileNames(directory + "/" +
fileString))
            {
                fileList.Add((directory + "/" + file));
            }
        }

        return fileList;
    } // End of GetFiles.
}
```

참고하십시오

- IsolatedStorageFile
- 격리된 스토리지

방법: 격리된 스토리지의 파일 읽기 및 쓰기

아티클 • 2024. 03. 14.

격리된 저장소에서 파일을 읽고 쓰기 위해, 스트림 판독기(IsolatedStorageFileStream 개체)를 가진 StreamReader 개체 또는 스트림 작성기(StreamWriter 개체)를 사용합니다.

예시

다음 코드 예제에서는 격리된 저장소를 가져오고 저장소에 TestStore.txt라는 파일이 있는지 여부를 확인합니다. 존재하지 않는 경우, 파일을 만들고 파일에 "Hello Isolated Storage"를 씁니다. TestStore.txt가 이미 있으면 예제 코드에서는 파일을 읽습니다.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null);

            if (isoStore.FileExists("TestStore.txt"))
            {
                Console.WriteLine("The file already exists!");
                using (IsolatedStorageFileStream isoStream = new
IsolatedStorageFileStream("TestStore.txt", FileMode.Open, isoStore))
                {
                    using (StreamReader reader = new
StreamReader(isoStream))
                    {
                        Console.WriteLine("Reading contents:");
                        Console.WriteLine(reader.ReadToEnd());
                    }
                }
            }
            else
            {
                using (IsolatedStorageFileStream isoStream = new
IsolatedStorageFileStream("TestStore.txt", FileMode.CreateNew, isoStore))
```

```
                {
                    using (StreamWriter writer = new
StreamWriter(isoStream))
                    {
                        writer.WriteLine("Hello Isolated Storage");
                        Console.WriteLine("You have written to the file.");
                    }
                }
            }
        }
    }
}
```

참고 항목

- IsolatedStorageFile
- IsolatedStorageFileStream
- System.IO.FileMode
- System.IO.FileAccess
- System.IO.StreamReader
- System.IO.StreamWriter
- 파일 및 스트림 I/O
- 격리된 스토리지

방법: 격리된 스토리지의 파일 및 디렉터리 삭제

아티클 • 2025. 04. 30.

격리된 스토리지 파일 내에서 디렉터리 및 파일을 삭제할 수 있습니다. 저장소 내에서 파일 및 디렉터리 이름은 운영 체제에 따라 달라지고 가상 파일 시스템의 루트에 상대적으로 지정됩니다. Windows 운영 체제에서는 대/소문자를 구분하지 않습니다.

클래스는 System.IO.IsolatedStorage.IsolatedStorageFile 디렉터리와 파일을 DeleteDirectoryDeleteFile삭제하는 두 가지 메서드를 제공합니다. IsolatedStorageException 존재하지 않는 파일 또는 디렉터리를 삭제하려고 하면 예외가 throw됩니다. 이름에 와일드카드 문자를 포함하면 DeleteDirectory가 IsolatedStorageException 예외를 던지고, DeleteFile가 ArgumentException 예외를 던집니다.

DeleteDirectory 디렉터리에 파일 또는 하위 디렉터리가 포함되어 있으면 메서드가 실패합니다. GetFileNames 및 GetDirectoryNames 메서드를 사용하여 기존 파일 및 디렉토리를 검색할 수 있습니다. 저장소의 가상 파일 시스템을 검색하는 방법에 대한 자세한 내용은 방법: 격리된 스토리지에서 기존 파일 및 디렉터리 찾기를 참조하세요.

예시

다음 코드 예제에서는 여러 디렉터리와 파일을 만든 다음 삭제합니다.

C#

```
using System;
using System.IO.IsolatedStorage;
using System.IO;

public class DeletingFilesDirectories
{
    public static void Main()
    {
        // Get a new isolated store for this user domain and assembly.
        // Put the store into an isolatedStorageFile object.

        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null,
null);

        Console.WriteLine("Creating Directories:");

        // This code creates several different directories.

        isoStore.CreateDirectory("TopLevelDirectory");
```

```
        Console.WriteLine("TopLevelDirectory");
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
        Console.WriteLine("TopLevelDirectory/SecondLevel");

        // This code creates two new directories, one inside the other.

        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory");
        Console.WriteLine();

        // This code creates a few files and places them in the directories.

        Console.WriteLine("Creating Files:");

        // This file is placed in the root.

        IsolatedStorageFileStream isoStream1 = new
IsolatedStorageFileStream("InTheRoot.txt",
            FileMode.Create, isoStore);
        Console.WriteLine("InTheRoot.txt");

        isoStream1.Close();

        // This file is placed in the InsideDirectory.

        IsolatedStorageFileStream isoStream2 = new IsolatedStorageFileStream(
            "AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt",
FileMode.Create, isoStore);
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console.WriteLine();

        isoStream2.Close();

        Console.WriteLine("Deleting File:");

        // This code deletes the HereIAm.txt file.

isoStore.DeleteFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console.WriteLine();

        Console.WriteLine("Deleting Directory:");

        // This code deletes the InsideDirectory.

        isoStore.DeleteDirectory("AnotherTopLevelDirectory/InsideDirectory/");
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/");
        Console.WriteLine();
    } // End of main.
}
```

참고하십시오

- System.IO.IsolatedStorage.IsolatedStorageFile
- 격리된 스토리지

.NET의 파이프 연산

2025. 06. 17.

파이프는 프로세스 간 통신을 위한 수단을 제공합니다. 파이프에는 두 가지 유형이 있습니다.

- 익명 파이프.

  익명 파이프는 로컬 컴퓨터에서 프로세스 간 통신을 제공합니다. 익명 파이프는 명명된 파이프보다 오버헤드가 적지만 제한된 서비스를 제공합니다. 익명 파이프는 단방향이며 네트워크를 통해 사용할 수 없습니다. 단일 서버 인스턴스만 지원합니다. 익명 파이프는 스레드 간의 통신이나 프로세스가 생성될 때 파이프 핸들을 자식 프로세스에 쉽게 전달할 수 있는 부모와 자식 프로세스 간의 통신에 유용합니다.

  .NET에서는 AnonymousPipeServerStream 및 AnonymousPipeClientStream 클래스를 사용하여 익명 파이프를 구현합니다.

  방법: 로컬 프로세스 간 통신에 익명 파이프 사용

- 명명된 파이프

  명명된 파이프는 파이프 서버와 하나 이상의 파이프 클라이언트 간에 프로세스 간 통신을 제공합니다. 명명된 파이프는 단방향 또는 이중 파이프일 수 있습니다. 메시지 기반 통신을 지원하며 여러 클라이언트가 동일한 파이프 이름을 사용하여 서버 프로세스에 동시에 연결할 수 있습니다. 명명 파이프는 연결 프로세스가 원격 서버에서 자신의 권한을 사용할 수 있도록 하는 권한 대행도 지원합니다.

  .NET에서는 NamedPipeServerStream 및 NamedPipeClientStream 클래스를 사용하여 명명된 파이프를 구현합니다.

  방법: 네트워크 프로세스 간 통신에 명명된 파이프 사용

참고하십시오

- 파일 및 스트림 I/O
- 방법: 로컬 프로세스 간 통신에 익명 파이프 사용
- 방법: 네트워크 프로세스 간 통신에 명명된 파이프 사용

방법: 로컬 프로세스 간 통신에 익명 파이프 사용

아티클 • 2025. 03. 29.

익명 파이프는 로컬 컴퓨터에서 프로세스 간 통신을 제공합니다. 명명된 파이프보다 적은 기능을 제공하지만 오버헤드가 적습니다. 익명 파이프를 사용하여 로컬 컴퓨터에서 프로세스 간 통신을 더 쉽게 만들 수 있습니다. 네트워크를 통한 통신에는 익명 파이프를 사용할 수 없습니다.

익명 파이프를 구현하려면 AnonymousPipeServerStream 및 AnonymousPipeClientStream 클래스를 사용합니다.

예제 1

다음 예제에서는 익명 파이프를 사용하여 부모 프로세스에서 자식 프로세스로 문자열을 보내는 방법을 보여 줍니다. 이 예제에서는 PipeDirection 값이 Out인 AnonymousPipeServerStream 개체를 부모 프로세스에서 생성합니다. 그런 다음 부모 프로세스는 클라이언트 핸들을 사용하여 AnonymousPipeClientStream 개체를 생성해 자식 프로세스를 만듭니다. 자식 프로세스의 PipeDirection 값은 In.

그런 다음 부모 프로세스는 사용자가 제공한 문자열을 자식 프로세스로 보냅니다. 문자열은 자식 프로세스에서 콘솔에 표시됩니다.

다음 예제에서는 서버 프로세스를 보여줍니다.

C#

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Diagnostics;

class PipeServer
{
    static void Main()
    {
        Process pipeClient = new Process();

        pipeClient.StartInfo.FileName = "pipeClient.exe";

        using (AnonymousPipeServerStream pipeServer =
            new AnonymousPipeServerStream(PipeDirection.Out,
            HandleInheritability.Inheritable))
        {
            Console.WriteLine($"[SERVER] Current TransmissionMode:
```

```
{pipeServer.TransmissionMode}.");

            // Pass the client process a handle to the server.
            pipeClient.StartInfo.Arguments =
                pipeServer.GetClientHandleAsString();
            pipeClient.StartInfo.UseShellExecute = false;
            pipeClient.Start();

            pipeServer.DisposeLocalCopyOfClientHandle();

            try
            {
                // Read user input and send that to the client process.
                using (StreamWriter sw = new StreamWriter(pipeServer))
                {
                    sw.AutoFlush = true;
                    // Send a 'sync message' and wait for client to receive
it.
                    sw.WriteLine("SYNC");
                    pipeServer.WaitForPipeDrain();
                    // Send the console input to the client process.
                    Console.Write("[SERVER] Enter text: ");
                    sw.WriteLine(Console.ReadLine());
                }
            }
            // Catch the IOException that is raised if the pipe is broken
            // or disconnected.
            catch (IOException e)
            {
                Console.WriteLine($"[SERVER] Error: {e.Message}");
            }
        }

        pipeClient.WaitForExit();
        pipeClient.Close();
        Console.WriteLine("[SERVER] Client quit. Server terminating.");
    }
}
```

예제 2

다음 예제에서는 클라이언트 프로세스를 보여줍니다. 서버 프로세스는 클라이언트 프로세스를 시작하고 해당 프로세스에 클라이언트 핸들을 제공합니다. 클라이언트 코드의 결과 실행 파일은 `pipeClient.exe` 이름을 지정하고 서버 프로세스를 실행하기 전에 서버 실행 파일과 동일한 디렉터리에 복사해야 합니다.

C#

```
using System;
using System.IO;
```

```
using System.IO.Pipes;

class PipeClient
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            using (PipeStream pipeClient =
                new AnonymousPipeClientStream(PipeDirection.In, args[0]))
            {
                Console.WriteLine($"[CLIENT] Current TransmissionMode:
{pipeClient.TransmissionMode}.");

                using (StreamReader sr = new StreamReader(pipeClient))
                {
                    // Display the read text to the console
                    string temp;

                    // Wait for 'sync message' from the server.
                    do
                    {
                        Console.WriteLine("[CLIENT] Wait for sync...");
                        temp = sr.ReadLine();
                    }
                    while (!temp.StartsWith("SYNC"));

                    // Read the server data and echo to the console.
                    while ((temp = sr.ReadLine()) != null)
                    {
                        Console.WriteLine("[CLIENT] Echo: " + temp);
                    }
                }
            }
        }
        Console.Write("[CLIENT] Press Enter to continue...");
        Console.ReadLine();
    }
}
```

참고하십시오

- 파이프
- 방법: 네트워크 프로세스 간 통신에 명명된 파이프 사용

방법: 네트워크 프로세스 간 통신에 명명된 파이프 사용

아티클 • 2025. 03. 29.

명명된 파이프는 파이프 서버와 하나 이상의 파이프 클라이언트 간에 프로세스 간 통신을 제공합니다. 로컬 컴퓨터에서 프로세스 간 통신을 제공하는 익명 파이프보다 더 많은 기능을 제공합니다. 명명된 파이프는 네트워크를 통한 전체 이중 통신을 지원하며, 여러 서버 인스턴스와 메시지 기반 통신, 클라이언트 가장이 가능합니다. 이를 통해 연결된 프로세스는 원격 서버에서 자신의 권한 집합을 사용할 수 있습니다.

> **ⓘ 중요**
>
> Linux의 .NET은 이러한 API 구현에 UDS(Unix Domain Sockets)를 사용합니다.

이름 파이프를 구현하려면 NamedPipeServerStream 및 NamedPipeClientStream 클래스를 사용합니다.

예제 1

다음 예제에서는 NamedPipeServerStream 클래스를 사용하여 명명된 파이프를 만드는 방법을 보여 줍니다. 이 예제에서 서버 프로세스는 네 개의 스레드를 만듭니다. 각 스레드는 클라이언트 연결을 수락할 수 있습니다. 그러면 연결된 클라이언트 프로세스에서 서버에 파일 이름을 제공합니다. 클라이언트에 충분한 권한이 있는 경우 서버 프로세스는 파일을 열고 해당 내용을 클라이언트로 다시 보냅니다.

C#

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Text;
using System.Threading;

public class PipeServer
{
    private static int numThreads = 4;

    public static void Main()
    {
        int i;
        Thread?[] servers = new Thread[numThreads];

        Console.WriteLine("\n*** Named pipe server stream with impersonation
```

```
example ***\n");
        Console.WriteLine("Waiting for client connect...\n");
        for (i = 0; i < numThreads; i++)
        {
            servers[i] = new Thread(ServerThread);
            servers[i]?.Start();
        }
        Thread.Sleep(250);
        while (i > 0)
        {
            for (int j = 0; j < numThreads; j++)
            {
                if (servers[j] != null)
                {
                    if (servers[j]!.Join(250))
                    {
                        Console.WriteLine($"Server
thread[{servers[j]!.ManagedThreadId}] finished.");
                        servers[j] = null;
                        i--;    // decrement the thread watch count
                    }
                }
            }
        }
        Console.WriteLine("\nServer threads exhausted, exiting.");
    }

    private static void ServerThread(object? data)
    {
        NamedPipeServerStream pipeServer =
            new NamedPipeServerStream("testpipe", PipeDirection.InOut,
numThreads);

        int threadId = Thread.CurrentThread.ManagedThreadId;

        // Wait for a client to connect
        pipeServer.WaitForConnection();

        Console.WriteLine($"Client connected on thread[{threadId}].");
        try
        {
            // Read the request from the client. Once the client has
            // written to the pipe its security token will be available.

            StreamString ss = new StreamString(pipeServer);

            // Verify our identity to the connected client using a
            // string that the client anticipates.

            ss.WriteString("I am the one true server!");
            string filename = ss.ReadString();

            // Read in the contents of the file while impersonating the
client.
            ReadFileToStream fileReader = new ReadFileToStream(ss,
```

```
filename);

            // Display the name of the user we are impersonating.
            Console.WriteLine($"Reading file: {filename} on
thread[{threadId}] as user: {pipeServer.GetImpersonationUserName()}.");
            pipeServer.RunAsClient(fileReader.Start);
        }
        // Catch the IOException that is raised if the pipe is broken
        // or disconnected.
        catch (IOException e)
        {
            Console.WriteLine($"ERROR: {e.Message}");
        }
        pipeServer.Close();
    }
}

// Defines the data protocol for reading and writing strings on our stream
public class StreamString
{
    private Stream ioStream;
    private UnicodeEncoding streamEncoding;

    public StreamString(Stream ioStream)
    {
        this.ioStream = ioStream;
        streamEncoding = new UnicodeEncoding();
    }

    public string ReadString()
    {
        int len = 0;

        len = ioStream.ReadByte() * 256;
        len += ioStream.ReadByte();
        byte[] inBuffer = new byte[len];
        ioStream.Read(inBuffer, 0, len);

        return streamEncoding.GetString(inBuffer);
    }

    public int WriteString(string outString)
    {
        byte[] outBuffer = streamEncoding.GetBytes(outString);
        int len = outBuffer.Length;
        if (len > UInt16.MaxValue)
        {
            len = (int)UInt16.MaxValue;
        }
        ioStream.WriteByte((byte)(len / 256));
        ioStream.WriteByte((byte)(len & 255));
        ioStream.Write(outBuffer, 0, len);
        ioStream.Flush();

        return outBuffer.Length + 2;
```

```
    }
}

// Contains the method executed in the context of the impersonated user
public class ReadFileToStream
{
    private string fn;
    private StreamString ss;

    public ReadFileToStream(StreamString str, string filename)
    {
        fn = filename;
        ss = str;
    }

    public void Start()
    {
        string contents = File.ReadAllText(fn);
        ss.WriteString(contents);
    }
}
```

예제 2

다음 예제에서는 NamedPipeClientStream 클래스를 사용하는 클라이언트 프로세스를 보여줍니다. 클라이언트는 서버 프로세스에 연결하고 파일 이름을 서버로 보냅니다. 이 예제에서는 사칭을 사용하므로 클라이언트 애플리케이션의 ID에 파일에 대한 접근 권한이 있어야 합니다. 그런 다음 서버는 파일의 내용을 클라이언트로 다시 보냅니다. 그런 다음 파일 내용이 콘솔에 표시됩니다.

C#

```
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.Security.Principal;
using System.Text;
using System.Threading;

public class PipeClient
{
    private static int numClients = 4;

    public static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            if (args[0] == "spawnclient")
            {
```

```
                var pipeClient =
                    new NamedPipeClientStream(".", "testpipe",
                        PipeDirection.InOut, PipeOptions.None,
                        TokenImpersonationLevel.Impersonation);

                Console.WriteLine("Connecting to server...\n");
                pipeClient.Connect();

                var ss = new StreamString(pipeClient);
                // Validate the server's signature string.
                if (ss.ReadString() == "I am the one true server!")
                {
                    // The client security token is sent with the first
write.
                    // Send the name of the file whose contents are returned
                    // by the server.
                    ss.WriteString("c:\\textfile.txt");

                    // Print the file to the screen.
                    Console.Write(ss.ReadString());
                }
                else
                {
                    Console.WriteLine("Server could not be verified.");
                }
                pipeClient.Close();
                // Give the client process some time to display results
before exiting.
                Thread.Sleep(4000);
            }
        }
        else
        {
            Console.WriteLine("\n*** Named pipe client stream with
impersonation example ***\n");
            StartClients();
        }
    }

    // Helper function to create pipe client processes
    private static void StartClients()
    {
        string currentProcessName = Environment.CommandLine;

        // Remove extra characters when launched from Visual Studio
        currentProcessName = currentProcessName.Trim('"', ' ');

        currentProcessName = Path.ChangeExtension(currentProcessName,
".exe");
        Process?[] plist = new Process?[numClients];

        Console.WriteLine("Spawning client processes...\n");

        if (currentProcessName.Contains(Environment.CurrentDirectory))
        {
```

```
            currentProcessName =
currentProcessName.Replace(Environment.CurrentDirectory, String.Empty);
        }

        // Remove extra characters when launched from Visual Studio
        currentProcessName = currentProcessName.Replace("\\", String.Empty);
        currentProcessName = currentProcessName.Replace("\"", String.Empty);

        int i;
        for (i = 0; i < numClients; i++)
        {
            // Start 'this' program but spawn a named pipe client.
            plist[i] = Process.Start(currentProcessName, "spawnclient");
        }
        while (i > 0)
        {
            for (int j = 0; j < numClients; j++)
            {
                if (plist[j] != null)
                {
                    if (plist[j]!.HasExited)
                    {
                        Console.WriteLine($"Client process[{plist[j]?.Id}]
has exited.");
                        plist[j] = null;
                        i--;    // decrement the process watch count
                    }
                    else
                    {
                        Thread.Sleep(250);
                    }
                }
            }
        }
        Console.WriteLine("\nClient processes finished, exiting.");
    }
}

// Defines the data protocol for reading and writing strings on our stream.
public class StreamString
{
    private Stream ioStream;
    private UnicodeEncoding streamEncoding;

    public StreamString(Stream ioStream)
    {
        this.ioStream = ioStream;
        streamEncoding = new UnicodeEncoding();
    }

    public string ReadString()
    {
        int len;
        len = ioStream.ReadByte() * 256;
        len += ioStream.ReadByte();
```

```
            var inBuffer = new byte[len];
            ioStream.Read(inBuffer, 0, len);

            return streamEncoding.GetString(inBuffer);
        }

        public int WriteString(string outString)
        {
            byte[] outBuffer = streamEncoding.GetBytes(outString);
            int len = outBuffer.Length;
            if (len > UInt16.MaxValue)
            {
                len = (int)UInt16.MaxValue;
            }
            ioStream.WriteByte((byte)(len / 256));
            ioStream.WriteByte((byte)(len & 255));
            ioStream.Write(outBuffer, 0, len);
            ioStream.Flush();

            return outBuffer.Length + 2;
        }
    }
```

강력한 프로그래밍

이 예제의 클라이언트 및 서버 프로세스는 동일한 컴퓨터에서 실행되므로 NamedPipeClientStream 개체에 제공된 서버 이름은 `"."`. 클라이언트 및 서버 프로세스가 별도의 컴퓨터에 있는 경우 `"."` 서버 프로세스를 실행하는 컴퓨터의 네트워크 이름으로 바꿉 수 있습니다.

참고하십시오

- TokenImpersonationLevel
- GetImpersonationUserName
- 파이프
- 방법: 로컬 프로세스 간 통신을 위한 익명 파이프 사용법

System.IO.Pipelines

System.IO.Pipelines 는 .NET에서 고성능 I/O를 더 쉽게 만들 수 있도록 설계된 라이브러리입니다. 이 패키지는 광범위한 호환성, .NET Framework 및 최신 .NET을 위해 .NET Standard를 대상으로 합니다. 최신 .NET 버전 `System.IO.Pipelines` 에서는 공유 프레임워크에 포함되며 별도의 NuGet 패키지가 필요하지 않습니다.

라이브러리는 System.IO.Pipelines NuGet 패키지로도 사용할 수 있습니다.

System.IO.Pipelines는 어떤 문제를 해결하나요?

스트리밍 데이터를 구문 분석하는 앱은 특수하고 비정상적인 코드 흐름이 많은 상용구 코드로 구성됩니다. 상용구와 특수 사례 코드는 복잡하고 유지관리가 어렵습니다.

`System.IO.Pipelines`는 다음을 위해 설계되었습니다.

- 고성능으로 스트리밍 데이터를 구문 분석합니다.
- 코드 복잡성 감소

이 코드는 클라이언트에서 줄로 구분된 메시지(구분 기호로 구분 `'\n'`됨)를 수신하는 TCP 서버에 일반적입니다.

C#

```
async Task ProcessLinesAsync(NetworkStream stream)
{
    var buffer = new byte[1024];
    await stream.ReadAsync(buffer, 0, buffer.Length);

    // Process a single line from the buffer
    ProcessLine(buffer);
}
```

위의 코드에는 몇 가지 문제가 있습니다.

- `ReadAsync`에 대한 단일 호출에서 전체 메시지(줄의 끝)를 수신하지 못할 수 있습니다.
- `stream.ReadAsync`의 결과를 무시합니다. `stream.ReadAsync`가 읽은 데이터의 양을 반환합니다.
- 단일 `ReadAsync` 호출에서 여러 줄을 읽는 경우를 처리하지 않습니다.
- 각 읽기에 `byte` 배열을 할당합니다.

위의 문제를 해결하려면 다음을 변경합니다.

- 새 줄을 찾을 때까지 들어오는 데이터를 버퍼링합니다.

- 버퍼에 반환된 모든 줄을 구문 분석합니다.
- 줄이 1KB(1024바이트)보다 클 수 있습니다. 코드는 구분 기호가 버퍼 내의 전체 줄에 맞도록 찾을 때까지 입력 버퍼의 크기를 조정해야 합니다.
  - 버퍼 크기를 조정하는 경우 입력에 더 긴 줄이 표시되면 더 많은 버퍼 복사본이 생성됩니다.
  - 불필요한 공간을 줄이려면 줄 읽기에 사용되는 버퍼를 압축합니다.
- 메모리를 반복해서 할당하지 않도록 버퍼 풀링을 사용하는 것이 좋습니다.
- 이 코드는 다음과 같은 몇 가지 문제를 해결합니다.

C#

```
async Task ProcessLinesAsync(NetworkStream stream)
{
    byte[] buffer = ArrayPool<byte>.Shared.Rent(1024);
    var bytesBuffered = 0;
    var bytesConsumed = 0;

    while (true)
    {
        // Calculate the amount of bytes remaining in the buffer.
        var bytesRemaining = buffer.Length - bytesBuffered;

        if (bytesRemaining == 0)
        {
            // Double the buffer size and copy the previously buffered data into
the new buffer.
            var newBuffer = ArrayPool<byte>.Shared.Rent(buffer.Length * 2);
            Buffer.BlockCopy(buffer, 0, newBuffer, 0, buffer.Length);
            // Return the old buffer to the pool.
            ArrayPool<byte>.Shared.Return(buffer);
            buffer = newBuffer;
            bytesRemaining = buffer.Length - bytesBuffered;
        }

        var bytesRead = await stream.ReadAsync(buffer, bytesBuffered,
bytesRemaining);
        if (bytesRead == 0)
        {
            // EOF
            break;
        }

        // Keep track of the amount of buffered bytes.
        bytesBuffered += bytesRead;
        var linePosition = -1;

        do
        {
            // Look for a EOL in the buffered data.
```

```
            linePosition = Array.IndexOf(buffer, (byte)'\n', bytesConsumed,
                                         bytesBuffered - bytesConsumed);

            if (linePosition >= 0)
            {
                // Calculate the length of the line based on the offset.
                var lineLength = linePosition - bytesConsumed;

                // Process the line.
                ProcessLine(buffer, bytesConsumed, lineLength);

                // Move the bytesConsumed to skip past the line consumed (including
\n).
                bytesConsumed += lineLength + 1;
            }
        }
        while (linePosition >= 0);
    }
}
```

이전 코드는 복잡하며, 식별된 모든 문제를 해결하지는 않습니다. 고성능 네트워킹은 일반적으로 성능을 최대화하는 매우 복잡한 코드를 작성하는 것을 의미합니다. `System.IO.Pipelines`는 이러한 종류의 코드 작성을 더 쉽게 만들기 위해 설계되었습니다.

파이프

Pipe 클래스를 사용하여 `PipeWriter/PipeReader` 쌍을 만듭니다. `PipeWriter`에 작성된 모든 데이터는 `PipeReader`에서 사용할 수 있습니다.

C#

```
var pipe = new Pipe();
PipeReader reader = pipe.Reader;
PipeWriter writer = pipe.Writer;
```

파이프 기본 사용 방법

C#

```
async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();
    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
}
```

```
async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    const int minimumBufferSize = 512;

    while (true)
    {
        // Allocate at least 512 bytes from the PipeWriter.
        Memory<byte> memory = writer.GetMemory(minimumBufferSize);
        try
        {
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);
            if (bytesRead == 0)
            {
                break;
            }
            // Tell the PipeWriter how much was read from the Socket.
            writer.Advance(bytesRead);
        }
        catch (Exception ex)
        {
            LogError(ex);
            break;
        }

        // Make the data available to the PipeReader.
        FlushResult result = await writer.FlushAsync();

        if (result.IsCompleted)
        {
            break;
        }
    }

     // By completing PipeWriter, tell the PipeReader that there's no more data
coming.
    await writer.CompleteAsync();
}

async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync();
        ReadOnlySequence<byte> buffer = result.Buffer;

        while (TryReadLine(ref buffer, out ReadOnlySequence<byte> line))
        {
            // Process the line.
            ProcessLine(line);
        }

        // Tell the PipeReader how much of the buffer has been consumed.
        reader.AdvanceTo(buffer.Start, buffer.End);

        // Stop reading if there's no more data coming.
```

```
        if (result.IsCompleted)
        {
            break;
        }
    }

    // Mark the PipeReader as complete.
    await reader.CompleteAsync();
}

bool TryReadLine(ref ReadOnlySequence<byte> buffer, out ReadOnlySequence<byte>
line)
{
    // Look for a EOL in the buffer.
    SequencePosition? position = buffer.PositionOf((byte)'\n');

    if (position == null)
    {
        line = default;
        return false;
    }

    // Skip the line + the \n.
    line = buffer.Slice(0, position.Value);
    buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
    return true;
}
```

두 루프는 읽기 및 쓰기를 처리합니다.

- `FillPipeAsync`는 `Socket`에서 읽고 `PipeWriter`에 씁니다.
- `ReadPipeAsync`는 `PipeReader`에서 읽고 들어오는 줄을 구문 분석합니다.

명시적 버퍼가 할당되지 않습니다. 모든 버퍼 관리는 `PipeReader` 및 `PipeWriter` 구현에 위임됩니다. 버퍼 관리를 위임하면 더 쉽게 코드 사용을 비즈니스 논리에만 집중할 수 있습니다.

첫 번째 루프에서 다음을 수행합니다.

- PipeWriter.GetMemory(Int32)는 기본 작성기에서 메모리를 가져오기 위해 호출됩니다.
- PipeWriter.Advance(Int32)는 버퍼에 기록된 데이터의 양을 `PipeWriter`에 알리기 위해 호출됩니다.
- PipeWriter.FlushAsync는 `PipeReader`에서 데이터를 사용할 수 있도록 호출됩니다.

두 번째 루프에서 `PipeReader`는 `PipeWriter`로 작성된 버퍼를 사용합니다. 버퍼는 소켓에서 나옵니다. PipeReader.ReadAsync에 대한 호출:

- 두 가지 중요 한 정보를 포함하는 ReadResult를 반환합니다.
  - ReadOnlySequence<T> 형식으로 읽은 데이터입니다.
  - EOF(데이터의 끝)에 도달했는지 여부를 나타내는 부울 `IsCompleted`입니다.

EOL(줄의 끝) 구분 기호를 찾은 후 줄을 구문 분석합니다.

- 논리는 버퍼를 처리하여 이미 처리된 작업을 건너뜁니다.
- PipeReader.AdvanceTo를 호출하여 사용되고 검사된 데이터의 양을 `PipeReader`에 알려줍니다.

판독기 및 기록기 루프는 호출 PipeReader.Complete 및 PipeWriter.Complete로 끝납니다. 호출 `Complete` 하면 기본 `Pipe` 할당된 메모리가 해제됩니다.

역 압력 및 흐름 제어

다음과 같이 읽기와 구문 분석이 함께 작동하는 것이 가장 좋습니다.

- 쓰기 스레드는 네트워크의 데이터를 사용하며 이를 버퍼에 저장합니다.
- 구문 분석 스레드는 적절한 데이터 구조를 생성합니다.

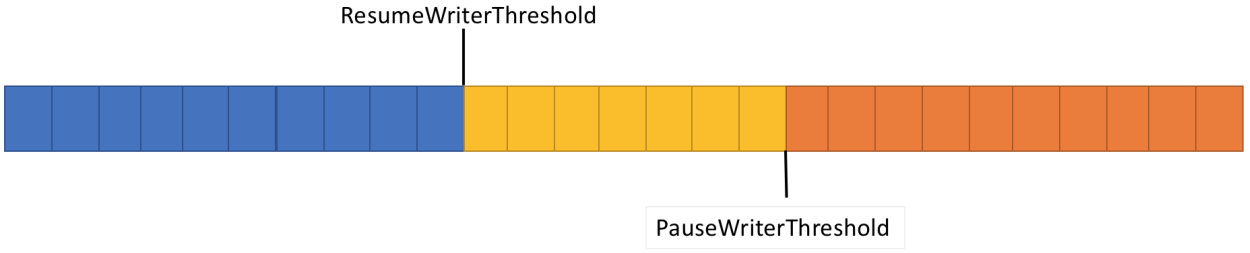
일반적으로 구문 분석에는 네트워크에서 데이터 블록을 복사하는 것보다 시간이 더 걸립니다.

- 읽기 스레드가 구문 분석 스레드 앞에 있습니다.
- 읽기 스레드의 속도가 느려야 하거나 구문 분석 스레드에 대한 데이터를 저장하기 위해 더 많은 메모리를 할당해야 합니다.

성능을 최적화하기 위해 잦은 일시 중지와 더 많은 메모리 할당 사이에서 균형을 유지합니다.

위의 문제를 해결하기 위해 `Pipe`에는 데이터 흐름을 제어하는 두 가지 설정이 있습니다.

- PauseWriterThreshold: FlushAsync 일시 중지를 호출하기 전에 버퍼링되어야 하는 데이터의 양을 결정합니다.
- ResumeWriterThreshold: PipeWriter.FlushAsync 호출을 다시 시작하기 전에 판독기가 관찰해야 하는 데이터의 양을 결정합니다.



:

- `ValueTask<FlushResult>`의 데이터 크기가 `Pipe`를 넘으면 불완전한 `PauseWriterThreshold`를 반환합니다.
- `ValueTask<FlushResult>`가 `ResumeWriterThreshold`보다 작아지면 완료됩니다.

하나의 값이 사용될 경우 발생할 수 있는 빠른 순환을 방지하기 위해 두 값이 사용됩니다.

예제

C#

```
// The Pipe will start returning incomplete tasks from FlushAsync until
// the reader examines at least 5 bytes.
var options = new PipeOptions(pauseWriterThreshold: 10, resumeWriterThreshold: 5);
var pipe = new Pipe(options);
```

파이프 스케줄러

일반적으로 `async` 및 `await`를 사용하는 경우 비동기 코드는 TaskScheduler 또는 현재 SynchronizationContext에서 다시 시작됩니다.

I/O를 수행하는 경우 I/O가 수행되는 위치를 세부적으로 제어하는 것이 중요합니다. 이 컨트롤을 사용하면 CPU 캐시를 효과적으로 활용할 수 있습니다. 효율적인 캐싱은 웹 서버와 같은 고성능 앱에 매우 중요합니다. PipeScheduler는 비동기 콜백이 실행되는 위치에 대한 제어를 제공합니다. 기본적으로:

- 현재 SynchronizationContext가 사용됩니다.
- `SynchronizationContext`가 없으면 스레드 풀을 사용하여 콜백을 실행합니다.

C#

```
public static void Main(string[] args)
{
    var writeScheduler = new SingleThreadPipeScheduler();
    var readScheduler = new SingleThreadPipeScheduler();

    // Tell the Pipe what schedulers to use and disable the SynchronizationContext.
    var options = new PipeOptions(readerScheduler: readScheduler,
                                  writerScheduler: writeScheduler,
                                  useSynchronizationContext: false);
    var pipe = new Pipe(options);
}

// This is a sample scheduler that async callbacks on a single dedicated thread.
public class SingleThreadPipeScheduler : PipeScheduler
{
    private readonly BlockingCollection<(Action<object> Action, object State)>
_queue =
     new BlockingCollection<(Action<object> Action, object State)>();
    private readonly Thread _thread;

    public SingleThreadPipeScheduler()
    {
```

```
            _thread = new Thread(DoWork);
            _thread.Start();
        }

        private void DoWork()
        {
            foreach (var item in _queue.GetConsumingEnumerable())
            {
                item.Action(item.State);
            }
        }

        public override void Schedule(Action<object?> action, object? state)
        {
            if (state is not null)
            {
                _queue.Add((action, state));
            }
            // else log the fact that _queue.Add was not called.
        }
    }
```

PipeScheduler.ThreadPool은 스레드 풀에 대한 콜백을 큐에 대기시키는 PipeScheduler 구현입니다. `PipeScheduler.ThreadPool`은 기본값이며 일반적으로 최상의 선택입니다.
PipeScheduler.Inline은 교착 상태와 같은 의도하지 않은 결과를 발생시킬 수 있습니다.

파이프 다시 설정

개체를 재사용하는 `Pipe` 것이 효율적인 경우가 많습니다. 파이프를 다시 설정하려면 PipeReader와 Reset가 모두 완료될 때 `PipeReader` `PipeWriter`을 호출합니다.

PipeReader

PipeReader는 호출자를 대신하여 메모리를 관리합니다. **항상**PipeReader.AdvanceTo를 호출한 후 PipeReader.ReadAsync를 호출합니다. 이를 통해 `PipeReader`는 호출자가 메모리를 작업을 완료하는 시기를 알 수 있어서 메모리를 추적할 수 있습니다. PipeReader.ReadAsync에서 반환된 ReadOnlySequence<T>은(는) PipeReader.AdvanceTo가 호출될 때까지만 유효합니다. ReadOnlySequence<T>를 호출한 후에는 PipeReader.AdvanceTo를 사용할 수 없습니다.

PipeReader.AdvanceTo는 두 개의 SequencePosition 인수를 사용합니다.

- 첫 번째 인수는 사용된 메모리의 양을 결정합니다.
- 두 번째 인수는 관찰된 버퍼의 양을 결정합니다.

데이터를 사용됨으로 표시하면 파이프가 메모리를 기본 버퍼 풀로 반환할 수 있다는 의미입니다. 데이터를 관찰됨으로 표시하면 PipeReader.ReadAsync에 대한 다음 호출의 수행 내용을 제어

합니다. 모든 항목을 관찰됨으로 표시하면 파이프에 더 많은 데이터를 쓸 때까지 PipeReader.ReadAsync에 대한 다음 호출이 반환되지 않는다는 의미입니다. 다른 값은 관찰된 데이터와 관찰되지 않은 데이터를 포함하여 PipeReader.ReadAsync 호출에 즉시 반영하지만 이미 사용된 데이터는 반환하지 않습니다.

스트리밍 데이터 시나리오 읽기

스트리밍 데이터를 읽을 때 다음과 같은 몇 가지 일반적인 패턴이 나타납니다.

- 데이터 스트림이 지정된 경우 단일 메시지를 구문 분석합니다.
- 데이터 스트림이 지정된 경우 사용 가능한 모든 메시지를 구문 분석합니다.

다음 예제에서는 `TryParseLines` 메서드를 사용하여 ReadOnlySequence<T>에서 메시지를 구문 분석합니다. `TryParseLines`는 단일 메시지를 구문 분석하고 입력 버퍼를 업데이트하여 버퍼에서 구문 분석된 메시지를 자릅니다. `TryParseLines` 은 .NET의 일부가 아닙니다. 이 메서드는 다음 섹션에서 사용되는 사용자 작성 메서드입니다.

C#

```
bool TryParseLines(ref ReadOnlySequence<byte> buffer, out Message message);
```

단일 메시지 읽기

이 코드는 `PipeReader`에서 단일 메시지를 읽어 호출자에게 반환합니다.

C#

```
async ValueTask<Message?> ReadSingleMessageAsync(PipeReader reader,
 CancellationToken cancellationToken = default)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(cancellationToken);
        ReadOnlySequence<byte> buffer = result.Buffer;

        // In the event that no message is parsed successfully, mark consumed
        // as nothing and examined as the entire buffer.
        SequencePosition consumed = buffer.Start;
        SequencePosition examined = buffer.End;

        try
        {
            if (TryParseLines(ref buffer, out Message message))
            {
                // A single message was successfully parsed so mark the start of
the
```

```
                // parsed buffer as consumed. TryParseLines trims the buffer to
                // point to the data after the message was parsed.
                consumed = buffer.Start;

                // Examined is marked the same as consumed here, so the next call
                // to ReadSingleMessageAsync will process the next message if
there's
                // one.
                examined = consumed;

                return message;
            }

            // There's no more data to be processed.
            if (result.IsCompleted)
            {
                if (buffer.Length > 0)
                {
                    // The message is incomplete and there's no more data to
process.
                    throw new InvalidDataException("Incomplete message.");
                }

                break;
            }
        }
        finally
        {
            reader.AdvanceTo(consumed, examined);
        }
    }

    return null;
}
```

앞의 코드가 하는 역할은 다음과 같습니다.

- 단일 메시지를 구문 분석합니다.
- 소비된 `SequencePosition`을 업데이트하고, 검사된 `SequencePosition`이 잘린 입력 버퍼의 시작을 가리키도록 합니다.

`SequencePosition`는 입력 버퍼에서 구문 분석된 메시지를 제거하므로 두 개의 `TryParseLines` 인수가 업데이트됩니다. 일반적으로 버퍼에서 단일 메시지를 구문 분석할 때 검사된 위치는 다음 중 하나여야 합니다.

- 메시지 끝
- 메시지를 찾을 수 없는 경우 받은 버퍼의 끝

단일 메시지 사례에서 오류가 발생할 가능성이 가장 높습니다. *잘못된 값을 검사*에 전달하면 메모리 부족 예외 또는 무한 루프가 발생할 수 있습니다. 자세한 내용은 이 문서의 PipeReader의 일반적인 문제 섹션을 참조하세요.

> ⓘ Important
>
> `ReadSingleMessageAsync`는 `PipeReader.CompleteAsync`를 호출하지 않습니다. 호출자는 `PipeReader`을(를) 완료할 책임이 있습니다. 내부 `PipeReader.CompleteAsync` 호출 `ReadSingleMessageAsync` 은 더 이상 데이터를 읽을 수 없다는 신호를 보내 후속 메시지를 읽지 못하게 합니다.

여러 메시지 읽기

이 코드는 `PipeReader`에서 모든 메시지를 읽어 각 메시지에 대해 `ProcessMessageAsync`을(를) 호출합니다.

C#

```
async Task ProcessMessagesAsync(PipeReader reader, CancellationToken
cancellationToken = default)
{
    try
    {
        while (true)
        {
            ReadResult result = await reader.ReadAsync(cancellationToken);
            ReadOnlySequence<byte> buffer = result.Buffer;

            try
            {
                // Process all messages from the buffer, modifying the input buffer
on each
                // iteration.
                while (TryParseLines(ref buffer, out Message message))
                {
                    await ProcessMessageAsync(message);
                }

                // There's no more data to be processed.
                if (result.IsCompleted)
                {
                    if (buffer.Length > 0)
                    {
                        // The message is incomplete and there's no more data to
process.
                        throw new InvalidDataException("Incomplete message.");
                    }
                    break;
                }
            }
            finally
            {
                // Since all messages in the buffer are being processed, you can
```

```
use the
                // remaining buffer's Start and End position to determine consumed
and examined.
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
    finally
    {
        await reader.CompleteAsync();
    }
}
```

`ProcessMessagesAsync` 전체 메시지 읽기 루프를 소유하므로 완료되면 호출됩니다 PipeReader.CompleteAsync. 단일 메시지 사례와 달리 호출자는 판독기를 완료할 필요가 없습니다. `ProcessMessagesAsync`는 `PipeReader`의 수명에 대한 전체 소유권을 가집니다.

취소

:

- CancellationToken 전달을 지원합니다.
- 읽기 보류 중에 OperationCanceledException이 취소되는 경우 `CancellationToken`이 throw 됩니다.
- PipeReader.CancelPendingRead을 통해 현재 읽기 작업을 취소하는 방법을 지원하여 예외 증가를 방지합니다. `PipeReader.CancelPendingRead`를 호출하면 PipeReader.ReadAsync에 대한 현재 또는 다음 호출이 ReadResult가 `IsCanceled`로 설정된 `true`를 반환합니다. 이는 비 파괴적이고 예외적이지 않은 방식으로 기존 읽기 루프를 중지하는 데 유용합니다.

C#

```
private PipeReader reader;

public MyConnection(PipeReader reader)
{
    this.reader = reader;
}

public void Abort()
{
    // Cancel the pending read so the process loop ends without an exception.
    reader.CancelPendingRead();
}

public async Task ProcessMessagesAsync()
{
    try
    {
```

```
        while (true)
        {
            ReadResult result = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = result.Buffer;

            try
            {
                if (result.IsCanceled)
                {
                    // The read was canceled. You can quit without reading the
existing data.
                    break;
                }

                // Process all messages from the buffer, modifying the input buffer
on each
                // iteration.
                while (TryParseLines(ref buffer, out Message message))
                {
                    await ProcessMessageAsync(message);
                }

                // There's no more data to be processed.
                if (result.IsCompleted)
                {
                    break;
                }
            }
            finally
            {
                // Since all messages in the buffer are being processed, you can
use the
                // remaining buffer's Start and End position to determine consumed
and examined.
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
    finally
    {
        await reader.CompleteAsync();
    }
}
```

PipeReader의 일반적인 문제

- 잘못된 값을 `consumed` 또는 `examined`에 전달하면 이미 읽은 데이터를 다시 읽게 될 수 있습니다.
- `buffer.End`를 검사된 대로 전달하면 다음과 같은 결과가 발생할 수 있습니다.
  - 중단된 데이터

    - 데이터가 소비되지 않으면 궁극적으로 OOM(메모리 부족) 예외가 발생합니다. (예: 버퍼에서 한 번에 하나의 메시지를 처리하는 경우 `PipeReader.AdvanceTo(position, buffer.End)`)

- 잘못된 값을 `consumed` 전달하면 `examined` 무한 루프가 발생할 수 있습니다. 예를 들어 `buffer.Start`가 변경되지 않으면, PipeReader.ReadAsync에 대한 다음 호출이 새 데이터가 도착하기 전에 즉시 반환됩니다.
- `consumed` 또는 `examined`에 잘못된 값을 전달하면 무한 버퍼링이 발생할 수 있으며, 이는 결국 OOM(Out of Memory)으로 이어질 수 있습니다.
- 호출 PipeReader.AdvanceTo 후 ReadOnlySequence<T>를 사용하면 메모리가 손상될 수 있습니다(해제 후 사용).
- 호출 Complete/CompleteAsync 에 실패하면 메모리 누수가 발생할 수 있습니다.
- 버퍼를 처리하기 전에 ReadResult.IsCompleted를 확인하고 읽기 논리를 종료하면 데이터 손실이 발생합니다. 루프 종료 조건은 `ReadResult.Buffer.IsEmpty` 및 `ReadResult.IsCompleted`를 기반으로 해야 합니다. 잘못하면 무한 루프가 발생할 수 있습니다.

문제 코드

❌ 데이터 손실

`ReadResult`가 `IsCompleted`로 설정된 경우 `true`는 최종 데이터 세그먼트를 반환할 수 있습니다. 읽기 루프를 종료하기 전에 해당 데이터를 읽지 않으면 데이터가 손실됩니다.

> ⚠ **Warning**
>
> 다음 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 다음 샘플은 **PipeReader의 일반적인 문제를** 설명하기 위해 제공됩니다.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> dataLossBuffer = result.Buffer;

    if (result.IsCompleted)
        break;
```

```
    Process(ref dataLossBuffer, out Message message);

    reader.AdvanceTo(dataLossBuffer.Start, dataLossBuffer.End);
}
```

> ⚠ Warning
>
> 앞의 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 이전 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

❌ 무한 루프

`Result.IsCompleted`이(가) `true`이지만 버퍼에 완전한 메시지가 없는 경우, 다음 논리는 무한 루프를 초래할 수 있습니다.

> ⚠ Warning
>
> 다음 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 다음 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> infiniteLoopBuffer = result.Buffer;
    if (result.IsCompleted && infiniteLoopBuffer.IsEmpty)
        break;

    Process(ref infiniteLoopBuffer, out Message message);

    reader.AdvanceTo(infiniteLoopBuffer.Start, infiniteLoopBuffer.End);
}
```

> ⚠ Warning
>
> 앞의 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 이전 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

동일한 문제가 있는 또 다른 코드는 다음과 같습니다. `ReadResult.IsCompleted`를 확인하기 전에 비어 있지 않은 버퍼를 확인하는 것입니다. 버퍼 내에 `else if`가 포함되어 있다면, 버퍼에 완전한 메시지가 없을 경우 무한히 반복됩니다.

> ⚠ Warning
>
> 다음 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 다음 샘플은 **PipeReader의 일반적인 문제를** 설명하기 위해 제공됩니다.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> infiniteLoopBuffer = result.Buffer;

    if (!infiniteLoopBuffer.IsEmpty)
        Process(ref infiniteLoopBuffer, out Message message);

    else if (result.IsCompleted)
        break;

    reader.AdvanceTo(infiniteLoopBuffer.Start, infiniteLoopBuffer.End);
}
```

> ⚠ Warning
>
> 앞의 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 이전 샘플은 **PipeReader의 일반적인 문제를** 설명하기 위해 제공됩니다.

❌ 응답하지 않는 애플리케이션

`examined` 위치에서 PipeReader.AdvanceTo`buffer.End`을(를) 무조건 호출할 경우, 단일 메시지를 구문 분석할 때 애플리케이션이 응답하지 않을 가능성이 있습니다. PipeReader.AdvanceTo에 대한 다음 호출은 다음까지 반환되지 않습니다.

- 파이프에 기록된 데이터가 더 있음
- 새 데이터는 이전에 검사되지 않았음

> ⚠ Warning

다음 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 다음 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> hangBuffer = result.Buffer;

    Process(ref hangBuffer, out Message message);

    if (result.IsCompleted)
        break;

    reader.AdvanceTo(hangBuffer.Start, hangBuffer.End);

    if (message != null)
        return message;
}
```

> ⚠ Warning
>
> 앞의 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 이전 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

❌ OOM(메모리 부족)

특정 조건이 충족될 때까지 이 코드는 OutOfMemoryException가 발생할 때까지 계속 버퍼링을 유지합니다.

- 최대 메시지 크기가 없습니다.
- `PipeReader`에서 반환된 데이터는 완전한 메시지를 생성하지 않습니다. 예를 들어 다른 쪽에서는 큰 메시지(예: 4GB 메시지)를 작성합니다.

> ⚠ Warning
>
> 다음 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 다음 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> thisCouldOutOfMemory = result.Buffer;

    Process(ref thisCouldOutOfMemory, out Message message);

    if (result.IsCompleted)
        break;

    reader.AdvanceTo(thisCouldOutOfMemory.Start, thisCouldOutOfMemory.End);

    if (message != null)
        return message;
}
```

> ⚠ Warning
>
> 앞의 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 이전 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

❌ 메모리 손상

버퍼를 읽는 도우미를 작성할 때 호출 Advance하기 전에 반환된 페이로드를 복사합니다. 다음 예제에서는 삭제된 메모리를 `Pipe` 반환하고 다음 작업(읽기/쓰기)에 다시 사용할 수 있습니다.

> ⚠ Warning
>
> 다음 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 다음 샘플은 PipeReader의 일반적인 문제를 설명하기 위해 제공됩니다.

C#

```
public class Message
{
    public ReadOnlySequence<byte> CorruptedPayload { get; set; }
}
```

C#

```
    Environment.FailFast("This code is terrible, don't use it!");
    Message message = null;

    while (true)
    {
        ReadResult result = await reader.ReadAsync(cancellationToken);
        ReadOnlySequence<byte> buffer = result.Buffer;

        ReadHeader(ref buffer, out int length);

        if (length <= buffer.Length)
        {
            message = new Message
            {
                // Slice the payload from the existing buffer
                CorruptedPayload = buffer.Slice(0, length)
            };

            buffer = buffer.Slice(length);
        }

        if (result.IsCompleted)
            break;

        reader.AdvanceTo(buffer.Start, buffer.End);

        if (message != null)
        {
            // This code is broken since reader.AdvanceTo() was called with a
position *after* the buffer
            // was captured.
            break;
        }
    }

    return message;
}
```

> ⚠ **Warning**
>
> 앞의 코드를 사용하지 **마세요**. 이 샘플을 사용하면 데이터가 손실되거나 중단되고 보안 문제가 발생하며, 복사되지 **않습니다**. 이전 샘플은 **PipeReader의 일반적인 문제를** 설명하기 위해 제공됩니다.

PipeWriter

PipeWriter는 호출자를 대신해 쓰기 위한 버퍼를 관리합니다. `PipeWriter`는 IBufferWriter<byte>를 구현합니다. `IBufferWriter<byte>` 는 추가 버퍼 복사본 없이 쓰기를 수행

하기 위해 버퍼에 대한 액세스를 제공합니다.

C#

```
async Task WriteHelloAsync(PipeWriter writer, CancellationToken cancellationToken =
default)
{
    // Request at least 5 bytes from the PipeWriter.
    Memory<byte> memory = writer.GetMemory(5);

    // Write directly into the buffer.
    int written = Encoding.ASCII.GetBytes("Hello".AsSpan(), memory.Span);

    // Tell the writer how many bytes were written.
    writer.Advance(written);

    await writer.FlushAsync(cancellationToken);
}
```

이전 코드는 다음과 같습니다.

- `PipeWriter`을 사용하여 GetMemory에서 최소 5바이트의 버퍼를 요청합니다.
- 반환된 `"Hello"`에 ASCII 문자열 `Memory<byte>`를 위한 바이트를 씁니다.
- Advance를 호출하여 버퍼에 쓴 바이트 수를 표시합니다.
- 기본 디바이스로 바이트를 전송하는 `PipeWriter`를 플러시합니다.

이전 작성 메서드에서는 `PipeWriter`에서 제공하는 버퍼를 사용합니다. 다음을 사용할 수도 있습니다.PipeWriter.WriteAsync

- 기존 버퍼를 `PipeWriter`에 복사합니다.
- GetSpan, Advance를 적절히 호출하고, FlushAsync를 호출합니다.

C#

```
async Task WriteHelloAsync(PipeWriter writer, CancellationToken cancellationToken =
default)
{
    byte[] helloBytes = Encoding.ASCII.GetBytes("Hello");

    // Write helloBytes to the writer, there's no need to call Advance here
    // (Write does that).
    await writer.WriteAsync(helloBytes, cancellationToken);
}
```

취소

FlushAsync에서는 CancellationToken 전달을 지원합니다. `CancellationToken`을 전달하면 플러시가 보류 중인 동안 토큰이 취소되는 경우 `OperationCanceledException`을 반환됩니다. PipeWriter.FlushAsync는 예외를 발생시키지 않고 PipeWriter.CancelPendingFlush을 통해 현재 플러시 작업을 취소하는 메서드를 지원합니다. `PipeWriter.CancelPendingFlush`를 호출하면, PipeWriter.FlushAsync 또는 `PipeWriter.WriteAsync`에 대한 현재 또는 다음 호출은 FlushResult를 `IsCanceled`로 설정한 `true`를 반환합니다. 이는 비 파괴적이고 예외적이지 않은 방식으로 생성 플러시를 중지하는 데 유용합니다.

PipeWriter의 일반적인 문제

- GetSpan 및 GetMemory는 적어도 요청된 양의 메모리를 포함하는 버퍼를 반환합니다. 정확한 버퍼 크기를 가정하지 **마세요**.
- 연속 호출은 동일한 버퍼 또는 동일한 크기의 버퍼를 반환하도록 보장되지 않습니다.
- 추가 데이터를 계속 작성하려면 Advance를 호출한 후 새 버퍼를 요청해야 합니다. 이전에 획득한 버퍼에 쓸 수 없습니다.
- GetMemory에 대한 불완전한 호출이 있는 상태에서 GetSpan 또는 FlushAsync을 호출하는 것은 안전하지 않습니다.
- Complete 또는 CompleteAsync를 호출할 경우, 데이터가 플러시되지 않은 상태에서 메모리가 손상될 수 있습니다.

PipeReader 및 PipeWriter에 대한 팁

다음 팁을 사용하여 클래스를 System.IO.Pipelines 성공적으로 사용합니다.

- 해당하는 경우 예외를 포함하여 항상 PipeReader 및 PipeWriter를 완료합니다.
- 항상 PipeReader.AdvanceTo를 호출한 후 PipeReader.ReadAsync를 호출합니다.
- 정기적으로 `await` PipeWriter.FlushAsync 글을 쓰는 동안 항상 FlushResult.IsCompleted를 확인합니다. 읽기 프로그램이 완료되었으며 더 이상 작성된 내용에 대해 신경 쓰지 않음을 나타내기 때문에 `IsCompleted`가 `true`이면 쓰기 중단합니다.
- PipeWriter.FlushAsync에 액세스 권한을 부여할 내용을 작성한 후, `PipeReader`을(를) 호출하십시오.
- `FlushAsync`가 완료되기 전까지는 판독기가 시작할 수 없으므로 `FlushAsync`를 호출하지 마세요. 이는 교착 상태를 유발할 수 있습니다.
- 하나의 컨텍스트만 `PipeReader` 또는 `PipeWriter`를 "소유"하거나 액세스하는지 확인합니다. 이러한 형식은 스레드로부터 안전하지 않습니다.
- ReadResult.Buffer를 호출하거나 PipeReader.AdvanceTo을 완료한 후에는 `PipeReader`에 액세스하지 않습니다.

IDuplexPipe

IDuplexPipe 는 읽기 및 쓰기를 모두 지원하는 형식에 대한 계약입니다. 예를 들어 네트워크 연결은 .로 `IDuplexPipe`표시됩니다.

`Pipe` 및 `PipeReader`를 포함하는 `PipeWriter`와 달리 `IDuplexPipe`는 전이중 연결의 단일 측면을 나타냅니다. `PipeWriter`에 쓰는 내용은 `PipeReader`에서 읽히지 않습니다.

스트림

스트림 데이터를 읽거나 쓸 때는 일반적으로 역직렬 변환기를 사용하여 데이터를 읽고 직렬 변환기를 사용하여 데이터를 씁니다. 대부분의 읽기 및 쓰기 스트림 API에는 `Stream` 매개 변수가 있습니다. 기존 API와 더욱 쉽게 통합할 수 있도록 `PipeReader` 및 `PipeWriter`는 AsStream 메서드를 제공합니다. AsStream은 `Stream` 또는 `PipeReader` 주위에 `PipeWriter` 구현을 반환합니다.

스트림 예제

정적 `Create` 메서드를 사용하여 주어진 Stream 객체 및 선택적 생성 옵션으로 `PipeReader` 및 `PipeWriter` 인스턴스를 생성합니다.

StreamPipeReaderOptions를 사용하면 다음 매개 변수로 `PipeReader` 인스턴스 생성을 제어할 수 있습니다.

- StreamPipeReaderOptions.BufferSize은 풀에서 메모리를 대여할 때 사용되는 최소 버퍼 크기(바이트)이며, 기본값은 `4096`입니다.
- StreamPipeReaderOptions.LeaveOpen 플래그는 `PipeReader`가 완료된 후 기본 스트림을 열어 둘지 여부를 결정하며, 기본값은 `false`입니다.
- StreamPipeReaderOptions.MinimumReadSize은 새 버퍼가 할당되기 전에 버퍼에 남은 바이트의 임계값을 나타내며, 기본값은 `1024`입니다.
- StreamPipeReaderOptions.Pool은 메모리를 할당할 때 사용되는 `MemoryPool<byte>`이며, 기본값은 `null`입니다.

StreamPipeWriterOptions를 사용하면 다음 매개 변수로 `PipeWriter` 인스턴스 생성을 제어할 수 있습니다.

- StreamPipeWriterOptions.LeaveOpen 플래그는 `PipeWriter`가 완료된 후 기본 스트림을 열어 둘지 여부를 결정하며, 기본값은 `false`입니다.
- StreamPipeWriterOptions.MinimumBufferSize은 Pool에서 메모리를 대여할 때 사용할 최소 버퍼 크기를 나타내며, 기본값은 `4096`입니다.
- StreamPipeWriterOptions.Pool은 메모리를 할당할 때 사용되는 `MemoryPool<byte>`이며, 기본값은 `null`입니다.

> ⓘ **Important**
>
> `PipeReader` 및 `PipeWriter` 인스턴스를 `Create` 메서드를 사용하여 생성할 때 `Stream` 개체 수명을 고려합니다. 판독기 또는 기록기가 스트림 사용을 마친 후에도 스트림에 액세스해야 하는 경우, 생성 옵션에서 `LeaveOpen` 플래그를 `true`로 설정합니다. 그렇지 않으면 스트림이 닫힙니다.

이 코드는 스트림에서 `Create` 메서드를 사용하여 `PipeReader` 인스턴스와 `PipeWriter` 인스턴스를 생성하는 방법을 보여줍니다.

C#

```
using System.Buffers;
using System.IO.Pipelines;
using System.Text;

class Program
{
    static async Task Main()
    {
        using var stream = File.OpenRead("lorem-ipsum.txt");

        var reader = PipeReader.Create(stream);
        var writer = PipeWriter.Create(
            Console.OpenStandardOutput(),
            new StreamPipeWriterOptions(leaveOpen: true));

        WriteUserCancellationPrompt();

        var processMessagesTask = ProcessMessagesAsync(reader, writer);
        var userCanceled = false;
        var cancelProcessingTask = Task.Run(() =>
        {
            while (char.ToUpperInvariant(Console.ReadKey().KeyChar) != 'C')
            {
                WriteUserCancellationPrompt();
            }

            userCanceled = true;

            // No exceptions thrown
            reader.CancelPendingRead();
            writer.CancelPendingFlush();
        });

        await Task.WhenAny(cancelProcessingTask, processMessagesTask);

        Console.WriteLine(
            $"\n\nProcessing {(userCanceled ? "cancelled" : "completed")}.\n");
    }
```

```
static void WriteUserCancellationPrompt() =>
    Console.WriteLine("Press 'C' to cancel processing...\n");

static async Task ProcessMessagesAsync(
    PipeReader reader,
    PipeWriter writer)
{
    try
    {
        while (true)
        {
            ReadResult readResult = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = readResult.Buffer;

            try
            {
                if (readResult.IsCanceled)
                {
                    break;
                }

                if (TryParseLines(ref buffer, out string message))
                {
                    FlushResult flushResult =
                        await WriteMessagesAsync(writer, message);

                    if (flushResult.IsCanceled || flushResult.IsCompleted)
                    {
                        break;
                    }
                }

                if (readResult.IsCompleted)
                {
                    if (!buffer.IsEmpty)
                    {
                        throw new InvalidDataException("Incomplete message.");
                    }
                    break;
                }
            }
            finally
            {
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
    catch (Exception ex)
    {
        Console.Error.WriteLine(ex);
    }
    finally
    {
        await reader.CompleteAsync();
```

```
            await writer.CompleteAsync();
        }
    }

    static bool TryParseLines(
        ref ReadOnlySequence<byte> buffer,
        out string message)
    {
        SequencePosition? position;
        StringBuilder outputMessage = new();

        while(true)
        {
            position = buffer.PositionOf((byte)'\n');

            if (!position.HasValue)
                break;


outputMessage.Append(Encoding.ASCII.GetString(buffer.Slice(buffer.Start,
position.Value)))
                         .AppendLine();

            buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
        };

        message = outputMessage.ToString();
        return message.Length != 0;
    }

    static ValueTask<FlushResult> WriteMessagesAsync(
        PipeWriter writer,
        string message) =>
        writer.WriteAsync(Encoding.ASCII.GetBytes(message));
}
```

애플리케이션은 StreamReader를 사용하여 *lorem-ipsum.txt* 파일을 스트림으로 읽으며 빈 줄로 끝나야 합니다. FileStream이 PipeReader.Create 개체를 인스턴스화하는 `PipeReader`에 전달됩니다. 그런 다음, 콘솔 애플리케이션이 PipeWriter.Create을 사용하여 표준 출력 스트림을 Console.OpenStandardOutput()에 전달합니다. 이 예제에서는 취소를 지원합니다.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 31.

.NET에서 버퍼 작업

2025. 06. 17.

이 문서에서는 여러 버퍼에서 실행되는 데이터를 읽는 데 도움이 되는 형식의 개요를 제공합니다. 주로 PipeReader 개체를 지원하는 데 사용됩니다.

IBufferWriter<T>

System.Buffers.IBufferWriter<T> 는 동기 버퍼링된 쓰기에 대한 계약입니다. 가장 낮은 수준에서 인터페이스는 다음과 같습니다.

- 기본이며 사용하기 어렵지 않습니다.
- Memory<T> 또는 Span<T>에 대한 액세스를 허용합니다. `Memory<T>` 또는 `Span<T>`에 기록할 수 있으며 `T` 항목이 얼마나 기록되었는지 확인할 수 있습니다.

C#

```
void WriteHello(IBufferWriter<byte> writer)
{
    // Request at least 5 bytes.
    Span<byte> span = writer.GetSpan(5);
    ReadOnlySpan<char> helloSpan = "Hello".AsSpan();
    int written = Encoding.ASCII.GetBytes(helloSpan, span);

    // Tell the writer how many bytes were written.
    writer.Advance(written);
}
```

앞의 메서드는 다음과 같습니다.

- `IBufferWriter<byte>`을 사용하여 `GetSpan(5)`에서 최소 5바이트의 버퍼를 요청합니다.
- ASCII 문자열 "Hello"에 대한 바이트를 반환된 `Span<byte>` 값에 씁니다.
- IBufferWriter<T>를 호출하여 버퍼에 쓴 바이트 수를 표시합니다.

이 작성 방법은 `Memory<T>`에서 제공하는 /`Span<T>` `IBufferWriter<T>` 버퍼를 사용합니다. 대안으로 Write 확장 메서드를 사용하여 기존 버퍼를 `IBufferWriter<T>`에 복사할 수 있습니다. `Write`은 적절한 호출 `GetSpan`/`Advance` 작업을 수행하므로 다음을 작성한 후에는 호출 `Advance` 할 필요가 없습니다.

C#

```
void WriteHello(IBufferWriter<byte> writer)
{
    byte[] helloBytes = Encoding.ASCII.GetBytes("Hello");
```

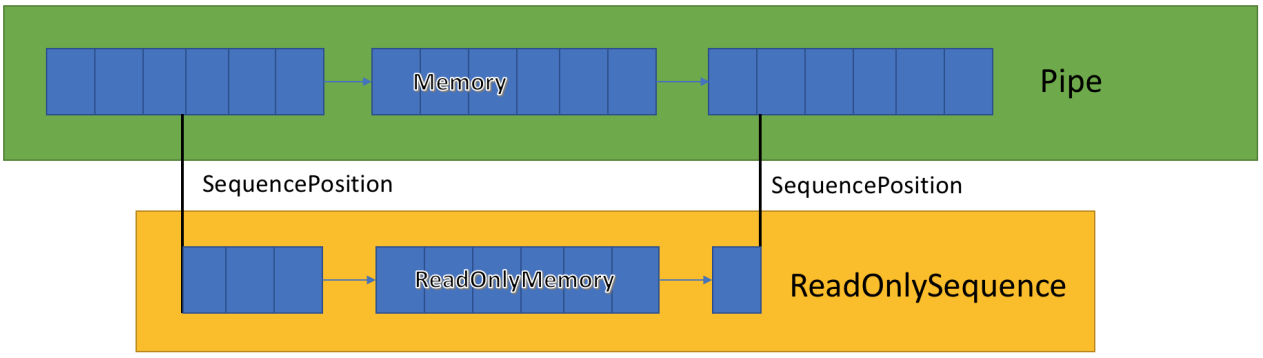
```
    // Write helloBytes to the writer. There's no need to call Advance here
    // since Write calls Advance.
    writer.Write(helloBytes);
}
```

ArrayBufferWriter<T> 는 백업 저장소가 단일 연속 배열인 구현 `IBufferWriter<T>` 입니다.

IBufferWriter 일반적인 문제

- `GetSpan` 및 `GetMemory`는 적어도 요청된 양의 메모리를 포함하는 버퍼를 반환합니다. 정확한 버퍼 크기를 가정하지 마세요.
- 연속 호출이 동일한 버퍼 또는 동일한 크기의 버퍼를 반환한다는 보장은 없습니다.
- 추가 데이터를 계속 작성하려면 `Advance`를 호출한 후 새 버퍼를 요청해야 합니다. 이전에 획득한 버퍼는 호출된 후에 `Advance` 쓸 수 없습니다.

ReadOnlySequence<T>



ReadOnlySequence<T> 는 연속 또는 연속되지 않은 시퀀스를 나타낼 수 있는 구조체입니다 `T`. 다음에서 구성할 수 있습니다.

1. `T[]`
2. `ReadOnlyMemory<T>`
3. 시퀀스의 시작 및 끝 위치를 나타내는 연결된 목록 노드 ReadOnlySequenceSegment<T> 및 인덱스 쌍입니다.

세 번째 표현은 다음과 같은 다양한 작업에 성능에 영향을 주기 때문에 가장 흥미로운 표현입니다.`ReadOnlySequence<T>`

테이블 확장

표현	수술	복잡성
`T[]`/`ReadOnlyMemory<T>`	`Length`	`O(1)`
`T[]`/`ReadOnlyMemory<T>`	`GetPosition(long)`	`O(1)`
`T[]`/`ReadOnlyMemory<T>`	`Slice(int, int)`	`O(1)`
`T[]`/`ReadOnlyMemory<T>`	`Slice(SequencePosition, SequencePosition)`	`O(1)`
`ReadOnlySequenceSegment<T>`	`Length`	`O(1)`
`ReadOnlySequenceSegment<T>`	`GetPosition(long)`	`O(number of segments)`
`ReadOnlySequenceSegment<T>`	`Slice(int, int)`	`O(number of segments)`
`ReadOnlySequenceSegment<T>`	`Slice(SequencePosition, SequencePosition)`	`O(1)`

이 혼합 표현으로 인해 `ReadOnlySequence<T>`는 인덱스를 정수가 아닌 `SequencePosition`로 표시합니다. A `SequencePosition`:

- 출처의 `ReadOnlySequence<T>`에서 기원한 인덱스를 나타내는 불투명 값입니다.
- 정수와 개체의 두 부분으로 구성됩니다. 이 두 값이 나타내는 것은 .의 `ReadOnlySequence<T>`구현과 연결됩니다.

데이터 액세스

`ReadOnlySequence<T>`는 데이터를 `ReadOnlyMemory<T>`의 열거 가능한 형태로 제공합니다. 기본 foreach를 사용하여 각 세그먼트를 열거할 수 있습니다.

C#

```
long FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    long position = 0;

    foreach (ReadOnlyMemory<byte> segment in buffer)
    {
        ReadOnlySpan<byte> span = segment.Span;
        var index = span.IndexOf(data);
        if (index != -1)
        {
            return position + index;
        }

        position += span.Length;
    }
```

```
    return -1;
}
```

앞의 메서드는 각 세그먼트에서 특정 바이트를 검색합니다. 각 세그먼트 `SequencePosition` ReadOnlySequence<T>.TryGet 를 추적해야 하는 경우 더 적합합니다. 다음 샘플에서는 정수 대신 반환 `SequencePosition` 하도록 이전 코드를 변경합니다. `SequencePosition` 를 반환하면 호출자가 구체적인 인덱스에서 데이터를 얻기 위한 두 번째 검사를 방지할 수 있는 이점을 제공합니다.

C#

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    SequencePosition position = buffer.Start;
    SequencePosition result = position;

    while (buffer.TryGet(ref position, out ReadOnlyMemory<byte> segment))
    {
        ReadOnlySpan<byte> span = segment.Span;
        var index = span.IndexOf(data);
        if (index != -1)
        {
            return buffer.GetPosition(index, result);
        }

        result = position;
    }
    return null;
}
```

`SequencePosition` 조합 및 `TryGet` 열거자처럼 작동합니다. 위치 필드는 각 반복의 시작 부분에서 수정되어 `ReadOnlySequence<T>` 내 각 세그먼트의 시작점이 됩니다.

위의 메서드는 .에 확장 메서드로 존재합니다 `ReadOnlySequence<T>`. PositionOf 를 사용하여 이전 코드를 간소화할 수 있습니다.

C#

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data) =>
buffer.PositionOf(data);
```

ReadOnlySequence<T>를 처리합니다

데이터를 시퀀스 내의 `ReadOnlySequence<T>` 여러 세그먼트로 분할할 수 있으므로 처리가 어려울 수 있습니다. 최상의 성능을 위해 코드를 두 개의 경로로 분할합니다.

- 단일 세그먼트 사례를 다루는 빠른 경로입니다.
- 세그먼트 간에 분할된 데이터를 처리하는 느린 경로입니다.

다중 분할 시퀀스에서 데이터를 처리하는 데 사용할 수 있는 몇 가지 방법이 있습니다.

- SelectionMode를 사용합니다.
- 세그먼트별로 데이터를 구문 분석하고, 구문 분석된 세그먼트 내의 `SequencePosition`와 그 인덱스를 추적합니다. 이렇게 하면 불필요한 할당을 방지할 수 있지만 특히 작은 버퍼의 경우 비효율적일 수 있습니다.
- `ReadOnlySequence<T>` 연속 배열에 복사하고 단일 버퍼처럼 처리합니다.
  - 크기 `ReadOnlySequence<T>` 가 작은 경우 stackalloc 연산자를 사용하여 데이터를 스택 할당 버퍼에 복사하는 것이 합리적일 수 있습니다.
  - `ReadOnlySequence<T>`를 ArrayPool<T>.Shared을/를 사용하여 풀된 배열에 복사합니다.
  - ReadOnlySequence<T>.ToArray()을 사용합니다. 힙에 새 `T[]`를 할당하게 되므로 부하가 많은 경로에서는 권장되지 않습니다.

다음 예제에서는 처리 `ReadOnlySequence<byte>`에 대한 몇 가지 일반적인 사례를 보여 줍니다.

이진 데이터 처리

다음 예제에서는 `ReadOnlySequence<byte>` 시작부터 4 바이트 big-endian 정수 길이를 구문 분석합니다.

C#

```
bool TryParseHeaderLength(ref ReadOnlySequence<byte> buffer, out int length)
{
    // If there's not enough space, the length can't be obtained.
    if (buffer.Length < 4)
    {
        length = 0;
        return false;
    }

    // Grab the first 4 bytes of the buffer.
    var lengthSlice = buffer.Slice(buffer.Start, 4);
    if (lengthSlice.IsSingleSegment)
    {
        // Fast path since it's a single segment.
        length = BinaryPrimitives.ReadInt32BigEndian(lengthSlice.First.Span);
    }
    else
    {
        // There are 4 bytes split across multiple segments. Since it's so small,
it
        // can be copied to a stack allocated buffer. This avoids a heap
allocation.
        Span<byte> stackBuffer = stackalloc byte[4];
```

```
        lengthSlice.CopyTo(stackBuffer);
        length = BinaryPrimitives.ReadInt32BigEndian(stackBuffer);
    }

    // Move the buffer 4 bytes ahead.
    buffer = buffer.Slice(lengthSlice.End);

    return true;
}
```

텍스트 데이터 처리

다음 예제는 다음과 같습니다.

- `\r\n`에서 첫 번째 줄 바꿈(`ReadOnlySequence<byte>`)을 찾아 'line'이라는 out 매개변수를 통해 반환합니다.
- 입력 버퍼에서 `\r\n`를 제외하고 해당 줄을 트리밍합니다.

C#

```
static bool TryParseLine(ref ReadOnlySequence<byte> buffer, out
ReadOnlySequence<byte> line)
{
    SequencePosition position = buffer.Start;
    SequencePosition previous = position;
    var index = -1;
    line = default;

    while (buffer.TryGet(ref position, out ReadOnlyMemory<byte> segment))
    {
        ReadOnlySpan<byte> span = segment.Span;

        // Look for \r in the current segment.
        index = span.IndexOf((byte)'\r');

        if (index != -1)
        {
            // Check next segment for \n.
            if (index + 1 >= span.Length)
            {
                var next = position;
                if (!buffer.TryGet(ref next, out ReadOnlyMemory<byte>
nextSegment))
                {
                    // You're at the end of the sequence.
                    return false;
                }
                else if (nextSegment.Span[0] == (byte)'\n')
                {
                    //  A match was found.
                    break;
```

```
                }
            }
            // Check the current segment of \n.
            else if (span[index + 1] == (byte)'\n')
            {
                // It was found.
                break;
            }
        }

        previous = position;
    }

    if (index != -1)
    {
        // Get the position just before the \r\n.
        var delimeter = buffer.GetPosition(index, previous);

        // Slice the line (excluding \r\n).
        line = buffer.Slice(buffer.Start, delimeter);

        // Slice the buffer to get the remaining data after the line.
        buffer = buffer.Slice(buffer.GetPosition(2, delimeter));
        return true;
    }

    return false;
}
```

빈 세그먼트

`ReadOnlySequence<T>` 내부에 빈 세그먼트를 저장하는 것이 유효합니다. 세그먼트를 명시적으로 열거하는 동안 빈 세그먼트가 발생할 수 있습니다.

C#

```
static void EmptySegments()
{
    // This logic creates a ReadOnlySequence<byte> with 4 segments,
    // two of which are empty.
    var first = new BufferSegment(new byte[0]);
    var last = first.Append(new byte[] { 97 })
                    .Append(new byte[0]).Append(new byte[] { 98 });

    // Construct the ReadOnlySequence<byte> from the linked list segments.
    var data = new ReadOnlySequence<byte>(first, 0, last, 1);

    // Slice using numbers.
    var sequence1 = data.Slice(0, 2);

    // Slice using SequencePosition pointing at the empty segment.
    var sequence2 = data.Slice(data.Start, 2);
```

```
    Console.WriteLine($"sequence1.Length={sequence1.Length}"); //
sequence1.Length=2
    Console.WriteLine($"sequence2.Length={sequence2.Length}"); //
sequence2.Length=2

    // sequence1.FirstSpan.Length=1
    Console.WriteLine($"sequence1.FirstSpan.Length={sequence1.FirstSpan.Length}");

    // Slicing using SequencePosition will Slice the ReadOnlySequence<byte>
directly
    // on the empty segment!
    // sequence2.FirstSpan.Length=0
    Console.WriteLine($"sequence2.FirstSpan.Length={sequence2.FirstSpan.Length}");

    // The following code prints 0, 1, 0, 1.
    SequencePosition position = data.Start;
    while (data.TryGet(ref position, out ReadOnlyMemory<byte> memory))
    {
        Console.WriteLine(memory.Length);
    }
}

class BufferSegment : ReadOnlySequenceSegment<byte>
{
    public BufferSegment(Memory<byte> memory)
    {
        Memory = memory;
    }

    public BufferSegment Append(Memory<byte> memory)
    {
        var segment = new BufferSegment(memory)
        {
            RunningIndex = RunningIndex + Memory.Length
        };
        Next = segment;
        return segment;
    }
}
```

앞의 코드는 빈 세그먼트가 있는 `ReadOnlySequence<byte>` 세그먼트를 만들고 빈 세그먼트가 다양한 API에 미치는 영향을 보여 줍니다.

- `ReadOnlySequence<T>.Slice` 빈 세그먼트를 `SequencePosition` 가리키는 경우 해당 세그먼트가 유지됩니다.
- `ReadOnlySequence<T>.Slice` int가 있는 경우 빈 세그먼트를 건너뜁니다.
- 열거형을 사용하면 `ReadOnlySequence<T>` 빈 세그먼트를 열거합니다.

ReadOnlySequence<T> 및 SequencePosition의 잠재적 문제

`ReadOnlySequence<T>` / `SequencePosition`와 일반 `ReadOnlySpan<T>`/`ReadOnlyMemory<T>`/`T[]`/`int`을 비교할 때 몇 가지 비정상적인 결과가 있습니다.

- `SequencePosition` 는 절대 위치가 아닌 특정 `ReadOnlySequence<T>`위치에 대한 위치 표식입니다. 특정 `ReadOnlySequence<T>`에 상대적이므로 원래 위치 외부에서 `ReadOnlySequence<T>` 사용하는 경우 의미가 없습니다.
- 산술 계산은 `SequencePosition` 없이는 `ReadOnlySequence<T>`에서 수행할 수 없습니다. 즉, 기본적인 것들을 `position++`와 같이 수행하는 것은 `position = ReadOnlySequence<T>.GetPosition(1, position)`로 작성된다는 의미입니다.
- `GetPosition(long)` 는 음수 인덱스를 지원하지 **않습니다** . 즉, 모든 세그먼트를 걷지 않고는 두 번째 문자에서 마지막 문자를 얻는 것은 불가능합니다.
- 두 `SequencePosition`을(를) 비교할 수 없어 어려움이 생깁니다.
  - 한 위치가 다른 위치보다 크거나 작은지 여부를 알 수 있습니다.
  - 몇 가지 구문 분석 알고리즘을 작성합니다.
- `ReadOnlySequence<T>`가 개체 참조보다 크며 가능한 경우 in 또는 ref에 의해 전달되어야 합니다. `ReadOnlySequence<T>`을 `in` 또는 `ref` 통해 전달하면 구조체의 복사본이 줄어듭니다.
- 빈 세그먼트:
  - `ReadOnlySequence<T>` 내에서 유효합니다.
  - `ReadOnlySequence<T>.TryGet` 메서드를 사용하여 반복할 때 나타날 수 있습니다.
  - `ReadOnlySequence<T>.Slice()` 방법과 `SequencePosition` 객체를 사용하여 시퀀스를 자르는 방식으로 나타낼 수 있습니다.

SequenceReader<T>

SequenceReader<T>:

- .NET Core 3.0에서 도입된 새 형식으로, `ReadOnlySequence<T>`의 처리를 간소화하는 데 사용됩니다.
- 단일 세그먼트 `ReadOnlySequence<T>`와 다중 세그먼트 `ReadOnlySequence<T>` 간의 차이점을 하나로 만듭니다.
- 세그먼트 간에 분할되거나 분할되지 않을 수 있는 이진 및 텍스트 데이터(`byte` 및 `char`)를 읽기 위한 도우미를 제공합니다.

이진 데이터와 구분된 데이터 처리를 처리하는 기본 제공 메서드가 있습니다. 다음 섹션에서는 `SequenceReader<T>`와 함께하는 동일한 메서드들이 어떻게 나타나는지 보여 줍니다.

데이터 액세스

`SequenceReader<T>` 에는 직접 내부 데이터를 열거하는 메서드가 있습니다 `ReadOnlySequence<T>` . 다음 코드는 한 번에 `ReadOnlySequence<byte>`를 `byte` 처리하는 예제입니다.

C#

```
while (reader.TryRead(out byte b))
{
    Process(b);
}
```

메서드 `CurrentSpan` 에서 수동으로 수행한 작업과 유사한 현재 세그먼트 `Span`를 노출합니다.

위치 사용

다음 코드는 `FindIndexOf`를 사용하는 예제 구현입니다`SequenceReader<T>`.

C#

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    var reader = new SequenceReader<byte>(buffer);

    while (!reader.End)
    {
        // Search for the byte in the current span.
        var index = reader.CurrentSpan.IndexOf(data);
        if (index != -1)
        {
            // It was found, so advance to the position.
            reader.Advance(index);

            return reader.Position;
        }
        // Skip the current segment since there's nothing in it.
        reader.Advance(reader.CurrentSpan.Length);
    }

    return null;
}
```

이진 데이터 처리

다음 예제에서는 `ReadOnlySequence<byte>` 시작부터 4 바이트 big-endian 정수 길이를 구문 분석합니다.

C#

```
bool TryParseHeaderLength(ref ReadOnlySequence<byte> buffer, out int length)
{
    var reader = new SequenceReader<byte>(buffer);
```

```
    return reader.TryReadBigEndian(out length);
}
```

텍스트 데이터 처리

C#

```
static ReadOnlySpan<byte> NewLine => new byte[] { (byte)'\r', (byte)'\n' };

static bool TryParseLine(ref ReadOnlySequence<byte> buffer,
                         out ReadOnlySequence<byte> line)
{
    var reader = new SequenceReader<byte>(buffer);

    if (reader.TryReadTo(out line, NewLine))
    {
        buffer = buffer.Slice(reader.Position);

        return true;
    }

    line = default;
    return false;
}
```

SequenceReader<T> 일반적인 문제

- `SequenceReader<T>` 변경 가능한 구조체이므로 항상 참조로 전달되어야 합니다.
- `SequenceReader<T>` 는 동기 메서드에서만 사용할 수 있고 필드에 저장할 수 없도록 ref 구조체 입니다. 자세한 내용은 할당 방지를 참조하세요.
- `SequenceReader<T>` 는 전달 전용 판독기로 사용하도록 최적화되어 있습니다. `Rewind`는 다른 `Read` `Peek`API 및 `IsNext` API를 활용하여 해결할 수 없는 작은 백업을 위한 것입니다.

메모리 매핑된 파일

아티클 • 2025. 04. 07.

메모리 매핑된 파일에는 가상 메모리에 있는 파일의 내용이 포함됩니다. 파일과 메모리 공간 간의 이 매핑을 사용하면 여러 프로세스를 포함한 애플리케이션이 메모리를 직접 읽고 쓰면서 파일을 수정할 수 있습니다. 관리 코드를 사용하여 Memory-Mapped 파일 관리에 설명된 대로 네이티브 Windows 함수가 메모리 매핑된 파일에 액세스하는 것과 동일한 방식으로 메모리 매핑된 파일에 액세스할 수 있습니다.

다음과 같은 두 가지 유형의 메모리 매핑 파일이 있습니다.

- 영속적 메모리 매핑 파일

  지속형 파일은 디스크의 원본 파일과 연결된 메모리 매핑 파일입니다. 마지막 프로세스가 파일 작업을 마치면 데이터가 디스크의 원본 파일에 저장됩니다. 이러한 메모리 매핑 파일은 매우 큰 소스 파일 작업에 적합합니다.

- 비지속 메모리 매핑 파일

  비지속형 파일은 디스크의 파일과 연결되지 않은 메모리 매핑 파일입니다. 마지막 프로세스가 파일 작업을 마치면 데이터가 손실되고 가비지 수집에 의해 파일이 회수됩니다. 이러한 파일은 IPC(프로세스 간 통신)를 위한 공유 메모리를 만드는 데 적합합니다.

프로세스, 뷰 및 메모리 관리

메모리 매핑된 파일은 여러 프로세스에서 공유할 수 있습니다. 프로세스는 파일을 만든 프로세스에서 할당한 일반 이름을 사용하여 동일한 메모리 매핑 파일에 매핑할 수 있습니다.

메모리 매핑된 파일을 사용하려면 전체 메모리 매핑 파일 또는 해당 파일의 일부 보기를 만들어야 합니다. 메모리 매핑된 파일의 동일한 부분에 여러 보기를 만들어 동시 메모리를 만들 수도 있습니다. 두 뷰가 동시에 유지되려면 동일한 메모리 매핑 파일에서 만들어야 합니다.

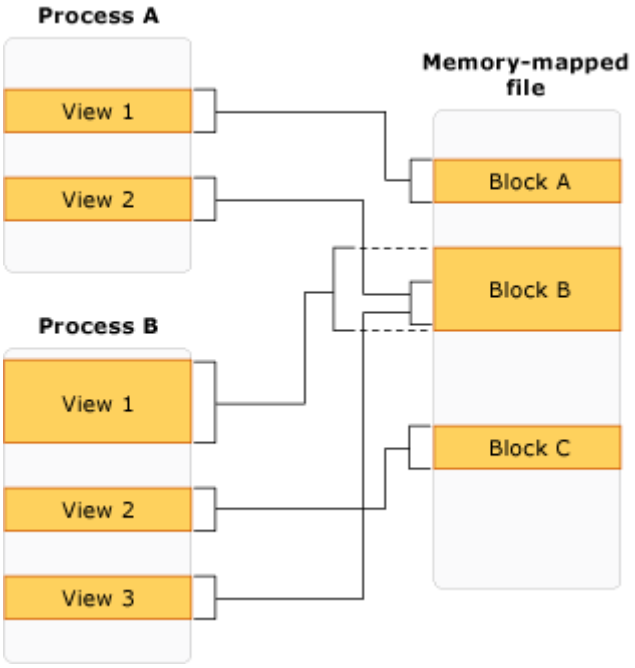
파일이 메모리 매핑에 사용할 수 있는 애플리케이션의 논리 메모리 공간 크기보다 큰 경우에도 여러 보기가 필요할 수 있습니다(32비트 컴퓨터의 경우 2GB).

스트림 액세스 뷰와 임의 액세스 보기의 두 가지 보기 유형이 있습니다. 파일에 대한 순차적 액세스를 위해 스트림 액세스 뷰를 사용합니다. 이는 비지속형 파일 및 IPC에 권장됩니다. 임의 액세스 보기는 지속형 파일 작업에 선호됩니다.

메모리 매핑된 파일은 운영 체제의 메모리 관리자를 통해 액세스되므로 파일이 자동으로 여러 페이지로 분할되고 필요에 따라 액세스됩니다. 메모리 관리를 직접 처리할 필요가 없습니다.

다음 그림에서는 여러 프로세스에서 동일한 메모리 매핑된 파일에 동시에 여러 뷰와 겹치는 보기를 가질 수 있는 방법을 보여 줍니다.

다음 이미지는 메모리 매핑된 파일에 대한 여러 겹의 중첩된 보기를 보여줍니다.



Memory-Mapped 파일을 사용한 프로그래밍

다음 표에서는 메모리 매핑된 파일 개체와 해당 멤버를 사용하기 위한 가이드를 제공합니다.

테이블 확장

과업	사용할 메서드 또는 속성
디스크의 MemoryMappedFile 파일에서 지속형 메모리 매핑 파일을 나타내는 개체를 가져오려면	MemoryMappedFile.CreateFromFile 메서드.
디스크의 파일과 연결되지 않은 지속되지 않는 메모리 매핑 파일을 나타내는 MemoryMappedFile 개체를 얻으려면	MemoryMappedFile.CreateNew 메서드. -또는- MemoryMappedFile.CreateOrOpen 메서드.
기존 메모리 매핑 파일의 개체 MemoryMappedFile를 가져오려면 (지속형 또는 비지속성 파일) 사용하십시오.	MemoryMappedFile.OpenExisting 메서드.
메모리 매핑된 파일에 대해 순차적으로 액세스할 수 있는 뷰의 개체를 얻으려면 UnmanagedMemoryStream	MemoryMappedFile.CreateViewStream 메서드.
메모리 매핑된 파일에 대한 무작위 접근 보기를 위한 개체를 가져오기 위해	MemoryMappedFile.CreateViewAccessor 메서드.

과업	사용할 메서드 또는 속성
UnmanagedMemoryAccessor	
관리되지 않는 코드에서 사용할 개체 SafeMemoryMappedViewHandle를 얻기 위해	MemoryMappedFile.SafeMemoryMappedFileHandle 속성 -또는- MemoryMappedViewAccessor.SafeMemoryMappedViewHandle 속성 -또는- MemoryMappedViewStream.SafeMemoryMappedViewHandle 속성
뷰가 생성될 때까지 메모리 할당을 지연하려면(비지속형 파일만 해당) (현재 시스템 페이지 크기를 확인하려면 속성을 사용합니다 Environment.SystemPageSize .)	CreateNew 값이 있는 메서드입니다 MemoryMappedFileOptions.DelayAllocatePages . -또는- CreateOrOpen 메서드 중 MemoryMappedFileOptions 열거형을 매개 변수로 사용하는 것입니다.

안전

열거형을 매개변수로 사용하는 다음 메서드를 사용하여 메모리 매핑 파일을 생성할 때 MemoryMappedFileAccess 액세스 권한을 적용할 수 있습니다.

- MemoryMappedFile.CreateFromFile
- MemoryMappedFile.CreateNew
- MemoryMappedFile.CreateOrOpen

매개 변수로 사용하는 MemoryMappedFileRights 메서드를 사용하여 OpenExisting 기존 메모리 매핑 파일을 열기 위한 액세스 권한을 지정할 수 있습니다.

또한 미리 정의된 액세스 규칙을 포함하는 개체를 포함 MemoryMappedFileSecurity 할 수 있습니다.

메모리 매핑된 파일에 새 액세스 또는 변경된 액세스 규칙을 적용하려면 이 메서드를 SetAccessControl 사용합니다. 기존 파일에서 액세스 또는 감사 규칙을 검색하려면 이 메서드를 GetAccessControl 사용합니다.

예시

지속형 Memory-Mapped 파일

메서드는 CreateFromFile 디스크의 기존 파일에서 메모리 매핑된 파일을 만듭니다.

다음 예제에서는 매우 큰 파일의 일부의 메모리 매핑된 뷰를 만들고 해당 부분을 조작합니다.

C#

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

class Program
{
    static void Main(string[] args)
    {
        long offset = 0x10000000; // 256 megabytes
        long length = 0x20000000; // 512 megabytes

        // Create the memory-mapped file.
        using (var mmf =
MemoryMappedFile.CreateFromFile(@"c:\ExtremelyLargeImage.data",
FileMode.Open,"ImgA"))
        {
            // Create a random access view, from the 256th megabyte (the offset)
            // to the 768th megabyte (the offset plus length).
            using (var accessor = mmf.CreateViewAccessor(offset, length))
            {
                int colorSize = Marshal.SizeOf(typeof(MyColor));
                MyColor color;

                // Make changes to the view.
                for (long i = 0; i < length; i += colorSize)
                {
                    accessor.Read(i, out color);
                    color.Brighten(10);
                    accessor.Write(i, ref color);
                }
            }
        }
    }
}

public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brighter.
    public void Brighten(short value)
```

```
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
        Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
    }
}
```

다음 예제에서는 다른 프로세스에 대해 동일한 메모리 매핑 파일을 엽니다.

C#

```
using System;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

class Program
{
    static void Main(string[] args)
    {
        // Assumes another process has created the memory-mapped file.
        using (var mmf = MemoryMappedFile.OpenExisting("ImgA"))
        {
            using (var accessor = mmf.CreateViewAccessor(4000000, 2000000))
            {
                int colorSize = Marshal.SizeOf(typeof(MyColor));
                MyColor color;

                // Make changes to the view.
                for (long i = 0; i < 1500000; i += colorSize)
                {
                    accessor.Read(i, out color);
                    color.Brighten(20);
                    accessor.Write(i, ref color);
                }
            }
        }
    }
}

public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brigher.
    public void Brighten(short value)
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
```

```
            Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
        }
    }
```

비지속형 Memory-Mapped 파일

CreateNew 및 CreateOrOpen 메서드는 디스크의 기존 파일에 매핑되지 않은 메모리 매핑된 파일을 생성합니다.

다음 예제는 메모리 매핑된 파일에 부울 값을 쓰는 세 개의 개별 프로세스(콘솔 애플리케이션)로 구성됩니다. 다음 작업 시퀀스가 발생합니다.

1. `Process A` 는 메모리 매핑된 파일을 만들고 해당 파일에 값을 씁니다.
2. `Process B` 는 메모리 매핑된 파일을 열고 값을 씁니다.
3. `Process C` 는 메모리 매핑된 파일을 열고 값을 씁니다.
4. `Process A` 는 메모리 매핑된 파일의 값을 읽고 표시합니다.
5. 메모리 매핑된 파일로 완료된 후에 `Process A` 는 가비지 수집을 통해 파일이 즉시 회수됩니다.

이 예제를 실행하려면 다음을 수행합니다.

1. 애플리케이션을 컴파일하고 세 개의 명령 프롬프트 창을 엽니다.
2. 첫 번째 명령 프롬프트 창에서 실행 `Process A`합니다.
3. 두 번째 명령 프롬프트 창에서 을 실행합니다 `Process B`.
4. `Process A`으로 돌아가서 Enter 키를 누르세요.
5. 세 번째 명령 프롬프트 창에서 을 실행합니다 `Process C`.
6. `Process A`으로 돌아가서 Enter 키를 누릅니다.

출력 `Process A` 은 다음과 같습니다.

콘솔

```
Start Process B and press ENTER to continue.
Start Process C and press ENTER to continue.
Process A says: True
Process B says: False
Process C says: True
```

A 처리

C#

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process A:
    static void Main(string[] args)
    {
        using (MemoryMappedFile mmf = MemoryMappedFile.CreateNew("testmap",
10000))
        {
            bool mutexCreated;
            Mutex mutex = new Mutex(true, "testmapmutex", out mutexCreated);
            using (MemoryMappedViewStream stream = mmf.CreateViewStream())
            {
                BinaryWriter writer = new BinaryWriter(stream);
                writer.Write(1);
            }
            mutex.ReleaseMutex();

            Console.WriteLine("Start Process B and press ENTER to continue.");
            Console.ReadLine();

            Console.WriteLine("Start Process C and press ENTER to continue.");
            Console.ReadLine();

            mutex.WaitOne();
            using (MemoryMappedViewStream stream = mmf.CreateViewStream())
            {
                BinaryReader reader = new BinaryReader(stream);
                Console.WriteLine($"Process A says: {reader.ReadBoolean()}");
                Console.WriteLine($"Process B says: {reader.ReadBoolean()}");
                Console.WriteLine($"Process C says: {reader.ReadBoolean()}");
            }
            mutex.ReleaseMutex();
        }
    }
}
```

프로세스 B

C#

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
```

```
using System.Threading;

class Program
{
    // Process B:
    static void Main(string[] args)
    {
        try
        {
            using (MemoryMappedFile mmf =
MemoryMappedFile.OpenExisting("testmap"))
            {

                Mutex mutex = Mutex.OpenExisting("testmapmutex");
                mutex.WaitOne();

                using (MemoryMappedViewStream stream = mmf.CreateViewStream(1, 0))
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(0);
                }
                mutex.ReleaseMutex();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Memory-mapped file does not exist. Run Process A
first.");
        }
    }
}
```

프로세스 C

C#

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process C:
    static void Main(string[] args)
    {
        try
        {
            using (MemoryMappedFile mmf =
MemoryMappedFile.OpenExisting("testmap"))
            {

                Mutex mutex = Mutex.OpenExisting("testmapmutex");
```

```
                mutex.WaitOne();

                using (MemoryMappedViewStream stream = mmf.CreateViewStream(2, 0))
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(1);
                }
                mutex.ReleaseMutex();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Memory-mapped file does not exist. Run Process A
first, then B.");
        }
    }
}
```

참고하십시오

- 파일 및 스트림 I/O

System.IO.FileStream 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스를 FileStream 사용하여 파일 시스템에서 파일을 읽고, 쓰고, 열고, 닫고, 파이프, 표준 입력 및 표준 출력을 비롯한 다른 파일 관련 운영 체제 핸들을 조작할 수 있습니다. 메서드 Read, Write, CopyTo, 및 Flush를 사용하여 동기 작업을 수행하거나, 메서드 ReadAsync, WriteAsync, CopyToAsync, 및 FlushAsync를 사용하여 비동기 작업을 수행할 수 있습니다. 비동기 메서드를 사용하여 주 스레드를 차단하지 않고 리소스 집약적 파일 작업을 수행합니다. 이 성능 고려 사항은 시간이 많이 걸리는 스트림 작업이 UI 스레드를 차단하고 앱이 작동하지 않는 것처럼 표시되도록 할 수 있는 Windows 8.x 스토어 앱 또는 데스크톱 앱에서 특히 중요합니다. FileStream 는 더 나은 성능을 위해 입력 및 출력을 버퍼링합니다.

> ⓘ **중요**
>
> 이 형식은 IDisposable 인터페이스를 구현합니다. 형식 사용을 마쳤으면 직접 또는 간접적으로 삭제해야 합니다. 형식을 직접 삭제하려면 Dispose `try` / 블록에서 해당 `catch` 메서드를 호출합니다. 간접적으로 삭제하려면 `using` (C#) 또는 `Using` (Visual Basic)와 같은 언어 구문을 사용합니다. 자세한 내용은 인터페이스 항목의 "IDisposable을 구현하는 개체 사용" 섹션을 IDisposable 참조하세요.

이 속성은 IsAsync 파일 핸들이 비동기적으로 열렸는지 여부를 검색합니다. 생성자에 FileStream, `isAsync`, 또는 `useAsync` 매개 변수가 있을 때 `options` 클래스의 인스턴스를 만들 때 이 값을 지정합니다. 속성이 `true` 인 경우, 스트림은 오버랩된 I/O를 사용하여 파일 작업을 비동기적으로 수행합니다. 그러나 IsAsync 속성은 `true`, ReadAsync 또는 WriteAsync 메서드를 호출하기 위해 CopyToAsync일 필요가 없습니다. IsAsync 속성이 `false` 있고 비동기 읽기 및 쓰기 작업을 호출하는 경우 UI 스레드는 여전히 차단되지 않지만 실제 I/O 작업은 동기적으로 수행됩니다.

이 메서드는 Seek 파일에 대한 임의 액세스를 지원합니다. Seek 를 사용하면 파일 내의 모든 위치로 읽기/쓰기 위치를 이동할 수 있습니다. 이 작업은 바이트 오프셋 참조 지점 매개 변수를 사용하여 수행됩니다. 바이트 오프셋은 열거형의 세 멤버 SeekOrigin 가 나타내는 대로 시작, 현재 위치 또는 기본 파일의 끝일 수 있는 검색 참조 지점을 기준으로 합니다.

> ⓘ **참고**
>
> 디스크 파일은 항상 임의 액세스를 지원합니다. 생성 당시 CanSeek 속성 값은 기본 파일 형식에 따라 `true` 또는 `false` 로 설정됩니다. winbase.h에 정의된 대로 기본 파일 형식이

FILE_TYPE_DISK인 경우, CanSeek 속성 값은 `true` 입니다. 그렇지 않으면 속성 값은 CanSeek `false` 입니다.

프로세스가 파일의 일부를 잠갔다가 종료되거나, 해결되지 않은 잠금이 있는 파일을 닫으면 동작이 정의되지 않습니다.

디렉터리 작업 및 기타 파일 작업에 대해서는 File, Directory, Path 클래스를 참조하세요. 이 File 클래스는 주로 파일 경로를 기반으로 개체를 만들기 FileStream 위한 정적 메서드가 있는 유틸리티 클래스입니다. 클래스는 MemoryStream 바이트 배열에서 스트림을 만들고 클래스와 비슷합니다 FileStream .

일반적인 파일 및 디렉터리 작업 목록은 일반적인 I/O 작업을 참조하세요.

스트림 위치 변경 탐지

FileStream 개체의 핸들에 배타적 보류가 없는 경우 다른 스레드는 파일 핸들에 동시에 액세스하고 파일 핸들과 연결된 운영 체제의 파일 포인터 위치를 변경할 수 있습니다. 이 경우 개체의 캐시된 위치 FileStream 와 버퍼의 캐시된 데이터가 손상될 수 있습니다. 개체는 FileStream 캐시된 버퍼에 액세스하는 메서드에 대한 검사를 정기적으로 수행하여 운영 체제의 핸들 위치가 개체에서 사용하는 FileStream 캐시된 위치와 동일한지 확인합니다.

메서드 호출 Read 에서 핸들 위치의 예기치 않은 변경이 감지되면 .NET은 버퍼의 내용을 삭제하고 파일에서 스트림을 다시 읽습니다. 이는 파일의 크기 및 파일 스트림의 위치에 영향을 줄 수 있는 다른 프로세스에 따라 성능에 영향을 줄 수 있습니다.

메서드 호출 Write 에서 핸들 위치의 예기치 않은 변경이 감지되면 버퍼의 내용이 삭제되고 예외가 IOException throw됩니다.

FileStream 개체가 핸들을 노출하기 위해 SafeFileHandle 속성에 액세스할 때 또는 생성자에서 FileStream 속성을 SafeFileHandle 개체에 제공할 때 해당 핸들에 대한 배타적인 보류가 없습니다.

System.IO.FileSystemWatcher 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

지정된 디렉터리의 변경 내용을 감시하는 데 사용합니다 FileSystemWatcher . 지정된 디렉터리의 파일 및 하위 디렉터리의 변경 내용을 확인할 수 있습니다. 로컬 컴퓨터, 네트워크 드라이브 또는 원격 컴퓨터에서 파일을 감시하는 구성 요소를 만들 수 있습니다.

모든 파일의 변경 내용을 확인하려면 속성을 빈 문자열("")로 설정 Filter 하거나 와일드카드("*.*")를 사용합니다. 특정 파일을 보려면 속성을 파일 이름으로 설정합니다 Filter . 예를 들어 파일 MyDoc.txt변경 내용을 확인하려면 속성을 "MyDoc.txt"로 설정합니다 Filter . 특정 형식의 파일에서 변경 내용을 확인할 수도 있습니다. 예를 들어 텍스트 파일의 변경 내용을 확인하려면 속성을 "*.txt"로 설정합니다 Filter .

디렉터리 또는 파일에서 확인할 수 있는 몇 가지 유형의 변경 내용이 있습니다. 예를 들어, `Attributes` 및 `LastWrite` 날짜와 시간 또는 파일이나 디렉터리의 `Size` 변경 사항을 확인할 수 있습니다. 해당 작업은 NotifyFilter 속성을 NotifyFilters 값 중 하나로 설정하여 수행됩니다. 볼 수 있는 변경 내용 유형에 대한 자세한 내용은 다음을 참조하세요 NotifyFilters.

파일 또는 디렉터리 이름 바꾸기, 삭제 또는 생성을 감시할 수 있습니다. 예를 들어 텍스트 파일 이름 바꾸기를 감시하려면 Filter 속성을 "*.txt"로 설정하고, 매개 변수에 WaitForChanged를 지정하여 Renamed 메서드를 호출합니다.

Windows 운영 체제는 FileSystemWatcher에 의해 생성된 버퍼에서 발생한 파일 변경 사항을 구성 요소에 알립니다. 짧은 시간에 많은 변경 내용이 있는 경우 버퍼가 오버플로할 수 있습니다. 이렇게 하면 구성 요소가 디렉터리의 변경 내용을 추적하지 않으며 일괄 알림만 제공합니다. 디스크로 교환할 수 없는 페이징되지 않은 메모리에서 제공되므로 속성을 사용하여 InternalBufferSize 버퍼의 크기를 늘리는 것은 비용이 많이 들기 때문에 버퍼를 파일 변경 이벤트를 놓치지 않을 만큼 작게 유지합니다. 버퍼 오버플로를 방지하려면 원치 않는 변경 알림을 필터링할 수 있도록 해당 및 NotifyFilter 속성을 사용합니다IncludeSubdirectories.

인스턴스 FileSystemWatcher의 초기 속성 값 목록은 생성자를 참조 FileSystemWatcher 하세요.

클래스 사용 시 고려 사항 FileSystemWatcher :

- 숨겨진 파일은 무시되지 않습니다.
- 일부 시스템에서는 FileSystemWatcher 짧은 8.3 파일 이름 형식을 사용하여 파일 변경 내용을 보고합니다. 예를 들어 "LongFileName.LongExtension"의 변경 내용은 "LongFil~.Lon"으로 보고될 수 있습니다.
- 이 클래스에는 모든 멤버에 적용되는 클래스 수준의 링크 요청 및 상속 요청이 포함됩니다. 직접 호출자 또는 파생 클래스가 완전 신뢰 권한을 가지지 않은 경우, SecurityException가 발생합니다. 보안 요구 사항에 대한 자세한 내용은 링크 요구를 참조하세요.

- 네트워크를 통해 디렉터리를 모니터링하기 위해 InternalBufferSize 속성에 설정할 수 있는 최대 크기는 64KB입니다.

폴더 복사 및 이동

운영 체제 및 FileSystemWatcher 개체는 잘라내기 및 붙여넣기 작업 또는 이동 동작을 폴더 및 해당 내용에 대한 이름 바꾸기 동작으로 해석합니다. 파일이 있는 폴더를 잘라내어 감시 FileSystemWatcher 중인 폴더에 붙여넣으면 개체는 폴더만 새 폴더로 보고하지만 기본적으로 이름이 바뀌기 때문에 해당 내용이 보고되지 않습니다.

폴더의 내용이 감시된 폴더로 이동되거나 복사되었음을 알리려면 다음 표에 설명된 대로 이벤트 처리기 메서드를 제공합니다 OnChangedOnRenamed .

테이블 확장

이벤트 처리기	처리된 이벤트	수행
OnChanged	Changed, , CreatedDeleted	파일 특성, 만든 파일 및 삭제된 파일의 변경 내용을 보고합니다.
OnRenamed	Renamed	이름이 바뀐 파일 및 폴더의 이전 경로와 새 경로를 나열하고 필요한 경우 재귀적으로 확장합니다.

이벤트 및 버퍼 크기

다음과 같이 발생하는 파일 시스템 변경 이벤트에는 몇 가지 요인이 영향을 줄 수 있습니다.

- 일반적인 파일 시스템 작업으로 두 개 이상의 이벤트가 발생할 수 있습니다. 예를 들어, 파일이 한 디렉터리에서 다른 디렉터리로 이동되면, 여러 OnChanged 이벤트와 일부 OnCreated 및 OnDeleted 이벤트가 발생할 수 있습니다. 파일 이동은 여러 개의 간단한 작업으로 구성된 복잡한 작업이므로 여러 이벤트가 발생합니다. 마찬가지로 일부 애플리케이션(예: 바이러스 백신 소프트웨어)은 FileSystemWatcher에 의해 감지되는 추가 파일 시스템 이벤트를 발생시킬 수 있습니다.
- FileSystemWatcher는 디스크가 전환되거나 제거되지 않는 한 디스크를 감시할 수 있습니다. FileSystemWatcher 타임스탬프를 변경하고 속성을 변경할 수 없으므로 CD 및 DVD에 대한 이벤트를 발생시키지 않습니다. 구성 요소가 제대로 작동하려면 원격 컴퓨터에 필요한 플랫폼 중 하나가 설치되어 있어야 합니다.

FileSystemWatcher 버퍼 크기를 초과하면 이벤트가 누락될 수 있습니다. 이벤트 누락을 방지하려면 다음 지침을 따르세요.

- 속성을 설정하여 버퍼 크기를 늘입니다 InternalBufferSize .

- 파일 이름이 긴 파일을 피하세요. 긴 파일 이름은 버퍼를 가득 채울 수 있습니다. 더 짧은 이름을 사용하여 이러한 파일의 이름을 바꾸는 것이 좋습니다.
- 이벤트 처리 코드를 최대한 짧게 유지합니다.

System.AppContext 클래스

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 AppContext 클래스를 사용하면 라이브러리 작성자가 사용자에게 새로운 기능을 위한 균일한 옵트아웃 메커니즘을 제공할 수 있습니다. 옵트아웃 요청을 전달하기 위해 구성 요소 간에 느슨하게 결합된 계약을 설정합니다. 이 기능은 일반적으로 기존 기능을 변경할 때 중요합니다. 반대로 새 기능에 대한 암시적 옵트인이 이미 있습니다.

라이브러리 개발자를 위한 AppContext

라이브러리는 클래스를 AppContext 사용하여 호환성 스위치를 정의하고 노출하는 반면 라이브러리 사용자는 라이브러리 동작에 영향을 주도록 해당 스위치를 설정할 수 있습니다. 기본적으로 라이브러리는 새 기능을 제공하며 스위치가 설정된 경우에만 변경합니다(즉, 이전 기능을 제공). 이렇게 하면 라이브러리가 이전 동작에 의존하는 호출자를 계속 지원하면서 기존 API에 대한 새 동작을 제공할 수 있습니다.

스위치 이름 정의

라이브러리 소비자가 동작 변경을 옵트아웃할 수 있도록 하는 가장 일반적인 방법은 명명된 스위치를 정의하는 것입니다. 해당 `value` 요소는 스위치 Boolean 의 이름과 해당 값으로 구성된 이름/값 쌍입니다. 기본적으로 스위치는 항상 암시적으로 `false`새 동작을 제공하며 기본적으로 새 동작을 옵트인합니다. 스위치를 `true`에 설정하면 레거시 동작이 활성화됩니다. 스위치를 `false`로 명시적으로 설정하면 새 동작이 제공됩니다.

라이브러리에서 공개하는 공식 계약이므로 스위치 이름에 일관된 형식을 사용하는 것이 좋습니다. 다음은 두 가지 명백한 형식입니다.

- *스위치.네임스페이스*. 스위치 이름
- *스위치.라이브러리.스위치 이름*

스위치를 정의하고 문서화하면 호출자는 메서드를 AppContext.SetSwitch(String, Boolean) 프로그래밍 방식으로 호출하여 사용할 수 있습니다. .NET Framework 앱은 AppContextSwitchOverrides< 요소를 애플리케이션 구성 파일에 추가>하거나 레지스트리를 사용하여 스위치를 사용할 수도 있습니다. 호출자가 구성 스위치의 AppContext 값을 사용하고 설정하는 방법에 대한 자세한 내용은 라이브러리 소비자에 대한 AppContext 섹션을 참조하세요.

.NET Framework에서 공용 언어 런타임이 애플리케이션을 실행하면 레지스트리의 호환성 설정을 자동으로 읽고 애플리케이션 구성 파일을 로드하여 애플리케이션 인스턴스 AppContext 를 채웁니다. AppContext 인스턴스는 호출자 또는 런타임에 의해 프로그래밍 방식으로 채워지므로

.NET Framework 앱은 인스턴스를 구성 SetSwitch 하기 위해 메서드 호출 AppContext 과 같은 작업을 수행할 필요가 없습니다.

설정 확인

소비자가 스위치의 값을 선언했는지 확인하고, AppContext.TryGetSwitch 메서드를 호출하여 적절한 조치를 취할 수 있습니다. 인수가 `true` 발견되면 메서드가 반환 `switchName` 되고 해당 `isEnabled` 인수는 스위치의 값을 나타냅니다. 그렇지 않으면 메서드가 반환됩니다 `false`.

예시

다음 예제에서는 클래스를 사용하여 AppContext 고객이 라이브러리 메서드의 원래 동작을 선택할 수 있도록 하는 방법을 보여 줍니다. 다음은 이름이 `StringLibrary` 1.0인 라이브러리의 버전 1.0입니다. 더 큰 문자열 내에서 하위 문자열의 시작 인덱스를 결정하기 위해 서수 비교를 수행하는 메서드를 정의 `SubstringStartsAt` 합니다.

C#

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("1.0.0.0")]

public static class StringLibrary1
{
    public static int SubstringStartsAt(string fullString, string substr)
    {
        return fullString.IndexOf(substr, StringComparison.Ordinal);
    }
}
```

다음 예제에서는 라이브러리를 사용하여 "고고학자"에서 부분 문자열 "archæ"의 시작 인덱스 찾기를 찾습니다. 메서드가 서수 비교를 수행하므로 부분 문자열을 찾을 수 없습니다.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        string value = "The archaeologist";
        string substring = "archæ";
        int position = StringLibrary1.SubstringStartsAt(value, substring);
        if (position >= 0)
            Console.WriteLine($"'{substring}' found in '{value}' starting at
```

```
position {position}");
        else
            Console.WriteLine($"'{substring}' not found in '{value}'");
    }
}
// The example displays the following output:
//       'archæ' not found in 'The archaeologist'
```

그러나 라이브러리 버전 2.0은 문화권 구분 비교를 `SubstringStartsAt` 사용하도록 메서드를 변경합니다.

C#

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("2.0.0.0")]

public static class StringLibrary2
{
    public static int SubstringStartsAt(string fullString, string substr)
    {
        return fullString.IndexOf(substr, StringComparison.CurrentCulture);
    }
}
```

앱이 라이브러리의 새 버전에 대해 실행되도록 다시 컴파일되면 이제 부분 문자열 "archæ"가 "고고학자"의 인덱스 4에서 발견되었다고 보고합니다.

C#

```
using System;

public class Example2
{
    public static void Main()
    {
        string value = "The archaeologist";
        string substring = "archæ";
        int position = StringLibrary2.SubstringStartsAt(value, substring);
        if (position >= 0)
            Console.WriteLine($"'{substring}' found in '{value}' starting at
position {position}");
        else
            Console.WriteLine($"'{substring}' not found in '{value}'");
    }
}
// The example displays the following output:
//       'archæ' found in 'The archaeologist' starting at position 4
```

이 변경은 스위치를 정의하여 원래 동작에 의존하는 애플리케이션을 중단하지 않도록 방지할 수 있습니다. 이 경우 스위치의 이름이 지정 `StringLibrary.DoNotUseCultureSensitiveComparison` 됩니다. 기본값인 `false`은 라이브러리가 버전 2.0의 문화권 구분 비교를 수행해야 함을 나타냅니다. `true` 는 라이브러리가 해당 버전 1.0 서수 비교를 수행해야 했음을 나타냅니다. 이전 코드를 약간 수정하면 라이브러리 소비자가 스위치를 설정하여 메서드가 수행하는 비교 종류를 결정할 수 있습니다.

C#

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("2.0.0.0")]

public static class StringLibrary
{
   public static int SubstringStartsAt(string fullString, string substr)
   {
      bool flag;
      if
(AppContext.TryGetSwitch("StringLibrary.DoNotUseCultureSensitiveComparison", out
flag) && flag == true)
         return fullString.IndexOf(substr, StringComparison.Ordinal);
      else
         return fullString.IndexOf(substr, StringComparison.CurrentCulture);
   }
}
```

.NET Framework 애플리케이션은 다음 구성 파일을 사용하여 버전 1.0 동작을 복원할 수 있습니다.

XML

```
<configuration>
   <runtime>
      <AppContextSwitchOverrides
value="StringLibrary.DoNotUseCultureSensitiveComparison=true" />
   </runtime>
</configuration>
```

애플리케이션이 구성 파일을 사용하여 실행되면 다음 출력이 생성됩니다.

출력

```
'archæ' not found in 'The archaeologist'
```

라이브러리 소비자를 위한 AppContext

라이브러리 AppContext 의 소비자인 경우 클래스를 사용하면 라이브러리 또는 라이브러리 메서드의 옵트아웃 메커니즘을 활용하여 새 기능을 사용할 수 있습니다. 호출하는 클래스 라이브러리의 개별 메서드는 새 동작을 사용하거나 사용하지 않도록 설정하는 특정 스위치를 정의합니다. 스위치의 값은 부울입니다. 일반적으로 기본값인 경우 `false`새 동작이 활성화됩니다. 이 경우 `true`새 동작은 사용하지 않도록 설정되고 멤버는 이전과 같이 동작합니다.

코드에서 메서드를 호출 AppContext.SetSwitch(String, Boolean) 하여 스위치 값을 설정할 수 있습니다. 인수는 `switchName` 스위치 이름을 정의하고 `isEnabled` 속성은 스위치의 값을 정의합니다. AppContext 정적 클래스이므로 애플리케이션별 도메인 기준으로 사용할 수 있습니다. 애플리케이션 범위에서 AppContext.SetSwitch(String, Boolean)를 호출하면, 이는 애플리케이션에만 영향을 줍니다.

.NET Framework 앱에는 스위치의 값을 설정하는 추가 방법이 있습니다.

- app.config 파일의 > 섹션에 요소를 추가합니다. 스위치에는 스위치 이름과 해당 값을 모두 포함하는 키/값 쌍을 나타내는 문자열인 단일 특성 `value`이 있습니다.

  여러 스위치를 정의하려면 AppContextSwitchOverrides< 요소의 특성에서> 각 스위치의 `value` 키/값 쌍을 세미콜론으로 구분합니다. 이 경우 요소의 `<AppContextSwitchOverrides>` 형식은 다음과 같습니다.

  ```xml
  <AppContextSwitchOverrides value="switchName1=value1;switchName2=value2" />
  ```

  `<AppContextSwitchOverrides>` 요소를 사용하여 구성 설정을 정의하면 애플리케이션 범위가 있습니다. 즉, 애플리케이션에만 영향을 줍니다.

  > **참고**
  >
  > .NET Framework에서 정의한 스위치에 대한 자세한 내용은 **AppContextSwitchOverrides< 요소를 참조>하세요**.

- 레지스트리에 항목을 추가합니다. **HKLM\SOFTWARE\Microsoft\.NETFramework\AppContext** 하위 키에 새 문자열 값을 추가합니다. 항목의 이름을 스위치 이름으로 설정합니다. 해당 값을 다음 옵션 `True` `true` `False` `false`중 하나로 설정합니다. 런타임에서 다른 값이 발견되면 스위치를 무시합니다.

64비트 운영 체제에서는 동일한 항목을 **HKLM\SOFTWARE\Wow6432Node\Microsoft\.NETFramework\AppContext** 하위 키에도 추가해야 합니다.

레지스트리를 사용하여 스위치를 AppContext 정의하면 컴퓨터 범위가 있습니다. 즉, 컴퓨터에서 실행되는 모든 애플리케이션에 영향을 줍니다.

ASP.NET 및 ASP.NET Core 애플리케이션의 경우 web.config 파일의 <appSettings> 섹션에 <Add> 요소를 추가하여 스위치를 설정합니다. 다음은 그 예입니다.

XML

```
<appSettings>
   <add key="AppContext.SetSwitch:switchName1" value="switchValue1" />
   <add key="AppContext.SetSwitch:switchName2" value="switchValue2" />
</appSettings>
```

둘 이상의 방법으로 동일한 스위치를 설정하는 경우 다른 설정을 재정의하는 우선 순위는 다음과 같습니다.

1. 프로그래밍 방식 설정입니다.
2. app.config 파일(.NET Framework 앱의 경우) 또는 web.config 파일(ASP.NET Core 앱의 경우)의 설정입니다.
3. 레지스트리 설정(.NET Framework 앱에만 해당).

참고하십시오

- AppContext 스위치

.NET의 콘솔 앱

2025. 06. 17.

.NET 애플리케이션은 클래스를 System.Console 사용하여 문자를 읽고 콘솔에 문자를 쓸 수 있습니다. 콘솔의 데이터는 표준 입력 스트림에서 읽고, 콘솔에 대한 데이터는 표준 출력 스트림에 기록되고, 콘솔에 대한 오류 데이터는 표준 오류 출력 스트림에 기록됩니다. 애플리케이션이 시작되면 이러한 스트림은 콘솔과 자동으로 연결되며, 각각 In, Out, Error 속성으로 표시됩니다.

Console.In 속성의 값은 System.IO.TextReader 객체인 반면, Console.Out 및 Console.Error 속성의 값은 System.IO.TextWriter 객체입니다. 이러한 속성을 콘솔을 나타내지 않는 스트림과 연결하여 스트림을 입력 또는 출력을 위해 다른 위치로 가리킬 수 있습니다. 예를 들어 Console.Out 속성을 System.IO.StreamWriter로 설정하여 System.IO.FileStream 메서드를 통해 Console.SetOut를 캡슐화하면 출력을 파일로 리디렉션할 수 있습니다. Console.In 및 Console.Out 속성은 동일한 스트림을 참조할 필요가 없습니다.

> **참고**
>
> C#, Visual Basic 및 C++의 예제를 포함하여 콘솔 애플리케이션을 빌드하는 방법에 대한 자세한 내용은 클래스에 대한 Console 설명서를 참조하세요.

예를 들어 Windows Forms 애플리케이션에 콘솔이 없으면 정보를 쓸 콘솔이 없으므로 표준 출력 스트림에 기록된 출력이 표시되지 않습니다. 액세스할 수 없는 콘솔에 정보를 쓰면 예외가 발생하지 않습니다. (Visual Studio의 프로젝트 속성 페이지에서와 같이 항상 애플리케이션 유형을 **콘솔 애플리케이션**으로 변경할 수 있습니다.)

System.Console 클래스에는 콘솔에서 개별 문자 또는 전체 줄을 읽을 수 있는 메서드가 있습니다. 다른 메서드는 데이터 및 형식 문자열을 변환한 다음 서식이 지정된 문자열을 콘솔에 씁니다. 문자열 서식 지정에 대한 자세한 내용은 서식 지정 형식을 참조하세요.

> **팁**
>
> 콘솔 애플리케이션에는 기본적으로 시작되는 메시지 펌프가 없습니다. 따라서 Microsoft Win32 타이머에 대한 콘솔 호출이 실패할 수 있습니다.

참고하십시오

- System.Console
- 서식 유형

System.Console 클래스

아티클 • 2025. 03. 29.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

콘솔은 사용자가 컴퓨터 키보드를 통해 텍스트 입력을 입력하고 컴퓨터 터미널에서 텍스트 출력을 읽어 운영 체제 또는 텍스트 기반 콘솔 애플리케이션과 상호 작용하는 운영 체제 창입니다. 예를 들어 Windows 운영 체제에서 콘솔을 명령 프롬프트 창이라고 하며 MS-DOS 명령을 수락합니다. Console 클래스는 콘솔에서 문자를 읽고 문자를 쓰는 애플리케이션에 대한 기본 지원을 제공합니다.

콘솔 I/O 스트림

콘솔 애플리케이션이 시작되면 운영 체제는 표준 입력 스트림, 표준 출력 스트림 및 표준 오류 출력 스트림이라는 세 개의 I/O 스트림을 콘솔에 자동으로 연결합니다. 애플리케이션은 표준 입력 스트림에서 사용자 입력을 읽을 수 있습니다. 표준 출력 스트림에 일반 데이터를 씁니다. 및 표준 오류 출력 스트림에 오류 데이터를 씁니다. 이러한 스트림은 애플리케이션에 Console.In, Console.Out및 Console.Error 속성의 값으로 표시됩니다.

기본적으로 In 속성 값은 키보드를 나타내는 System.IO.TextReader 개체이며 Out 및 Error 속성의 값은 콘솔 창을 나타내는 System.IO.TextWriter 개체입니다. 그러나 이러한 속성을 콘솔 창이나 키보드를 나타내지 않는 스트림으로 설정할 수 있습니다. 예를 들어 이러한 속성을 파일을 나타내는 스트림으로 설정할 수 있습니다. 표준 입력, 표준 출력 또는 표준 오류 스트림을 리디렉션하려면 Console.SetIn, Console.SetOut또는 Console.SetError 메서드를 각각 호출합니다. 이러한 스트림을 사용하는 I/O 작업은 동기화됩니다. 즉, 여러 스레드가 스트림에서 읽거나 쓸 수 있습니다. 즉, 개체가 콘솔 스트림을 나타내는 경우 일반적으로 비동기 메서드(예: TextReader.ReadLineAsync)가 동기적으로 실행됩니다.

> **참고**
>
> Console 클래스를 사용하여 서버 애플리케이션과 같은 무인 애플리케이션에서 출력을 표시하지 마세요. Console.Write 및 Console.WriteLine 같은 메서드에 대한 호출은 GUI 애플리케이션에 영향을 주지 않습니다.

기본 스트림이 콘솔로 전송될 때 정상적으로 작동하는 Console 클래스 멤버는 스트림이 파일로 리디렉션되는 경우 예외를 throw할 수 있습니다. 표준 스트림을 리디렉션하는 경우 System.IO.IOException 예외를 catch하도록 애플리케이션을 작성하십시오. IsOutputRedirected, IsInputRedirected및 IsErrorRedirected 속성을 사용하여 System.IO.IOException 예외를 throw하는 작업을 수행하기 전에 표준 스트림이 리디렉션되는지 여부를 확인할 수도 있습니다.

In, Out및 Error 속성으로 표시되는 스트림 개체의 멤버를 명시적으로 호출하는 것이 유용할 수 있습니다. 예를 들어 기본적으로 Console.ReadLine 메서드는 표준 입력 스트림에서 입력을 읽습니다. 마찬가지로, Console.WriteLine 메서드는 표준 출력 스트림에 데이터를 쓰고, 그 뒤에 Environment.NewLine에서 찾을 수 있는 기본 줄 종료 문자열이 있습니다. 그러나 Console 클래스는 표준 오류 출력 스트림에 데이터를 쓰는 해당 메서드 또는 해당 스트림에 기록된 데이터의 줄 종료 문자열을 변경하는 속성을 제공하지 않습니다.

Out 또는 Error 속성의 TextWriter.NewLine 속성을 다른 줄 종료 문자열로 설정하여 이 문제를 해결할 수 있습니다. 예를 들어 다음 C# 문은 표준 오류 출력 스트림의 줄 종료 문자열을 두 캐리지 리턴 및 줄 피드 시퀀스로 설정합니다.

```
Console.Error.NewLine = "\r\n\r\n";
```

그런 다음, 다음 C# 문과 같이 오류 출력 스트림 개체의 WriteLine 메서드를 명시적으로 호출할 수 있습니다.

```
Console.Error.WriteLine();
```

화면 버퍼 및 콘솔 창

콘솔의 두 가지 밀접하게 관련된 기능은 화면 버퍼와 콘솔 창입니다. 텍스트는 실제로 본체가 소유한 스트림에서 읽거나 쓰여지지만 화면 버퍼라고 하는 콘솔이 소유한 영역에서 읽거나 쓰는 것처럼 보입니다. 화면 버퍼는 콘솔의 특성이며 각 그리드 교집합 또는 문자 셀에 문자를 포함할 수 있는 행과 열의 사각형 눈금으로 구성됩니다. 각 문자에는 고유한 전경색이 있고 각 문자 셀에는 고유한 배경색이 있습니다.

화면 버퍼는 콘솔 창이라는 사각형 영역을 통해 볼 수 있습니다. 콘솔 창은 콘솔의 또 다른 특성입니다. 운영 체제 창인 콘솔 자체가 아닙니다. 콘솔 창은 행과 열로 정렬되고 화면 버퍼의 크기보다 작거나 같으며 기본 화면 버퍼의 여러 영역을 보려면 이동할 수 있습니다. 화면 버퍼가 콘솔 창보다 큰 경우 콘솔은 화면 버퍼 영역 위에 콘솔 창의 위치를 변경할 수 있도록 스크롤 막대를 자동으로 표시합니다.

커서는 현재 텍스트를 읽거나 쓰는 화면 버퍼 위치를 나타냅니다. 커서를 숨기거나 표시할 수 있으며 높이를 변경할 수 있습니다. 커서가 표시되면 콘솔 창 위치가 자동으로 이동되므로 커서가 항상 표시됩니다.

화면 버퍼의 문자 셀 좌표 원점은 왼쪽 위 모서리이며 커서와 콘솔 창의 위치는 해당 원점과 상대적으로 측정됩니다. 0부터 시작하는 인덱스를 사용하여 위치를 지정합니다. 즉, 맨 위 행을 행 0으로 지정하고 맨 왼쪽 열을 열 0으로 지정합니다. 행 및 열 인덱스의 최대값은 Int16.MaxValue.

콘솔에 대한 유니코드 지원

일반적으로 콘솔은 입력을 읽고 시스템 로캘이 기본적으로 정의하는 현재 콘솔 코드 페이지를 사용하여 출력을 씁니다. 코드 페이지는 사용 가능한 유니코드 문자의 하위 집합만 처리할 수 있으므로 특정 코드 페이지에서 매핑되지 않은 문자를 표시하려고 하면 콘솔에서 모든 문자를 표시하거나 정확하게 나타낼 수 없습니다. 다음 예제에서는 이 문제를 보여 줍니다. U+0410에서 U+044F까지의 키릴 자모 알파벳 문자를 콘솔에 표시하려고 합니다. 콘솔 코드 페이지 437을 사용하는 시스템에서 예제를 실행하는 경우 키릴 자모 문자가 코드 페이지 437의 문자에 매핑되지 않으므로 각 문자가 물음표(?)로 바뀝 있습니다.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        // Create a Char array for the modern Cyrillic alphabet,
        // from U+0410 to U+044F.
        int nChars = 0x044F - 0x0410 + 1;
        char[] chars = new char[nChars];
        ushort codePoint = 0x0410;
        for (int ctr = 0; ctr < chars.Length; ctr++)
        {
            chars[ctr] = (char)codePoint;
            codePoint++;
        }

        Console.WriteLine($"Current code page:
{Console.OutputEncoding.CodePage}\n");
        // Display the characters.
        foreach (var ch in chars)
        {
            Console.Write("{0}  ", ch);
            if (Console.CursorLeft >= 70)
                Console.WriteLine();
        }
    }
}
// The example displays the following output:
//    Current code page: 437
//
//    ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?
//    ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?
//    ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?  ?
```

Console 클래스는 코드 페이지를 지원하는 것 외에도 UTF8Encoding 클래스로 UTF-8 인코딩을 지원합니다. .NET Framework 4.5부터 Console 클래스는 UnicodeEncoding 클래스

를 사용한 UTF-16 인코딩도 지원합니다. 콘솔에 유니코드 문자를 표시하려면 OutputEncoding 속성을 UTF8Encoding 또는 UnicodeEncoding설정합니다.

유니코드 문자를 지원하려면 인코더가 특정 유니코드 문자를 인식해야 하며 해당 문자를 렌더링하는 데 필요한 문자 모양이 있는 글꼴도 필요합니다. 콘솔에 유니코드 문자를 성공적으로 표시하려면 콘솔 글꼴을 비 래스터 또는 TrueType 글꼴(예: Consolas 또는 Lucida Console)로 설정해야 합니다. 다음 예제에서는 프로그래밍 방식으로 래스터 글꼴에서 Lucida 콘솔로 글꼴을 변경할 수 있는 방법을 보여 줍니다.

C#

```
using System;
using System.Runtime.InteropServices;

public class Example2
{
    [DllImport("kernel32.dll", SetLastError = true)]
    static extern IntPtr GetStdHandle(int nStdHandle);

    [DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError =
true)]
    static extern bool GetCurrentConsoleFontEx(
           IntPtr consoleOutput,
           bool maximumWindow,
           ref CONSOLE_FONT_INFO_EX lpConsoleCurrentFontEx);

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool SetCurrentConsoleFontEx(
           IntPtr consoleOutput,
           bool maximumWindow,
           CONSOLE_FONT_INFO_EX consoleCurrentFontEx);

    private const int STD_OUTPUT_HANDLE = -11;
    private const int TMPF_TRUETYPE = 4;
    private const int LF_FACESIZE = 32;
    private static IntPtr INVALID_HANDLE_VALUE = new IntPtr(-1);

    public static unsafe void Main()
    {
        string fontName = "Lucida Console";
        IntPtr hnd = GetStdHandle(STD_OUTPUT_HANDLE);
        if (hnd != INVALID_HANDLE_VALUE)
        {
            CONSOLE_FONT_INFO_EX info = new CONSOLE_FONT_INFO_EX();
            info.cbSize = (uint)Marshal.SizeOf(info);
            bool tt = false;
            // First determine whether there's already a TrueType font.
            if (GetCurrentConsoleFontEx(hnd, false, ref info))
            {
                tt = (info.FontFamily & TMPF_TRUETYPE) == TMPF_TRUETYPE;
                if (tt)
                {
```

```
                    Console.WriteLine("The console already is using a
TrueType font.");
                    return;
                }
                // Set console font to Lucida Console.
                CONSOLE_FONT_INFO_EX newInfo = new CONSOLE_FONT_INFO_EX();
                newInfo.cbSize = (uint)Marshal.SizeOf(newInfo);
                newInfo.FontFamily = TMPF_TRUETYPE;
                IntPtr ptr = new IntPtr(newInfo.FaceName);
                Marshal.Copy(fontName.ToCharArray(), 0, ptr,
fontName.Length);
                // Get some settings from current font.
                newInfo.dwFontSize = new COORD(info.dwFontSize.X,
info.dwFontSize.Y);
                newInfo.FontWeight = info.FontWeight;
                SetCurrentConsoleFontEx(hnd, false, newInfo);
            }
        }
    }

    [StructLayout(LayoutKind.Sequential)]
    internal struct COORD
    {
        internal short X;
        internal short Y;

        internal COORD(short x, short y)
        {
            X = x;
            Y = y;
        }
    }

    [StructLayout(LayoutKind.Sequential)]
    internal unsafe struct CONSOLE_FONT_INFO_EX
    {
        internal uint cbSize;
        internal uint nFont;
        internal COORD dwFontSize;
        internal int FontFamily;
        internal int FontWeight;
        internal fixed char FaceName[LF_FACESIZE];
    }
}
```

그러나 TrueType 글꼴은 문자 모양의 하위 집합만 표시할 수 있습니다. 예를 들어 Lucida 콘솔 글꼴은 U+0021에서 U+FB02까지 사용 가능한 약 64,000자 중 643개만 표시합니다. 특정 글꼴이 지원하는 문자를 보려면 제어판에서 **글꼴** 애플릿을 열고, **문자** 찾기 옵션을 선택하고, **문자 맵** 창의 **글꼴** 목록에서 검사할 문자 집합의 글꼴을 선택합니다.

Windows에서는 글꼴 연결을 사용하여 특정 글꼴에서 사용할 수 없는 문자 모양을 표시합니다. 추가 문자 집합을 표시하는 글꼴 연결에 대한 자세한 내용은 세계화 단계별: 글꼴

참조하세요. 연결된 글꼴은 레지스트리의 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontLink\SystemLink 하위 키에 정의됩니다. 이 하위 키와 연결된 각 항목은 기본 글꼴의 이름에 해당하며 해당 값은 글꼴 파일과 기본 글꼴에 연결된 글꼴을 정의하는 문자열 배열입니다. 배열의 각 멤버는 연결된 글꼴을 정의하고 글꼴-파일 이름, *글꼴 이름*형식을 사용합니다. 다음 예제에서는 Simsun.ttc라는 글꼴 파일에 있는 SimSun 이라는 연결된 글꼴을 프로그래밍 방식으로 정의하여 간체 한 문자를 표시하는 방법을 보여 줍니다.

C#

```
using Microsoft.Win32;
using System;

public class Example
{
   public static void Main()
   {
      string valueName = "Lucida Console";
      string newFont = "simsun.ttc,SimSun";
      string[] fonts = null;
      RegistryValueKind kind = 0;
      bool toAdd;

      RegistryKey key = Registry.LocalMachine.OpenSubKey(
                @"Software\Microsoft\Windows
NT\CurrentVersion\FontLink\SystemLink",
                true);
      if (key == null) {
         Console.WriteLine("Font linking is not enabled.");
      }
      else {
         // Determine if the font is a base font.
         string[] names = key.GetValueNames();
         if (Array.Exists(names, s => s.Equals(valueName,
                                    StringComparison.OrdinalIgnoreCase)))
{
            // Get the value's type.
            kind = key.GetValueKind(valueName);

            // Type should be RegistryValueKind.MultiString, but we can't be
sure.
            switch (kind) {
               case RegistryValueKind.String:
                  fonts = new string[] { (string) key.GetValue(valueName) };
                  break;
               case RegistryValueKind.MultiString:
                  fonts = (string[]) key.GetValue(valueName);
                  break;
               case RegistryValueKind.None:
                  // Do nothing.
                  fonts = new string[] { };
```

```
                    break;
                }
                // Determine whether SimSun is a linked font.
                if (Array.FindIndex(fonts, s =>s.IndexOf("SimSun",
                                                StringComparison.OrdinalIgnoreCase)
>=0) >= 0) {
                   Console.WriteLine("Font is already linked.");
                   toAdd = false;
                }
                else {
                   // Font is not a linked font.
                   toAdd = true;
                }
            }
            else {
               // Font is not a base font.
               toAdd = true;
               fonts = new string[] { };
            }

            if (toAdd) {
               Array.Resize(ref fonts, fonts.Length + 1);
               fonts[fonts.GetUpperBound(0)] = newFont;
               // Change REG_SZ to REG_MULTI_SZ.
               if (kind == RegistryValueKind.String)
                  key.DeleteValue(valueName, false);

               key.SetValue(valueName, fonts, RegistryValueKind.MultiString);
               Console.WriteLine("SimSun added to the list of linked fonts.");
            }
         }

         if (key != null) key.Close();
      }
   }
```

콘솔에 대한 유니코드 지원에는 다음과 같은 제한 사항이 있습니다.

- UTF-32 인코딩은 지원되지 않습니다. 지원되는 유니코드 인코딩은 각각 UTF8Encoding 및 UnicodeEncoding 클래스로 표현되는 UTF-8 및 UTF-16입니다.
- 양방향 출력은 지원되지 않습니다.
- 연결된 글꼴 파일에 정의된 경우에도 기본 다국어 평면 외부(즉, 서로게이트 쌍)의 문자 표시는 지원되지 않습니다.
- 복잡한 스크립트의 문자 표시는 지원되지 않습니다.
- 문자 시퀀스 결합(즉, 기본 문자와 하나 이상의 결합 문자로 구성된 문자)은 별도의 문자로 표시됩니다. 이 제한을 해결하려면 콘솔에 출력을 보내기 전에 String.Normalize 메서드를 호출하여 표시할 문자열을 정규화할 수 있습니다. 다음

예제에서는 결합 문자 시퀀스 U+0061 U+0308을 포함하는 문자열이 출력 문자열이 정규화되기 전에 두 문자로 콘솔에 표시되고 String.Normalize 메서드가 호출된 후 단일 문자로 표시됩니다.

```
C#

using System;
using System.IO;

public class Example1
{
    public static void Main()
    {
        char[] chars = { '\u0061', '\u0308' };

        string combining = new String(chars);
        Console.WriteLine(combining);

        combining = combining.Normalize();
        Console.WriteLine(combining);
    }
}
// The example displays the following output:
//       a"
//       ä
```

정규화는 문자의 유니코드 표준에 특정 결합 문자 시퀀스에 해당하는 미리 구성된 형식이 포함된 경우에만 실행 가능한 솔루션입니다.

- 글꼴이 프라이빗 사용 영역의 코드 포인트에 대한 문자 모양을 제공하는 경우 해당 문자 모양이 표시됩니다. 그러나 프라이빗 사용 영역의 문자는 애플리케이션별로 지정되므로 예상 문자 모양이 아닐 수 있습니다.

다음 예제에서는 콘솔에 유니코드 문자 범위를 표시합니다. 이 예제에서는 표시할 범위의 시작, 표시할 범위의 끝, 현재 콘솔 인코딩(`false`) 또는 UTF-16 인코딩(`true`)을 사용할지 여부 등 세 가지 명령줄 매개 변수를 허용합니다. 콘솔에서 TrueType 글꼴을 사용한다고 가정합니다.

```
C#

using System;
using System.IO;
using System.Globalization;
using System.Text;

public static class DisplayChars
{
    private static void Main(string[] args)
    {
```

```
        uint rangeStart = 0;
        uint rangeEnd = 0;
        bool setOutputEncodingToUnicode = true;
        // Get the current encoding so we can restore it.
        Encoding originalOutputEncoding = Console.OutputEncoding;

        try
        {
            switch (args.Length)
            {
                case 2:
                    rangeStart = uint.Parse(args[0],
NumberStyles.HexNumber);
                    rangeEnd = uint.Parse(args[1], NumberStyles.HexNumber);
                    setOutputEncodingToUnicode = true;
                    break;
                case 3:
                    if (!uint.TryParse(args[0], NumberStyles.HexNumber,
null, out rangeStart))
                        throw new ArgumentException(string.Format("{0} is
not a valid hexadecimal number.", args[0]));

                    if (!uint.TryParse(args[1], NumberStyles.HexNumber,
null, out rangeEnd))
                        throw new ArgumentException(string.Format("{0} is
not a valid hexadecimal number.", args[1]));

                    _ = bool.TryParse(args[2], out
setOutputEncodingToUnicode);
                    break;
                default:
                    Console.WriteLine("Usage: {0} <{1}> <{2}> [{3}]",
                        Environment.GetCommandLineArgs()[0],
                        "startingCodePointInHex",
                        "endingCodePointInHex",
                        "<setOutputEncodingToUnicode?{true|false,
default:false}>");
                    return;
            }

            if (setOutputEncodingToUnicode)
            {
                try
                {
                    // Set encoding using endianness of this system.
                    // We're interested in displaying individual Char
objects, so
                    // we don't want a Unicode BOM or exceptions to be
thrown on
                    // invalid Char values.
                    Console.OutputEncoding = new
UnicodeEncoding(!BitConverter.IsLittleEndian, false);
                    Console.WriteLine("\nOutput encoding set to UTF-16");
                }
                catch (IOException)
```

```
                {
                    Console.OutputEncoding = new UTF8Encoding();
                    Console.WriteLine("Output encoding set to UTF-8");
                }
            }
            else
            {
                Console.WriteLine($"The console encoding is
{Console.OutputEncoding.EncodingName} (code page
{Console.OutputEncoding.CodePage})");
            }
            DisplayRange(rangeStart, rangeEnd);
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Restore console environment.
            Console.OutputEncoding = originalOutputEncoding;
        }
    }

    public static void DisplayRange(uint start, uint end)
    {
        const uint upperRange = 0x10FFFF;
        const uint surrogateStart = 0xD800;
        const uint surrogateEnd = 0xDFFF;

        if (end <= start)
        {
            uint t = start;
            start = end;
            end = t;
        }

        // Check whether the start or end range is outside of last plane.
        if (start > upperRange)
            throw new ArgumentException(string.Format("0x{0:X5} is outside
the upper range of Unicode code points (0x{1:X5})",
                                                      start, upperRange));
        if (end > upperRange)
            throw new ArgumentException(string.Format("0x{0:X5} is outside
the upper range of Unicode code points (0x{0:X5})",
                                                      end, upperRange));

        // Since we're using 21-bit code points, we can't use U+D800 to
U+DFFF.
        if ((start < surrogateStart & end > surrogateStart) || (start >=
surrogateStart & start <= surrogateEnd))
            throw new ArgumentException(string.Format("0x{0:X5}-0x{1:X5}
includes the surrogate pair range 0x{2:X5}-0x{3:X5}",
                                                      start, end,
surrogateStart, surrogateEnd));
```

```
        uint last = RoundUpToMultipleOf(0x10, end);
        uint first = RoundDownToMultipleOf(0x10, start);

        uint rows = (last - first) / 0x10;

        for (uint r = 0; r < rows; ++r)
        {
            // Display the row header.
            Console.Write("{0:x5} ", first + 0x10 * r);

            for (uint c = 0; c < 0x10; ++c)
            {
                uint cur = (first + 0x10 * r + c);
                if (cur < start)
                {
                    Console.Write($" {(char)(0x20)} ");
                }
                else if (end < cur)
                {
                    Console.Write($" {(char)(0x20)} ");
                }
                else
                {
                    // the cast to int is safe, since we know that val <=
upperRange.
                    String chars = Char.ConvertFromUtf32((int)cur);
                    // Display a space for code points that are not valid
characters.
                    if (CharUnicodeInfo.GetUnicodeCategory(chars[0]) ==

UnicodeCategory.OtherNotAssigned)
                        Console.Write($" {(char)(0x20)} ");
                    // Display a space for code points in the private use
area.
                    else if (CharUnicodeInfo.GetUnicodeCategory(chars[0]) ==

UnicodeCategory.PrivateUse)
                        Console.Write($" {(char)(0x20)} ");
                    // Is surrogate pair a valid character?
                    // Note that the console will interpret the high and low
surrogate
                    // as separate (and unrecognizable) characters.
                    else if (chars.Length > 1 &&
CharUnicodeInfo.GetUnicodeCategory(chars, 0) ==

UnicodeCategory.OtherNotAssigned)
                        Console.Write($" {(char)(0x20)} ");
                    else
                        Console.Write($" {chars} ");
                }

                switch (c)
                {
                    case 3:
                    case 11:
```

```
                        Console.Write("-");
                        break;
                    case 7:
                        Console.Write("--");
                        break;
                }
            }

            Console.WriteLine();
            if (0 < r && r % 0x10 == 0)
                Console.WriteLine();
        }
    }

    private static uint RoundUpToMultipleOf(uint b, uint u)
    {
        return RoundDownToMultipleOf(b, u) + b;
    }

    private static uint RoundDownToMultipleOf(uint b, uint u)
    {
        return u - (u % b);
    }
}
// If the example is run with the command line
//       DisplayChars 0400 04FF true
// the example displays the Cyrillic character set as follows:
//       Output encoding set to UTF-16
//       00400  È  Ë  Ђ  Ѓ - Є  S  I  Ї -- J  Љ  Њ  Ћ - Ќ  Ѝ  Ў  Џ
//       00410  A  Б  В  Г - Д  Е  Ж  З -- И  Й  К  Л - М  Н  О  П
//       00420  Р  С  Т  У - Ф  Х  Ц  Ч -- Ш  Щ  Ъ  Ы - Ь  Э  Ю  Я
//       00430  а  б  в  г - д  е  ж  з -- и  й  к  л - м  н  о  п
//       00440  р  с  т  у - ф  х  ц  ч -- ш  щ  ъ  ы - ь  э  ю  я
//       00450  è  ё  ђ  ѓ - є  s  i  ї -- j  љ  њ  ћ - ќ  ѝ  ў  џ
//       00460  Ѡ  ѡ  Ѣ  ѣ - Ѧ  ѧ  Ѫ  ѫ -- Ѭ  ѭ  Ѯ  ѯ - Ѳ  ѳ  Ѵ  ѵ
//       00470  Ѱ  ѱ  Ѳ  ѳ - V  v  Ѷ  ѷ -- Ѹ  ѹ  Ѻ  ѻ - Ѽ  ѽ  Ѿ  ѿ
//       00480  Ҁ  ҁ  ҂  ҃ - ҆  ҇  ҈  ҉ --  Ҋҋ  Ҍ  ҍ - Ҏ  ҏ  Ґ  ґ
//       00490  Ғ  ғ  Ҕ  ҕ - Җ  җ  Ҙ  ҙ -- Қ  қ  Ҝ  ҝ - Ҟ  ҟ  Ҡ  ҡ
//       004a0  Ҡ  ҡ  Ң  ң - Ҥ  ҥ  Ҧ  ҧ -- Ҩ  ҩ  Ҫ  ҫ - Ҵ  ҵ  Ҷ  ҷ
//       004b0  Ҹ  ҹ  Һ  һ - Ҽ  ҽ  Ҿ  ҿ -- Ӄ  ӄ  Ӆ  ӆ - Ӎ  ӎ  ӏ
//       004c0  I  Ӂ  ӂ  Ӄ - ӄ  Ӆ  ӆ  Ӈ -- ӈ  Ӊ  ӊ  Ӌ - ӌ  Ӎ  ӎ  I
//       004d0  Ӑ  ӑ  Ӓ  ӓ - Æ  æ  Ӗ  ӗ -- Ә  ә  Ӛ  ӛ - Ӟ  ӟ  Ӡ  ӡ
//       004e0  З  з  Ӣ  ӣ - Ӥ  ӥ  Ӧ  ӧ -- Ө  ө  Ӫ  ӫ - Ӯ  ӯ  Ӱ  ӱ
//       004f0  Ӳ  ӳ  Ӵ  ӵ - Ӹ  ӹ  Ӻ  ӻ -- Ӽ  ӽ  Ӿ  ӿ - Ҳ  ҳ  Ҵ  ҵ
```

일반적인 작업

Console 클래스에는 콘솔 입력을 읽고 콘솔 출력을 쓰기 위한 다음 메서드가 포함되어 있습니다.

- ReadKey 메서드의 오버로드는 개별 문자를 읽습니다.

- ReadLine 메서드는 전체 입력 줄을 읽습니다.
- Write 메서드 오버로드는 값 형식의 인스턴스, 문자 배열 또는 개체 집합을 서식이 지정되거나 서식이 지정되지 않은 문자열로 변환한 다음 해당 문자열을 콘솔에 씁니다.
- WriteLine 메서드의 병렬 집합은 Write 오버로드와 동일한 문자열을 출력하지만 줄 종료 문자열도 추가합니다.

Console 클래스에는 다음 작업을 수행하는 메서드와 속성도 포함되어 있습니다.

- 화면 버퍼의 크기를 가져오기 또는 설정합니다. BufferHeight 및 BufferWidth 속성을 사용하면 각각 버퍼 높이와 너비를 얻거나 설정할 수 있으며 SetBufferSize 메서드를 사용하면 단일 메서드 호출에서 버퍼 크기를 설정할 수 있습니다.
- 콘솔 창의 크기를 가져오기 또는 설정합니다. WindowHeight 및 WindowWidth 속성을 사용하면 창 높이와 너비를 각각 얻거나 설정할 수 있으며, SetWindowSize 메서드를 사용하면 단일 메서드 호출에서 창 크기를 설정할 수 있습니다.
- 커서의 크기를 가져오기 또는 설정합니다. CursorSize 속성은 문자 셀에 있는 커서의 높이를 지정합니다.
- 화면 버퍼를 기준으로 콘솔 창의 위치를 설정하거나 가져옵니다. WindowTop 및 WindowLeft 속성을 사용하면 콘솔 창에 표시되는 화면 버퍼의 맨 위 행과 맨 왼쪽 열을 얻거나 설정할 수 있으며, SetWindowPosition 메서드를 사용하면 단일 메서드 호출에서 이러한 값을 설정할 수 있습니다.
- CursorTop 및 CursorLeft 속성을 가져오거나 설정하여 커서 위치를 가져오거나 설정하거나 SetCursorPosition 메서드를 호출하여 커서의 위치를 설정합니다.
- MoveBufferArea 또는 Clear 메서드를 호출하여 화면 버퍼에서 데이터를 이동하거나 지울 수 있습니다.
- ForegroundColor 및 BackgroundColor 속성을 사용하여 전경색과 배경색을 얻거나 설정하거나 ResetColor 메서드를 호출하여 배경색과 전경을 기본 색으로 다시 설정합니다.
- Beep 메서드를 호출하여 콘솔 스피커를 통해 경고음 소리를 재생합니다.

.NET Core 참고 사항

데스크톱의 .NET Framework에서 Console 클래스는 일반적으로 코드 페이지 인코딩인 `GetConsoleCP` 및 `GetConsoleOutputCP`반환된 인코딩을 사용합니다. 예를 들어 문화권이 영어(미국)인 시스템에서 코드 페이지 437은 기본적으로 사용되는 인코딩입니다. 그러나

.NET Core는 이러한 인코딩의 제한된 하위 집합만 사용할 수 있습니다. 이 경우 Encoding.UTF8 콘솔의 기본 인코딩으로 사용됩니다.

앱이 특정 코드 페이지 인코딩에 의존하는 경우, Console 메서드를 호출하기 전에 *을 수행하여 계속 사용할 수 있도록* 하십시오.

1. CodePagesEncodingProvider.Instance 속성에서 EncodingProvider 개체를 검색합니다.

2. EncodingProvider 개체를 Encoding.RegisterProvider 메서드에 전달하여 인코딩 공급자가 지원하는 추가 인코딩을 사용할 수 있도록 합니다.

Console 클래스는 Console 출력 메서드를 호출하기 전에 인코딩 공급자를 등록한 경우 UTF8 대신 기본 시스템 인코딩을 자동으로 사용합니다.

예시

다음 예제에서는 표준 입력 및 출력 스트림에서 데이터를 읽고 데이터를 쓰는 방법을 보여 줍니다. 이러한 스트림은 SetIn 및 SetOut 메서드를 사용하여 리디렉션할 수 있습니다.

C#

```
using System;

public class Example4
{
    public static void Main()
    {
        Console.Write("Hello ");
        Console.WriteLine("World!");
        Console.Write("Enter your name: ");
        string name = Console.ReadLine();
        Console.Write("Good day, ");
        Console.Write(name);
        Console.WriteLine("!");
    }
}
// The example displays output similar to the following:
//       Hello World!
//       Enter your name: James
//       Good day, James!
```

System.Random 클래스

아티클 • 2024. 01. 08.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

클래스는 Random 난수에 대한 특정 통계 요구 사항을 충족하는 숫자 시퀀스를 생성하는 알고리즘인 의사 난수 생성기를 나타냅니다.

의사 난수는 유한한 숫자 집합에서 동일한 확률로 선택됩니다. 선택한 숫자는 수학 알고리즘을 사용하여 선택하는 데 사용되기 때문에 완전히 임의적인 것은 아니지만 실제 용도로는 충분히 무작위입니다. 클래스의 Random 구현은 도널드 E. Knuth의 빼기 난수 생성기 알고리즘의 수정 된 버전을 기반으로합니다. 자세한 내용은 D. E. Knuth를 참조하세요. *컴퓨터 프로그래밍의 예술, 볼륨 2: 반수 알고리즘*. 애디슨 웨슬리, 독서, MA, 제 3 판, 1997.

임의 암호를 만드는 데 적합한 것과 같이 암호화적으로 안전한 난수를 생성하려면 클래스를 RNGCryptoServiceProvider 사용하거나 클래스 System.Security.Cryptography.RandomNumberGenerator를 파생합니다.

난수 생성기 인스턴스화

클래스 생성자에 시드 값(의사 난수 생성 알고리즘의 시작 값)을 제공하여 난수 생성기를 Random 인스턴스화합니다. 시드 값을 명시적으로 또는 암시적으로 제공할 수 있습니다.

- Random(Int32) 생성자는 사용자가 제공하는 명시적 시드 값을 사용합니다.
- Random() 생성자는 기본 시드 값을 사용합니다. 난수 생성기를 인스턴스화하는 가장 일반적인 방법입니다.

.NET Framework에서 기본 시드 값은 시간에 따라 다릅니다. .NET Core에서 기본 시드 값은 스레드 정적 의사 난수 생성기에 의해 생성됩니다.

동일한 시드가 별도의 Random 개체에 사용되는 경우 동일한 일련의 난수를 생성합니다. 임의 값을 처리하는 테스트 제품군을 만들거나 난수에서 데이터를 파생시키는 게임을 재생하는 데 유용할 수 있습니다. 그러나 Random 다른 버전의 .NET Framework에서 실행되는 프로세스의 개체는 동일한 시드 값으로 인스턴스화되더라도 다른 일련의 난수를 반환할 수 있습니다.

서로 다른 난수 시퀀스를 생성하려면 시드 값을 시간에 따라 달라지게 하여 새 인스턴스 Random마다 다른 계열을 생성할 수 있습니다. 매개 변수가 있는 Random(Int32)Int32 생성자는 현재 시간의 틱 수에 따라 값을 사용할 수 있지만 매개 변수가 없는 Random() 생성자는 시스템 클록을 사용하여 시드 값을 생성합니다. 그러나 .NET Framework에서는 클록에 유한 해상도가 있으므로 매개 변수가 없는 생성자를 사용하여 연속적으로 서로 다

른 Random 개체를 만들면 동일한 난수 시퀀스를 생성하는 난수 생성기가 만들어집니다. 다음 예제에서는 .NET Framework 애플리케이션에서 연속적으로 인스턴스화되는 두 Random 개체가 동일한 일련의 난수를 생성하는 방법을 보여 줍니다. 대부분의 Windows 시스템에서 Random 는 서로 15밀리초 이내에 만들어진 개체의 시드 값이 동일할 수 있습니다.

C#

```
byte[] bytes1 = new byte[100];
byte[] bytes2 = new byte[100];
Random rnd1 = new Random();
Random rnd2 = new Random();

rnd1.NextBytes(bytes1);
rnd2.NextBytes(bytes2);

Console.WriteLine("First Series:");
for (int ctr = bytes1.GetLowerBound(0);
     ctr <= bytes1.GetUpperBound(0);
     ctr++) {
   Console.Write("{0, 5}", bytes1[ctr]);
   if ((ctr + 1) % 10 == 0) Console.WriteLine();
}

Console.WriteLine();

Console.WriteLine("Second Series:");
for (int ctr = bytes2.GetLowerBound(0);
     ctr <= bytes2.GetUpperBound(0);
     ctr++) {
   Console.Write("{0, 5}", bytes2[ctr]);
   if ((ctr + 1) % 10 == 0) Console.WriteLine();
}

// The example displays output like the following:
//       First Series:
//          97  129  149   54   22  208  120  105   68  177
//         113  214   30  172   74  218  116  230   89   18
//          12  112  130  105  116  180  190  200  187  120
//           7  198  233  158   58   51   50  170   98   23
//          21    1  113   74  146  245   34  255   96   24
//         232  255   23    9  167  240  255   44  194   98
//          18  175  173  204  169  171  236  127  114   23
//         167  202  132   65  253   11  254   56  214  127
//         145  191  104  163  143    7  174  224  247   73
//          52    6  231  255    5  101   83  165  160  231
//
//       Second Series:
//          97  129  149   54   22  208  120  105   68  177
//         113  214   30  172   74  218  116  230   89   18
//          12  112  130  105  116  180  190  200  187  120
//           7  198  233  158   58   51   50  170   98   23
//          21    1  113   74  146  245   34  255   96   24
```

```
//         232  255   23    9  167  240  255   44  194   98
//          18  175  173  204  169  171  236  127  114   23
//         167  202  132   65  253   11  254   56  214  127
//         145  191  104  163  143    7  174  224  247   73
//          52    6  231  255    5  101   83  165  160  231
```

이 문제를 방지하려면 여러 개체 대신 단일 Random 개체를 만듭니다. .NET Core의 `Random` 클래스에는 이 제한이 없습니다.

여러 인스턴스화 방지

.NET Framework에서 꽉 꽉 루프 또는 빠른 연속으로 두 난수 생성기를 초기화하면 동일한 난수 시퀀스를 생성할 수 있는 두 개의 난수 생성기가 만들어집니다. 대부분의 경우 이는 개발자의 의도가 아니며 난수 생성기를 인스턴스화하고 초기화하는 것은 비교적 비용이 많이 드는 프로세스이므로 성능 문제가 발생할 수 있습니다.

성능을 향상시키고 동일한 숫자 시퀀스를 생성하는 별도의 난수 생성기를 실수로 만들지 않도록 하려면 하나의 난수를 생성하는 새 Random 개체를 만드는 Random 대신 시간이 지남에 따라 많은 난수를 생성하는 개체를 만드는 것이 좋습니다.

그러나 클래스는 Random 스레드로부터 안전하지 않습니다. 여러 스레드에서 메서드를 호출 Random 하는 경우 다음 섹션에서 설명하는 지침을 따릅니다.

스레드로부터의 안전성

개별 Random 개체를 인스턴스화하는 대신 앱에 필요한 모든 난수를 생성하는 단일 Random 인스턴스를 만드는 것이 좋습니다. 그러나 Random 개체는 스레드로부터 안전하지 않습니다. 앱이 여러 스레드에서 메서드를 호출 Random 하는 경우 동기화 개체를 사용하여 한 번에 하나의 스레드만 난수 생성기에 액세스할 수 있도록 해야 합니다. 스레드로부터 안전한 방식으로 개체에 Random 액세스할 수 없도록 하려면 난수를 반환하는 메서드를 호출하면 0이 반환됩니다.

다음 예제에서는 C# lock 문, F# 잠금 함수 및 Visual Basic SyncLock 문을 사용하여 스레드로부터 안전한 방식으로 11개 스레드에서 단일 난수 생성기에 액세스하도록 합니다. 각 스레드는 2백만 개의 난수를 생성하고, 생성된 난수 수를 계산하고, 해당 합계를 계산한 다음, 실행이 완료되면 모든 스레드의 합계를 업데이트합니다.

C#

```
using System;
using System.Threading;

public class Example13
{
```

```
    [ThreadStatic] static double previous = 0.0;
    [ThreadStatic] static int perThreadCtr = 0;
    [ThreadStatic] static double perThreadTotal = 0.0;
    static CancellationTokenSource source;
    static CountdownEvent countdown;
    static Object randLock, numericLock;
    static Random rand;
    double totalValue = 0.0;
    int totalCount = 0;

    public Example13()
    {
        rand = new Random();
        randLock = new Object();
        numericLock = new Object();
        countdown = new CountdownEvent(1);
        source = new CancellationTokenSource();
    }

    public static void Main()
    {
        Example13 ex = new Example13();
        Thread.CurrentThread.Name = "Main";
        ex.Execute();
    }

    private void Execute()
    {
        CancellationToken token = source.Token;

        for (int threads = 1; threads <= 10; threads++)
        {
            Thread newThread = new Thread(this.GetRandomNumbers);
            newThread.Name = threads.ToString();
            newThread.Start(token);
        }
        this.GetRandomNumbers(token);

        countdown.Signal();
        // Make sure all threads have finished.
        countdown.Wait();
        source.Dispose();

        Console.WriteLine("\nTotal random numbers generated: {0:N0}",
totalCount);
        Console.WriteLine("Total sum of all random numbers: {0:N2}",
totalValue);
        Console.WriteLine("Random number mean: {0:N4}", totalValue /
totalCount);
    }

    private void GetRandomNumbers(Object o)
    {
        CancellationToken token = (CancellationToken)o;
        double result = 0.0;
```

```
        countdown.AddCount(1);

        try
        {
            for (int ctr = 0; ctr < 2000000; ctr++)
            {
                // Make sure there's no corruption of Random.
                token.ThrowIfCancellationRequested();

                lock (randLock)
                {
                    result = rand.NextDouble();
                }
                // Check for corruption of Random instance.
                if ((result == previous) && result == 0)
                {
                    source.Cancel();
                }
                else
                {
                    previous = result;
                }
                perThreadCtr++;
                perThreadTotal += result;
            }

            Console.WriteLine("Thread {0} finished execution.",
                              Thread.CurrentThread.Name);
            Console.WriteLine("Random numbers generated: {0:N0}",
perThreadCtr);
            Console.WriteLine("Sum of random numbers: {0:N2}",
perThreadTotal);
            Console.WriteLine("Random number mean: {0:N4}\n", perThreadTotal
/ perThreadCtr);

            // Update overall totals.
            lock (numericLock)
            {
                totalCount += perThreadCtr;
                totalValue += perThreadTotal;
            }
        }
        catch (OperationCanceledException e)
        {
            Console.WriteLine("Corruption in Thread {1}", e.GetType().Name,
Thread.CurrentThread.Name);
        }
        finally
        {
            countdown.Signal();
        }
    }
}
// The example displays output like the following:
//       Thread 6 finished execution.
```

```
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 1,000,491.05
//        Random number mean: 0.5002
//
//        Thread 10 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,329.64
//        Random number mean: 0.4997
//
//        Thread 4 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 1,000,166.89
//        Random number mean: 0.5001
//
//        Thread 8 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,628.37
//        Random number mean: 0.4998
//
//        Thread Main finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,920.89
//        Random number mean: 0.5000
//
//        Thread 3 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,370.45
//        Random number mean: 0.4997
//
//        Thread 7 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,330.92
//        Random number mean: 0.4997
//
//        Thread 9 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 1,000,172.79
//        Random number mean: 0.5001
//
//        Thread 5 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 1,000,079.43
//        Random number mean: 0.5000
//
//        Thread 1 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,817.91
//        Random number mean: 0.4999
//
//        Thread 2 finished execution.
//        Random numbers generated: 2,000,000
//        Sum of random numbers: 999,930.63
//        Random number mean: 0.5000
//
//
```

```
//       Total random numbers generated: 22,000,000
//       Total sum of all random numbers: 10,998,238.98
//       Random number mean: 0.4999
```

이 예제에서는 다음과 같은 방법으로 스레드 보안을 보장합니다.

- 이 ThreadStaticAttribute 특성은 생성된 총 난수 수와 각 스레드의 합계를 추적하는 스레드 지역 변수를 정의하는 데 사용됩니다.
- 잠금(C#의 `lock` 문, `lock` F#의 함수 및 `SyncLock` Visual Basic의 문)은 모든 스레드에서 생성된 모든 난수의 총 개수 및 합계에 대한 변수에 대한 액세스를 보호합니다.
- 세마포(CountdownEvent개체)는 다른 모든 스레드가 실행을 완료할 때까지 기본 스레드가 차단되도록 하는 데 사용됩니다.
- 이 예제에서는 난수 생성 메서드에 대한 두 번의 연속 호출에서 0을 반환하는지 여부를 확인하여 난수 생성기가 손상되었는지 여부를 검사. 손상이 감지되면 이 예제에서는 개체를 CancellationTokenSource 사용하여 모든 스레드를 취소해야 한다는 신호를 표시합니다.
- 각 난수를 생성하기 전에 각 스레드는 개체의 CancellationToken 상태를 검사. 취소가 요청되면 이 예제에서는 메서드를 CancellationToken.ThrowIfCancellationRequested 호출하여 스레드를 취소합니다.

다음 예제는 개체 대신 Thread 개체와 람다 식을 사용 Task 한다는 점을 제외하고 첫 번째 예제와 동일합니다.

C#

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

public class Example15
{
    static Object randLock, numericLock;
    static Random rand;
    static CancellationTokenSource source;
    double totalValue = 0.0;
    int totalCount = 0;

    public Example15()
    {
        rand = new Random();
        randLock = new Object();
        numericLock = new Object();
        source = new CancellationTokenSource();
    }

    public static async Task Main()
    {
```

```
        Example15 ex = new Example15();
        Thread.CurrentThread.Name = "Main";
        await ex.Execute();
    }

    private async Task Execute()
    {
        List<Task> tasks = new List<Task>();

        for (int ctr = 0; ctr <= 10; ctr++)
        {
            CancellationToken token = source.Token;
            int taskNo = ctr;
            tasks.Add(Task.Run(() =>
               {
                    double previous = 0.0;
                    int taskCtr = 0;
                    double taskTotal = 0.0;
                    double result = 0.0;

                    for (int n = 0; n < 2000000; n++)
                    {
                        // Make sure there's no corruption of Random.
                        token.ThrowIfCancellationRequested();

                        lock (randLock)
                        {
                            result = rand.NextDouble();
                        }
                        // Check for corruption of Random instance.
                        if ((result == previous) && result == 0)
                        {
                            source.Cancel();
                        }
                        else
                        {
                            previous = result;
                        }
                        taskCtr++;
                        taskTotal += result;
                    }

                    // Show result.
                    Console.WriteLine("Task {0} finished execution.",
taskNo);
                    Console.WriteLine("Random numbers generated: {0:N0}",
taskCtr);
                    Console.WriteLine("Sum of random numbers: {0:N2}",
taskTotal);
                    Console.WriteLine("Random number mean: {0:N4}\n",
taskTotal / taskCtr);

                    // Update overall totals.
                    lock (numericLock)
                    {
```

```
                        totalCount += taskCtr;
                        totalValue += taskTotal;
                    }
                },
            token));
        }
        try
        {
            await Task.WhenAll(tasks.ToArray());
            Console.WriteLine("\nTotal random numbers generated: {0:N0}",
totalCount);
            Console.WriteLine("Total sum of all random numbers: {0:N2}",
totalValue);
            Console.WriteLine("Random number mean: {0:N4}", totalValue /
totalCount);
        }
        catch (AggregateException e)
        {
            foreach (Exception inner in e.InnerExceptions)
            {
                TaskCanceledException canc = inner as TaskCanceledException;
                if (canc != null)
                    Console.WriteLine("Task #{0} cancelled.", canc.Task.Id);
                else
                    Console.WriteLine("Exception: {0}",
inner.GetType().Name);
            }
        }
        finally
        {
            source.Dispose();
        }
    }
}
// The example displays output like the following:
//       Task 1 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 1,000,502.47
//       Random number mean: 0.5003
//
//       Task 0 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 1,000,445.63
//       Random number mean: 0.5002
//
//       Task 2 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 1,000,556.04
//       Random number mean: 0.5003
//
//       Task 3 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 1,000,178.87
//       Random number mean: 0.5001
//
```

```
//       Task 4 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 999,819.17
//       Random number mean: 0.4999
//
//       Task 5 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 1,000,190.58
//       Random number mean: 0.5001
//
//       Task 6 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 999,720.21
//       Random number mean: 0.4999
//
//       Task 7 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 999,000.96
//       Random number mean: 0.4995
//
//       Task 8 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 999,499.33
//       Random number mean: 0.4997
//
//       Task 9 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 1,000,193.25
//       Random number mean: 0.5001
//
//       Task 10 finished execution.
//       Random numbers generated: 2,000,000
//       Sum of random numbers: 999,960.82
//       Random number mean: 0.5000
//
//
//       Total random numbers generated: 22,000,000
//       Total sum of all random numbers: 11,000,067.33
//       Random number mean: 0.5000
```

이 예제는 다음과 같은 방법으로 첫 번째 예제와 다릅니다.

- 생성된 난수의 수와 각 작업의 합계를 추적하는 변수는 작업에 로컬이므로 특성을 사용할 ThreadStaticAttribute 필요가 없습니다.
- 정적 Task.WaitAll 메서드는 모든 작업이 완료되기 전에 기본 스레드가 완료되지 않도록 하는 데 사용됩니다. 개체가 CountdownEvent 필요하지 않습니다.
- 작업 취소로 인한 예외가 메서드에 Task.WaitAll 표시됩니다. 이전 예제에서는 각 스레드에서 처리됩니다.

다양한 유형의 난수 생성

난수 생성기는 다음과 같은 종류의 난수를 생성할 수 있는 메서드를 제공합니다.

- 일련의 Byte 값입니다. 메서드가 메서드로 반환 NextBytes 할 요소 수에 초기화된 배열을 전달하여 바이트 값의 수를 결정합니다. 다음 예제에서는 20바이트를 생성합니다.

```
C#

Random rnd = new Random();
Byte[] bytes = new Byte[20];
rnd.NextBytes(bytes);
for (int ctr = 1; ctr <= bytes.Length; ctr++)
{
    Console.Write("{0,3}   ", bytes[ctr - 1]);
    if (ctr % 10 == 0) Console.WriteLine();
}

// The example displays output like the following:
//       141    48   189    66   134   212   211    71   161    56
//       181   166   220   133     9   252   222    57    62    62
```

- 단일 정수입니다. 메서드를 호출하여 정수에서 최대값(Int32.MaxValue - 1)까지 정수, 메서드를 호출 Next() 하여 0과 특정 값 사이의 정수 또는 메서드를 호출 Next(Int32)Next(Int32, Int32) 하여 값 범위 내의 정수 중에서 선택할 수 있습니다. 매개 변수가 있는 오버로드에서 지정된 최대값은 배타적입니다. 즉, 생성된 실제 최대 수는 지정된 값보다 1보다 작습니다.

  다음 예제에서는 메서드를 Next(Int32, Int32) 호출하여 -10에서 10 사이의 10개의 난수를 생성합니다. 메서드에 대한 두 번째 인수는 메서드에서 반환된 임의 값 범위의 배타적 상한을 지정합니다. 즉, 메서드가 반환할 수 있는 가장 큰 정수는 이 값보다 1보다 작습니다.

```
C#

Random rnd = new Random();
for (int ctr = 0; ctr < 10; ctr++)
{
    Console.Write("{0,3}   ", rnd.Next(-10, 11));
}

// The example displays output like the following:
//    2     9    -3     2     4    -7    -3    -8    -8     5
```

- 메서드를 호출 NextDouble 하여 0.0에서 1.0 미만까지의 단일 부동 소수점 값입니다. 메서드에서 반환된 난수의 배타적 상한은 1이므로 실제 상한은 0.9999999999999978입니다. 다음 예제에서는 10개의 임의 부동 소수점 숫자를 생성합니다.

C#

```
Random rnd = new Random();
for (int ctr = 0; ctr < 10; ctr++)
{
    Console.Write("{0,-19:R}   ", rnd.NextDouble());
    if ((ctr + 1) % 3 == 0) Console.WriteLine();
}

// The example displays output like the following:
//    0.7911680553998649    0.0903414949264105    0.79776258291572455
//    0.615568345233597     0.652644504165577     0.84023809378977776
//    0.099662564741290441   0.91341467383942321  0.96018602045261581
//    0.74772306473354022
```

ⓘ 중요

이 **Next(Int32, Int32)** 메서드를 사용하면 반환된 난수의 범위를 지정할 수 있습니다. 그러나 `maxValue` 반환된 상위 범위를 지정하는 매개 변수는 포괄 값이 아닌 배타적인 값입니다. 즉, 메서드 호출 `Next(0, 100)` 은 0에서 100 사이가 아닌 0에서 99 사이의 값을 반환합니다.

임의 Random 부울 값을 생성하고, 지정된 범위에서 임의 부동 소수점 값을 생성하고, 임의의 64비트 정수 생성을 생성하고, 배열 또는 컬렉션에서 고유한 요소를 검색하는 등의 작업에 클래스를 사용할 수도 있습니다.

사용자 고유의 알고리즘 대체

클래스에서 Random 상속하고 난수 생성 알고리즘을 제공하여 고유한 난수 생성기를 구현할 수 있습니다. 고유한 알고리즘을 제공하려면 난수 생성 알고리즘을 Sample 구현하는 메서드를 재정의해야 합니다. 재정의된 메서드를 Next()호출하도록 메서드 Next(Int32, Int32)및 NextBytes 메서드를 재정의해야 합니다 Sample . 및 NextDouble 메서드를 재정 Next(Int32) 의할 필요가 없습니다.

클래스에서 Random 파생되고 기본 의사 난수 생성기를 수정하는 예제는 참조 페이지를 참조 Sample 하세요.

동일한 임의 값 시퀀스 검색

소프트웨어 테스트 시나리오와 게임 플레이에서 동일한 난수 시퀀스를 생성하려는 경우가 있습니다. 동일한 난수 시퀀스로 테스트하면 회귀를 감지하고 버그 수정을 확인할 수 있습니다. 게임에서 동일한 난수 시퀀스를 사용하면 이전 게임을 재생할 수 있습니다.

생성자에 동일한 시드 값을 제공하여 동일한 난수 시퀀스를 생성할 Random(Int32) 수 있습니다. 시드 값은 의사 난수 생성 알고리즘의 시작 값을 제공합니다. 다음 예제에서는 100100을 임의의 시드 값으로 사용하여 개체를 인스턴스화 Random 하고, 20개의 임의 부동 소수점 값을 표시하고, 시드 값을 유지합니다. 그런 다음 시드 값을 복원하고, 새 난수 생성기를 인스턴스화하고, 동일한 20개의 임의 부동 소수점 값을 표시합니다. 이 예제에서는 다른 버전의 .NET에서 실행되는 경우 다양한 난수 시퀀스를 생성할 수 있습니다.

C#

```
using System;
using System.IO;

public class Example12
{
    public static void Main()
    {
        int seed = 100100;
        ShowRandomNumbers(seed);
        Console.WriteLine();

        PersistSeed(seed);

        DisplayNewRandomNumbers();
    }

    private static void ShowRandomNumbers(int seed)
    {
        Random rnd = new Random(seed);
        for (int ctr = 0; ctr <= 20; ctr++)
            Console.WriteLine(rnd.NextDouble());
    }

    private static void PersistSeed(int seed)
    {
        FileStream fs = new FileStream(@".\seed.dat", FileMode.Create);
        BinaryWriter bin = new BinaryWriter(fs);
        bin.Write(seed);
        bin.Close();
    }

    private static void DisplayNewRandomNumbers()
    {
        FileStream fs = new FileStream(@".\seed.dat", FileMode.Open);
        BinaryReader bin = new BinaryReader(fs);
        int seed = bin.ReadInt32();
        bin.Close();

        Random rnd = new Random(seed);
        for (int ctr = 0; ctr <= 20; ctr++)
            Console.WriteLine(rnd.NextDouble());
    }
}
```

```
// The example displays output like the following:
//       0.500193602172748
//       0.0209461245783354
//       0.465869495396442
//       0.195512794514891
//       0.928583675496552
//       0.729333720509584
//       0.381455668891527
//       0.0508996467343064
//       0.019261200921266
//       0.258578445417145
//       0.0177532266908107
//       0.983277184415272
//       0.483650274334313
//       0.0219647376900375
//       0.165910115077118
//       0.572085966622497
//       0.805291457942357
//       0.927985211335116
//       0.4228545699375
//       0.523320379910674
//       0.157783938645285
//
//       0.500193602172748
//       0.0209461245783354
//       0.465869495396442
//       0.195512794514891
//       0.928583675496552
//       0.729333720509584
//       0.381455668891527
//       0.0508996467343064
//       0.019261200921266
//       0.258578445417145
//       0.0177532266908107
//       0.983277184415272
//       0.483650274334313
//       0.0219647376900375
//       0.165910115077118
//       0.572085966622497
//       0.805291457942357
//       0.927985211335116
//       0.4228545699375
//       0.523320379910674
//       0.157783938645285
```

난수의 고유 시퀀스 검색

클래스의 Random 인스턴스에 서로 다른 시드 값을 제공하면 각 난수 생성기가 서로 다른 값 시퀀스를 생성합니다. 생성자를 호출하여 명시적으로 또는 생성자를 호출 Random(Int32)Random() 하여 암시적으로 시드 값을 제공할 수 있습니다. 대부분의 개발자는 시스템 클록을 사용하는 매개 변수가 없는 생성자를 호출합니다. 다음 예제에서는

이 방법을 사용하여 두 인스턴스 Random 를 인스턴스화합니다. 각 인스턴스는 일련의 10개의 임의 정수를 표시합니다.

C#

```
using System;
using System.Threading;

public class Example16
{
    public static void Main()
    {
        Console.WriteLine("Instantiating two random number generators...");
        Random rnd1 = new Random();
        Thread.Sleep(2000);
        Random rnd2 = new Random();

        Console.WriteLine("\nThe first random number generator:");
        for (int ctr = 1; ctr <= 10; ctr++)
            Console.WriteLine("   {0}", rnd1.Next());

        Console.WriteLine("\nThe second random number generator:");
        for (int ctr = 1; ctr <= 10; ctr++)
            Console.WriteLine("   {0}", rnd2.Next());
    }
}
// The example displays output like the following:
//       Instantiating two random number generators...
//
//       The first random number generator:
//          643164361
//          1606571630
//          1725607587
//          2138048432
//          496874898
//          1969147632
//          2034533749
//          1840964542
//          412380298
//          47518930
//
//       The second random number generator:
//          1251659083
//          1514185439
//          1465798544
//          517841554
//          1821920222
//          195154223
//          1538948391
//          1548375095
//          546062716
//          897797880
```

그러나 시스템 클록은 유한 해상도로 인해 약 15밀리초 미만의 시간 차이를 감지하지 못합니다. 따라서 코드가 .NET Framework에서 오버로드를 호출 Random() 하여 두 Random 개체를 연속으로 인스턴스화하는 경우 실수로 개체에 동일한 시드 값을 제공할 수 있습니다. (.NET Core의 Random 클래스에는 이 제한이 없습니다.) 이전 예제에서 이를 보려면 메서드 호출을 Thread.Sleep 주석으로 처리하고 예제를 다시 컴파일하고 실행합니다.

이러한 일이 발생하지 않도록 하려면 여러 개체가 아닌 단일 Random 개체를 인스턴스화하는 것이 좋습니다. 그러나 Random 스레드로부터 안전하지 않으므로 여러 스레드에서 인스턴스에 액세스하는 경우 일부 동기화 디바이스를 Random 사용해야 합니다. 자세한 내용은 스레드 안전 섹션을 참조하세요. 또는 이전 예제에서 사용된 메서드와 같은 Sleep 지연 메커니즘을 사용하여 인스턴스화가 15밀리초 이상 떨어져 있는지 확인할 수 있습니다.

지정된 범위에서 정수 검색

메서드를 호출 Next(Int32, Int32) 하여 지정된 범위에서 정수를 검색할 수 있습니다. 이를 통해 난수 생성기에서 반환하려는 숫자의 하한과 상한을 모두 지정할 수 있습니다. 상한은 포괄 값이 아니라 배타적입니다. 즉, 메서드에서 반환하는 값 범위에 포함되지 않습니다. 다음 예제에서는 이 메서드를 사용하여 -10에서 10 사이의 임의 정수를 생성합니다. 메서드 호출의 인수 값 `maxValue` 으로 원하는 값보다 큰 11을 지정합니다.

C#

```
Random rnd = new Random();
for (int ctr = 1; ctr <= 15; ctr++)
{
    Console.Write("{0,3}    ", rnd.Next(-10, 11));
    if (ctr % 5 == 0) Console.WriteLine();
}

// The example displays output like the following:
//        -2     -5     -1     -2     10
//        -3      6     -4     -8      3
//        -7     10      5     -2      4
```

지정된 숫자 수의 정수를 검색합니다.

메서드를 Next(Int32, Int32) 호출하여 지정된 숫자 수의 숫자를 검색할 수 있습니다. 예를 들어 4자리 숫자(즉, 1000에서 9999까지의 숫자)를 검색하려면 다음 예제와 `minValue` 같이 값이 1000이고 `maxValue` 값이 10000인 메서드를 호출 Next(Int32, Int32) 합니다.

C#

```
Random rnd = new Random();
for (int ctr = 1; ctr <= 50; ctr++)
{
    Console.Write("{0,3}    ", rnd.Next(1000, 10000));
    if (ctr % 10 == 0) Console.WriteLine();
}

// The example displays output like the following:
//    9570    8979    5770    1606    3818    4735    8495    7196    7070
2313
//    5279    6577    5104    5734    4227    3373    7376    6007    8193
5540
//    7558    3934    3819    7392    1113    7191    6947    4963    9179
7907
//    3391    6667    7269    1838    7317    1981    5154    7377    3297
5320
//    9869    8694    2684    4949    2999    3019    2357    5211    9604
2593
```

지정된 범위에서 부동 소수점 값 검색

이 메서드는 NextDouble 0에서 1 미만까지의 임의 부동 소수점 값을 반환합니다. 그러나 다른 범위에서 임의 값을 생성하려는 경우가 많습니다.

원하는 최소값과 최대값 사이의 간격이 1이면 원하는 시작 간격과 0 사이의 차이를 메서드에서 반환된 수에 NextDouble 추가할 수 있습니다. 다음 예제에서는 -1에서 0 사이의 10개의 난수를 생성하기 위해 이 작업을 수행합니다.

C#

```
Random rnd = new Random();
for (int ctr = 1; ctr <= 10; ctr++)
    Console.WriteLine(rnd.NextDouble() - 1);

// The example displays output like the following:
//       -0.930412760437658
//       -0.164699016215605
//       -0.9851692803135
//       -0.43468508843085
//       -0.177202483255976
//       -0.776813320245972
//       -0.0713201854710096
//       -0.0912875561468711
//       -0.540621722368813
//       -0.232211863730201
```

하한이 0이지만 상한이 1보다 큰 임의의 부동 소수점 숫자를 생성하려면(또는 하한이 -1보다 작고 상한이 0인 음수의 경우) 0이 아닌 바운드를 곱합니다. 다음 예제에서는 이 작

업을 수행하여 0에서 0까지의 2천만 개의 임의 부동 소수점 숫자를 생성합니다 Int64.MaxValue. 또한 메서드에서 생성된 임의 값의 분포도 표시합니다.

C#

```
const long ONE_TENTH = 922337203685477581;

Random rnd = new Random();
double number;
int[] count = new int[10];

// Generate 20 million integer values between.
for (int ctr = 1; ctr <= 20000000; ctr++)
{
    number = rnd.NextDouble() * Int64.MaxValue;
    // Categorize random numbers into 10 groups.
    count[(int)(number / ONE_TENTH)]++;
}
// Display breakdown by range.
Console.WriteLine("{0,28} {1,32}   {2,7}\n", "Range", "Count", "Pct.");
for (int ctr = 0; ctr <= 9; ctr++)
    Console.WriteLine("{0,25:N0}-{1,25:N0}  {2,8:N0}   {3,7:P2}", ctr *
ONE_TENTH,
                      ctr < 9 ? ctr * ONE_TENTH + ONE_TENTH - 1 :
Int64.MaxValue,
                      count[ctr], count[ctr] / 20000000.0);

// The example displays output like the following:
//                            Range                               Count
Pct.
//
//                             0-  922,337,203,685,477,580  1,996,148    9.98
%
//      922,337,203,685,477,581-1,844,674,407,370,955,161  2,000,293   10.00
%
//    1,844,674,407,370,955,162-2,767,011,611,056,432,742  2,000,094   10.00
%
//    2,767,011,611,056,432,743-3,689,348,814,741,910,323  2,000,159   10.00
%
//    3,689,348,814,741,910,324-4,611,686,018,427,387,904  1,999,552   10.00
%
//    4,611,686,018,427,387,905-5,534,023,222,112,865,485  1,998,248    9.99
%
//    5,534,023,222,112,865,486-6,456,360,425,798,343,066  2,000,696   10.00
%
//    6,456,360,425,798,343,067-7,378,697,629,483,820,647  2,001,637   10.01
%
//    7,378,697,629,483,820,648-8,301,034,833,169,298,228  2,002,870   10.01
%
//    8,301,034,833,169,298,229-9,223,372,036,854,775,807  2,000,303   10.00
%
```

메서드가 정수에 대해 수행하는 것처럼 두 임의 값 사이에 임의 Next(Int32, Int32) 부동 소수점 숫자를 생성하려면 다음 수식을 사용합니다.

C#

```
Random.NextDouble() * (maxValue - minValue) + minValue
```

다음 예제에서는 10.0에서 11.0까지의 1백만 개의 난수를 생성하고 해당 분포를 표시합니다.

C#

```
Random rnd = new Random();
int lowerBound = 10;
int upperBound = 11;
int[] range = new int[10];
for (int ctr = 1; ctr <= 1000000; ctr++)
{
    Double value = rnd.NextDouble() * (upperBound - lowerBound) +
lowerBound;
    range[(int)Math.Truncate((value - lowerBound) * 10)]++;
}

for (int ctr = 0; ctr <= 9; ctr++)
{
    Double lowerRange = 10 + ctr * .1;
    Console.WriteLine("{0:N1} to {1:N1}: {2,8:N0}  ({3,7:P2})",
                      lowerRange, lowerRange + .1, range[ctr],
                      range[ctr] / 1000000.0);
}

// The example displays output like the following:
//       10.0 to 10.1:   99,929  ( 9.99 %)
//       10.1 to 10.2:  100,189  (10.02 %)
//       10.2 to 10.3:   99,384  ( 9.94 %)
//       10.3 to 10.4:  100,240  (10.02 %)
//       10.4 to 10.5:   99,397  ( 9.94 %)
//       10.5 to 10.6:  100,580  (10.06 %)
//       10.6 to 10.7:  100,293  (10.03 %)
//       10.7 to 10.8:  100,135  (10.01 %)
//       10.8 to 10.9:   99,905  ( 9.99 %)
//       10.9 to 11.0:   99,948  ( 9.99 %)
```

임의 부울 값 생성

클래스는 Random 값을 생성하는 Boolean 메서드를 제공하지 않습니다. 그러나 이를 위해 고유한 클래스 또는 메서드를 정의할 수 있습니다. 다음 예제에서는 단일 메서드 `NextBoolean`를 사용하여 클래스 `BooleanGenerator`를 정의합니다. 클래스는

`BooleanGenerator` 개체를 Random 프라이빗 변수로 저장합니다. 메서드는 `NextBoolean` 메서드를 Random.Next(Int32, Int32) 호출하고 결과를 메서드에 Convert.ToBoolean(Int32) 전달합니다. 2는 난수의 상한을 지정하는 인수로 사용됩니다. 이 값은 배타적 값이므로 메서드 호출은 0 또는 1을 반환합니다.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        // Instantiate the Boolean generator.
        BooleanGenerator boolGen = new BooleanGenerator();
        int totalTrue = 0, totalFalse = 0;

        // Generate 1,0000 random Booleans, and keep a running total.
        for (int ctr = 0; ctr < 1000000; ctr++)
        {
            bool value = boolGen.NextBoolean();
            if (value)
                totalTrue++;
            else
                totalFalse++;
        }
        Console.WriteLine("Number of true values:  {0,7:N0} ({1:P3})",
                          totalTrue,
                          ((double)totalTrue) / (totalTrue + totalFalse));
        Console.WriteLine("Number of false values: {0,7:N0} ({1:P3})",
                          totalFalse,
                          ((double)totalFalse) / (totalTrue + totalFalse));
    }
}

public class BooleanGenerator
{
    Random rnd;

    public BooleanGenerator()
    {
        rnd = new Random();
    }

    public bool NextBoolean()
    {
        return rnd.Next(0, 2) == 1;
    }
}
// The example displays output like the following:
//       Number of true values:  500,004 (50.000 %)
//       Number of false values: 499,996 (50.000 %)
```

이 예제에서는 임의의 Boolean 값을 생성하는 별도의 클래스를 만드는 대신 단일 메서드를 정의했을 수 있습니다. 그러나 이 경우 각 메서드 호출에서 새 Random 인스턴스를 Random 인스턴스화하지 않도록 개체를 클래스 수준 변수로 정의해야 합니다. Visual Basic에서 임의 인스턴스는 메서드에서 NextBoolean 정적 변수로 정의할 수 있습니다. 다음 예제에서는 구현을 제공합니다.

C#

```
Random rnd = new Random();

int totalTrue = 0, totalFalse = 0;

// Generate 1,000,000 random Booleans, and keep a running total.
for (int ctr = 0; ctr < 1000000; ctr++)
{
    bool value = NextBoolean();
    if (value)
        totalTrue++;
    else
        totalFalse++;
}
Console.WriteLine("Number of true values:  {0,7:N0} ({1:P3})",
                  totalTrue,
                  ((double)totalTrue) / (totalTrue + totalFalse));
Console.WriteLine("Number of false values: {0,7:N0} ({1:P3})",
                  totalFalse,
                  ((double)totalFalse) / (totalTrue + totalFalse));

bool NextBoolean()
{
    return rnd.Next(0, 2) == 1;
}

// The example displays output like the following:
//       Number of true values:  499,777 (49.978 %)
//       Number of false values: 500,223 (50.022 %)
```

임의의 64비트 정수 생성

메서드의 Next 오버로드는 32비트 정수를 반환합니다. 그러나 경우에 따라 64비트 정수로 작업하는 것이 좋습니다. 다음과 같이 이 작업을 수행할 수 있습니다.

1. 메서드를 NextDouble 호출하여 배정밀도 부동 소수점 값을 검색합니다.

2. 해당 값을 곱합니다 Int64.MaxValue.

다음 예제에서는 이 기술을 사용하여 2,000만 개의 임의 정수를 생성하고 10개의 동일한 그룹으로 분류합니다. 그런 다음, 각 그룹의 숫자를 0에서 으로 계산하여 난수의 분포를

평가합니다 Int64.MaxValue. 예제의 출력에서 알 수 있듯이 숫자는 긴 정수 범위를 통해 더 많거나 적게 균등하게 분산됩니다.

C#

```
const long ONE_TENTH = 922337203685477581;

Random rnd = new Random();
long number;
int[] count = new int[10];

// Generate 20 million long integers.
for (int ctr = 1; ctr <= 20000000; ctr++)
{
    number = (long)(rnd.NextDouble() * Int64.MaxValue);
    // Categorize random numbers.
    count[(int)(number / ONE_TENTH)]++;
}
// Display breakdown by range.
Console.WriteLine("{0,28} {1,32}   {2,7}\n", "Range", "Count", "Pct.");
for (int ctr = 0; ctr <= 9; ctr++)
    Console.WriteLine("{0,25:N0}-{1,25:N0}  {2,8:N0}   {3,7:P2}", ctr *
ONE_TENTH,
                      ctr < 9 ? ctr * ONE_TENTH + ONE_TENTH - 1 :
Int64.MaxValue,
                      count[ctr], count[ctr] / 20000000.0);

// The example displays output like the following:
//                            Range                               Count
Pct.
//
//                             0-  922,337,203,685,477,580  1,996,148    9.98
%
//      922,337,203,685,477,581-1,844,674,407,370,955,161  2,000,293   10.00
%
//    1,844,674,407,370,955,162-2,767,011,611,056,432,742  2,000,094   10.00
%
//    2,767,011,611,056,432,743-3,689,348,814,741,910,323  2,000,159   10.00
%
//    3,689,348,814,741,910,324-4,611,686,018,427,387,904  1,999,552   10.00
%
//    4,611,686,018,427,387,905-5,534,023,222,112,865,485  1,998,248    9.99
%
//    5,534,023,222,112,865,486-6,456,360,425,798,343,066  2,000,696   10.00
%
//    6,456,360,425,798,343,067-7,378,697,629,483,820,647  2,001,637   10.01
%
//    7,378,697,629,483,820,648-8,301,034,833,169,298,228  2,002,870   10.01
%
//    8,301,034,833,169,298,229-9,223,372,036,854,775,807  2,000,303   10.00
%
```

비트 조작을 사용하는 대체 기술은 실제로 난수를 생성하지 않습니다. 이 기술은 두 개의 정수, 왼쪽 시프트 1-32비트 및 OU를 함께 생성하도록 호출 Next() 합니다. 이 기술에는 두 가지 제한 사항이 있습니다.

1. 비트 31은 부호 비트이므로 결과 정수의 비트 31 값은 항상 0입니다. 이 문제는 임의의 0 또는 1을 생성하고, 31비트를 왼쪽으로 이동하고, 원래 임의 긴 정수로 ORing하여 해결할 수 있습니다.

2. 더 심각하게, 반환 Next() 되는 값이 0이 될 확률이기 때문에 범위 0x0-0x00000000FFFFFFFF 임의의 숫자가 있으면 거의 없을 것입니다.

지정된 범위에서 바이트 검색

메서드의 Next 오버로드를 사용하면 난수 범위를 지정할 수 있지만 메서드는 NextBytes 지정하지 않습니다. 다음 예제에서는 반환된 `NextBytes` 바이트 범위를 지정할 수 있는 메서드를 구현합니다. 메서드에서 Random 파생되고 오버로드되는 클래스를 정의 `Random2` 합니다`NextBytes`.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        Random2 rnd = new Random2();
        Byte[] bytes = new Byte[10000];
        int[] total = new int[101];
        rnd.NextBytes(bytes, 0, 101);

        // Calculate how many of each value we have.
        foreach (var value in bytes)
            total[value]++;

        // Display the results.
        for (int ctr = 0; ctr < total.Length; ctr++)
        {
            Console.Write("{0,3}: {1,-3}   ", ctr, total[ctr]);
            if ((ctr + 1) % 5 == 0) Console.WriteLine();
        }
    }
}

public class Random2 : Random
{
    public Random2() : base()
    { }
```

```
    public Random2(int seed) : base(seed)
    { }

    public void NextBytes(byte[] bytes, byte minValue, byte maxValue)
    {
        for (int ctr = bytes.GetLowerBound(0); ctr <=
bytes.GetUpperBound(0); ctr++)
            bytes[ctr] = (byte)Next(minValue, maxValue);
    }
}
// The example displays output like the following:
//         0: 115     1: 119     2: 92      3: 98      4: 92
//         5: 102     6: 103     7: 84      8: 93      9: 116
//        10: 91     11: 98     12: 106    13: 91     14: 92
//        15: 101    16: 100    17: 96     18: 97     19: 100
//        20: 101    21: 106    22: 112    23: 82     24: 85
//        25: 102    26: 107    27: 98     28: 106    29: 102
//        30: 109    31: 108    32: 94     33: 101    34: 107
//        35: 101    36: 86     37: 100    38: 101    39: 102
//        40: 113    41: 95     42: 96     43: 89     44: 99
//        45: 81     46: 89     47: 105    48: 100    49: 85
//        50: 103    51: 103    52: 93     53: 89     54: 91
//        55: 97     56: 105    57: 97     58: 110    59: 86
//        60: 116    61: 94     62: 117    63: 98     64: 110
//        65: 93     66: 102    67: 100    68: 105    69: 83
//        70: 81     71: 97     72: 85     73: 70     74: 98
//        75: 100    76: 110    77: 114    78: 83     79: 90
//        80: 96     81: 112    82: 102    83: 102    84: 99
//        85: 81     86: 100    87: 93     88: 99     89: 118
//        90: 95     91: 124    92: 108    93: 96     94: 104
//        95: 106    96: 99     97: 99     98: 92     99: 99
//       100: 108
```

메서드는 `NextBytes(Byte[], Byte, Byte)` 메서드 호출을 Next(Int32, Int32) 래핑하고 바이트 배열에서 반환하려는 최소값과 최대값(이 경우 0 및 101)보다 큰 값을 지정합니다. 메서드에서 반환 Next 된 정수 값이 데이터 형식의 Byte 범위 내에 있다고 확신하므로 안전하게 캐스팅하거나(C# 및 F#) 정수에서 바이트로 변환할 수 있습니다(Visual Basic의 경우).

배열 또는 컬렉션에서 임의로 요소 검색

난수는 배열 또는 컬렉션에서 값을 검색하는 인덱스 역할을 하는 경우가 많습니다. 임의 인덱스 값을 검색하려면 메서드를 Next(Int32, Int32) 호출하고 배열의 하한을 인수 `minValue` 값으로 사용하고 배열의 상한보다 큰 값을 인수 값 `maxValue` 으로 사용할 수 있습니다. 0부터 시작하는 배열의 경우 해당 속성과 동일 Length 하거나 메서드에서 반환한 Array.GetUpperBound 값보다 큰 배열입니다. 다음 예제에서는 도시 배열에서 미국 도시 이름을 임의로 검색합니다.

C#

```
String[] cities = { "Atlanta", "Boston", "Chicago", "Detroit",
                    "Fort Wayne", "Greensboro", "Honolulu", "Indianapolis",
                    "Jersey City", "Kansas City", "Los Angeles",
                    "Milwaukee", "New York", "Omaha", "Philadelphia",
                    "Raleigh", "San Francisco", "Tulsa", "Washington" };
Random rnd = new Random();
int index = rnd.Next(0, cities.Length);
Console.WriteLine("Today's city of the day: {0}",
                  cities[index]);

// The example displays output like the following:
//   Today's city of the day: Honolulu
```

배열 또는 컬렉션에서 고유 요소 검색

난수 생성기는 항상 중복 값을 반환할 수 있습니다. 숫자 범위가 작아지거나 생성된 값 수가 커지면 중복 가능성이 커집니다. 임의 값이 고유해야 하는 경우 중복을 보정하기 위해 더 많은 숫자가 생성되어 성능이 점점 저하됩니다.

이 시나리오를 처리하는 여러 가지 기술이 있습니다. 한 가지 일반적인 해결 방법은 검색할 값이 포함된 배열 또는 컬렉션과 임의 부동 소수점 숫자가 포함된 병렬 배열을 만드는 것입니다. 두 번째 배열은 첫 번째 배열을 만들 때 난수로 채워지고 Array.Sort(Array, Array) 메서드는 병렬 배열의 값을 사용하여 첫 번째 배열을 정렬하는 데 사용됩니다.

예를 들어 솔리테어 게임을 개발하는 경우 각 카드 한 번만 사용되는지 확인하려고 합니다. 카드 검색하기 위해 난수를 생성하고 해당 카드 이미 처리되었는지 여부를 추적하는 대신 데크를 정렬하는 데 사용할 수 있는 난수의 병렬 배열을 만들 수 있습니다. 데크가 정렬되면 앱에서 포인터를 기본 데크에서 다음 카드 인덱스로 나타낼 수 있습니다.

다음 예제에서 이 방법을 보여 줍니다. 재생 카드 나타내는 클래스와 `Dealer` 순서가 섞인 카드 데크를 처리하는 클래스를 정의 `Card` 합니다. `Dealer` 클래스 생성자는 클래스 범위가 있고 데크의 모든 카드 나타내는 배열과 배열과 동일한 수의 요소를 가지며 임의로 생성된 Double 값으로 `deck` 채워진 로컬 `order` 배열의 두 `deck` 배열을 채웁니다. Array.Sort(Array, Array) 그런 다음 배열의 값을 기준으로 배열을 `deck` 정렬하기 위해 메서드를 호출합니다`order`.

C#

```
using System;

// A class that represents an individual card in a playing deck.
public class Card
{
```

```
    public Suit Suit;
    public FaceValue FaceValue;

    public override String ToString()
    {
        return String.Format("{0:F} of {1:F}", this.FaceValue, this.Suit);
    }
}

public enum Suit { Hearts, Diamonds, Spades, Clubs };

public enum FaceValue
{
    Ace = 1, Two, Three, Four, Five, Six,
    Seven, Eight, Nine, Ten, Jack, Queen,
    King
};

public class Dealer
{
    Random rnd;
    // A deck of cards, without Jokers.
    Card[] deck = new Card[52];
    // Parallel array for sorting cards.
    Double[] order = new Double[52];
    // A pointer to the next card to deal.
    int ptr = 0;
    // A flag to indicate the deck is used.
    bool mustReshuffle = false;

    public Dealer()
    {
        rnd = new Random();
        // Initialize the deck.
        int deckCtr = 0;
        foreach (var suit in Enum.GetValues(typeof(Suit)))
        {
            foreach (var faceValue in Enum.GetValues(typeof(FaceValue)))
            {
                Card card = new Card();
                card.Suit = (Suit)suit;
                card.FaceValue = (FaceValue)faceValue;
                deck[deckCtr] = card;
                deckCtr++;
            }
        }

        for (int ctr = 0; ctr < order.Length; ctr++)
            order[ctr] = rnd.NextDouble();

        Array.Sort(order, deck);
    }

    public Card[] Deal(int numberToDeal)
    {
```

```
        if (mustReshuffle)
        {
            Console.WriteLine("There are no cards left in the deck");
            return null;
        }

        Card[] cardsDealt = new Card[numberToDeal];
        for (int ctr = 0; ctr < numberToDeal; ctr++)
        {
            cardsDealt[ctr] = deck[ptr];
            ptr++;
            if (ptr == deck.Length)
                mustReshuffle = true;

            if (mustReshuffle & ctr < numberToDeal - 1)
            {
                Console.WriteLine("Can only deal the {0} cards remaining on
the deck.",
                                  ctr + 1);
                return cardsDealt;
            }
        }
        return cardsDealt;
    }
}

public class Example17
{
    public static void Main()
    {
        Dealer dealer = new Dealer();
        ShowCards(dealer.Deal(20));
    }

    private static void ShowCards(Card[] cards)
    {
        foreach (var card in cards)
            if (card != null)
                Console.WriteLine("{0} of {1}", card.FaceValue, card.Suit);
    }
}
// The example displays output like the following:
//       Six of Diamonds
//       King of Clubs
//       Eight of Clubs
//       Seven of Clubs
//       Queen of Clubs
//       King of Hearts
//       Three of Spades
//       Ace of Clubs
//       Four of Hearts
//       Three of Diamonds
//       Nine of Diamonds
//       Two of Hearts
//       Ace of Hearts
```

```
//        Three of Hearts
//        Four of Spades
//        Eight of Hearts
//        Queen of Diamonds
//        Two of Clubs
//        Four of Diamonds
//        Jack of Hearts
```

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

Microsoft.Extensions.AI 라이브러리

.NET 개발자는 앱에서 점점 더 다양한 AI(인공 지능) 서비스를 통합하고 상호 작용해야 합니다. 라이브러리는 `Microsoft.Extensions.AI` 생성 AI 구성 요소를 나타내기 위한 통합된 접근 방식을 제공하고 다양한 AI 서비스와의 원활한 통합 및 상호 운용성을 가능하게 합니다. 이 문서에서는 라이브러리를 소개하고 시작하는 데 도움이 되는 자세한 사용 예제를 제공합니다.

패키지

Microsoft.Extensions.AI.Abstractions 패키지는 다음을 비롯한 IChatClientIEmbeddingGenerator<TInput,TEmbedding>핵심 교환 유형을 제공합니다. LLM 클라이언트를 제공하는 모든 .NET 라이브러리는 인터페이스를 `IChatClient` 구현하여 소비 코드와 원활하게 통합할 수 있습니다.

Microsoft.Extensions.AI 패키지에는 `Microsoft.Extensions.AI.Abstractions` 패키지에 대한 암시적 종속성이 있습니다. 이 패키지를 사용하면 친숙한 종속성 주입 및 미들웨어 패턴을 사용하여 자동 함수 도구 호출, 원격 분석 및 캐싱과 같은 구성 요소를 애플리케이션에 쉽게 통합할 수 있습니다. 예를 들어 채팅 클라이언트 파이프라인에 OpenTelemetry 지원을 추가하는 UseOpenTelemetry(ChatClientBuilder, ILoggerFactory, String, Action<OpenTelemetryChatClient>) 확장 메서드를 제공합니다.

참조할 패키지

생성 AI 구성 요소로 작업하기 위한 상위 수준의 유틸리티에 액세스하려면 `Microsoft.Extensions.AI` 패키지를 참조하세요(더 나은 대안으로, `Microsoft.Extensions.AI`는 를 참조합니다). 대부분의 소비 애플리케이션 및 서비스는 추상화의 `Microsoft.Extensions.AI` 구체적인 구현을 제공하는 하나 이상의 라이브러리와 함께 패키지를 참조해야 합니다.

추상화 구현을 제공하는 라이브러리는 일반적으로 `Microsoft.Extensions.AI.Abstractions`만 참조합니다.

패키지 설치

NuGet 패키지를 설치하는 방법에 대한 자세한 내용은 .NET 애플리케이션에서 dotnet 패키지 추가 또는 패키지 종속성 관리를 참조하세요.

API 및 기능

- IChatClient 인터페이스
- IEmbeddingGenerator 인터페이스

- IImageGenerator 인터페이스(실험적)

`IChatClient` 인터페이스

IChatClient 인터페이스는 채팅 기능을 제공하는 AI 서비스와 상호 작용하는 클라이언트 추상화 작업을 정의합니다. 이 방법에는 텍스트, 이미지 및 오디오와 같은 다중 모달 콘텐츠를 사용하여 메시지를 보내고 받는 기능이 포함되며, 이는 전체 집합으로 또는 점진적 스트리밍으로 이루어질 수 있습니다.

자세한 내용 및 자세한 사용 예제는 IChatClient 인터페이스 사용을 참조하세요.

`IEmbeddingGenerator` 인터페이스

IEmbeddingGenerator 인터페이스는 임베딩의 일반적인 생성기를 나타냅니다. 제네릭 타입 매개변수의 경우, `TInput`는 포함되는 입력 값의 타입이고, `TEmbedding`는 Embedding 클래스를 상속받는 생성된 포함의 타입입니다.

자세한 내용 및 자세한 사용 예제는 IEmbeddingGenerator 인터페이스 사용을 참조하세요.

IImageGenerator 인터페이스(실험적)

인터페이스는 IImageGenerator 텍스트 프롬프트 또는 기타 입력에서 이미지를 만들기 위한 생성기를 나타냅니다. 이 인터페이스를 사용하면 애플리케이션이 일관된 API를 통해 다양한 AI 서비스의 이미지 생성 기능을 통합할 수 있습니다. 인터페이스는 텍스트-이미지 생성(호출 GenerateAsync(ImageGenerationRequest, ImageGenerationOptions, CancellationToken))과 이미지 크기 및 형식에 대한 구성 옵션을 지원합니다. 라이브러리의 다른 인터페이스와 마찬가지로 캐싱, 원격 분석 및 기타 교차 절단 문제를 위한 미들웨어로 구성할 수 있습니다.

자세한 내용은 AI를 사용하여 텍스트에서 이미지 생성을 참조하세요.

Microsoft.Extensions.AI 사용하여 빌드

다음과 같은 방법으로 `Microsoft.Extensions.AI`을(를) 빌드하는 것을 시작할 수 있습니다.

- **라이브러리 개발자**: AI 서비스에 대한 클라이언트를 제공하는 라이브러리를 소유한 경우 라이브러리에서 인터페이스를 구현하는 것이 좋습니다. 이를 통해 사용자는 추상화로 NuGet 패키지를 쉽게 통합할 수 있습니다. 예제는 IChatClient 구현 예제 및 IEmbeddingGenerator 구현 예제를 참조하세요.
- **서비스 소비자**: AI 서비스를 사용하는 라이브러리를 개발하는 경우 특정 AI 서비스에 대한 하드코딩 대신 추상화 기능을 사용합니다. 이 접근 방식을 통해 소비자는 선호하는 공급자를 유연하게 선택할 수 있습니다.

- **애플리케이션 개발자**: 추상화로 앱 통합을 간소화합니다. 이렇게 하면 모델 및 서비스 전반에서 이식성을 구현하고, 테스트 및 모의 작업을 용이하게 하고, 에코시스템에서 제공하는 미들웨어를 활용하며, 애플리케이션의 여러 부분에서 다른 서비스를 사용하더라도 앱 전체에서 일관된 API를 유지 관리합니다.
- **에코시스템 기여자는**: 에코시스템에 기여하려는 경우 사용자 지정 미들웨어 구성 요소를 작성하는 것이 좋습니다.

자세한 샘플은 dotnet/ai-samples GitHub 리포지토리를 참조하세요. 엔드 투 엔드 샘플은 eShopSupport를 참조하세요.

참고하십시오

- 구조적 출력을 사용하여 응답 요청
- .NET을 사용하여 AI 채팅 앱 빌드
- .NET에서 종속성 주입
- .NET에서의 캐싱
- .NET에서 HTTP 처리기 속도 제한

Last updated on 2025. 12. 12.

.NET 종속성 주입

.NET은 클래스와 해당 *종속성* 간에 IoC(Inversion of Control) 를 달성하기 위한 기술인 DI(종속성 주입) 소프트웨어 디자인 패턴을 지원합니다. .NET의 종속성 주입은 구성, 로깅 및 옵션 패턴과 함께 프레임워크에 기본 제공된 부분입니다.

'종속성'은 다른 개체가 종속된 개체입니다. 다음 `MessageWriter` 클래스에는 `Write` 다른 클래스가 의존할 수 있는 메서드가 있습니다.

C#

```
public class MessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
    }
}
```

클래스는 해당 `MessageWriter` 메서드를 사용할 클래스의 `Write` 인스턴스를 만들 수 있습니다. 다음 예제에서, `MessageWriter` 클래스는 클래스의 `Worker`입니다.

C#

```
public class Worker : BackgroundService
{
    private readonly MessageWriter _messageWriter = new();

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

이 경우, `Worker` 클래스는 `MessageWriter` 클래스를 만들고 직접 의존합니다. 이와 같은 하드 코딩된 종속성은 문제가 되며 다음과 같은 이유로 피해야 합니다.

- 다른 구현으로 `MessageWriter`을 바꾸기 위해, `Worker` 클래스를 수정해야 합니다.
- 종속성을 포함하는 경우, `MessageWriter` 클래스도 그것들을 구성해야 합니다. 여러 클래스가 `MessageWriter`에 종속되어 있는 대형 프로젝트에서는 구성 코드가 앱 전체에 분산됩니다.
- 이 구현은 단위 테스트하기가 어렵습니다. 앱에서 모의 또는 스텁 `MessageWriter` 클래스를 사용해야 하지만, 이 방법에서는 가능하지 않습니다.

개념

종속성 주입은 다음을 통해 하드 코딩된 종속성 문제를 해결합니다.

- 인퍼테이스 또는 기본 클래스를 사용하여 종속성 구현을 추상화합니다.
- *서비스 컨테이너*의 종속성 등록

  .NET는 서비스 컨테이너인 IServiceProvider를 기본 제공합니다. 서비스는 일반적으로 앱 시작 시 등록되고 IServiceCollection에 추가됩니다. 모든 서비스가 추가되면 서비스 컨테이너를 만드는 데 사용합니다 BuildServiceProvider .
- 서비스가 사용되는 클래스의 생성자에 서비스를 삽입합니다.

  프레임워크가 종속성의 인스턴스를 만들고 더 이상 필요하지 않으면 삭제하는 작업을 담당합니다.

> **팁**
>
> 종속성 주입 용어에서는 *서비스*가 일반적으로 `IMessageWriter` 같은 다른 개체에게 서비스를 제공하는 개체를 의미합니다. 이 서비스는 웹 서비스를 사용할 수 있지만 웹 서비스와는 관련이 없습니다.

예를 들어, `IMessageWriter` 인터페이스가 `Write` 메서드를 정의하고 있다고 가정합니다. 이 인터페이스는 이전에 표시된 구체적인 형식 `MessageWriter`으로 구현됩니다. 다음 샘플 코드는 구체적인 형식`IMessageWriter`으로 `MessageWriter` 서비스를 등록합니다. 이 메서드는 AddSingleton 싱글톤 *수명*을 사용하여 서비스를 등록합니다. 즉, 앱이 종료될 때까지 처리되지 않습니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHostedService<Worker>();
builder.Services.AddSingleton<IMessageWriter, MessageWriter>();

using IHost host = builder.Build();

host.Run();

// <SnippetMW>
public class MessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
    }
}
```

```
// </SnippetMW>

// <SnippetIMW>
public interface IMessageWriter
{
    void Write(string message);
}
// </SnippetIMW>

// <SnippetWorker>
public sealed class Worker(IMessageWriter messageWriter) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
            await Task.Delay(1_000, stoppingToken);
        }
    }
}

// </SnippetWorker>
```

앞의 코드 예제에서 강조 표시된 줄은 다음과 같습니다.

- 호스트 앱 작성기 인스턴스를 만듭니다.
- `Worker`로 등록하고 인터페이스를 `IMessageWriter` 클래스의 해당 구현을 사용하여 싱글톤 서비스로 등록하여 서비스를 구성합니다`MessageWriter`.
- 호스트를 빌드하고 실행합니다.

호스트에는 종속성 주입 서비스 공급자가 포함되어 있습니다. 또한 `Worker`를 자동으로 인스턴스화하고 해당 `IMessageWriter` 구현을 인수로 제공하는 데 필요한 기타 모든 관련 서비스도 포함되어 있습니다.

DI 패턴을 사용하면 작업자 서비스는 구체적인 형식 `MessageWriter`을 사용하지 않고 구현하는 인터페이스만 `IMessageWriter` 사용합니다. 이 설계를 사용하면 작업자 서비스를 수정하지 않고도 작업자 서비스에서 사용하는 구현을 쉽게 변경할 수 있습니다. 작업자 서비스 또한 *인스턴스를 생성하지*`MessageWriter`. DI 컨테이너는 인스턴스를 만듭니다.

이제 `MessageWriter`를 프레임워크 제공 로깅 서비스를 사용하는 형식으로 교체하려고 합니다. `LoggingMessageWriter` 클래스는 생성자에서 ILogger<TCategoryName>를 요청하여 의존하도록 만드세요.

C#

```
public class LoggingMessageWriter(
    ILogger<LoggingMessageWriter> logger) : IMessageWriter
```

```
{
    public void Write(string message) =>
        logger.LogInformation("Info: {Msg}", message);
}
```

`MessageWriter`에서 `LoggingMessageWriter`로 전환하려면, 이 새 `AddSingleton` 구현을 등록하기 위해 호출을 `IMessageWriter`로 업데이트합니다.

C#

```
builder.Services.AddSingleton<IMessageWriter, LoggingMessageWriter>();
```

> **팁**
>
> 컨테이너는 `ILogger<TCategoryName>`을 활용하여 해결 하므로 생성된 모든(제네릭) 형식을 등록할 필요가 없습니다.

생성자 주입 동작

IServiceProvider 또는 기본 제공 서비스 컨테이너 ActivatorUtilities를 사용하여 서비스를 해결할 수 있습니다. `ActivatorUtilities` 는 컨테이너에 등록되지 않고 일부 프레임워크 기능과 함께 사용되는 개체를 만듭니다.

생성자에는 종속성 주입으로 제공되지 않는 인수를 사용할 수 있지만, 인수에 기본값을 할당해야 합니다.

`IServiceProvider` 또는 `ActivatorUtilities`이(가) 서비스를 해결할 때, 생성자 주입에는 *공용* 생성자가 필요합니다.

서비스를 해결할 때 `ActivatorUtilities`에서 생성자 주입은 적용 가능한 생성자가 하나만 존재해야 합니다. 생성자 오버로드가 지원되지만, 해당 인수가 모두 종속성 주입으로 처리될 수 있는 하나의 오버로드만 존재할 수 있습니다.

생성자 선택 규칙

형식에서 둘 이상의 생성자를 정의하는 경우 서비스 공급자는 사용할 생성자를 결정하는 논리를 포함합니다. 형식의 DI 확인이 가능한 대부분의 매개 변수가 있는 생성자가 선택됩니다. 다음 예제 서비스를 고려합니다.

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // ...
    }

    public ExampleService(ServiceA serviceA, ServiceB serviceB)
    {
        // ...
    }
}
```

이전 코드에서는 로깅이 추가되었으며 서비스 공급자에서 확인할 수 있지만 `ServiceA` 형식 및 `ServiceB` 형식은 그렇지 않다고 가정합니다. 매개변수를 가진 `ILogger<ExampleService>` 생성자가 `ExampleService` 인스턴스를 해결합니다. 더 많은 매개 변수를 정의하는 생성자가 있더라도 `ServiceA` 및 `ServiceB` 형식은 DI에서 해결할 수 없습니다.

생성자를 발견할 때 모호성이 있으면 예외가 throw됩니다. 다음 C# 서비스 예를 살펴보세요.

> ⚠ **Warning**
>
> 모호한 DI에서 해석 가능한 형식 매개 변수가 포함된 이 `ExampleService` 코드는 예외를 발생시킵니다. 이렇게 **하지 마세요**. "모호한 DI 확인 가능 형식"의 의미를 표시하기 위한 것입니다.

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // ...
    }

    public ExampleService(IOptions<ExampleOptions> options)
    {
        // ...
    }
}
```

위 예제에는 세 가지 생성자가 있습니다. 첫 번째 생성자는 매개 변수가 없으며 서비스 공급자의 서비스가 필요하지 않습니다. 로깅 및 옵션이 모두 DI 컨테이너에 추가되었고 DI 확인 가능 서비스라고 가정합니다. DI 컨테이너가 `ExampleService` 유형을 해석하려고 시도할 때, 두 생성자가 모호하기 때문에 예외가 발생합니다.

대신 두 DI 확인 가능한 형식을 모두 허용하는 생성자를 정의하여 모호성을 방지합니다.

C#

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(
        ILogger<ExampleService> logger,
        IOptions<ExampleOptions> options)
    {
        // ...
    }
}
```

범위 유효성 검사

범위가 지정된 서비스는 해당 서비스를 만든 컨테이너에 의해 삭제됩니다. 범위가 지정된 서비스가 루트 컨테이너에 만들어지면 앱이 종료될 때 루트 컨테이너에서만 삭제되므로 서비스의 수명이 효과적으로 싱글톤 으로 승격됩니다. 서비스 범위의 유효성 검사는 `BuildServiceProvider`가 호출될 경우 이러한 상황을 감지합니다.

앱이 개발 환경에서 실행되고 CreateApplicationBuilder 를 호출하여 호스트를 빌드하는 경우 기본 서비스 공급자는 다음을 확인하는 검사를 수행합니다.

- 범위가 지정된 서비스는 루트 서비스 공급자에서 확인되지 않습니다.
- 범위가 지정된 서비스는 싱글톤에 삽입되지 않습니다.

범위 시나리오

IServiceScopeFactory는 항상 singleton으로 등록되지만, IServiceProvider는 포함하는 클래스의 수명에 따라 달라질 수 있습니다. 예를 들어 범위에서 서비스를 해결하고 해당 서비스가 IServiceProvider를 사용하는 경우, 그것은 범위 내 인스턴스입니다.

와 같은 IHostedService구현 BackgroundService내에서 범위 지정 서비스를 달성하려면 생성자 주입을 통해 서비스 종속성을 삽입*하지 마세요*. 대신 IServiceScopeFactory를 주입하고 범위를 만든 후 범위의 종속성을 확인하여 적절한 서비스 수명을 사용하도록 합니다.

C#

```
namespace WorkerScope.Example;

public sealed class Worker(
    ILogger<Worker> logger,
    IServiceScopeFactory serviceScopeFactory)
    : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            using (IServiceScope scope = serviceScopeFactory.CreateScope())
            {
                try
                {
                    logger.LogInformation(
                        "Starting scoped work, provider hash: {hash}.",
                        scope.ServiceProvider.GetHashCode());

                    var store =
scope.ServiceProvider.GetRequiredService<IObjectStore>();
                    var next = await store.GetNextAsync();
                    logger.LogInformation("{next}", next);

                    var processor =
scope.ServiceProvider.GetRequiredService<IObjectProcessor>();
                    await processor.ProcessAsync(next);
                    logger.LogInformation("Processing {name}.", next.Name);

                    var relay =
scope.ServiceProvider.GetRequiredService<IObjectRelay>();
                    await relay.RelayAsync(next);
                    logger.LogInformation("Processed results have been relayed.");

                    var marked = await store.MarkAsync(next);
                    logger.LogInformation("Marked as processed: {next}", marked);
                }
                finally
                {
                    logger.LogInformation(
                        "Finished scoped work, provider hash: {hash}.{nl}",
                        scope.ServiceProvider.GetHashCode(), Environment.NewLine);
                }
            }
        }
    }
}
```

위의 코드에서 앱이 실행되는 동안 백그라운드 서비스는 다음과 같습니다.

- IServiceScopeFactory에 종속됩니다.

- IServiceScope 다른 서비스를 해결하기 위한 항목을 만듭니다.
- 사용할 범위가 지정된 서비스를 확인합니다.
- 개체 처리에 대한 작업을 수행한 후 해당 개체를 릴레이하고 마지막으로 해당 개체를 처리된 것으로 표시합니다.

샘플 소스 코드에서 IHostedService 구현이 범위가 지정된 서비스 수명의 이점을 활용하는 방법을 확인할 수 있습니다.

키 지정된 서비스

서비스를 등록하고 키에 따라 조회를 수행할 수 있습니다. 즉, 여러 서비스를 다른 키로 등록하고 조회에 이 키를 사용할 수 있습니다.

예를 들어, 인터페이스 `IMessageWriter`(`MemoryMessageWriter` 및 `QueueMessageWriter`)의 서로 다른 구현이 있는 경우를 생각해 보세요.

키를 매개 변수로 지원하는 서비스 등록 메서드(앞서 본)의 오버로드를 사용하여 이러한 서비스를 등록할 수 있습니다.

C#

```
services.AddKeyedSingleton<IMessageWriter, MemoryMessageWriter>("memory");
services.AddKeyedSingleton<IMessageWriter, QueueMessageWriter>("queue");
```

`key`는 `string`에만 제한되지 않습니다. 원하는 `key`을 사용할 수 있으며, 단 `object`가 `Equals`를 올바르게 구현해야 합니다.

`IMessageWriter`를 사용하는 클래스의 생성자에서 FromKeyedServicesAttribute를 추가하여 확인할 서비스의 키를 지정합니다.

C#

```
public class ExampleService
{
    public ExampleService(
        [FromKeyedServices("queue")] IMessageWriter writer)
    {
        // Omitted for brevity...
    }
}
```

KeyedService.AnyKey 속성

이 속성은 KeyedService.AnyKey 키 지정된 서비스를 사용하기 위한 특수 키를 제공합니다. 모든 키와 일치하는 대체(fallback)로 사용하여 `KeyedService.AnyKey` 서비스를 등록할 수 있습니다. 명시적 등록이 없는 키에 대한 기본 구현을 제공하려는 경우에 유용합니다.

C#

```
var services = new ServiceCollection();

// Register a fallback cache for any key.
services.AddKeyedSingleton<ICache>(KeyedService.AnyKey, (sp, key) =>
{
    // Create a cache instance based on the key.
    return new DefaultCache(key?.ToString() ?? "unknown");
});

// Register a specific cache for the "premium" key.
services.AddKeyedSingleton<ICache>("premium", new PremiumCache());

var provider = services.BuildServiceProvider();

// Requesting with "premium" key returns PremiumCache.
var premiumCache = provider.GetKeyedService<ICache>("premium");
Console.WriteLine($"Premium key: {premiumCache}");

// Requesting with any other key uses the AnyKey fallback.
var basicCache = provider.GetKeyedService<ICache>("basic");
Console.WriteLine($"Basic key: {basicCache}");

var standardCache = provider.GetKeyedService<ICache>("standard");
Console.WriteLine($"Standard key: {standardCache}");
```

앞의 예에서:

- 키 `ICache` 로 요청하면 `"premium"` 인스턴스가 반환됩니다`PremiumCache`.
- `ICache` 및 다른 키(예: `"basic"` 또는 `"standard"`)로 요청하면 `DefaultCache` 대체를 사용하여 `AnyKey` 새로 생성됩니다.

> ⓘ **Important**
>
> .NET 10부터 `GetKeyedService()`를 `KeyedService.AnyKey`와 함께 호출하면 InvalidOperationException 예외가 발생합니다. 왜냐하면 `AnyKey`는 등록 대체(fallback)로 의도된 것이지 쿼리 키가 아니기 때문입니다. 자세한 내용은 AnyKey를 사용하여 GetKeyedService() 및 GetKeyedServices()의 문제 해결을 참조하세요.

참고하십시오

- 빠른 시작: 종속성 주입 기본 사항
- 자습서: .NET에서 종속성 주입 사용
- 종속성 주입 지침
- ASP.NET Core에서 종속성 주입
- DI 앱 개발을 위한 NDC 컨퍼런스 패턴
- 명시적 종속성 원칙
- Inversion of Control 컨테이너 및 종속성 주입 패턴(Martin Fowler)

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 24.

서비스 등록

이 문서에서는 서비스 및 프레임워크 제공 서비스의 등록 그룹에 대해 설명합니다. 또한 .NET에서 제공하는 서비스 등록 확장 메서드에 대한 세부 정보도 제공합니다.

확장 메서드를 사용하여 서비스 그룹 등록

.NET은 관련 서비스 그룹을 등록하는 규칙을 사용합니다. 규칙은 단일 `Add{GROUP_NAME}` 확장 메서드를 사용하여 프레임워크 기능에 필요한 모든 서비스를 등록하는 것입니다. 예를 들어 AddOptions 확장 메서드는 옵션을 사용하는 데 필요한 모든 서비스를 등록합니다.

프레임워크 제공 서비스

사용 가능한 호스트 또는 앱 작성기 패턴을 사용하는 경우 기본값이 적용되고 프레임워크에서 서비스를 등록합니다. 가장 인기 있는 호스트 및 앱 작성기 패턴 중 일부를 고려합니다.

- Host.CreateDefaultBuilder()
- Host.CreateApplicationBuilder()
- WebHost.CreateDefaultBuilder()
- WebApplication.CreateBuilder()
- WebAssemblyHostBuilder.CreateDefault
- MauiApp.CreateBuilder

이러한 API `IServiceCollection` 중 하나에서 작성기를 만든 후 호스트를 구성한 방법에 따라 프레임워크에서 정의된 서비스가 있습니다. .NET 템플릿 기반 앱의 경우 프레임워크는 수백 개의 서비스를 등록할 수 있습니다.

다음 표에는 프레임워크에서 등록한 서비스들의 작은 샘플이 나열되어 있습니다.

테이블 확장

서비스 유형	평생
Microsoft.Extensions.DependencyInjection.IServiceScopeFactory	Singleton
IHostApplicationLifetime	Singleton
Microsoft.Extensions.Logging.ILogger<TCategoryName>	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	Transient

서비스 유형	평생
Microsoft.Extensions.Options.IOptions<TOptions>	Singleton
System.Diagnostics.DiagnosticListener	Singleton
System.Diagnostics.DiagnosticSource	Singleton

등록 방법

프레임워크는 특정 시나리오에서 유용한 서비스 등록 확장 메서드를 제공합니다.

테이블 확장

메서드	자동 개체 삭제	여러 구현	인수 전달

`Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()`
 	
예제:
 	
`services.AddSingleton<IMyDep, MyDep>();`
	Yes	Yes	아니오

`Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})`
 	
예제:
 	
`services.AddSingleton<IMyDep>(sp => new MyDep());`
`services.AddSingleton<IMyDep>(sp => new MyDep(99));`
	Yes	Yes	Yes

`Add{LIFETIME}<{IMPLEMENTATION}>()`
 	
예제:
 	
`services.AddSingleton<MyDep>();`
	Yes	아니오	아니오

`AddSingleton<{SERVICE}>(new {IMPLEMENTATION})`
 	
예제:
 	
`services.AddSingleton<IMyDep>(new MyDep());`
`services.AddSingleton<IMyDep>(new MyDep(99));`
	아니오	Yes	Yes

`AddSingleton(new {IMPLEMENTATION})` 예제:	아니오	아니오	Yes

메서드	자동 개체 삭제	여러 구현	인수 전달
`services.AddSingleton(new MyDep());` `services.AddSingleton(new MyDep(99));`			

형식 삭제에 대한 자세한 내용은 서비스 삭제를 참조하세요.

구현 형식만으로 서비스를 등록하는 것은 동일한 구현 및 서비스 형식으로 해당 서비스를 등록하는 것과 같습니다. 예를 들어 다음 코드를 고려합니다.

C#

```
services.AddSingleton<ExampleService>();
```

이는 동일한 형식의 서비스 및 구현에 서비스를 등록하는 것과 같습니다.

C#

```
services.AddSingleton<ExampleService, ExampleService>();
```

이러한 동등성 때문에 명시적 서비스 형식을 사용하지 않는 메서드를 사용하여 서비스의 여러 구현을 등록할 수 없습니다. 이러한 메서드는 서비스의 여러 인스턴스를 등록할 수 있지만 모두 동일한 구현 형식을 갖습니다.

서비스 등록 방법을 사용하여 동일한 서비스 유형의 여러 서비스 인스턴스를 등록할 수 있습니다. 다음 예제에서는 `AddSingleton`를 서비스 형식으로 사용하여 `IMessageWriter`을 두 번 호출합니다. 두 번째 `AddSingleton` 호출은 `IMessageWriter`로 확인되면 이전 호출을 재정의하고 `IEnumerable<IMessageWriter>`를 통해 여러 서비스가 확인되면 이전 호출에 추가됩니다. 서비스는 `IEnumerable<{SERVICE}>`를 통해 해결될 때 등록된 순서로 나타납니다.

C#

```
using ConsoleDI.IEnumerableExample;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
builder.Services.AddSingleton<IMessageWriter, LoggingMessageWriter>();
builder.Services.AddSingleton<ExampleService>();

using IHost host = builder.Build();

_ = host.Services.GetService<ExampleService>();

await host.RunAsync();
```

위의 샘플 소스 코드는 `IMessageWriter`의 두 가지 구현을 등록합니다.

C#

```
using System.Diagnostics;

namespace ConsoleDI.IEnumerableExample;

public sealed class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is LoggingMessageWriter);

        var dependencyArray = messageWriters.ToArray();
        Trace.Assert(dependencyArray[0] is ConsoleMessageWriter);
        Trace.Assert(dependencyArray[1] is LoggingMessageWriter);
    }
}
```

`ExampleService`는 두 가지 생성자 매개 변수인 단일 `IMessageWriter` 및 `IEnumerable<IMessageWriter>`를 정의합니다. 단일 `IMessageWriter` 항목은 등록할 마지막 구현인 반면 `IEnumerable<IMessageWriter>` 등록된 모든 구현을 나타냅니다.

또한 프레임워크에서는 아직 등록된 구현이 없는 경우에만 서비스를 등록하는 `TryAdd{LIFETIME}` 확장 메서드를 제공합니다.

다음 예제에서 `AddSingleton`을 호출하면 `ConsoleMessageWriter`가 `IMessageWriter`에 대한 구현으로 등록됩니다. `TryAddSingleton`에 대한 호출은 `IMessageWriter`에 이미 등록된 구현이 있으므로 아무런 효과가 없습니다.

C#

```
services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
services.TryAddSingleton<IMessageWriter, LoggingMessageWriter>();
```

`TryAddSingleton` 이미 추가되고 "try"가 실패하므로 효과가 없습니다. 다음 `ExampleService` 을 어설션합니다.

C#

```
public class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
```

```
    {
        Trace.Assert(messageWriter is ConsoleMessageWriter);
        Trace.Assert(messageWriters.Single() is ConsoleMessageWriter);
    }
}
```

자세한 내용은 다음을 참조하세요.

- TryAdd
- TryAddTransient
- TryAddScoped
- TryAddSingleton

TryAddEnumerable(ServiceDescriptor) 메서드는 '동일한 형식'의 구현이 아직 없는 경우에만 서비스를 등록합니다. 여러 서비스가 `IEnumerable<{SERVICE}>`를 통해 해결됩니다. 서비스를 등록할 때 동일한 형식 중 하나가 아직 추가되지 않은 경우 인스턴스를 추가합니다. 라이브러리 작성자는 컨테이너에 있는 특정 구현의 여러 복사본이 등록되지 않도록 `TryAddEnumerable`을 사용합니다.

다음 예제에서 `TryAddEnumerable`을 처음 호출하면 `MessageWriter`가 `IMessageWriter1`에 대한 구현으로 등록됩니다. 두 번째 호출에서는 `MessageWriter`에 대한 `IMessageWriter2`를 등록합니다. 세 번째 호출은 `IMessageWriter1`에 `MessageWriter`의 등록된 구현이 이미 있으므로 아무런 효과가 없습니다.

C#

```
public interface IMessageWriter1 { }
public interface IMessageWriter2 { }

public class MessageWriter : IMessageWriter1, IMessageWriter2
{
}

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1,
MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter2,
MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1,
MessageWriter>());
```

동일한 형식의 여러 구현을 등록하는 경우를 제외하고 서비스 등록은 순서 독립적입니다.

`IServiceCollection`은 ServiceDescriptor 개체의 컬렉션입니다. 다음 예에서는 `ServiceDescriptor`을 만든 후 추가하여 서비스를 등록하는 방법을 보여 줍니다.

C#

```
string secretKey = Configuration["SecretKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMessageWriter),
    _ => new DefaultMessageWriter(secretKey),
    ServiceLifetime.Transient);

services.Add(descriptor);
```

기본 제공 `Add{LIFETIME}` 메서드는 같은 방법을 사용합니다. 예를 들어 AddScoped 소스 코드 를 참조하세요.

Last updated on 2026. 01. 24.

서비스 수명

서비스는 일시적, 범위가 지정된 또는 싱글톤 수명에 등록할 수 있습니다. 등록된 각 서비스의 수명을 적절히 선택합니다.

Transient

일시적 수명이 있는 서비스는 서비스 컨테이너에서 요청할 때마다 만들어집니다. 서비스를 임시로 등록하려면 .를 호출합니다 AddTransient.

요청을 처리하는 앱에서 Transient 서비스는 요청이 끝날 때 삭제됩니다. 이 수명은 서비스가 요청 시마다 해결되고 생성되므로 요청 당 할당이 발생합니다. 자세한 내용은 일시적 및 공유 인스턴스에 대한 IDisposable 지침을 참조하세요.

범위

웹 애플리케이션의 경우 범위가 *지정된* 수명은 서비스가 클라이언트 요청(연결)당 한 번 생성됨을 나타냅니다. 요청을 처리하는 앱에서 Scoped 서비스는 요청이 끝날 때 삭제됩니다. 를 호출 AddScoped하여 범위가 지정된 서비스를 등록합니다.

> ⓘ **참고 항목**
>
> Entity Framework Core를 사용하는 경우 AddDbContext 확장 메서드는 기본적으로 범위가 지정된 수명으로 `DbContext` 형식을 등록합니다.

범위가 지정된 서비스는 항상 범위 내에서 사용해야 합니다. 즉, 암시적 범위(예: ASP.NET Core의 요청별 범위) 또는 명시적으로 만든 IServiceScopeFactory.CreateScope()범위입니다.

싱글톤 내에서 생성자 주입을 사용하거나 를 요청함으로써 범위가 지정된 서비스를 싱글톤에서 해결하지 마세요. 이렇게 하면 범위가 지정된 서비스가 싱글톤처럼 동작하므로 후속 요청을 처리할 때 상태가 잘못될 수 있습니다.

명시적 범위를 IServiceScopeFactory로 만들고 사용할 경우, 싱글톤 내에서 범위가 지정된 서비스를 해결하는 것이 허용됩니다.

다음을 수행해도 괜찮습니다.

- 범위가 지정된 서비스 또는 임시 서비스에서 싱글톤 서비스를 해결합니다.
- 다른 범위가 지정된 서비스 또는 임시 서비스에서 범위가 지정된 서비스를 해결합니다.

기본적으로 개발 환경에서 수명이 더 긴 다른 서비스에서 서비스를 해결하면 예외가 throw됩니다. 자세한 내용은 범위 유효성 검사를 참조하세요.

Singleton

싱클톤 수명 서비스는 다음과 같은 경우 생성됩니다.

- 처음 요청되는 경우
- 개발자가 구현 인스턴스를 컨테이너에 직접 제공하는 경우 (이 방법은 거의 필요하지 않습니다.)

종속성 주입 컨테이너로부터 서비스 구현에 대한 모든 후속 요청은 동일한 인스턴스를 사용합니다. 앱에 싱글톤 동작이 필요한 경우 서비스 컨테이너가 서비스 수명을 관리하도록 허용합니다. 싱글톤 디자인 패턴을 구현하지 말고 싱글톤을 삭제하는 코드를 제공합니다. 컨테이너에서 서비스를 해결한 코드에서는 서비스를 삭제하면 안 됩니다. 형식 또는 팩터리가 싱글톤으로 등록된 경우 컨테이너에서 싱글톤을 자동으로 삭제합니다.

AddSingleton를 사용하여 싱글톤 서비스를 등록합니다. 싱글톤 서비스는 스레드로부터 안전해야 하며 상태 비저장 서비스에서 자주 사용됩니다.

요청을 처리하는 앱에서 싱글톤 서비스는 애플리케이션 종료 시 ServiceProvider가 삭제될 때 삭제됩니다. 앱이 종료될 때까지 메모리가 해제되지 않으므로 싱글톤 서비스에서 메모리 사용을 고려합니다.

Last updated on 2026. 01. 24.

빠른 시작: .NET의 종속성 주입 기본 사항

이 빠른 시작 가이드에서는 .NET 콘솔 앱을 작성하고, 수동으로 ServiceCollection 및 해당 ServiceProvider을 생성합니다. DI(종속성 주입)를 사용하여 서비스를 등록하고 해결하는 방법을 알아봅니다. 이 문서에서는 Microsoft.Extensions.DependencyInjection NuGet 패키지를 사용하여 .NET에서 DI의 기본 사항을 보여 줍니다.

> **참고 항목**
>
> 이 문서에서는 **일반 호스트** 기능을 활용하지 않습니다. 보다 포괄적인 가이드는 **.NET에서 종속성 주입 사용을** 참조하세요.

시작하기

시작하려면 **DI.Basics**라는 새 .NET 콘솔 애플리케이션을 만듭니다. Visual Studio에서 **파일** > **새** > **프로젝트를** 선택하거나 .NET CLI를 사용하여 명령어를 입력합니다`dotnet new console`.

다음으로 프로젝트 파일에서 Microsoft.Extensions.DependencyInjection 에 패키지 참조를 추가합니다. 패키지를 추가한 후 프로젝트가 *DI.Basics.csproj* 파일의 다음 XML과 유사한지 확인합니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.DependencyInjection"
Version="10.0.5" />
  </ItemGroup>

</Project>
```

종속성 주입 기본 사항

종속성 주입은 하드 코딩된 종속성을 제거하고 애플리케이션을 보다 유지 관리하고 테스트할 수 있도록 하는 데 사용할 수 있는 디자인 패턴입니다. DI는 클래스와 해당 종속성 간에

IoC(Inversion of Control) 를 달성하기 위한 기술입니다.

Microsoft.Extensions.DependencyInjection.Abstractions NuGet 패키지는 .NET에서 DI에 대한 추상성을 정의합니다.

- IServiceCollection: 서비스 설명자 컬렉션에 대한 계약을 정의합니다.
- IServiceProvider: 서비스 개체를 검색하는 메커니즘을 정의합니다.
- ServiceDescriptor: 서비스 유형, 구현 및 수명을 가진 서비스에 대해 설명합니다.

.NET에서는 서비스를 추가하고 `IServiceCollection`에서 해당 서비스를 구성하여 DI를 관리합니다. 서비스를 등록한 후 메서드를 BuildServiceProvider 호출하여 인스턴스를 빌드합니다 `IServiceProvider` . `IServiceProvider`는 등록된 모든 서비스의 컨테이너 역할을 하며, 이를 사용하여 서비스를 해결합니다.

예제 서비스 만들기

모든 서비스가 동일하게 만들어지는 것은 아닙니다. 일부 서비스에는 서비스 컨테이너가 해당 인스턴스를 받을 때마다(*일시적*) 새 인스턴스가 필요한 반면, 다른 서비스는 요청(*범위*) 또는 앱의 전체 수명(*싱글톤*)에서 공유되어야 합니다. 서비스 수명에 대한 자세한 내용은 서비스 수명을 참조하세요.

마찬가지로 일부 서비스는 구체적인 형식만 노출하고 다른 서비스는 인터페이스와 구현 형식 간의 계약으로 표현됩니다. 이러한 개념을 설명하는 데 도움이 되는 여러 가지 서비스 변형을 만듭니다.

IConsole.cs 새 C# 파일을 만들고 다음 코드를 추가합니다.

C#

```
public interface IConsole
{
    void WriteLine(string message);
}
```

이 파일은 `IConsole` 단일 메서드 `WriteLine`를 노출하는 인터페이스를 정의합니다. 다음으로 *DefaultConsole.cs* 새 C# 파일을 만들고 다음 코드를 추가합니다.

C#

```
internal sealed class DefaultConsole : IConsole
{
    public bool IsEnabled { get; set; } = true;

    void IConsole.WriteLine(string message)
    {
```

```
        if (IsEnabled is false)
        {
            return;
        }

        Console.WriteLine(message);
    }
}
```

앞의 코드는 인터페이스의 기본 구현을 `IConsole` 나타냅니다. `WriteLine` 메서드는 `IsEnabled` 속성을 기반으로 콘솔에 조건부로 씁니다.

> **팁**
>
> 구현의 명명은 개발 팀이 동의해야 하는 선택 사항입니다. `Default` 접두사는 인터페이스의 *기본* 구현을 나타내는 일반적인 규칙이지만 필수는 *아닙니다*.

다음으로 , *IGreetingService.cs* 파일을 만들고 다음 C# 코드를 추가합니다.

C#

```
public interface IGreetingService
{
    string Greet(string name);
}
```

그런 다음 *DefaultGreetingService.cs* 새 C# 파일을 추가하고 다음 코드를 추가합니다.

C#

```
internal sealed class DefaultGreetingService(
    IConsole console) : IGreetingService
{
    public string Greet(string name)
    {
        var greeting = $"Hello, {name}!";

        console.WriteLine(greeting);

        return greeting;
    }
}
```

앞의 코드는 인터페이스의 기본 구현을 `IGreetingService` 나타냅니다. 서비스 구현에는 `IConsole` 기본 생성자 매개 변수가 필요합니다. `Greet` 메서드는 다음 작업을 수행합니다.

- `greeting`을(를) 기반으로 `name`을(를) 생성합니다.

- `WriteLine` 인스턴스에서 `IConsole` 메서드를 호출합니다.
- 호출자에게 `greeting` 반환합니다.

이 `DefaultGreetingService` 클래스는 서비스 구현을 통해 상속을 방지할 수 있으며, 어셈블리에 대한 액세스를 제한하기 위해 `seal`을 사용 `internal`할 수 있음을 보여 줍니다.

마지막으로 만들 서비스는 *FarewellService.cs* 파일입니다. 계속하기 전에 다음 C# 코드를 추가합니다.

C#

```
public class FarewellService(IConsole console)
{
    public string SayGoodbye(string name)
    {
        var farewell = $"Goodbye, {name}!";

        console.WriteLine(farewell);

        return farewell;
    }
}
```

인터페이스 `FarewellService` 가 아닌 구체적인 형식을 나타냅니다. 소비자가 액세스할 수 있도록, `public`로 선언해야 합니다. 다른 서비스 구현 유형을 `internal` 및 `sealed`로 선언하는 것과 달리, 이 코드는 모든 서비스가 반드시 인터페이스여야 하는 것은 아님을 보여줍니다.

`Program` 클래스 업데이트

Program.cs 파일을 열고 기존 코드를 다음 C# 코드로 바꿉다.

C#

```
using Microsoft.Extensions.DependencyInjection;

// 1. Create the service collection.
var services = new ServiceCollection();

// 2. Register (add and configure) the services.
services.AddSingleton<IConsole>(
    implementationFactory: static _ => new DefaultConsole
    {
        IsEnabled = true
    });
services.AddSingleton<IGreetingService, DefaultGreetingService>();
services.AddSingleton<FarewellService>();

// 3. Build the service provider from the service collection.
```

```
var serviceProvider = services.BuildServiceProvider();

// 4. Resolve the services that you need.
var greetingService = serviceProvider.GetRequiredService<IGreetingService>();
var farewellService = serviceProvider.GetRequiredService<FarewellService>();

// 5. Use the services
var greeting = greetingService.Greet("David");
var farewell = farewellService.SayGoodbye("David");
```

앞의 코드는 다음 방법을 보여 줍니다.

- 새 `ServiceCollection` 인스턴스를 만듭니다.
- 이곳에서 서비스를 등록하고 구성합니다.`ServiceCollection`
  - `IConsole` 구현 팩터리 오버로드를 사용하여 서비스를 제공합니다. `DefaultConsole` 형식을 반환하고, `IsEnabled` 속성은 `true`로 설정합니다.
  - `IGreetingService` 서비스의 해당 구현 유형은 `DefaultGreetingService`입니다.
  - `FarewellService` 서비스는 구체적인 형식으로 제공됩니다.
- `ServiceProvider`을(를) `ServiceCollection`에서 빌드합니다.
- `IGreetingService` 및 `FarewellService` 서비스를 해결합니다.
- 해결된 서비스를 사용하여 이름이 지정된 `David`사람에게 인사하고 작별 인사를 합니다.

`IsEnabled` `DefaultConsole`의 `false` 속성을 업데이트하면 `Greet` 및 `SayGoodbye` 메서드가 결과 메시지를 콘솔에 기록하는 단계를 생략합니다. 이 변경은 `IConsole` 서비스가 `IGreetingService` 및 `FarewellService` 서비스에 앱의 동작에 영향을 주는 *의존성*으로 *주입*됨을 보여줍니다.

이러한 모든 서비스는 싱글톤으로 등록됩니다. 이 샘플의 경우 *일시적* 또는 *범위가 지정된* 서비스로 등록하는 경우 동일하게 작동합니다.

> **ⓘ Important**
>
> 이 모순된 예제에서는 서비스 수명이 중요하지 않습니다. 실제 애플리케이션에서 각 서비스의 수명을 신중하게 고려합니다.

샘플 앱 실행

샘플 앱을 실행하려면 Visual Studio 또는 Visual Studio Code에서 `F5` 키를 누르거나 터미널에서 명령을 실행 `dotnet run` 합니다. 앱이 완료되면 다음과 같은 출력이 표시됩니다.

콘솔

```
Hello, David!
```

```
Goodbye, David!
```

서비스 설명자

서비스를 `ServiceCollection` 추가하는 데 가장 일반적으로 사용되는 API는 다음과 같은 수명 명명된 제네릭 확장 메서드입니다.

- `AddSingleton<TService>`
- `AddTransient<TService>`
- `AddScoped<TService>`

이러한 메서드는 ServiceDescriptor 인스턴스를 생성하고 그것을 `ServiceCollection`에 추가하는 편리한 메서드입니다. 서비스 `ServiceDescriptor` 유형, 구현 유형 및 수명을 가진 서비스를 설명하는 간단한 클래스입니다. 구현 팩터리 및 인스턴스를 설명할 수도 있습니다.

`ServiceCollection`에 등록한 각 서비스에 대해, 인스턴스 `ServiceDescriptor`를 사용하여 `Add` 메서드를 직접 호출할 수도 있습니다. 다음 예제를 고려하세요.

C#

```
services.Add(ServiceDescriptor.Describe(
    serviceType: typeof(IConsole),
    implementationFactory: static _ => new DefaultConsole
    {
        IsEnabled = true
    },
    lifetime: ServiceLifetime.Singleton));
```

이전의 코드는 `IConsole`에 `ServiceCollection` 서비스가 등록된 방법과 동일합니다. 이 메서드는 `Add` 메서드가 `IConsole` 서비스를 설명하는 `ServiceDescriptor` 인스턴스를 추가합니다. 정적 메서드 `ServiceDescriptor.Describe`는 다양한 `ServiceDescriptor` 생성자에 위임합니다. 서비스에 해당하는 코드를 고려합니다.`IGreetingService`

C#

```
services.Add(ServiceDescriptor.Describe(
    serviceType: typeof(IGreetingService),
    implementationType: typeof(DefaultGreetingService),
    lifetime: ServiceLifetime.Singleton));
```

앞의 `IGreetingService` 코드는 서비스 유형, 구현 유형 및 수명을 사용하여 서비스를 설명합니다. 마지막으로 서비스에 해당하는 코드를 고려합니다.`FarewellService`

C#

```
services.Add(ServiceDescriptor.Describe(
    serviceType: typeof(FarewellService),
    implementationType: typeof(FarewellService),
    lifetime: ServiceLifetime.Singleton));
```

앞의 코드는 구체적인 `FarewellService` 형식을 서비스 및 구현 형식으로 설명합니다. 서비스는 싱글톤 서비스로 등록됩니다.

참고하십시오

- .NET 종속성 주입
- 종속성 주입 지침
- ASP.NET Core에서 종속성 주입

Last updated on 2026. 03. 26.

자습서: .NET에서 종속성 주입 사용

이 자습서에서는 .NETDI(종속성 주입)를 사용하는 방법을 보여 줍니다. DI는 서비스를 추가하고 IServiceCollection에서 구성하여 관리됩니다. IHost 인터페이스는 등록된 모든 서비스의 컨테이너 역할을 하는 IServiceProvider 인스턴스를 노출합니다.

이 튜토리얼에서는 다음을 배우게 됩니다:

- ✓ 종속성 주입을 사용하는 .NET 콘솔 앱을 만듭니다.
- ✓ 제네릭 호스트를 빌드하고 구성합니다.
- ✓ 여러 인터페이스 및 해당 구현을 작성합니다.
- ✓ DI에 대한 서비스 수명 및 범위 지정을 사용합니다.

필수 조건

- .NET Core 8.0 SDK 이상.
- 새 .NET 애플리케이션을 만들고 NuGet 패키지를 설치하는 방법에 대해 잘 알고 있습니다.

새 콘솔 애플리케이션 만들기

dotnet new 명령 또는 IDE 새 프로젝트 마법사를 사용하여 **ConsoleDI라는 새 .NET 콘솔 애플리케이션을 만듭니다.Example**. 프로젝트에 Microsoft.Extensions.Hosting NuGet 패키지를 추가합니다.

새 콘솔 앱 프로젝트 파일은 다음과 유사합니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>ConsoleDI.Example</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder"
Version="10.0.2" />
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="10.0.2" />
  </ItemGroup>

</Project>
```

> **ⓘ Important**
>
> 이 예제에서는 앱을 빌드하고 실행하려면 Microsoft.Extensions.Hosting NuGet 패키지가 필요합니다. 일부 메타패키지에는 `Microsoft.Extensions.Hosting` 패키지가 포함될 수 있습니다. 이 경우 명시적 패키지 참조가 필요하지 않습니다.

인터페이스 추가

이 샘플 앱에서는 종속성 주입이 서비스 수명을 처리하는 방법을 알아봅니다. 다양한 서비스 수명을 나타내는 여러 인터페이스를 만듭니다. 프로젝트 루트 디렉터리에 다음 인터페이스를 추가합니다.

IReportServiceLifetime.cs

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ConsoleDI.Example;

public interface IReportServiceLifetime
{
    Guid Id { get; }

    ServiceLifetime Lifetime { get; }
}
```

`IReportServiceLifetime` 인터페이스는 다음을 정의합니다.

- 서비스의 고유 식별자를 나타내는 `Guid Id` 속성입니다.
- 서비스 수명을 나타내는 ServiceLifetime 속성입니다.

IExampleTransientService.cs

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ConsoleDI.Example;

public interface IExampleTransientService : IReportServiceLifetime
{
    ServiceLifetime IReportServiceLifetime.Lifetime => ServiceLifetime.Transient;
}
```

IExampleScopedService.cs

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ConsoleDI.Example;

public interface IExampleScopedService : IReportServiceLifetime
{
    ServiceLifetime IReportServiceLifetime.Lifetime => ServiceLifetime.Scoped;
}
```

IExampleSingletonService.cs

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ConsoleDI.Example;

public interface IExampleSingletonService : IReportServiceLifetime
{
    ServiceLifetime IReportServiceLifetime.Lifetime => ServiceLifetime.Singleton;
}
```

`IReportServiceLifetime`의 모든 하위 인터페이스는 기본값으로 `IReportServiceLifetime.Lifetime`을 명시적으로 구현합니다. 예를 들어 `IExampleTransientService` `IReportServiceLifetime.Lifetime` 값을 사용하여 `ServiceLifetime.Transient` 명시적으로 구현합니다.

기본 구현 추가

예제 구현 모두는 `Id`에서 얻은 결과로 Guid.NewGuid() 속성을 초기화합니다. 다양한 서비스에 대한 다음 기본 구현 클래스를 프로젝트 루트 디렉터리에 추가합니다.

ExampleTransientService.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ExampleTransientService : IExampleTransientService
{
    Guid IReportServiceLifetime.Id { get; } = Guid.NewGuid();
}
```

ExampleScopedService.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ExampleScopedService : IExampleScopedService
{
    Guid IReportServiceLifetime.Id { get; } = Guid.NewGuid();
}
```

ExampleSingletonService.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ExampleSingletonService : IExampleSingletonService
{
    Guid IReportServiceLifetime.Id { get; } = Guid.NewGuid();
}
```

각 구현은 `internal sealed` 정의되고 해당 인터페이스를 구현합니다. `internal` 또는 `sealed` 필요는 없습니다. 그러나 구현을 `internal` 처리하여 구현 유형이 외부 소비자에게 유출되지 않도록 하는 것이 일반적입니다. 또한 각 형식은 확장되지 않으므로 다음과 같이 `sealed` 표시됩니다. 예를 들어 `ExampleSingletonService` `IExampleSingletonService` 구현합니다.

DI가 필요한 서비스 추가

콘솔 앱에 서비스 역할을 하는 다음 서비스 수명 보고자 클래스를 추가합니다.

ServiceLifetimeReporter.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ServiceLifetimeReporter(
    IExampleTransientService transientService,
    IExampleScopedService scopedService,
    IExampleSingletonService singletonService)
{
    public void ReportServiceLifetimeDetails(string lifetimeDetails)
    {
        Console.WriteLine(lifetimeDetails);

        LogService(transientService, "Always different");
        LogService(scopedService, "Changes only with lifetime");
        LogService(singletonService, "Always the same");
    }
```

```
    private static void LogService<T>(T service, string message)
        where T : IReportServiceLifetime =>
        Console.WriteLine(
            $"    {typeof(T).Name}: {service.Id} ({message})");
}
```

`ServiceLifetimeReporter` 앞에서 언급한 각 서비스 인터페이스, 즉 `IExampleTransientService`, `IExampleScopedService` 및 `IExampleSingletonService` 필요로 하는 생성자를 정의합니다. 이 개체는 소비자가 지정된 `lifetimeDetails` 매개 변수를 사용하여 서비스에 대해 보고할 수 있는 단일 메서드를 노출합니다. 호출될 때 `ReportServiceLifetimeDetails` 메서드는 서비스 수명 메시지와 함께 각 서비스의 고유 식별자를 기록합니다. 로그 메시지는 서비스 수명을 시각화하는 데 도움이 됩니다.

DI용 서비스 등록

다음 코드로 *Program.cs* 업데이트합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using ConsoleDI.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddTransient<IExampleTransientService, ExampleTransientService>();
builder.Services.AddScoped<IExampleScopedService, ExampleScopedService>();
builder.Services.AddSingleton<IExampleSingletonService, ExampleSingletonService>();
builder.Services.AddTransient<ServiceLifetimeReporter>();
using IHost host = builder.Build();

ExemplifyServiceLifetime(host.Services, "Lifetime 1");
ExemplifyServiceLifetime(host.Services, "Lifetime 2");

await host.RunAsync();

static void ExemplifyServiceLifetime(IServiceProvider hostProvider, string lifetime)
{
    using IServiceScope serviceScope = hostProvider.CreateScope();
    IServiceProvider provider = serviceScope.ServiceProvider;
    ServiceLifetimeReporter logger =
provider.GetRequiredService<ServiceLifetimeReporter>();
    logger.ReportServiceLifetimeDetails(
        $"{lifetime}: Call 1 to provider.GetRequiredService<ServiceLifetimeReporter>
()");

    Console.WriteLine("...");

    logger = provider.GetRequiredService<ServiceLifetimeReporter>();
    logger.ReportServiceLifetimeDetails(
```

```
        $"{lifetime}: Call 2 to provider.GetRequiredService<ServiceLifetimeReporter>
()");

    Console.WriteLine();
}
```

각 `services.Add{LIFETIME}<{SERVICE}>` 확장 메서드는 서비스를 추가(및 잠재적으로 구성)합니다. 앱은 이 규칙을 따르는 것이 좋습니다. 공식 Microsoft 패키지를 작성하지 않는 한 확장 메서드를 Microsoft.Extensions.DependencyInjection 네임스페이스에 배치하지 마세요. `Microsoft.Extensions.DependencyInjection` 네임스페이스 내에 정의된 확장 메서드:

- 더 많은 지시문 없이 `using`에 표시됩니다.
- 이러한 확장 메서드가 일반적으로 호출되는 `using` 또는 `Program` 클래스에서 필요한 `Startup` 지시문 수를 줄입니다.

앱:

- IHostApplicationBuilder 인스턴스를 호스트 작성기 설정로 만듭니다.
- 서비스를 구성하고 해당 서비스 수명을 사용하여 추가합니다.
- Build() 호출하고 IHost인스턴스를 할당합니다.
- `ExemplifyServiceLifetime`를 호출하고 IHost.Services을 전달합니다.

결론

이 샘플 앱에서는 여러 인터페이스와 해당 구현을 만들었습니다. 이러한 각 서비스는 고유하게 식별되고 ServiceLifetime와 짝지어집니다. 샘플 앱은 인터페이스에 대해 서비스 구현을 등록하는 방법과 지원 인터페이스 없이 순수 클래스를 등록하는 방법을 보여 줍니다. 그런 다음 샘플 앱은 생성자 매개 변수로 정의된 종속성이 런타임에 확인되는 방법을 보여 줍니다.

앱을 실행하면 다음과 유사한 출력이 표시됩니다.

C#

```
// Sample output:
// Lifetime 1: Call 1 to provider.GetRequiredService<ServiceLifetimeReporter>()
//     IExampleTransientService: d08a27fa-87d2-4a06-98d7-2773af886125 (Always
different)
//     IExampleScopedService: 402c83c9-b4ed-4be1-b78c-86be1b1d908d (Changes only
with lifetime)
//     IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b (Always the
same)
// ...
// Lifetime 1: Call 2 to provider.GetRequiredService<ServiceLifetimeReporter>()
//     IExampleTransientService: b43d68fb-2c7b-4a9b-8f02-fc507c164326 (Always
different)
//     IExampleScopedService: 402c83c9-b4ed-4be1-b78c-86be1b1d908d (Changes only
with lifetime)
```

```
//     IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b (Always the
same)
//
// Lifetime 2: Call 1 to provider.GetRequiredService<ServiceLifetimeReporter>()
//     IExampleTransientService: f3856b59-ab3f-4bbd-876f-7bab0013d392 (Always
different)
//     IExampleScopedService: bba80089-1157-4041-936d-e96d81dd9d1c (Changes only
with lifetime)
//     IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b (Always the
same)
// ...
// Lifetime 2: Call 2 to provider.GetRequiredService<ServiceLifetimeReporter>()
//     IExampleTransientService: a8015c6a-08cd-4799-9ec3-2f2af9cbbfd2 (Always
different)
//     IExampleScopedService: bba80089-1157-4041-936d-e96d81dd9d1c (Changes only
with lifetime)
//     IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b (Always the
same)
```

앱 출력에서 다음을 확인할 수 있습니다.

- Transient 서비스는 항상 다릅니다. 서비스를 검색할 때마다 새 인스턴스가 만들어집니다.
- Scoped 서비스는 새 범위에서만 변경되지만 범위 내에서는 동일한 인스턴스입니다.
- Singleton 서비스는 항상 동일합니다. 새 인스턴스는 한 번만 만들어집니다.

참고하십시오

- 종속성 주입 지침
- 빠른 시작: 종속성 주입 기본 사항
- ASP.NET Core에서 종속성 주입

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 24.

종속성 주입 지침

이 문서에서는 .NET 애플리케이션에서 DI(종속성 주입)를 구현하기 위한 일반적인 지침과 모범 사례를 제공합니다.

종속성 주입을 위한 서비스 디자인

종속성 주입을 위한 서비스를 디자인하는 경우

- 상태 저장 정적 클래스 및 멤버를 사용하지 마세요. 글로벌 상태를 피하기 위해, 싱글톤 서비스를 사용하도록 앱을 설계하세요.
- 서비스 내의 종속 클래스를 직접 인스턴스화하지 마세요. 직접 인스턴스화는 코드를 특정 구현에 결합합니다.
- 서비스를 작고 잘 구성되고 쉽게 테스트할 수 있도록 만듭니다.

클래스에 주입된 종속성이 많은 경우 클래스가 역할이 너무 많고 SRP(단일 책임 원칙)을 위반하는 것일 수 있습니다. 해당 책임 몇 가지를 새로운 클래스로 이동하여 클래스를 리팩터링해 보세요.

서비스 삭제

컨테이너는 자신이 만든 형식을 정리하며 Dispose 인스턴스에서 IDisposable를 호출합니다. 개발자는 컨테이너에서 해결된 서비스를 절대 처리해서는 안 됩니다. 형식 또는 팩터리가 싱글톤으로 등록된 경우 컨테이너에서 싱글톤을 자동으로 삭제합니다.

다음 예제에서는 서비스가 서비스 컨테이너에 의해 만들어지고 자동으로 삭제됩니다.

C#

```
namespace ConsoleDisposable.Example;

public sealed class TransientDisposable : IDisposable
{
    public void Dispose() => Console.WriteLine($"
{nameof(TransientDisposable)}.Dispose()");
}
```

위의 삭제 가능한 형식은 임시 수명을 갖도록 만들어진 것입니다.

C#

```
namespace ConsoleDisposable.Example;

public sealed class ScopedDisposable : IDisposable
{
```

```
    public void Dispose() => Console.WriteLine($"
{nameof(ScopedDisposable)}.Dispose()");
}
```

본 일회용품은 제한된 수명을 갖도록 설계되었습니다.

C#

```
namespace ConsoleDisposable.Example;

public sealed class SingletonDisposable : IDisposable
{
    public void Dispose() => Console.WriteLine($"
{nameof(SingletonDisposable)}.Dispose()");
}
```

위의 삭제 가능한 형식은 싱글톤 수명을 갖도록 만들어진 것입니다.

C#

```
using ConsoleDisposable.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddTransient<TransientDisposable>();
builder.Services.AddScoped<ScopedDisposable>();
builder.Services.AddSingleton<SingletonDisposable>();

using IHost host = builder.Build();

ExemplifyDisposableScoping(host.Services, "Scope 1");
Console.WriteLine();

ExemplifyDisposableScoping(host.Services, "Scope 2");
Console.WriteLine();

await host.RunAsync();

static void ExemplifyDisposableScoping(IServiceProvider services, string scope)
{
    Console.WriteLine($"{scope}...");

    using IServiceScope serviceScope = services.CreateScope();
    IServiceProvider provider = serviceScope.ServiceProvider;

    _ = provider.GetRequiredService<TransientDisposable>();
    _ = provider.GetRequiredService<ScopedDisposable>();
    _ = provider.GetRequiredService<SingletonDisposable>();
}
```

디버그 콘솔은 실행 후 다음 샘플 출력을 보여 줍니다.

콘솔

```
Scope 1...
ScopedDisposable.Dispose()
TransientDisposable.Dispose()

Scope 2...
ScopedDisposable.Dispose()
TransientDisposable.Dispose()

info: Microsoft.Hosting.Lifetime[0]
      Application started.Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
     Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
     Content root path: .\configuration\console-di-disposable\bin\Debug\net5.0
info: Microsoft.Hosting.Lifetime[0]
     Application is shutting down...
SingletonDisposable.Dispose()
```

서비스 컨테이너에서 만들지 않은 서비스

다음 코드를 생각해 봅시다.

C#

```
// Register example service in IServiceCollection.
builder.Services.AddSingleton(new ExampleService());
```

앞의 코드에서 다음을 수행합니다.

- `ExampleService` 인스턴스가 서비스 컨테이너에서 만들어지지 **않았습니다**.
- 프레임워크가 서비스를 자동으로 삭제하지 **않습니다**.
- 개발자가 서비스 삭제를 담당합니다.

임시 및 공유 인스턴스에 대한 IDisposable 지침

임시적인 제한 수명

시나리오

앱에는 다음 시나리오 중 하나에 대해 임시 수명으로 IDisposable 인스턴스가 필요합니다.

- 인스턴스가 루트 범위(루트 컨테이너)에서 확인됩니다.
- 범위가 끝나기 전에 인스턴스를 삭제해야 합니다.

Solution

부모 범위 밖에서 인스턴스를 생성하려면 팩터리 패턴을 사용합니다. 이 경우 앱에는 `Create` 일반적으로 최종 형식의 생성자를 직접 호출하는 메서드가 있습니다. 최종 형식에 다른 종속성이 있는 경우 팩터리는 다음을 수행할 수 있습니다.

- 해당 생성자에서 IServiceProvider을 수신합니다.
- 해당 종속성에서 컨테이너를 사용하는 동안 컨테이너 외부의 인스턴스를 인스턴스화하기 위해 ActivatorUtilities.CreateInstance을 사용합니다.

공유 인스턴스 및 제한 수명

시나리오

앱은 여러 서비스에서 공유 IDisposable 인스턴스가 필요하지만 IDisposable 인스턴스는 수명이 제한되어 있어야 합니다.

Solution

인스턴스를 범위가 지정된 수명으로 등록합니다. IServiceScopeFactory.CreateScope을 사용하여 새 IServiceScope를 만듭니다. 범위의 IServiceProvider를 사용하여 필요한 서비스를 가져옵니다. 더 이상 필요하지 않은 범위를 삭제합니다.

일반 `IDisposable` 지침

- 임시 수명에 IDisposable 인스턴스를 등록하지 마세요. 더 이상 사용하지 않을 때 해결된 서비스를 수동으로 삭제할 수 있도록 팩터리 패턴을 대신 사용합니다.
- 루트 범위에서 임시 또는 범위가 지정된 수명을 가진 인스턴스를 해결하지 마세요. 유일한 예외는 앱이 만들거나 다시 만들고 삭제하는 IServiceProvider경우이지만 이는 이상적인 패턴이 아닙니다.
- DI를 통한 IDisposable 종속성 수신은 수신자가 자체적으로 IDisposable를 구현할 필요가 없습니다. IDisposable 종속성의 수신자는 해당 종속성에서 Dispose를 호출하지 않아야 합니다.
- 범위를 사용하여 서비스 수명을 제어합니다. 범위는 계층적이지 않으며 범위 간 특수 연결이 없습니다.

리소스 정리에 대한 자세한 내용은 메서드 구현 Dispose 또는 메서드 구현을 DisposeAsync 참조하세요. 또한 리소스 정리와 관련하여 컨테이너가 삭제 가능한 임시 서비스를 캡처 시나리오를 살펴보세요.

기본 서비스 컨테이너 바꾸기

기본 제공 서비스 컨테이너는 프레임워크 및 대부분의 소비자 앱의 요구를 충족하기 위한 것입니다. 다음과 같이 지원하지 않는 특정 기능이 필요하지 않는 한 기본 제공 컨테이너를 사용하는

것이 좋습니다.

- 속성 삽입
- 자식 컨테이너
- 사용자 지정 수명 관리
- 초기화 지연에 대한 `Func<T>` 지원
- 규칙 기반 등록

ASP.NET Core 앱에서 사용할 수 있는 타사 컨테이너는 다음과 같습니다.

- Autofac
- DryIoc
- 그레이스
- LightInject
- 라마
- Stashbox
- 간단한 인젝터

스레드 안전성

스레드 안전한 싱글톤 서비스를 생성합니다. 싱글톤 서비스가 임시 서비스에 종속되어 있는 경우 임시 서비스는 싱글톤에서 사용하는 방법에 따라 스레드 보안을 요구할 수도 있습니다. AddSingleton<TService>(IServiceCollection, Func<IServiceProvider,TService>)에 대한 두 번째 인수와 같은 단일 서비스의 팩터리 메서드는 스레드로부터 안전할 필요가 없습니다. 형식(`static`) 생성자와 같이 이 메서드는 단일 스레드에서 한 번만 호출됩니다.

또한 내장 .NET 종속성 주입 컨테이너에서 서비스를 확인하는 프로세스는 스레드 안전성을 보장합니다. `IServiceProvider` 또는 `IServiceScope`가 빌드된 이후에는 여러 스레드에서 동시에 서비스를 호출하는 것이 안전합니다.

> ⓘ **참고 항목**
>
> DI 컨테이너 자체의 스레드 안전은 서비스 생성 및 해결이 안전하다는 보장만 보장합니다. 확인된 서비스 인스턴스 자체를 스레드 안전하게 만들지는 않습니다. 공유 변경 가능한 상태를 보유하는 모든 서비스(특히 싱글톤)는 동시에 액세스하는 경우 자체 동기화 논리를 구현해야 합니다.

Recommendations

- `async/await` 및 `Task` 기반 서비스 확인은 지원되지 않습니다. C#은 비동기 생성자를 지원하지 않으므로, 서비스를 동기식으로 확인한 후 비동기 메서드를 사용합니다.
- 데이터 및 구성을 서비스 컨테이너에 직접 저장하지 마세요. 예를 들어 사용자의 쇼핑 카트는 일반적으로 서비스 컨테이너에 추가하지 말아야 합니다. 구성은 옵션 패턴을 사용해야 합니다. 마찬가지로 다른 개체에 대한 액세스를 허용하기 위해서만 존재하는 "데이터 보유자" 개체를 사용하지 마세요. DI를 통해 실제 항목을 요청하는 것이 좋습니다.
- 서비스에 정적 액세스를 사용하지 마십시오. 예를 들어 다른 곳에서 사용하기 위해 정적 필드 또는 속성으로 캡처하지 IApplicationBuilder.ApplicationServices 마세요.
- DI 팩터리를 빠르고 동기식으로 유지하세요.
- *'서비스 로케이터 패턴'*을 사용하지 마세요. 예를 들어 DI를 대신 사용할 수 있는 경우 서비스 인스턴스를 가져오기 위해 GetService를 호출하지 마세요.
- 피해야 하는 또 다른 서비스 로케이터 변형은 런타임에 종속성을 해결하는 팩터리를 주입하는 것입니다. 이러한 두 가지 방법 모두 제어 반전 전략을 혼합합니다.
- 서비스를 구성할 때 BuildServiceProvider 호출을 방지합니다. `BuildServiceProvider` 호출은 일반적으로 개발자가 다른 서비스를 등록할 때 서비스를 해결하려고 할 때 발생합니다. 대신 이러한 이유로 `IServiceProvider`를 포함하는 오버로드를 사용합니다.
- 컨테이너가 삭제를 위해 삭제 가능한 임시 서비스를 캡처합니다. 따라서 최상위 컨테이너에서 해결할 경우 메모리 누수가 발생할 수 있습니다.
- 범위 유효성 검사를 사용하여 범위가 지정된 서비스를 캡처하는 싱글톤이 앱에 없는지 확인합니다. 자세한 내용은 범위 유효성 검사를 참조하세요.
- 만들거나 전역적으로 공유하는 데 비용이 많이 드는 자체 상태의 서비스에만 싱글톤 수명을 사용합니다. 상태 자체가 없는 서비스에는 싱글톤 수명을 사용하지 마세요. 대부분의 .NET IoC 컨테이너는 기본 범위로 "Transient"를 사용합니다. 싱글톤에 대한 고려 사항 및 단점:
  - **스레드 안전**성: 스레드로부터 안전한 방식으로 싱글톤을 구현해야 합니다.
  - **결합**: 관련 없는 요청을 결합할 수 있습니다.
  - **테스트 과제**: 공유 상태 및 결합으로 인해 단위 테스트가 더 어려워질 수 있습니다.
  - **메모리 영향**: 싱글톤은 애플리케이션의 수명 동안 메모리에 큰 개체 그래프를 활성 상태로 유지할 수 있습니다.
  - **내결함성**: 싱글톤 또는 해당 종속성 트리의 일부가 실패하면 쉽게 복구되지 않습니다.
  - **구성 다시 로드**: 싱글톤은 일반적으로 구성 값의 "핫 다시 로드"를 지원할 수 없습니다.
  - **범위 누수**: 싱글톤은 실수로 범위 한정 혹은 일시적인 종속성을 캡처하여 사실상 싱글톤으로 격상시키고 의도하지 않은 부작용을 일으킬 수 있습니다.
  - **초기화 오버헤드**: 서비스를 확인할 때 IoC 컨테이너는 싱글톤 인스턴스를 조회해야 합니다. 아직 존재하지 않는 경우, 스레드 안전한 방식으로 만들어야 합니다. 반면, 무상태 임시 서비스는 만들고 파괴하는 데 매우 저렴합니다.

모든 권장 사항 집합과 마찬가지로 권장 사항을 무시해야 하는 상황이 발생할 수 있습니다. 예외는 드물며 프레임워크 자체 내에서는 대부분 특별한 경우입니다.

DI는 정적/전역 개체 액세스 패턴의 '대안'입니다. 정적 개체 액세스와 혼합하면 DI의 이점을 인식하지 못할 수 있습니다.

안티 패턴 예제

이 문서의 지침 외에도 **피해야 하는** 몇 가지 안티패턴이 있습니다. 이러한 안티 패턴 중 일부는 런타임 자체를 개발하면서 배웁니다.

> ⚠ **Warning**
>
> 다음은 안티패턴의 예입니다. 코드를 *복사하지 말고*, 이러한 패턴을 *사용하지 말고*, 이러한 패턴을 반드시 피하세요.

컨테이너가 삭제 가능한 임시 서비스를 캡처

구현하는 IDisposable 서비스를 등록하는 경우 기본적으로 DI 컨테이너는 이러한 참조를 유지합니다. 애플리케이션이 중지될 때 컨테이너에서 해결된 항목은 컨테이너가 삭제될 때까지 삭제되지 않으며, 범위에서 해결된 항목은 범위가 삭제될 때까지 삭제되지 않습니다. 컨테이너 수준에서 해결하면 메모리 누수가 발생할 수 있습니다.

```C#
static void TransientDisposablesWithoutDispose()
{
    var services = new ServiceCollection();
    services.AddTransient<ExampleDisposable>();
    ServiceProvider serviceProvider = services.BuildServiceProvider();

    for (int i = 0; i < 1000; ++i)
    {
        _ = serviceProvider.GetRequiredService<ExampleDisposable>();
    }

    // serviceProvider.Dispose();
}
```

위의 안티 패턴에서는 1,000개의 `ExampleDisposable` 개체가 인스턴스화되고 루팅됩니다. 인스턴스가 삭제될 때까지 `serviceProvider` 삭제되지 않습니다.

메모리 누수를 디버깅하는 방법에 대한 자세한 내용은 .NET에서 메모리 누수 디버깅을 참조하세요.

비동기 DI 팩터리에서 교착 상태가 발생할 수 있음

'DI 팩터리'라는 용어는 `Add{LIFETIME}`을 호출할 때 존재하는 오버로드 메서드를 의미합니다. `Func<IServiceProvider, T>`가 등록된 서비스인 경우 `T`를 허용하는 오버로드가 있으며, 매개 변수의 이름은 `implementationFactory`입니다. `implementationFactory`는 람다 식, 로컬 함수 또는 메서드로 제공될 수 있습니다. 팩터리가 비동기일 경우에, Task<TResult>.Result를 사용하면 교착 상태가 발생합니다.

```C#
static void DeadLockWithAsyncFactory()
{
    var services = new ServiceCollection();
    services.AddSingleton<Foo>(implementationFactory: provider =>
    {
        Bar bar = GetBarAsync(provider).Result;
        return new Foo(bar);
    });

    services.AddSingleton<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider();
    _ = serviceProvider.GetRequiredService<Foo>();
}
```

위의 코드에서는 `implementationFactory`에 본문이 Task<TResult>.Result 반환 메서드에서 `Task<Bar>`를 호출하는 람다 식이 지정됩니다. 이로 인해 ***교착 상태가 발생***합니다. `GetBarAsync` 메서드는 단순히 Task.Delay를 사용하여 비동기 작업을 에뮬레이트한 다음 GetRequiredService<T>(IServiceProvider)를 호출합니다.

```C#
static async Task<Bar> GetBarAsync(IServiceProvider serviceProvider)
{
    // Emulate asynchronous work operation
    await Task.Delay(1000);

    return serviceProvider.GetRequiredService<Bar>();
}
```

비동기 지침에 대한 자세한 내용은 비동기 프로그래밍: 중요 정보 및 조언을 참조하세요. 교착 상태 디버깅에 대한 자세한 내용은 .NET에서 교착 상태 디버깅을 참조하세요.

이 안티 패턴을 실행 중이고 교착 상태가 발생하는 경우 Visual Studio의 병렬 스택 창에서 두 개의 스레드가 대기하는 것을 볼 수 있습니다. 자세한 내용은 병렬 스택 창에서 스레드 및 작업 보기를 참조하세요.

조임 종속성

Mark Seemann 이 만든 "포로 종속성" 이라는 용어는 수명이 긴 서비스가 수명이 짧은 서비스 포로를 보유하는 서비스 수명이 잘못 구성되었음을 의미합니다.

```C#
static void CaptiveDependency()
{
    var services = new ServiceCollection();
    services.AddSingleton<Foo>();
    services.AddScoped<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider();
    // Enable scope validation
    // using ServiceProvider serviceProvider = services.BuildServiceProvider(validateScopes: true);

    _ = serviceProvider.GetRequiredService<Foo>();
}
```

위의 코드에서 `Foo`는 싱글톤으로 등록되고 `Bar`는 범위가 지정되는데, 표면적으로는 유효한 것으로 보입니다. 그러나 `Foo`의 구현을 생각해보세요.

```C#
namespace DependencyInjection.AntiPatterns;

public class Foo(Bar bar)
{
}
```

`Foo` 개체는 `Bar` 개체를 필요로 하며, `Foo`는 싱글톤이고 `Bar`는 범위가 지정된 상태이기 때문에, 이는 잘못된 구성입니다. 현재 `Foo`는 오직 한 번 인스턴스화되며, `Bar`를 잡아두고 그 수명은 `Bar`의 의도된 범위 수명보다 깁니다. `validateScopes: true`를 BuildServiceProvider(IServiceCollection, Boolean)에 전달하여 범위를 유효성 검사하는 것이 좋습니다. 범위 InvalidOperationException 의 유효성을 검사할 때 "싱글톤 'Foo'에서 범위가 지정된 서비스 'Bar'를 사용할 수 없습니다."와 유사한 메시지가 표시됩니다.

자세한 내용은 범위 유효성 검사를 참조하세요.

범위가 지정된 서비스를 싱글톤으로 사용하기

범위가 지정된 서비스를 사용하는 경우 범위를 만들지 않거나 기존 범위 내에 있는 경우 서비스는 싱글톤이 됩니다.

```C#
static void ScopedServiceBecomesSingleton()
{
    var services = new ServiceCollection();
    services.AddScoped<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider(validateScopes: true);
    using (IServiceScope scope = serviceProvider.CreateScope())
    {
        // Correctly scoped resolution
        Bar correct = scope.ServiceProvider.GetRequiredService<Bar>();
    }

    // Not within a scope, becomes a singleton
    Bar avoid = serviceProvider.GetRequiredService<Bar>();
}
```

위의 코드에서는 `Bar`가 IServiceScope 내에서 검색되며 이는 올바른 패턴입니다. 안티 패턴은 범위 밖에서 `Bar`를 검색하는 것입니다. 예제 검색이 잘못된 패턴임을 표시하기 위해 변수의 이름이 `avoid`로 지정되어 있습니다.

참고하십시오

- .NET에서 종속성 주입
- 빠른 시작: 종속성 주입 기본 사항
- 자습서: .NET에서 종속성 주입 사용

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 24.

종속성 주입 자동 활성화

NuGet 패키지는 Microsoft.Extensions.DependencyInjection.AutoActivation 첫 번째 사용을 기다리지 않고 애플리케이션 시작 시 싱글톤 서비스를 자동으로 활성화하는 기능을 제공합니다. 이 방법은 애플리케이션이 시작될 때 서비스가 요청을 즉시 처리할 준비가 되도록 하여 초기 요청에 대한 대기 시간을 최소화하는 데 도움이 될 수 있습니다.

자동 활성화를 사용하는 경우

자동 활성화는 다음과 같은 시나리오에서 유용합니다.

- 지연 초기화의 오버헤드를 방지하여 첫 번째 요청의 대기 시간을 최소화하려고 합니다.
- 서비스는 시작 시 캐시를 준비하거나 연결 설정과 같은 초기화 작업을 수행해야 합니다.
- 애플리케이션 수명 시작부터 일관된 응답 시간을 원합니다.
- 서비스에는 요청 처리 중이 아니라 시작 중에 실행하려는 비용이 많이 드는 생성자가 있습니다.

시작하기

자동 활성화를 시작하려면 Microsoft.Extensions.DependencyInjection.AutoActivation NuGet 패키지를 설치하십시오.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.DependencyInjection.AutoActivation
```

자세한 내용은 dotnet add package 또는 .NET 애플리케이션의 패키지 종속성 관리 참조하세요.

자동 활성화를 위한 서비스 등록

자동 활성화 패키지는 서비스 공급자가 빌드될 때 자동으로 활성화되는 서비스를 등록하는 확장 메서드를 제공합니다. 자동 활성화를 위해 서비스를 등록하는 방법에는 여러 가지가 있습니다.

AddActivatedSingleton

이 메서드는 AddActivatedSingleton 서비스를 싱글톤으로 등록하고 시작 시 활성화되도록 합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();

services.AddActivatedSingleton<MyService>();

ServiceProvider provider = services.BuildServiceProvider();
```

`MyService`는 호출될 때 인스턴스화되며, 이는 처음 서비스 공급자로부터 요청될 때가 아닙니다.

ActivateSingleton

ActivateSingleton 메서드는 이미 등록된 싱글톤 서비스가 시작 시 활성화되도록 표시합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();

services.AddSingleton<MyService>();
services.ActivateSingleton<MyService>();

ServiceProvider provider = services.BuildServiceProvider();
```

이 방법은 등록 코드를 변경하지 않고 시작 시 활성화하려는 기존 서비스 등록이 있는 경우에 유용합니다.

TryAddActivatedSingleton

이 메서드는 TryAddActivatedSingleton 아직 등록되지 않은 경우에만 자동 활성화를 위해 서비스를 조건부로 등록합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();

services.TryAddActivatedSingleton<MyService>();
services.TryAddActivatedSingleton<MyService>(); // This has no effect
```

```
ServiceProvider provider = services.BuildServiceProvider();
```

이 메서드는 종속성 주입 프레임워크의 다른 `TryAdd*` 메서드와 동일한 패턴을 따르며 서비스가 한 번만 등록되도록 합니다.

전체 예제

콘솔 애플리케이션에서 자동 활성화를 사용하는 방법을 보여 주는 전체 예제는 다음과 같습니다.

C#

```
public class CacheWarmer
{
    public CacheWarmer()
    {
        Console.WriteLine("Warming up cache at startup...");
        // Perform expensive initialization
        WarmUpCache();
    }

    private void WarmUpCache()
    {
        // Cache warming logic here
        Console.WriteLine("Cache warmed up successfully!");
    }
}
```

이 애플리케이션이 시작되면 애플리케이션이 `CacheWarmer` 요청 처리를 시작하기 전에 생성자의 메시지가 인쇄되어 시작 시 서비스가 활성화되었는지 확인합니다.

종속성을 사용하여 자동 활성화

자동 활성화 서비스는 종속성 주입을 통해 다른 서비스에 따라 달라질 수 있습니다.

C#

```
public class StartupTaskRunner
{
    private readonly ILogger<StartupTaskRunner> _logger;

    public StartupTaskRunner(ILogger<StartupTaskRunner> logger)
    {
        _logger = logger;
        _logger.LogInformation("Running startup tasks...");
        RunTasks();
```

```
    }

    private void RunTasks()
    {
        // Execute startup tasks
        _logger.LogInformation("Startup tasks completed.");
    }
}
```

이 예제에서는 `StartupTaskRunner` 종속성 주입 컨테이너에서 `ILogger<StartupTaskRunner>`자동으로 제공하는 서비스에 종속됩니다.

모범 사례

자동 활성화를 사용하는 경우 다음 모범 사례를 고려합니다.

- **싱글톤에만 사용**: 자동 활성화는 싱글톤 서비스를 위해 설계되었습니다. 수명이 짧은 서비스(범위가 지정되거나 일시적)는 자동 활성화되지 않아야 합니다.
- **빠른 시작 유지**: 자동 활성화 서비스는 애플리케이션 시작 지연을 방지하기 위해 초기화를 신속하게 완료해야 합니다. 장기 실행 초기화의 경우 백그라운드 서비스 또는 호스트된 서비스를 사용하는 것이 좋습니다.
- **오류를 정상적으로 처리**합니다. 자동 활성화 서비스를 생성할 때 발생한 예외로 인해 애플리케이션이 시작되지 않습니다. 서비스 생성자에 강력한 오류 처리를 보장합니다.
- **호스팅된 서비스 고려**: 진행 중인 작업 또는 비동기 초기화를 수행해야 하는 서비스의 경우 자동 활성화 대신 사용하는 IHostedService 것이 좋습니다.

참고하십시오

- .NET의 종속성 주입
- 종속성 주입 지침
- .NET의 워커 서비스

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

.NET의 구성

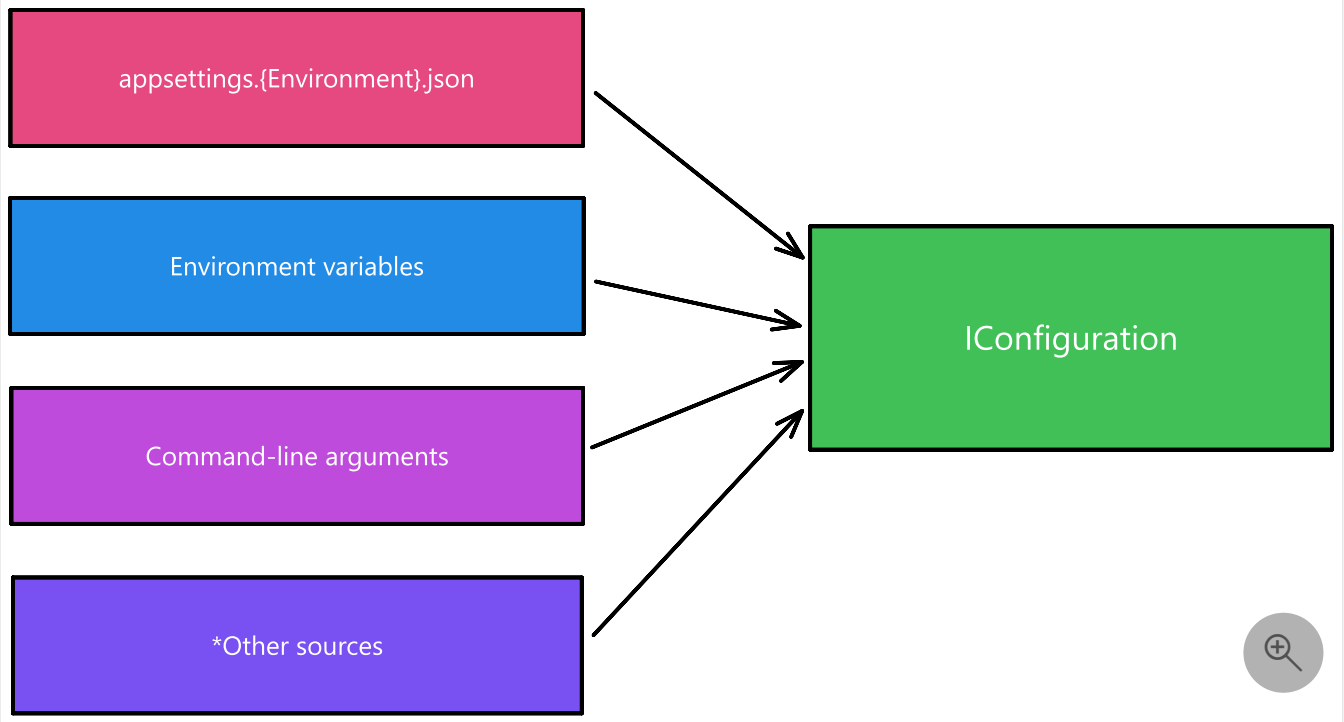
.NET의 구성은 하나 이상의구성 공급자를 사용하여 수행됩니다. 구성 공급자는 다양한 구성 원본을 사용하여 키-값 쌍에서 구성 데이터를 읽습니다.

- 설정 파일(예: *appsettings.json*
- 환경 변수
- Azure Key Vault
- Azure 앱 구성
- 명령줄 인수
- 사용자 지정 공급자(설치 또는 생성)
- 디렉터리 파일
- 메모리 내 .NET 개체
- 타사 공급자

> ⓘ **참고 항목**
>
> .NET 런타임 자체를 구성하는 방법에 대한 자세한 내용은 **.NET 런타임 구성 설정**참조하세요.

개념 및 추상화

하나 이상의 구성 원본이 지정된 경우 IConfiguration 형식은 구성 데이터의 통합 보기를 제공합니다. 구성은 읽기 전용이며 구성 패턴은 프로그래밍 방식으로 쓰기 가능하도록 설계되지 않았습니다. `IConfiguration` 인터페이스는 다음 다이어그램과 같이 모든 구성 원본의 단일 표현입니다.



콘솔 앱 구성

dotnet 새 명령 템플릿 또는 Visual Studio를 사용하여 만든 .NET 콘솔 앱은 기본적으로 구성 기능을 노출*하지 않습니다*. 새 .NET 콘솔 애플리케이션에서 구성을 추가하려면 Microsoft.Extensions.Configuration에 패키지 참조를 추가합니다. 이 패키지는 .NET 앱에서 구성을 위한 기초입니다. ConfigurationBuilder 및 관련 형식을 제공합니다.

C#

```
using Microsoft.Extensions.Configuration;

var configuration = new ConfigurationBuilder()
    .AddInMemoryCollection(new Dictionary<string, string?>()
    {
        ["SomeKey"] = "SomeValue"
    })
    .Build();

Console.WriteLine(configuration["SomeKey"]);

// Outputs:
//   SomeValue
```

앞의 코드는 다음과 같습니다.

- 새 ConfigurationBuilder 인스턴스를 만듭니다.
- 구성 작성기에서 키-값 쌍의 메모리 내 컬렉션을 추가합니다.
- Build() 메서드를 호출하여 IConfiguration 인스턴스를 만듭니다.
- `SomeKey` 키의 값을 콘솔에 씁니다.

이 예제에서는 메모리 내 구성을 사용하지만 파일 기반, 환경 변수, 명령줄 인수 및 기타 구성 원본에 대한 기능을 노출하는 많은 구성 공급자를 사용할 수 있습니다. 자세한 내용은 .NET 구성 공급자를 참조하세요.

대체 호스팅 방법

일반적으로 앱은 구성을 읽는 것 이상의 작업을 수행합니다. 종속성 주입, 로깅 및 기타 서비스를 사용할 가능성이 높습니다. 이러한 서비스를 사용하는 앱에는 .NET 제네릭 호스트 방법을 사용하는 것이 좋습니다. 대신 Microsoft.Extensions.Hosting에 패키지 참조를 추가하는 것이  좋습니다. 다음 코드와 일치하도록 *Program.cs* 파일을 수정합니다.

C#

```
using Microsoft.Extensions.Hosting;

using IHost host = Host.CreateApplicationBuilder(args).Build();

// Application code should start here.

await host.RunAsync();
```

Host.CreateApplicationBuilder(String[]) 메서드는 가장 높은 우선 순위에서 가장 낮은 우선 순위까지 다음 순서로 앱에 대한 기본 구성을 제공합니다.

1. 명령줄 인수는 명령줄 구성 공급자를 사용하여에서 제공합니다.
2. 환경 변수 구성 공급자를 통해 설정되는 환경 변수.
3. .
4. *appsettings.*`Environment`*JSON 구성 공급자*를 사용하여. 예를 들어 *appsettings.****생산****.json* 및 *appsettings.****개발****.json*.
5. *JSON 구성 공급자*사용하여 appsettings.json.
6. ChainedConfigurationProvider: 기존 `IConfiguration`를 원본으로 추가합니다.

구성 공급자를 추가하면 이전 구성 값이 재정의됩니다. 예를 들어 명령줄 구성 공급자 마지막으로 추가되었기 때문에 다른 공급자의 모든 값을 재정의합니다. `SomeKey` *appsettings.json* 환경 모두에서 설정된 경우 환경 값은 *appsettings.json*후에 추가되었기 때문에 사용됩니다.

바인딩

.NET 구성 추상화 사용의 주요 이점 중 하나는 구성 값을 .NET 개체의 인스턴스에 *바인딩* 하는 기능입니다. 예를 들어 JSON 구성 공급자는 *appsettings.json* 파일을 .NET 개체에 매핑하는 데 사용할 수 있으며 종속성 주입함께 사용됩니다. 이렇게 하면 클래스를 사용하여 관련 설정 그룹에 강력한 형식의 액세스를 제공하는 옵션 패턴사용할 수 있습니다. 기본 바인더는 리플렉션 기반이지만, 쉽게 활성화할 수 있는 원본 생성기 대체이 있습니다.

.NET 구성은 다양한 추상화 기능을 제공합니다. 다음 인터페이스를 고려합니다.

- IConfiguration: 키/값 애플리케이션 구성 속성 집합을 나타냅니다.
- IConfigurationRoot: `IConfiguration` 계층의 루트를 나타냅니다.
- IConfigurationSection: 애플리케이션 구성 값의 섹션을 나타냅니다.

이러한 추상화는 기본 구성 공급자(IConfigurationProvider)와 관련이 없습니다. 즉, `IConfiguration` 인스턴스를 사용하여 여러 공급자의 구성 값에 액세스할 수 있습니다.

바인더는 다양한 방법을 사용하여 구성 값을 처리할 수 있습니다.

- 기본 제공 변환기를 사용한 원시 타입의 직접 역직렬화
- 복합 형식에 유형이 하나 있을 때의 TypeConverter.
- 속성이 있는 복합 형식에 대한 리플렉션입니다.

> ⓘ **참고 항목**
>
> 바인더에는 몇 가지 제한 사항이 있습니다.
>
> - 속성은 프라이빗 setter가 있거나 해당 형식을 변환할 수 없는 경우 무시됩니다.
> - 해당 구성 키가 없는 속성은 무시됩니다.

바인딩 계층 구조

구성 값은 계층적 데이터를 포함할 수 있습니다. 계층적 개체는 구성 키에서 `:` 구분 기호를 사용하여 표시됩니다. 구성 값에 액세스하려면 `:` 문자를 사용하여 계층을 구분합니다. 예를 들어 다음 구성 값을 고려합니다.

JSON

```
{
  "Parent": {
    "FavoriteNumber": 7,
    "Child": {
      "Name": "Example",
      "GrandChild": {
        "Age": 3
      }
    }
  }
}
```

다음 표에서는 앞의 예제 JSON에 대한 예제 키와 해당 값을 나타냅니다.

테이블 확장

열쇠	값
`"Parent:FavoriteNumber"`	`7`
`"Parent:Child:Name"`	`"Example"`
`"Parent:Child:GrandChild:Age"`	`3`

고급 바인딩 시나리오

구성 바인더에는 특정 형식으로 작업할 때 특정 동작 및 제한 사항이 있습니다. 이 섹션에는 다음 하위 섹션이 포함되어 있습니다.

- 딕셔너리에 바인딩
- 콜론이 있는 사전 키
- IReadOnly* 형식에 바인딩
- 매개 변수가 있는 생성자를 사용하여 바인딩

딕셔너리에 바인딩

값이 변경 가능한 컬렉션 형식(예: 배열 또는 목록)인 위치에 구성 Dictionary<TKey,TValue> 을 바인딩하는 경우 동일한 키에 대한 반복 바인딩은 컬렉션 값을 바꾸는 대신 확장합니다.

다음 예제에서는 이 동작을 보여 줍니다.

C#

```
IConfiguration config = new ConfigurationBuilder()
    .AddInMemoryCollection()
    .Build();

config["Queue:0"] = "Value1";
var dict = new Dictionary<string, string[]>() { { "Queue", new[] { "InitialValue" }
} };

Console.WriteLine("=== Dictionary Binding with Collection Values ===");
Console.WriteLine($"Initially: {string.Join(", ", dict["Queue"])}");

// In .NET 7+, binding extends the collection instead of replacing it.
config.Bind(dict);
Console.WriteLine($"After Bind: {string.Join(", ", dict["Queue"])}");

config["Queue:1"] = "Value2";
config.Bind(dict);
Console.WriteLine($"After 2nd Bind: {string.Join(", ", dict["Queue"])}");
```

자세한 내용은 구성을 사전에 바인딩하여 값을 확장하는 방법을 참조하십시오.

콜론이 있는 사전 키

콜론(`:`) 문자는 구성 키에서 계층 구분 기호로 예약됩니다. 즉, 구성을 바인딩할 때 사전 키에서 콜론을 사용할 수 없습니다. 키에 콜론(예: URL 또는 기타 형식 식별자)이 포함된 경우 구성 시스템은 이를 리터럴 문자가 아닌 계층 경로로 해석합니다. 다음 해결 방법을 고려합니다.

- 구성 키에서 대체 구분 기호 문자(예: 이중 밑줄 `__`)를 사용하고 필요한 경우 프로그래밍 방식으로 변환합니다.
- System.Text.Json 또는 키에 콜론을 지원하는 유사한 라이브러리를 사용하여 구성을 원시 JSON으로 수동으로 역직렬화합니다.
- 콜론을 사용하여 안전한 키를 원하는 키로 변환하는 사용자 지정 매핑 계층을 만듭니다.

IReadOnly* 형식에 바인딩

구성 바인더는 `IReadOnlyList<T>` 및 `IReadOnlyDictionary<TKey, TValue>`와 같은 읽기 전용 컬렉션 인터페이스에 직접 바인딩하는 것을 지원하지 않습니다. 이러한 인터페이스에는 바인더가 컬렉션을 채우는 데 필요한 메커니즘이 부족합니다.

읽기 전용 컬렉션을 사용하려면 바인더가 채우는 속성에 변경 가능한 형식을 사용한 다음, 소비자를 위한 읽기 전용 인터페이스로 노출합니다.

C#

```
Console.WriteLine("=== IReadOnly* Types (NOT Directly Supported) ===");

var readonlyConfig = new ConfigurationBuilder()
    .AddInMemoryCollection(new Dictionary<string, string?>
    {
        ["Settings:Values:0"] = "Item1",
        ["Settings:Values:1"] = "Item2",
        ["Settings:Values:2"] = "Item3",
    })
    .Build();

// This class uses List<string> for binding, exposes as IReadOnlyList<string>.
var settings = new SettingsWithReadOnly();
readonlyConfig.GetSection("Settings").Bind(settings);

Console.WriteLine("Values bound to mutable List, exposed as IReadOnlyList:");
foreach (var value in settings.ValuesReadOnly)
{
    Console.WriteLine($"  {value}");
}
```

구성 클래스 구현:

C#

```
class SettingsWithReadOnly
{
    // Use mutable type for binding
    public List<string> Values { get; set; } = [];

    // Expose as read-only for consumers
    public IReadOnlyList<string> ValuesReadOnly => Values;
}
```

이 접근 방식을 사용하면 바인더가 소비자에게 변경할 수 없는 `IReadOnlyList<string>` 인터페이스를 제공하는 동안 변경 가능한 `List<string>`를 구성할 수 있습니다.

매개 변수가 있는 생성자를 사용하여 바인딩

.NET 7부터 구성 바인더는 매개 변수가 있는 단일 공용 생성자를 사용하여 형식에 대한 바인딩을 지원합니다. 이렇게 하면 변경할 수 없는 형식 및 레코드를 구성에서 직접 채울 수 있습니다.

C#

```
Console.WriteLine("=== Parameterized Constructor Binding ===");

var ctorConfig = new ConfigurationBuilder()
    .AddInMemoryCollection(new Dictionary<string, string?>
    {
        ["AppSettings:Name"] = "MyApp",
        ["AppSettings:MaxConnections"] = "100",
        ["AppSettings:Timeout"] = "30"
    })
    .Build();

// Binding to a type with a single parameterized constructor
var appSettings = ctorConfig.GetSection("AppSettings").Get<AppSettings>();
if (appSettings != null)
{
    Console.WriteLine($"Name: {appSettings.Name}");
    Console.WriteLine($"MaxConnections: {appSettings.MaxConnections}");
    Console.WriteLine($"Timeout: {appSettings.Timeout}");
}
```

변경할 수 없는 설정 클래스:

C#

```
// Immutable type with single parameterized constructor.
class AppSettings
```

```
{
    public string Name { get; }
    public int MaxConnections { get; }
    public int Timeout { get; }

    public AppSettings(string name, int maxConnections, int timeout)
    {
        Name = name;
        MaxConnections = maxConnections;
        Timeout = timeout;
    }
}
```

> **ⓘ Important**
>
> 바인더는 매개 변수가 있는 단일 공용 생성자가 있는 형식만 지원합니다. 형식에 여러 공용 매개 변수가 있는 생성자가 있는 경우 바인더는 사용할 생성자를 결정할 수 없으며 바인딩이 실패합니다. 매개 변수가 있는 단일 생성자 또는 속성 setter와 함께 매개 변수가 없는 생성자를 사용합니다.

기본 예제

제네릭 호스트 접근 방식의 도움 없이 기본 형식의 구성 값에 액세스하려면 ConfigurationBuilder 형식을 직접 사용합니다.

> **팁**
>
> System.Configuration.ConfigurationBuilder 형식은 Microsoft.Extensions.Configuration.ConfigurationBuilder 형식과 다릅니다. 이 모든 콘텐츠는 `Microsoft.Extensions.*` NuGet 패키지 및 네임스페이스와 관련이 있습니다.

다음 C# 프로젝트를 고려합니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
```

```
    <Content Include="appsettings.json">
      <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </Content>
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder"
Version="10.0.3" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json"
Version="10.0.3" />
    <PackageReference
Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="10.0.3"
/>
  </ItemGroup>

</Project>
```

위의 프로젝트 파일은 다음과 같은 여러 구성 NuGet 패키지를 참조합니다.

- Microsoft.Extensions.Configuration.Binder: 을 위해 구성 공급자에서 개체를 데이터에 바인딩하는 기능입니다.
- Microsoft.Extensions.Configuration.Json: `Microsoft.Extensions.Configuration`대한 JSON 구성 공급자 구현입니다.
- Microsoft.Extensions.Configuration.EnvironmentVariables: `Microsoft.Extensions.Configuration`대한 환경 변수 구성 공급자 구현입니다.

예제 *appsettings.json* 파일을 고려합니다.

JSON

```
{
    "Settings": {
        "KeyOne": 1,
        "KeyTwo": true,
        "KeyThree": {
            "Message": "Oh, that's nice...",
            "SupportedVersions": {
                "v1": "1.0.0",
                "v3": "3.0.7"
            }
        },
        "IPAddressRange": [
            "46.36.198.121",
            "46.36.198.122",
            "46.36.198.123",
            "46.36.198.124",
            "46.36.198.125"
        ]
    }
}
```

이제 이 JSON 파일을 고려할 때 구성 작성기를 직접 사용하는 예제 소비 패턴은 다음과 같습니다.

C#

```
using Microsoft.Extensions.Configuration;

// Build a config object, using env vars and JSON providers.
IConfigurationRoot config = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json")
    .AddEnvironmentVariables()
    .Build();

// Get values from the config given their key and their target type.
Settings? settings = config.GetRequiredSection("Settings").Get<Settings>();

// Write the values to the console.
Console.WriteLine($"KeyOne = {settings?.KeyOne}");
Console.WriteLine($"KeyTwo = {settings?.KeyTwo}");
Console.WriteLine($"KeyThree:Message = {settings?.KeyThree?.Message}");

// Application code which might rely on the config could start here.

// This will output the following:
//   KeyOne = 1
//   KeyTwo = True
//   KeyThree:Message = Oh, that's nice...
```

앞의 C# 코드는 다음과 같습니다.

- ConfigurationBuilder를 (하나의) 인스턴스화합니다.
- JSON 구성 공급자가 인식할 `"appsettings.json"` 파일을 추가합니다.
- 환경 변수 구성 공급자가 인식할 수 있는 환경 변수를 추가합니다.
- `"Settings"` 인스턴스를 사용하여 필요한 `Settings` 섹션 및 해당 `config` 인스턴스를 가져옵니다.

`Settings` 개체의 모양은 다음과 같습니다.

C#

```
public sealed class Settings
{
    public required int KeyOne { get; set; }
    public required bool KeyTwo { get; set; }
    public required NestedSettings KeyThree { get; set; } = null!;
}
```

C#

```
public sealed class NestedSettings
{
    public required string Message { get; set; } = null!;
}
```

호스팅을 사용하여 기본 예제

`IConfiguration` 값에 액세스하려면 Microsoft.Extensions.Hosting NuGet 패키지를 다시 사용할 수 있습니다. 새 콘솔 애플리케이션을 만들고 다음 프로젝트 파일 내용을 붙여넣습니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="appsettings.json">
      <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </Content>
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder"
Version="10.0.3" />
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="10.0.3" />
  </ItemGroup>

</Project>
```

위의 프로젝트 파일은 다음을 정의합니다.

- 애플리케이션이 실행 파일입니다.
- *appsettings.json* 파일은 프로젝트가 컴파일될 때 출력 디렉터리에 복사됩니다.
- `Microsoft.Extensions.Hosting` NuGet 패키지 참조가 추가됩니다.

다음 내용을 사용하여 프로젝트의 루트에 *appsettings.json* 파일을 추가합니다.

JSON

```
{
    "KeyOne": 1,
    "KeyTwo": true,
    "KeyThree": {
```

```
        "Message": "Thanks for checking this out!"
    }
}
```

Program.cs 파일의 내용을 다음 C# 코드로 바꿉니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

using IHost host = Host.CreateApplicationBuilder(args).Build();

// Ask the service provider for the configuration abstraction.
IConfiguration config = host.Services.GetRequiredService<IConfiguration>();

// Get values from the config given their key and their target type.
int keyOneValue = config.GetValue<int>("KeyOne");
bool keyTwoValue = config.GetValue<bool>("KeyTwo");
string? keyThreeNestedValue = config.GetValue<string>("KeyThree:Message");

// Write the values to the console.
Console.WriteLine($"KeyOne = {keyOneValue}");
Console.WriteLine($"KeyTwo = {keyTwoValue}");
Console.WriteLine($"KeyThree:Message = {keyThreeNestedValue}");

// Application code which might rely on the config could start here.

await host.RunAsync();

// This will output the following:
//   KeyOne = 1
//   KeyTwo = True
//   KeyThree:Message = Thanks for checking this out!
```

이 애플리케이션을 실행할 때 `Host.CreateApplicationBuilder` JSON 구성을 검색하고 `IConfiguration` 인스턴스를 통해 노출하는 동작을 정의합니다. `host` 인스턴스에서 서비스 공급자에게 `IConfiguration` 인스턴스를 요청한 다음 값을 요청할 수 있습니다.

> **팁**
>
> 이러한 방식으로 원시 `IConfiguration` 인스턴스를 사용하면 편리하지만 크기가 잘 조정되지 않습니다. 애플리케이션의 복잡성이 증가하고 해당 구성이 더 복잡해지면 ***옵션 패턴*** 대신 사용하는 것이 좋습니다.

인덱서 API를 호스팅하고 사용하는 기본 예제

이전 예제와 동일한 *appsettings.json* 파일 내용을 고려합니다.

JSON

```
{
    "SupportedVersions": {
        "v1": "1.0.0",
        "v3": "3.0.7"
    },
    "IPAddressRange": [
        "46.36.198.123",
        "46.36.198.124",
        "46.36.198.125"
    ]
}
```

Program.cs 파일의 내용을 다음 C# 코드로 바꿉니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

using IHost host = Host.CreateApplicationBuilder(args).Build();

// Ask the service provider for the configuration abstraction.
IConfiguration config = host.Services.GetRequiredService<IConfiguration>();

// Get values from the config given their key and their target type.
string? ipOne = config["IPAddressRange:0"];
string? ipTwo = config["IPAddressRange:1"];
string? ipThree = config["IPAddressRange:2"];
string? versionOne = config["SupportedVersions:v1"];
string? versionThree = config["SupportedVersions:v3"];

// Write the values to the console.
Console.WriteLine($"IPAddressRange:0 = {ipOne}");
Console.WriteLine($"IPAddressRange:1 = {ipTwo}");
Console.WriteLine($"IPAddressRange:2 = {ipThree}");
Console.WriteLine($"SupportedVersions:v1 = {versionOne}");
Console.WriteLine($"SupportedVersions:v3 = {versionThree}");

// Application code which might rely on the config could start here.

await host.RunAsync();

// This will output the following:
//     IPAddressRange:0 = 46.36.198.123
//     IPAddressRange:1 = 46.36.198.124
//     IPAddressRange:2 = 46.36.198.125
```

```
//     SupportedVersions:v1 = 1.0.0
//     SupportedVersions:v3 = 3.0.7
```

각 키가 문자열이고 값이 문자열인 인덱서 API를 사용하여 값에 액세스합니다. 구성은 속성, 개체, 배열 및 사전을 지원합니다.

구성 공급자

다음 표에서는 .NET Core 앱에서 사용할 수 있는 구성 공급자를 보여 줍니다.

테이블 확장

구성 공급자	에서 구성을 제공합니다.
Azure 앱 구성	Azure App Configuration (애저 앱 설정)
Azure Key Vault	Azure Key Vault (애저 키 볼트)
명령줄	명령줄 매개 변수
사용자 지정	사용자 지정 원본
환경 변수	환경 변수
파일	JSON, XML 및 INI 파일
파일당 키	디렉터리 파일
기억	메모리 내 컬렉션
앱 비밀(비밀 관리자)	사용자 프로필 디렉터리의 파일

> **팁**
>
> 구성 공급자가 추가되는 순서가 중요합니다. 여러 구성 공급자를 사용하고 둘 이상의 공급자가 동일한 키를 지정하는 경우 마지막으로 추가된 공급자가 사용됩니다.

다양한 구성 공급자에 대한 자세한 내용은 .NET 구성 공급자를 참조하세요.

또한 참고하세요

- .NET의 구성 공급자
- 사용자 지정 구성 공급자 구현
- 구성 버그를 github.com/dotnet/runtime 리포지토리에 만들어야 합니다.
- ASP.NET Core에서 구성

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 09.

.NET의 구성 공급자

아티클 • 2024. 12. 16.

.NET에서 구성은 구성 공급자를 사용하여 수행할 수 있습니다. 여러 유형의 공급자는 다양한 구성 원본을 사용합니다. 이 문서에서는 모든 다양한 구성 공급자와 해당 소스를 자세히 설명합니다.

- 파일 구성 공급자
- 환경 변수 구성 공급자
- 명령줄 구성 공급자
- 파일별 키 구성 공급자
- 메모리 구성 공급자

파일 구성 공급자

FileConfigurationProvider는 파일 시스템에서 구성을 로드하기 위한 기본 클래스입니다. 다음 구성 공급자는 `FileConfigurationProvider`에서 파생됩니다.

- JSON 구성 공급자
- XML 구성 공급자
- INI 구성 공급자

키는 대/소문자를 구분하지 않습니다. 모든 파일 구성 공급자는 단일 공급자에서 중복 키를 찾을 때 FormatException을 throw합니다.

JSON 구성 공급자

JsonConfigurationProvider 클래스는 JSON 파일에서 구성을 로드합니다. Microsoft.Extensions.Configuration.Json NuGet 패키지를 설치합니다.

오버로드는 다음을 지정할 수 있습니다.

- 파일이 선택 사항인지 여부
- 파일이 변경되면 구성을 다시 로드하는지 여부

다음 코드를 생각해 봅시다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using ConsoleJson.Example;
```

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.Sources.Clear();

IHostEnvironment env = builder.Environment;

builder.Configuration
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true);

TransientFaultHandlingOptions options = new();
builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
    .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

앞의 코드가 하는 역할은 다음과 같습니다.

- CreateApplicationBuilder(String[]) 메서드에 기본적으로 추가된 기존 구성 공급자를 모두 지웁니다.
- 다음 옵션을 사용하여 *appsettings.json* 및 *appsettings.*`Environment`*.json* 파일을 로드하도록 JSON 구성 공급자를 구성합니다.
  - `optional: true`: 파일은 선택 사항입니다.
  - `reloadOnChange: true`: 변경 내용이 저장되면 파일이 다시 로드됩니다.

> **ⓘ 중요**
>
> 로 IConfigurationBuilder.Add할 때 추가된 구성 공급자가 `IConfigurationSource` 목록의 끝에 추가됩니다. 여러 공급자가 키를 찾은 경우 키를 읽는 마지막 공급자가 이전 공급자를 재정의합니다.

다양한 구성 설정을 사용하는 예제 *appsettings.json* 파일은 다음과 같습니다.

JSON

```
{
    "SecretKey": "Secret key value",
    "TransientFaultHandlingOptions": {
        "Enabled": true,
```

```
        "AutoRetryDelay": "00:00:07"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

구성 공급자가 추가된 후 IConfigurationBuilder 인스턴스에서 IConfigurationBuilder.Build()를 호출하여 IConfigurationRoot 개체를 가져올 수 있습니다. 구성 루트는 구성 계층 구조의 루트를 나타냅니다. 구성의 섹션은 .NET 개체의 인스턴스에 바인딩되고 나중에 IOptions<TOptions> 종속성 주입을 통해 제공될 수 있습니다.

> **참고**
>
> JSON 파일의 *빌드 작업* 및 *출력 디렉터리*에 복사 속성을 각각 *콘텐츠* 및 *새 버전이면 복사(또는 항상 복사)*로 설정해야 합니다.

다음과 같이 정의된 `TransientFaultHandlingOptions` 클래스를 고려하세요.

C#

```
namespace ConsoleJson.Example;

public sealed class TransientFaultHandlingOptions
{
    public bool Enabled { get; set; }
    public TimeSpan AutoRetryDelay { get; set; }
}
```

다음 코드는 구성 루트를 빌드하고, 섹션을 `TransientFaultHandlingOptions` 클래스 형식에 바인딩하고, 바인딩된 값을 콘솔 창에 출력합니다.

C#

```
TransientFaultHandlingOptions options = new();
builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
    .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");
```

애플리케이션은 다음 샘플 출력을 씁니다.

C#

```
// Sample output:
//    TransientFaultHandlingOptions.Enabled=True
//    TransientFaultHandlingOptions.AutoRetryDelay=00:00:07
```

XML 구성 공급자

XmlConfigurationProvider 클래스는 런타임에 있는 XML 파일에서 구성을 로드합니다. Microsoft.Extensions.Configuration.Xml NuGet 패키지를 설치합니다.

다음 코드에서는 XML 구성 공급자를 사용한 XML 파일의 구성을 보여 줍니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.Sources.Clear();

builder.Configuration
    .AddXmlFile("appsettings.xml", optional: true, reloadOnChange: true)
    .AddXmlFile("repeating-example.xml", optional: true, reloadOnChange:
true);

builder.Configuration.AddEnvironmentVariables();

if (args is { Length: > 0 })
{
    builder.Configuration.AddCommandLine(args);
}

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

앞의 코드가 하는 역할은 다음과 같습니다.

- CreateApplicationBuilder(String[]) 메서드에 기본적으로 추가된 기존 구성 공급자를 모두 지웁니다.
- 다음 옵션을 사용하여 *appsettings.xml* 및 *repeating-example.xml* 파일을 로드하도록 XML 구성 공급자를 구성합니다.

    - `optional: true`: 파일은 선택 사항입니다.
    - `reloadOnChange: true`: 변경 내용이 저장되면 파일이 다시 로드됩니다.
- 환경 변수 구성 공급자를 구성합니다.
- 지정된 `args`에 인수가 포함된 경우 명령줄 구성 공급자를 구성합니다.

XML 설정은 환경 변수 구성 공급자 및 명령줄 구성 공급자의 설정에 따라 재정의됩니다.

다양한 구성 설정을 사용하는 예제 *appsettings.xml* 파일은 다음과 같습니다.

XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <SecretKey>Secret key value</SecretKey>
  <TransientFaultHandlingOptions>
    <Enabled>true</Enabled>
    <AutoRetryDelay>00:00:07</AutoRetryDelay>
  </TransientFaultHandlingOptions>
  <Logging>
    <LogLevel>
      <Default>Information</Default>
      <Microsoft>Warning</Microsoft>
    </LogLevel>
  </Logging>
</configuration>
```

팁

WinForms 앱에서 `IConfiguration` 형식을 사용하려면 Microsoft.Extensions.Configuration.Xml NuGet 패키지에 대한 참조를 추가합니다. ConfigurationBuilder를 인스턴스화하고 AddXmlFile 및 Build()에 대한 호출을 연결합니다. 자세한 내용은 .NET Docs 이슈 #29679를 참조하세요.

.NET 5 이하 버전에서는 동일한 요소 이름을 사용하는 반복 요소를 구분하는 `name` 특성을 추가합니다. .NET 6 이상 버전에서 XML 구성 공급자는 반복 요소를 자동으로 인덱싱합니다. 즉, 키에 "0" 인덱스를 사용하고 요소가 하나만 있는 경우를 제외하고는 `name` 특성을 지정할 필요가 없습니다. (.NET 6 이상으로 업그레이드하는 경우 이 동작 변경으로 인해 중단이 발생할 수 있습니다. 자세한 내용은 반복 XML 요소에 인덱스 포함을 참조하세요.)

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
```

```
    <key name="key0">value 00</key>
    <key name="key1">value 01</key>
  </section>
  <section name="section1">
    <key name="key0">value 10</key>
    <key name="key1">value 11</key>
  </section>
</configuration>
```

다음 코드는 이전 구성 파일을 읽고 키 및 값을 표시합니다.

C#

```
IConfigurationRoot configurationRoot = builder.Configuration;

string key00 = "section:section0:key:key0";
string key01 = "section:section0:key:key1";
string key10 = "section:section1:key:key0";
string key11 = "section:section1:key:key1";

string? val00 = configurationRoot[key00];
string? val01 = configurationRoot[key01];
string? val10 = configurationRoot[key10];
string? val11 = configurationRoot[key11];

Console.WriteLine($"{key00} = {val00}");
Console.WriteLine($"{key01} = {val01}");
Console.WriteLine($"{key10} = {val10}");
Console.WriteLine($"{key10} = {val11}");
```

애플리케이션은 다음 샘플 출력을 작성합니다.

C#

```
// Sample output:
//    section:section0:key:key0 = value 00
//    section:section0:key:key1 = value 01
//    section:section1:key:key0 = value 10
//    section:section1:key:key0 = value 11
```

특성을 사용하여 값을 제공할 수 있습니다.

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
```

```
  </section>
</configuration>
```

이전 구성 파일은 `value`와 함께 다음 키를 로드합니다.

- `key:attribute`
- `section:key:attribute`

INI 구성 공급자

IniConfigurationProvider 클래스는 런타임에 있는 INI 파일에서 구성을 로드합니다. Microsoft.Extensions.Configuration.Ini NuGet 패키지를 설치합니다.

다음 코드는 모든 구성 공급자를 지우고 두 개의 INI 파일을 소스로 사용하여 `IniConfigurationProvider`를 추가합니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Configuration.Sources.Clear();

IHostEnvironment env = builder.Environment;

builder.Configuration
    .AddIniFile("appsettings.ini", optional: true, reloadOnChange: true)
    .AddIniFile($"appsettings.{env.EnvironmentName}.ini", true, true);

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

다양한 구성 설정을 사용하는 예제 *appsettings.ini* 파일은 다음과 같습니다.

ini

```
SecretKey="Secret key value"

[TransientFaultHandlingOptions]
Enabled=True
AutoRetryDelay="00:00:07"

[Logging:LogLevel]
```

```
Default=Information
Microsoft=Warning
```

다음 코드는 이전 구성 설정을 콘솔 창에 기록하여 표시합니다.

C#

```
foreach ((string key, string? value) in
    builder.Configuration.AsEnumerable().Where(t => t.Value is not null))
{
    Console.WriteLine($"{key}={value}");
}
```

애플리케이션은 다음 샘플 출력을 작성합니다.

C#

```
// Sample output:
//    TransientFaultHandlingOptions:Enabled=True
//    TransientFaultHandlingOptions:AutoRetryDelay=00:00:07
//    SecretKey=Secret key value
//    Logging:LogLevel:Microsoft=Warning
//    Logging:LogLevel:Default=Information
```

환경 변수 구성 공급자

기본 구성을 사용하여 EnvironmentVariablesConfigurationProvider는 *appsettings.json*, *appsettings.*`Environment`*.json*, 비밀 관리자를 읽은 후 환경 변수 키-값 쌍에서 구성을 로드합니다. 따라서 환경에서 읽은 키 값이 *appsettings.json*, *appsettings.*`Environment`*.json* 및 비밀 관리자에서 읽은 값을 재정의합니다.

`:` 구분 기호는 모든 플랫폼의 환경 변수 계층적 키에서 작동하지 않습니다. 예를 들어 `:` 구분 기호는 Bash 에서 지원되지 않습니다. 모든 플랫폼에서 지원되는 이중 밑줄(`__`)은 환경 변수의 모든 `:` 구분 기호를 자동으로 바꿉니다.

`TransientFaultHandlingOptions` 클래스를 살펴보겠습니다.

C#

```
public class TransientFaultHandlingOptions
{
    public bool Enabled { get; set; }
    public TimeSpan AutoRetryDelay { get; set; }
}
```

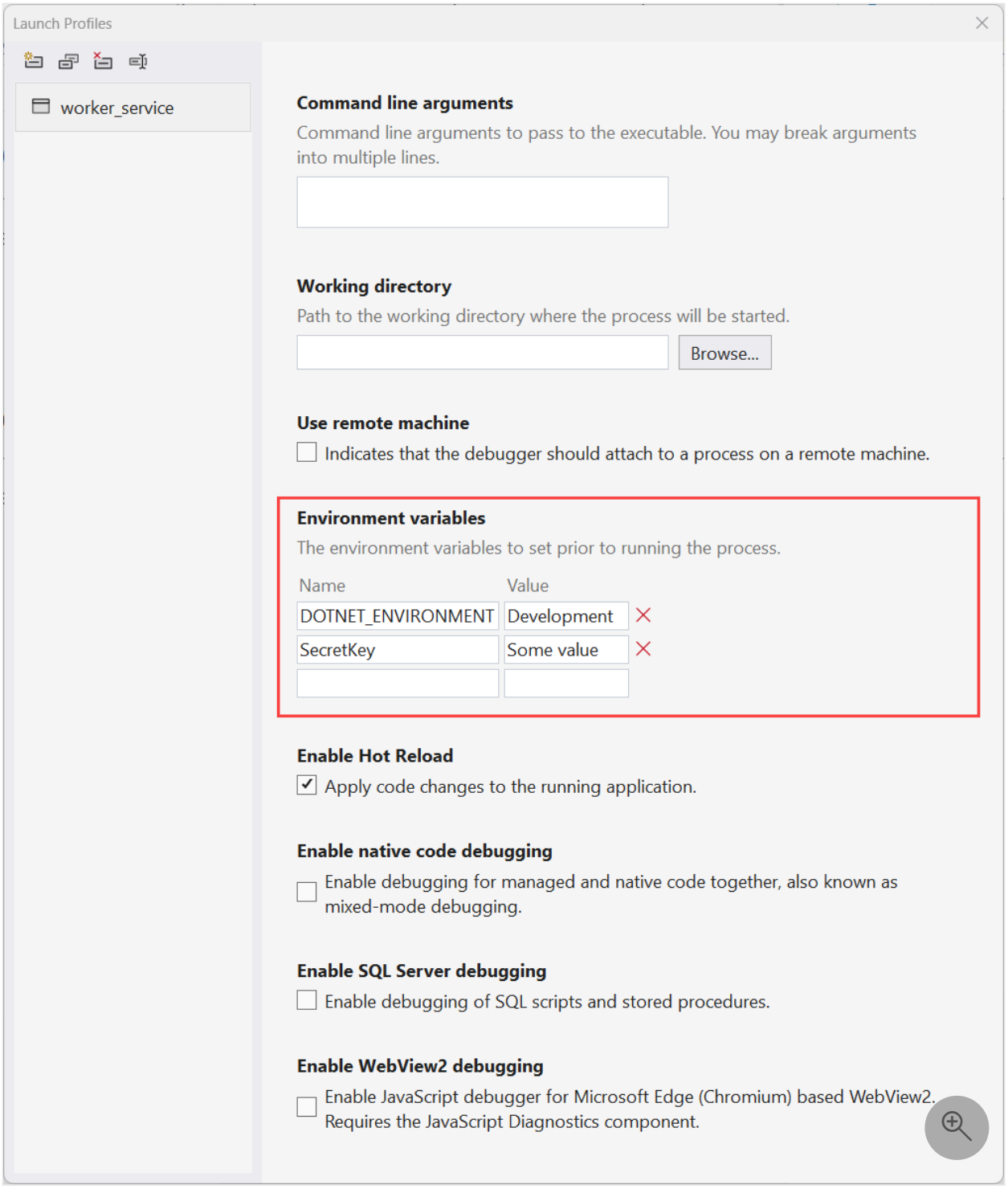
다음 `set` 명령은 `SecretKey` 및 `TransientFaultHandlingOptions`의 환경 키와 값을 설정합니다.

.NET CLI

```
set SecretKey="Secret key from environment"
set TransientFaultHandlingOptions__Enabled="true"
set TransientFaultHandlingOptions__AutoRetryDelay="00:00:13"
```

이러한 환경 설정은 설정된 명령 창에서 시작된 프로세스에서만 설정됩니다. Visual Studio에서 시작된 웹앱에서는 읽지 않습니다.

Visual Studio 2019 이상에서는 **시작 프로필** 대화 상자를 사용하여 환경 변수를 지정할 수 있습니다.



다음 setx 명령을 사용하여 Windows에서 환경 키 및 값을 설정할 수 있습니다. `set`와 달리, `setx` 설정은 유지됩니다. `/M`은 시스템 환경에서 변수를 설정합니다. `/M` 스위치를 사용하지 않으면 사용자 환경 변수가 설정됩니다.

.NET CLI

```
setx SecretKey "Secret key from setx environment" /M
setx TransientFaultHandlingOptions__Enabled "true" /M
setx TransientFaultHandlingOptions__AutoRetryDelay "00:00:05" /M
```

이전 명령이 *apsettings.json* 및 *appsettings.*`Environment`*json* 설정을 재정의하는지 테스트하려면:

- Visual Studio 사용: Visual Studio를 종료하고 다시 시작합니다.
- CLI 사용: 새 명령 창을 시작하고 `dotnet run`을 입력합니다.

접두사

환경 변수에 대한 접두사를 지정하려면 문자열을 사용하여 AddEnvironmentVariables를 호출합니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddEnvironmentVariables(prefix: "CustomPrefix_");
using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

위의 코드에서

- `config.AddEnvironmentVariables(prefix: "CustomPrefix_")`는 기본 구성 공급자 뒤에 추가됩니다. 구성 공급자 순서 지정 예제는 XML 구성 공급자를 참조하세요.
- `CustomPrefix_` 접두사를 사용하여 설정된 환경 변수는 기본 구성 공급자를 재정의합니다. 여기에는 접두사 없는 환경 변수가 포함됩니다.

구성 키-값 쌍을 읽으면 접두사는 제거됩니다.

기본 구성은 `DOTNET_` 접두사가 붙은 환경 변수 및 명령줄 인수를 로드합니다. `DOTNET_` 접두사는 .NET에서 호스트 및 앱 구성에 사용되지만 사용자 구성에는 사용되지 않습니다.

호스트 및 앱 구성에 대한 자세한 내용은 .NET 제네릭 호스트를 참조하세요.

연결 문자열 접두사

구성 API에는 네 개의 연결 문자열 환경 변수에 대한 특별한 처리 규칙이 있습니다. 해당 연결 문자열은 앱 환경의 Azure 연결 문자열을 구성하는 데 관련됩니다. 기본 구성을 사용하거나 `AddEnvironmentVariables`에 제공된 접두사가 없는 경우 표에 표시된 접두사가 붙은 환경 변수가 앱에 로드됩니다.

테이블 확장

연결 문자열 접두사	공급자
`CUSTOMCONNSTR_`	사용자 지정 공급자
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL Database
`SQLCONNSTR_`	SQL Server

표에 표시된 네 개 접두사 중 하나가 붙은 환경 변수가 검색되어 구성으로 로드되면 다음과 같이 됩니다.

- 환경 변수 접두사를 제거하고 구성 키 섹션(`ConnectionStrings`)을 추가하여 구성 키가 생성됩니다.
- 데이터베이스 연결 제공자(지정된 공급자가 없는 `CUSTOMCONNSTR_` 제외)를 나타내는 새 구성 키-값 쌍이 생성됩니다.

테이블 확장

환경 변수 키	변환된 구성 키	공급자 구성 항목
`CUSTOMCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	구성 항목이 생성되지 않습니다.
`MYSQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	키: `ConnectionStrings:{KEY}_ProviderName`: 값: `MySql.Data.MySqlClient`
`SQLAZURECONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	키: `ConnectionStrings:{KEY}_ProviderName`: 값: `System.Data.SqlClient`
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	키: `ConnectionStrings:{KEY}_ProviderName`: 값: `System.Data.SqlClient`

> **중요**
>
> 사용 가능한 가장 안전한 인증 흐름을 사용하는 것이 권장됩니다. Azure SQL에 연결하려는 경우, 권장되는 인증 방법은 **Azure 리소스에 대한 관리 ID**입니다.

launchSettings.json에 설정된 환경 변수

*launchSettings.json*에 설정된 환경 변수는 시스템 환경에 설정된 변수를 재정의합니다.

Azure App Service 설정

Azure App Service설정환경 변수 페이지에서 추가 선택합니다. Azure App Service 애플리케이션 설정은,

- 미사용 시 암호화되고 암호화된 채널을 통해 전송됩니다.
- 환경 변수로 노출됩니다.

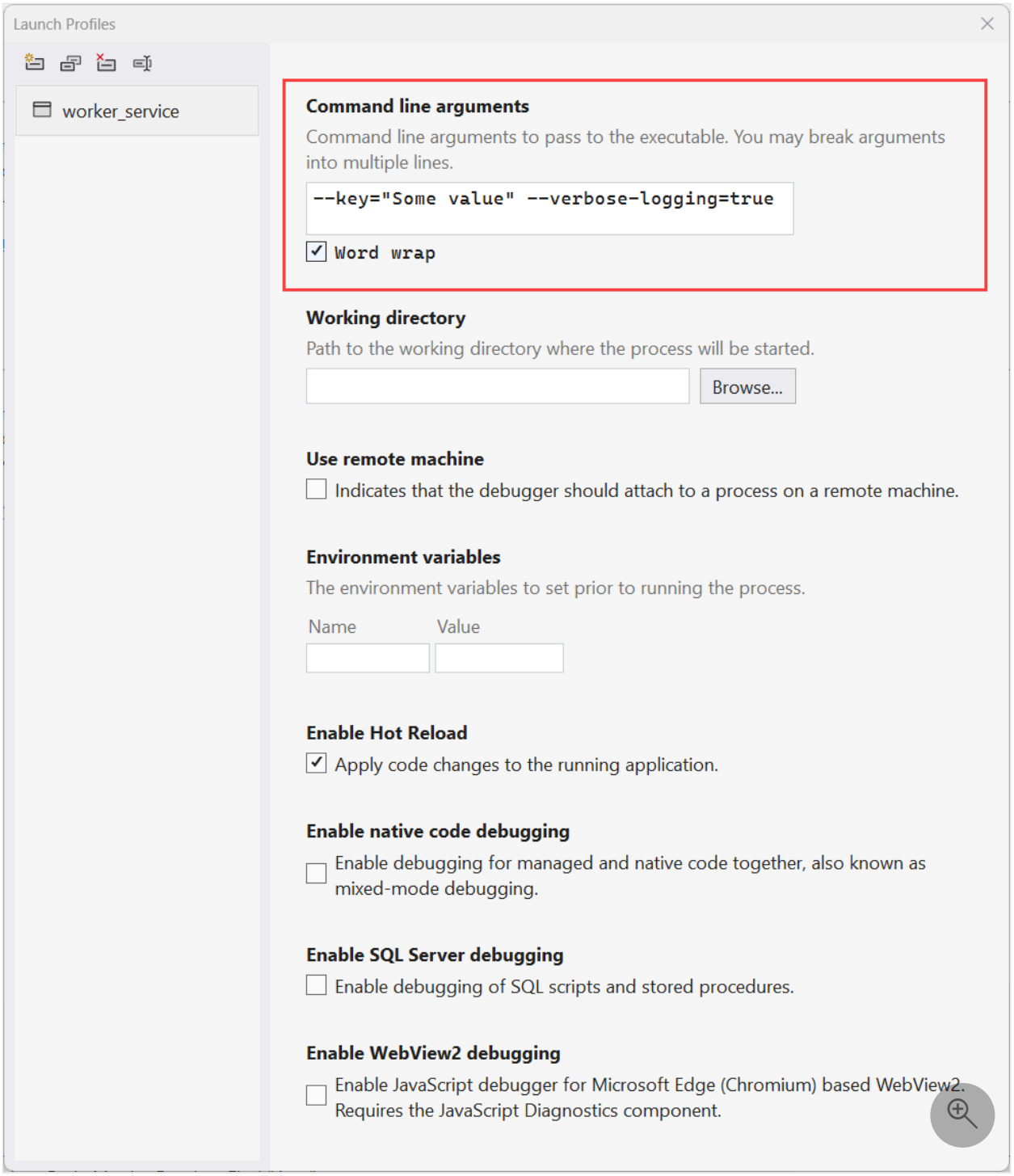
명령줄 구성 공급자

기본 구성을 사용하여 CommandLineConfigurationProvider는 다음 구성 소스 뒤에 명령줄 인수 키-값 쌍에서 구성을 로드합니다.

- *appsettings.json* 및 *appsettings.*`Environment`*.json* 파일
- `Development` 발 환경의 앱 비밀(비밀 관리자)
- 환경 변수입니다.

기본적으로 명령줄에 설정된 구성 값은 다른 모든 구성 공급자를 사용하여 설정된 구성 값을 재정의합니다.

Visual Studio 2019 이상에서는 **시작 프로필** 대화 상자를 사용하여 명령줄 인수를 지정할 수 있습니다.



명령줄 인수

다음 명령은 `=`를 사용하여 키 및 값을 설정합니다.

.NET CLI

```
dotnet run SecretKey="Secret key from command line"
```

다음 명령은 `/`를 사용하여 키 및 값을 설정합니다.

.NET CLI

```
dotnet run /SecretKey "Secret key set from forward slash"
```

다음 명령은 `--`를 사용하여 키 및 값을 설정합니다.

.NET CLI

```
dotnet run --SecretKey "Secret key set from double hyphen"
```

키 값은,

- `=` 다음에 와야 합니다. 또는 값이 공백에 다음에 오는 경우 키에 `--` 또는 `/` 접두사가 있어야 합니다.
- `=`이 사용된 경우 필요하지 않습니다. 예: `SomeKey=`.

같은 명령 내에서 `=`을 사용하는 명령줄 인수 키-값 쌍을 공백을 사용하는 키-값 쌍과 함께 사용하지 마세요.

파일별 키 구성 공급자

KeyPerFileConfigurationProvider는 디렉터리의 파일을 구성 키-값 쌍으로 사용합니다. 키는 파일 이름이고, 값은 파일의 콘텐츠입니다. 파일별 키 구성 공급자는 Docker 호스팅 시나리오에서 사용됩니다.

파일별 키 구성을 활성화하려면 AddKeyPerFile 인스턴스에서 ConfigurationBuilder 확장 메서드를 호출합니다. 파일의 `directoryPath`는 절대 경로여야 합니다.

오버로드에서는 다음을 지정할 수 있습니다.

- 소스를 구성하는 `Action<KeyPerFileConfigurationSource>` 대리자
- 디렉터리가 선택 사항인지 여부와 디렉터리의 경로

두 개의 밑줄(`__`)은 파일 이름에서 구성 키 구분 기호로 사용됩니다. 예를 들어, 파일 이름 `Logging__LogLevel__System`은 구성 키 `Logging:LogLevel:System`을 생성합니다.

호스트를 빌드할 때 `ConfigureAppConfiguration`을 호출하여 앱의 구성을 지정합니다.

C#

```
.ConfigureAppConfiguration((_, configuration) =>
{
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");
```

```
    configuration.AddKeyPerFile(directoryPath: path, optional: true);
})
```

메모리 구성 공급자

MemoryConfigurationProvider는 메모리 내 컬렉션을 구성 키-값 쌍으로 사용합니다.

다음 코드는 구성 시스템에 메모리 컬렉션을 추가합니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddInMemoryCollection(
    new Dictionary<string, string?>
    {
        ["SecretKey"] = "Dictionary MyKey Value",
        ["TransientFaultHandlingOptions:Enabled"] = bool.TrueString,
        ["TransientFaultHandlingOptions:AutoRetryDelay"] = "00:00:07",
        ["Logging:LogLevel:Default"] = "Warning"
    });

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

이전 코드에서 MemoryConfigurationBuilderExtensions.AddInMemoryCollection(IConfigurationBuilder, IEnumerable<KeyValuePair<String,String>>)은 기본 구성 공급자 뒤에 메모리 공급자를 추가합니다. 구성 공급자 순서 지정 예제는 XML 구성 공급자를 참조하세요.

참고 항목

- .NET의 구성
- .NET 일반 호스트
- 사용자 지정 구성 공급자 구현

구성 원본 생성기

아티클 • 2025. 03. 22.

.NET 8부터 특정 호출 사이트를 가로채 해당 기능을 생성하는 구성 바인딩 원본 생성기가 도입되었습니다. 이 기능은 리플렉션 기반 구현을 사용하지 않고도 네이티브 AOT(Ahead-Of-Time) 및 트리밍 친화적인 구성 바인더를 사용하는 방법을 제공합니다. 리플렉션에는 AOT 시나리오에서 지원되지 않는 동적 코드 생성이 필요합니다.

이 기능은 C# 12에서 도입된 C# 인터셉터가 등장하면서 가능합니다. 인터셉터를 사용하면 컴파일러가 특정 호출을 가로채 생성된 코드로 대체하는 소스 코드를 생성할 수 있습니다.

구성 원본 생성기 사용

구성 원본 생성기를 사용하도록 설정하려면 프로젝트 파일에 다음 속성을 추가합니다.

XML

```
<PropertyGroup>

<EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerator>
</PropertyGroup>
```

구성 원본 생성기를 사용하도록 설정하면 컴파일러는 구성 바인딩 코드를 포함하는 소스 파일을 생성합니다. 생성된 원본은 다음 클래스에서 바인딩 API를 가로채는 데 사용됩니다.

- Microsoft.Extensions.Configuration.ConfigurationBinder
- Microsoft.Extensions.DependencyInjection.OptionsBuilderConfigurationExtensions
- Microsoft.Extensions.DependencyInjection.OptionsConfigurationServiceCollectionExtensions

즉, 결국 이러한 다양한 바인딩 메서드를 호출하는 모든 API는 가로채 생성된 코드로 대체됩니다.

예제 사용

네이티브 AOT 앱으로 게시하도록 구성된 .NET 콘솔 애플리케이션을 고려합니다. 다음 코드에서는 구성 원본 생성기를 사용하여 구성 설정을 바인딩하는 방법을 보여 줍니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <RootNamespace>console_binder_gen</RootNamespace>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <PublishAot>true</PublishAot>
    <InvariantGlobalization>true</InvariantGlobalization>

<EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerat
or>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration"
Version="9.0.3" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder"
Version="9.0.3" />
  </ItemGroup>

</Project>
```

위의 프로젝트 파일은 속성을 `EnableConfigurationBindingGenerator`로 설정하여 구성 원본 생성기를 사용하도록 설정합니다`true`.

다음으로, Program.cs *파일을 고려합니다.*

C#

```
using Microsoft.Extensions.Configuration;

var builder = new ConfigurationBuilder()
    .AddInMemoryCollection(initialData: [
            new("port", "5001"),
            new("enabled", "true"),
            new("apiUrl", "https://jsonplaceholder.typicode.com/")
        ]);

var configuration = builder.Build();

var settings = new Settings();
configuration.Bind(settings);

// Write the values to the console.
Console.WriteLine($"Port = {settings.Port}");
Console.WriteLine($"Enabled = {settings.Enabled}");
Console.WriteLine($"API URL = {settings.ApiUrl}");

class Settings
{
```

```
    public int Port { get; set; }
    public bool Enabled { get; set; }
    public string? ApiUrl { get; set; }
}

// This will output the following:
//   Port = 5001
//   Enabled = True
//   API URL = https://jsonplaceholder.typicode.com/
```

앞의 코드가 하는 역할은 다음과 같습니다.

- 구성 작성기 인스턴스를 인스턴스화합니다.
- 세 가지 구성 원본 값을 호출 AddInMemoryCollection 하고 정의합니다.
- 구성을 빌드하기 위한 호출 Build() 입니다.
- 메서드를 ConfigurationBinder.Bind 사용하여 구성 값에 `Settings` 개체를 바인딩합니다.

애플리케이션이 빌드되면 구성 원본 생성기가 호출을 `Bind` 가로채 바인딩 코드를 생성합니다.

> **ⓘ 중요**
>
> `PublishAot` 속성이 `true`로 설정되거나 다른 AOT 경고가 사용되었고, `EnabledConfigurationBindingGenerator` 속성이 `false`로 설정되었을 때, 경고 `IL2026`가 발생합니다. 이 경고는 멤버가 RequiresUnreferencedCode 특징을 가지고 있을 때, 트리밍 시 문제가 발생할 수 있음을 나타냅니다. 자세한 내용은 IL2026을 참조하세요.

소스 생성 코드 탐색

다음 코드는 이전 예제의 구성 원본 생성기에서 생성됩니다.

C#

```
// <auto-generated/>

#nullable enable annotations
#nullable disable warnings

// Suppress warnings about [Obsolete] member usage in generated code.
#pragma warning disable CS0612, CS0618

namespace System.Runtime.CompilerServices
{
    using System;
```

```
    using System.CodeDom.Compiler;


[GeneratedCode("Microsoft.Extensions.Configuration.Binder.SourceGeneration",
"9.0.10.47305")]
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
    file sealed class InterceptsLocationAttribute : Attribute
    {
        public InterceptsLocationAttribute(int version, string data)
        {
        }
    }
}

namespace Microsoft.Extensions.Configuration.Binder.SourceGeneration
{
    using Microsoft.Extensions.Configuration;
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Globalization;
    using System.Runtime.CompilerServices;


[GeneratedCode("Microsoft.Extensions.Configuration.Binder.SourceGeneration",
"9.0.10.47305")]
    file static class BindingExtensions
    {
        #region IConfiguration extensions.
        /// <summary>Attempts to bind the given object instance to
configuration values by matching property names against configuration keys
recursively.</summary>
        [InterceptsLocation(1, "uDIs2gDFz/yEvxOzjNK4jnIBAABQcm9ncmFtLmNz")]
// D:\source\WorkerService1\WorkerService1\Program.cs(13,15)
        public static void Bind_Settings(this IConfiguration configuration,
object? instance)
        {
            ArgumentNullException.ThrowIfNull(configuration);

            if (instance is null)
            {
                return;
            }

            var typedObj = (global::Settings)instance;
            BindCore(configuration, ref typedObj, defaultValueIfNotFound:
false, binderOptions: null);
        }
        #endregion IConfiguration extensions.

        #region Core binding extensions.
        private readonly static Lazy<HashSet<string>> s_configKeys_Settings
= new(() => new HashSet<string>(StringComparer.OrdinalIgnoreCase) { "Port",
"Enabled", "ApiUrl" });
```

```
        public static void BindCore(IConfiguration configuration, ref
global::Settings instance, bool defaultValueIfNotFound, BinderOptions?
binderOptions)
        {
            ValidateConfigurationKeys(typeof(global::Settings),
s_configKeys_Settings, configuration, binderOptions);

            if (configuration["Port"] is string value0 &&
!string.IsNullOrEmpty(value0))
            {
                instance.Port = ParseInt(value0,
configuration.GetSection("Port").Path);
            }
            else if (defaultValueIfNotFound)
            {
                instance.Port = instance.Port;
            }

            if (configuration["Enabled"] is string value1 &&
!string.IsNullOrEmpty(value1))
            {
                instance.Enabled = ParseBool(value1,
configuration.GetSection("Enabled").Path);
            }
            else if (defaultValueIfNotFound)
            {
                instance.Enabled = instance.Enabled;
            }

            if (configuration["ApiUrl"] is string value2)
            {
                instance.ApiUrl = value2;
            }
            else if (defaultValueIfNotFound)
            {
                var currentValue = instance.ApiUrl;
                if (currentValue is not null)
                {
                    instance.ApiUrl = currentValue;
                }
            }
        }


        /// <summary>If required by the binder options, validates that there
are no unknown keys in the input configuration object.</summary>
        public static void ValidateConfigurationKeys(Type type,
Lazy<HashSet<string>> keys, IConfiguration configuration, BinderOptions?
binderOptions)
        {
            if (binderOptions?.ErrorOnUnknownConfiguration is true)
            {
                List<string>? temp = null;

                foreach (IConfigurationSection section in
```

```
configuration.GetChildren())
                {
                    if (!keys.Value.Contains(section.Key))
                    {
                        (temp ??= new List<string>
()).Add($"'{section.Key}'");
                    }
                }

                if (temp is not null)
                {
                    throw new
InvalidOperationException($"'ErrorOnUnknownConfiguration' was set on the
provided BinderOptions, but the following properties were not found on the
instance of {type}: {string.Join(", ", temp)}");
                }
            }
        }

        public static int ParseInt(string value, string? path)
        {
            try
            {
                return int.Parse(value, NumberStyles.Integer,
CultureInfo.InvariantCulture);
            }
            catch (Exception exception)
            {
                throw new InvalidOperationException($"Failed to convert
configuration value at '{path}' to type '{typeof(int)}'.", exception);
            }
        }

        public static bool ParseBool(string value, string? path)
        {
            try
            {
                return bool.Parse(value);
            }
            catch (Exception exception)
            {
                throw new InvalidOperationException($"Failed to convert
configuration value at '{path}' to type '{typeof(bool)}'.", exception);
            }
        }
        #endregion Core binding extensions.
    }
}
```

> ⓘ **참고**
>
> 이 생성된 코드는 구성 원본 생성기의 버전에 따라 변경될 수 있습니다.

생성된 코드는 `BindingExtensions` 클래스를 포함하며, 이 클래스에는 실제 바인딩을 수행하는 `BindCore` 메서드가 포함되어 있습니다. 메서드는 `Bind_Settings` 메서드를 `BindCore` 호출하고 인스턴스를 지정된 형식으로 캐스팅합니다.

생성된 코드를 보려면 프로젝트 파일에서 설정합니다 `<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>` . 이렇게 하면 개발자가 검사를 위해 파일을 볼 수 있습니다. Visual Studio의 솔루션 탐색기 프로젝트의 종속성 >

참조

- .NET의 구성
- Roslyn: 인터셉터 기능
- .NET의 옵션 패턴

.NET에서 사용자 지정 구성 공급자 구현

JSON, XML 및 INI 파일과 같은 일반적인 구성 원본에 사용할 수 있는 많은 구성 공급자 가 있습니다. 사용 가능한 공급자 중 하나가 애플리케이션 요구 사항에 맞지 않는 경우 사용자 지정 구성 공급자를 구현해야 할 수 있습니다. 이 문서에서는 데이터베이스를 구성 원본으로 사용하는 사용자 지정 구성 공급자를 구현하는 방법을 알아봅니다.

사용자 지정 구성 공급자

샘플 앱은 EF(Entity Framework) Core를 사용하여 데이터베이스에서 구성 키-값 쌍을 읽는 기본 구성 공급자를 만드는 방법을 보여 줍니다.

이 공급자의 특징은 다음과 같습니다.

- 데모용으로 EF 메모리 내 데이터베이스를 사용합니다.
  - 연결 문자열이 필요한 데이터베이스를 사용하려면 중간 구성에서 연결 문자열을 가져옵니다.
- 이 공급자는 시작 시 데이터베이스 테이블을 구성으로 읽어 들입니다. 이 공급자는 키별 기준으로 데이터베이스를 쿼리하지 않습니다.
- 변경 내용 다시 로드는 구현되지 않으므로 앱이 시작된 후 데이터베이스를 업데이트해도 앱의 구성에는 영향을 주지 않습니다.

데이터베이스에 `Settings` 구성 값을 저장하기 위한 레코드 형식 엔터티를 정의합니다. 예를 들어 *Models* 폴더에 *Settings.cs* 파일을 추가할 수 있습니다.

C#

```
namespace CustomProvider.Example.Models;

public record Settings(string Id, string? Value);
```

레코드 형식에 대한 자세한 내용은 C#의 레코드 형식을 참조하세요.

구성된 값을 저장 및 액세스하는 `EntityConfigurationContext`를 추가합니다.

공급자/EntityConfigurationContext.cs:

C#

```
using CustomProvider.Example.Models;
using Microsoft.EntityFrameworkCore;

namespace CustomProvider.Example.Providers;

public sealed class EntityConfigurationContext(string? connectionString) : DbContext
```

```
{
    public DbSet<Settings> Settings => Set<Settings>();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        _ = connectionString switch
        {
            { Length: > 0 } => optionsBuilder.UseSqlServer(connectionString),
            _ => optionsBuilder.UseInMemoryDatabase("InMemoryDatabase")
        };
    }
}
```

OnConfiguring(DbContextOptionsBuilder)를 재정의하면 적절한 데이터베이스 연결을 사용할 수 있습니다. 예를 들어 연결 문자열이 제공된 경우 SQL Server에 연결할 수 있습니다. 그렇지 않으면 메모리 내 데이터베이스를 사용할 수 있습니다.

IConfigurationSource를 구현하는 클래스를 만듭니다.

공급자/EntityConfigurationSource.cs:

C#

```
using Microsoft.Extensions.Configuration;

namespace CustomProvider.Example.Providers;

public sealed class EntityConfigurationSource(
    string? connectionString) : IConfigurationSource
{
    public IConfigurationProvider Build(IConfigurationBuilder builder) =>
        new EntityConfigurationProvider(connectionString);
}
```

ConfigurationProvider에서 상속하여 사용자 지정 구성 공급자를 만듭니다. 구성 공급자는 비어 있는 데이터베이스를 초기화합니다. 구성 키는 대/소문자를 구분하지 않으므로 데이터베이스를 초기화하는 데 사용되는 사전은 대/소문자를 구분하지 않는 비교자 (StringComparer.OrdinalIgnoreCase)를 사용하여 생성됩니다.

공급자/EntityConfigurationProvider.cs :

C#

```
using CustomProvider.Example.Models;
using Microsoft.Extensions.Configuration;

namespace CustomProvider.Example.Providers;

public sealed class EntityConfigurationProvider(
    string? connectionString)
```

```
    : ConfigurationProvider
{
    public override void Load()
    {
        using var dbContext = new EntityConfigurationContext(connectionString);

        dbContext.Database.EnsureCreated();

        Data = dbContext.Settings.Any()
            ? dbContext.Settings.ToDictionary(
                static c => c.Id,
                static c => c.Value, StringComparer.OrdinalIgnoreCase)
            : CreateAndSaveDefaultValues(dbContext);
    }

    static Dictionary<string, string?> CreateAndSaveDefaultValues(
        EntityConfigurationContext context)
    {
        var settings = new Dictionary<string, string?>(
            StringComparer.OrdinalIgnoreCase)
        {
            ["WidgetOptions:EndpointId"] = "b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67",
            ["WidgetOptions:DisplayLabel"] = "Widgets Incorporated, LLC.",
            ["WidgetOptions:WidgetRoute"] = "api/widgets"
        };

        context.Settings.AddRange(
            [.. settings.Select(static kvp => new Settings(kvp.Key, kvp.Value))]);

        context.SaveChanges();

        return settings;
    }
}
```

`AddEntityConfiguration` 확장 메서드를 사용하면 기본 인스턴스에 구성 원본을 추가할 수 있습니다`ConfigurationManager`.

Extensions/ConfigurationManagerExtensions.cs:

C#

```
using CustomProvider.Example.Providers;

namespace Microsoft.Extensions.Configuration;

public static class ConfigurationManagerExtensions
{
    public static ConfigurationManager AddEntityConfiguration(
        this ConfigurationManager manager)
    {
        var connectionString =
manager.GetConnectionString("WidgetConnectionString");
```

```
        IConfigurationBuilder configBuilder = manager;
        configBuilder.Add(new EntityConfigurationSource(connectionString));

        return manager;
    }
}
```

ConfigurationManager은/는 IConfigurationBuilder과 IConfigurationRoot의 구현이므로, 확장 메서드는 연결 문자열 구성에 액세스하고 `EntityConfigurationSource`를 추가할 수 있습니다.

다음 코드는 `EntityConfigurationProvider` 사용자 지정 을 사용하는 방법을 보여줍니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using CustomProvider.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddEntityConfiguration();

builder.Services.Configure<WidgetOptions>(
    builder.Configuration.GetSection("WidgetOptions"));

using IHost host = builder.Build();

WidgetOptions options = host.Services.GetRequiredService<IOptions<WidgetOptions>>
().Value;
Console.WriteLine($"DisplayLabel={options.DisplayLabel}");
Console.WriteLine($"EndpointId={options.EndpointId}");
Console.WriteLine($"WidgetRoute={options.WidgetRoute}");

await host.RunAsync();
// Sample output:
//    WidgetRoute=api/widgets
//    EndpointId=b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67
//    DisplayLabel=Widgets Incorporated, LLC.
```

공급자 서비스를 사용하기

사용자 지정 구성 공급자를 사용하려면 옵션 패턴을 사용할 수 있습니다. 샘플 앱을 사용하여 위젯 설정을 나타내는 옵션 개체를 정의합니다.

C#

```
namespace CustomProvider.Example;

public class WidgetOptions
{
    public required Guid EndpointId { get; set; }

    public required string DisplayLabel { get; set; } = null!;

    public required string WidgetRoute { get; set; } = null!;
}
```

Configure 호출은 구성 인스턴스를 등록하며, `TOptions`는 이에 바인딩됩니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using CustomProvider.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddEntityConfiguration();

builder.Services.Configure<WidgetOptions>(
    builder.Configuration.GetSection("WidgetOptions"));

using IHost host = builder.Build();

WidgetOptions options = host.Services.GetRequiredService<IOptions<WidgetOptions>>
().Value;
Console.WriteLine($"DisplayLabel={options.DisplayLabel}");
Console.WriteLine($"EndpointId={options.EndpointId}");
Console.WriteLine($"WidgetRoute={options.WidgetRoute}");

await host.RunAsync();
// Sample output:
//    WidgetRoute=api/widgets
//    EndpointId=b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67
//    DisplayLabel=Widgets Incorporated, LLC.
```

이전 코드는 구성의 `"WidgetOptions"` 섹션에서 `WidgetOptions` 객체를 설정합니다. 이렇게 하면 옵션 패턴이 활성화되고 EF 설정의 종속성 주입 준비 `IOptions<WidgetOptions>` 표현이 노출됩니다. 옵션은 궁극적으로 사용자 지정 구성 공급자에서 제공됩니다.

참고하십시오

- .NET의 구성

- .NET의 구성 공급자
- .NET의 옵션 패턴
- .NET에서 종속성 주입

Last updated on 2026. 01. 24.

.NET의 옵션 패턴

옵션 패턴은 클래스를 사용하여 관련 설정 그룹에 대한 강력한 형식의 액세스를 제공합니다. 구성 설정이 시나리오에 따라 별도 클래스로 격리된 경우 앱은 두 가지 중요한 소프트웨어 엔지니어링 원칙을 따릅니다.

- ISP(Interface Segregation Principle, 인터페이스 분리 원칙) 또는 캡슐화: 구성 설정에 종속된 시나리오(클래스)는 사용하는 구성 설정에 의해서만 결정됩니다.
- 관심사의 분리 앱의 다른 부분에 대한 설정은 다른 설정에 종속되거나 연결되지 않습니다.

옵션은 구성 데이터의 유효성을 검사하는 메커니즘도 제공합니다. 자세한 내용은 옵션 유효성 검사 섹션을 참조하세요.

계층적 구성 바인딩

관련 구성 값을 읽는 기본 방법은 옵션 패턴를 사용하는 것입니다. 옵션 패턴은 제네릭 형식 매개 변수 IOptions<TOptions>가 `TOptions`로 제한되는 `class` 인터페이스를 통해 사용할 수 있습니다. `IOptions<TOptions>`는 나중에 종속성 주입을 통해 제공할 수 있습니다. 자세한 내용은 .NET에서 종속성 주입을 참조하세요.

예를 들어, *appsettings.json* 파일에서 강조 표시된 구성 값을 읽으려면 다음을 수행합니다.

JSON

```
{
    "SecretKey": "Secret key value",
    "TransientFaultHandlingOptions": {
        "Enabled": true,
        "AutoRetryDelay": "00:00:07"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

다음 `TransientFaultHandlingOptions` 클래스를 만듭니다.

C#

```
public sealed class TransientFaultHandlingOptions
{
    public bool Enabled { get; set; }
```

```
    public TimeSpan AutoRetryDelay { get; set; }
}
```

옵션 패턴을 사용하는 경우 옵션 클래스는

- 매개 변수가 없는 공용 생성자를 사용하는 비추상이어야 합니다.
- 바인딩할 공용 읽기/쓰기 속성을 포함합니다(필드는 바인딩되지 ***않음***).

다음 코드는 *Program.cs* C# 파일의 일부이며 다음과 같습니다.

- ConfigurationBinder.Bind를 호출하여 `TransientFaultHandlingOptions` 클래스를 `"TransientFaultHandlingOptions"` 섹션에 바인딩합니다.
- 구성 데이터를 표시합니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using ConsoleJson.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.Sources.Clear();

IHostEnvironment env = builder.Environment;

builder.Configuration
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true);

TransientFaultHandlingOptions options = new();
builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
    .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();

// <Output>
// Sample output:
```

이전 코드에서 JSON 구성 파일에는 `"TransientFaultHandlingOptions"` 인스턴스에 바인딩된 `TransientFaultHandlingOptions` 섹션이 있습니다. 이렇게 하면 C# 개체 속성이 구성의 해당 값으로 수화됩니다.

ConfigurationBinder.Get<T>는 지정된 형식을 바인딩하고 반환합니다. `ConfigurationBinder.Get<T>`가 `ConfigurationBinder.Bind`를 사용하는 것보다 편리할 수 있습니다. 다음 코드에서는 `ConfigurationBinder.Get<T>`와 `TransientFaultHandlingOptions` 클래스를 함께 사용하는 방법을 보여 줍니다.

C#

```
var options =
    builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
        .Get<TransientFaultHandlingOptions>();

Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");
```

앞의 코드에서 `ConfigurationBinder.Get<T>`는 기본 구성에서 채워진 속성 값을 사용하여 `TransientFaultHandlingOptions` 개체의 인스턴스를 획득하는 데 사용됩니다.

> ⓘ **Important**
>
> ConfigurationBinder 클래스는 `.Bind(object instance)` 및 `.Get<T>()`와 같이 로 제한되지 여러 API를 노출합니다. **옵션 인터페이스**를 사용하는 경우 앞서 언급한 **옵션 클래스 제약 조건**을 준수해야 합니다.

옵션 패턴을 사용하는 경우 한 가지 대체 방법은 `"TransientFaultHandlingOptions"` 섹션을 바인딩하고 종속성 주입 서비스 컨테이너에 추가하는 것입니다. 다음 코드에서 `TransientFaultHandlingOptions`는 Configure를 통해 서비스 컨테이너에 추가되고 구성에 바인딩됩니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.Configure<TransientFaultHandlingOptions>(
    builder.Configuration.GetSection(
        key: nameof(TransientFaultHandlingOptions)));
```

앞의 예에서 `builder`는 HostApplicationBuilder의 인스턴스입니다.

> 팁
>
> `key` 매개 변수는 검색할 구성 섹션의 이름입니다. 이를 표현하는 형식의 이름과 일치할 필요가 '없습니다'. 예를 들어 `"FaultHandling"`라는 섹션이 `TransientFaultHandlingOptions` 클

래스에 의해 표현될 수 있습니다. 이 인스턴스에서는 `"FaultHandling"`을 GetSection 함수에 대신 전달합니다. `nameof` 연산자는 명명된 섹션이 해당하는 형식과 일치하는 경우 편의를 위해 사용됩니다.

위의 코드를 사용하여 다음 코드는 위치 옵션을 읽습니다.

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

public sealed class ExampleService(IOptions<TransientFaultHandlingOptions> options)
{
    private readonly TransientFaultHandlingOptions _options = options.Value;

    public void DisplayValues()
    {
        Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{_options.Enabled}");
        Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{_options.AutoRetryDelay}");
    }
}
```

위 코드에서는 앱을 시작한 후의 JSON 구성 파일의 변경 사항을 읽지 **않습니다**. 앱이 시작된 후 변경 내용을 읽으려면 IOptionsSnapshot 또는 IOptionsMonitor를 사용하여 변경 내용이 발생할 때마다 모니터링하고 그에 따라 대응합니다.

옵션 인터페이스

:

- 다음을 지원하지 **않습니다**.
  - 앱이 시작된 후 구성 데이터 읽기
  - 명명된 옵션
- Singleton으로 등록되며 서비스 수명에 주입할 수 있습니다.

:

- 범위가 지정된 수명 또는 임시 수명에서 모든 주입 확인마다 옵션을 다시 계산하는 시나리오에서 유용합니다. 자세한 내용은 IOptionsSnapshot을 사용하여 업데이트된 데이터를 참조하세요.
- 범위 지정됨으로 등록되므로 Singleton 서비스에 주입할 수 없습니다.
- 명명된 옵션 지원합니다.

:

- 옵션을 검색하고 `TOptions` 인스턴스에 대한 옵션 알림을 관리하는 데 사용됩니다.
- Singleton으로 등록되며 서비스 수명에 주입할 수 있습니다.
- 지원:
  - 알림 변경
  - 명명된 옵션
  - 다시 로드할 수 있는 구성
  - 선택적 옵션 무효화(IOptionsMonitorCache<TOptions>)

IOptionsFactory<TOptions>는 새로운 옵션 인스턴스를 만듭니다. 단일 Create 메서드가 있습니다. 기본 구현에서는 등록된 모든 IConfigureOptions<TOptions> 및 IPostConfigureOptions<TOptions>를 사용하며 먼저 구성을 모두 실행한 다음, 사후 구성을 수행합니다. IConfigureNamedOptions<TOptions>와 IConfigureOptions<TOptions>를 구별하며 적절한 인터페이스만 호출합니다.

IOptionsMonitorCache<TOptions>는 IOptionsMonitor<TOptions>에서 `TOptions` 인스턴스를 캐시하는 데 사용됩니다. IOptionsMonitorCache<TOptions>는 모니터의 옵션 인스턴스를 무효화하므로 값이 다시 계산됩니다(TryRemove). TryAdd를 사용하여 값을 수동으로 도입할 수 있습니다. 모든 명명된 인스턴스를 필요에 따라 다시 만들어야 하는 경우 Clear 메서드가 사용됩니다.

IOptionsChangeTokenSource<TOptions>는 기본 IChangeToken 인스턴스에 대한 변경 내용을 추적하는 `TOptions`을 가져오는 데 사용됩니다. 변경 토큰 기본 요소에 대한 자세한 내용은 변경 알림을 참조하세요.

옵션 인터페이스의 이점

제네릭 래퍼 형식을 사용하면 DI(종속성 주입) 컨테이너에서 옵션의 수명을 분리할 수 있습니다. IOptions<TOptions>.Value 인터페이스는 일반 제약 조건을 포함하여 옵션 형식에 대한 추상화 계층을 제공합니다. 이 기능은 다음과 같은 이점을 제공합니다.

- `T` 구성 인스턴스의 평가는 IOptions<TOptions>.Value가 주입될 때가 아니라 액세스될 때로 지연됩니다. 이는 `T` 옵션을 다양한 위치에서 사용하고 `T`에 대한 변경 없이 수명 의미 체계를 선택할 수 있기 때문에 중요합니다.
- 형식 `T`의 옵션을 등록할 때 `T` 유형을 명시적으로 등록할 필요가 없습니다. 이는 간단한 기본값을 사용하여 라이브러리를 작성하고 호출자가 특정 수명을 갖는 옵션을 DI 컨테이너에 등록하도록 강제하지 않으려는 경우에 편리합니다.
- API의 관점에서 보면 `T` 형식에 제약 조건을 사용할 수 있습니다(이 경우 `T`가 참조 형식으로 제한됨).

IOptionsSnapshot을 사용하여 업데이트된 데이터 읽기

IOptionsSnapshot<TOptions>을 사용하면, 옵션은 액세스될 때 요청당 한 번씩 계산되고 요청의 수명 동안 캐시됩니다. 업데이트된 구성 값 읽기를 지원하는 구성 공급자를 사용하는 경우 앱을 시작한 후 구성 변경 내용을 읽습니다.

`IOptionsMonitor`와 `IOptionsSnapshot`의 차이점은 다음과 같습니다.

- `IOptionsMonitor`는 언제든지 현재 옵션 값을 검색하는 싱글톤 서비스로, 싱글톤 종속성에서 특히 유용합니다.
- `IOptionsSnapshot`은 범위가 지정된 서비스이며 `IOptionsSnapshot<T>` 개체가 생성될 때 옵션의 스냅샷을 제공합니다. 옵션 스냅숏은 임시 및 범위가 지정된 종속성과 함께 사용하도록 설계되었습니다.

다음 코드에서는 IOptionsSnapshot<TOptions> 메서드를 사용합니다.

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

public sealed class ScopedService(IOptionsSnapshot<TransientFaultHandlingOptions>
options)
{
    private readonly TransientFaultHandlingOptions _options = options.Value;

    public void DisplayValues()
    {
        Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{_options.Enabled}");
        Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{_options.AutoRetryDelay}");
    }
}
```

다음 코드에서는 `TransientFaultHandlingOptions`가 바인딩되는 구성 인스턴스를 등록합니다.

C#

```
builder.Services
    .Configure<TransientFaultHandlingOptions>(
        configurationRoot.GetSection(
            nameof(TransientFaultHandlingOptions)));
```

앞의 코드에서 `Configure<TOptions>` 메서드는 `TOptions`가 바인딩할 구성 인스턴스를 등록하고 구성이 변경될 때 옵션을 업데이트하는 데 사용됩니다.

IOptionsMonitor

`IOptionsMonitor` 유형은 변경 알림을 지원하며 앱이 구성 소스 변경에 동적으로 응답해야 할 수 있는 시나리오를 활성화합니다. 이 기능은 앱이 시작된 후 구성 데이터의 변경 내용에 대응해야 하는 경우에 유용합니다. 변경 알림은 다음과 같은 파일 시스템 기반 구성 공급자에 대해서만 지원됩니다.

- Microsoft.Extensions.Configuration.Ini
- Microsoft.Extensions.Configuration.Json
- Microsoft.Extensions.Configuration.KeyPerFile
- Microsoft.Extensions.Configuration.UserSecrets
- Microsoft.Extensions.Configuration.Xml

옵션 모니터를 사용하려면 구성 섹션에서 동일한 방식으로 옵션 개체를 구성해야 합니다.

C#

```
builder.Services
    .Configure<TransientFaultHandlingOptions>(
        configurationRoot.GetSection(
            nameof(TransientFaultHandlingOptions)));
```

다음 예제에서는 IOptionsMonitor<TOptions>를 사용합니다.

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

public sealed class MonitorService(IOptionsMonitor<TransientFaultHandlingOptions>
monitor)
{
    public void DisplayValues()
    {
        TransientFaultHandlingOptions options = monitor.CurrentValue;

        Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{options.Enabled}");
        Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");
    }
}
```

위 코드에서는 앱을 시작한 후 JSON 구성 파일 변경 사항을 읽습니다.

> **팁**
>
> Docker 컨테이너나 네트워크 공유 같은 일부 파일 시스템은 변경 알림을 안정적으로 전송할 수 없습니다. 이러한 환경에서 IOptionsMonitor<TOptions> 인터페이스를 사용하는 경우 `DOTNET_USE_POLLING_FILE_WATCHER` 환경 변수를 `1` 또는 `true`로 설정하여 파일 시스템에서 변경 내용을 폴링합니다. 변경 내용이 폴링되는 간격은 4초마다이며 구성할 수 없습니다.
>
> Docker 컨테이너에 대한 자세한 내용은 **.NET 앱 컨테이너화**를 참조하세요.

명명된 옵션은 IConfigureNamedOptions 사용을 지원

명명된 옵션:

- 여러 구성 섹션이 동일한 속성에 바인딩되는 경우에 유용합니다.
- 대/소문자를 구분합니다.

다음 *appsettings.json* 파일을 고려하세요.

JSON

```
{
  "Features": {
    "Personalize": {
      "Enabled": true,
      "ApiKey": "aGEgaGEgeW91IHRob3VnaHQgdGhhdCB3YXMgcmVhbGx5IHNvbWV0aGluZw=="
    },
    "WeatherStation": {
      "Enabled": true,
      "ApiKey": "QXJlIHlvdSBhdHRlbXB0aW5nIHRvIGhhY2sgdXM/"
    }
  }
}
```

`Features:Personalize` 및 `Features:WeatherStation`를 바인딩하는 두 개의 클래스를 만드는 대신 각 섹션에 다음 클래스가 사용됩니다.

C#

```
public class Features
{
    public const string Personalize = nameof(Personalize);
```

```
    public const string WeatherStation = nameof(WeatherStation);

    public bool Enabled { get; set; }
    public string ApiKey { get; set; }
}
```

다음 코드는 명명된 옵션을 구성합니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

// Omitted for brevity...

builder.Services.Configure<Features>(
    Features.Personalize,
    builder.Configuration.GetSection("Features:Personalize"));

builder.Services.Configure<Features>(
    Features.WeatherStation,
    builder.Configuration.GetSection("Features:WeatherStation"));
```

다음 코드는 명명된 옵션을 표시합니다.

C#

```
public sealed class Service
{
    private readonly Features _personalizeFeature;
    private readonly Features _weatherStationFeature;

    public Service(IOptionsSnapshot<Features> namedOptionsAccessor)
    {
        _personalizeFeature = namedOptionsAccessor.Get(Features.Personalize);
        _weatherStationFeature = namedOptionsAccessor.Get(Features.WeatherStation);
    }
}
```

모든 옵션은 명명된 인스턴스입니다. IConfigureOptions<TOptions> 인스턴스는 `Options.DefaultName`인 `string.Empty` 인스턴스를 대상으로 지정한 것처럼 처리됩니다. 또한 IConfigureNamedOptions<TOptions>는 IConfigureOptions<TOptions>를 구현합니다. IOptionsFactory<TOptions>의 기본 구현에는 각 옵션을 적절하게 사용하기 위한 논리가 있습니다. `null` 명명된 옵션은 특정 명명된 인스턴스 대신 모든 명명된 인스턴스를 대상으로 지정하는 데 사용됩니다. ConfigureAll 및 PostConfigureAll에서 이 규칙을 사용합니다.

OptionsBuilder API

OptionsBuilder<TOptions>는 `TOptions` 인스턴스를 구성하는 데 사용됩니다. `OptionsBuilder`는 `AddOptions<TOptions>(string optionsName)` 호출에 대한 단일 매개 변수이므로 모든 후속 호출에 나타나는 대신 명명된 옵션 생성을 간소화합니다. 옵션 유효성 검사 및 서비스 종속성을 허용하는 `ConfigureOptions` 오버로드는 `OptionsBuilder`를 통해서만 사용할 수 있습니다.

`OptionsBuilder`는 옵션 유효성 검사 섹션에서 사용됩니다.

DI 서비스를 사용하여 옵션 구성

옵션을 구성할 때 종속성 주입을 사용하여 등록된 서비스에 액세스하고 이를 사용하여 옵션을 구성할 수 있습니다. 이는 옵션을 구성하기 위해 서비스에 액세스해야 할 때 유용합니다. 다음 두 가지 방법으로 옵션을 구성하는 동안 DI에서 서비스에 액세스할 수 있습니다.

- 구성 대리자를 의 <에 전달합니다. `OptionsBuilder<TOptions>`는 최대 5개의 서비스를 사용하여 옵션을 구성할 수 있는 Configure의 오버로드를 제공합니다.

  C#

  ```
  builder.Services
      .AddOptions<MyOptions>("optionalName")
      .Configure<ExampleService, ScopedService, MonitorService>(
          (options, es, ss, ms) =>
              options.Property = DoSomethingWith(es, ss, ms));
  ```

- IConfigureOptions<TOptions> 또는 IConfigureNamedOptions<TOptions>를 구현하는 형식을 만들고 해당 형식을 서비스로 등록합니다.

서비스 만들기가 더 복잡하므로 구성 위임을 Configure에 전달하는 것이 좋습니다. 형식을 만드는 작업은 Configure를 호출할 때 프레임워크에서 수행하는 작업에 해당합니다. Configure을 호출하면 지정된 일반 서비스 유형을 허용하는 생성자가 있는 일시적인 일반 IConfigureNamedOptions<TOptions>가 등록됩니다.

옵션 유효성 검사

옵션 유효성 검사를 통해 옵션 값의 유효성을 검사할 수 있습니다.

다음 *appsettings.json* 파일을 고려하세요.

JSON

```
{
  "MyCustomSettingsSection": {
    "SiteTitle": "Amazing docs from Awesome people!",
    "Scale": 10,
```

```
    "VerbosityLevel": 32
  }
}
```

다음 클래스는 `"MyCustomSettingsSection"` 구성 섹션에 바인딩되고 몇 가지 `DataAnnotations` 규칙을 적용합니다.

C#

```
using System.ComponentModel.DataAnnotations;

namespace ConsoleJson.Example;

public sealed class SettingsOptions
{
    public const string ConfigurationSectionName = "MyCustomSettingsSection";

    [Required]
    [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]
    public required string SiteTitle { get; set; }

    [Required]
    [Range(0, 1_000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public required int Scale { get; set; }

    [Required]
    public required int VerbosityLevel { get; set; }
}
```

위의 `SettingsOptions` 클래스에서 `ConfigurationSectionName` 속성에는 바인딩할 구성 섹션의 이름이 포함됩니다. 이 시나리오에서 옵션 개체는 구성 섹션의 이름을 제공합니다.

> **팁**
>
> 구성 섹션 이름은 바인딩되는 구성 개체와 관련이 없습니다. 즉, `"FooBarOptions"`라는 구성 섹션은 `ZedOptions`라는 옵션 개체에 바인딩할 수 있습니다. 일반적으로 이름을 동일하게 지정할 수 있지만 반드시 그럴 필요가 '없으며' 실제로 이름 충돌이 발생할 수 있습니다.

코드는 다음과 같습니다.

- AddOptions를 호출하여 < 클래스에 바인딩되는 >를 가져옵니다.
- ValidateDataAnnotations를 호출하여 `DataAnnotations`를 사용한 유효성 검사를 사용하도록 설정합니다.

C#

```
builder.Services
    .AddOptions<SettingsOptions>()
    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations();
```

`ValidateDataAnnotations` 확장 메서드는 Microsoft.Extensions.Options.DataAnnotations NuGet 패키지에서 정의됩니다.

다음 코드는 구성 값을 표시하거나 유효성 검사 오류를 보고합니다.

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

public sealed class ValidationService
{
    private readonly ILogger<ValidationService> _logger;
    private readonly IOptions<SettingsOptions> _config;

    public ValidationService(
        ILogger<ValidationService> logger,
        IOptions<SettingsOptions> config)
    {
        _config = config;
        _logger = logger;

        try
        {
            SettingsOptions options = _config.Value;
        }
        catch (OptionsValidationException ex)
        {
            foreach (string failure in ex.Failures)
            {
                _logger.LogError("Validation error: {FailureMessage}", failure);
            }
        }
    }
}
```

다음 코드는 대리자를 사용하여 보다 복잡한 유효성 검사 규칙을 적용합니다.

C#

```
builder.Services
    .AddOptions<SettingsOptions>()
    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations()
```

```
    .Validate(config =>
    {
        if (config.Scale != 0)
        {
            return config.VerbosityLevel > config.Scale;
        }

        return true;
    }, "VerbosityLevel must be > than Scale.");
```

유효성 검사는 런타임에 발생하지만, 시작 시 `ValidateOnStart`에 대한 호출을 연결하여 대신 수행하도록 구성할 수 있습니다.

C#

```
builder.Services
    .AddOptions<SettingsOptions>()
    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations()
    .Validate(config =>
    {
        if (config.Scale != 0)
        {
            return config.VerbosityLevel > config.Scale;
        }

        return true;
    }, "VerbosityLevel must be > than Scale.")
    .ValidateOnStart();
```

특정 옵션 유형에 대해 시작 시 유효성 검사를 사용하도록 설정하려면 API를 AddOptionsWithValidateOnStart<TOptions>(IServiceCollection, String) 사용합니다.

C#

```
builder.Services
    .AddOptionsWithValidateOnStart<SettingsOptions>()
    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations()
    .Validate(config =>
    {
        if (config.Scale != 0)
        {
            return config.VerbosityLevel > config.Scale;
        }

        return true;
    }, "VerbosityLevel must be > than Scale.");
```

복잡한 유효성 검사의 경우 `IValidateOptions`

다음 클래스는 IValidateOptions<TOptions>를 구현합니다.

C#

```
using System.Text;
using System.Text.RegularExpressions;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

sealed partial class ValidateSettingsOptions(
    IConfiguration config)
    : IValidateOptions<SettingsOptions>
{
    public SettingsOptions? Settings { get; private set; } =
        config.GetSection(SettingsOptions.ConfigurationSectionName)
              .Get<SettingsOptions>();

    public ValidateOptionsResult Validate(string? name, SettingsOptions options)
    {
        StringBuilder? failure = null;

        if (!ValidationRegex().IsMatch(options.SiteTitle))
        {
            (failure ??= new()).AppendLine($"{options.SiteTitle} doesn't match
RegEx");
        }

        if (options.Scale is < 0 or > 1_000)
        {
            (failure ??= new()).AppendLine($"{options.Scale} isn't within Range 0 -
1000");
        }

        if (Settings is { Scale: 0 } && Settings.VerbosityLevel <= Settings.Scale)
        {
            (failure ??= new()).AppendLine("VerbosityLevel must be > than Scale.");
        }

        return failure is not null
            ? ValidateOptionsResult.Fail(failure.ToString())
            : ValidateOptionsResult.Success;
    }

    [GeneratedRegex("^[a-zA-Z''-'\\s]{1,40}$")]
    private static partial Regex ValidationRegex();
}
```

`IValidateOptions`를 사용하면 유효성 검사 코드를 클래스로 이동할 수 있습니다.

> ⓘ **참고 항목**

이 예제 코드는 Microsoft.Extensions.Configuration.Json NuGet 패키지를 사용합니다.

이전 코드를 사용하면 다음 코드로 서비스를 구성할 때 유효성 검사가 사용하도록 설정됩니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

// Omitted for brevity...

builder.Services.Configure<SettingsOptions>(
    builder.Configuration.GetSection(
        SettingsOptions.ConfigurationSectionName));

builder.Services.TryAddEnumerable(
    ServiceDescriptor.Singleton
        <IValidateOptions<SettingsOptions>, ValidateSettingsOptions>());
```

재귀 유효성 검사와 `ValidateObjectMembers` `ValidateEnumeratedItems`

기본적으로 `DataAnnotations` 유효성 검사는 옵션 클래스 자체의 속성만 유효성을 검사합니다. 컬렉션에서 중첩된 개체 또는 항목의 유효성을 재귀적으로 검사하지 않습니다. 재귀 유효성 검사를 사용하도록 설정하려면 ValidateObjectMembersAttribute 및 ValidateEnumeratedItemsAttribute 특성을 사용합니다.

- 이 ValidateObjectMembersAttribute 특성을 사용하면 중첩된 개체의 재귀 유효성을 검사할 수 있습니다.
- 이 ValidateEnumeratedItemsAttribute 특성을 사용하면 열거 가능한 개체의 재귀 유효성을 검사할 수 있습니다.

다음 중첩된 옵션 클래스를 고려합니다.

C#

```
public sealed class DatabaseOptions
{
    [Required]
    [MinLength(1)]
    public string ConnectionString { get; set; } = string.Empty;

    [Range(1, 100)]
    public int MaxRetries { get; set; } = 3;

    [Range(1, 300)]
    public int TimeoutSeconds { get; set; } = 30;
}
```

C#

```
public sealed class ServerOptions
{
    [Required]
    [RegularExpression(@"^[a-zA-Z0-9\-\.]+$")]
    public string HostName { get; set; } = string.Empty;

    [Required]
    [Range(1, 65535)]
    public int Port { get; set; }
}
```

C#

```
public sealed class ApplicationOptionsWithAttribute
{
    public const string ConfigurationSectionName = "ApplicationWithAttribute";

    [Required]
    public required string ApplicationName { get; set; }

    // Validation recurses into DatabaseOptions.
    [ValidateObjectMembers]
    public DatabaseOptions Database { get; set; } = new();

    // Validation recurses into each ServerOptions in the list.
    [ValidateEnumeratedItems]
    public List<ServerOptions> Servers { get; set; } = [];
}
```

앞의 코드 `Database` 에서 속성은 형식 `DatabaseOptions`의 중첩된 개체입니다.

- 속성에 `[ValidateObjectMembers]` 특성이 적용되지 않으면 `DatabaseOptions` 속성에 대한 유효성 검사 특성(예: 에 있는 `[Required]`)이 평가되지 않습니다. `[ValidateObjectMembers]` 가 적용 시, 유효성 검사는 `Database` 속성에 재귀하고, `DataAnnotations` 특성에 따라 멤버의 유효성을 검사합니다.
- 컬렉션 속성인 `[ValidateEnumeratedItems]`에 `Servers` 특성이 적용되지 않으면 개별 `ServerOptions` 항목의 유효성 검사 특성이 평가되지 않습니다. `[ValidateEnumeratedItems]` 특성이 적용되면, 목록의 각 `ServerOptions` 항목이 `DataAnnotations` 특성에 따라 검증됩니다.

> **팁**
>
> `ValidateObjectMembersAttribute` 둘 다 `ValidateEnumeratedItemsAttribute` 컴파일 시간 옵션 유효성 검사 원본 생성기를 사용하여 성능을 향상합니다. 자세한 내용은 **컴파일 시간 옵션 유효성 검사 원본 생성**을 참조하세요.

옵션 사후 구성

IPostConfigureOptions<TOptions>를 사용하여 사후 구성을 설정합니다. 사후 구성은 모든 IConfigureOptions<TOptions> 구성이 발생한 후에 실행되며, 구성을 재정의해야 하는 시나리오에서 유용할 수 있습니다.

C#

```
builder.Services.PostConfigure<CustomOptions>(customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

PostConfigure를 사용하여 명명된 옵션을 사후 구성할 수 있습니다.

C#

```
builder.Services.PostConfigure<CustomOptions>("named_options_1", customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

모든 구성 인스턴스를 사후 구성하려면 PostConfigureAll을 사용합니다.

C#

```
builder.Services.PostConfigureAll<CustomOptions>(customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

참고 항목

- .NET의 구성
- .NET 라이브러리 작성자를 위한 옵션 패턴 지침

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 01. 22.

컴파일 시간 옵션 유효성 검사 원본 생성

2025. 06. 17.

옵션 패턴에서는 옵션의 유효성을 검사하기 위한 다양한 메서드가 표시됩니다. 이러한 메서드에는 데이터 주석 특성을 사용하거나 사용자 지정 유효성 검사기를 사용하는 것이 포함됩니다. 데이터 주석 특성은 런타임에 유효성을 검사하며 성능 비용이 발생할 수 있습니다. 이 문서에서는 옵션 유효성 검사 원본 생성기를 활용하여 컴파일 시간에 최적화된 유효성 검사 코드를 생성하는 방법을 보여 줍니다.

자동 IValidateOptions 구현 생성

옵션 패턴 문서에서는 옵션의 유효성을 검사하기 위해 인터페이스를 IValidateOptions<TOptions> 구현하는 방법을 보여 줍니다. 옵션 유효성 검사 원본 생성기는 옵션 클래스에서 `IValidateOptions` 데이터 주석 특성을 활용하여 인터페이스 구현을 자동으로 만들 수 있습니다.

다음 콘텐츠는 옵션 패턴 에 표시된 주석 특성 예제를 사용하고 옵션 유효성 검사 원본 생성기를 사용하도록 변환합니다.

다음 *appsettings.json* 파일을 고려하세요.

JSON

```
{
    "MyCustomSettingsSection": {
        "SiteTitle": "Amazing docs from awesome people!",
        "Scale": 10,
        "VerbosityLevel": 32
    }
}
```

다음 클래스는 `"MyCustomSettingsSection"` 구성 섹션에 바인딩되고 몇 가지 `DataAnnotations` 규칙을 적용합니다.

C#

```
using System.ComponentModel.DataAnnotations;

namespace ConsoleJson.Example;

public sealed class SettingsOptions
{
    public const string ConfigurationSectionName = "MyCustomSettingsSection";

    [Required]
```

```
    [RegularExpression(@"^[a-zA-Z''!''\s]{1,40}$")]
    public required string SiteTitle { get; set; }

    [Required]
    [Range(0, 1_000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public required int? Scale { get; set; }

    [Required]
    public required int? VerbosityLevel { get; set; }
}
```

위의 `SettingsOptions` 클래스에서 `ConfigurationSectionName` 속성에는 바인딩할 구성 섹션의 이름이 포함됩니다. 이 시나리오에서 옵션 개체는 구성 섹션의 이름을 제공합니다. 다음 데이터 주석 특성이 사용됩니다.

- RequiredAttribute: 속성이 필요한지 지정합니다.
- RegularExpressionAttribute: 속성 값이 지정된 정규식 패턴과 일치해야 임을 지정합니다.
- RangeAttribute: 속성 값이 지정된 범위 내에 있어야 임을 지정합니다.

> **팁**
>
> 속성 외에도 `RequiredAttribute` **필수** 한정자도 사용합니다. 옵션 객체의 소비자가 속성 값을 설정하는 것을 잊지 않도록 하는 데 도움이 됩니다. 유효성 검사 소스 생성 기능과는 관련이 없습니다.

다음 코드는 구성 섹션을 options 개체에 바인딩하고 데이터 주석의 유효성을 검사하는 방법을 예시합니다.

C#

```
using ConsoleJson.Example;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services
    .AddOptions<SettingsOptions>()

.Bind(builder.Configuration.GetSection(SettingsOptions.ConfigurationSectionName));

builder.Services
    .AddSingleton<IValidateOptions<SettingsOptions>, ValidateSettingsOptions>();

using IHost app = builder.Build();
```

```
var settingsOptions =
    app.Services.GetRequiredService<IOptions<SettingsOptions>>().Value;

await app.RunAsync();
```

> **팁**
>
> `<PublishAot>true</PublishAot>` 파일에 포함하여 AOT 컴파일을 사용하도록 설정하면 코드에서 IL2025 및 IL3050과 같은 경고를 생성할 수 있습니다. 이러한 경고를 완화하려면 구성 원본 생성기를 사용하는 것이 좋습니다. 구성 원본 생성기를 사용하도록 설정하려면 프로젝트 파일에 속성을 `<EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerator>` 추가합니다.

옵션 유효성 검사를 위해 컴파일 시간 원본 생성을 활용하여 성능 최적화 유효성 검사 코드를 생성하고 리플렉션이 필요하지 않도록 하여 AOT 호환 앱 빌드를 원활하게 수행할 수 있습니다. 다음 코드에서는 옵션 유효성 검사 원본 생성기를 사용하는 방법을 보여 줍니다.

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

[OptionsValidator]
public partial class ValidateSettingsOptions : IValidateOptions<SettingsOptions>
{
}
```

빈 부분 클래스의 OptionsValidatorAttribute 존재는 `IValidateOptions`을(를) 유효성 검사하는 `SettingsOptions` 인터페이스 구현을 생성하도록 옵션 유효성 검사 원본 생성기에 지시합니다. 옵션 유효성 검사 원본 생성기에서 생성되는 코드는 다음 예제와 유사합니다.

C#

```
// <auto-generated/>
#nullable enable
#pragma warning disable CS1591 // Compensate for
https://github.com/dotnet/roslyn/issues/54103
namespace ConsoleJson.Example
{
    partial class ValidateSettingsOptions
    {
        /// <summary>
        /// Validates a specific named options instance (or all when <paramref
```

```
name="name"/> is <see langword="null" />).
        /// </summary>
        /// <param name="name">The name of the options instance being validated.
</param>
        /// <param name="options">The options instance.</param>
        /// <returns>Validation result.</returns>

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Opti
ons.SourceGeneration", "8.0.9.3103")]
        [System.Diagnostics.CodeAnalysis.UnconditionalSuppressMessage("Trimming",
"IL2026:RequiresUnreferencedCode",
             Justification = "The created ValidationContext object is used in a
way that never call reflection")]
        public global::Microsoft.Extensions.Options.ValidateOptionsResult
Validate(string? name, global::ConsoleJson.Example.SettingsOptions options)
        {
            global::Microsoft.Extensions.Options.ValidateOptionsResultBuilder?
builder = null;
            var context = new
global::System.ComponentModel.DataAnnotations.ValidationContext(options);
            var validationResults = new
global::System.Collections.Generic.List<global::System.ComponentModel.DataAnnotati
ons.ValidationResult>();
            var validationAttributes = new
global::System.Collections.Generic.List<global::System.ComponentModel.DataAnnotati
ons.ValidationAttribute>(2);

            context.MemberName = "SiteTitle";
            context.DisplayName = string.IsNullOrEmpty(name) ?
"SettingsOptions.SiteTitle" : $"{name}.SiteTitle";

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A1
);

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A2
);
            if
(!global::System.ComponentModel.DataAnnotations.Validator.TryValidateValue(options
.SiteTitle, context, validationResults, validationAttributes))
            {
                (builder ??= new()).AddResults(validationResults);
            }

            context.MemberName = "Scale";
            context.DisplayName = string.IsNullOrEmpty(name) ?
"SettingsOptions.Scale" : $"{name}.Scale";
            validationResults.Clear();
            validationAttributes.Clear();

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A1
);

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A3
);
            if
```

```
(!global::System.ComponentModel.DataAnnotations.Validator.TryValidateValue(options
.Scale, context, validationResults, validationAttributes))
            {
                (builder ??= new()).AddResults(validationResults);
            }

            context.MemberName = "VerbosityLevel";
            context.DisplayName = string.IsNullOrEmpty(name) ?
"SettingsOptions.VerbosityLevel" : $"{name}.VerbosityLevel";
            validationResults.Clear();
            validationAttributes.Clear();

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A1
);
            if
(!global::System.ComponentModel.DataAnnotations.Validator.TryValidateValue(options
.VerbosityLevel, context, validationResults, validationAttributes))
            {
                (builder ??= new()).AddResults(validationResults);
            }

            return builder is null ?
global::Microsoft.Extensions.Options.ValidateOptionsResult.Success :
builder.Build();
        }
    }
}
namespace __OptionValidationStaticInstances
{

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Opti
ons.SourceGeneration", "8.0.9.3103")]
    file static class __Attributes
    {
        internal static readonly
global::System.ComponentModel.DataAnnotations.RequiredAttribute A1 = new
global::System.ComponentModel.DataAnnotations.RequiredAttribute();

        internal static readonly
global::System.ComponentModel.DataAnnotations.RegularExpressionAttribute A2 = new
global::System.ComponentModel.DataAnnotations.RegularExpressionAttribute(
            "^[a-zA-Z''-'\\s]{1,40}$");

        internal static readonly
__OptionValidationGeneratedAttributes.__SourceGen__RangeAttribute A3 = new
__OptionValidationGeneratedAttributes.__SourceGen__RangeAttribute(
            (int)0,
            (int)1000)
        {
            ErrorMessage = "Value for {0} must be between {1} and {2}."
        };
    }
}
namespace __OptionValidationStaticInstances
{
```

```
[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Options.SourceGeneration", "8.0.9.3103")]
    file static class __Validators
    {
    }
}
namespace __OptionValidationGeneratedAttributes
{

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Options.SourceGeneration", "8.0.9.3103")]
    [global::System.AttributeUsage(global::System.AttributeTargets.Property | global::System.AttributeTargets.Field | global::System.AttributeTargets.Parameter, AllowMultiple = false)]
    file class __SourceGen__RangeAttribute : global::System.ComponentModel.DataAnnotations.ValidationAttribute
    {
        public __SourceGen__RangeAttribute(int minimum, int maximum) : base()
        {
            Minimum = minimum;
            Maximum = maximum;
            OperandType = typeof(int);
        }
        public __SourceGen__RangeAttribute(double minimum, double maximum) : base()
        {
            Minimum = minimum;
            Maximum = maximum;
            OperandType = typeof(double);
        }
        public __SourceGen__RangeAttribute(global::System.Type type, string minimum, string maximum) : base()
        {
            OperandType = type;
            NeedToConvertMinMax = true;
            Minimum = minimum;
            Maximum = maximum;
        }
        public object Minimum { get; private set; }
        public object Maximum { get; private set; }
        public bool MinimumIsExclusive { get; set; }
        public bool MaximumIsExclusive { get; set; }
        public global::System.Type OperandType { get; }
        public bool ParseLimitsInInvariantCulture { get; set; }
        public bool ConvertValueInInvariantCulture { get; set; }
        public override string FormatErrorMessage(string name) =>

string.Format(global::System.Globalization.CultureInfo.CurrentCulture, GetValidationErrorMessage(), name, Minimum, Maximum);
        private bool NeedToConvertMinMax { get; }
        private bool Initialized { get; set; }
        public override bool IsValid(object? value)
        {
            if (!Initialized)
```

```
            {
                if (Minimum is null || Maximum is null)
                {
                    throw new global::System.InvalidOperationException("The
minimum and maximum values must be set to valid values.");
                }
                if (NeedToConvertMinMax)
                {
                    System.Globalization.CultureInfo culture =
ParseLimitsInInvariantCulture ?
global::System.Globalization.CultureInfo.InvariantCulture :
global::System.Globalization.CultureInfo.CurrentCulture;
                    Minimum = ConvertValue(Minimum, culture) ?? throw new
global::System.InvalidOperationException("The minimum and maximum values must be
set to valid values.");
                    Maximum = ConvertValue(Maximum, culture) ?? throw new
global::System.InvalidOperationException("The minimum and maximum values must be
set to valid values.");
                }
                int cmp =
((global::System.IComparable)Minimum).CompareTo((global::System.IComparable)Maximu
m);
                if (cmp > 0)
                {
                    throw new global::System.InvalidOperationException("The
maximum value '{Maximum}' must be greater than or equal to the minimum value
'{Minimum}'.");
                }
                else if (cmp == 0 && (MinimumIsExclusive || MaximumIsExclusive))
                {
                    throw new global::System.InvalidOperationException("Cannot use
exclusive bounds when the maximum value is equal to the minimum value.");
                }
                Initialized = true;
            }

            if (value is null or string { Length: 0 })
            {
                return true;
            }

            System.Globalization.CultureInfo formatProvider =
ConvertValueInInvariantCulture ?
global::System.Globalization.CultureInfo.InvariantCulture :
global::System.Globalization.CultureInfo.CurrentCulture;
            object? convertedValue;

            try
            {
                convertedValue = ConvertValue(value, formatProvider);
            }
            catch (global::System.Exception e) when (e is
global::System.FormatException or global::System.InvalidCastException or
global::System.NotSupportedException)
            {
```

```
                return false;
            }

            var min = (global::System.IComparable)Minimum;
            var max = (global::System.IComparable)Maximum;

            return
                (MinimumIsExclusive ? min.CompareTo(convertedValue) < 0 :
min.CompareTo(convertedValue) <= 0) &&
                (MaximumIsExclusive ? max.CompareTo(convertedValue) > 0 :
max.CompareTo(convertedValue) >= 0);
        }
        private string GetValidationErrorMessage()
        {
            return (MinimumIsExclusive, MaximumIsExclusive) switch
            {
                (false, false) => "The field {0} must be between {1} and {2}.",
                (true, false) => "The field {0} must be between {1} exclusive and
{2}.",
                (false, true) => "The field {0} must be between {1} and {2}
exclusive.",
                (true, true) => "The field {0} must be between {1} exclusive and
{2} exclusive.",
            };
        }
        private object? ConvertValue(object? value,
System.Globalization.CultureInfo formatProvider)
        {
            if (value is string stringValue)
            {
                value = global::System.Convert.ChangeType(stringValue,
OperandType, formatProvider);
            }
            else
            {
                value = global::System.Convert.ChangeType(value, OperandType,
formatProvider);
            }
            return value;
        }
    }
}
```

생성된 코드는 성능에 최적화되어 있으며 리플렉션에 의존하지 않습니다. AOT 호환도 가능합니다. 생성된 코드는 *Validators.g.cs* 파일에 배치됩니다.

> ⓘ **참고**

옵션 유효성 검사 원본 생성기를 사용하도록 설정하기 위해 추가 단계를 수행할 필요가 없습니다. 프로젝트에서 Microsoft.Extensions.Options 버전 8 이상을 참조하거나 ASP.NET 애플리케이션을 빌드할 때 기본적으로 자동으로 사용하도록 설정됩니다.

수행해야 하는 유일한 단계는 시작 코드에 다음을 추가하는 것입니다.

C#

```
builder.Services
    .AddSingleton<IValidateOptions<SettingsOptions>, ValidateSettingsOptions>();
```

참고

옵션 유효성 검사 원본 생성기를 사용하는 경우 호출 OptionsBuilderDataAnnotationsExtensions.ValidateDataAnnotations<TOptions>(OptionsBuilder<TOptions>) 이 필요하지 않습니다.

애플리케이션이 옵션 개체에 액세스하려고 하면 옵션 유효성 검사에 대해 생성된 코드가 실행되어 옵션 개체의 유효성을 검사합니다. 다음 코드 조각은 옵션 개체에 액세스하는 방법을 보여줍니다.

C#

```
var settingsOptions =
    app.Services.GetRequiredService<IOptions<SettingsOptions>>().Value;
```

대체된 데이터 주석 속성

생성된 코드를 면밀히 검토하면 처음에 속성RangeAttribute에 적용된 것과 같은 `SettingsOptions.Scale` 원래 데이터 주석 특성이 사용자 지정 특성(예: `__SourceGen__RangeAttribute` 사용자 지정 특성)으로 대체된 것을 확인할 수 있습니다. 이 대체 작업은 리플렉션에 의존하여 유효성 검사를 수행하므로 이루어집니다 `RangeAttribute`. 반면, `__SourceGen__RangeAttribute` 성능에 최적화된 사용자 지정 특성이며 리플렉션에 의존하지 않으므로 코드 AOT와 호환됩니다. 특성 대체의 동일한 패턴이 MaxLengthAttribute, MinLengthAttribute, LengthAttribute, 그리고 RangeAttribute에 적용됩니다.

사용자 지정 데이터 주석 특성을 개발하는 모든 사용자의 경우 유효성 검사를 위해 리플렉션을 사용하지 않는 것이 좋습니다. 대신 리플렉션에 의존하지 않는 강력한 형식의 코드를 만드는 것이 좋습니다. 이 방법은 AOT 빌드와의 원활한 호환성을 보장합니다.

참고하십시오

옵션 패턴

.NET 라이브러리 작성자용 옵션 패턴 지침

2025. 06. 17.

종속성 주입의 도움으로 서비스 및 해당 구성을 등록하면 *옵션 패턴을* 사용할 수 있습니다. 옵션 패턴은 라이브러리(및 서비스)를 사용하는 소비자가 자신의 옵션 클래스에 연결된 옵션 인터페이스의 인스턴스를 요구할 수 있도록 합니다. 강력한 형식의 개체를 통해 구성 옵션을 사용하면 일관된 값 표현을 보장하고, 데이터 주석으로 유효성을 검사할 수 있으며, 문자열 값을 수동으로 구문 분석해야 하는 부담이 제거됩니다. 라이브러리 소비자가 사용할 수 있는 많은 구성 공급자가 있습니다. 이러한 공급자를 사용하면 소비자는 여러 가지 방법으로 라이브러리를 구성할 수 있습니다.

.NET 라이브러리 작성자로서 라이브러리 소비자에게 옵션 패턴을 올바르게 노출하는 방법에 대한 일반적인 지침을 알아봅니다. 동일한 작업을 수행하는 다양한 방법과 몇 가지 고려 사항이 있습니다.

명명 규칙

규칙에 따라 서비스 등록을 담당하는 확장 메서드의 이름은 `Add{Service}` `{Service}` 의미 있고 설명적인 이름입니다. `Add{Service}` 확장 메서드는 ASP.NET Core 와 .NET에서 일반적입니다.

✔ 서비스를 다른 제품과 구분하는 이름을 고려합니다.

❌ 공식 Microsoft 패키지에서 .NET 에코시스템의 일부인 이름을 사용하지 마세요.

✔ 확장 메서드 `{Type}Extensions`를 노출하는 정적 클래스의 이름을 지정하는 것이 좋습니다. 여기서 확장 중인 형식은 다음과 `{Type}` 같습니다.

네임스페이스 지침

Microsoft 패키지는 `Microsoft.Extensions.DependencyInjection` 네임스페이스를 사용하여 다양한 서비스 제공을 등록하는 작업을 통합합니다.

✔ 패키지 제품을 명확하게 식별하는 네임스페이스를 고려합니다.

❌ 공식적이지 않은 Microsoft 패키지에는 `Microsoft.Extensions.DependencyInjection` 네임스페이스를 사용하지 마세요.

매개변수 없는

서비스가 최소한의 명시적 구성이나 명시적 구성 없이 작동할 수 있는 경우 매개 변수가 없는 확장 메서드를 고려합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
        this IServiceCollection services)
    {
        services.AddOptions<LibraryOptions>()
            .Configure(options =>
            {
                // Specify default option values
            });

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

앞의 코드에서는 `AddMyLibraryService`.

- 의 인스턴스를 확장합니다. IServiceCollection
- OptionsServiceCollectionExtensions.AddOptions<TOptions>(IServiceCollection)을/를 `LibraryOptions` 형식 매개 변수를 사용하여 호출합니다.
- 기본 옵션 값을 지정하는 호출 Configure을 연결합니다.

`IConfiguration` 매개 변수

소비자에게 많은 옵션을 노출하는 라이브러리를 작성하는 경우 매개 변수 확장 메서드를 `IConfiguration` 요구하는 것이 좋습니다. 예상되는 `IConfiguration` 인스턴스는 IConfiguration.GetSection 함수를 사용하여 구성의 명명된 섹션에 범위가 지정되어야 합니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
      this IServiceCollection services,
      IConfiguration namedConfigurationSection)
```

```
    {
        // Default library options are overridden
        // by bound configuration values.
        services.Configure<LibraryOptions>(namedConfigurationSection);

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

> **팁**
>
> 이 Configure<TOptions>(IServiceCollection, IConfiguration) 메서드는 NuGet 패키지의 Microsoft.Extensions.Options.ConfigurationExtensions 일부입니다.

앞의 코드에서는 `AddMyLibraryService`.

- 의 인스턴스를 확장합니다. IServiceCollection
- 매개 변수 정의 IConfiguration `namedConfigurationSection`
- Configure<TOptions>(IServiceCollection, IConfiguration) 호출은 `LibraryOptions`의 제네릭 형식 매개 변수와 `namedConfigurationSection` 인스턴스를 전달하여 구성합니다.

이 패턴의 소비자는 명명된 섹션의 범위가 지정된 `IConfiguration` 인스턴스를 제공합니다.

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(
    builder.Configuration.GetSection("LibraryOptions"));

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

호출 `.AddMyLibraryService` 은 IServiceCollection 유형에 대해 수행됩니다.

라이브러리 작성자로서 기본값을 지정하는 것은 사용자에게 달려 있습니다.

ⓘ **참고**

구성을 옵션 인스턴스에 바인딩할 수 있습니다. 그러나 이름 충돌이 발생할 위험이 있으므로 오류가 발생합니다. 또한 이러한 방식으로 수동으로 바인딩하는 경우 옵션 패턴의 사용량을 한 번 읽기로 제한합니다. 설정 변경 내용은 다시 바인딩되지 않습니다. 따라서 소비자는 IOptionsMonitor 인터페이스를 사용할 수 없습니다.

C#

```
services.AddOptions<LibraryOptions>()
    .Configure<IConfiguration>(
        (options, configuration) =>
            configuration.GetSection("LibraryOptions").Bind(options));
```

대신 확장 메서드를 BindConfiguration 사용해야 합니다. 이 확장 메서드는 구성을 옵션 인스턴스에 바인딩하고 구성 섹션에 대한 변경 토큰 원본도 등록합니다. 이를 통해 소비자는 IOptionsMonitor 인터페이스를 사용할 수 있습니다.

구성 섹션 경로 매개 변수

라이브러리의 소비자는 기본 `TOptions` 형식을 바인딩하는 구성 섹션 경로를 지정할 수 있습니다. 이 시나리오에서는 확장 메서드에서 매개 변수를 `string` 정의합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
      this IServiceCollection services,
      string configSectionPath)
    {
        services.AddOptions<SupportOptions>()
            .BindConfiguration(configSectionPath)
            .ValidateDataAnnotations()
            .ValidateOnStart();

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
```

```
    }
}
```

앞의 코드에서는 `AddMyLibraryService`.

- 의 인스턴스를 확장합니다. IServiceCollection
- 매개 변수 정의 `string configSectionPath`
- 호출:
  - AddOptions 의 제네릭 형식 매개 변수를 사용하여 `SupportOptions`
  - BindConfiguration 지정된 `configSectionPath` 매개 변수를 사용하여
  - ValidateDataAnnotations 데이터 주석 유효성 검사를 사용하도록 설정하려면
  - ValidateOnStart 런타임이 아닌 시작 시 유효성 검사를 적용하려면

다음 예제에서는 Microsoft.Extensions.Options.DataAnnotations NuGet 패키지를 사용하여 데이터 주석 유효성 검사를 사용하도록 설정합니다. 클래스는 `SupportOptions` 다음과 같이 정의됩니다.

C#

```
using System.ComponentModel.DataAnnotations;

public sealed class SupportOptions
{
    [Url]
    public string? Url { get; set; }

    [Required, EmailAddress]
    public required string Email { get; set; }

    [Required, DataType(DataType.PhoneNumber)]
    public required string PhoneNumber { get; set; }
}
```

다음 JSON *appsettings.json* 파일이 사용되어 있다고 상상해 보세요.

JavaScript

```
{
    "Support": {
        "Url": "https://support.example.com",
        "Email": "help@support.example.com",
        "PhoneNumber": "+1(888)-SUPPORT"
    }
}
```

`Action<TOptions>` 매개 변수

라이브러리의 소비자는 옵션 클래스의 인스턴스를 생성하는 람다 식을 제공하는 데 관심이 있을 수 있습니다. 이 시나리오에서는 확장 메서드에서 매개 변수를 `Action<LibraryOptions>` 정의합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
        this IServiceCollection services,
        Action<LibraryOptions> configureOptions)
    {
        services.Configure(configureOptions);

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

앞의 코드에서는 `AddMyLibraryService`.

- 의 인스턴스를 확장합니다. IServiceCollection
- Action<T> 매개 변수 `configureOptions`을 정의하는 `T`가 `LibraryOptions`입니다.
- Configure가 `configureOptions` 작업을 호출합니다.

이 패턴을 따르는 소비자는 `Action<LibraryOptions>` 매개 변수를 충족하는 람다 식(또는 대리자)을 사용합니다.

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(options =>
{
    // User defined option values
    // options.SomePropertyValue = ...
});
```

```
using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

Options 인스턴스 매개 변수

라이브러리의 소비자는 인라인 옵션 인스턴스를 제공하는 것을 선호할 수 있습니다. 이 시나리오에서는 options 개체의 인스턴스를 사용하는 확장 메서드를 노출합니다 `LibraryOptions`.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
      this IServiceCollection services,
      LibraryOptions userOptions)
    {
        services.AddOptions<LibraryOptions>()
            .Configure(options =>
            {
                // Overwrite default option values
                // with the user provided options.
                // options.SomeValue = userOptions.SomeValue;
            });

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

앞의 코드에서는 `AddMyLibraryService`.

- 의 인스턴스를 확장합니다. IServiceCollection
- OptionsServiceCollectionExtensions.AddOptions<TOptions>(IServiceCollection)을/를 `LibraryOptions` 형식 매개 변수를 사용하여 호출합니다.
- 지정된 Configure 인스턴스에서 재정의할 수 있는 기본 옵션 값을 지정하는 호출을 연결 `userOptions`합니다.

이 패턴의 소비자는 원하는 속성 값을 인라인으로 정의하는 클래스의 `LibraryOptions` 인스턴스를 제공합니다.

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(new LibraryOptions
{
    // Specify option values
    // SomePropertyValue = ...
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

구성 후 설정

모든 구성 옵션 값이 바인딩되거나 지정되면 사후 구성 기능을 사용할 수 있습니다. 앞에서 설명한 것과 동일한 Action<TOptions> 매개 변수 를 노출하면 호출 PostConfigure하도록 선택할 수 있습니다. 사후 구성 실행은 모든 `.Configure` 호출 후에 실행됩니다. `PostConfigure`를 고려하실 몇 가지 이유가 있습니다.

- **실행 순서**: `.Configure` 호출에 설정된 구성 값을 다시 재정의할 수 있습니다.
- **유효성 검사**: 다른 모든 구성이 적용된 후 기본값이 설정되었는지 확인할 수 있습니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
      this IServiceCollection services,
      Action<LibraryOptions> configureOptions)
    {
        services.PostConfigure(configureOptions);

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();
```

```
        return services;
    }
}
```

앞의 코드에서는 `AddMyLibraryService`.

- 의 인스턴스를 확장합니다. IServiceCollection
- Action<T> 매개 변수 `configureOptions`을 정의하는 `T`가 `LibraryOptions`입니다.
- PostConfigure가 `configureOptions` 작업을 호출합니다.

이 패턴의 소비자는 비포스트 구성 시나리오에서 매개변수 `Action<TOptions>`처럼, 매개변수 를 만족시키는 람다 식(또는 대리자)을 제공합니다.

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(options =>
{
    // Specify option values
    // options.SomePropertyValue = ...
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

참고하십시오

- .NET의 옵션 패턴
- .NET에서 종속성 주입
- 종속성 주입 지침

C# 및 .NET에서 로깅

.NET은 ILogger API를 통해 고성능의 구조화된 로깅을 지원하여 애플리케이션 동작을 모니터링하고 문제를 진단하는 데 도움을 줍니다. 다른 대상에 로그를 쓰도록 다른 로깅 공급자 를 구성합니다. 기본 로깅 공급자는 기본 제공되며 많은 타사 공급자를 사용할 수 있습니다.

시작하기

이 첫 번째 예제에서는 기본 사항을 보여 주지만 간단한 콘솔 앱에만 적합합니다. 이 샘플 콘솔 앱은 다음 NuGet 패키지를 사용합니다.

- Microsoft.Extensions.Logging
- Microsoft.Extensions.Logging.Console

다음 섹션에서는 규모, 성능, 구성 및 일반적인 프로그래밍 패턴을 고려하여 코드를 개선하는 방법을 알아보세요.

C#

```
using Microsoft.Extensions.Logging;

using ILoggerFactory factory = LoggerFactory.Create(builder =>
builder.AddConsole());
ILogger logger = factory.CreateLogger("Program");
logger.LogInformation("Hello World! Logging is {Description}.", "fun");
```

위의 예제는 다음과 같습니다.

- ILoggerFactory를 만듭니다. `ILoggerFactory`에는 로그 메시지가 전송되는 위치를 결정하는 모든 구성이 저장됩니다. 이 경우 로그 메시지가 콘솔에 기록되도록 콘솔 로깅 공급자 를 구성합니다.
- "Program"이라는 범주를 사용하여 ILogger을(를) 만듭니다. 범주는 `string`에 의해 `ILogger` 객체가 기록한 각 메시지와 연결된 요소입니다. 로그를 검색하거나 필터링할 때 동일한 클래스(또는 범주)의 로그 메시지를 그룹화합니다.
- LogInformation을 호출하여 `Information` 수준에 메시지를 로그합니다. 로그 수준은 기록된 이벤트의 심각도를 나타내고 덜 중요한 로그 메시지를 필터링합니다. 로그 항목에는 메시지 템플릿`"Hello World! Logging is {Description}."`과(와) 키-값 쌍 `Description = fun`도 포함됩니다. 키 이름(또는 자리 표시자)은 템플릿의 중괄호 안에 있는 단어에서 비롯되며 값은 나머지 메서드 인수에서 가져옵니다.

이 예의 프로젝트 파일에는 두 개의 NuGet 패키지가 포함되어 있습니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Logging" Version="10.0.2" />
    <PackageReference Include="Microsoft.Extensions.Logging.Console"
Version="10.0.2" />
  </ItemGroup>

</Project>
```

> **팁**
>
> 모든 로깅 예제 소스 코드는 **샘플 브라우저**에서 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET의 로깅**을 참조하세요.

단순하지 않은 앱 로그인

덜 간단한 시나리오에서 로깅할 때 이전 예제를 다음과 같이 변경하는 것이 좋습니다.

- 애플리케이션에서 DI(종속성 주입) 또는 ASP.NET의 WebApplication이나 `ILoggerFactory`와 같은 호스트를 사용하는 경우, 개체를 직접 만들지 말고, 해당 DI 컨테이너에서 제공하는 `ILogger` 개체를 사용하세요. 자세한 내용은 DI 및 호스트 통합을 참조하세요.
- 로깅을 위한 컴파일 시간 원본 생성은 일반적으로 확장 메서드(예:)`ILogger`에 대한 `LogInformation` 더 나은 대안입니다. 원본 생성 로깅은 더 나은 성능과 강력한 입력을 제공하며 메서드 전체에 상수가 분산되지 `string` 않도록 합니다. 단점은 이 기술을 사용하려면 좀 더 많은 코드가 필요하다는 것입니다.

  C#

```
using Microsoft.Extensions.Logging;

internal partial class Program
{
    static void Main(string[] args)
    {
        using ILoggerFactory factory = LoggerFactory.Create(builder =>
builder.AddConsole());
        ILogger logger = factory.CreateLogger("Program");
```

```
        LogStartupMessage(logger, "fun");
    }

    [LoggerMessage(Level = LogLevel.Information, Message = "Hello World!
Logging is {Description}.")]
    static partial void LogStartupMessage(ILogger logger, string description);
}
```

- 로그 범주 이름을 선택하는 데 권장되는 방법은 로그 메시지를 만드는 클래스의 정규화된 이름을 사용하는 것입니다. 이렇게 하면 로그 메시지를 생성한 코드와 다시 연결하고 로그를 필터링할 때 적절한 수준의 제어를 제공합니다. 메서드의 형식 매개 변수에서 클래스 형식을 지정합니다 CreateLogger .

C#

```
using Microsoft.Extensions.Logging;

internal class Program
{
    static void Main(string[] args)
    {
        using ILoggerFactory factory = LoggerFactory.Create(builder =>
builder.AddConsole());
        ILogger logger = factory.CreateLogger<Program>();
        logger.LogInformation("Hello World! Logging is {Description}.",
"fun");
    }
}
```

- 콘솔 로그를 유일한 프로덕션 모니터링 솔루션으로 사용하지 않는 경우 사용하려는 로깅 공급자를 추가합니다. 예를 들어 OpenTelemetry 를 사용하여 OTLP(OpenTelemetry 프로토콜)를 통해 로그를 보냅니다.

C#

```
using Microsoft.Extensions.Logging;
using OpenTelemetry.Logs;

using ILoggerFactory factory = LoggerFactory.Create(builder =>
{
    builder.AddOpenTelemetry(logging =>
    {
        logging.AddOtlpExporter();
    });
});
ILogger logger = factory.CreateLogger("Program");
logger.LogInformation("Hello World! Logging is {Description}.", "fun");
```

호스트 및 종속성 주입과 통합

애플리케이션에서 종속성 주입 (DI) 또는 WebApplication과 같은 ASP.NET 호스트나 제너릭 호스트를 사용하는 경우, 객체를 직접 만들지 않고 DI 컨테이너에서 `ILoggerFactory` 및 `ILogger` 객체를 사용하십시오.

DI에서 ILogger 가져오기

이 예제에서는 ASP.NET 최소 API를 사용하여 호스트된 앱에서 ILogger 개체를 가져옵니다.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<ExampleHandler>();

var app = builder.Build();

var handler = app.Services.GetRequiredService<ExampleHandler>();
app.MapGet("/", handler.HandleRequest);

app.Run();

partial class ExampleHandler(ILogger<ExampleHandler> logger)
{
    public string HandleRequest()
    {
        LogHandleRequest(logger);
        return "Hello World";
    }

    [LoggerMessage(LogLevel.Information, "ExampleHandler.HandleRequest was
called")]
    public static partial void LogHandleRequest(ILogger logger);
}
```

위의 예제는 다음과 같습니다.

- `ExampleHandler`(이)라는 싱글톤 서비스를 만들고 들어오는 웹 요청을 매핑하여 `ExampleHandler.HandleRequest` 함수를 실행합니다.
- 줄 12는 C# 12에 추가된 기능인 ExampleHandler에 대한 기본 생성자 정의합니다. 이전 스타일 C# 생성자를 사용하는 것은 동일하게 잘 작동하지만 좀 더 장황합니다.
- 생성자는 `ILogger<ExampleHandler>` 형식의 매개 변수를 정의합니다. ILogger<TCategoryName>은(는) ILogger에서 파생되며 `ILogger` 개체에 있는 범주를 나타냅니다. DI 컨테이너는 올바른 범주를 가진 `ILogger`을(를) 찾아 생성자 인수로 제공합니다.

해당 범주의 `ILogger`이(가) 아직 없는 경우 DI 컨테이너는 서비스 공급자의 `ILoggerFactory`에서 자동으로 만듭니다.

- `logger` 생성자에서 받은 매개 변수는 `HandleRequest` 함수에서 로깅 목적으로 사용됩니다.

호스트가 제공하는 ILoggerFactory

호스트 작성기는 기본 구성을 초기화한 다음 호스트가 빌드되면 구성된 `ILoggerFactory` 개체를 호스트의 DI 컨테이너에 추가합니다. 호스트를 빌드하기 전에 다른 호스트에서 또는 유사한 API를 통해 HostApplicationBuilder.LoggingWebApplicationBuilder.Logging로깅 구성을 조정합니다. 호스트는 *appsettings.json* 및 환경 변수와 같은 기본 구성 원본의 로깅 구성도 적용합니다. 자세한 내용은 .NET의 구성을 참조하세요.

이 예제는 이전 예제를 확장하여 `ILoggerFactory`에서 제공하는 `WebApplicationBuilder`을(를) 사용자 지정합니다. 이는 OTLP(OpenTelemetry 프로토콜)를 통해 로그를 전송하는 로깅 공급자로 OpenTelemetry를 추가합니다.

C#

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Logging.AddOpenTelemetry(logging => logging.AddOtlpExporter());
builder.Services.AddSingleton<ExampleHandler>();
WebApplication app = builder.Build();
```

DI를 사용하여 ILoggerFactory 만들기

호스트 없이 DI 컨테이너를 사용하는 경우 AddLogging을(를) 사용하여 컨테이너에 `ILoggerFactory`을(를) 구성하고 추가합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

// Add services to the container including logging
var services = new ServiceCollection();
services.AddLogging(builder => builder.AddConsole());
services.AddSingleton<ExampleService>();
IServiceProvider serviceProvider = services.BuildServiceProvider();

// Get the ExampleService object from the container
ExampleService service = serviceProvider.GetRequiredService<ExampleService>();

// Do some pretend work
service.DoSomeWork(10, 20);

class ExampleService(ILogger<ExampleService> logger)
```

```
{
    public void DoSomeWork(int x, int y)
    {
        logger.LogInformation("DoSomeWork was called. x={X}, y={Y}", x, y);
    }
}
```

위의 예제는 다음과 같습니다.

- 콘솔에 쓰도록 구성된 `ILoggerFactory`이(가) 포함된 DI 서비스 컨테이너 생성함
- 컨테이너에 싱글톤 `ExampleService` 추가됨
- DI 컨테이너에서 인스턴스를 `ExampleService` 만들었으며 생성자 인수로 사용할 인스턴스도 자동으로 만들었습니다 `ILogger<ExampleService>` .
- `ExampleService.DoSomeWork`이(가) 호출되어 `ILogger<ExampleService>`을(를) 사용하여 콘솔에 메시지를 기록했습니다.

로깅 구성

코드 또는 구성 파일 및 환경 변수와 같은 외부 원본을 통해 로깅 구성을 설정합니다. 애플리케이션을 다시 빌드하지 않고 변경할 수 있으므로 가능한 경우 외부 구성을 사용하는 것이 좋습니다. 그러나 로깅 공급자 설정과 같은 일부 작업은 코드에서만 구성할 수 있습니다.

코드 없이 로깅 구성

호스트 `"Logging"` 앱의 경우 *appsettings*`.{Environment}`*.json* 파일 섹션은 일반적으로 로깅 구성을 제공합니다. 호스트를 사용하지 않는 앱의 경우 외부 구성 원본을 명시적으로 설정 하거나 코드에서 구성합니다 .

다음 *appsettings.Development.json* 파일은 .NET 작업자 서비스 템플릿으로 생성되었습니다.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

앞의 JSON에서:

- `"Default"`, `"Microsoft"` 및 `"Microsoft.Hosting.Lifetime"` 로그 수준 범주가 지정됩니다.

- 값은 `"Default"` 달리 지정되지 않은 모든 범주에 적용되며 모든 범주 `"Information"`에 대한 모든 기본값을 효과적으로 만듭니다. 범주의 값을 지정하여 이 동작을 재정의합니다.
- `"Microsoft"` 범주는 `"Microsoft"`로 시작하는 모든 범주에 적용됩니다.
- `"Microsoft"` 범주는 `Warning` 이상의 로그 수준에서 로그됩니다.
- `"Microsoft.Hosting.Lifetime"` 범주는 `"Microsoft"` 범주보다 더 구체적이므로 `"Microsoft.Hosting.Lifetime"` 범주는 로그 수준 `"Information"` 이상에서 로그됩니다.
- 특정 로그 공급자는 지정되지 않으므로 `LogLevel`Windows EventLog를 제외한 모든 사용 가능한 로깅 공급자에 적용됩니다.

`Logging` 속성은 LogLevel 및 로그 공급자 속성을 포함할 수 있습니다. `LogLevel` 속성은 선택한 범주에 대해 로그할 최소 수준을 지정합니다. 위의 JSON에서는 `Information` 및 `Warning` 로그 수준을 지정했습니다. `LogLevel` 로그의 심각도를 나타내며 0에서 6 사이의 범위입니다.

`Trace` = 0, `Debug` = 1, `Information` = 2, `Warning` = 3, `Error` = 4, `Critical` = 5 및 `None` = 6

`LogLevel`을 지정하면 지정된 수준 이상의 메시지에 대해 로깅을 사용하도록 설정됩니다. 위의 JSON에서 `Default` 범주는 `Information` 이상에 대해 로그됩니다. 예를 들어 `Information`, `Warning`, `Error` 및 `Critical` 메시지가 로그됩니다. `LogLevel` 지정하지 않으면 로깅은 `Information` 수준으로 기본 설정됩니다. 자세한 내용은 로그 수준을 참조하세요.

공급자 속성에서 `LogLevel` 속성을 지정할 수 있습니다. 공급자 아래의 `LogLevel`은 해당 공급자에 대해 로그할 수준을 지정하고 공급자 이외의 로그 설정을 재정의합니다. 다음 *appsettings.json* 파일을 고려하세요.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Error",
            "Microsoft": "Warning"
        },
        "Debug": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft.Hosting": "Trace"
            }
        },
        "EventSource": {
            "LogLevel": {
                "Default": "Warning"
            }
        }
    }
}
```

`Logging.{ProviderName}.LogLevel`의 설정은 `Logging.LogLevel`의 설정을 재정의합니다. 위의 JSON에서는 `Debug` 공급자의 기본 로그 수준이 `Information`로 설정되었습니다.

`Logging:Debug:LogLevel:Default:Information`

위 설정은 `Information`을 제외하고 모든 `Logging:Debug:` 범주에 대해 `Microsoft.Hosting` 로그 수준을 지정합니다. 특정 범주를 나열하면 기본 범주가 특정 범주로 대체됩니다. 위의 JSON에서 `Logging:Debug:LogLevel` 범주 `"Microsoft.Hosting"` 및 `"Default"`는 `Logging:LogLevel`의 설정을 재정의합니다.

다음 중 한 가지에 대한 최소 로그 수준을 지정합니다.

- 특정 공급자: 예를 들면 `Logging:EventSource:LogLevel:Default:Information`과 같습니다.
- 특정 범주: 예를 들어 `Logging:LogLevel:Microsoft:Warning`
- 모든 공급자와 모든 범주: `Logging:LogLevel:Default:Warning`

최소 수준 이하의 로그는 ***다음이 아닙니다***.

- 공급자에 전달됩니다.
- 기록되거나 표시됩니다.

모든 로그를 표시하지 않으려면 LogLevel.None을 지정합니다. `LogLevel.None`의 값은 `LogLevel.Critical`(5)보다 높은 6입니다.

공급자가 로그 범위를 지원하는 경우 `IncludeScopes`는 사용 가능 여부를 나타냅니다. 자세한 내용은 로그 범위을 참고하세요.

다음 *appsettings.json* 파일에는 모든 기본 제공 공급자에 대한 설정이 포함되어 있습니다.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Error",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Warning"
        },
        "Debug": {
            "LogLevel": {
                "Default": "Information"
            }
        },
        "Console": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft.Extensions.Hosting": "Warning",
                "Default": "Information"
```

```
            }
        },
        "EventSource": {
            "LogLevel": {
                "Microsoft": "Information"
            }
        },
        "EventLog": {
            "LogLevel": {
                "Microsoft": "Information"
            }
        },
        "AzureAppServicesFile": {
            "IncludeScopes": true,
            "LogLevel": {
                "Default": "Warning"
            }
        },
        "AzureAppServicesBlob": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft": "Information"
            }
        },
        "ApplicationInsights": {
            "LogLevel": {
                "Default": "Information"
            }
        }
    }
}
```

앞의 예제에서:

- 범주 및 수준은 제안된 값이 아닙니다. 샘플은 모든 기본 공급자를 보여 줍니다.
- `Logging.{ProviderName}.LogLevel`의 설정은 `Logging.LogLevel`의 설정을 재정의합니다. 예를 들어 `Debug.LogLevel.Default` 수준은 `LogLevel.Default` 수준을 무시합니다.
- 각 공급자의 별칭을 사용합니다. 각 공급자는 정규화된 형식 이름 대신 구성에 사용할 수 있는 *별칭*을 정의합니다. 기본 제공 공급자의 별칭은 다음과 같습니다.
  - `Console`
  - `Debug`
  - `EventSource`
  - `EventLog`
  - `AzureAppServicesFile`
  - `AzureAppServicesBlob`
  - `ApplicationInsights`

명령줄, 환경 변수 및 기타 구성으로 로그 수준 설정

구성 공급자를 사용하여 로그 수준을 설정합니다. 예를 들어, 이름이 `Logging:LogLevel:Microsoft`이고 값이 `Information`인 지속형 환경 변수를 생성합니다.

명령줄

로그 수준 값이 지정되면 지속형 환경 변수를 만들고 할당합니다.

CMD

```
:: Assigns the env var to the value
setx "Logging__LogLevel__Microsoft" "Information" /M
```

*명령 프롬프트*의 '새' 인스턴스에서 환경 변수를 읽습니다.

CMD

```
:: Prints the env var value
echo %Logging__LogLevel__Microsoft%
```

이전 환경 설정은 환경에서 유지됩니다. .NET 작업자 서비스 템플릿을 사용하여 만든 앱을 사용할 때 설정을 테스트하려면 환경 변수가 할당된 후에 프로젝트 디렉터리에서 `dotnet run` 명령을 사용합니다.

.NET CLI

```
dotnet run
```

팁

환경 변수를 설정한 후에는 IDE(통합 개발 환경)를 다시 시작하여 새로 추가된 환경 변수를 사용할 수 있는지 확인하세요.

Azure App Service에서 페이지의 >을 선택합니다. Azure App Service 애플리케이션 설정은,

- 미사용 시 암호화되고 암호화된 채널을 통해 전송됩니다.
- 환경 변수로 노출됩니다.

환경 변수를 사용하여 .NET 구성 값을 설정하는 방법에 대한 자세한 내용은 환경 변수를 참조하세요.

코드를 사용하여 로깅 구성

코드에서 로깅을 구성하려면 ILoggingBuilder API를 사용합니다. 다른 위치에서 액세스할 수 있습니다.

- `ILoggerFactory`을(를) 직접 만들 때 LoggerFactory.Create에서 구성합니다.
- 호스트 없이 DI를 사용하는 경우 LoggingServiceCollectionExtensions.AddLogging에서 구성합니다.
- 호스트를 사용하는 경우, HostApplicationBuilder.Logging, WebApplicationBuilder.Logging 또는 다른 호스트의 특정 API를 사용하여 구성합니다.

이 예제에서는 콘솔 로깅 공급자 및 여러 필터를 설정하는 방법을 보여 줍니다.

C#

```
using Microsoft.Extensions.Logging;

using var loggerFactory = LoggerFactory.Create(static builder =>
{
    builder
        .AddFilter("Microsoft", LogLevel.Warning)
        .AddFilter("System", LogLevel.Warning)
        .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
        .AddConsole();
});

ILogger logger = loggerFactory.CreateLogger<Program>();
logger.LogDebug("Hello {Target}", "Everyone");
```

앞의 예제 AddFilter 에서는 다양한 범주에 대해 사용하도록 설정된 로그 수준을 조정합니다. AddConsole 는 콘솔 로깅 공급자를 추가합니다. 기본적으로 `Debug` 심각도가 있는 로그는 사용하도록 설정되지 않지만 구성에서 필터를 조정했기 때문에 디버그 메시지 "Hello Everyone"이 콘솔에 표시됩니다.

필터링 규칙 적용 방식

ILogger<TCategoryName> 개체를 만들 때 ILoggerFactory 개체는 공급자마다 해당 로거에 적용할 단일 규칙을 선택합니다. 인스턴스는 `ILogger` 선택한 규칙에 따라 작성하는 모든 메시지를 필터링합니다. 사용 가능한 규칙 중에서 각 공급자 및 범주 쌍과 가장 관련이 많은 규칙이 선택됩니다.

지정된 범주에 대한 `ILogger`가 생성되면 다음 알고리즘이 각 공급자에 사용됩니다.

- 공급자 또는 공급자의 별칭과 일치하는 모든 규칙을 선택합니다. 일치하는 규칙이 없는 경우 빈 공급자가 있는 모든 규칙을 선택합니다.
- 앞 단계의 결과에서 일치하는 범주 접두사가 가장 긴 규칙을 선택합니다. 일치하는 규칙이 없는 경우 범주를 지정하지 않는 규칙을 모두 선택합니다.

- 여러 규칙을 선택하는 경우 **마지막** 규칙을 사용합니다.
- 규칙을 선택하지 않은 경우 LoggingBuilderExtensions.SetMinimumLevel(ILoggingBuilder, LogLevel)를 사용하여 최소 로깅 수준을 지정합니다.

로그 범주

`ILogger` 개체가 생성되면 '범주'가 지정됩니다. 해당 범주는 `ILogger`의 해당 인스턴스에서 만든 각 로그 메시지에 포함됩니다. 범주 문자열은 임의로 지정되지만 규칙은 정규화된 클래스 이름을 사용하는 것입니다. 예를 들어 서비스가 다음 개체와 같이 정의된 애플리케이션에서 범주는 `"Example.DefaultService"`일 수 있습니다.

C#

```
namespace Example
{
    public class DefaultService : IService
    {
        private readonly ILogger<DefaultService> _logger;

        public DefaultService(ILogger<DefaultService> logger) =>
            _logger = logger;

        // ...
    }
}
```

추가 분류가 필요한 경우 규칙은 정규화된 클래스 이름에 하위 범주를 추가하여 계층적 이름을 사용하고 LoggerFactory.CreateLogger을(를) 사용하여 범주를 명시적으로 지정하는 것입니다.

C#

```
namespace Example
{
    public class DefaultService : IService
    {
        private readonly ILogger _logger;

        public DefaultService(ILoggerFactory loggerFactory) =>
            _logger =
loggerFactory.CreateLogger("Example.DefaultService.CustomCategory");

        // ...
    }
}
```

고정 이름을 사용하여 호출 `CreateLogger` 하는 것은 이벤트를 범주별로 구성할 수 있도록 여러 클래스/형식에서 사용할 때 유용합니다.

`ILogger<T>`는 `CreateLogger`의 정규화된 형식 이름으로 `T`를 호출하는 것과 동일합니다.

로그 수준

다음 표에서는 LogLevel 값, 편리한 `Log{LogLevel}` 확장 메서드 및 추천 사용법을 설명합니다.

테이블 확장

LogLevel	가치	메서드	Description
흔적	0	LogTrace	가장 자세한 메시지를 포함합니다. 이러한 메시지에는 중요한 앱 데이터가 포함될 수 있습니다. 해당 메시지는 기본적으로 사용하지 않도록 설정되며 프로덕션에서 사용하도록 설정하면 ***안 됩니다***.
디버그	1	LogDebug	디버깅 및 개발을 위한 수준입니다. 너무 많으므로 프로덕션에서는 주의해서 사용합니다.
정보	2	LogInformation	앱의 일반적인 흐름을 추적합니다. 장기 값이 있을 수 있습니다.
경고	3	LogWarning	비정상적이거나 예기치 않은 사건을 위한 것입니다. 일반적으로 앱의 오류를 발생시키지 않는 오류 또는 조건을 포함합니다.
Error	4	LogError	처리할 수 없는 오류 및 예외의 경우 해당 메시지는 전체 앱 오류가 아닌 현재 작업 또는 요청의 오류를 의미합니다.
중요	5	LogCritical	즉각적인 대응이 필요한 오류에 대해. 예: 데이터 손실 시나리오, 디스크 공간 부족.
없음	6		메시지를 기록하지 않도록 지정합니다.

위의 표에서는 `LogLevel`이 심각도가 낮은 것에서 높은 것 순으로 표시됩니다.

Log 메서드의 첫 번째 매개 변수인 LogLevel은 로그의 심각도를 나타냅니다. 대부분의 개발자는 `Log(LogLevel, ...)` 대신 Log{LogLevel} 확장 메서드를 호출합니다. `Log{LogLevel}` 확장 메서드는 `Log` 메서드를 호출하고 `LogLevel`을 지정합니다. 예를 들어 다음 두 로깅 호출은 기능이 동일하며 같은 로그를 생성합니다.

C#

```
public void LogDetails()
{
    var logMessage = "Details for log.";

    _logger.Log(LogLevel.Information, AppLogEvents.Details, logMessage);
    _logger.LogInformation(AppLogEvents.Details, logMessage);
}
```

`AppLogEvents.Details`는 이벤트 ID이며 상수 Int32 값으로 암시적으로 표시됩니다.
`AppLogEvents`는 여러 개의 명명된 식별자 상수를 노출하고 로그 이벤트 ID 섹션에 표시되는 클래스입니다.

다음 코드는 `Information` 및 `Warning` 로그를 만듭니다.

C#

```
public async Task<T> GetAsync<T>(string id)
{
    _logger.LogInformation(AppLogEvents.Read, "Reading value for {Id}", id);

    var result = await _repository.GetAsync(id);
    if (result is null)
    {
        _logger.LogWarning(AppLogEvents.ReadNotFound, "GetAsync({Id}) not found",
id);
    }

    return result;
}
```

앞의 코드에서 첫 번째 `Log{LogLevel}` 매개 변수`AppLogEvents.Read`는 로그 이벤트 ID입니다. 두 번째 매개 변수는 나머지 메서드 매개 변수가 제공하는 인수 값에 대한 자리 표시자를 포함하고 있는 메시지 템플릿입니다. 메서드 매개 변수는 이 문서의 뒷부분에 있는 메시지 템플릿 섹션에 설명되어 있습니다.

적절한 로그 수준을 구성하고 올바른 `Log{LogLevel}` 메서드를 호출하여 특정 스토리지 매체에 기록되는 로그 출력의 크기를 제어합니다. 다음은 그 예입니다.

- 생산 중
  - `Trace` 또는 `Debug` 수준 로깅은 자세한 로그 메시지를 대량으로 생성합니다. 비용을 관리하고 데이터 스토리지 제한을 초과하지 않으려면 `Trace` 및 `Debug` 수준 메시지를 대용량, 저비용 데이터 저장소에 로그합니다. `Trace` 및 `Debug`을 특정 범주로 제한하는 것이 좋습니다.
  - `Warning`~`Critical` 수준의 로깅은 적은 로그 메시지를 생성합니다.
    - 비용과 스토리지 제한이 일반적으로 문제가 되지 않습니다.
    - 로그가 적으므로 데이터 저장소를 더 유연하게 선택할 수 있습니다.
- 개발 중:
  - `Warning`로 설정합니다.
  - 문제를 해결할 때는 `Trace` 또는 `Debug` 메시지를 추가합니다. 출력을 제한하려면 조사 중인 범주에 대해서만 `Trace` 또는 `Debug`을 설정합니다.

다음 JSON에서는 `Logging:Console:LogLevel:Microsoft:Information`을 설정합니다.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Microsoft": "Warning"
        },
        "Console": {
            "LogLevel": {
                "Microsoft": "Information"
            }
        }
    }
}
```

로그 이벤트 ID

각 로그에서 '이벤트 식별자'를 지정할 수 있으며, 는 EventId와 선택적 `Id` 읽기 전용 속성을 포함하는 구조체입니다.`Name` 샘플 소스 코드는 `AppLogEvents` 클래스를 사용하여 이벤트 ID를 정의합니다.

C#

```
using Microsoft.Extensions.Logging;

internal static class AppLogEvents
{
    internal static EventId Create = new(1000, "Created");
    internal static EventId Read = new(1001, "Read");
    internal static EventId Update = new(1002, "Updated");
    internal static EventId Delete = new(1003, "Deleted");

    // These are also valid EventId instances, as there's
    // an implicit conversion from int to an EventId
    internal const int Details = 3000;
    internal const int Error = 3001;

    internal static EventId ReadNotFound = 4000;
    internal static EventId UpdateNotFound = 4001;

    // ...
}
```

> **팁**
>
> `int`을(를) `EventId`(으)로 변환하는 방법에 대한 자세한 내용은 **EventId.Implicit(Int32에서 EventId로) 연산자**를 참조하세요.

이벤트 ID는 이벤트 집합을 연결합니다. 예를 들어 리포지토리에서 값 읽기와 관련된 모든 로그는 `1001`일 수 있습니다.

로깅 공급자는 ID 필드, 로깅 메시지에서 이벤트 ID를 기록하거나 전혀 기록하지 않을 수 있습니다. 디버그 공급자는 이벤트 ID를 표시하지 않습니다. 콘솔 공급자는 범주 뒤에 대괄호로 이벤트 ID를 표시합니다.

콘솔

```
info: Example.DefaultService.GetAsync[1001]
      Reading value for a1b2c3
warn: Example.DefaultService.GetAsync[4000]
      GetAsync(a1b2c3) not found
```

일부 로깅 공급자는 ID에 대한 필터링을 허용하는 필드에 이벤트 ID를 저장합니다.

로그 메시지 템플릿

각 로그 API는 메시지 템플릿을 사용합니다. 메시지 템플릿에는 인수가 제공되는 자리 표시자를 포함할 수 있습니다. 번호가 아닌 자리 표시자의 이름을 사용합니다. 값을 제공하는 데 사용되는 매개 변수를 결정하는 것은 자리 표시자의 이름이 아닌 순서입니다. 다음 코드에서는 매개 변수 이름이 메시지 템플릿의 순서와 일치하지 않습니다.

C#

```
string p1 = "param1";
string p2 = "param2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

앞의 코드는 매개 변수 값을 순서대로 사용하여 로그 메시지를 만듭니다.

text

```
Parameter values: param1, param2
```

> **참고 항목**
>
> 서수 기반인 단일 메시지 템플릿 내에서 여러 자리 표시자를 사용할 때는 주의해야 합니다. 이름은 인수를 자리 표시자에 맞추는 데 사용되지 않습니다.

이 방법을 사용하면 로깅 공급자가 의미 체계 또는 구조적 로깅을 구현할 수 있습니다. 인수 자체는 서식이 지정된 메시지 템플릿뿐만 아니라 로깅 시스템에 전달됩니다. 따라서 로깅 공급자는 매개 변수 값을 필드로 저장할 수 있습니다. 다음 로거 메서드를 살펴보세요.

C#

```
_logger.LogInformation("Getting item {Id} at {RunTime}", id, DateTime.Now);
```

Azure Table Storage에 로그할 경우를 예로 듭니다.

- 각 Azure Table 엔터티에는 `ID` 및 `RunTime` 속성이 있을 수 있습니다.
- 속성이 있는 테이블은 로그된 데이터의 쿼리를 쉽게 합니다. 예를 들어 문자 메시지의 시간 초과를 구문 분석하지 않고도 특정 `RunTime` 범위 내에 있는 모든 로그를 쿼리를 통해 찾을 수 있습니다.

로그 메시지 템플릿 서식 지정

로그 메시지 템플릿은 자리 표시자 형식을 지원합니다. 템플릿은 지정된 형식 인수에 유효한 형식 을 지정할 수 있습니다. 예를 들어 다음 `Information` 로거 메시지 템플릿을 살펴보겠습니다.

C#

```
_logger.LogInformation("Logged on {PlaceHolderName:MMMM dd, yyyy}",
DateTimeOffset.UtcNow);
// Logged on January 06, 2022
```

앞의 예제에서 `DateTimeOffset` 인스턴스는 로거 메시지 템플릿의 `PlaceHolderName`에 해당하는 형식입니다. 이 이름은 값이 서수 기반이므로 무엇이든 될 수 있습니다. `MMMM dd, yyyy` 서식은 `DateTimeOffset` 형식에 유효합니다.

`DateTime` 및 `DateTimeOffset` 서식에 대한 자세한 내용은 사용자 지정 날짜 및 시간 서식 문자열 을 참조하세요.

예시

다음 예제에서는 `{}` 자리 표시자 구문을 사용하여 메시지 템플릿의 서식을 지정하는 방법을 보여 줍니다. 또한 `{}` 자리 표시자 구문을 이스케이프하는 예제가 출력과 함께 표시됩니다. 마지막으로 템플릿 자리 표시자를 사용하여 문자열 보간의 예시도 보여집니다.

C#

```
logger.LogInformation("Number: {Number}", 1);              // Number: 1
logger.LogInformation("{{Number}}: {Number}", 3);          // {Number}: 3
logger.LogInformation($"{{{{Number}}}}: {{Number}}", 5);   // {Number}: 5
```

> **팁**

- 대부분의 경우 로깅할 때 로그 메시지 템플릿 서식을 사용해야 합니다. 문자열 보간을 사용하면 성능 문제가 발생할 수 있습니다.
- 코드 분석 규칙 **CA2254: 템플릿은 정적 표현식이어야 함**은 로그 메시지가 적절한 서식을 사용하지 않는 위치에 대해 경고하는 데 도움이 됩니다.

예외 기록

로거 메서드에는 예외 매개 변수를 사용하는 오버로드가 있습니다.

C#

```
public void Test(string id)
{
    try
    {
        if (id is "none")
        {
            throw new Exception("Default Id detected.");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(
            AppLogEvents.Error, ex,
            "Failed to process iteration: {Id}", id);
    }
}
```

예외 로깅은 공급자별로 다릅니다.

기본 로그 수준

기본 로그 수준이 설정되지 않은 경우 기본 로그 수준 값은 .입니다 `Information`.

예를 들어 다음 작업자 서비스 앱을 살펴보세요.

- .NET 작업자 템플릿을 사용하여 만들었습니다.
- *appsettings.json* 및 *appsettings.Development.json*을 삭제하거나 이름을 변경했습니다.

이전 설정을 사용하여 개인정보처리방침이나 홈페이지로 이동하면 범주 이름에 `Trace`가 포함된 `Debug`, `Information` 및 `Microsoft` 메시지가 많이 생성됩니다.

다음 코드는 기본 로그 수준이 구성에서 설정되지 않은 경우 기본 로그 수준을 설정합니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.SetMinimumLevel(LogLevel.Warning);

using IHost host = builder.Build();

await host.RunAsync();
```

필터 함수

필터 함수는 구성 또는 코드를 통해 규칙이 할당되지 않은 모든 공급자와 범주에 대해 호출됩니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddFilter((provider, category, logLevel) =>
{
    return provider.Contains("ConsoleLoggerProvider")
        && (category.Contains("Example") || category.Contains("Microsoft"))
        && logLevel >= LogLevel.Information;
});

using IHost host = builder.Build();

await host.RunAsync();
```

앞의 코드는 범주에 `Example` 또는 `Microsoft`가 포함되어 있고 로그 수준이 `Information` 이상인 경우 콘솔 로그를 표시합니다.

로그 범위

*범위*는 논리 작업 집합을 그룹화합니다. 이 그룹화는 집합의 일부로 만든 각 로그에 동일한 데이터를 연결할 수 있습니다. 예를 들어 트랜잭션 처리의 일부로 생성되는 모든 로그에는 트랜잭션 ID가 포함될 수 있습니다.

범위:

- IDisposable 메서드에서 반환하는 BeginScope 형식입니다.
- 삭제될 때까지 유지됩니다.

범위를 지원하는 공급자는 다음과 같습니다.

- `Console`
- AzureAppServicesFile 및 AzureAppServicesBlob
- ApplicationInsightsLoggerProvider

`using` 블록에 로거 호출을 래핑하여 범위를 사용합니다.

C#

```
public async Task<T> GetAsync<T>(string id)
{
    T result;
    var transactionId = Guid.NewGuid().ToString();

    using (_logger.BeginScope(new List<KeyValuePair<string, object>>
        {
            new KeyValuePair<string, object>("TransactionId", transactionId),
        }))
    {
        _logger.LogInformation(
            AppLogEvents.Read, "Reading value for {Id}", id);

        var result = await _repository.GetAsync(id);
        if (result is null)
        {
            _logger.LogWarning(
                AppLogEvents.ReadNotFound, "GetAsync({Id}) not found", id);
        }
    }

    return result;
}
```

다음 JSON은 콘솔 공급자를 위해 스코프를 활성화합니다.

JSON

```
{
    "Logging": {
        "Debug": {
            "LogLevel": {
                "Default": "Information"
            }
        },
        "Console": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft": "Warning",
                "Default": "Information"
            }
        },
        "LogLevel": {
```

```
            "Default": "Debug"
        }
    }
}
```

다음은 콘솔 공급자에 대한 범위를 활성화하는 코드입니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.ClearProviders();
builder.Logging.AddSimpleConsole(options => options.IncludeScopes = true);

using IHost host = builder.Build();

await host.RunAsync();
```

Main에서 로그 만들기

다음 코드는 호스트를 빌드한 후 DI에서 `Main` 인스턴스를 가져와 `ILogger`에 로그인합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

using IHost host = Host.CreateApplicationBuilder(args).Build();

var logger = host.Services.GetRequiredService<ILogger<Program>>();
logger.LogInformation("Host created.");

await host.RunAsync();
```

위의 코드는 두 개의 NuGet 패키지를 사용합니다.

- Microsoft.Extensions.Hosting
- Microsoft.Extensions.Logging

프로젝트 파일은 다음과 유사합니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
```

```
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="7.0.1" />
    <PackageReference Include="Microsoft.Extensions.Logging" Version="7.0.0" />
  </ItemGroup>

</Project>
```

비동기 로거 메서드 미지원

로깅은 매우 빨라서 비동기 코드의 성능 비용을 들일 필요가 없습니다. 로깅 데이터 저장소가 느린 경우 직접 로깅 데이터 저장소에 작성하지 마세요. 로그 메시지를 처음에 빠른 저장소에 작성한 다음, 나중에 느린 저장소로 이동하는 것이 좋습니다. 예를 들어 SQL Server에 로그하는 경우 `Log` 메서드는 동기식이므로 `Log` 메서드에서 직접 로그하지 마세요. 대신 동기적으로 로그 메시지를 메모리 내 큐에 추가하고 백그라운드 작업자가 큐에서 메시지를 풀하여 SQL Server에 대해 비동기 데이터 푸시 작업을 수행하도록 합니다.

실행 중인 앱에서 로그 수준 변경

로깅 API는 앱이 실행되는 동안 로그 수준을 변경하는 시나리오를 포함하지 않습니다. 그러나 일부 구성 공급자는 로깅 구성에 즉시 적용되는 구성을 다시 로드할 수 있습니다. 예를 들어 파일 구성 공급자는 기본적으로 로깅 구성을 다시 로드합니다. 앱이 실행되는 동안 코드에서 구성을 변경하는 경우 앱은 IConfigurationRoot.Reload 를 호출하여 앱의 로깅 구성을 업데이트할 수 있습니다.

NuGet 패키지

ILogger<TCategoryName> 및 ILoggerFactory 인터페이스와 구현은 대부분의 .NET SDK에 암시적 패키지 참조로 포함되어 있습니다. 암시적으로 참조되지 않는 경우 다음 NuGet 패키지에서 명시적으로 사용할 수도 있습니다.

- 인터페이스는 Microsoft.Extensions.Logging.Abstractions 에 있습니다.
- 기본 구현은 Microsoft.Extensions.Logging 에 있습니다.

암시적 패키지 참조를 포함하는 .NET SDK에 대한 자세한 내용은 .NET SDK: 암시적 네임스페이스에 대한 테이블을 참조하세요.

참고하십시오

- .NET의 로깅 공급자
- .NET에서 사용자 지정 로깅 공급자 구현
- 콘솔 로그 서식 지정
- .NET의 고성능 로깅
- .NET 라이브러리 작성자를 위한 로깅 지침
- 로깅 버그는 github.com/dotnet/runtime 리포지토리에 생성해야 함

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 05.

.NET 환경의 로깅 제공자

로깅 공급자는 로그를 표준 출력으로만 표시하는 `Console` 공급자를 제외하고 로그를 유지합니다. 예를 들어 Azure Application Insights 공급자는 Azure Application Insights에 로그를 저장합니다. 여러 공급자를 사용하도록 설정할 수 있습니다.

기본 .NET 작업자 앱 템플릿:

- 제네릭 호스트사용합니다.
- 다음의 로깅 공급자를 추가하는 CreateApplicationBuilder을 호출하십시오.
  - 콘솔
  - 디버그
  - EventSource
  - EventLog (Windows에만 해당)

C#

```
using Microsoft.Extensions.Hosting;

using IHost host = Host.CreateApplicationBuilder(args).Build();

// Application code should start here.

await host.RunAsync();
```

앞의 코드는 .NET Worker 앱 템플릿으로 만든 `Program` 클래스를 보여 줍니다. 다음 몇 가지 섹션에서는 제네릭 호스트를 사용하는 .NET Worker 앱 템플릿을 기반으로 샘플을 제공합니다.

`Host.CreateApplicationBuilder`추가한 기본 로깅 공급자 집합을 재정의하려면 `ClearProviders` 호출하고 원하는 로깅 공급자를 추가합니다. 예를 들어 다음 코드는 다음과 같습니다.

- ClearProviders 호출하여 작성기에서 모든 ILoggerProvider 인스턴스를 제거합니다.
- 콘솔 로깅 공급자를 추가합니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.ClearProviders();
builder.Logging.AddConsole();
```

다른 공급자는 다음을 참조하세요.

- 기본 제공 로깅 공급자.
- 타사 로깅 공급자.

ILogger에 의존하는 서비스 구성

`ILogger<T>`종속된 서비스를 구성하려면 생성자 주입을 사용하거나 팩터리 메서드를 제공합니다. 팩터리 메서드 방법은 다른 옵션이 없는 경우에만 권장됩니다. 예를 들어 DI에서 제공하는 `ILogger<T>` 인스턴스가 필요한 서비스를 고려합니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddSingleton<IExampleService>(
    container => new DefaultExampleService
    {
        Logger = container.GetRequiredService<ILogger<IExampleService>>()
    });
```

앞의 코드는 DI 컨테이너가 <인스턴스를 처음으로 생성해야 할 때 실행되는 >. 이러한 방식으로 등록된 서비스에 액세스할 수 있습니다.

기본 제공 로깅 공급자

Microsoft 확장에는 런타임 라이브러리의 일부로 다음 로깅 공급자가 포함됩니다.

- 콘솔
- 디버그
- EventSource
- EventLog

다음 로깅 공급자는 Microsoft에서 제공하지만 런타임 라이브러리의 일부로는 제공하지 않습니다. NuGet 패키지에서 설치해야 합니다.

- AzureAppServicesFile 및 AzureAppServicesBlob
- ApplicationInsights

Console

`Console` 공급자는 콘솔에 출력을 기록합니다.

Debug

`Debug` 공급자는 특히 System.Diagnostics.Debug 메서드를 통해 디버거가 연결된 경우에만 Debug.WriteLine 클래스를 사용하여 로그 출력을 작성합니다. DebugLoggerProvider `ILogger` 인터페이스를 구현하는 로거 클래스의 인스턴스를 만듭니다.

이벤트 원본

`EventSource` 공급자는 `Microsoft-Extensions-Logging`이라는 이름의 플랫폼 간 이벤트 원본에 기록합니다. Windows에서 공급자는 ETW사용합니다.

dotnet 추적 도구

dotnet-trace 도구는 실행 중인 프로세스의 .NET Core 추적을 수집할 수 있도록 하는 크로스 플랫폼 명령줄 인터페이스(CLI) 전역 도구입니다. 이 도구는 Microsoft.Extensions.Logging.EventSource을 사용하여 LoggingEventSource 공급자 데이터를 수집합니다.

설치 지침은 dotnet-trace을(를) 참조하세요. `dotnet-trace`사용하는 진단 자습서는 .NET Core 높은 CPU 사용량 디버그참조하세요.

Windows 이벤트 로그

`EventLog` 공급자는 Windows 이벤트 로그에 로그 출력을 보냅니다. 다른 공급자와 달리 `EventLog` 공급자는 기본 비 공급자 설정을 상속하지 않습니다 . `EventLog` 로그 설정을 지정하지 않으면 기본값으로 `LogLevel.Warning`로 설정됩니다.

LogLevel.Warning보다 낮은 이벤트를 기록하려면 로그 수준을 명시적으로 설정합니다. 다음 예제에서는 이벤트 로그 기본 로그 수준을 LogLevel.Information설정합니다.

JSON

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

AddEventLog 오버로드EventLogSettings전달할 수 있습니다. `null` 지정하지 않으면 다음 기본 설정이 사용됩니다.

- `LogName`: "애플리케이션"
- `SourceName`: ".NET 런타임"
- `MachineName`: 로컬 컴퓨터 이름이 사용됩니다.

다음 코드는 `SourceName`의 기본값인 `".NET Runtime"`을 `CustomLogs`로 변경합니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddEventLog(
    config => config.SourceName = "CustomLogs");

using IHost host = builder.Build();

host.Run();
```

Azure App Service

Microsoft.Extensions.Logging.AzureAppServices 공급자 패키지는 Azure App Service 앱의 파일 시스템의 텍스트 파일에 로그를 쓰고 Azure Storage 계정에서 blob Storage .

공급자 패키지는 런타임 라이브러리에 포함되지 않습니다. 공급자를 사용하려면 프로젝트에 공급자 패키지를 추가합니다.

공급자 설정을 구성하려면 다음 예제와 같이 AzureFileLoggerOptions 및 AzureBlobLoggerOptions사용합니다.

C#

```
using Microsoft.Extensions.Logging.AzureAppServices;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args)

builder.Logging.AddAzureWebAppDiagnostics();
builder.Services.Configure<AzureFileLoggerOptions>(options =>
{
    options.FileName = "azure-diagnostics-";
    options.FileSizeLimit = 50 * 1024;
    options.RetainedFileCountLimit = 5;
});
builder.Services.Configure<AzureBlobLoggerOptions>(options =>
{
    options.BlobName = "log.txt";
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

Azure App Service에 배포된 경우 앱은 Azure Portal의 App Service 페이지의 **App Service 로그** 섹션에 있는 설정을 사용합니다. 다음 설정이 업데이트되면 앱을 다시 시작하거나 다시 배포할 필요 없이 변경 내용이 즉시 적용됩니다.

로그 파일의 기본 위치는 *D:\home\LogFiles\Application* 폴더에 있습니다. 다른 기본값은 공급자에 따라 다릅니다.

- **애플리케이션 로깅(파일 시스템)**: 기본 파일 시스템 파일 이름은 *diagnostics-yyyymmdd.txt*. 기본 파일 크기 제한은 10MB이고, 유지되는 파일의 기본 최대 수는 2입니다.
- **애플리케이션 로깅(Blob)**: 기본 Blob 이름은 *{app-name}/yyyy/mm/dd/hh/{guid}_applicationLog.txt*.

이 공급자는 프로젝트가 Azure 환경에서 실행되는 경우에만 로그합니다.

Azure 로그 스트리밍

Azure 로그 스트리밍은 다음에서 로그 활동을 실시간으로 볼 수 있도록 지원합니다.

- 앱 서버
- 웹 서버
- 실패한 요청 추적

Azure 로그 스트리밍을 구성하려면 다음을 수행합니다.

- 앱 포털 페이지에서 **App Service 로그** 페이지로 이동하세요.
- **애플리케이션 로깅(파일 시스템)**을 **로**에 설정합니다.
- 로그 **수준**을 선택하세요. 이 설정은 Azure 로그 스트리밍에만 적용됩니다.

로그 스트림 페이지로 이동하여 로그를 봅니다. 기록된 메시지는 `ILogger` 인터페이스를 사용하여 기록됩니다.

Azure Application Insights

Microsoft.Extensions.Logging.ApplicationInsights 공급자 패키지는 로그를 Azure Application Insights에 씁니다. Application Insights는 웹앱을 모니터링하고 원격 분석 데이터를 쿼리하고 분석하기 위한 도구를 제공하는 서비스입니다. 이 공급자를 사용하는 경우 Application Insights 도구를 사용하여 로그를 쿼리하고 분석할 수 있습니다.

자세한 내용은 다음 리소스를 참조하세요.

- Application Insights 개요
- .NET Core ILogger 로그 ApplicationInsightsLoggerProvider - 나머지 Application Insights 원격 분석 없이 로깅 공급자를 구현하려는 경우 여기에서 시작합니다.
- Application Insights 로깅 어댑터.
- Application Insights에서 제공하는 도구를 사용하여 애플리케이션의 성능과 안정성을 향상시키는 방법을 알아봅니다.

로깅 공급자 디자인 고려 사항

ILoggerProvider 인터페이스의 고유한 구현과 해당 사용자 지정 ILogger구현을 개발하려는 경우 다음 사항을 고려하세요.

- ILogger.Log 메서드는 동기적입니다.
- 로그 상태 및 개체의 수명은 가정해서는 안 됩니다.

구현은 `ILoggerProvider`의 `ILogger` 메서드를 통해 ILoggerProvider.CreateLogger를 생성합니다. 구현에서 비차단 방식으로 로깅 메시지를 큐에 대기하려는 경우, 먼저 메시지를 구체화하거나 로그 항목을 구체화하는 데 사용되는 개체 상태를 직렬화해야 합니다. 이로 인해 해제된 개체로 인해 발생할 수 있는 잠재적인 예외가 방지됩니다.

자세한 내용은 .NET사용자 지정 로깅 공급자 구현을 참조하세요.

타사 로깅 공급자

다음은 다양한 .NET 워크로드에서 작동하는 몇 가지 타사 로깅 프레임워크입니다.

- elmah.io (GitHub 리포지토리)
- EFLogger (GitHub 리포지토리)
- Gelf (GitHub 리포지토리)
- JSNLog (GitHub 저장소)
- KissLog.net (GitHub 리포지토리)
- Log4Net (GitHub 리포지토리)
- NLog (GitHub 저장소)
- NReco.Logging (GitHub 리포지토리)
- Sentry (GitHub 리포지토리)
- Serilog (GitHub 리포지토리)
- Stackdriver (GitHub 리포지토리)

일부 타사 프레임워크는 의미 체계 로깅(구조적 로깅이라고도 함)을 수행할 수 있습니다.

타사 프레임워크 사용은 기본 제공 공급자 중 하나를 사용하는 것과 유사합니다.

1. 프로젝트에 NuGet 패키지를 추가합니다.
2. 로깅 프레임워크에서 제공하는 `ILoggerFactory` 또는 `ILoggingBuilder` 확장 메서드를 호출합니다.

자세한 내용은 각 공급자의 설명서를 참조하세요. 타사 로깅 공급자는 Microsoft에서 지원되지 않습니다.

참고하십시오

- .NET에서의 로깅.
- .NET사용자 지정 로깅 공급자를 구현합니다.
- .NET에서의 고성능 로깅 .

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 05.

컴파일 타임 로그 소스 생성하기

원본 생성 로깅은 최신 .NET 앱에 매우 사용 가능하고 성능이 뛰어난 로깅 솔루션을 제공하도록 설계되었습니다. 자동 생성된 소스 코드는 ILogger 기능과 함께 LoggerMessage.Define 인터페이스를 사용합니다.

이 소스 생성기는 LoggerMessageAttribute가 `partial` 로깅 메서드에서 사용될 때 트리거됩니다. 트리거되면 데코레이팅하는 메서드의 구현이 `partial` 자동으로 생성됩니다. 문제가 있는 경우 적절한 사용에 대한 힌트를 사용하여 컴파일 시간 진단을 생성합니다. 이 컴파일 시간 로깅 솔루션은 이전에 사용 가능한 로깅 방법보다 런타임에 훨씬 더 빠릅니다. boxing과 임시 할당, 복사를 가능한 최대한으로 제거합니다.

기본 사용법

`LoggerMessageAttribute`를 사용하려면 사용하는 클래스와 메서드가 `partial`이어야 합니다. 코드 생성기는 컴파일 시간에 트리거되고 `partial` 메서드의 구현을 생성합니다.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(
        ILogger logger, string hostName);
}
```

위의 예제에서 로깅 메서드는 `static`이고 로그 수준은 특성 정의에 지정됩니다. 정적 컨텍스트에서 사용하는 `LoggerMessageAttribute` 경우 인스턴스를 `ILogger` 인수로 전달해야 합니다. 또는 매개 변수에 `this` 한정자를 `ILogger` 추가하여 메서드를 확장 메서드로 정의합니다.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(
        this ILogger logger, string hostName);
}
```

비정적 컨텍스트에서도 특성을 사용하도록 선택할 수 있습니다. 로깅 메서드가 인스턴스 메서드로 선언된 다음 예제를 살펴보세요. 이 컨텍스트에서 로깅 메서드는 포함하는 클래스의 `ILogger` 필드에 액세스하여 로거를 가져옵니다.

C#

```
public partial class InstanceLoggingExample
{
    private readonly ILogger _logger;

    public InstanceLoggingExample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public partial void CouldNotOpenSocket(string hostName);
}
```

.NET 9부터 로깅 메서드는 포함하는 클래스의 `ILogger` 기본 생성자 매개 변수에서 로거를 추가로 가져올 수 있습니다.

C#

```
public partial class InstanceLoggingExample(ILogger logger)
{
    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public partial void CouldNotOpenSocket(string hostName);
}
```

필드와 기본 생성자 매개 변수가 둘 다 `ILogger` 있는 경우 로깅 메서드는 필드에서 로거를 가져옵니다.

동적 로그 수준

로그 수준은 코드에 정적으로 빌드되지 않고 동적이어야 하는 경우도 있습니다. 특성에서 로그 수준을 생략하고 대신 로깅 메서드에 대한 매개 변수로 요구하여 이렇게 할 수 있습니다.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(
        ILogger logger,
        LogLevel level, /* Dynamic log level as parameter, rather than defined in
attribute. */
        string hostName);
}
```

`Message` 속성

`LoggerMessageAttribute`의 속성 Message은 선택 사항입니다. 생략할 경우, String.Empty가 메시지에 사용됩니다. 그러나 로깅 메서드에 해당 템플릿 자리 표시자가 없는 매개 변수가 있는 경우 컴파일러는 SYSLIB1015 경고를 내보냅니다. 이러한 매개 변수는 로그 상태에 저장되지만 형식이 지정된 로그 출력에는 표시되지 않습니다. 로그 상태를 표면화하는 구조적 로깅 공급자만 해당 로그 상태를 노출합니다.

로그 메서드 제약 조건

데코레이팅된 `LoggerMessageAttribute` 로깅 메서드는 다음 요구 사항을 충족해야 합니다.

- 로깅 메서드는 `partial`이어야 하며 `void`(을)를 반환해야 합니다.
- 로깅 메서드 이름은 밑줄로 시작하지 '않아야' 합니다.
- 로깅 메서드의 매개 변수 이름은 밑줄로 시작하지 '않아야' 합니다.
- 로깅 메서드는 제네릭 형식 매개 변수를 지원하지만 C# 13 `allows ref struct` 제약 조건은 지원되지 않습니다.
- 로깅 메서드 매개 변수는 , `params`또는 `scoped` 한정자를 사용할 `out`수 없으며 형식일 `ref struct` 수 없습니다.
- 로깅 메서드가 `static`인 경우 `ILogger` 인스턴스가 매개 변수로 필요합니다.

코드 생성 모델은 최신 C# 컴파일러, 즉 버전 9 이상으로 컴파일되는 코드에 따라 달라집니다. 언어 버전 변경에 대한 자세한 내용은 C# 언어 버전 관리를 참조하세요.

로그 메서드 구조

서명은 ILogger.Log를 수락하고 필요에 따라 LogLevel 및 선택적으로 Exception를 수락합니다. 다음 코드 예제를 참조하십시오.

```
C#
```

```
public interface ILogger
{
    void Log<TState>(
        Microsoft.Extensions.Logging.LogLevel logLevel,
        Microsoft.Extensions.Logging.EventId eventId,
        TState state,
        System.Exception? exception,
        Func<TState, System.Exception?, string> formatter);
}
```

일반적으로 `ILogger`, `LogLevel`, `Exception`의 첫 번째 인스턴스는 특별히 소스 생성기의 로그 메서드 시그니처에서 처리됩니다. 후속 인스턴스는 메시지 템플릿에 대한 일반 매개 변수처럼 처리됩니다.

C#

```
// This is a valid attribute usage
[LoggerMessage(
    EventId = 110, Level = LogLevel.Debug, Message = "M1 {Ex3} {Ex2}")]
public static partial void ValidLogMethod(
    ILogger logger,
    Exception ex,
    Exception ex2,
    Exception ex3);

// This causes a warning
[LoggerMessage(
    EventId = 0, Level = LogLevel.Debug, Message = "M1 {Ex} {Ex2}")]
public static partial void WarningLogMethod(
    ILogger logger,
    Exception ex,
    Exception ex2);
```

> **ⓘ Important**
>
> 내보낸 경고는 `LoggerMessageAttribute`의 올바른 사용법에 관한 세부 정보를 제공합니다. 앞의 예제에서 `WarningLogMethod`은 `DiagnosticSeverity.Warning`에 대한 `SYSLIB0025`을 보고합니다.

콘솔

```
Don't include a template for `ex` in the logging message since it is implicitly
taken care of.
```

대/소문자를 구분하지 않는 템플릿 이름 지원

생성기는 메시지 템플릿의 항목과 로그 메시지의 인수 이름 간에 대/소문자를 구분하지 않는 비교를 수행합니다. 즉, `ILogger`(이)가 상태를 열거할 때 메시지 템플릿에서 인수를 선택하므로 로그를 더 쉽게 사용할 수 있습니다.

C#

```
public partial class LoggingExample
{
    private readonly ILogger _logger;

    public LoggingExample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 10,
        Level = LogLevel.Information,
        Message = "Welcome to {City} {Province}!")]
    public partial void LogMethodSupportsPascalCasingOfNames(
        string city, string province);

    public void TestLogging()
    {
        LogMethodSupportsPascalCasingOfNames("Vancouver", "BC");
    }
}
```

`JsonConsole` 포맷터를 사용할 경우 예제 로깅 출력을 살펴보세요.

JSON

```
{
  "EventId": 13,
  "LogLevel": "Information",
  "Category": "LoggingExample",
  "Message": "Welcome to Vancouver BC!",
  "State": {
    "Message": "Welcome to Vancouver BC!",
    "City": "Vancouver",
    "Province": "BC",
    "{OriginalFormat}": "Welcome to {City} {Province}!"
  }
}
```

확정되지 않은 매개 변수 순서

로그 메서드 매개 변수의 순서 지정에는 제약 조건이 없습니다. 개발자는 `ILogger` 약간 어색해 보일 수 있지만 마지막 매개 변수로 정의할 수 있습니다.

C#

```
[LoggerMessage(
    EventId = 110,
    Level = LogLevel.Debug,
    Message = "M1 {Ex3} {Ex2}")]
static partial void LogMethod(
    Exception ex,
    Exception ex2,
    Exception ex3,
    ILogger logger);
```

팁

로그 메서드의 매개 변수 순서는 템플릿 자리 표시자의 순서에 해당할 필요가 없습니다. 대신 템플릿의 자리 표시자 이름은 매개 변수와 일치해야 합니다. 다음 `JsonConsole` 출력과 오류 순서를 살펴보세요.

JSON

```
{
  "EventId": 110,
  "LogLevel": "Debug",
  "Category": "ConsoleApp.Program",
  "Message": "M1 System.Exception: Third time's the charm. System.Exception:
This is the second error.",
  "State": {
    "Message": "M1 System.Exception: Third time's the charm. System.Exception:
This is the second error.",
    "ex2": "System.Exception: This is the second error.",
    "ex3": "System.Exception: Third time's the charm.",
    "{OriginalFormat}": "M1 {Ex3} {Ex2}"
  }
}
```

추가 로깅 예제

다음 샘플에서는 이벤트 이름을 검색하고, 로그 수준을 동적으로 설정하고, 로깅 매개 변수를 포맷하는 방법을 보여 줍니다. 로깅 메서드는 다음과 같습니다.

- `LogWithCustomEventName`: `LoggerMessage` 특성을 통해 이벤트 이름을 검색합니다.
- `LogWithDynamicLogLevel`: 로그 수준을 동적으로 설정하여 구성 입력에 따라 로그 수준을 설정할 수 있도록 합니다.
- `UsingFormatSpecifier`: 형식 지정자를 사용하여 로깅 매개 변수의 형식을 지정합니다.

C#

```
public partial class LoggingSample
{
    private readonly ILogger _logger;

    public LoggingSample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 20,
        Level = LogLevel.Critical,
        Message = "Value is {Value:E}")]
    public static partial void UsingFormatSpecifier(
        ILogger logger, double value);

    [LoggerMessage(
        EventId = 9,
        Level = LogLevel.Trace,
        Message = "Fixed message",
        EventName = "CustomEventName")]
    public partial void LogWithCustomEventName();

    [LoggerMessage(
        EventId = 10,
        Message = "Welcome to {City} {Province}!")]
    public partial void LogWithDynamicLogLevel(
        string city, LogLevel level, string province);

    public void TestLogging()
    {
        LogWithCustomEventName();

        LogWithDynamicLogLevel("Vancouver", LogLevel.Warning, "BC");
        LogWithDynamicLogLevel("Vancouver", LogLevel.Information, "BC");

        UsingFormatSpecifier(logger, 12345.6789);
    }
}
```

`SimpleConsole` 포맷터를 사용할 경우 예제 로깅 출력을 살펴보세요.

콘솔

```
trce: LoggingExample[9]
      Fixed message
warn: LoggingExample[10]
      Welcome to Vancouver BC!
info: LoggingExample[10]
      Welcome to Vancouver BC!
```

```
crit: LoggingExample[20]
      Value is 1.234568E+004
```

`JsonConsole` 포맷터를 사용할 경우 예제 로깅 출력을 살펴보세요.

JSON


```
{
  "EventId": 9,
  "LogLevel": "Trace",
  "Category": "LoggingExample",
  "Message": "Fixed message",
  "State": {
    "Message": "Fixed message",
    "{OriginalFormat}": "Fixed message"
  }
}
{
  "EventId": 10,
  "LogLevel": "Warning",
  "Category": "LoggingExample",
  "Message": "Welcome to Vancouver BC!",
  "State": {
    "Message": "Welcome to Vancouver BC!",
    "city": "Vancouver",
    "province": "BC",
    "{OriginalFormat}": "Welcome to {City} {Province}!"
  }
}
{
  "EventId": 10,
  "LogLevel": "Information",
  "Category": "LoggingExample",
  "Message": "Welcome to Vancouver BC!",
  "State": {
    "Message": "Welcome to Vancouver BC!",
    "city": "Vancouver",
    "province": "BC",
    "{OriginalFormat}": "Welcome to {City} {Province}!"
  }
}
{
  "EventId": 20,
  "LogLevel": "Critical",
  "Category": "LoggingExample",
  "Message": "Value is 1.234568E+004",
  "State": {
    "Message": "Value is 1.234568E+004",
    "value": 12345.6789,
    "{OriginalFormat}": "Value is {Value:E}"
  }
}
```

로그에서 중요한 정보 수정

중요한 데이터를 로깅할 때는 우발적인 노출을 방지하는 것이 중요합니다. 컴파일 시간 생성 로깅 메서드가 있더라도 원시 중요한 값을 로깅하면 데이터 누수 및 규정 준수 문제가 발생할 수 있습니다.

Microsoft.Extensions.Telemetry 라이브러리는 .NET 애플리케이션에 대한 고급 로깅 및 원격 분석 보강 기능을 제공합니다. 로그를 작성할 때 분류된 데이터에 자동으로 수정을 적용하도록 로깅 파이프라인을 확장합니다. 이를 통해 편집을 로깅 워크플로에 통합하여 애플리케이션 전체에서 데이터 보호 정책을 적용할 수 있습니다. 정교한 원격 분석 및 로깅 인사이트가 필요한 애플리케이션을 위해 빌드되었습니다.

수정을 사용하도록 설정하려면 Microsoft.Extensions.Compliance.Redaction 라이브러리를 사용합니다. 이 라이브러리는 출력에 안전하도록 중요한 데이터(예: 지우기, 마스킹 또는 해시)를 변환하는 구성 요소인 **편집기를** 제공합니다. 데이터 **분류**에 따라 편집기가 선택되어 민감도(예: 개인, 개인 또는 공용)에 따라 데이터에 레이블을 지정할 수 있습니다.

소스 생성 로깅 메서드에서 편집을 사용하려면 다음을 수행해야 합니다.

1. 데이터 분류 시스템을 사용하여 중요한 데이터를 분류합니다.
2. DI 컨테이너의 각 분류에 대한 편집기를 등록하고 구성합니다.
3. 로깅 파이프라인에서 편집 기능을 활성화합니다.
4. 중요한 데이터가 노출되지 않도록 로그를 확인합니다.

예를 들어 프라이빗으로 간주되는 매개 변수가 있는 로그 메시지가 있는 경우:

C#

```
[LoggerMessage(0, LogLevel.Information, "User SSN: {SSN}")]
public static partial void LogPrivateInformation(
    this ILogger logger,
    [MyTaxonomyClassifications.Private] string SSN);
```

다음과 유사한 설정이 있어야 합니다.

C#

```
using Microsoft.Extensions.Telemetry;
using Microsoft.Extensions.Compliance.Redaction;

var services = new ServiceCollection();
services.AddLogging(builder =>
{
    // Enable redaction.
    builder.EnableRedaction();
});
```

```
services.AddRedaction(builder =>
{
    // configure redactors for your data classifications
    builder.SetRedactor<StarRedactor>(MyTaxonomyClassifications.Private);
});

public void TestLogging()
{
    LogPrivateInformation("MySSN");
}
```

출력은 다음과 같아야 합니다.

```
User SSN: *****
```

이 방법을 사용하면 컴파일 시간 생성된 로깅 API를 사용하는 경우에도 수정된 데이터만 기록됩니다. 다양한 데이터 형식 또는 분류에 다른 편집기를 사용하고 편집 논리를 중앙에서 업데이트할 수 있습니다.

데이터를 분류하는 방법에 대한 자세한 내용은 .NET 참조하세요. 편집 및 편집자에 대한 자세한 내용은 .NET에서 데이터 편집을 참조하세요.

요약

C# 원본 생성기의 출현으로 성능이 뛰어난 로깅 API를 작성하는 것이 더 쉽습니다. 소스 생성기 방법을 사용하는 경우 다음과 같은 몇 가지 주요 이점이 있습니다.

- 로깅 구조를 보존하고 메시지 템플릿에 필요한 정확한 형식 구문을 사용하도록 설정할 수 있습니다.
- 템플릿 자리 표시자에 대한 대체 이름을 제공하고 형식 지정자를 사용할 수 있습니다.
- `string`을(를) 만드는 것 외에 다른 작업을 수행하기 전에 저장 방법에 대한 복잡성 없이 모든 원본 데이터를 있는 그대로 전달할 수 있습니다.
- 로깅 관련 진단을 제공하고 중복 이벤트 ID에 대한 경고를 내보낸다.

또한 LoggerMessage.Define을 수동으로 사용하는 것에 비해 다음과 같은 이점이 있습니다.

- 더 짧고 간단한 구문: 상용구 코딩 대신 선언적 특성 사용
- 단계별 개발자 환경: 이 생성기는 개발자가 올바른 작업을 수행하는 데 도움이 되도록 경고를 제공합니다.
- 임의 개수의 로깅 매개 변수 지원: `LoggerMessage.Define`은 최대 6개를 지원합니다.
- 동적 로그 수준 지원: `LoggerMessage.Define` 만으로는 이 작업을 수행할 수 없습니다.

참고하십시오

- .NET에서의 로깅
- .NET의 고성능 로그 기록
- 콘솔 로그 서식 지정
- .NET에서의 데이터 수정
- .NET의 데이터 분류
- NuGet(누겟): Microsoft.Extensions.Logging.Abstractions

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 18.

.NET 라이브러리 작성자용 로깅 지침

라이브러리 작성자로서 로깅 노출은 소비자에게 라이브러리의 내부 작동에 대한 인사이트를 제공하는 좋은 방법입니다. 이 지침은 다른 .NET 라이브러리 및 프레임워크와 일치하는 방식으로 로깅을 노출하는 데 도움이 됩니다. 또한 일반적인 성능 병목 상태를 방지하는 데 도움이 됩니다.

인터페이스를 사용하는 경우 `ILoggerFactory`

로그를 내보내는 라이브러리를 작성할 때 로그를 ILogger 기록하려면 개체가 필요합니다. 해당 개체를 가져오기 위해 API는 ILogger<TCategoryName> 매개 변수를 수락하거나, ILoggerFactory를 호출한 후 ILoggerFactory.CreateLogger를 수락할 수 있습니다. 어떤 방법을 선호해야 하나요?

- 모든 클래스가 로그를 내보낼 수 있도록 여러 클래스에 전달할 수 있는 로깅 개체가 필요한 경우 사용합니다 `ILoggerFactory`. 각 클래스는 클래스와 동일한 이름을 가진 별도의 범주를 사용하여 로그를 만드는 것이 좋습니다. 이렇게 하려면, 로그를 내보내는 각 클래스에 대해 고유한 `ILogger<TCategoryName>` 개체를 생성하기 위해 팩터리가 필요합니다. 일반적인 예로는 라이브러리에 대한 공용 진입점 API 또는 내부적으로 도우미 클래스를 만들 수 있는 형식의 공용 생성자가 포함됩니다.
- 한 클래스에서만 사용되고 절대 공유되지 않는 로깅 개체가 필요한 경우 로그를 생성하는 형식인 `` `ILogger<TCategoryName>` ``을 사용하세요, 여기서 `` `TCategoryName` ``은(는) 로그를 생성하는 형식입니다. 이 예제의 일반적인 예는 종속성 주입으로 만든 클래스에 대한 생성자입니다.

시간이 지남에 따라 안정적으로 유지해야 하는 공용 API를 디자인하는 경우 나중에 내부 구현을 리팩터링할 수 있습니다. 클래스가 처음에 내부 도우미 형식을 만들지 않더라도 코드가 진화함에 따라 변경될 수 있습니다. 공용 `ILoggerFactory` API를 변경하지 않고 새 클래스에 대한 새 `ILogger<TCategoryName>` 개체를 만들 수 있습니다.

자세한 내용은 필터링 규칙이 적용되는 방법을 참조하세요.

소스 코드 생성 로깅 선호

API는 `ILogger` API를 사용하는 두 가지 방법을 지원합니다. 예를 들어 LoggerExtensions.LogError 메서드와 LoggerExtensions.LogInformation 메서드를 호출하거나 로깅 원본 생성기를 사용하여 강력한 형식의 로깅 메서드를 정의할 수 있습니다. 대부분의 경우 원본 생성기는 뛰어난 성능과 더 강력한 입력을 제공하므로 권장됩니다. 또한 메시지 템플릿, ID 및 로그 수준과 같은 로깅 관련 문제를 호출 코드에서 격리합니다. 소스 생성되지 않은 접근 방식은 코드를 보다 간결하게 만들기 위해 이러한 이점을 포기하려는 시나리오에 주로 유용합니다.

C#

```
using Microsoft.Extensions.Logging;

namespace Logging.LibraryAuthors;

internal static partial class LogMessages
{
    [LoggerMessage(
        Message = "Sold {Quantity} of {Description}",
        Level = LogLevel.Information)]
    internal static partial void LogProductSaleDetails(
        this ILogger logger,
        int quantity,
        string description);
}
```

앞의 코드는 다음과 같습니다.

- `partial class` `LogMessages`로 명명되어 `static` 및 `ILogger` 형식에 대한 확장 메서드를 정의하는 데 사용할 수 있도록 정의합니다.
- `LogProductSaleDetails` 확장 메서드를 `LoggerMessage` 특성과 `Message` 템플릿으로 장식합니다.
- `LogProductSaleDetails`를 선언하며, 이는 `ILogger`를 확장하고 `quantity`와 `description`를 허용합니다.

> **팁**
>
> 디버깅하는 동안 소스 생성 코드는 호출하는 코드와 동일한 어셈블리의 일부이기 때문에 한 단계씩 실행할 수 있습니다.

비용이 많이 드는 매개 변수 평가를 방지하는 데 사용 `IsEnabled`

매개 변수를 평가하는 데 비용이 많이 드는 상황이 있을 수 있습니다. 이전 예제를 확장하면 매개 변수가 `description` `string` 계산에 비용이 많이 든다고 가정합니다. 판매 중인 제품은 친숙한 제품 설명을 받을 수 있으며, 데이터베이스 쿼리나 파일에서 읽는 것에 의존할 수 있습니다. 이러한 상황에서는 소스 생성기에 가드 `IsEnabled`를 건너뛰고 호출 사이트에서 가드 `IsEnabled`를 수동으로 추가하도록 지시할 수 있습니다. 이렇게 하면 사용자가 가드가 호출되는 위치를 확인할 수 있으며 컴퓨팅 비용이 많이 들 수 있는 매개 변수가 진정으로 필요한 경우에만 평가되도록 할 수 있습니다. 다음 코드를 생각해 봅시다.

C#

```
using Microsoft.Extensions.Logging;
```

```
namespace Logging.LibraryAuthors;

internal static partial class LogMessages
{
    [LoggerMessage(
        Message = "Sold {Quantity} of {Description}",
        Level = LogLevel.Information,
        SkipEnabledCheck = true)]
    internal static partial void LogProductSaleDetails(
        this ILogger logger,
        int quantity,
        string description);
}
```

확장 메서드 `LogProductSaleDetails`가 호출되면 `IsEnabled` 가드가 수동으로 호출되고, 비용이 많이 드는 매개변수 평가는 필요할 때에만 수행됩니다. 다음 코드를 생각해 봅시다.

C#

```
if (_logger.IsEnabled(LogLevel.Information))
{
    // Expensive parameter evaluation
    var description = product.GetFriendlyProductDescription();

    _logger.LogProductSaleDetails(
        quantity,
        description);
}
```

자세한 내용은 .NET의 컴파일 시간 로깅 원본 생성 및 고성능 로깅을 참조하세요.

로깅에서 문자열 보간을 사용하지 마십시오

일반적인 실수는 문자열 보간 을 사용하여 로그 메시지를 작성하는 것입니다. 로깅의 문자열 보간은 해당 `LogLevel` 문자열이 활성화되지 않은 경우에도 평가되므로 성능에 문제가 있습니다. 문자열 보간 대신 로그 메시지 템플릿, 서식 및 인수 목록을 사용합니다. 자세한 내용은 .NET: 로그 메시지 템플릿의 로깅을 참조하세요.

no-op 로깅 기본값 사용

로깅 API를 `ILogger` 또는 `ILoggerFactory` 형태로 제공하는 라이브러리를 사용할 때, 로거를 제공하지 않으려고 할 때가 있을 수 있습니다. 이러한 경우 Microsoft.Extensions.Logging.Abstractions NuGet 패키지는 no-op 로깅 기본값을 제공합니다.

라이브러리 소비자는 `ILoggerFactory`이(가) 제공되지 않는 경우 *null 로깅*을 기본적으로 사용할 수 있습니다. *null 로깅*의 사용은 형식이 null이 아니므로 형식을 nullable(`ILoggerFactory?`)로 정의하는 것과 다릅니다. 이러한 편의 기반 형식은 아무것도 기록하지 않으며 기본적으로 no-ops입니다. 해당하는 경우 사용 가능한 추상화 형식을 사용하는 것이 좋습니다.

- NullLogger.Instance
- Microsoft.Extensions.Logging.Abstractions.NullLogger<T>
- NullLoggerFactory.Instance
- NullLoggerProvider.Instance

Last updated on 2026. 02. 05.

.NET에서 사용자 지정 로깅 공급자 구현

일반적인 로깅 요구 사항에 사용할 수 많은 로깅 공급자가 있습니다. 그러나 사용 가능한 공급자 중 하나가 애플리케이션 요구 사항에 맞지 않는 경우 사용자 지정 ILoggerProvider 을 구현해야 할 수 있습니다. 이 문서에서는 콘솔에서 로그의 색을 지정하는 데 사용할 수 있는 사용자 지정 로깅 공급자를 구현하는 방법을 알아봅니다.

> **팁**
>
> 사용자 지정 로깅 공급자 예제 소스 코드는 **문서 GitHub 리포지토리** 에서 사용할 수 있습니다.

샘플 사용자 지정 로거 구성

샘플 로거는 다음 구성 유형을 사용하여 로그 수준 및 이벤트 ID당 다른 색 콘솔 항목을 만듭니다.

C#

```
using Microsoft.Extensions.Logging;

public sealed class ColorConsoleLoggerConfiguration
{
    public int EventId { get; set; }

    public Dictionary<LogLevel, ConsoleColor> LogLevelToColorMap { get; set; } =
new()
    {
        [LogLevel.Information] = ConsoleColor.Green
    };
}
```

위의 코드는 수준에 대한 기본 색을 `Information`로 `Green` 설정합니다. `EventId` 암시적으로 0입니다.

사용자 지정 로거 만들기

다음 코드 조각은 `ILogger` 구현을 보여줍니다.

C#

```
using Microsoft.Extensions.Logging;

public sealed class ColorConsoleLogger(
```

```
    string name,
    Func<ColorConsoleLoggerConfiguration> getCurrentConfig) : ILogger
{
    public IDisposable? BeginScope<TState>(TState state)
        where TState : notnull => default!;

    public bool IsEnabled(LogLevel logLevel) =>
        getCurrentConfig().LogLevelToColorMap.ContainsKey(logLevel);

    public void Log<TState>(
        LogLevel logLevel,
        EventId eventId,
        TState state,
        Exception? exception,
        Func<TState, Exception?, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        ColorConsoleLoggerConfiguration config = getCurrentConfig();
        if (config.EventId == 0 || config.EventId == eventId.Id)
        {
            ConsoleColor originalColor = Console.ForegroundColor;

            Console.ForegroundColor = config.LogLevelToColorMap[logLevel];
            Console.WriteLine($"[{eventId.Id,2}: {logLevel,-12}]");

            Console.ForegroundColor = originalColor;
            Console.Write($"     {name} - ");

            Console.ForegroundColor = config.LogLevelToColorMap[logLevel];
            Console.Write($"{formatter(state, exception)}");

            Console.ForegroundColor = originalColor;
            Console.WriteLine();
        }
    }
}
```

각 로거 인스턴스는 일반적으로 로거가 만들어지는 형식인 범주 이름을 전달하여 인스턴스화됩니다. 메서드 `IsEnabled`가 요청된 로그 수준(즉, 구성의 로그 수준 사전)이 활성화되었는지를 확인하기 위해 `getCurrentConfig().LogLevelToColorMap.ContainsKey(logLevel)`를 검사합니다.

모든 소비자가 ILogger.IsEnabled 호출할 수 있고 이전에 확인되었다는 보장은 없으므로 ILogger.Log 구현 내에서 `Log` 호출하는 것이 좋습니다. `IsEnabled` 메서드는 대부분의 구현에서 매우 빠릅니다.

C#

```
if (!IsEnabled(logLevel))
{
    return;
}
```

로거는 `name`과 `Func<ColorConsoleLoggerConfiguration>`로 현재 구성을 반환하는 객체와 함께 인스턴스화됩니다.

> ⓘ **Important**
>
> ILogger.Log 구현은 `config.EventId` 값이 설정되어 있는지 확인합니다. `config.EventId` 설정되지 않았거나 정확한 `logEntry.EventId`일치하면 로거가 색으로 기록됩니다.

사용자 지정 로거 공급자

`ILoggerProvider` 개체는 로거 인스턴스를 만듭니다. 범주별로 로거 인스턴스를 만들 필요는 없지만 NLog 또는 log4net과 같은 일부 로거에 적합합니다. 이 전략을 사용하면 다음 예제와 같이 범주별로 다른 로깅 출력 대상을 선택할 수 있습니다.

C#

```
using System.Collections.Concurrent;
using System.Runtime.Versioning;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

[UnsupportedOSPlatform("browser")]
[ProviderAlias("ColorConsole")]
public sealed class ColorConsoleLoggerProvider : ILoggerProvider
{
    private readonly IDisposable? _onChangeToken;
    private ColorConsoleLoggerConfiguration _currentConfig;
    private readonly ConcurrentDictionary<string, ColorConsoleLogger> _loggers =
        new(StringComparer.OrdinalIgnoreCase);

    public ColorConsoleLoggerProvider(
        IOptionsMonitor<ColorConsoleLoggerConfiguration> config)
    {
        _currentConfig = config.CurrentValue;
        _onChangeToken = config.OnChange(updatedConfig => _currentConfig =
updatedConfig);
    }

    public ILogger CreateLogger(string categoryName) =>
        _loggers.GetOrAdd(categoryName, name => new ColorConsoleLogger(name,
GetCurrentConfig));
```

```
    private ColorConsoleLoggerConfiguration GetCurrentConfig() => _currentConfig;

    public void Dispose()
    {
        _loggers.Clear();
        _onChangeToken?.Dispose();
    }
}
```

앞의 코드에서 CreateLogger(String) 범주 이름당 `ColorConsoleLogger` 단일 인스턴스를 만들고 ConcurrentDictionary<TKey,TValue>저장합니다.

클래스는 `ColorConsoleLoggerProvider` 다음 두 가지 특성으로 데코레이팅됩니다.

C#

```
[UnsupportedOSPlatform("browser")]
[ProviderAlias("ColorConsole")]
public sealed class ColorConsoleLoggerProvider : ILoggerProvider
```

- UnsupportedOSPlatformAttribute: `ColorConsoleLogger` 형식은 에서 `"browser"`.
- ProviderAliasAttribute: 구성 섹션에서는 `"ColorConsole"` 키를 사용하여 옵션을 정의할 수 있습니다.

구성은 유효한 구성 공급자사용하여 지정할 수 있습니다. 다음 *appsettings.json* 파일을 고려하세요.

JSON

```
{
    "Logging": {
        "ColorConsole": {
            "LogLevelToColorMap": {
                "Information": "DarkGreen",
                "Warning": "Cyan",
                "Error": "Red"
            }
        }
    }
}
```

appsettings.json 파일은 로그 수준의 색이 DarkGreen, `ColorConsoleLoggerConfiguration` 개체에 설정된 기본값을 재정의하도록 지정합니다.

사용자 지정 로거의 사용 및 등록

규칙에 따라 서비스는 애플리케이션의 시작 루틴의 일부로 종속성 주입을 위해 등록됩니다. 이 예제에서는 로깅 서비스가 *Program.cs* 파일에서 직접 등록됩니다.

사용자 지정 로깅 공급자와 해당 로거를 추가하려면, IHostApplicationBuilder.Logging 속성의 ILoggingBuilder에서 사용자 지정 확장 메서드 `AddColorConsoleLogger`를 호출하여 ILoggerProvider를 추가하세요.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.ClearProviders();
builder.Logging.AddColorConsoleLogger(configuration =>
{
    // Replace value of "Cyan" from appsettings.json.
    configuration.LogLevelToColorMap[LogLevel.Warning]
        = ConsoleColor.DarkCyan;
    // Replace value of "Red" from appsettings.json.
    configuration.LogLevelToColorMap[LogLevel.Error]
        = ConsoleColor.DarkRed;
});

using IHost host = builder.Build();

ILogger<Program> logger = host.Services.GetRequiredService<ILogger<Program>>();

logger.LogDebug(1, "Does this line get hit?");    // Not logged
logger.LogInformation(3, "Nothing to see here."); // Logs in ConsoleColor.DarkGreen
logger.LogWarning(5, "Warning... that was odd."); // Logs in ConsoleColor.DarkCyan
logger.LogError(7, "Oops, there was an error.");  // Logs in ConsoleColor.DarkRed
logger.LogTrace(5, "== 120.");                    // Not logged

await host.RunAsync();
```

규칙에 따라 `ILoggingBuilder` 확장 메서드는 사용자 지정 공급자를 등록하는 데 사용됩니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Configuration;

public static class ColorConsoleLoggerExtensions
{
    public static ILoggingBuilder AddColorConsoleLogger(
        this ILoggingBuilder builder)
```

```
    {
        builder.AddConfiguration();

        builder.Services.TryAddEnumerable(
            ServiceDescriptor.Singleton<ILoggerProvider, ColorConsoleLoggerProvider>
());

        LoggerProviderOptions.RegisterProviderOptions
            <ColorConsoleLoggerConfiguration, ColorConsoleLoggerProvider>
(builder.Services);

        return builder;
    }

    public static ILoggingBuilder AddColorConsoleLogger(
        this ILoggingBuilder builder,
        Action<ColorConsoleLoggerConfiguration> configure)
    {
        builder.AddColorConsoleLogger();
        builder.Services.Configure(configure);

        return builder;
    }
}
```

`ILoggingBuilder` 하나 이상의 `ILogger` 인스턴스를 만듭니다. `ILogger` 인스턴스는 프레임워크에서 정보를 기록하는 데 사용됩니다.

인스턴스화 코드는 *appsettings.json* 파일에서 LogLevel.Warning 및 LogLevel.Error의 색 값을 재정의합니다.

이 간단한 앱을 실행하면 다음 이미지와 유사한 색 출력이 콘솔 창에 렌더링됩니다.

참고하십시오

- .NET의 로깅
- .NET의 로깅 공급자

- .NET에서 종속성 주입
- .NET의 고성능 로깅

Last updated on 2026. 02. 06.

.NET의 고성능 로깅

.NET 6 이상 버전의 고성능 로깅 시나리오의 경우 LoggerMessageAttribute을 사용합니다. 이 방법은 런타임에 boxing, 임시 할당 및 메시지 템플릿 구문 분석을 제거하여 최상의 성능을 제공합니다.

원본 생성 로깅은 다음과 같은 로거 확장 메서드에 비해 다음과 같은 LogInformationLogDebug 성능 이점을 제공합니다.

- **박싱 제거**: 로거 확장 메서드에서는 값 형식(예: `int`)을 `object`로 변환하는 "박싱"이 필요합니다. 소스 생성 로깅은 강력한 형식의 매개 변수를 사용하여 'boxing'을 방지합니다.
- **컴파일 시간에 템플릿을 구문 분석합니다.** 로거 확장 메서드는 로그 메시지를 쓸 때마다 메시지 템플릿(명명된 형식 문자열)을 구문 분석해야 합니다. 소스 생성 로깅은 컴파일 시간에 템플릿을 한 번 구문 분석합니다.
- **할당을 줄입니다.** 원본 생성기는 개체 할당 및 임시 메모리 사용을 최소화하는 최적화된 코드를 만듭니다.

샘플 앱은 우선 순위 큐 처리 작업자 서비스를 사용하는 고성능 로깅 기능을 보여 줍니다. 앱은 작업 항목을 우선 순위에 따라 처리합니다. 이러한 작업이 수행되면 원본에서 생성된 로깅을 사용하여 로그 메시지가 생성됩니다.

> **팁**
>
> 모든 로깅 예제 소스 코드는 **샘플 브라우저**에서 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET의 로깅**을 참조하세요.

원본 생성을 사용하여 로거 메시지 정의

.NET 6 이상에서는 `partial`로 명시된 데코레이터 메서드를 LoggerMessageAttribute하여 고성능 로그 메시지를 생성합니다. 소스 생성기는 컴파일 시간에 구현을 만듭니다.

기본 로깅 방법

간단한 로그 메시지의 경우 이벤트 ID, 로그 수준 및 메시지 템플릿을 지정하는 특성을 사용하여 부분 메서드를 정의합니다.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 13,
```

```
        Level = LogLevel.Critical,
        Message = "Epic failure processing item!")]
    public static partial void FailedToProcessWorkItem(
        ILogger logger, Exception ex);
}
```

메시지 템플릿은 메서드 매개 변수로 채워진 자리 표시자를 사용합니다. 자리 표시자 이름은 템플릿 전체에서 설명적이고 일관적이어야 합니다. 구조적 로그 데이터 내에서 속성 이름 역할을 합니다. 자리 표시자 이름에는 파스칼 표기법을 사용하는 것이 좋습니다. 예: `{Item}`, `{DateTime}`

코드에서 로깅 메서드를 호출합니다. 예를 들어 작업 항목 처리 중에 예외가 발생하는 경우:

C#

```
try
{
    // Process work item.
}
catch (Exception ex)
{
    Log.FailedToProcessWorkItem(logger, ex);
}
```

이 코드는 다음과 같은 콘솔 출력을 생성합니다.

콘솔

```
crit: WorkerServiceOptions.Example.Worker[13]
      Epic failure processing item!
      System.Exception: Failed to verify communications.
```

매개 변수를 사용하여 로깅

로그 메시지에 매개 변수를 전달하려면 메서드 매개 변수로 추가합니다. 매개 변수 이름은 메시지 템플릿의 자리 표시자와 일치합니다.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 1,
        Level = LogLevel.Information,
        Message = "Processing priority item: {Item}")]
    public static partial void PriorityItemProcessed(
        ILogger logger, WorkItem item);
}
```

로거 및 매개 변수 값을 사용하여 메서드를 호출합니다.

C#

```
var workItem = queue.Dequeue();
Log.PriorityItemProcessed(logger, workItem);
```

이 코드는 다음과 같은 콘솔 출력을 생성합니다.

콘솔

```
info: WorkerServiceOptions.Example.Worker[1]
      Processing priority item: Priority-Extreme (50db062a-9732-4418-936d-
110549ad79e4): 'Verify communications'
```

구조적 로깅 저장소는 이벤트 ID와 함께 제공될 때 이벤트 이름을 사용하여 로깅을 보강할 수 있습니다. 예를 들어 Serilog 는 이벤트 이름을 사용합니다.

원본 생성을 사용하여 로거 메시지 범위 정의

로그 범위를 정의하여 일련의 로그 메시지를 추가 컨텍스트로 래핑할 수 있습니다. 소스 생성 로깅을 사용하여 `LoggerMessageAttribute` 메서드를 표준 `ILogger.BeginScope` 메서드와 결합합니다.

appsettings.json콘솔 로거 섹션에서 `IncludeScopes`.

JSON

```
{
    "Logging": {
        "Console": {
            "IncludeScopes": true
        },
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

다음 `BeginScope`을 사용하여 소스 생성 로깅 메서드를 만들고 범위 안에서 래핑합니다.

C#

```
public static partial class Log
{
```

```
    [LoggerMessage(
        EventId = 1,
        Level = LogLevel.Information,
        Message = "Processing priority item: {Item}")]
    public static partial void PriorityItemProcessed(
        ILogger logger, WorkItem item);
}
```

애플리케이션 코드의 범위 내에서 로깅 메서드를 사용합니다.

C#

```
using (_logger.BeginScope("Processing scope, started at: {DateTime}", DateTime.Now))
{
    Log.PriorityItemProcessed(_logger, workItem);
}
```

앱의 콘솔 출력에서 로그 메시지를 검사합니다. 다음 결과는 로그 범위 메시지가 포함된 로그 메시지의 우선 순위 순서를 보여줍니다.

콘솔

```
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-Extreme (7d153ef9-8894-4282-836a-
8e5e38319fb3): 'Verify communications'
```

레거시 접근 방식: LoggerMessage.Define(.NET Framework 및 .NET Core 3.1용)

.NET 6에서 원본 생성 로깅이 도입되기 전에 권장되는 고성능 로깅 방법은 메서드를 사용하여 LoggerMessage.Define 캐시 가능한 대리자를 만드는 것이었습니다. 이 방법은 이전 버전과의 호환성을 위해 계속 지원되지만 새 코드는 소스 생성 로깅 `LoggerMessageAttribute` 을 대신 사용해야 합니다.

클래스는 LoggerMessage가 로거 확장 메서드(예: LogInformation, LogDebug)에 비해 개체 할당이 적고 계산 오버헤드가 줄어드는 캐시 가능한 대리자를 생성할 수 있는 기능을 제공합니다. LoggerMessage 로거 확장 메서드에 비해 다음과 같은 성능 이점을 제공합니다.

- 로거 확장 메서드는 `int`에 대한 `object`와 같은 "boxing"(변환) 값 형식이 필요합니다. LoggerMessage 패턴은 강력한 형식의 매개 변수가 있는 정적 Action 필드 및 확장 메서드를 사용하여 boxing을 방지합니다.
- 로거 확장 메서드는 로그 메시지가 기록될 때마다 메시지 템플릿(명명된 형식 문자열)을 구문 분석해야 합니다. LoggerMessage는 메시지가 정의될 때 템플릿 구문 분석이 한번만 필요합니다.

> ⓘ **참고 항목**
>
> 사용하는 `LoggerMessage.Define` 코드를 유지 관리하는 경우 **원본 생성 로깅**으로 마이그레이션하는 것이 좋습니다. .NET Framework 또는 .NET Core 3.1 애플리케이션의 경우 `LoggerMessage.Define`를 계속 사용하십시오.

로거 메시지 정의

Define(LogLevel, EventId, String)를 사용하여 메시지 로깅을 위한 대리자를 Action 만듭니다. Define 오버로드는 명명된 형식 문자열(템플릿)에 최대 6개의 형식 매개 변수를 전달할 수 있습니다.

메서드에 Define 제공된 문자열은 보간된 문자열이 아니라 템플릿입니다. 자리 표시자는 형식이 지정된 순서대로 채워집니다. 템플릿의 자리 표시자 이름은 템플릿 전체에서 설명적이고 일관적이어야 합니다. 구조적 로그 데이터 내에서 속성 이름 역할을 합니다. 자리 표시자 이름에는 파스칼 표기법을 사용하는 것이 좋습니다. 예: `{Item}`, `{DateTime}`

각 로그 메시지는 ActionLoggerMessage.Define에서 만든 정적 필드에 저장됩니다. 예를 들어 샘플 앱은 작업 항목 처리에 대한 로그 메시지를 설명하는 필드를 만듭니다.

C#

```
private static readonly Action<ILogger, Exception> s_failedToProcessWorkItem;
```

Action을(를) 지정합니다.

- 로그 수준입니다.
- 정적 확장 메서드의 이름을 가진 고유 이벤트 식별자(EventId)입니다.
- 메시지 템플릿(명명된 형식 문자열)입니다.

작업 항목이 처리를 위해 큐에서 해제되면 작업자 서비스 앱은 다음을 설정합니다.

- 로그 수준을 LogLevel.Critical로 설정합니다.
- `13`의 이벤트 ID를 `FailedToProcessWorkItem` 메서드의 이름으로 사용합니다.
- 문자열에 대한 메시지 템플릿(명명된 형식 문자열)입니다.

C#

```
s_failedToProcessWorkItem = LoggerMessage.Define(
    LogLevel.Critical,
    new EventId(13, nameof(FailedToProcessWorkItem)),
    "Epic failure processing item!");
```

이 LoggerMessage.Define 메서드는 로그 메시지를 나타내는 대리자를 Action 구성하고 정의하는 데 사용됩니다.

구조적 로깅 저장소는 이벤트 ID와 함께 제공될 때 이벤트 이름을 사용하여 로깅을 보강할 수 있습니다. 예를 들어 Serilog 는 이벤트 이름을 사용합니다.

강력히 형식화된 Action 확장 메서드를 통해 호출됩니다. 이 메서드는 `PriorityItemProcessed` 작업 항목이 처리될 때마다 메시지를 기록합니다. `FailedToProcessWorkItem` 은 예외가 발생할 경우 호출됩니다.

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem = priorityQueue.ProcessNextHighestPriority();

                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
            }
            catch (Exception ex)
            {
                logger.FailedToProcessWorkItem(ex);
            }

            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

앱의 콘솔 출력을 검사합니다.

콘솔

```
crit: WorkerServiceOptions.Example.Worker[13]
      Epic failure processing item!
      System.Exception: Failed to verify communications.
         at WorkerServiceOptions.Example.Worker.ExecuteAsync(CancellationToken
stoppingToken) in
         ..\Worker.cs:line 27
```

로그 메시지에 매개 변수를 전달하려면 정적 필드를 만들 때 최대 6개의 형식을 정의합니다. 샘플 앱은 필드에 대한 `WorkItem` 형식을 정의함으로써 Action 항목을 처리할 때 작업 항목 세부 정보를 기록합니다.

C#

```
private static readonly Action<ILogger, WorkItem, Exception>
s_processingPriorityItem;
```

대리자의 로그 메시지 템플릿은 제공된 형식에서 자리 표시자 값을 받습니다. 샘플 앱은 항목 매개 변수가 다음과 같은 작업 항목을 추가하기 위한 대리자를 `WorkItem`정의합니다.

C#

```
s_processingPriorityItem = LoggerMessage.Define<WorkItem>(
    LogLevel.Information,
    new EventId(1, nameof(PriorityItemProcessed)),
    "Processing priority item: {Item}");
```

작업 항목이 처리 `PriorityItemProcessed`되고 있는 로깅에 대한 정적 확장 메서드는 작업 항목 인수 값을 수신하고 대리자에게 Action 전달합니다.

C#

```
public static void PriorityItemProcessed(
    this ILogger logger, WorkItem workItem) =>
    s_processingPriorityItem(logger, workItem, default!);
```

작업자 서비스의 `ExecuteAsync` 메서드 `PriorityItemProcessed` 에서 메시지를 기록하기 위해 호출됩니다.

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem = priorityQueue.ProcessNextHighestPriority();

                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
```

```
            }
            catch (Exception ex)
            {
                logger.FailedToProcessWorkItem(ex);
            }

            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

앱의 콘솔 출력을 검사합니다.

콘솔

```
info: WorkerServiceOptions.Example.Worker[1]
      Processing priority item: Priority-Extreme (50db062a-9732-4418-936d-
110549ad79e4): 'Verify communications'
```

로그 수준 보호 최적화

해당 `Log*` 메서드를 호출하기 전에 LogLevel와 ILogger.IsEnabled(LogLevel)를 확인하여 성능을 최적화할 수 있습니다. 지정된 `LogLevel`에 대해 로깅이 구성되지 않으면 ILogger.Log이(가) 호출되지 않습니다. 또한 값 형식 boxing과 매개 변수를 나타내기 위한 `object[]`의 할당이 방지됩니다.

자세한 내용은 다음을 참조하세요.

- .NET 런타임의 마이크로 벤치마크
- 로그 수준 검사에 대한 배경 및 동기 부여

참고하십시오

- .NET의 로깅
- 컴파일 시간 로깅 원본 생성

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 05.

.NET의 로그 샘플링

.NET은 중요한 정보를 잃지 않고 애플리케이션에서 내보내는 로그의 볼륨을 제어할 수 있는 로그 샘플링 기능을 제공합니다. 사용할 수 있는 샘플링 전략은 다음과 같습니다.

- 추적 기반 샘플링: 현재 추적의 샘플링 결정에 따른 샘플 로그입니다.
- 임의 확률 샘플링: 구성된 확률 규칙을 기반으로 하는 샘플 로그입니다.
- 사용자 지정 샘플링: 고유한 사용자 지정 샘플링 전략을 구현합니다. 자세한 내용은 사용자 지정 샘플링 구현을 참조하세요.

> ⓘ **참고 항목**
>
> 한 번에 하나의 샘플러만 사용할 수 있습니다. 여러 샘플러를 등록하는 경우 마지막 샘플러가 사용됩니다.

로그 샘플링은 애플리케이션에서 내보낸 로그를 보다 세밀하게 제어하여 필터링 기능을 확장합니다. 단순히 로그를 사용하거나 사용하지 않도록 설정하는 대신 샘플링을 구성하여 그 중 일부만 내보낼 수 있습니다.

예를 들어 필터링은 일반적으로 (로그 내보내지 않음) 또는 `0` (모든 로그 내보내기) 같은 `1` 확률을 사용하지만, 샘플링을 사용하면 10개의% 로그를 내보내거나 `0.1` 25개의%내보내는 것과 같은 `0.25` 중간 값을 선택할 수 있습니다.

시작하기

시작하려면 Microsoft.Extensions.Telemetry NuGet 패키지를 설치합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Telemetry
```

자세한 내용은 dotnet add package 또는 .NET 애플리케이션에서 패키지 종속성 관리를 참조하세요.

추적 기반 샘플링 구성

추적 기반 샘플링은 로그가 기본 항목과 일관되게 샘플링되도록 합니다 Activity. 추적과 로그 간의 상관 관계를 유지하려는 경우에 유용합니다. 가이드에 설명된 대로 추적 샘플링을 사용하도록 설정한 다음, 그에 따라 추적 기반 로그 샘플링을 구성할 수 있습니다.

C#

```
builder.Logging.AddTraceBasedSampler();
```

추적 기반 샘플링을 사용하도록 설정하면 기본 Activity 샘플링이 샘플링된 경우에만 로그가 내보내집니다. 샘플링 결정은 현재 Recorded 값에서 가져옵니다.

임의 확률 샘플링 구성

무작위 확률 샘플링을 사용하면 구성된 확률 규칙에 따라 로그를 샘플링할 수 있습니다. 다음과 관련된 규칙을 정의할 수 있습니다.

- 로그 범주
- 로그 수준
- 이벤트 ID

규칙으로 임의 확률 샘플링을 구성하는 방법에는 여러 가지가 있습니다.

파일 기반 구성

*appsettings.json*구성 섹션을 만듭니다. 예를 들면 다음과 같습니다.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    }
  },

  "RandomProbabilisticSampler": {
    "Rules": [
      {
        "CategoryName": "Microsoft.AspNetCore.*",
        "Probability": 0.25,
        "LogLevel": "Information"
      },
      {
        "CategoryName": "System.*",
        "Probability": 0.1
      },
      {
```

```
        "EventId": 1001,
        "Probability": 0.05
      }
    ]
  }
}
```

이전 구성은 다음과 같습니다.

- 모든 수준의 `System.`로 시작하는 범주에서 로그의 10%를 샘플링합니다.
- 로 시작하는 `Microsoft.AspNetCore.`의 LogLevel.Information범주에서 로그의 25%를 샘플링합니다.
- 모든 범주 및 수준의 이벤트 ID가 1001인 로그의 5개% 샘플링합니다.
- 다른 모든 로그의 100% 샘플링합니다.

> ⓘ **Important**
>
> 값은 Probability 0에서 1까지의 값을 가진 확률을 나타냅니다. 예를 들어 0.25는 25개의% 로그가 샘플링됨을 의미합니다. 0은 로그가 샘플링되지 않음을 의미하고 1은 모든 로그가 샘플링됨을 의미합니다. "이러한 경우 숫자 값 0과 1은 특정 규칙에 대한 모든 로그를 효과적으로 비활성화하거나 활성화하는 데 사용할 수 있습니다." 확률은 0보다 작거나 1보다 클 수 없으며 애플리케이션에서 발생할 경우 예외가 throw됩니다.

샘플러를 구성에 등록하려면 다음 코드를 고려합니다.

C#

```
builder.Logging.AddRandomProbabilisticSampler(builder.Configuration);
```

실행 중인 앱에서 샘플링 규칙 변경

임의 확률 샘플링은 인터페이스를 통해 IOptionsMonitor<TOptions> 런타임 구성 업데이트를 지원합니다. 파일 구성 공급자와 같이 다시 로드를 지원하는 구성 공급자를 사용하는 경우 애플리케이션을 다시 시작하지 않고 런타임에 샘플링 규칙을 업데이트할 수 있습니다.

예를 들어 효과적으로 no-op역할을 하는 다음 *appsettings.json*사용하여 애플리케이션을 시작할 수 있습니다.

JSON

```
{
  "Logging": {
    "RandomProbabilisticSampler": {
      "Rules": [
```

```
        {
          "Probability": 1
        }
      ]
    }
  }
}
```

앱이 실행되는 동안 다음 구성으로 *appsettings.json* 업데이트할 수 있습니다.

JSON

```
{
  "Logging": {
    "RandomProbabilisticSampler": {
      "Rules": [
        {
          "Probability": 0.01,
          "LogLevel": "Information"
        }
      ]
    }
  }
}
```

새 규칙은, 예를 들어, 이전 구성에서는 자동으로 적용됩니다. 이 경우, LogLevel.Information을 포함한 로그의 1%가 샘플링됩니다.

샘플링 규칙 적용 방법

알고리즘은 로그 필터링과 매우 유사하지만 몇 가지 차이점이 있습니다.

로그 샘플링 규칙 평가는 각 로그 레코드에서 수행되지만 캐싱과 같은 성능 최적화가 있습니다. 다음 알고리즘은 지정된 범주의 각 로그 레코드에 사용됩니다.

- 로거의 로그 수준과 같거나 높은 규칙을 `LogLevel` 선택합니다.
- `EventId`가 정의되지 않았거나 정의되었고 로그 이벤트 ID와 같은 규칙을 선택합니다.
- 가장 긴 일치 범주 접두사를 가진 규칙을 선택합니다. 일치하는 규칙이 없는 경우 범주를 지정하지 않는 규칙을 모두 선택합니다.
- 여러 규칙을 선택하는 경우 **마지막** 규칙을 사용합니다.
- 규칙을 선택하지 않으면 샘플링이 적용되지 않습니다. 예를 들어 로그 레코드는 평소와 같이 내보내집니다.

인라인 코드 구성

C#

```
builder.Logging.AddRandomProbabilisticSampler(options =>
{
    options.Rules.Add(
        new RandomProbabilisticSamplerFilterRule(
            probability: 0.05d,
            eventId : 1001));
});
```

이전 구성은 다음과 같습니다.

- 모든 범주 및 수준의 이벤트 ID가 1001인 로그의 5개% 샘플링합니다.
- 다른 모든 로그의 100% 샘플링합니다.

단순 확률 구성

기본 시나리오의 경우 지정된 수준 이하의 모든 로그에 적용되는 단일 확률 값을 구성할 수 있습니다.

C#

```
builder.Logging.AddRandomProbabilisticSampler(0.01, LogLevel.Information);
builder.Logging.AddRandomProbabilisticSampler(0.1, LogLevel.Warning);
```

위 코드는 Warning 로그의 10%와 Information 이하 로그의 1%를 샘플링하는 샘플러를 등록합니다. 구성에 Information규칙이 없는 경우에는 Warning 로그 중 10%를 샘플링하고, Information를 포함한 그 아래 모든 수준도 샘플링했을 것입니다.

사용자 지정 샘플링 구현

LoggingSampler 추상 클래스에서 파생하고 추상 멤버를 재정의하여 사용자 지정 샘플링 전략을 만들 수 있습니다. 이렇게 하면 샘플링 동작을 특정 요구 사항에 맞게 조정할 수 있습니다. 예를 들어 사용자 지정 샘플러가 다음을 수행할 수 있습니다.

- 로그 상태의 특정 키/값 쌍의 존재 및 값에 따라 샘플링 결정을 내립니다.
- 미리 정의된 시간 간격 내의 로그 수가 특정 임계값 미만으로 유지되는 경우에만 로그를 내보내는 것과 같은 속도 제한 논리를 적용합니다.

사용자 지정 샘플러를 구현하려면 다음 단계를 수행합니다.

1. 에서 상속되는 클래스를 만듭니다.LoggingSampler
2. 사용자 지정 샘플링 논리를 정의하기 위해 LoggingSampler.ShouldSample 메서드를 재정의합니다.

3. 확장 메서드를 사용하여 로깅 파이프라인에 사용자 지정 샘플러를 AddSampler 등록합니다.

필터링되지 않은 각 로그 레코드에 LoggingSampler.ShouldSample 대해 메서드가 정확히 한 번 호출됩니다. 반환 값은 로그 레코드를 내보내야 하는지 여부를 결정합니다.

성능 고려 사항

로그 샘플링은 CPU 사용량이 약간 증가하면서 스토리지 비용을 절감하도록 설계되었습니다. 애플리케이션에서 저장하는 데 비용이 많이 드는 많은 양의 로그를 생성하는 경우 샘플링은 해당 볼륨을 줄이는 데 도움이 될 수 있습니다. 적절하게 구성된 경우 샘플링은 인시던트 진단에 중요한 정보를 손실하지 않고 스토리지 비용을 절감할 수 있습니다.

기본 제공 샘플링은 벤치마크를 참조하세요.

샘플링을 사용하는 시기에 대한 로그 수준 지침

테이블 확장

로그 수준	Recommendation
Trace	일반적으로 프로덕션에서 이러한 로그를 사용하지 않도록 설정하므로 샘플링을 적용하지 마세요.
Debug	일반적으로 프로덕션에서 이러한 로그를 사용하지 않도록 설정하므로 샘플링을 적용하지 마세요.
Information	반드시 샘플링을 적용하세요
Warning	샘플링 적용 고려
Error	샘플링 적용 안 함
Critical	샘플링 적용 안 함

모범 사례

- 더 높은 샘플링 속도로 시작하고 필요에 따라 아래로 조정합니다.
- 범주 기반 규칙을 사용하여 특정 구성 요소를 대상으로 지정합니다.
- 분산 추적을 사용하는 경우 추적 기반 샘플링을 구현하는 것이 좋습니다.
- 샘플링 규칙의 효율성을 전체적으로 모니터링합니다.
- 애플리케이션에 적합한 균형을 찾으세요. 샘플링 속도가 너무 낮으면 관찰 가능성을 줄일 수 있지만 속도가 너무 높으면 비용이 증가할 수 있습니다.

참고하십시오

- .NET의 로깅
- .NET의 고성능 로깅
- OpenTelemetry 추적 샘플링

Last updated on 2026. 02. 05.

.NET의 로그 버퍼링

.NET은 특정 조건이 충족될 때까지 로그 배출을 지연할 수 있는 로그 버퍼링 기능을 제공합니다. 로그 버퍼링은 다음을 수행하려는 시나리오에서 유용합니다.

- 내보내는지 여부를 결정하기 전에 특정 작업에서 모든 로그를 수집합니다.
- 정상적인 작업 중에 로그가 내보내지는 것을 방지하지만 오류가 발생할 때 내보냅니다.
- 스토리지에 기록된 로그 수를 줄여 성능을 최적화합니다.

버퍼링된 로그는 프로세스 메모리의 임시 순환 버퍼에 저장되며 다음 조건이 적용됩니다.

- 버퍼가 가득 차면 가장 오래된 로그가 삭제되고 내보내지 않습니다.
- 버퍼링된 로그를 내보내려면, Flush() 클래스나 GlobalLogBuffer 클래스에서 PerRequestLogBuffer을(를) 호출할 수 있습니다.
- 버퍼를 플러시하지 않으면 애플리케이션이 실행될 때 버퍼링된 로그가 결국 삭제되므로 해당 로그가 비활성화된 것처럼 효과적으로 동작합니다.

사용할 수 있는 버퍼링 전략에는 다음 두 가지가 있습니다.

- 전역 버퍼링: 전체 애플리케이션에서 로그를 버퍼링합니다.
- 요청별 버퍼링: 사용 가능한 경우 각 개별 HTTP 요청에 대한 로그를 버퍼링합니다. 그렇지 않으면 전역 버퍼로 버퍼링합니다.

> ⓘ **참고 항목**
>
> 로그 버퍼링은 .NET 9 이상 버전에서 사용할 수 있습니다.

로그 버퍼링은 모든 로깅 공급자에서 작동합니다. 사용하는 로깅 공급자가 인터페이스를 IBufferedLogger 구현하지 않는 경우 로그 버퍼링은 버퍼를 플러시할 때 각 버퍼링된 로그 레코드에서 직접 로그 메서드를 호출합니다.

로그 버퍼링은 로그를 일시적으로 캡처하고 저장할 수 있도록 하여 필터링 기능을 확장합니다. 버퍼링을 통해 즉각적인 내보내기 또는 삭제 결정을 내리는 대신 메모리에 로그를 보관하고 나중에 로그를 내보할지 여부를 결정할 수 있습니다.

시작하기

시작하려면 위해 Microsoft.Extensions.Telemetry NuGet 패키지를 설치 합니다. 또는 위해 Microsoft.AspNetCore.Diagnostics.Middleware NuGet 패키지를 설치 합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Telemetry
dotnet add package Microsoft.AspNetCore.Diagnostics.Middleware
```

패키지를 추가하는 방법에 대한 자세한 내용은 .NET 애플리케이션에서 dotnet add package 또는 Manage package dependencies를 참조하세요.

전역 버퍼링

전역 버퍼링을 사용하면 전체 애플리케이션에서 로그를 버퍼링할 수 있습니다. 필터 규칙을 사용하여 버퍼링할 로그를 구성하고, 필요에 따라 버퍼를 플러시하여 해당 로그를 내보낼 수 있습니다.

간단한 구성

특정 로그 수준 이하에서 전역 버퍼링을 사용하도록 설정하려면 해당 수준을 지정합니다.

C#

```
// Add the Global buffer to the logging pipeline.
hostBuilder.Logging.AddGlobalBuffer(LogLevel.Information);
```

위의 구성을 사용하면 수준 LogLevel.Information 이하의 로그를 버퍼링할 수 있습니다.

파일 기반 구성

*appsettings.json*구성 섹션을 만듭니다. 예를 들면 다음과 같습니다.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information"
    },
```

```
    "GlobalLogBuffering": {
      "MaxBufferSizeInBytes": 104857600,
      "MaxLogRecordSizeInBytes": 51200,
      "AutoFlushDuration": "00:00:30",
      "Rules": [
        {
          "CategoryName": "BufferingDemo",
          "LogLevel": "Information"
        },
        {
          "EventId": 1001
        }
      ]
    }
  }
}
```

이전 구성은 다음과 같습니다.

- 시작이 `BufferingDemo`인 범주의 로그를 수준 LogLevel.Information 이하에서 버퍼링합니다.
- 이벤트 ID가 1001인 모든 로그를 버퍼링합니다.
- 최대 버퍼 크기를 약 100MB로 설정합니다.
- 최대 로그 레코드 크기를 50KB로 설정합니다.
- 수동 플러시 후 30초의 자동 플러시 기간을 설정합니다.

구성에 로그 버퍼링을 등록하려면 다음 코드를 고려합니다.

C#

```
// Add the Global buffer to the logging pipeline.
hostBuilder.Logging.AddGlobalBuffer(hostBuilder.Configuration.GetSection("Logging")
);
```

인라인 코드 구성

C#

```
// Add the Global buffer to the logging pipeline.
hostBuilder.Logging.AddGlobalBuffer(options =>
{
    options.MaxBufferSizeInBytes = 104857600; // 100 MB
    options.MaxLogRecordSizeInBytes = 51200; // 50 KB
    options.AutoFlushDuration = TimeSpan.FromSeconds(30);
    options.Rules.Add(new LogBufferingFilterRule(
        categoryName: "BufferingDemo",
        logLevel: LogLevel.Information));
```

```
    options.Rules.Add(new LogBufferingFilterRule(eventId: 1001));
});
```

이전 구성은 다음과 같습니다.

- 시작이 `BufferingDemo`인 범주의 로그를 수준 LogLevel.Information 이하에서 버퍼링합니다.
- 이벤트 ID가 1001인 모든 로그를 버퍼링합니다.
- 최대 버퍼 크기를 약 100MB로 설정합니다.
- 최대 로그 레코드 크기를 50KB로 설정합니다.
- 수동 플러시 후 30초의 자동 플러시 기간을 설정합니다.

버퍼 비우기

버퍼링된 로그를 플러시하려면 추상 클래스를 `GlobalLogBuffer` 삽입하고 메서드를 호출합니다 `Flush()` .

cs

```cs
public class MyService
{
    private readonly GlobalLogBuffer _buffer;

    public MyService(GlobalLogBuffer buffer)
    {
        _buffer = buffer;
    }

    public void HandleException(Exception ex)
    {
        _buffer.Flush();

        // After flushing, log buffering will be temporarily suspended (= all logs will be emitted immediately)
        // for the duration specified by AutoFlushDuration.
    }
}
```

요청별 버퍼링

요청별 버퍼링은 ASP.NET Core 애플리케이션과 관련이 있으며 각 HTTP 요청에 대해 독립적으로 로그를 버퍼링할 수 있습니다. 각 요청의 버퍼는 요청이 시작되고 요청이 종료될 때 삭제될 때 만들어지므로 버퍼를 플러시하지 않으면 요청이 종료되면 로그가 손실됩니다. 이렇게 하면 오류가 발생하는 경우와 같이 실제로 필요한 경우에만 버퍼를 플러시하는 것이 유용합니다.

요청별 버퍼링은 전역 버퍼링과 긴밀하게 결합됩니다. 로그 항목이 요청별 버퍼에 버퍼링되어야 하지만 버퍼링 시도 시 활성 HTTP 컨텍스트가 없는 경우 대신 전역 버퍼로 버퍼링됩니다. 버퍼 플러시가 트리거되면 요청당 버퍼가 먼저 플러시되고 전역 버퍼가 플러시됩니다.

간단한 구성

특정 로그 수준 이하의 로그만 버퍼링하려면 다음을 수행합니다.

cs

```cs
builder.Logging.AddPerIncomingRequestBuffer(LogLevel.Information);
```

파일 기반 구성

*appsettings.json*에 구성 섹션을 만듭니다.

JSON

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.*": "None"
    },

    "PerIncomingRequestLogBuffering": {
      "AutoFlushDuration": "00:00:05",
      "Rules": [
        {
          "CategoryName": "PerRequestLogBufferingFileBased.*",
          "LogLevel": "Information"
        }
      ]
    }
  }
}
```

이전 구성은 다음과 같습니다.

- 시작이 `PerRequestLogBufferingFileBased.`인 범주의 로그를 수준 LogLevel.Information 이하에서 버퍼링합니다.
- 수동 플러시 후 5초의 자동 플러시 기간을 설정합니다.

구성에 로그 버퍼링을 등록하려면 다음 코드를 고려합니다.

cs

```
builder.Logging.AddPerIncomingRequestBuffer(builder.Configuration.GetSection("Loggi
ng"));
```

인라인 코드 구성

cs

```
builder.Logging.AddPerIncomingRequestBuffer(options =>
{
    options.AutoFlushDuration = TimeSpan.FromSeconds(5);
    options.Rules.Add(new
Microsoft.Extensions.Diagnostics.Buffering.LogBufferingFilterRule("PerRequestLogBuf
feringCodeBased.*", LogLevel.Information));
});
```

이전 구성은 다음과 같습니다.

- 시작이 `PerRequestLogBufferingFileBased.`인 범주의 로그를 수준 LogLevel.Information 이하에서 버퍼링합니다.
- 수동 플러시 후 5초의 자동 플러시 기간을 설정합니다.

요청당 버퍼를 플러시합니다

현재 요청에 대해 버퍼링된 로그를 플러시하려면 추상 클래스를 `PerRequestLogBuffer` 삽입하고 메서드를 호출합니다 `Flush()` .

cs

```
[ApiController]
[Route("[controller]")]
public class HomeController : ControllerBase
{
    private readonly ILogger<HomeController> _logger;
    private readonly PerRequestLogBuffer _buffer;

    public HomeController(ILogger<HomeController> logger, PerRequestLogBuffer
buffer)
    {
        _logger = logger;
        _buffer = buffer;
    }

    [HttpGet("index/{id}")]
    public IActionResult Index(int id)
    {
        try
        {
```

```
            _logger.RequestStarted(id);

            // Simulate exception every 10th request
            if (id % 10 == 0)
            {
                throw new Exception("Simulated exception in controller");
            }

            _logger.RequestEnded(id);

            return Ok();
        }
        catch
        {
            _logger.ErrorMessage(id);
            _buffer.Flush();

            _logger.ExceptionHandlingFinished(id);

            return StatusCode(500, "An error occurred.");
        }
    }
}
```

> ⓘ **참고 항목**
>
> 요청당 버퍼를 플러시하면 전역 버퍼도 플러시됩니다.

버퍼링 규칙 적용 방법

로그 버퍼링 규칙 평가는 각 로그 레코드에서 수행됩니다. 각 로그 레코드에 다음 알고리즘이 사용됩니다.

1. 로그 항목이 규칙과 일치하면 즉시 내보내는 대신 버퍼링됩니다.
2. 로그 항목이 규칙과 일치하지 않으면 그대로 내보내집니다.
3. 버퍼 크기 제한에 도달하면 가장 오래된 버퍼링된 로그 항목이 삭제됩니다(내보내지 않음!).
4. 로그 항목 크기가 최대 로그 레코드 크기보다 크면 버퍼링되지 않으며 정상적으로 내보내집니다.

각 로그 레코드에 대해 알고리즘은 다음을 확인합니다.

- 로그 수준이 규칙의 로그 수준과 일치하면(같거나 낮음)
- 범주 이름이 규칙의 `CategoryName` 접두사로 시작하는 경우
- 이벤트 ID가 규칙의 `EventId`와 일치하면.
- 이벤트 이름이 규칙의 `EventName`과(와) 일치하는 경우.

- 규칙의 `Attributes`과(와) 특성이 일치하는 경우.

실행 중인 앱에서 버퍼 필터링 규칙 변경

전역 버퍼링 및 요청별 버퍼링 모두 인터페이스를 통한 IOptionsMonitor<TOptions> 런타임 구성 업데이트를 지원합니다. 파일 구성 공급자와 같이 다시 로드를 지원하는 구성 공급자를 사용하는 경우 애플리케이션을 다시 시작하지 않고 런타임에 필터링 규칙을 업데이트할 수 있습니다.

예를 들어, 다음의 *appsettings.json*을 사용하여 애플리케이션을 시작할 수 있습니다. 이렇게 하면 LogLevel.Information으로 시작하는 수준 및 범주의 로그에 대해 `PerRequestLogBufferingFileBased.` 수준으로 로그 버퍼링이 가능합니다.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.*": "None"
    },

    "PerIncomingRequestLogBuffering": {
      "AutoFlushDuration": "00:00:05",
      "Rules": [
        {
          "CategoryName": "PerRequestLogBufferingFileBased.*",
          "LogLevel": "Information"
        }
      ]
    }
  }
}
```

앱이 실행되는 동안 다음 구성으로 *appsettings.json* 업데이트할 수 있습니다.

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.*": "None"
    },

    "PerIncomingRequestLogBuffering": {
```

```
      "Rules": [
        {
          "LogLevel": "Information"
        }
      ]
    }
  }
}
```

새 규칙이 자동으로 적용됩니다. 예를 들어, 이전 구성에서는 LogLevel.Information 수준의 모든 로그가 버퍼링됩니다.

성능 고려 사항

로그 버퍼링은 메모리 사용량과 로그 스토리지 비용 간의 절충을 제공합니다. 메모리의 로그를 버퍼링하면 다음을 수행할 수 있습니다.

1. 런타임 조건에 따라 로그를 선택적으로 내보낸다.
2. 불필요한 로그를 스토리지에 기록하지 않고 삭제합니다.

그러나 특히 처리량이 높은 애플리케이션에서 메모리 사용량을 염두에 두어야 합니다. 과도한 메모리 사용을 방지하기 위해 적절한 버퍼 크기 제한을 구성합니다.

모범 사례

- 애플리케이션의 메모리 제약 조건에 따라 적절한 버퍼 크기 제한을 설정합니다.
- 웹 애플리케이션에 대해 요청별 버퍼링을 사용하여 요청별로 로그를 격리합니다.
- 메모리 사용량 및 로그 가용성의 균형을 맞추기 위해 자동 플러시 기간을 신중하게 구성합니다.
- 중요한 이벤트(예: 오류 및 경고)에 대한 명시적 플러시 트리거를 구현합니다.
- 프로덕션에서 버퍼 메모리 사용량을 모니터링하여 허용 가능한 한도 내에서 유지되도록 합니다.

제한점

- .NET 8 및 이전 버전에서는 로그 버퍼링이 지원되지 않습니다.
- 로그 순서는 유지되지 않습니다. 그러나 원래 타임스탬프는 유지됩니다.
- 각 로깅 공급자당 사용자 지정 구성은 지원되지 않습니다. 모든 공급자에 동일한 구성이 사용됩니다.
- 로그 범위는 지원되지 않습니다. 즉, 메서드를 BeginScope 사용하는 경우 버퍼링된 로그 레코드는 범위와 연결되지 않습니다.

- 원래 로그 레코드의 모든 정보가 유지되는 것은 아닙니다. 로그 버퍼링은 내부적으로 BufferedLogRecord 클래스를 플러시할 때 사용하며, 다음 속성은 항상 비어 있습니다.
  - ActivitySpanId
  - ActivityTraceId
  - ManagedThreadId
  - MessageTemplate

참고하십시오

- 로그 샘플링
- .NET의 로깅
- .NET의 고성능 로깅

Last updated on 2026. 02. 05.

콘솔 로그 서식 지정

네임스페이 `Microsoft.Extensions.Logging.Console` 스는 콘솔 로그에서 사용자 지정 서식 지정을 지원합니다. 세 가지 미리 정의된 서식 지정 옵션을 사용할 수 있습니다SimpleSystemdJson.

> **Important**
>
> .NET 5 ConsoleLoggerFormat 이전에는 열거형을 사용하여 로그 형식을 선택할 수 있었습니다. 형식은 사람이 읽을 수 있는 `Default` 또는 한 줄로 구성된 `Systemd` 형식 중에서 선택할 수 있습니다. 그러나 이러한 항목은 사용자 지정할 수 **없으며** 이제 더 이상 사용되지 않습니다.

이 문서에서는 콘솔 로그 포맷터에 대해 알아봅니다. 샘플 소스 코드는 다음 방법을 보여 줍니다.

- 새 포맷터를 등록합니다.
- 코드 또는 구성을 통해 사용할 등록된 포맷터를 선택합니다.
- 사용자 지정 포맷터를 구현합니다. 구성을 IOptionsMonitor<TOptions>을 통해 업데이트하고 사용자 지정 색 서식을 사용하도록 설정합니다.

> **팁**
>
> 모든 로깅 예제 소스 코드는 **샘플 브라우저**에서 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET의 로깅**을 참조하세요.

포맷터 등록

Console 로깅 공급자에는 미리 정의된 여러 포맷터가 있으며 사용자 고유의 사용자 지정 포맷터를 작성하는 기능을 노출합니다. 사용 가능한 포맷터를 등록하려면 해당 `Add{Type}Console` 확장 메서드를 사용합니다.

테이블 확장

사용 가능한 형식	형식을 등록하는 방법
ConsoleFormatterNames.Json	ConsoleLoggerExtensions.AddJsonConsole
ConsoleFormatterNames.Simple	ConsoleLoggerExtensions.AddSimpleConsole
ConsoleFormatterNames.Systemd	ConsoleLoggerExtensions.AddSystemdConsole

Simple

콘솔 포맷터 `Simple`를 사용하려면 `AddSimpleConsole`을 통해 등록하세요.

C#

```
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory =
    LoggerFactory.Create(builder =>
        builder.AddSimpleConsole(options =>
        {
            options.IncludeScopes = true;
            options.SingleLine = true;
            options.TimestampFormat = "HH:mm:ss ";
        }));

ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
using (logger.BeginScope("[scope is enabled]"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("Logs contain timestamp and log level.");
    logger.LogInformation("Each log message is fit in a single line.");
}
```

이전 샘플 소스 코드에서 ConsoleFormatterNames.Simple 포맷터가 등록되었습니다. 각 로그 메시지의 시간 및 로그 수준과 같은 정보를 래핑할 뿐만 아니라 ANSI 색 포함 및 메시지 들여쓰기를 허용하는 기능을 로그에 제공합니다.

이 샘플 앱을 실행하면 아래와 같이 로그 메시지의 형식이 지정됩니다.

```
10:29:22 info: Program[0] => [scope is enabled] Hello World!
10:29:22 info: Program[0] => [scope is enabled] Logs contain timestamp and log level.
10:29:22 info: Program[0] => [scope is enabled] Each log message is fit in a single line.
```

Systemd

ConsoleFormatterNames.Systemd 콘솔 로거:

- "Syslog" 로그 수준 형식 및 심각도를 사용합니다.
- 색을 사용하여 메시지의 서식을 **지정하지** 않습니다.
- 항상 메시지를 한 줄로 기록합니다.

컨테이너는 일반적으로 콘솔 로깅을 사용하는 경우가 많으며, 이러한 경우에 `Systemd` 유용합니다. `Simple` 또한 콘솔 로거는 한 줄로 기록되는 압축 버전을 사용하도록 설정하고 이전 샘플에 표시된 대로 색을 사용하지 않도록 설정할 수도 있습니다.

C#

```
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory =
    LoggerFactory.Create(builder =>
        builder.AddSystemdConsole(options =>
        {
            options.IncludeScopes = true;
            options.TimestampFormat = "HH:mm:ss ";
        }));

ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
using (logger.BeginScope("[scope is enabled]"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("Logs contain timestamp and log level.");
    logger.LogInformation("Systemd console logs never provide color options.");
    logger.LogInformation("Systemd console logs always appear in a single line.");
}
```

이 예제에서는 다음 로그 메시지와 유사한 출력을 생성합니다.

```
<6>10:29:55 Program[0] => [scope is enabled] Hello World!
<6>10:29:55 Program[0] => [scope is enabled] Logs contain timestamp and log level.
<6>10:29:55 Program[0] => [scope is enabled] Systemd console logs never provide color options.
<6>10:29:55 Program[0] => [scope is enabled] Systemd console logs always appear in a single line.
```

Json

JSON 형식 `Json` 으로 로그를 작성하기 위해 콘솔 포맷터가 사용됩니다. 샘플 소스 코드는 ASP.NET Core 앱이 등록하는 방법을 보여줍니다. 템플릿을 `webapp` 사용하여 dotnet new 명령을 사용하여 새 ASP.NET Core 앱을 만듭니다.

.NET CLI

```
dotnet new webapp -o Console.ExampleFormatters.Json
```

템플릿 코드를 사용하여 앱을 실행할 때 아래의 기본 로그 형식을 가져옵니다.

콘솔

```
info: Console.ExampleFormatters.Json.Startup[0]
      Hello .NET friends!
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

```
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\snippets\logging\console-formatter-json
```

기본적으로 `Simple` 콘솔 로그 포맷터는 기본 구성으로 선택됩니다. `AddJsonConsole` 호출 하여 이를 변경합니다.

C#

```
using System.Text.Json;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddJsonConsole(options =>
{
    options.IncludeScopes = false;
    options.TimestampFormat = "HH:mm:ss ";
    options.JsonWriterOptions = new JsonWriterOptions
    {
        Indented = true
    };
});

using IHost host = builder.Build();

var logger =
    host.Services
        .GetRequiredService<ILoggerFactory>()
        .CreateLogger<Program>();

logger.LogInformation("Hello .NET friends!");

await host.RunAsync();
```

또는 *appsettings.json* 파일에 있는 것과 같은 로깅 구성을 사용하여 이를 구성할 수도 있습니다.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        },
        "Console": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            },
```

```
            "FormatterName": "json",
            "FormatterOptions": {
                "SingleLine": true,
                "IncludeScopes": true,
                "TimestampFormat": "HH:mm:ss ",
                "UseUtcTimestamp": true,
                "JsonWriterOptions": {
                    "Indented": true
                }
            }
        }
    },
    "AllowedHosts": "*"
}
```

위의 변경 내용으로 앱을 다시 실행하면 로그 메시지의 형식이 JSON으로 지정됩니다.

JSON

```
{
  "Timestamp": "02:28:19 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Console.ExampleFormatters.Json.Startup",
  "Message": "Hello .NET friends!",
  "State": {
    "Message": "Hello .NET friends!",
    "{OriginalFormat}": "Hello .NET friends!"
  }
}
{
  "Timestamp": "02:28:21 ",
  "EventId": 14,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Now listening on: https://localhost:5001",
  "State": {
    "Message": "Now listening on: https://localhost:5001",
    "address": "https://localhost:5001",
    "{OriginalFormat}": "Now listening on: {address}"
  }
}
{
  "Timestamp": "02:28:21 ",
  "EventId": 14,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Now listening on: http://localhost:5000",
  "State": {
    "Message": "Now listening on: http://localhost:5000",
    "address": "http://localhost:5000",
    "{OriginalFormat}": "Now listening on: {address}"
  }
}
```

```
{
  "Timestamp": "02:28:21 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Application started. Press Ctrl\u002BC to shut down.",
  "State": {
    "Message": "Application started. Press Ctrl\u002BC to shut down.",
    "{OriginalFormat}": "Application started. Press Ctrl\u002BC to shut down."
  }
}
{
  "Timestamp": "02:28:21 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Hosting environment: Development",
  "State": {
    "Message": "Hosting environment: Development",
    "envName": "Development",
    "{OriginalFormat}": "Hosting environment: {envName}"
  }
}
{
  "Timestamp": "02:28:21 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Content root path: .\\snippets\\logging\\console-formatter-json",
  "State": {
    "Message": "Content root path: .\\snippets\\logging\\console-formatter-json",
    "contentRoot": ".\\snippets\\logging\\console-formatter-json",
    "{OriginalFormat}": "Content root path: {contentRoot}"
  }
}
```


팁

`Json` 콘솔 포맷터는 기본적으로 각 메시지를 한 줄로 기록합니다. 포맷터를 구성할 때 더 읽기 쉽게 하려면 JsonWriterOptions.Indented를 `true`로 설정합니다.

주의

JSON 콘솔 포맷터를 사용하는 경우 이미 JSON으로 serialize된 로그 메시지를 전달하지 마세요. 로깅 인프라 자체는 로그 메시지의 serialization을 관리합니다. 따라서 이미 serialize된 로그 메시지를 전달하면 이중 직렬화되어 잘못된 형식의 출력이 발생합니다.

구성으로 포맷터 설정

이전 샘플에서는 프로그래밍 방식으로 포맷터를 등록하는 방법을 보여 줍니다. 또는 구성을 사용하여 이 작업을 수행할 수 있습니다. 이전 웹 애플리케이션 샘플 소스 코드를 고려합니다. *Program.cs* 파일에서 호출 `ConfigureLogging` 하지 않고 *appsettings.json* 파일을 업데이트하면 동일한 결과를 얻을 수 있습니다. 업데이트 `appsettings.json` 된 파일은 다음과 같이 포맷터를 구성합니다.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        },
        "Console": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            },
            "FormatterName": "json",
            "FormatterOptions": {
                "SingleLine": true,
                "IncludeScopes": true,
                "TimestampFormat": "HH:mm:ss ",
                "UseUtcTimestamp": true,
                "JsonWriterOptions": {
                    "Indented": true
                }
            }
        }
    },
    "AllowedHosts": "*"
}
```

설정해야 하는 두 가지 키 값은 다음과 `"FormatterName"`같습니다`"FormatterOptions"`. 값이 설정된 `"FormatterName"` 포맷터가 이미 등록된 경우 해당 포맷터가 선택되고 노드 내에서 `"FormatterOptions"` 키로 제공되는 한 해당 속성을 구성할 수 있습니다. 미리 정의된 포맷터 이름은 ConsoleFormatterNames에서 예약되어 있습니다.

- ConsoleFormatterNames.Json
- ConsoleFormatterNames.Simple
- ConsoleFormatterNames.Systemd

사용자 지정 포맷터 구현

사용자 지정 포맷터를 구현하려면 다음을 수행해야 합니다.

- 사용자 지정 포맷터를 나타내는 하위 클래스 ConsoleFormatter 를 만듭니다.
- 사용자 지정 포맷터를 등록하려면 다음을 사용하세요.
  - ConsoleLoggerExtensions.AddConsole
  - ConsoleLoggerExtensions.AddConsoleFormatter<TFormatter,TOptions> (ILoggingBuilder, Action<TOptions>)

이 작업을 처리하는 확장 메서드를 만듭니다.

C#

```
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.Custom;

public static class ConsoleLoggerExtensions
{
    public static ILoggingBuilder AddCustomFormatter(
        this ILoggingBuilder builder,
        Action<CustomOptions> configure) =>
        builder.AddConsole(options => options.FormatterName = "customName")
            .AddConsoleFormatter<CustomFormatter, CustomOptions>(configure);
}
```

다음과 `CustomOptions` 같이 정의됩니다.

C#

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.Custom;

public sealed class CustomOptions : ConsoleFormatterOptions
{
    public string? CustomPrefix { get; set; }
}
```

앞의 코드에서 옵션은 .의 ConsoleFormatterOptions하위 클래스입니다.

`AddConsoleFormatter` API:

- 의 하위 클래스를 등록합니다 `ConsoleFormatter`.
- 구성을 처리합니다. 변경 토큰을 사용하여 옵션 패턴 및 IOptionsMonitor 인터페이스에 따라 업데이트를 동기화합니다.

C#

```
using Console.ExampleFormatters.Custom;
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory =
    LoggerFactory.Create(builder =>
        builder.AddCustomFormatter(options =>
            options.CustomPrefix = " ~~~~~ "));

ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
using (logger.BeginScope("TODO: Add logic to enable scopes"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("TODO: Add logic to enable timestamp and log level
info.");
}
```

`CustomFormatter`의 하위 클래스를 정의합니다: `ConsoleFormatter`

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.Custom;

public sealed class CustomFormatter : ConsoleFormatter, IDisposable
{
    private readonly IDisposable? _optionsReloadToken;
    private CustomOptions _formatterOptions;

    public CustomFormatter(IOptionsMonitor<CustomOptions> options)
        // Case insensitive
        : base("customName") =>
        (_optionsReloadToken, _formatterOptions) =
            (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

    private void ReloadLoggerOptions(CustomOptions options) =>
        _formatterOptions = options;
```

```
    public override void Write<TState>(
        in LogEntry<TState> logEntry,
        IExternalScopeProvider? scopeProvider,
        TextWriter textWriter)
    {
        string? message =
            logEntry.Formatter?.Invoke(
                logEntry.State, logEntry.Exception);

        if (message is null)
        {
            return;
        }

        CustomLogicGoesHere(textWriter);
        textWriter.WriteLine(message);
    }

    private void CustomLogicGoesHere(TextWriter textWriter)
    {
        textWriter.Write(_formatterOptions.CustomPrefix);
    }

    public void Dispose() => _optionsReloadToken?.Dispose();
}
```

앞의 `CustomFormatter.Write<TState>` API는 각 로그 메시지 주위에 래핑되는 텍스트를 지정합니다. 표준 `ConsoleFormatter`은 최소한 범위, 타임스탬프, 그리고 로그의 심각도 수준을 포함하고 관리할 수 있어야 합니다. 또한 로그 메시지에서 ANSI 색을 인코딩하고 원하는 들여쓰기도 제공할 수 있습니다. 구현에는 `CustomFormatter.Write<TState>` 이러한 기능이 부족합니다.

서식을 추가로 사용자 지정하는 방법은 네임스페이스의 기존 구현을 `Microsoft.Extensions.Logging.Console` 참조하세요.

- SimpleConsoleFormatter .
- SystemdConsoleFormatter
- JsonConsoleFormatter

사용자 지정 구성 옵션

로깅 확장성을 추가로 사용자 지정하려면 모든 ConsoleFormatterOptions에서 파생 클래스를 구성할 수 있습니다. 예를 들어 JSON 구성 공급자 를 사용하여 사용자 지정 옵션을 정의할 수 있습니다. 먼저 서브클래스를 정의합니다 ConsoleFormatterOptions .

C#

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.CustomWithConfig;
```

```
public sealed class CustomWrappingConsoleFormatterOptions : ConsoleFormatterOptions
{
    public string? CustomPrefix { get; set; }

    public string? CustomSuffix { get; set; }
}
```

앞의 콘솔 포맷터 옵션 클래스는 접두사와 접미사를 나타내는 두 개의 사용자 지정 속성을 정의합니다. 다음으로, 콘솔 포맷터 옵션을 구성할 *appsettings.json* 파일을 정의합니다.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        },
        "Console": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            },
            "FormatterName": "CustomTimePrefixingFormatter",
            "FormatterOptions": {
                "CustomPrefix": "|-<[",
                "CustomSuffix": "]>-|",
                "SingleLine": true,
                "IncludeScopes": true,
                "TimestampFormat": "HH:mm:ss.ffff ",
                "UseUtcTimestamp": true,
                "JsonWriterOptions": {
                    "Indented": true
                }
            }
        }
    },
    "AllowedHosts": "*"
}
```

이전 JSON 구성 파일에서 다음을 수행합니다.

- 노드는 `"Logging"`를 `"Console"`로 정의합니다.
- 노드 `"Console"` 에서 `"CustomTimePrefixingFormatter"`의 `"FormatterName"`을 지정하여 사용자 지정 포맷터에 매핑합니다.
- 노드는 `"FormatterOptions"`, `"CustomPrefix"`, `"CustomSuffix"`, 및 다른 몇 가지 파생 옵션을 정의합니다.

> **팁**
>
> `$.Logging.Console.FormatterOptions` JSON 경로는 예약되어 있으며, AddConsoleFormatter 확장 메서드를 사용하여 추가하면 사용자 지정 ConsoleFormatterOptions에 매핑됩니다. 이 기능은 사용 가능한 속성 외에도 사용자 지정 속성을 정의하는 기능을 제공합니다.

다음의 `CustomDatePrefixingFormatter`을 고려하십시오.

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.CustomWithConfig;

public sealed class CustomTimePrefixingFormatter : ConsoleFormatter, IDisposable
{
    private readonly IDisposable? _optionsReloadToken;
    private CustomWrappingConsoleFormatterOptions _formatterOptions;

    public CustomTimePrefixingFormatter(
        IOptionsMonitor<CustomWrappingConsoleFormatterOptions> options)
        // Case insensitive
        : base(nameof(CustomTimePrefixingFormatter))
    {
        _optionsReloadToken = options.OnChange(ReloadLoggerOptions);
        _formatterOptions = options.CurrentValue;
    }

    private void ReloadLoggerOptions(CustomWrappingConsoleFormatterOptions options)
=>
        _formatterOptions = options;

    public override void Write<TState>(
        in LogEntry<TState> logEntry,
        IExternalScopeProvider? scopeProvider,
        TextWriter textWriter)
    {
        string message =
            logEntry.Formatter(
                logEntry.State, logEntry.Exception);

        if (message == null)
        {
            return;
        }

        WritePrefix(textWriter);
```

```
        textWriter.Write(message);
        WriteSuffix(textWriter);
    }

    private void WritePrefix(TextWriter textWriter)
    {
        DateTime now = _formatterOptions.UseUtcTimestamp
            ? DateTime.UtcNow
            : DateTime.Now;

        textWriter.Write($"""
            {_formatterOptions.CustomPrefix}
{now.ToString(_formatterOptions.TimestampFormat)}
            """);
    }

    private void WriteSuffix(TextWriter textWriter) =>
        textWriter.WriteLine($" {_formatterOptions.CustomSuffix}");

    public void Dispose() => _optionsReloadToken?.Dispose();
}
```

이전 포맷터 구현에서:

- `CustomWrappingConsoleFormatterOptions` 변경 내용이 모니터링되고 그에 따라 업데이트됩니다.
- 작성된 메시지는 구성된 접두사 및 접미사로 래핑됩니다.
- 타임스탬프는 접두사 뒤가 아니라 구성된 ConsoleFormatterOptions.UseUtcTimestamp 값과 ConsoleFormatterOptions.TimestampFormat 값을 사용하여 메시지 앞에 추가됩니다.

사용자 지정 포맷터 구현과 함께 사용자 지정 구성 옵션을 사용하려면 호출 ConfigureLogging(IHostBuilder, Action<HostBuilderContext,ILoggingBuilder>)할 때 추가합니다.

C#

```
using Console.ExampleFormatters.CustomWithConfig;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddConsole()
    .AddConsoleFormatter<
        CustomTimePrefixingFormatter, CustomWrappingConsoleFormatterOptions>();
using IHost host = builder.Build();

ILoggerFactory loggerFactory = host.Services.GetRequiredService<ILoggerFactory>();
ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
```

```
using (logger.BeginScope("Logging scope"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("The .NET developer community happily welcomes you.");
}
```

다음 콘솔 출력은 이 `CustomTimePrefixingFormatter`출력을 사용할 때 예상되는 것과 유사합니다.

콘솔

```
|-<[ 15:03:15.6179 Hello World! ]>-|
|-<[ 15:03:15.6347 The .NET developer community happily welcomes you. ]>-|
```

사용자 지정 색 서식 구현

사용자 지정 로깅 포맷터에서 색 기능을 올바르게 활성화하려면, SimpleConsoleFormatterOptions를 확장하십시오. 이 포맷터에는 로그에서 색상을 사용할 수 있도록 도와주는 유용한 SimpleConsoleFormatterOptions.ColorBehavior 속성이 있습니다.

`CustomColorOptions`에서 파생된 `SimpleConsoleFormatterOptions`를 생성하세요.

C#

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.Custom;

public class CustomColorOptions : SimpleConsoleFormatterOptions
{
    public string? CustomPrefix { get; set; }
}
```

다음으로 형식이 지정된 로그 메시지 내에 ANSI 코딩된 색을 편리하게 포함할 수 있는 몇 가지 확장 메서드 `TextWriterExtensions` 를 클래스에 작성합니다.

C#

```
namespace Console.ExampleFormatters.Custom;

public static class TextWriterExtensions
{
    const string DefaultForegroundColor = "\x1B[39m\x1B[22m";
    const string DefaultBackgroundColor = "\x1B[49m";

    public static void WriteWithColor(
        this TextWriter textWriter,
        string message,
```

```
        ConsoleColor? background,
        ConsoleColor? foreground)
    {
        // Order:
        //   1. background color
        //   2. foreground color
        //   3. message
        //   4. reset foreground color
        //   5. reset background color

        var backgroundColor = background.HasValue ?
GetBackgroundColorEscapeCode(background.Value) : null;
        var foregroundColor = foreground.HasValue ?
GetForegroundColorEscapeCode(foreground.Value) : null;

        if (backgroundColor != null)
        {
            textWriter.Write(backgroundColor);
        }
        if (foregroundColor != null)
        {
            textWriter.Write(foregroundColor);
        }

        textWriter.WriteLine(message);

        if (foregroundColor != null)
        {
            textWriter.Write(DefaultForegroundColor);
        }
        if (backgroundColor != null)
        {
            textWriter.Write(DefaultBackgroundColor);
        }
    }

    static string GetForegroundColorEscapeCode(ConsoleColor color) =>
        color switch
        {
            ConsoleColor.Black => "\x1B[30m",
            ConsoleColor.DarkRed => "\x1B[31m",
            ConsoleColor.DarkGreen => "\x1B[32m",
            ConsoleColor.DarkYellow => "\x1B[33m",
            ConsoleColor.DarkBlue => "\x1B[34m",
            ConsoleColor.DarkMagenta => "\x1B[35m",
            ConsoleColor.DarkCyan => "\x1B[36m",
            ConsoleColor.Gray => "\x1B[37m",
            ConsoleColor.Red => "\x1B[1m\x1B[31m",
            ConsoleColor.Green => "\x1B[1m\x1B[32m",
            ConsoleColor.Yellow => "\x1B[1m\x1B[33m",
            ConsoleColor.Blue => "\x1B[1m\x1B[34m",
            ConsoleColor.Magenta => "\x1B[1m\x1B[35m",
            ConsoleColor.Cyan => "\x1B[1m\x1B[36m",
            ConsoleColor.White => "\x1B[1m\x1B[37m",
```

```
            _ => DefaultForegroundColor
        };

    static string GetBackgroundColorEscapeCode(ConsoleColor color) =>
        color switch
        {
            ConsoleColor.Black => "\x1B[40m",
            ConsoleColor.DarkRed => "\x1B[41m",
            ConsoleColor.DarkGreen => "\x1B[42m",
            ConsoleColor.DarkYellow => "\x1B[43m",
            ConsoleColor.DarkBlue => "\x1B[44m",
            ConsoleColor.DarkMagenta => "\x1B[45m",
            ConsoleColor.DarkCyan => "\x1B[46m",
            ConsoleColor.Gray => "\x1B[47m",

            _ => DefaultBackgroundColor
        };
}
```

사용자 지정 색 적용을 처리하는 사용자 지정 색 포맷터는 다음과 같이 정의할 수 있습니다.

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.Custom;

public sealed class CustomColorFormatter : ConsoleFormatter, IDisposable
{
    private readonly IDisposable? _optionsReloadToken;
    private CustomColorOptions _formatterOptions;

    private bool ConsoleColorFormattingEnabled =>
        _formatterOptions.ColorBehavior == LoggerColorBehavior.Enabled ||
        _formatterOptions.ColorBehavior == LoggerColorBehavior.Default &&
        System.Console.IsOutputRedirected == false;

    public CustomColorFormatter(IOptionsMonitor<CustomColorOptions> options)
        // Case insensitive
        : base("customName") =>
        (_optionsReloadToken, _formatterOptions) =
            (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

    private void ReloadLoggerOptions(CustomColorOptions options) =>
        _formatterOptions = options;

    public override void Write<TState>(
        in LogEntry<TState> logEntry,
        IExternalScopeProvider? scopeProvider,
        TextWriter textWriter)
    {
```

```
        if (logEntry.Exception is null)
        {
            return;
        }

        string? message =
            logEntry.Formatter?.Invoke(
                logEntry.State, logEntry.Exception);

        if (message is null)
        {
            return;
        }

        CustomLogicGoesHere(textWriter);
        textWriter.WriteLine(message);
    }

    private void CustomLogicGoesHere(TextWriter textWriter)
    {
        if (ConsoleColorFormattingEnabled)
        {
            textWriter.WriteWithColor(
                _formatterOptions.CustomPrefix ?? string.Empty,
                ConsoleColor.Black,
                ConsoleColor.Green);
        }
        else
        {
            textWriter.Write(_formatterOptions.CustomPrefix);
        }
    }

    public void Dispose() => _optionsReloadToken?.Dispose();
}
```

애플리케이션을 실행하면 `CustomPrefix`가 `FormatterOptions.ColorBehavior`일 때 로그에 메시지가 녹색으로 표시됩니다`Enabled`.

> **참고 항목**
>
> LoggerColorBehavior `Disabled`이(가) 있는 경우, 로그 메시지는 로그 메시지 내에 포함된 ANSI 색 코드를 *해석하지 않습니다*. 대신 원시 메시지를 출력합니다. 예를 들어 다음을 고려합니다.
>
> C#
>
> ```
> logger.LogInformation("Random log \x1B[42mwith green background\x1B[49m message");
> ```

그런 다음 문자 그대로의 문자열이 출력되고 색이 *지정되지 않습니다.*

출력

```
Random log \x1B[42mwith green background\x1B[49m message
```

참고하십시오

- .NET의 로깅
- .NET에서 사용자 지정 로깅 공급자 구현
- .NET의 고성능 로깅

Last updated on 2026. 02. 05.

.NET 일반 호스트

이 문서에서는 Microsoft.Extensions.Hosting NuGet 패키지에서 사용할 수 있는 .NET 제네릭 호스트를 구성하고 빌드하기 위한 다양한 패턴에 대해 알아봅니다. .NET 제네릭 호스트는 앱 시작 및 수명 관리를 담당합니다. 작업자 서비스 템플릿은 .NET 제네릭 호스트 HostApplicationBuilder를 만듭니다. 제네릭 호스트는 콘솔 앱과 같은 다른 유형의 .NET 애플리케이션과 함께 사용할 수 있습니다.

*호스트*는 앱의 리소스 및 수명 기능을 캡슐화하는 객체입니다:

- DI(종속성 주입)
- 로깅
- 구성
- 앱 종료
- `IHostedService` 구현

호스트가 시작될 때 서비스 컨테이너의 호스티드 서비스 컬렉션에 등록된 IHostedService.StartAsync의 각 구현에서 IHostedService을 호출합니다. 워커 서비스 앱에서 `IHostedService` 인스턴스를 포함하는 모든 BackgroundService 구현에는 BackgroundService.ExecuteAsync 메서드가 호출됩니다.

하나의 개체에 앱의 모든 상호 종속적 리소스를 포함하는 주요 원인은 수명 관리 즉, 앱 시작 및 종료에 대한 제어 때문입니다.

호스트 작성기 옵션

.NET은 제네릭 호스트를 구성하고 빌드하는 두 가지 방법을 제공합니다.

- IHostApplicationBuilder (`Host.CreateApplicationBuilder`): .NET 6에서 도입된 이 방법은 선형 속성 기반 구성 스타일을 사용합니다. 서비스, 구성 및 로깅은 작성기 개체(예 `builder.Services` `builder.Configuration`:)의 속성에 직접 액세스하여 구성됩니다. 이 방법은 새 프로젝트에 권장되며 현재 .NET 템플릿의 기본값입니다.

- IHostBuilder(`Host.CreateDefaultBuilder`): 연결된 확장 메서드(예 `ConfigureServices` `ConfigureAppConfiguration`:)를 통해 구성을 수행하는 기존의 콜백 기반 접근 방식입니다. 완전히 지원되지만 이 레거시 접근 방식은 기존 코드베이스와의 호환성을 유지하는 데 가장 적합합니다.

두 방법 모두 동일한 핵심 기능과 기본 동작을 제공합니다. 최신 .NET 패턴 및 더 간단한 구성 코드에 맞게 새 프로젝트를 선택합니다 `IHostApplicationBuilder` . 기존 애플리케이션을 유지 관리하거나 타사 라이브러리에 콜백 기반 패턴이 필요한 경우에 사용합니다 `IHostBuilder` .

호스트 설정

호스트는 일반적으로 `Program` 클래스의 코드로 구성, 빌드 및 실행됩니다. `Main` 메서드는 다음 작업을 수행합니다.

IHostApplicationBuilder

- CreateApplicationBuilder 메서드를 호출하여 작성기 개체를 만들고 구성합니다.
- Build()를 호출하여 IHost 인스턴스를 만듭니다.
- 호스트 개체에 대해 Run 또는 RunAsync 메서드를 호출합니다.

.NET 작업자 서비스 템플릿은 다음과 같은 코드를 생성하여 제네릭 호스트를 만듭니다.

C#

```
using Example.WorkerService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();

IHost host = builder.Build();
host.Run();
```

작업자 서비스에 대한 자세한 내용은 .NET의 작업자 서비스를 참조하세요.

호스트 작성기 설정

IHostApplicationBuilder

CreateApplicationBuilder 메서드는 다음 작업을 수행합니다.

- 콘텐츠 루트를 GetCurrentDirectory()에서 반환된 경로로 설정합니다.
- 다음에서 호스트 구성을 로드합니다.
  - 접두사가 `DOTNET_`인 환경 변수.
  - 명령줄 인수.
- 다음에서 앱 구성을 로드합니다.
  - *appsettings.json*.
  - *appsettings. {Environment}.json*.
  - 비밀 관리자: 앱이 `Development` 환경에서 실행되는 경우
  - 환경 변수입니다.
  - 명령줄 인수.
- 다음 로깅 공급자를 추가합니다.

    - 콘솔
    - 디버그
    - 이벤트소스 (EventSource)
    - EventLog(Windows에서 실행 중인 경우에만)
- 환경이 일 때 범위 유효성 검사 및 `Development`를 활성화합니다.

HostApplicationBuilder.Services는 Microsoft.Extensions.DependencyInjection.IServiceCollection 인스턴스입니다. 이러한 서비스는 등록된 서비스를 확인하기 위해 종속성 주입과 함께 사용되는 IServiceProvider를 빌드하는 데 사용됩니다.

프레임워크에서 제공하는 서비스

IHostBuilder.Build() 또는 HostApplicationBuilder.Build()를 호출하면 다음 서비스가 자동으로 등록됩니다.

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment

추가 시나리오 기반 호스트 작성기

웹용으로 빌드하거나 분산 애플리케이션을 빌드하는 경우 다른 호스트 작성기를 사용해야 할 수도 있습니다. 다음 추가 호스트 작성기 목록을 고려합니다.

- DistributedApplicationBuilder: 분산 앱을 만들기 위한 작성기입니다. 자세한 내용은 Aspire를 참조하세요.
- WebApplicationBuilder: 웹 애플리케이션 및 서비스용 작성기입니다. 자세한 내용은 ASP.NET Core 을 참조하세요.
- WebHostBuilder: `IWebHost`에 대한 작성기입니다. 자세한 내용은 ASP.NET Core 웹 호스트를 참조하세요.

`IHostApplicationLifetime`

IHostApplicationLifetime 서비스를 모든 클래스에 주입하여 시작 후 및 정상 종료 작업을 처리합니다. 인터페이스의 세 가지 속성은 앱 시작 및 앱 중지 이벤트 처리기 메서드를 등록하는 데 사용되는 취소 토큰입니다. 인터페이스에는 StopApplication() 메서드도 포함됩니다.

다음 예시는 IHostedService 이벤트를 등록하는 IHostedLifecycleService 및 `IHostApplicationLifetime` 구현입니다:

C#

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace AppLifetime.Example;

public sealed class ExampleHostedService : IHostedService, IHostedLifecycleService
{
    private readonly ILogger _logger;

    public ExampleHostedService(
        ILogger<ExampleHostedService> logger,
        IHostApplicationLifetime appLifetime)
    {
        _logger = logger;

        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);
    }

    Task IHostedLifecycleService.StartingAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("1. StartingAsync has been called.");

        return Task.CompletedTask;
    }

    Task IHostedService.StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("2. StartAsync has been called.");

        return Task.CompletedTask;
    }

    Task IHostedLifecycleService.StartedAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("3. StartedAsync has been called.");

        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("4. OnStarted has been called.");
    }

    private void OnStopping()
    {
        _logger.LogInformation("5. OnStopping has been called.");
    }

    Task IHostedLifecycleService.StoppingAsync(CancellationToken cancellationToken)
    {
```

```
        _logger.LogInformation("6. StoppingAsync has been called.");

        return Task.CompletedTask;
    }

    Task IHostedService.StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("7. StopAsync has been called.");

        return Task.CompletedTask;
    }

    Task IHostedLifecycleService.StoppedAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("8. StoppedAsync has been called.");

        return Task.CompletedTask;
    }

    private void OnStopped()
    {
        _logger.LogInformation("9. OnStopped has been called.");
    }
}
```

`ExampleHostedService` 구현을 추가하도록 작업자 서비스 템플릿을 수정할 수 있습니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using AppLifetime.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHostedService<ExampleHostedService>();
using IHost host = builder.Build();

await host.RunAsync();
```

애플리케이션은 다음 샘플 출력을 작성합니다.

C#

```
// Sample output:
//     info: AppLifetime.Example.ExampleHostedService[0]
//           1.StartingAsync has been called.
//     info: AppLifetime.Example.ExampleHostedService[0]
//           2.StartAsync has been called.
//     info: AppLifetime.Example.ExampleHostedService[0]
//           3.StartedAsync has been called.
//     info: AppLifetime.Example.ExampleHostedService[0]
```

```
//           4.OnStarted has been called.
//     info: Microsoft.Hosting.Lifetime[0]
//           Application started. Press Ctrl+C to shut down.
//     info: Microsoft.Hosting.Lifetime[0]
//           Hosting environment: Production
//     info: Microsoft.Hosting.Lifetime[0]
//           Content root path: ..\app-lifetime\bin\Debug\net8.0
//     info: AppLifetime.Example.ExampleHostedService[0]
//           5.OnStopping has been called.
//     info: Microsoft.Hosting.Lifetime[0]
//           Application is shutting down...
//     info: AppLifetime.Example.ExampleHostedService[0]
//           6.StoppingAsync has been called.
//     info: AppLifetime.Example.ExampleHostedService[0]
//           7.StopAsync has been called.
//     info: AppLifetime.Example.ExampleHostedService[0]
//           8.StoppedAsync has been called.
//     info: AppLifetime.Example.ExampleHostedService[0]
//           9.OnStopped has been called.
```

출력에는 다양한 수명 주기 이벤트의 순서가 모두 표시됩니다.

1. `IHostedLifecycleService.StartingAsync`
2. `IHostedService.StartAsync`
3. `IHostedLifecycleService.StartedAsync`
4. `IHostApplicationLifetime.ApplicationStarted`

예를 들어, `Ctrl` + `C`를 눌러 애플리케이션이 중지되면 다음 이벤트가 발생합니다.

1. `IHostApplicationLifetime.ApplicationStopping`
2. `IHostedLifecycleService.StoppingAsync`
3. `IHostedService.StopAsync`
4. `IHostedLifecycleService.StoppedAsync`
5. `IHostApplicationLifetime.ApplicationStopped`

`IHostLifetime`

IHostLifetime 구현은 호스트가 시작될 때와 중지될 때를 제어합니다. 등록된 마지막 구현이 사용됩니다. `Microsoft.Extensions.Hosting.Internal.ConsoleLifetime`은 기본 `IHostLifetime` 구현입니다. 종료의 수명 메커니즘에 대한 자세한 내용은 호스트 종료를 참조하세요.

`IHostLifetime` 인터페이스는 IHostLifetime.WaitForStartAsync 메서드를 노출하며, `IHost.StartAsync`이 시작될 때 호출되어 완료될 때까지 기다렸다가 계속 진행합니다. 이는 외부 이벤트에서 신호를 보낼 때까지 시작을 지연시키는 데 사용할 수 있습니다.

또한 `IHostLifetime` 인터페이스는 IHostLifetime.StopAsync에서 호출되어 호스트가 중지 중이며 종료할 때가 되었음을 나타내는 `IHost.StopAsync` 메서드를 노출합니다.

`IHostEnvironment`

IHostEnvironment 서비스를 클래스에 삽입하여 다음 설정에 대한 정보를 가져옵니다.

- IHostEnvironment.ApplicationName
- IHostEnvironment.ContentRootFileProvider
- IHostEnvironment.ContentRootPath
- IHostEnvironment.EnvironmentName

또한 `IHostEnvironment` 서비스는 다음 확장 메서드를 사용하여 환경을 평가하는 기능을 제공합니다.

- HostEnvironmentEnvExtensions.IsDevelopment
- HostEnvironmentEnvExtensions.IsEnvironment
- HostEnvironmentEnvExtensions.IsProduction
- HostEnvironmentEnvExtensions.IsStaging

호스트 구성

호스트 구성은 IHostEnvironment 구현의 속성을 구성하는 데 사용됩니다.

IHostApplicationBuilder

호스트 구성은 HostApplicationBuilderSettings.Configuration 속성에서 사용할 수 있으며 환경 구현은 IHostApplicationBuilder.Environment 속성에서 사용할 수 있습니다. 호스트를 구성하려면 `Configuration` 속성에 액세스하고 사용 가능한 확장 메서드를 호출합니다.

호스트 구성을 추가하려면 다음 예를 고려합니다.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilderSettings settings = new()
{
    Args = args,
    Configuration = new ConfigurationManager(),
    ContentRootPath = Directory.GetCurrentDirectory(),
};
```

```
settings.Configuration.AddJsonFile("hostsettings.json", optional: true);
settings.Configuration.AddEnvironmentVariables(prefix: "PREFIX_");
settings.Configuration.AddCommandLine(args);

HostApplicationBuilder builder = Host.CreateApplicationBuilder(settings);

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

앞의 코드가 하는 역할은 다음과 같습니다.

- 콘텐츠 루트를 GetCurrentDirectory()에서 반환된 경로로 설정합니다.
- 다음에서 호스트 구성을 로드합니다.
  - *hostsettings.json*.
  - 접두사가 `PREFIX_`인 환경 변수.
  - 명령줄 인수.

앱 구성

IHostApplicationBuilder

앱 구성은 공용 IHostApplicationBuilder.Configuration 속성을 통해 액세스됩니다. 이 속성을 사용하면 소비자가 사용 가능한 확장 메서드를 사용하여 기존 구성을 읽거나 변경할 수 있습니다.

자세한 내용은 .NET의 구성을 참조하세요.

호스트 종료

호스트된 프로세스를 중지하는 방법에는 여러 가지가 있습니다. 가장 일반적으로 호스트된 프로세스는 다음과 같은 방법으로 중지할 수 있습니다.

- 누군가 Run 또는 HostingAbstractionsHostExtensions.WaitForShutdown을 호출하지 않을 경우 앱은 보통 `Main`을 완료하고 종료됩니다.
- 앱이 크래시되는 경우
- 앱이 SIGKILL (또는 `Ctrl` + `Z`)를 사용하여 강제로 종료되는 경우.

호스팅 코드는 이러한 시나리오를 처리할 책임이 없습니다. 프로세스 소유자는 다른 앱과 동일하게 이를 처리해야 합니다. 호스티드 서비스 프로세스를 중지할 수 있는 다른 방법은 여러 가지

가 있습니다.

- `ConsoleLifetime`을 사용하는 경우(UseConsoleLifetime), 다음 신호를 수신하고 호스트를 정상적으로 중지하려고 시도합니다.
  - SIGINT (또는 `Ctrl` + `C`).
  - SIGQUIT (또는 Windows에서 CtrlBreak Unix의 Ctrl).
  - SIGTERM (`docker stop` 같은 다른 앱에서 보냄)
- 앱이 Environment.Exit을 호출할 경우

기본 제공된 호스팅 논리는 이러한 시나리오, 특히 `ConsoleLifetime` 클래스를 처리합니다. `ConsoleLifetime`은 애플리케이션의 정상 종료를 위해 '종료' 신호인 SIGINT, SIGQUIT 및 SIGTERM을 처리하려고 합니다.

.NET 6 이전에는 .NET 코드에서 SIGTERM을 정상적으로 처리할 수 없었습니다. 이 제한을 해결하기 위해 `ConsoleLifetime`은 System.AppDomain.ProcessExit을 구독합니다. `ProcessExit`가 발생하면 `ConsoleLifetime`은 `ProcessExit` 스레드를 중지하고 차단하도록 호스트에 신호를 보내 호스트가 중지되기를 기다립니다.

프로세스 종료 핸들러는 애플리케이션의 정리 코드(예: IHost.StopAsync과 HostingAbstractionsHostExtensions.Run 메서드에서 `Main` 이후의 코드)를 실행할 수 있도록 허용합니다.

그러나 SIGTERM이 `ProcessExit`가 발생하는 유일한 방법이 아니었기 때문에 이 방식에는 다른 문제가 있었습니다. SIGTERM은 앱 코드가 `Environment.Exit`를 호출할 때도 발생합니다. `Environment.Exit`은 `Microsoft.Extensions.Hosting` 앱 모델에서 프로세스를 종료하는 정상적인 방법이 아닙니다. `ProcessExit` 이벤트를 발생시키고 프로세스를 종료합니다. `Main` 메서드의 끝이 실행되지 않습니다. 백그라운드 및 포그라운드 스레드가 종료되고 `finally` 블록이 실행되지 *않습니다*.

`ConsoleLifetime`이 호스트가 종료되기를 기다리는 동안 `ProcessExit`를 차단했기 때문에 이 동작으로 인해 의 `Environment.Exit`이 `ProcessExit`에 대한 호출 대기도 차단됩니다. 또한, 시그템 처리가 프로세스를 정상적으로 종료하려고 시도했기 때문에 `ConsoleLifetime`은 ExitCode를 `0`으로 설정하고, 를 차단했습니다.

.NET 6에서는 POSIX 신호가 지원 및 처리됩니다. `ConsoleLifetime`은 SIGTERM을 적절하게 처리하고 `Environment.Exit`가 호출될 때 더 이상 관련되지 않습니다.

> **팁**
>
> .NET 6 이상에서는 `ConsoleLifetime`이 더 이상 `Environment.Exit` 시나리오를 처리하는 논리를 포함하지 않습니다. `Environment.Exit`을 호출하고 정리 논리를 수행해야 하는 앱은

`ProcessExit`을 구독할 수 있습니다. 이러한 시나리오에서는 호스팅이 더 이상 호스트를 정상적으로 중지하려고 시도하지 않습니다.

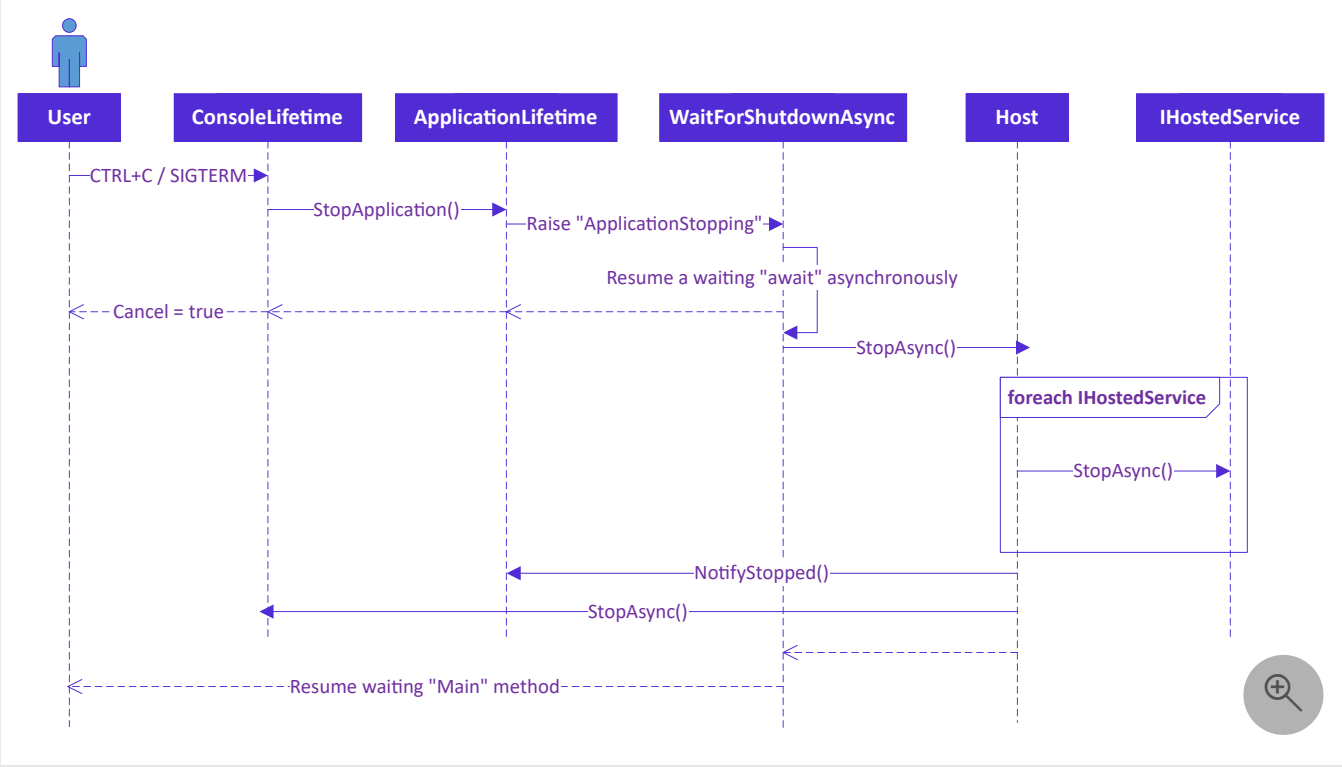
애플리케이션에서 호스팅을 사용하는 경우 호스트를 정상적으로 중지하려면 IHostApplicationLifetime.StopApplication 대신 `Environment.Exit`을 호출하면 됩니다.

호스팅 종료 프로세스

다음 시퀀스 다이어그램은 호스팅 코드에서 신호를 내부적으로 처리하는 방법을 보여 줍니다. 대부분의 사용자는 이 프로세스를 이해할 필요가 없습니다. 그러나 깊은 이해가 필요한 개발자의 경우 좋은 시각적 자료가 시작하는 데 도움이 될 수 있습니다.

호스트가 시작된 후 사용자가 `Run` 또는 `WaitForShutdown`을 호출하면 처리기가 IApplicationLifetime.ApplicationStopping에 등록됩니다. 실행이 `WaitForShutdown`에서 일시 중지되어 `ApplicationStopping` 이벤트가 발생할 때까지 기다립니다. `Main` 메서드는 즉시 반환되지 않으며 앱은 `Run` 또는 `WaitForShutdown`이 반환될 때까지 계속 실행됩니다.

프로세스에 신호를 보내면 다음 시퀀스가 시작됩니다.



1. 제어가 `ConsoleLifetime`에서 `ApplicationLifetime`으로 흘러서 `ApplicationStopping` 이벤트를 발생시킵니다. 이는 `WaitForShutdownAsync` 신호를 보내서 `Main` 실행 코드의 차단을 해제합니다. 그동안 이 POSIX 신호가 처리되었으므로 POSIX 신호 처리기는 `Cancel = true`를 반환합니다.
2. `Main` 실행 코드가 다시 실행되기 시작하고 호스트에 `StopAsync()`를 알리면 호스트되는 모든 서비스가 중지되고 다른 중지된 이벤트를 발생시킵니다.

3. 마지막으로 `WaitForShutdown`이 존재하므로 모든 애플리케이션이 코드 정리를 실행하고 `Main` 메서드를 정상적으로 종료합니다.

웹 서버 시나리오의 호스트 종료

HTTP/1.1 및 HTTP/2 프로토콜 모두에 대해 Kestrel에서 정상 종료가 작동하는 다양한 일반적인 시나리오와 트래픽을 원활하게 드레이닝하기 위해 부하 분산 장치를 사용하여 다양한 환경에서 이를 구성하는 방법이 있습니다. 웹 서버 구성은 이 문서의 범위를 벗어나지만 ASP.NET Core Kestrel 웹 서버에 대한 옵션 구성 설명서에서 자세한 내용을 확인할 수 있습니다.

호스트가 종료 신호(예: `Ctrl` + `C` 또는 `StopAsync`)를 수신하면, ApplicationStopping신호를 통해 애플리케이션에 알립니다. 정상적으로 완료해야 하는 장기 실행 작업이 있는 경우 이 이벤트를 구독해야 합니다.

다음으로 호스트는 구성할 수 있는 종료 시간 제한(기본값 30초)을 사용하여 IServer.StopAsync를 호출합니다. Kestrel(및 Http.Sys)은 포트 바인딩을 닫고 새 연결 수락을 중지합니다. 또한 현재 연결에 새 요청 처리를 중지하라고 지시합니다. HTTP/2 및 HTTP/3의 경우 예비 `GOAWAY` 메시지가 클라이언트로 전송됩니다. HTTP/1.1의 경우 요청이 순서대로 처리되므로 연결 루프를 중지합니다. IIS는 503 상태 코드가 있는 새 요청을 거부하여 다르게 동작합니다.

활성 요청은 종료 시간 제한까지 완료되어야 합니다. 시간 제한 전에 모두 완료되면 서버는 더 빨리 제어권을 호스트에 반환합니다. 제한 시간이 만료되면 보류 중인 연결과 요청이 강제로 중단되어 로그와 클라이언트에 오류가 발생할 수 있습니다.

부하 분산 장치 고려 사항

부하 분산 장치 작업 시 클라이언트를 새 대상으로 원활하게 전환하려면 다음 단계를 따릅니다.

- 새 인스턴스를 가져와 트래픽 밸런싱을 시작합니다(크기 조정 목적으로 이미 여러 인스턴스가 있을 수 있음).
- 부하 분산 장치 구성에서 이전 인스턴스를 사용하지 않도록 설정하거나 제거하여 새 트래픽 수신을 중지합니다.
- 이전 인스턴스에 종료 신호를 보냅니다.
- 드레이닝되거나 시간이 초과될 때까지 기다립니다.

참고 항목

- .NET에서 종속성 주입
- .NET의 로깅
- .NET의 구성
- .NET의 Worker Services

- ASP.NET Core 웹 호스트
- ASP.NET Core Kestrel 웹 서버 구성
- 제네릭 호스트 버그는 github.com/dotnet/runtime 리포지토리에 만들어져야 합니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 04.

비동기 상태 관리

NuGet 패키지는 Microsoft.Extensions.AsyncState 현재 비동기 컨텍스트 내에서 개체를 저장하고 검색하는 기능을 제공합니다. 이 패키지는 특히 비동기 작업에서 여러 개체를 공유해야 하는 경우 직접 사용에 AsyncLocal<T> 비해 성능 및 유용성 향상을 제공합니다.

AsyncState를 사용하는 이유

.NET 비동기 컨텍스트에서 앰비언트 데이터를 관리하기 위한 `AsyncLocal<T>` 제공하지만 직접 사용하면 단점이 있을 수 있습니다.

- **성능**: 각 `AsyncLocal<T>` 인스턴스는 오버헤드를 추가합니다. 여러 개체가 비동기 컨텍스트를 통과해야 하는 경우 많은 `AsyncLocal<T>` 인스턴스를 관리하면 성능에 영향을 미칠 수 있습니다.
- **추상화**: `AsyncLocal<T>`을(를) 직접 사용하여 코드를 특정 구현에 묶어두면, 나중에 최적화하거나 변경하기가 더 어려워집니다.
- **수명 관리**: AsyncState 패키지는 명시적 API를 통해 앰비언트 데이터의 수명을 더 잘 제어할 수 있습니다.

패키지는 `Microsoft.Extensions.AsyncState` 다음을 제공하여 이러한 문제를 해결합니다.

- 필요한 인스턴스 수를 `AsyncLocal<T>` 줄이는 최적화된 구현입니다.
- 앰비언트 데이터를 저장하고 검색하기 위한 정리 추상화입니다.
- 종속성 주입과 통합하여 더 쉽게 테스트하고 구성합니다.

시작하기

시작하려면 비동기 상태 관리를 위해 Microsoft.Extensions.AsyncState NuGet 패키지를 설치하십시오.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.AsyncState
```

자세한 내용은 dotnet add package 또는 .NET 애플리케이션의 패키지 종속성 관리 참조하세요.

비동기 상태 서비스 등록

확장 메서드를 사용하여 AddAsyncState 종속성 주입 컨테이너에 비동기 상태 서비스를 등록합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();

services.AddAsyncState();

ServiceProvider provider = services.BuildServiceProvider();
```

이 등록을 통해 IAsyncContext<T> 및 IAsyncState 인터페이스를 종속성 주입에 사용할 수 있습니다.

IAsyncContext 사용

인터페이스는 IAsyncContext<T> 현재 비동기 컨텍스트에서 값을 가져와서 설정하는 메서드를 제공합니다.

C#

```
var context = provider.GetRequiredService<IAsyncContext<UserContext>>();

// Set a value in the async context
var userContext = new UserContext { UserId = "12345", UserName = "Alice" };
context.Set(userContext);

// Retrieve the value
if (context.TryGet(out var retrievedContext))
{
    Console.WriteLine($"User: {retrievedContext.UserName}");
}
```

C#

```
public record class UserContext
{
    public required string UserId { get; set; }
    public required string UserName { get; set; }
}
```

컨텍스트에 설정된 값은 비동기 작업을 통해 전달되므로 동일한 비동기 컨텍스트 내에서 실행되는 모든 코드에서 사용할 수 있습니다.

IAsyncState 사용

IAsyncState 인터페이스는 `IAsyncContext<T>`이(가) 확장하는 기본 수명 주기 인터페이스입니다. 비동기 상태 수명 주기를 관리하기 위한 `Initialize()` 및 `Reset()` 메서드를 제공합니다. 비동기 상태 값을 형식화된 방식으로 액세스하려면 대신 `IAsyncContext<T>`를 사용합니다.

실제 예제: 요청 상관 관계

비동기 상태의 일반적인 사용 사례는 HTTP 요청에서 상관 관계 정보를 유지하는 것입니다.

C#

```
public class RequestProcessor(IAsyncContext<CorrelationContext> asyncContext)
{

    public async Task ProcessRequestAsync(string correlationId)
    {
        // Set correlation context at the beginning of request processing
        asyncContext.Set(new CorrelationContext { CorrelationId = correlationId });

        // The correlation ID flows through all async operations
        await Step1Async();
        await Step2Async();
    }

    private async Task Step1Async()
    {
        await Task.Yield();

        if (asyncContext.TryGet(out var context))
        {
            Console.WriteLine($"Step 1 - Correlation ID: {context.CorrelationId}");
        }
    }

    private async Task Step2Async()
    {
        await Task.Yield();

        if (asyncContext.TryGet(out var context))
        {
            Console.WriteLine($"Step 2 - Correlation ID: {context.CorrelationId}");
        }
    }
}
```

C#

```
public class CorrelationContext
{
    public required string CorrelationId { get; set; }
}
```

이 예제에서 상관 관계 ID는 요청 처리 시작 시 한 번 설정되며 매개 변수로 전달하지 않고도 모든 후속 비동기 작업에서 자동으로 사용할 수 있습니다.

ASP.NET Core 통합

ASP.NET Core 애플리케이션에서는 비동기 상태를 사용하여 애플리케이션을 통해 요청 관련 정보를 흐를 수 있습니다.

C#

```
public class RequestHandler(IAsyncContext<RequestMetadata> asyncContext)
{

    public async Task<object> GetDataAsync()
    {
        await Task.Yield();

        if (asyncContext.TryGet(out var metadata))
        {
            var duration = DateTimeOffset.UtcNow - metadata.StartTime;
            return new
            {
                RequestId = metadata.RequestId,
                RequestPath = metadata.RequestPath,
                Duration = duration.TotalMilliseconds
            };
        }

        return new { Error = "No request metadata available" };
    }
}
```

C#

```
public record class RequestMetadata
{
    public required string RequestId { get; set; }
    public required string RequestPath { get; set; }
    public DateTimeOffset StartTime { get; set; }
}
```

모범 사례

비동기 상태를 사용하는 경우 다음 모범 사례를 고려합니다.

- **상태 크기 제한**: 메모리 오버헤드를 줄이고 성능을 유지하기 위해 비동기 상태 개체를 작게 유지합니다.
- **초기화 상태**: 모든 다운스트림 비동기 작업에서 사용할 수 있도록 가능한 한 빨리 비동기 상태 값을 설정합니다.
- **정리 상태**: 장기 실행 애플리케이션에서 메모리 누수 방지를 위해 더 이상 필요하지 않은 경우 비동기 상태를 다시 설정하거나 지웁니다.
- **적절한 인터페이스 사용**: 현재 비동기 컨텍스트 내에서 형식화된 값을 가져와서 설정하는 데 사용합니다 `IAsyncContext<T>` . 초기화를 직접 관리하고 수명 주기를 다시 설정해야 하는 경우에 사용합니다 `IAsyncState` .
- **타입 안전성**: 제네릭 사전을 사용하여 형식 보안을 유지하고 코드 clarity 개선하는 대신 특정 컨텍스트 형식을 만듭니다.

참고하십시오

- .NET의 종속성 주입
- AsyncLocal<T>
- ASP.NET Core 미들웨어

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

FakeTimeProvider를 사용하여 테스트

NuGet 패키지는 Microsoft.Extensions.TimeProvider.Testing 시간에 따라 달라지는 코드를 결정적으로 테스트할 수 있는 클래스를 제공합니다 `FakeTimeProvider` . 이 가짜 구현을 사용하면 테스트 내에서 시스템 시간을 제어하여 예측 가능하고 반복 가능한 결과를 보장할 수 있습니다.

FakeTimeProvider를 사용하는 이유

현재 시간에 따라 달라지거나 타이머를 사용하는 코드를 테스트하는 것은 어려울 수 있습니다.

- **비결정적 테스트**: 실시간에 의존하는 테스트는 일관되지 않은 결과를 생성할 수 있습니다.
- **느린 테스트**: 실제 통과 시간을 기다려야 하는 테스트는 테스트 실행 속도가 상당히 느려질 수 있습니다.
- **경합 조건**: 시간 종속 논리는 재현하기 어려운 경합 조건을 도입할 수 있습니다.
- **엣지 케이스**: 특정 시간(예: 자정 또는 월 경계)에서 시간 기반 논리를 테스트하는 것은 실제 시간으로는 어렵습니다.

다음과 같이 `FakeTimeProvider` 을 통해 이러한 문제에 대처합니다.

- 현재 시간에 대한 완전한 제어를 제공합니다.
- 기다리지 않고 즉시 시간을 진행할 수 있습니다.
- 시간 기반 동작의 결정적 테스트를 사용하도록 설정합니다.
- 에지 사례 및 경계 조건을 쉽게 테스트할 수 있도록 합니다.

시작하기

`FakeTimeProvider`를 시작하려면 Microsoft.Extensions.TimeProvider.Testing NuGet 패키지를 설치하세요.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.TimeProvider.Testing
```

자세한 내용은 dotnet add package 또는 .NET 애플리케이션에서 패키지 종속성 관리를 참조하세요.

기본 사용법

`FakeTimeProvider`는 테스트에 제어 가능한 시간을 제공하기 위해 TimeProvider를 확장합니다.

C#

```
var fakeTimeProvider = new FakeTimeProvider();

// Get the current time (defaults to January 1, 2000, midnight UTC).
Console.WriteLine($"Start time: {fakeTimeProvider.GetUtcNow()}");
```

특정 시간에 맞춰 초기화

특정 시작 시간으로 초기화 `FakeTimeProvider` 할 수 있습니다.

C#

```
DateTimeOffset startTime = new(2025, 10, 20, 12, 0, 0, TimeSpan.Zero);
fakeTimeProvider = new FakeTimeProvider(startTime);

Console.WriteLine($"Started at: {fakeTimeProvider.GetUtcNow()}");
```

사전 시간

메서드는 `Advance` 지정된 기간 동안 시간을 앞으로 이동합니다.

C#

```
// Advance time by 30 minutes.
fakeTimeProvider.Advance(TimeSpan.FromMinutes(30));
Console.WriteLine($"After advancing 30 minutes: {fakeTimeProvider.GetUtcNow()}");
```

표준 시간대 구성

가짜 시간 공급자에 대한 현지 표준 시간대를 설정합니다.

C#

```
var timeZoneId = OperatingSystem.IsWindows() ? "Pacific Standard Time" :
"America/Los_Angeles";
var pacificTimeZone = TimeZoneInfo.FindSystemTimeZoneById(timeZoneId);
fakeTimeProvider.SetLocalTimeZone(pacificTimeZone);
```

```
var localTime = fakeTimeProvider.GetLocalNow();
Console.WriteLine($"Local time: {localTime}");
```

지연된 작업 테스트

`FakeTimeProvider` 는 지연이 포함된 작업을 테스트하는 데 특히 유용합니다.

C#

```
public class DelayedOperationTests
{
    [Fact]
    public async Task DelayedOperation_CompletesAfterDelay()
    {
        // Arrange
        var fakeTimeProvider = new FakeTimeProvider();
        var operation = new DelayedOperation(fakeTimeProvider);

        // Act
        Task task = operation.ExecuteAsync(TimeSpan.FromMinutes(5));

        // Assert - operation should not be complete yet
        Assert.False(task.IsCompleted);

        // Advance time by 5 minutes
        fakeTimeProvider.Advance(TimeSpan.FromMinutes(5));

        // Wait for the task to complete
        await task;

        // Operation should now be complete
        Assert.True(task.IsCompleted);
    }
}

public class DelayedOperation(TimeProvider timeProvider)
{
    public async Task ExecuteAsync(TimeSpan delay)
    {
        await Task.Delay(delay, timeProvider);
    }
}
```

주기적 작업 테스트

타이머를 사용하여 주기적으로 실행되는 테스트 작업:

C#

```
public class PeriodicOperationTests
{
    [Fact]
    public void PeriodicOperation_ExecutesAtIntervals()
    {
        // Arrange
        var fakeTimeProvider = new FakeTimeProvider();
        var counter = new PeriodicCounter(fakeTimeProvider);
        counter.Start(TimeSpan.FromSeconds(10));

        // Act & Assert
        Assert.Equal(0, counter.Count);

        // Advance by 10 seconds
        fakeTimeProvider.Advance(TimeSpan.FromSeconds(10));
        Assert.Equal(1, counter.Count);

        // Advance by 20 more seconds
        fakeTimeProvider.Advance(TimeSpan.FromSeconds(20));
        Assert.Equal(3, counter.Count);

        // Clean up
        counter.Stop();
    }
}

public class PeriodicCounter(TimeProvider timeProvider)
{
    private ITimer? _timer;

    public int Count { get; private set; }

    public void Start(TimeSpan interval)
    {
        _timer = timeProvider.CreateTimer(
            callback: _ => Count++,
            state: null,
            dueTime: interval,
            period: interval);
    }

    public void Stop()
    {
        _timer?.Dispose();
    }
}
```

시간 기반 비즈니스 논리 테스트

특정 시간 또는 날짜에 따라 달라지는 비즈니스 논리를 테스트합니다.

C#

```
public class SubscriptionTests
{
    [Fact]
    public void Subscription_ExpiresAfterOneYear()
    {
        // Arrange
        var fakeTimeProvider = new FakeTimeProvider();
        var startDate = new DateTimeOffset(2025, 1, 1, 0, 0, 0, TimeSpan.Zero);
        fakeTimeProvider.SetUtcNow(startDate);

        var subscription = new Subscription(fakeTimeProvider);
        subscription.Activate();

        // Assert - subscription is active
        Assert.True(subscription.IsActive);

        // Act - advance time by 11 months
        fakeTimeProvider.Advance(TimeSpan.FromDays(30 * 11));
        Assert.True(subscription.IsActive);

        // Advance time by 2 more months
        fakeTimeProvider.Advance(TimeSpan.FromDays(60));
        Assert.False(subscription.IsActive);
    }
}

public class Subscription(TimeProvider timeProvider)
{
    private DateTimeOffset _activationDate;

    public void Activate()
    {
        _activationDate = timeProvider.GetUtcNow();
    }

    public bool IsActive
    {
        get
        {
            DateTimeOffset currentTime = timeProvider.GetUtcNow();
            DateTimeOffset expirationDate = _activationDate.AddYears(1);
            return currentTime < expirationDate;
        }
    }
}
```

종속성 주입과의 통합

종속성 주입에 등록된 서비스의 테스트에 `FakeTimeProvider`를 사용합니다.

C#

```
public class CacheServiceTests
{
    [Fact]
    public void Cache_ExpiresAfterTimeout()
    {
        // Arrange
        var fakeTimeProvider = new FakeTimeProvider();

        var services = new ServiceCollection();
        services.AddSingleton<TimeProvider>(fakeTimeProvider);
        services.AddSingleton<CacheService>();

        ServiceProvider provider = services.BuildServiceProvider();
        CacheService cache = provider.GetRequiredService<CacheService>();

        // Act
        cache.Set("key", "value", TimeSpan.FromMinutes(10));

        // Assert - value is present
        Assert.True(cache.TryGet("key", out string? value));
        Assert.Equal("value", value);

        // Advance time beyond expiration
        fakeTimeProvider.Advance(TimeSpan.FromMinutes(11));

        // Value should be expired
        Assert.False(cache.TryGet("key", out _));
    }
}

public class CacheService(TimeProvider timeProvider)
{
    private readonly Dictionary<string, CacheEntry> _cache = [];

    public void Set(string key, string value, TimeSpan expiration)
    {
        DateTimeOffset expiresAt = timeProvider.GetUtcNow() + expiration;
        _cache[key] = new CacheEntry(value, expiresAt);
    }

    public bool TryGet(string key, out string? value)
    {
        if (_cache.TryGetValue(key, out CacheEntry? entry))
        {
            if (timeProvider.GetUtcNow() < entry.ExpiresAt)
            {
                value = entry.Value;
                return true;
            }

            // Entry expired, remove it
            _cache.Remove(key);
        }
```

```
        value = null;
        return false;
    }

    private record CacheEntry(string Value, DateTimeOffset ExpiresAt);
}
```

모범 사례

사용하는 `FakeTimeProvider` 경우 다음 모범 사례를 고려합니다.

- **TimeProvider 삽입**: 항상 `TimeProvider` 이나 DateTime를 직접 사용하는 대신 DateTimeOffset을(를) 종속성으로 삽입해야 합니다. 이렇게 하면 코드를 테스트할 수 있습니다.
- **UTC 시간 사용**: 비즈니스 논리에서 UTC 시간으로 작업하고 표시에 필요한 경우에만 현지 시간으로 변환합니다.
- **경계 사례 테스트**: `FakeTimeProvider` 를 사용하여 자정, 월 경계, 일광 절약 시간 전환, 윤년과 같은 경계 사례를 테스트합니다.
- **타이머 정리**: 테스트에서 리소스가 누출되는 것을 방지하기 위해 만든 `CreateTimer` 타이머를 삭제합니다.
- **의도적으로 시간을 진행시키기**: 테스트의 동작을 명확하고 예측 가능하게 하기 위해 테스트에서 시간을 명시적으로 진행시키십시오.
- **실시간 및 가짜 시간을 혼합**하지 마세요. 이로 인해 예측 불가능한 동작으로 `TimeProvider.System` 이어질 수 있으므로, 동일한 테스트에서 실제 `FakeTimeProvider` 와 혼합하지 마세요.

참고하십시오

- TimeProvider
- .NET의 단위 테스트
- .NET에서 종속성 주입

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 02.

개인 정보 보호 및 규정 준수에 대한 감사 보고서

NuGet 패키지는 Microsoft.Extensions.AuditReports 컴파일되는 코드에 대한 감사 보고서를 생성하는 기능을 제공합니다. 이러한 보고서는 개인 정보 감사, 규정 준수 검토 및 애플리케이션이 수집하고 전송하는 원격 분석 데이터를 이해하는 데 특히 유용합니다.

감사 보고서를 사용하는 이유

감사 보고서는 조직이 규정 준수 및 투명성을 유지하는 데 도움이 됩니다.

- **개인 정보 준수: 개인** 정보 보호에 중요한 데이터에 액세스하거나 기록되는 모든 위치를 식별합니다.
- **원격 분석 추적**: 애플리케이션에서 생성하는 메트릭 및 원격 분석을 이해합니다.
- **코드 검토**: 코드베이스 전체에서 데이터 분류 사용량을 검토합니다.
- **규정 준수 감사**: 규정 준수 및 보안 감사에 대한 설명서를 제공합니다.
- **데이터 거버넌스**: 데이터 처리 관행이 조직 정책과 일치하는지 확인합니다.

시작하기

`Microsoft.Extensions.AuditReports` 패키지는 컴파일 시 보고서를 생성하는 빌드 타임 도구입니다. 개발 종속성으로 설치합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.AuditReports
```

자세한 내용은 dotnet add package 또는 .NET 애플리케이션의 패키지 종속성 관리 참조하세요.

보고서 형식

패키지는 세 가지 유형의 보고서를 생성할 수 있습니다.

테이블 확장

보고서 유형	Description
Metrics	애플리케이션에서 내보내는 메트릭을 이해할 수 있도록 코드에 사용되는 소스 생성 메트릭 정의에 대한 보고서를 생성합니다.
규정 준수	개인 정보 또는 중요한 정보를 처리하는 원본 생성 로깅 방법을 포함하여 개인 정보 보호에 중요한 데이터의 사용에 대한 보고서를 생성합니다.
메타데이터	메트릭과 규정 준수 정보를 모두 결합하는 포괄적인 보고서를 생성합니다.

보고서 생성 구성

project 파일에서 MSBuild 속성을 설정하여 보고서 생성을 구성합니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>

    <!-- Enable audit report generation -->
    <GenerateComplianceReport>true</GenerateComplianceReport>

    <!-- Specify report output path (optional) -->
    <ComplianceReportOutputPath>$(OutputPath)compliance-
report.json</ComplianceReportOutputPath>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.AuditReports" Version="10.0.0"
/>
  </ItemGroup>

</Project>
```

준수 보고서 생성

준수 보고서를 생성하려면 속성을 `GenerateComplianceReport` `true`로 설정합니다.

XML

```xml
<PropertyGroup>
  <GenerateComplianceReport>true</GenerateComplianceReport>
</PropertyGroup>
```

이 보고서는 특히 로깅 작업에서 개인 정보 보호에 중요한 데이터를 처리하는 코드를 식별합니다.

메트릭 보고서 생성

메트릭 보고서를 생성하려면 `GenerateMetricsReport` 속성을 `true`으로 설정하십시오.

XML

```
<PropertyGroup>
  <GenerateMetricsReport>true</GenerateMetricsReport>
  <MetricsReportOutputPath>$(OutputPath)metrics-
report.json</MetricsReportOutputPath>
</PropertyGroup>
```

이 보고서는 애플리케이션에서 생성된 모든 메트릭을 문서화합니다.

메타데이터 보고서 생성

규정 준수 및 메트릭 정보를 모두 포함하는 포괄적인 보고서의 경우:

XML

```
<PropertyGroup>
  <GenerateMetadataReport>true</GenerateMetadataReport>
  <MetadataReportOutputPath>$(OutputPath)metadata-
report.json</MetadataReportOutputPath>
</PropertyGroup>
```

예: 준수 보고서 출력

규정 준수 보고를 사용하도록 설정된 project 빌드하는 경우 개인 정보 보호에 중요한 데이터 사용을 식별하는 JSON 파일을 가져옵니다.

JSON

```
{
  "version": "1.0",
  "reportType": "compliance",
  "generatedAt": "2025-10-20T12:00:00Z",
  "entries": [
    {
      "filePath": "Services/UserService.cs",
      "lineNumber": 42,
      "memberName": "LogUserActivity",
```

```
      "dataClassification": "PersonalData",
      "message": "Logs user email address"
    },
    {
      "filePath": "Controllers/AccountController.cs",
      "lineNumber": 88,
      "memberName": "LogLoginAttempt",
      "dataClassification": "AuthenticationData",
      "message": "Logs authentication attempt with username"
    }
  ]
}
```

데이터 분류와 함께 사용

감사 보고서는 다음의 데이터 분류 특성 `Microsoft.Extensions.Compliance.Abstractions`과 함께 작동합니다.

C#

```
using Microsoft.Extensions.Compliance.Classification;
using Microsoft.Extensions.Logging;

public class UserService
{
    private readonly ILogger<UserService> _logger;

    public UserService(ILogger<UserService> logger)
    {
        _logger = logger;
    }

    [LoggerMessage(Level = LogLevel.Information, Message = "User {Email} logged
in")]
    public partial void LogUserLogin(
        [PrivateData] string email);
}
```

규정 준수 보고가 활성화된 상태에서 이 코드를 빌드하면, 보고서가 `email`를 `PrivateData`로 분류함을 식별합니다.

보고서 출력 위치 구성

감사 보고서에 대한 사용자 지정 경로를 지정합니다.

XML

```
<PropertyGroup>
  <!-- Generate all report types -->
  <GenerateComplianceReport>true</GenerateComplianceReport>
  <GenerateMetricsReport>true</GenerateMetricsReport>

  <!-- Custom output locations -->

<ComplianceReportOutputPath>$(OutputPath)audit\compliance.json</ComplianceReportOut
putPath>

<MetricsReportOutputPath>$(OutputPath)audit\metrics.json</MetricsReportOutputPath>
</PropertyGroup>
```

CI/CD 파이프라인과 통합

감사 보고서는 자동화된 규정 준수 검사를 위해 CI/CD 파이프라인에 통합될 수 있습니다.

YAML

```yaml
# Example GitHub Actions workflow
name: Compliance Check

on:
  pull_request:
    branches: [ main ]

jobs:
  audit:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Setup .NET
      uses: actions/setup-dotnet@v3
      with:
        dotnet-version: '10.0.x'

    - name: Build with compliance report
      run: dotnet build -p:GenerateComplianceReport=true

    - name: Upload compliance report
      uses: actions/upload-artifact@v3
      with:
        name: compliance-report
        path: '**/compliance-report.json'

    - name: Analyze compliance report
      run: |
        # Add script to analyze the compliance report
```

```
        # and fail the build if violations are found
        ./scripts/check-compliance.sh
```

예: 메트릭 보고서 출력

메트릭 보고서는 애플리케이션에서 생성하는 메트릭을 문서화합니다.

JSON

```json
{
  "version": "1.0",
  "reportType": "metrics",
  "generatedAt": "2025-10-20T12:00:00Z",
  "metrics": [
    {
      "name": "http_request_duration",
      "description": "HTTP request duration in milliseconds",
      "unit": "milliseconds",
      "type": "histogram",
      "tags": ["endpoint", "method", "status_code"]
    },
    {
      "name": "active_connections",
      "description": "Number of active connections",
      "unit": "connections",
      "type": "gauge",
      "tags": ["connection_type"]
    }
  ]
}
```

실제 예제: 개인 정보 감사 워크플로

다음은 개인 정보 감사를 설정하는 방법을 보여 주는 전체 예제입니다.

Project 파일 구성:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>

    <!-- Enable compliance reporting -->
    <GenerateComplianceReport>true</GenerateComplianceReport>

<ComplianceReportOutputPath>$(OutputPath)audit\compliance.json</ComplianceReportOut
```

```
putPath>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.AuditReports" Version="10.0.0"
/>
    <PackageReference Include="Microsoft.Extensions.Compliance.Abstractions"
Version="10.0.0" />
    <PackageReference Include="Microsoft.Extensions.Telemetry.Abstractions"
Version="10.0.0" />
  </ItemGroup>

</Project>
```

데이터 분류를 사용하여 코드:

C#

```
using Microsoft.Extensions.Compliance.Classification;
using Microsoft.Extensions.Logging;

public partial class OrderService(ILogger<OrderService> logger)
{

    [LoggerMessage(Level = LogLevel.Information, Message = "Order created for
customer {CustomerId}")]
    public partial void LogOrderCreated(
        [PublicData] string customerId);

    [LoggerMessage(Level = LogLevel.Information, Message = "Payment processed for
{CardNumber}")]
    public partial void LogPaymentProcessed(
        [PrivateData] string cardNumber);
}
```

이 project 빌드할 때 규정 준수 보고서는 `cardNumber` 개인 정보 보호에 민감한 로깅을 식별합니다.

모범 사례

감사 보고서를 사용하는 경우 다음 모범 사례를 고려합니다.

- **조기 통합**: 개발 초기에 프로젝트에 감사 보고를 추가하여 개인 정보 문제를 더 빨리 파악합니다.
- **검토 자동화**: 지속적인 규정 준수 모니터링을 위해 감사 보고서 생성을 CI/CD 파이프라인에 통합합니다.
- **정기적으로 검토**: 코드 검토 중 및 릴리스 전에 감사 보고서를 정기적으로 검토합니다.
- **데이터 분류**: 데이터 분류 특성을 일관되게 사용하여 정확한 감사 보고서를 보장합니다.

- **저장소 보고서**: 규정 준수 설명서 및 기록 추적에 대한 감사 보고서를 보관합니다.
- **버전 제어**: 시간이 지남에 따라 감사 보고서의 변경 내용을 추적하여 데이터 처리가 어떻게 진화하는지 이해합니다.
- **보안 검사**: 보안 및 개인 정보 검사 도구에 대한 입력으로 감사 보고서를 사용합니다.

참고하십시오

- .NET의 데이터 분류
- .NET에서의 로깅
- .NET의 고성능 로그 기록
- 컴파일 시간 로깅 원본 생성

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

복원력 있는 앱 개발 소개

복원력은 일시적인 오류로부터 복구하고 계속 작동하는 앱의 기능입니다. .NET 프로그래밍의 컨텍스트에서 복원력은 오류를 정상적으로 처리하고 신속하게 복구할 수 있는 앱을 디자인하여 달성됩니다. .NET에서 복원력 있는 앱을 빌드하는 데 도움이 되도록 NuGet에서 다음 두 패키지를 사용할 수 있습니다.

테이블 확장

NuGet 패키지	설명
Microsoft.Extensions.Resilience	이 NuGet 패키지는 일시적인 오류에 대해 앱을 강화하는 메커니즘을 제공합니다.
Microsoft.Extensions.Http.Resilience	이 NuGet 패키지는 HttpClient 클래스에 대한 복원력 메커니즘을 제공합니다.

이 두 NuGet 패키지는 인기 있는 오픈 소스 프로젝트인 Polly를 기반으로 빌드됩니다. Polly는 개발자가 재시도, 회로 차단기, 시간 제한, 격벽 격리, 속도 제한, 대체 및 헤징과 같은 전략을 유창하고 스레드로부터 안전한 방식으로 표현할 수 있는 .NET 복원력 및 일시적인 오류 처리 라이브러리입니다.

> **Important**
>
> Microsoft.Extensions.Http.Polly NuGet 패키지는 더 이상 사용되지 않습니다. 앞에서 언급한 패키지 중 하나를 대신 사용합니다.

시작하기

.NET에서 복원력을 시작하려면 Microsoft.Extensions.Resilience NuGet 패키지를 설치합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Resilience
```

자세한 내용은 .NET 애플리케이션에서 dotnet 패키지 추가 또는 패키지 종속성 관리를 참조하세요.

복원력 파이프라인 빌드

복원력을 사용하려면 먼저 복원력 기반 전략의 파이프라인을 빌드해야 합니다. 구성된 각 전략은 구성 순서대로 실행됩니다. 즉, 순서가 중요합니다. 진입점은 IServiceCollection 형식의 `AddResiliencePipeline` 확장 메서드입니다. 이 메서드는 파이프라인의 식별자와 파이프라인을 구성하는 대리자를 사용합니다. 대리자는 파이프라인에 복원력 전략을 추가하는 데 사용되는 인스턴스 `ResiliencePipelineBuilder`를 전달합니다.

다음 문자열 기반 `key` 예제를 고려합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Polly;
using Polly.CircuitBreaker;
using Polly.Registry;
using Polly.Retry;
using Polly.Timeout;

var services = new ServiceCollection();

const string key = "Retry-Timeout";

services.AddResiliencePipeline(key, static builder =>
{
    // See: https://www.pollydocs.org/strategies/retry.html
    builder.AddRetry(new RetryStrategyOptions
    {
        ShouldHandle = new PredicateBuilder().Handle<TimeoutRejectedException>()
    });

    // See: https://www.pollydocs.org/strategies/timeout.html
    builder.AddTimeout(TimeSpan.FromSeconds(1.5));
});
```

앞의 코드는 다음과 같습니다.

- 새 `ServiceCollection` 인스턴스를 만듭니다.
- 파이프라인을 식별하기 위해 `key`를 정의합니다.
- 인스턴스에 복원력 파이프라인을 추가합니다 `ServiceCollection` .
- 재시도 및 시간 제한 전략을 사용하여 파이프라인을 구성합니다.

각 파이프라인은 지정된 `key`파이프라인에 대해 구성되며 각 `key` 파이프라인은 공급자로부터 파이프라인을 가져올 때 해당 `ResiliencePipeline` 파이프라인을 식별하는 데 사용됩니다. 메서드 `key` 의 제네릭 형식 매개 변수입니다 `AddResiliencePipeline` .

탄력성 파이프라인 작성기 확장

파이프라인에 전략을 추가하려면 인스턴스에서 사용 가능한 `Add*` 확장 메서드 `ResiliencePipelineBuilder` 를 호출합니다.

- `AddRetry`: 문제가 일시적이고 사라질 수 있는 경우에 유용한 오류가 발생하면 다시 시도합니다.
- `AddCircuitBreaker`: 고장났거나 사용 중인 경우, 시간을 낭비하지 않고 상황을 악화시키지 않기 위해 시도를 중단하십시오. 이는 당신에게 도움이 됩니다.
- `AddTimeout`: 리소스를 확보하여 성능을 향상시킬 수 있는 시간이 너무 오래 걸리면 포기합니다.
- `AddRateLimiter`: 수락하는 요청 수를 제한하여 인바운드 로드를 제어할 수 있습니다.
- `AddConcurrencyLimiter`: 아웃바운드 로드를 제어할 수 있는 요청 수를 제한합니다.
- `AddFallback`: 오류가 발생할 때 다른 작업을 수행하여 사용자 환경을 개선합니다.
- `AddHedging`: 대기 시간이 높거나 오류가 발생할 경우 여러 요청을 실행하여 응답성을 향상시킬 수 있습니다.

자세한 내용은 복원력 전략을 참조하세요. 예를 들어 복원력 있는 HTTP 앱 빌드: 주요 개발 패턴을 참조하세요.

메트릭 확장

*증강*은 이름/값 쌍으로 이루어진 잘 알려진 상태로 원격 분석의 자동 증강입니다. 예를 들어 앱은 *작업* 및 *결과 코드를* 포함하는 로그를 열로 내보내 일부 작업의 결과를 나타낼 수 있습니다. 상황과 주변 컨텍스트에 따라, 원격 분석 백엔드로 전송되는 로그에 *클러스터 이름*, *프로세스 이름*, *지역*, *테넌트 ID* 등을 추가하는 확장을 실행합니다. 보강이 추가되면 앱 코드는 보강된 메트릭의 이점을 활용하기 위해 추가 작업을 수행할 필요가 없습니다.

풍부화 작동 방식

로그 및 메트릭을 생성하는 1,000개의 전역 분산 서비스 인스턴스를 상상해 보십시오. 서비스 대시보드에서 문제가 발생하면 문제가 있는 지역 또는 데이터 센터를 신속하게 식별하는 것이 중요합니다. 보강을 통해 메트릭 레코드에 분산 시스템의 오류를 정확히 파악하는 데 필요한 정보가 포함됩니다. 보강이 없으면 이 상태를 내부적으로 관리하고, 로깅 프로세스에 통합하고, 수동으로 전송하는 앱 코드가 부담됩니다. 보강은 이 프로세스를 간소화하여 앱의 논리에 영향을 주지 않고 원활하게 처리합니다.

복원성의 경우, 강화 요소를 추가할 때 다음과 같은 차원이 송신되는 원격 분석 데이터에 추가됩니다.

- `error.type`: 예외 정보의 카디널리티 버전이 낮습니다.
- `request.name`: 요청의 이름입니다.
- `request.dependency.name`: 종속성의 이름입니다.

내부적으로, 복원성 강화는 Polly의 원격 분석을 기반으로 구축됩니다 `MeteringEnricher`. 자세한 내용은 Polly: 계량 강화를 참조하세요.

복원력 보강 추가

복원력 파이프라인을 등록하는 것 외에도 복원력 보강을 등록할 수도 있습니다. 보강을 추가하려면 인스턴스에서 AddResilienceEnricher(IServiceCollection) 확장 메서드를 호출합니다 `IServiceCollection` .

C#

```
services.AddResilienceEnricher();
```

확장 메서드를 `AddResilienceEnricher` 호출하여 기본 Polly 라이브러리에 기본 제공되는 차원 위에 차원을 추가합니다. 다음 강화 차원이 추가됩니다.

- 이 메커니즘은 원격 분석을 위해 예외를 요약할 수 있도록 IExceptionSummarizer를 기반으로 한 예외 처리 기능을 제공합니다. 자세한 내용은 예외 요약을 참조하세요.
- RequestMetadata에 기반하여 원격 분석에 필요한 요청 메타데이터 보강을 요청합니다. 자세한 내용은 Polly: 원격 분석 메트릭을 참조하세요.

복원력 파이프라인 사용

구성된 복원력 파이프라인을 사용하려면 `ResiliencePipelineProvider<TKey>`에서 파이프라인을 가져와야 합니다. 이전에 `key` 파이프라인을 추가했을 때 그것은 `string` 형식이었으므로 `ResiliencePipelineProvider<string>`에서 파이프라인을 가져와야 합니다.

C#

```
using ServiceProvider provider = services.BuildServiceProvider();

ResiliencePipelineProvider<string> pipelineProvider =
    provider.GetRequiredService<ResiliencePipelineProvider<string>>();

ResiliencePipeline pipeline = pipelineProvider.GetPipeline(key);
```

앞의 코드는 다음과 같습니다.

- 인스턴스 `ServiceProvider`에서 `ServiceCollection`를 생성합니다.
- `ResiliencePipelineProvider<string>` 서비스 공급자에서 가져옵니다.
- `ResiliencePipeline`에서 `ResiliencePipelineProvider<string>`를 가져옵니다.

복원력 파이프라인 실행

복원력 파이프라인을 사용하려면 인스턴스에서 사용 가능한 `Execute*` 메서드를 호출합니다 `ResiliencePipeline` . 예를 들어 `ExecuteAsync` 메서드를 호출하는 예제를 살펴봅시다.

C#

```
await pipeline.ExecuteAsync(static cancellationToken =>
{
    // Code that could potentially fail.

    return ValueTask.CompletedTask;
});
```

앞의 코드는 `ExecuteAsync` 메서드 내에서 대리자를 실행합니다. 오류가 발생하면 구성된 전략이 실행됩니다. 예를 들어 세 번 다시 시도하도록 구성된 경우 `RetryStrategy` 대리자는 오류가 전파되기 전에 네 번 실행됩니다(초기 시도 1회 및 재시도 3회).

다음 단계

안정적인 HTTP 앱 빌드: 주요 개발 패턴 **재시도된 호출의 멱등성 처리 문제를** 고려합니다.

Last updated on 2025. 10. 20.

.NET의 네트워크 프로그래밍

아티클 • 2025. 02. 01.

.NET은 쉽고 빠르게 앱에 통합할 수 있는 인터넷 서비스의 계층화되고 확장 가능하며 관리되는 구현을 제공합니다. 네트워크 앱은 플러그형 프로토콜을 기반으로 구축하여 다양한 인터넷 프로토콜을 자동으로 활용하거나 플랫폼 간 소켓 인터페이스의 관리형 구현을 사용하여 소켓 수준에서 네트워크로 작업할 수 있습니다.

인터넷 앱

인터넷 앱은 정보를 요청하는 클라이언트 앱과 클라이언트의 정보 요청에 응답하는 서버 앱의 두 가지 종류로 광범위하게 분류할 수 있습니다. 클래식 인터넷 클라이언트 서버 앱은 사용자가 브라우저를 사용하여 전 세계 웹 서버에 저장된 문서 및 기타 데이터에 액세스하는 World Wide Web입니다.

앱은 이러한 역할 중 하나로만 제한되지 않습니다. 예를 들어 익숙한 중간 계층 앱 서버는 다른 서버에서 데이터를 요청하여 클라이언트의 요청에 응답합니다. 이 경우 서버와 클라이언트 모두의 역할을 합니다.

클라이언트 앱은 요청 및 응답에 사용할 요청된 인터넷 리소스 및 통신 프로토콜을 식별하여 요청합니다. 필요한 경우 클라이언트는 프록시 위치 또는 인증 정보(사용자 이름, 암호 등)와 같은 요청을 완료하는 데 필요한 추가 데이터도 제공합니다. 요청이 형성되면 요청을 서버로 보낼 수 있습니다.

리소스 식별

.NET은 URI(Uniform Resource Identifier)를 사용하여 요청된 인터넷 리소스 및 통신 프로토콜을 식별합니다. URI는 요청 및 응답에 대한 통신 프로토콜을 식별하는 스키마 식별자인 3개 이상의 조각과 4개의 조각으로 구성됩니다. DNS(도메인 이름 시스템) 호스트 이름 또는 인터넷에서 서버를 고유하게 식별하는 TCP 주소로 구성된 서버 식별자입니다. 서버에서 요청된 정보를 찾는 경로 식별자입니다. 및 클라이언트에서 서버로 정보를 전달하는 선택적 쿼리 문자열입니다.

System.Uri 형식은 URI(Uniform Resource Identifier)의 표현으로 사용되며 URI 부분에 쉽게 액세스할 수 있습니다. `Uri` 인스턴스를 만들려면 문자열을 전달할 수 있습니다.

C#

```
const string uriString =
    "https://learn.microsoft.com/en-us/dotnet/path?key=value#bookmark";
```

```
Uri canonicalUri = new(uriString);
Console.WriteLine(canonicalUri.Host);
Console.WriteLine(canonicalUri.PathAndQuery);
Console.WriteLine(canonicalUri.Fragment);
// Sample output:
//     learn.microsoft.com
//     /en-us/dotnet/path?key=value
//     #bookmark
```

`Uri` 클래스는 RFC 3986 따라 유효성 검사 및 정식화를 자동으로 수행합니다. 이러한 유효성 검사 및 정식화 규칙은 URI가 올바른 형식이고 URI가 정식 형식인지 확인하는 데 사용됩니다.

참고 항목

- 네트워킹에 대한 런타임 구성 옵션
- .NET HTTP 지원
- .NET에서의 소켓
- .NET의 TCP
- 자습서: C# 사용하여 .NET 콘솔 앱에서 HTTP 요청 만들기
- .NET 네트워킹 원격 분석
- .NET 네트워킹 개선

네트워크 사용 가능성

2025. 06. 17.

네임스페이 System.Net.NetworkInformation 스를 사용하면 네트워크 이벤트, 변경 내용, 통계 및 속성에 대한 정보를 수집할 수 있습니다. 이 문서에서는 클래스를 사용하여 System.Net.NetworkInformation.NetworkChange 네트워크 주소 또는 가용성이 변경되었는지 여부를 확인하는 방법을 알아봅니다. 또한 인터페이스 또는 프로토콜 기준으로 네트워크 통계 및 속성에 대해 알아봅니다. 마지막으로, 클래스를 System.Net.NetworkInformation.Ping 사용하여 원격 호스트에 연결할 수 있는지 여부를 확인합니다.

네트워크 변경 이벤트

이 System.Net.NetworkInformation.NetworkChange 클래스를 사용하면 네트워크 주소 또는 가용성이 변경되었는지 여부를 확인할 수 있습니다. 이 클래스를 사용하려면 변경 사항을 처리할 이벤트 처리기를 만들고 이를 a 또는 a NetworkAddressChangedEventHandlerNetworkAvailabilityChangedEventHandler와 연결합니다.

C#

```
NetworkChange.NetworkAvailabilityChanged += OnNetworkAvailabilityChanged;

static void OnNetworkAvailabilityChanged(
    object? sender, NetworkAvailabilityEventArgs networkAvailability) =>
    Console.WriteLine($"Network is available: {networkAvailability.IsAvailable}");

Console.WriteLine(
    "Listening changes in network availability. Press any key to continue.");
Console.ReadLine();

NetworkChange.NetworkAvailabilityChanged -= OnNetworkAvailabilityChanged;
```

앞의 C# 코드는 다음과 같습니다.

- 이벤트에 대한 이벤트 처리기를 등록합니다 NetworkChange.NetworkAvailabilityChanged .
- 이벤트 처리기는 단순히 가용성 상태를 콘솔에 씁니다.
- 코드가 네트워크 가용성의 변경 내용을 수신 대기하고 키 누름이 종료되기를 기다리는 것을 사용자에게 알리는 메시지가 콘솔에 기록됩니다.
- 이벤트 처리기를 등록 취소합니다.

C#

```
NetworkChange.NetworkAddressChanged += OnNetworkAddressChanged;

static void OnNetworkAddressChanged(
    object? sender, EventArgs args)
{
    foreach ((string name, OperationalStatus status) in
        NetworkInterface.GetAllNetworkInterfaces()
            .Select(networkInterface =>
                (networkInterface.Name, networkInterface.OperationalStatus)))
    {
        Console.WriteLine(
            $"{name} is {status}");
    }
}

Console.WriteLine(
    "Listening for address changes. Press any key to continue.");
Console.ReadLine();

NetworkChange.NetworkAddressChanged -= OnNetworkAddressChanged;
```

앞의 C# 코드는 다음과 같습니다.

- 이벤트에 대한 이벤트 처리기를 등록합니다 NetworkChange.NetworkAddressChanged .
- 이벤트 처리기는 NetworkInterface.GetAllNetworkInterfaces()를 순회하면서 그 이름과 작동 상태를 콘솔에 출력합니다.
- 코드가 네트워크 가용성의 변경 내용을 수신 대기하고 키 누름이 종료되기를 기다리는 것을 사용자에게 알리는 메시지가 콘솔에 기록됩니다.
- 이벤트 처리기를 등록 취소합니다.

네트워크 통계 및 속성

인터페이스 또는 프로토콜 기준으로 네트워크 통계 및 속성을 수집할 수 있습니다. NetworkInterface, NetworkInterfaceType, 및 PhysicalAddress 클래스는 특정 네트워크 인터페이스에 대한 정보를 제공하며, IPInterfaceProperties, IPGlobalProperties, IPGlobalStatistics, TcpStatistics, 및 UdpStatistics 클래스는 계층 3 및 계층 4 패킷에 대한 정보를 제공합니다.

C#

```
ShowStatistics(NetworkInterfaceComponent.IPv4);
ShowStatistics(NetworkInterfaceComponent.IPv6);

static void ShowStatistics(NetworkInterfaceComponent version)
{
    var properties = IPGlobalProperties.GetIPGlobalProperties();
    var stats = version switch
    {
```

```
        NetworkInterfaceComponent.IPv4 => properties.GetTcpIPv4Statistics(),
        _ => properties.GetTcpIPv6Statistics()
    };

    Console.WriteLine($"TCP/{version} Statistics");
    Console.WriteLine($"  Minimum Transmission Timeout :
{stats.MinimumTransmissionTimeout:#,#}");
    Console.WriteLine($"  Maximum Transmission Timeout :
{stats.MaximumTransmissionTimeout:#,#}");
    Console.WriteLine("  Connection Data");
    Console.WriteLine($"      Current :
{stats.CurrentConnections:#,#}");
    Console.WriteLine($"      Cumulative :
{stats.CumulativeConnections:#,#}");
    Console.WriteLine($"      Initiated  :
{stats.ConnectionsInitiated:#,#}");
    Console.WriteLine($"      Accepted :
{stats.ConnectionsAccepted:#,#}");
    Console.WriteLine($"      Failed Attempts :
{stats.FailedConnectionAttempts:#,#}");
    Console.WriteLine($"      Reset :
{stats.ResetConnections:#,#}");
    Console.WriteLine("  Segment Data");
    Console.WriteLine($"      Received :
{stats.SegmentsReceived:#,#}");
    Console.WriteLine($"      Sent :
{stats.SegmentsSent:#,#}");
    Console.WriteLine($"      Retransmitted :
{stats.SegmentsResent:#,#}");
    Console.WriteLine();
}
```

앞의 C# 코드는 다음과 같습니다.

- 사용자 지정 `ShowStatistics` 메서드를 호출하여 각 프로토콜에 대한 통계를 표시합니다.
- `ShowStatistics` 메서드는 IPGlobalProperties.GetIPGlobalProperties()를 호출하며, 지정된 NetworkInterfaceComponent에 따라 IPGlobalProperties.GetIPv4GlobalStatistics() 또는 IPGlobalProperties.GetIPv6GlobalStatistics()를 호출합니다.
- TcpStatistics이(가) 콘솔에 기록됩니다.

원격 호스트에 연결할 수 있는지 확인

클래스를 Ping 사용하여 원격 호스트가 실행 중인지, 네트워크에서 연결 가능한지 확인할 수 있습니다.

C#

```
using Ping ping = new();
```

```
string hostName = "stackoverflow.com";
PingReply reply = await ping.SendPingAsync(hostName);
Console.WriteLine($"Ping status for ({hostName}): {reply.Status}");
if (reply is { Status: IPStatus.Success })
{
    Console.WriteLine($"Address: {reply.Address}");
    Console.WriteLine($"Roundtrip time: {reply.RoundtripTime}");
    Console.WriteLine($"Time to live: {reply.Options?.Ttl}");
    Console.WriteLine();
}
```

앞의 C# 코드는 다음과 같습니다.

- 개체를 Ping 인스턴스화합니다.
- 호스트 이름 매개 변수를 사용하여 Ping.SendPingAsync(String)을 `"stackoverflow.com"`로 호출합니다.
- ping 상태가 콘솔에 기록됩니다.

참고하십시오

- .NET의 네트워크 프로그래밍
- NetworkInterface

IPv6(인터넷 프로토콜 버전 6) 개요

2025. 06. 17.

IPv6(인터넷 프로토콜 버전 6)은 인터넷의 네트워크 계층에 대한 표준 프로토콜 모음입니다. IPv6은 주소 고갈, 보안, 자동 구성, 확장성 등에 대한 현재 버전의 인터넷 프로토콜 제품군(IPv4라고 함)의 많은 문제를 해결하도록 설계되었습니다. IPv6은 피어 투 피어 및 모바일 애플리케이션을 비롯한 새로운 종류의 애플리케이션을 사용하도록 인터넷 기능을 확장합니다. 다음은 현재 IPv4 프로토콜의 주요 문제입니다.

- 주소 공간의 급속한 고갈.

  이로 인해 여러 개인 주소를 단일 공용 IP 주소에 매핑하는 NAT(Network Address Translators)가 사용되었습니다. 이 메커니즘에서 생성되는 주요 문제는 오버헤드 처리 및 엔드 투 엔드 연결 부족입니다.

- 계층 구조 지원이 부족합니다.

  IPv4는 내재된 미리 정의된 클래스 조직으로 인해 진정한 계층적 지원이 부족합니다. 네트워크 토폴로지와 실제로 매핑되는 방식으로 IP 주소를 구성하는 것은 불가능합니다. 이 중요한 디자인 결함으로 인해 인터넷의 모든 위치에 IPv4 패킷을 배달하기 위해 큰 라우팅 테이블이 필요합니다.

- 복잡한 네트워크 구성.

  IPv4를 사용하면 주소를 정적으로 할당하거나 DHCP와 같은 구성 프로토콜을 사용해야 합니다. 이상적인 상황에서 호스트는 DHCP 인프라 관리에 의존할 필요가 없습니다. 대신, 해당 위치의 네트워크 세그먼트에 따라 자신을 구성할 수 있습니다.

- 기본 제공 인증 및 기밀성이 부족합니다.

  IPv4는 교환된 데이터의 인증 또는 암호화를 제공하는 메커니즘에 대한 지원이 필요하지 않습니다. IPv6에서 변경합니다. IPSec(인터넷 프로토콜 보안)은 IPv6 지원 요구 사항입니다.

새 프로토콜 제품군은 다음 기본 요구 사항을 충족해야 합니다.

- 오버헤드가 낮은 대규모 라우팅 및 주소 지정
- 다양한 연결 상황에 대한 자동 구성입니다.
- 기본 제공 인증 및 기밀성

IPv6 주소 지정

IPv6에서는 주소 길이가 128비트입니다. 이러한 큰 주소 공간의 한 가지 이유는 사용 가능한 주소를 인터넷의 토폴로지가 반영된 라우팅 도메인의 계층 구조로 세분화하기 위해서입니다. 또 다른 이유는 디바이스를 네트워크에 연결하는 네트워크 어댑터(또는 인터페이스)의 주소를 매핑하는 것입니다. IPv6에는 네트워크 인터페이스 수준에 있는 가장 낮은 수준에서 주소를 확인하는 고유한 기능이 있으며 자동 구성 기능도 있습니다.

텍스트 표현

다음은 IPv6 주소를 텍스트 문자열로 나타내는 데 사용되는 세 가지 일반적인 양식입니다.

- **콜론 16진수 형식**:

  기본 설정 형식 `n:n:n:n:n:n:n:n`입니다. 각각 `n` 은 주소의 8개 16비트 요소 중 하나의 16진수 값을 나타냅니다. 예: `3FFE:FFFF:7654:FEDA:1245:BA98:3210:4562`.

- **압축된 형식**:

  주소 길이로 인해 긴 문자열 0을 포함하는 주소가 있는 것이 일반적입니다. 이러한 주소 작성을 간소화하려면 0 블록의 연속된 단일 시퀀스가 이중 콜론 기호(`::`)로 표현되는 압축된 형식을 사용합니다. 이 기호는 주소에 한 번만 나타날 수 있습니다. 예를 들어 압축된 형식의 멀티캐스트 주소 `FFED:0:0:0:0:BA98:3210:4562` 는 .입니다 `FFED::BA98:3210:4562`. 압축된 형식의 유니캐스트 주소 `3FFE:FFFF:0:0:8:800:20C4:0` 는 .입니다 `3FFE:FFFF::8:800:20C4:0`. 압축된 형식의 루프백 주소 `0:0:0:0:0:0:0:1` 는 .입니다 `::1`. 압축된 형식의 지정되지 않은 주소 `0:0:0:0:0:0:0:0` 는 `::` 입니다.

- **혼합 양식**:

  이 양식은 IPv4 및 IPv6 주소를 결합합니다. 이 경우 주소 형식은 각 n이 `n:n:n:n:n:n:d.d.d.d`6개의 IPv6 상위 16비트 주소 요소의 16진수 값을 나타내고 각 d는 IPv4 주소의 10진수 값을 나타내는 형식입니다.

주소 유형

주소의 선행 비트는 특정 IPv6 주소 유형을 정의합니다. 이러한 선행 비트를 포함하는 가변 길이 필드를 FP(서식 접두사)라고 합니다.

IPv6 유니캐스트 주소는 두 부분으로 나뉩니다. 첫 번째 부분은 주소 접두사를 포함하고, 두 번째 부분은 인터페이스 식별자를 포함합니다. IPv6 주소/접두사 조합을 표현하는 간결한 방법은 다음과 같습니다. ipv6-address/prefix-length.

다음은 64비트 접두사를 사용하는 주소의 예입니다.

`3FFE:FFFF:0:CD30:0:0:0:0/64`;

이 예제의 접두사는 .입니다 `3FFE:FFFF:0:CD30`. 주소는 압축된 형식 `3FFE:FFFF:0:CD30::/64`으로 작성할 수도 있습니다.

IPv6은 다음 주소 유형을 정의합니다.

- **유니캐스트 주소**:

  단일 인터페이스에 대한 식별자입니다. 이 주소로 전송된 패킷은 식별된 인터페이스로 전달됩니다. 유니캐스트 주소는 멀티캐스트 주소와 상위 8진수 값으로 구분됩니다. 멀티캐스트 주소의 상위 옥텟은 16진수 값이 FF입니다. 이 옥텟의 다른 모든 값은 유니캐스트 주소를 식별합니다. 다음은 다양한 유형의 유니캐스트 주소입니다.

  - **링크-로컬 주소**:

    이러한 주소는 단일 링크에서 사용되며 형식은 다음과 같습니다 `FE80::*InterfaceID*`. 링크-로컬 주소는 자동 주소 구성, 인접 검색 또는 라우터가 없을 때 링크의 노드 간에 사용됩니다. 링크-로컬 주소는 주로 시작 시와 시스템에서 더 큰 범위의 주소를 아직 획득하지 않은 경우에 사용됩니다.

  - **사이트-로컬 주소**:

  이러한 주소는 단일 사이트에서 사용되며 다음과 같은 형식 `FEC0::*SubnetID*:*InterfaceID*`을 갖습니다. 사이트-로컬 주소는 전역 접두사 없이 사이트 내에서 주소를 지정하는 데 사용됩니다.
  - **전역 IPv6 유니캐스트 주소**:

  이러한 주소는 인터넷을 통해 사용할 수 있으며 형식은 다음과 같습니다 `*GlobalRoutingPrefix*::*SubnetID*:*InterfaceID*`.

- **멀티캐스트 주소**:

  인터페이스 집합에 대한 식별자입니다(일반적으로 다른 노드에 속). 이 주소로 전송된 패킷은 주소로 식별된 모든 인터페이스로 전달됩니다. 멀티캐스트 주소 유형은 IPv4 브로드캐스트 주소를 대체합니다.

- **Anycast 주소**:

  인터페이스 집합에 대한 식별자입니다(일반적으로 다른 노드에 속). 이 주소로 전송된 패킷은 주소로 식별된 하나의 인터페이스로만 전달됩니다. 이는 라우팅 메트릭으로 식별되는 가장 가까운 인터페이스입니다. 애니캐스트 주소는 유니캐스트 주소 공간에서 가져온 것이며 구문적으로 구별할 수 없습니다. 주소 지정 인터페이스는 유니캐스트와 애니캐스트 주소를 구성의 함수로 구분합니다.

일반적으로 노드에는 항상 링크-로컬 주소가 있습니다. 사이트-로컬 주소와 하나 이상의 전역 주소가 있을 수 있습니다.

IPv6 라우팅

유연한 라우팅 메커니즘은 IPv6의 이점입니다. IPv4 네트워크 ID가 할당되고 할당되는 방식 때문에 인터넷 백본에 있는 라우터에서 큰 라우팅 테이블을 유지 관리해야 합니다. 이러한 라우터는 잠재적으로 인터넷의 모든 노드로 전달되는 패킷을 전달하는 모든 경로를 알고 있어야 합니다. 주소를 집계하는 기능을 통해 IPv6은 유연한 주소 지정을 허용하고 라우팅 테이블의 크기를 크게 줄입니다. 이 새로운 주소 지정 아키텍처에서 중간 라우터는 메시지를 적절하게 전달하기 위해 네트워크의 로컬 부분만 추적해야 합니다.

이웃 탐색

인접 검색에서 제공하는 기능 중 일부는 다음과 같습니다.

- **라우터 검색**: 이를 통해 호스트는 로컬 라우터를 식별할 수 있습니다.
- **주소 확인**: 이를 통해 노드는 해당 다음 홉 주소(주소 확인 프로토콜 [ARP]의 대체)에 대한 링크 계층 주소를 확인할 수 있습니다.
- **주소 자동 구성**: 이렇게 하면 호스트가 사이트-로컬 및 전역 주소를 자동으로 구성할 수 있습니다.

인접 검색은 다음을 포함하는 ICMPv6(IPv6) 메시지에 대한 인터넷 제어 메시지 프로토콜을 사용합니다.

- **라우터 광고**: 라우터가 의사 주기적으로 또는 라우터 요청에 응답하여 전송합니다. IPv6 라우터는 라우터 광고를 사용하여 가용성, 주소 접두사 및 기타 매개 변수를 보급합니다.
- **라우터 요청**: 호스트가 링크의 라우터가 라우터 광고를 즉시 보내도록 요청하도록 요청합니다.
- **인접 요청**: 주소 확인, 중복 주소 검색 또는 인접 항목에 연결할 수 있는지 확인하기 위해 노드에서 보냅니다.
- **인접 광고**: 인접한 요청에 응답하거나 이웃에게 링크 계층 주소의 변경 사항을 알리기 위해 노드에서 보냅니다.
- **리디렉션**: 라우터가 송신 노드에게 특정 목적지에 더 적합한 다음 홉 주소를 알려주기 위해 보냅니다.

IPv6 자동 구성

IPv6의 중요한 목표 중 하나는 노드 플러그 앤 플레이를 지원하는 것입니다. 즉, 노드를 IPv6 네트워크에 연결하고 사람의 개입 없이 자동으로 구성할 수 있어야 합니다.

자동 구성 유형

IPv6는 다음과 같은 유형의 자동 구성을 지원합니다.

- **상태를 저장하는 자동 구성**:

  이 유형의 구성은 노드 설치 및 관리를 위해 IPv6(DHCPv6)용 동적 호스트 구성 프로토콜이 필요하기 때문에 특정 수준의 사용자 개입이 필요합니다. DHCPv6 서버는 구성 정보를 제공하는 노드 목록을 유지합니다. 또한 서버에서 각 주소가 사용 중인 기간과 다시 할당할 수 있는 시기를 알 수 있도록 상태 정보를 유지 관리합니다.

- **상태 비지정 자동 구성**:

  이러한 유형의 구성은 소규모 조직 및 개인에게 적합합니다. 이 경우 각 호스트는 수신된 라우터 광고의 콘텐츠에서 해당 주소를 결정합니다. IEEE EUI-64 표준을 사용하여 주소의 네트워크 ID 부분을 정의하면 링크에서 호스트 주소의 고유성을 가정하는 것이 합리적입니다.

주소가 결정되는 방식에 관계없이 노드는 해당 잠재적 주소가 로컬 링크에 고유한지 확인해야 합니다. 이 작업은 잠재적인 주소로 인접 요청 메시지를 전송하여 수행됩니다. 노드가 응답을 수신하는 경우 주소가 이미 사용 중임을 알고 있으며 다른 주소를 확인해야 합니다.

IPv6 이동성

모바일 디바이스의 확산으로 인해 새로운 요구 사항이 도입되었습니다. 디바이스는 IPv6 인터넷에서 임의로 위치를 변경하고 기존 연결을 유지할 수 있어야 합니다. 이 기능을 제공하기 위해 모바일 노드에는 항상 연결할 수 있는 홈 주소가 할당됩니다. 모바일 노드가 집에 있는 경우 홈 링크에 연결하고 홈 주소를 사용합니다. 모바일 노드가 집에서 떨어져 있는 경우 일반적으로 라우터인 홈 에이전트는 통신 중인 노드와 모바일 노드 간에 메시지를 릴레이합니다.

IPv6 사용하지 않기 또는 사용하기

IPv6 프로토콜을 사용하려면 IPv6을 지원하는 운영 체제 버전을 실행 중인지 확인하고 운영 체제 및 네트워킹 클래스가 제대로 구성되었는지 확인합니다.

구성 단계

다음 표에서는 다양한 구성을 나열합니다.

테이블 확장

OS IPv6를 사용하도록 설정했나요?	코드 IPv6를 사용하도록 설정했나요?	설명
✗ 아니요	✗ 아니요	IPv6 주소를 구문 분석할 수 있습니다.
✗ 아니요	✓ 예	IPv6 주소를 구문 분석할 수 있습니다.

OS IPv6를 사용하도록 설정했나요?	코드 IPv6를 사용하도록 설정했나요?	설명
✔ 예	✘ 아니요	이름 확인 메서드 중 더 이상 사용되지 않는 것으로 표시되지 않은 것을 사용하여 IPv6 주소를 구문 분석하고 해결할 수 있습니다.
✔ 예	✔ 예	사용되지 않는 것으로 표시된 메서드를 포함하여 모든 메서드를 사용하여 IPv6 주소를 구문 분석하고 확인할 수 있습니다.

IPv6은 기본적으로 사용하도록 설정됩니다. 환경 변수에서 이 스위치를 구성하려면 환경 변수를 `DOTNET_SYSTEM_NET_DISABLEIPV6` 사용합니다. 자세한 내용은 .NET 환경 변수: DOTNET_SYSTEM_NET_DISABLEIPV6 참조하세요.

참고하십시오

- .NET의 네트워킹
- .NET에서의 소켓
- System.AppContext

.NET에서의 서비스 검색

이 문서에서는 라이브러리를 사용하는 `Microsoft.Extensions.ServiceDiscovery` 방법을 알아봅니다. 서비스 검색은 개발자가 물리적 주소(IP 주소 및 포트) 대신 논리 이름을 사용하여 외부 서비스를 참조하는 방법입니다.

시작하기

.NET에서 서비스 검색을 시작하려면 Microsoft.Extensions.ServiceDiscovery NuGet 패키지를 설치합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.ServiceDiscovery
```

또는 .NET 10+ SDK를 사용하는 경우:

.NET CLI

```
dotnet package add Microsoft.Extensions.ServiceDiscovery
```

자세한 내용은 .NET 애플리케이션에서 dotnet 패키지 추가 또는 패키지 종속성 관리를 참조하세요.

사용 예시

프로젝트의 *Program.cs* 파일에서 확장 메서드를 AddServiceDiscovery 호출하여 호스트에 서비스 검색을 추가하고 기본 서비스 엔드포인트 확인자를 구성합니다.

C#

```
builder.Services.AddServiceDiscovery();
```

서비스 검색을 추가하는 방법은, 개인 IHttpClientBuilder에게 `AddServiceDiscovery` 확장 메서드를 호출하는 것입니다.

C#

```
builder.Services.AddHttpClient<CatalogServiceClient>(static client =>
    {
        client.BaseAddress = new("https://catalog");
    })
    .AddServiceDiscovery();
```

또는 기본적으로 모든 HttpClient 인스턴스에 서비스 검색을 추가할 수 있습니다.

C#

```
builder.Services.ConfigureHttpClientDefaults(static http =>
{
    // Turn on service discovery by default
    http.AddServiceDiscovery();
});
```

HTTP(S) 엔드포인트를 확인할 때 스키마 선택

서비스를 배포할 때 로컬 및 HTTPS에서 서비스를 개발하고 테스트하는 동안 HTTP를 사용하는 것이 일반적입니다. 서비스 검색은 서비스 검색에 지정된 입력 문자열에 우선 순위 URI 스키마 목록을 지정할 수 있도록 하여 이를 지원합니다. 서비스 검색은 스키마에 대한 서비스를 순서대로 확인하려고 시도하며 엔드포인트를 찾은 후 중지됩니다. URI 스키마는 문자로 `+` 구분됩니다(예: `"https+http://basket"`. 서비스 검색은 먼저 `"basket"` 서비스의 HTTPS 엔드포인트를 찾으려고 시도하고, 찾을 수 없는 경우에는 HTTP 엔드포인트로 전환됩니다. HTTPS 엔드포인트가 있으면 서비스 검색에는 HTTP 엔드포인트가 포함되지 않습니다.

`AllowedSchemes`에서`AllowAllSchemes` 및 `ServiceDiscoveryOptions` 속성을 구성하여 스키마를 필터링할 수 있습니다. 속성 `AllowAllSchemes` 은 모든 스키마가 허용됨을 나타내는 데 사용됩니다. 기본적으로, `AllowAllSchemes`는 `true`이며 모든 스키마가 허용됩니다. 설정을 `AllowAllSchemes` 에서 `false`로 변경하고 허용된 스키마를 `AllowedSchemes` 속성에 추가하여 스키마를 제한할 수 있습니다. 예를 들어 HTTPS만 허용하려면 다음을 수행합니다.

C#

```
services.Configure<ServiceDiscoveryOptions>(options =>
{
    options.AllowAllSchemes = false;
    options.AllowedSchemes = ["https"];
});
```

모든 스키마를 명시적으로 허용하려면 `ServiceDiscoveryOptions.AllowAllSchemes` 속성을 `true` 로 설정합니다.

C#

```
services.Configure<ServiceDiscoveryOptions>(
    options => options.AllowAllSchemes = true);
```

구성에서 서비스 엔드포인트 해결

확장 메서드는 `AddServiceDiscovery` 기본적으로 구성 기반 엔드포인트 확인자를 추가합니다. 이 확인자는 .NET 구성 시스템에서 엔드포인트를 읽습니다. 라이브러리는 *appsettings.json*, 환경 변수 또는 기타 IConfiguration 원본을 통한 구성을 지원합니다.

서비스 *카탈로그*에 대한 엔드포인트를 `appsettings.json`를 통해 구성하는 방법을 보여주는 예입니다.

JSON

```
{
  "Services": {
    "catalog": {
      "https": [
        "localhost:8080",
        "10.46.24.90:80"
      ]
    }
  }
}
```

앞의 예제에서는 *카탈로그* `https://localhost:8080` `"https://10.46.24.90:80"`라는 서비스에 대해 두 개의 엔드포인트를 추가합니다. *카탈로그*가 확인될 때마다 이러한 엔드포인트 중 하나가 선택됩니다.

확장 메서드를 사용하여 AddServiceDiscoveryCore에서 IServiceCollection 호스트에 서비스 검색을 추가한 경우, AddConfigurationServiceEndpointProvider에서 `IServiceCollection` 확장 메서드를 호출하여 구성 기반 엔드포인트 확인자를 추가할 수 있습니다.

구성 / 설정

구성 확인자는 다음 구성 옵션을 제공하는 클래스를 ConfigurationServiceEndpointProviderOptions 사용하여 구성됩니다.

- **SectionName**: 서비스 엔드포인트를 포함하는 구성 섹션의 이름입니다. 기본값은 `"Services"`.
- **ApplyHostNameMetadata**: 호스트 이름 메타데이터를 확인된 엔드포인트에 적용해야 하는지 여부를 결정하는 데 사용되는 대리자입니다. 기본값은 을 반환하는 함수로 설정됩니다 `false`.

이러한 옵션을 구성하려면 애플리케이션의 `Configure` `IServiceCollection` 클래스 또는 `Startup` 파일 내에서 확장 메서드 `Program` 를 사용할 수 있습니다.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<ConfigurationServiceEndPointResolverOptions>(
    static options =>
    {
        options.SectionName = "MyServiceEndpoints";

        // Configure the logic for applying host name metadata
        options.ApplyHostNameMetadata = static endpoint =>
        {
            // Your custom logic here. For example:
            return endpoint.EndPoint is DnsEndPoint dnsEp
                && dnsEp.Host.StartsWith("internal");
        };
    });
```

앞의 예제에서는 서비스 엔드포인트에 대한 사용자 지정 섹션 이름을 설정하고 호스트 이름 메타데이터를 적용하기 위한 사용자 지정 조건부 논리를 제공하는 방법을 보여 줍니다.

플랫폼 제공 서비스 검색을 사용하여 서비스 엔드포인트 해결

Azure Container Apps 및 Kubernetes와 같은 특정 플랫폼(그에 따라 구성된 경우)은 서비스 검색 클라이언트 라이브러리 없이도 서비스 검색 기능을 제공합니다. 이러한 환경에서 애플리케이션이 배포되는 경우 플랫폼의 기본 제공 기능을 사용하는 것이 유리할 수 있습니다. 통과 확인자는 이 시나리오를 용이하게 하도록 설계되었습니다. 개발자 컴퓨터와 같은 다양한 환경에서 구성과 같은 대체 해결 프로그램을 사용할 수 있습니다. 중요한 것은 이러한 유연성은 코드 수정이나 조건부 가드 구현 없이도 달성됩니다.

패스스루 리졸버는 외부 해상도를 수행하지 않고, 대신 입력된 서비스 이름을 DnsEndPoint로 표현하여 엔드포인트를 확인합니다.

서비스 검색을 `AddServiceDiscovery` 확장 메서드를 통해 추가할 때, 통과 공급자는 기본적으로 구성됩니다.

호스트에 `AddServiceDiscoveryCore` 확장 메서드를 사용하여 서비스 검색을 추가한 경우, `IServiceCollection`에 AddPassThroughServiceEndpointProvider 확장 메서드를 호출하여 패스스루 제공자를 추가할 수 있습니다.

참고하십시오

- Aspire의 서비스 검색

Last updated on 2025. 10. 20.

.NET의 HTTP 지원

아티클 • 2025. 03. 22.

하이퍼텍스트 전송 프로토콜(또는 HTTP)은 웹 서버에서 리소스를 요청하기 위한 프로토콜입니다. System.Net.Http.HttpClient 클래스는 URI로 식별된 리소스에서 HTTP 요청을 보내고 HTTP 응답을 수신하는 기능을 노출합니다. 많은 유형의 리소스를 웹에서 사용할 수 있으며 HTTP는 이러한 리소스에 액세스하기 위한 요청 메서드 집합을 정의합니다.

HTTP 요청 메서드

요청 메서드는 먼저 *동사* 다음과 같은 특성으로 여러 요인을 통해 구분됩니다.

- 요청 메서드는 여러 번 성공적으로 처리되어도 결과를 변경하지 않을 경우 ***idempotent***입니다. 자세한 내용은 RFC 9110: 9.2.2를 참조하세요. Idempotent 메서드.
- 요청 메서드는 해당 응답을 다시 사용하도록 저장할 수 있는 경우 캐시 가능한. 자세한 내용은 RFC 9110: 섹션 9.2.3을 참조하세요. 메서드 및 캐싱.
- 요청 메서드는 리소스의 상태를 수정하지 않는 경우안전한 메서드로 간주됩니다. *모든* 안전한 메서드는 멱등 있지만 모든 *멱등* 메서드는 안전한 간주되지 않습니다. 자세한 내용은 RFC 9110: 섹션 9.2.1을 참조하세요. 안전한 메서드.

테이블 확장

HTTP 메서드	idempotent이다	캐시할 수 있나요?	안전합니다.
`GET`	✔ 예	✔ 예	✔ 예
`POST`	✘ 아니요	⚠†드물게	✘ 아니요
`PUT`	✔ 예	✘ 아니요	✘ 아니요
`PATCH`	✘ 아니요	✘ 아니요	✘ 아니요
`DELETE`	✔ 예	✘ 아니요	✘ 아니요
`HEAD`	✔ 예	✔ 예	✔ 예
`OPTIONS`	✔ 예	✘ 아니요	✔ 예
`TRACE`	✔ 예	✘ 아니요	✔ 예
`CONNECT`	✘ 아니요	✘ 아니요	✘ 아니요

> †`POST` 메서드는 적절한 `Cache-Control` 또는 `Expires` 응답 헤더가 있는 경우에만 캐시할 수 있습니다. 이것은 실제로 매우 드문 일입니다.

HTTP 상태 코드

.NET은 대부분의 인터넷 트래픽을 차지하는 HTTP 프로토콜에 대해 포괄적인 지원을 제공합니다. HttpClient 자세한 내용은 HttpClient 클래스사용하여 HTTP 요청 만들기를 참조하세요. 애플리케이션은 HttpRequestException을(를) 사용하여 HTTP 프로토콜 오류를 처리합니다. HTTP 상태 코드는 호출된 메서드가 응답 메시지를 반환하지 않는 경우 HttpResponseMessage에 HttpResponseMessage.StatusCode 또는 HttpRequestException에 HttpRequestException.StatusCode로 보고됩니다. 오류 처리에 대한 자세한 내용은 HTTP 오류 처리참조하고 상태 코드에 대한 자세한 내용은 RFC 9110, HTTP 의미 체계: 상태 코드 참조하세요.

정보 상태 코드

정보 상태 코드는 중간 응답을 반영합니다. 대부분의 중간 응답(예: HttpStatusCode.Continue)은 HttpClient 사용하여 내부적으로 처리되며 사용자에게 표시되지 않습니다.

테이블 확장

HTTP 상태 코드	`HttpStatusCode`
`100`	HttpStatusCode.Continue
`101`	HttpStatusCode.SwitchingProtocols
`102`	HttpStatusCode.Processing
`103`	HttpStatusCode.EarlyHints

성공적인 상태 코드

성공적인 상태 코드는 클라이언트의 요청이 성공적으로 수신, 이해 및 수락되었음을 나타냅니다.

테이블 확장

HTTP 상태 코드	`HttpStatusCode`
`200`	HttpStatusCode.OK

HTTP 상태 코드	`HttpStatusCode`
`201`	HttpStatusCode.Created
`202`	HttpStatusCode.Accepted
`203`	HttpStatusCode.NonAuthoritativeInformation
`204`	HttpStatusCode.NoContent
`205`	HttpStatusCode.ResetContent
`206`	HttpStatusCode.PartialContent
`207`	HttpStatusCode.MultiStatus
`208`	HttpStatusCode.AlreadyReported
`226`	HttpStatusCode.IMUsed

리디렉션 상태 코드

리디렉션 상태 코드를 사용하려면 사용자 에이전트가 요청을 수행하기 위한 조치를 취해야 합니다. 자동 리디렉션은 기본적으로 켜져 있으며 HttpClientHandler.AllowAutoRedirect 또는 SocketsHttpHandler.AllowAutoRedirect사용하여 변경할 수 있습니다.

테이블 확장

HTTP 상태 코드	`HttpStatusCode`
`300`	HttpStatusCode.MultipleChoices 또는 HttpStatusCode.Ambiguous
`301`	HttpStatusCode.MovedPermanently 또는 HttpStatusCode.Moved
`302`	HttpStatusCode.Found 또는 HttpStatusCode.Redirect
`303`	HttpStatusCode.SeeOther 또는 HttpStatusCode.RedirectMethod
`304`	HttpStatusCode.NotModified
`305`	HttpStatusCode.UseProxy
`306`	HttpStatusCode.Unused
`307`	HttpStatusCode.TemporaryRedirect 또는 HttpStatusCode.RedirectKeepVerb
`308`	HttpStatusCode.PermanentRedirect

클라이언트 오류 상태 코드

클라이언트 오류 상태 코드는 클라이언트의 요청이 잘못되었음을 나타냅니다.

테이블 확장

HTTP 상태 코드	`HttpStatusCode`
`400`	HttpStatusCode.BadRequest
`401`	HttpStatusCode.Unauthorized
`402`	HttpStatusCode.PaymentRequired
`403`	HttpStatusCode.Forbidden
`404`	HttpStatusCode.NotFound
`405`	HttpStatusCode.MethodNotAllowed
`406`	HttpStatusCode.NotAcceptable
`407`	HttpStatusCode.ProxyAuthenticationRequired
`408`	HttpStatusCode.RequestTimeout
`409`	HttpStatusCode.Conflict
`410`	HttpStatusCode.Gone
`411`	HttpStatusCode.LengthRequired
`412`	HttpStatusCode.PreconditionFailed
`413`	HttpStatusCode.RequestEntityTooLarge
`414`	HttpStatusCode.RequestUriTooLong
`415`	HttpStatusCode.UnsupportedMediaType
`416`	HttpStatusCode.RequestedRangeNotSatisfiable
`417`	HttpStatusCode.ExpectationFailed
`418`	나는 찻주전자
`421`	HttpStatusCode.MisdirectedRequest
`422`	HttpStatusCode.UnprocessableEntity
`423`	HttpStatusCode.Locked

HTTP 상태 코드	`HttpStatusCode`
`424`	HttpStatusCode.FailedDependency
`426`	HttpStatusCode.UpgradeRequired
`428`	HttpStatusCode.PreconditionRequired
`429`	HttpStatusCode.TooManyRequests
`431`	HttpStatusCode.RequestHeaderFieldsTooLarge
`451`	HttpStatusCode.UnavailableForLegalReasons

서버 오류 상태 코드

서버 오류 상태 코드는 서버가 요청을 처리하지 못하는 예기치 않은 조건이 발생했음을 나타냅니다.

테이블 확장

HTTP 상태 코드	`HttpStatusCode`
`500`	HttpStatusCode.InternalServerError
`501`	HttpStatusCode.NotImplemented
`502`	HttpStatusCode.BadGateway
`503`	HttpStatusCode.ServiceUnavailable
`504`	HttpStatusCode.GatewayTimeout
`505`	HttpStatusCode.HttpVersionNotSupported
`506`	HttpStatusCode.VariantAlsoNegotiates
`507`	HttpStatusCode.InsufficientStorage
`508`	HttpStatusCode.LoopDetected
`510`	HttpStatusCode.NotExtended
`511`	HttpStatusCode.NetworkAuthenticationRequired

참고하세요

- HttpClient 클래스 사용하여 HTTP 요청을 만듭니다.

- .NET을 사용하여 HTTP 클라이언트 팩터리
- HttpClient 사용하기 위한 지침
- .NET 네트워킹 개선

HttpClient 사용 지침

System.Net.Http.HttpClient 클래스는 URI로 식별되는 리소스에서 HTTP 요청을 보내고 HTTP 응답을 받습니다. HttpClient 인스턴스는 해당 인스턴스에서 실행되는 모든 요청에 적용되는 설정의 컬렉션이며, 각 인스턴스는 자체 연결 풀을 사용하여 요청을 다른 인스턴스와 격리합니다. .NET Core 2.1부터 SocketsHttpHandler 클래스는 구현을 제공하여 모든 플랫폼에서 일관된 동작이 가능합니다.

DNS 동작

HttpClient는 연결을 설정할 때 DNS 항목만 확인합니다. DNS 서버에서 지정한 TTL(Time to Live) 기간을 추적하지 않습니다. 일부 시나리오에서 발생할 수 있듯이 DNS 항목이 정기적으로 변경되는 경우 클라이언트는 이러한 업데이트를 적용하지 않습니다. 이 문제를 해결하려면 연결을 바꿀 때 DNS 조회가 반복되도록 속성을 설정 PooledConnectionLifetime 하여 연결의 수명을 제한합니다. 다음 예제를 고려하세요.

C#

```
var handler = new SocketsHttpHandler
{
    PooledConnectionLifetime = TimeSpan.FromMinutes(15) // Recreate every 15 minutes
};
var sharedClient = new HttpClient(handler);
```

위의 `HttpClient`는 15분 동안 연결을 다시 사용하도록 구성되었습니다. 지정한 PooledConnectionLifetime 시간 범위가 경과하고 연결이 마지막으로 연결된 요청(있는 경우)을 완료하면 연결이 닫힙니다. 큐에서 대기 중인 요청이 있는 경우 필요에 따라 새 연결이 만들어집니다.

15분 간격은 설명 목적으로 임의로 선택되었습니다. DNS 또는 기타 네트워크 변경의 예상 빈도에 따라 값을 선택해야 합니다.

풀링된 연결

HttpClient에 대한 연결 풀은 기본 SocketsHttpHandler에 연결됩니다. HttpClient 인스턴스가 삭제되면 풀 내의 기존 연결이 모두 삭제됩니다. 나중에 동일한 서버에 요청을 보내는 경우 새 연결을 다시 만들어야 합니다. 따라서 불필요한 연결 만들기로 인한 성능 저하가 발생합니다. 또한 TCP 포트는 연결이 닫힌 직후에 해제되지 않습니다. (자세한 내용은 `TIME-WAIT`의 TCP 를 참조하세요.) 요청 속도가 높으면 사용 가능한 포트의 운영 체제 제한이 소진될 수 있습니다. 포트 소진 문제를 방지하기 위해 최대한 많은 HTTP 요청에 인스턴스를 다시 사용할 것을 권장합니다.

권장 사용

수명 관리 측면에서 권장되는 `HttpClient` 사용을 요약하자면, *수명이 긴* 클라이언트를 사용하고 `PooledConnectionLifetime`(.NET Core 및 .NET 5+)을 설정하거나 *단기* 클라이언트를 `IHttpClientFactory`에서 생성하여 사용해야 합니다.

- .NET Core 및 .NET 5 이후 버전:
  - 예상된 DNS 변경 내용에 따라 원하는 간격(예: 2분)으로 설정하여 `static` 싱글톤 또는 HttpClientPooledConnectionLifetime 인스턴스를 사용합니다. 이렇게 하면 IHttpClientFactory의 오버헤드를 추가하지 않고 포트 소진 및 DNS 변경 문제가 모두 해결됩니다. 처리기를 모의 테스트해야 하는 경우 별도로 등록할 수 있습니다.

    > **팁**
    >
    > 제한된 수의 **HttpClient** 인스턴스만 사용하는 경우 이는 허용 가능한 전략이기도 합니다. 중요한 것은 이들이 요청마다 생성 및 삭제되지 않는다는 것입니다. 각 요청이 연결 풀을 포함하기 때문입니다. 여러 프록시가 있는 시나리오 또는 쿠키 처리를 완전히 사용하지 않도록 설정하지 않고 쿠키 컨테이너를 분리하려는 경우에는 둘 이상의 인스턴스를 사용해야 합니다.

  - IHttpClientFactory를 사용하면 다양한 사용 사례에 대해 다르게 구성된 클라이언트가 여러 개 있을 수 있습니다. 그러나 팩터리에서 만든 클라이언트는 수명이 짧아야 하며 일단 만들어진 클라이언트는 팩터리에서 더 이상 제어할 수 없습니다.

    팩터리는 HttpMessageHandler 인스턴스를 풀링하며, 팩터리에서 새 HttpClient 인스턴스를 만들 때 풀에서 수명이 만료되지 않은 처리기를 다시 사용할 수 있습니다. 이 재사용은 소켓 소진 문제를 방지합니다.

    IHttpClientFactory가 제공하는 구성 가능성이 필요한 경우 형식화된 클라이언트 접근 방식을 사용하는 것이 좋습니다.

- .NET Framework에서는 IHttpClientFactory를 사용하여 `HttpClient` 인스턴스를 관리합니다. 팩터리를 사용하지 않고 각 요청에 대한 새 클라이언트 인스턴스를 직접 만드는 경우 사용 가능한 포트를 소진할 수 있습니다.

  > **⚠ 경고**
  >
  > 앱에 쿠키가 필요한 경우 사용하지 않는 **IHttpClientFactory**것이 좋습니다. **HttpMessageHandler** 인스턴스를 풀링하면 **CookieContainer** 개체가 공유됩니다. 예기치 **CookieContainer** 않은 공유는 애플리케이션의 관련 없는 부분 간에 쿠키를 누출

할 수 있습니다. 또한 만료되면 HandlerLifetime 처리기가 재활용되므로 해당 처리기에 저장된 CookieContainer 모든 쿠키가 손실됩니다.

`HttpClient`을 사용한 `IHttpClientFactory` 수명 관리에 대한 자세한 내용은 IHttpClientFactory 지침을 참조하세요.

정적 클라이언트를 사용하는 복원력

다음 패턴을 활용하여 여러 복원력 파이프라인을 사용하도록 `static` 또는 *싱글톤* 클라이언트를 구성할 수 있습니다.

C#

```
using Microsoft.Extensions.Http.Resilience;
using Polly;

class MyClass
{
    static HttpClient? s_httpClient;

    MyClass()
    {
        var retryPipeline = new ResiliencePipelineBuilder<HttpResponseMessage>()
            .AddRetry(new HttpRetryStrategyOptions
            {
                BackoffType = DelayBackoffType.Exponential,
                MaxRetryAttempts = 3
            })
            .Build();

        var socketHandler = new SocketsHttpHandler
        {
            PooledConnectionLifetime = TimeSpan.FromMinutes(15)
        };
        var resilienceHandler = new ResilienceHandler(retryPipeline)
        {
            InnerHandler = socketHandler,
        };

        s_httpClient = new HttpClient(resilienceHandler);
    }
}
```

앞의 코드가 하는 역할은 다음과 같습니다.

- Microsoft.Extensions.Http.Resilience NuGet 패키지를 활용합니다.
- 각 시도에서 지연 간격을 기하급수적으로 늘려가는 재시도 파이프라인으로 구성된 일시적인 HTTP 오류 처리기를 지정합니다.

- `socketHandler`에 대한 풀된 연결 수명(15분)을 정의합니다.
- 재시도 논리를 사용하여 `socketHandler`를 `resilienceHandler`에 전달합니다.
- 지어진 `HttpClient`를 기반으로 공유된 `resilienceHandler`을 인스턴스화합니다.

참고

- .NET의 HTTP 지원
- .NET을 사용한 HTTP 클라이언트 팩터리
- HttpClient를 사용하여 HTTP 요청 만들기
- IHttpClientFactory를 사용하여 복원력 있는 HTTP 요청 구현

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

HttpWebRequest에서 HttpClient로의 마이그레이션 가이드

이 문서는 개발자가 HttpWebRequest, ServicePoint, ServicePointManager에서 HttpClient로 마이그레이션하는 과정을 안내하는 데 목적이 있습니다. 이전 API의 노후화와 HttpClient이 제공하는 성능 향상, 리소스 관리 개선, 보다 현대적이고 유연한 API 설계 등 다양한 이점으로 인해 마이그레이션이 필요했습니다. 이 문서에 설명된 단계를 따르면 개발자는 코드베이스를 원활하게 전환하고 HttpClient에서 제공하는 기능을 최대한 활용할 수 있습니다.

> ⚠ **Warning**
>
> `HttpWebRequest`, `ServicePoint`, `ServicePointManager`에서 `HttpClient`로 마이그레이션하는 것은 단순히 "있으면 좋은" 성능 향상에 그치지 않습니다. .NET(Core)로 전환하면 기존 `WebRequest` 로직의 성능이 크게 저하될 수 있다는 점을 이해하는 것이 중요합니다. `WebRequest`는 최소한의 호환성 계층으로 유지되기 때문에 많은 경우 연결 재사용과 같은 많은 최적화가 부족하기 때문입니다. 따라서 애플리케이션의 성능과 리소스 관리를 최신 표준에 맞게 유지하려면 `HttpClient`로 전환하는 것이 필수적입니다.

HttpWebRequest에서 HttpClient로 마이그레이션

몇 가지 예부터 시작하겠습니다:

HttpWebRequest 을 이용한 간단한 GET 요청

다음은 코드가 어떻게 표시되는지 보여주는 예시입니다:

```c#
HttpWebRequest request = WebRequest.CreateHttp(uri);
using WebResponse response = await request.GetResponseAsync();
```

HttpClient 을 이용한 간단한 GET 요청

다음은 코드가 어떻게 표시되는지 보여주는 예시입니다:

```c#
HttpClient client = new();
using HttpResponseMessage message = await client.GetAsync(uri);
```

HttpWebRequest 을 이용한 간단한 POST 요청

다음은 코드가 어떻게 표시되는지 보여주는 예시입니다:

```c#
HttpWebRequest request = WebRequest.CreateHttp(uri);
request.Method = "POST";
request.ContentType = "text/plain";
```

```
await using Stream stream = await request.GetRequestStreamAsync();
await stream.WriteAsync("Hello World!"u8.ToArray());
using WebResponse response = await request.GetResponseAsync();
```

HttpClient 을 이용한 간단한 POST 요청

다음은 코드가 어떻게 표시되는지 보여주는 예시입니다:

c#

```
HttpClient client = new();
using HttpResponseMessage responseMessage = await client.PostAsync(uri, new StringContent("Hello World!"));
```

`HttpWebRequest` to `HttpClient`, `SocketsHttpHandler` 마이그레이션 가이드

테이블 확장

HttpWebRequest이전 API	새 API	노트
`Accept`	Accept	예시: 요청 헤더 설정.
`Address`	RequestUri	예시: 리디렉션된 URI 가져오기.
`AllowAutoRedirect`	AllowAutoRedirect	예시: SocketsHttpHandler 속성 설정하기.
`AllowReadStreamBuffering`	직접적으로 동등한 API 없음	버퍼링 속성 사용.
`AllowWriteStreamBuffering`	직접적으로 동등한 API 없음	버퍼링 속성 사용.
`AuthenticationLevel`	직접적으로 동등한 API 없음	예시: 상호 인증 사용 설정.
`AutomaticDecompression`	AutomaticDecompression	예시: SocketsHttpHandler 속성 설정하기.
`CachePolicy`	직접적으로 동등한 API 없음	예시: CachePolicy 헤더 적용.
`ClientCertificates`	SslOptions.ClientCertificates	HttpClient에서 인증서 관련 속성 사용.
`Connection`	Connection	예시: 요청 헤더 설정.
`ConnectionGroupName`	동등한 API 없음	해결 방법 없음
`ContentLength`	ContentLength	예시: 콘텐츠 헤더 설정.
`ContentType`	ContentType	예시: 콘텐츠 헤더 설정.
`ContinueDelegate`	동등한 API 없음	해결 방법이 없습니다.
`ContinueTimeout`	Expect100ContinueTimeout	예시: SocketsHttpHandler 속성 설정.

HttpWebRequest이전 API	새 API	노트
`CookieContainer`	CookieContainer	예시: SocketsHttpHandler 속성 설정.
`Credentials`	Credentials	예시: SocketsHttpHandler 속성 설정.
`Date`	Date	예시: 요청 헤더 설정.
`DefaultCachePolicy`	직접적으로 동등한 API 없음	예시: CachePolicy 헤더 적용.
`DefaultMaximumErrorResponseLength`	직접적으로 동등한 API 없음	예시: HttpClient에서 MaximumErrorResponseLength 설정.
`DefaultMaximumResponseHeadersLength`	동등한 API 없음	MaxResponseHeadersLength을 대신 사용할 수 있습니다.
`DefaultWebProxy`	동등한 API 없음	Proxy을 대신 사용할 수 있습니다.
`Expect`	Expect	예시: 요청 헤더 설정.
`HaveResponse`	동등한 API 없음	`HttpResponseMessage` 인스턴스가 있는 것으로 암시됩니다.
`Headers`	Headers	예시: 요청 헤더 설정.
`Host`	Host	예시: 요청 헤더 설정.
`IfModifiedSince`	IfModifiedSince	예시: 요청 헤더 설정.
`ImpersonationLevel`	직접적으로 동등한 API 없음	예시: ImpersonationLevel 변경.
`KeepAlive`	직접적으로 동등한 API 없음	예시: 요청 헤더 설정.
`MaximumAutomaticRedirections`	MaxAutomaticRedirections	예시: SocketsHttpHandler 속성 설정하기.
`MaximumResponseHeadersLength`	MaxResponseHeadersLength	예시: SocketsHttpHandler 속성 설정하기.
`MediaType`	직접적으로 동등한 API 없음	예시: 콘텐츠 헤더 설정.
`Method`	Method	예시: HttpRequestMessage 속성 사용법.
`Pipelined`	동등한 API 없음	`HttpClient`은 파이프라이닝을 지원하지 않습니다.
`PreAuthenticate`	PreAuthenticate	
`ProtocolVersion`	`HttpRequestMessage.Version`	예시: HttpRequestMessage 속성 사용법.
`Proxy`	Proxy	예시: SocketsHttpHandler 속성 설정하기.

HttpWebRequest이전 API	새 API	노트
`ReadWriteTimeout`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.
`Referer`	Referrer	예시: 요청 헤더 설정.
`RequestUri`	RequestUri	예시: HttpRequestMessage 속성 사용법.
`SendChunked`	TransferEncodingChunked	예시: 요청 헤더 설정.
`ServerCertificateValidationCallback`	SslOptions.RemoteCertificateValidationCallback	예시: SocketsHttpHandler 속성 설정하기.
`ServicePoint`	동등한 API 없음	`ServicePoint`은 `HttpClient`의 일부가 아닙니다.
`SupportsCookieContainer`	동등한 API 없음	`true`의 경우 항상 `HttpClient`입니다.
`Timeout`	Timeout	
`TransferEncoding`	TransferEncoding	예시: 요청 헤더 설정.
`UnsafeAuthenticatedConnectionSharing`	동등한 API 없음	해결 방법 없음
`UseDefaultCredentials`	직접적으로 동등한 API 없음	예시: SocketsHttpHandler 속성 설정하기.
`UserAgent`	UserAgent	예시: 요청 헤더 설정.

`ServicePointManager` 사용법 마이그레이션

`ServicePointManager`는 정적 클래스이므로 해당 속성을 변경하면 애플리케이션 내에서 새로 생성된 모든 `ServicePoint` 객체에 전역적으로 영향을 미친다는 점에 유의해야 합니다. 예를 들어 같은 `ConnectionLimit` 속성을 수정하거나 `Expect100Continue`수정하면 모든 새 `ServicePoint` 인스턴스에 영향을 줍니다.

> **⚠ Warning**
>
> 최신 .NET에서 `HttpClient`은 `ServicePointManager`에 설정된 구성을 고려하지 않습니다.

ServicePointManager 속성 매핑

테이블 확장

ServicePointManager이전 API	새 API	노트
`CheckCertificateRevocationList`	SslOptions.CertificateRevocationCheckMode	예시: SocketsHttpHandler로 CRL 검사 활성화.
`DefaultConnectionLimit`	MaxConnectionsPerServer	예시: SocketsHttpHandler 속성 설정

ServicePointManager이전 API	새 API	노트
		하기.
`DnsRefreshTimeout`	동등한 API 없음	예시: DNS Round Robin 사용.
`EnableDnsRoundRobin`	동등한 API 없음	예시: DNS Round Robin 사용.
`EncryptionPolicy`	SslOptions.EncryptionPolicy	예시: SocketsHttpHandler 속성 설정하기.
`Expect100Continue`	ExpectContinue	예시: 요청 헤더 설정.
`MaxServicePointIdleTime`	PooledConnectionIdleTimeout	예시: SocketsHttpHandler 속성 설정하기.
`MaxServicePoints`	동등한 API 없음	`ServicePoint`은 `HttpClient`의 일부가 아닙니다.
`ReusePort`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.
`SecurityProtocol`	SslOptions.EnabledSslProtocols	예시: SocketsHttpHandler 속성 설정하기.
`ServerCertificateValidationCallback`	SslOptions.RemoteCertificateValidationCallback	둘 다 RemoteCertificateValidationCallback입니다.
`UseNagleAlgorithm`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.

> ⚠ **Warning**
>
> 최신 .NET에서는 `UseNagleAlgorithm` 및 `Expect100Continue` 속성의 기본값이 `false`으로 설정되어 있습니다. 이 값은 .NET Framework에서 기본적으로 `true`입니다.

ServicePointManager매핑 방법

테이블 확장

ServicePointManager이전 API	새 API	노트
`FindServicePoint`	동등한 API 없음	해결 방법 없음
`SetTcpKeepAlive`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.

ServicePoint 속성 매핑

테이블 확장

ServicePoint이전 API	새 API	노트
`Address`	`HttpRequestMessage.RequestUri`	요청 URL이며, 이 정보는 `HttpRequestMessage`에서 확인할 수 있습니다.
`BindIPEndPointDelegate`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.
`Certificate`	직접적으로 동등한 API 없음	이 정보는 `RemoteCertificateValidationCallback`에서 가져올 수 있습니다. 예시: 인증서 가져오기.
`ClientCertificate`	동등한 API 없음	예시: 상호 인증 사용 설정.
`ConnectionLeaseTimeout`	`SocketsHttpHandler.PooledConnectionLifetime`	HttpClient의 동등한 설정
`ConnectionLimit`	MaxConnectionsPerServer	예시: SocketsHttpHandler 속성 설정하기.
`ConnectionName`	동등한 API 없음	해결 방법 없음
`CurrentConnections`	동등한 API 없음	.NET의 네트워킹 원격 분석을 참조하세요.
`Expect100Continue`	ExpectContinue	예시: 요청 헤더 설정.
`IdleSince`	동등한 API 없음	해결 방법 없음
`MaxIdleTime`	PooledConnectionIdleTimeout	예시: SocketsHttpHandler 속성 설정하기.
`ProtocolVersion`	`HttpRequestMessage.Version`	예시: HttpRequestMessage 속성 사용법.
`ReceiveBufferSize`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.
`SupportsPipelining`	동등한 API 없음	`HttpClient`은 파이프라이닝을 지원하지 않습니다.
`UseNagleAlgorithm`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.

ServicePoint 메소드 매핑

테이블 확장

ServicePoint이전 API	새 API	노트
`CloseConnectionGroup`	해당 항목 없음	해결 방법 없음
`SetTcpKeepAlive`	직접적으로 동등한 API 없음	SocketsHttpHandle 및 ConnectCallback 사용법.

`HttpClient` 사용 및 `HttpRequestMessage` 속성

.NET에서 HttpClient로 작업할 때는 HTTP 요청 및 응답을 구성하고 사용자 지정할 수 있는 다양한 프로퍼티에 액세스할 수 있습니다. 이러한 속성을 이해하면 `HttpClient` 애플리케이션이 웹 서비스와 효율적이고 안전하게 통신할 수 있도록 최대한 활용할 수 있습니다.

예: 속성 사용 `HttpRequestMessage`

다음은 `HttpClient`와 `HttpRequestMessage`를 함께 사용하는 방법의 예입니다.

C#

```
var client = new HttpClient();

using var request = new HttpRequestMessage(HttpMethod.Post, "https://example.com"); // Method and
RequestUri usage
using var request = new HttpRequestMessage() // Alternative way to set RequestUri and Method
{
    RequestUri = new Uri("https://example.com"),
    Method = HttpMethod.Post
};
request.Headers.Add("Custom-Header", "value");
request.Content = new StringContent("somestring");

using var response = await client.SendAsync(request);
var protocolVersion = response.RequestMessage.Version; // Fetch `ProtocolVersion`.
```

예시: 리디렉션된 URI 가져오기

다음은 리디렉션된 URI를 가져오는 방법의 예시입니다(`HttpWebRequest.Address`과 동일).

C#

```
var client = new HttpClient();
using var response = await client.GetAsync(uri);
var redirectedUri = response.RequestMessage.RequestUri;
```

사용 현황 `SocketsHttpHandler` 및 `ConnectCallback`

`ConnectCallback`의 `SocketsHttpHandler` 속성을 통해 개발자는 TCP 연결 설정 프로세스를 사용자 지정할 수 있습니다. 이는 DNS 확인을 제어하거나 연결에 특정 소켓 옵션을 적용해야 하는 시나리오에 유용할 수 있습니다. `ConnectCallback`을 사용하면 `HttpClient`에서 사용하기 전에 연결 프로세스를 가로채서 수정할 수 있습니다.

예시: 소켓에 IP 주소 바인딩

`HttpWebRequest`을 사용하는 기존 접근 방식에서는 사용자 지정 로직을 사용하여 특정 IP 주소를 소켓에 바인딩했을 수 있습니다. `HttpClient`과 `ConnectCallback`를 사용하여 유사한 기능을 구현하는 방법은 다음과 같습니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

C#

```
HttpWebRequest request = WebRequest.CreateHttp(uri);
request.ServicePoint.BindIPEndPointDelegate = (servicePoint, remoteEndPoint, retryCount) =>
{
    // Bind to a specific IP address
```

```
    IPAddress localAddress = IPAddress.Parse("192.168.1.100");
    return new IPEndPoint(localAddress, 0);
};
using HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

`HttpClient` 및 `ConnectCallback`를 사용하는 새 코드:

C#

```
var handler = new SocketsHttpHandler
{
    ConnectCallback = async (context, cancellationToken) =>
    {
        // Bind to a specific IP address
        IPAddress localAddress = IPAddress.Parse("192.168.1.100");
        var socket = new Socket(localAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
        try
        {
            socket.Bind(new IPEndPoint(localAddress, 0));
            await socket.ConnectAsync(context.DnsEndPoint, cancellationToken);
            return new NetworkStream(socket, ownsSocket: true);
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
};
var client = new HttpClient(handler);
using var response = await client.GetAsync(uri);
```

예시: 특정 소켓 옵션 적용

TCP 킵얼라이브 활성화와 같은 특정 소켓 옵션을 적용해야 하는 경우 `ConnectCallback`을 사용하여 `HttpClient`에서 사용하기 전에 소켓을 구성할 수 있습니다. 실제로 `ConnectCallback`이 소켓 옵션을 더 유연하게 구성할 수 있습니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

C#

```
ServicePointManager.ReusePort = true;
HttpWebRequest request = WebRequest.CreateHttp(uri);
request.ServicePoint.SetTcpKeepAlive(true, 60000, 1000);
request.ServicePoint.ReceiveBufferSize = 8192;
request.ServicePoint.UseNagleAlgorithm = false;
using HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

`HttpClient` 및 `ConnectCallback`를 사용하는 새 코드:

C#

```
var handler = new SocketsHttpHandler
{
```

```
    ConnectCallback = async (context, cancellationToken) =>
    {
        var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
        try
        {
            // Setting TCP Keep Alive
            socket.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.KeepAlive, true);
            socket.SetSocketOption(SocketOptionLevel.Tcp, SocketOptionName.TcpKeepAliveTime, 60);
            socket.SetSocketOption(SocketOptionLevel.Tcp, SocketOptionName.TcpKeepAliveInterval,
1);

            // Setting ReceiveBufferSize
            socket.ReceiveBufferSize = 8192;

            // Enabling ReusePort
            socket.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.ReuseUnicastPort,
true);

            // Disabling Nagle Algorithm
            socket.NoDelay = true;

            await socket.ConnectAsync(context.DnsEndPoint, cancellationToken);
            return new NetworkStream(socket, ownsSocket: true);
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
};
var client = new HttpClient(handler);
using var response = await client.GetAsync(uri);
```

예: DNS 라운드 로빈 활성화

DNS Round Robin은 단일 도메인 이름과 연결된 IP 주소 목록을 순환하여 여러 서버에 네트워크 트래픽을 분산하는 데 사용되는 기술입니다. 이는 로드 밸런싱과 서비스 가용성 향상에 도움이 됩니다. `HttpClient`를 사용할 때, `ConnectCallback`의 `SocketsHttpHandler` 속성을 사용하여 IP 주소를 순환하고 DNS 확인을 수동으로 처리함으로써 DNS 라운드 로빈을 구현할 수 있습니다.

DNS 라운드 로빈을 `HttpClient`에서 활성화하려면, `ConnectCallback` 속성을 사용하여 DNS 항목을 수동으로 확인하고 IP 주소들을 순환할 수 있습니다. 다음은 `HttpWebRequest`과 `HttpClient`의 예시입니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

C#

```
ServicePointManager.DnsRefreshTimeout = 60000;
ServicePointManager.EnableDnsRoundRobin = true;
HttpWebRequest request = WebRequest.CreateHttp(uri);
using HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

이전 `HttpWebRequest` API에서는 이 기능이 기본 지원되었기 때문에 DNS Round Robin을 활성화하는 것이 간단했습니다. 그러나 최신 `HttpClient` API는 동일한 기본 제공 기능을 제공하지 않습니다. 그럼에도 불구하고

DNS 확인을 통해 반환된 IP 주소를 수동으로 순환하는 `DnsRoundRobinConnector`을 구현하여 유사한 동작을 얻을 수 있습니다.

`HttpClient`를 사용하는 새 코드:

C#

```
// This is available as NuGet Package: https://www.nuget.org/packages/DnsRoundRobin/
// The original source code can be found also here: https://github.com/MihaZupan/DnsRoundRobin
public sealed class DnsRoundRobinConnector : IDisposable
```

구현 `DnsRoundRobinConnector`은 DnsRoundRobin.cs 참조하세요.

`DnsRoundRobinConnector` 사용:

C#

```
private static readonly DnsRoundRobinConnector s_roundRobinConnector = new(
        dnsRefreshInterval: TimeSpan.FromSeconds(10),
        endpointConnectTimeout: TimeSpan.FromSeconds(5));
static async Task DnsRoundRobinConnectAsync()
{
    var handler = new SocketsHttpHandler
    {
        ConnectCallback = async (context, cancellation) =>
        {
            Socket socket = await DnsRoundRobinConnector.Shared.ConnectAsync(context.DnsEndPoint,
cancellation);
            // Or you can create and use your custom DnsRoundRobinConnector instance
            // Socket socket = await s_roundRobinConnector.ConnectAsync(context.DnsEndPoint,
cancellation);
            return new NetworkStream(socket, ownsSocket: true);
        }
    };
    var client = new HttpClient(handler);
    HttpResponseMessage response = await client.GetAsync(Uri);
}
```

예: 속성 설정 `SocketsHttpHandler`

`SocketsHttpHandler` 는 HTTP 연결을 관리하기 위한 고급 구성 옵션을 제공하는 .NET의 강력하고 유연한 처리기입니다. 다양한 속성을 `SocketsHttpHandler`설정하여 성능 최적화, 보안 향상 및 사용자 지정 연결 처리와 같은 특정 요구 사항을 충족하도록 HTTP 클라이언트의 동작을 미세 조정할 수 있습니다.

다양한 속성으로 구성 `SocketsHttpHandler` 하고 다음과 함께 `HttpClient`사용하는 방법의 예는 다음과 같습니다.

c#

```
var cookieContainer = new CookieContainer();
cookieContainer.Add(new Cookie("cookieName", "cookieValue"));

var handler = new SocketsHttpHandler
{
    AllowAutoRedirect = true,
```

```
    AutomaticDecompression = DecompressionMethods.All,
    Expect100ContinueTimeout = TimeSpan.FromSeconds(1),
    CookieContainer = cookieContainer,
    Credentials = new NetworkCredential("user", "pass"),
    MaxAutomaticRedirections = 10,
    MaxResponseHeadersLength = 1,
    Proxy = new WebProxy("http://proxyserver:8080"), // Don't forget to set UseProxy
    UseProxy = true,
};

var client = new HttpClient(handler);
using var response = await client.GetAsync(uri);
```

예시: ImpersonationLevel 변경

이 기능은 특정 플랫폼에 한정되어 있으며 다소 오래된 기능입니다. 해결 방법이 필요한 경우 코드의 이 섹션을 참조하세요.

에서 인증서 및 TLS 관련 속성 사용 `HttpClient`

`HttpClient`로 작업할 때 서버 인증서의 사용자 지정 유효성 검사 또는 서버 인증서 가져오기 등 다양한 목적으로 클라이언트 인증서를 처리해야 할 수 있습니다. `HttpClient`은 인증서를 효과적으로 관리하기 위한 몇 가지 속성과 옵션을 제공합니다.

예: 다음을 사용하여 인증서 해지 목록 확인 `SocketsHttpHandler`

`CheckCertificateRevocationList`의 `SocketsHttpHandler.SslOptions` 속성을 통해 개발자는 SSL/TLS 핸드셰이크 중 인증서 해지 목록(CRL) 확인을 사용하거나 사용하지 않도록 설정할 수 있습니다. 이 속성을 사용하도록 설정하면 클라이언트가 서버의 인증서가 해지되었는지 확인하여 연결의 보안을 강화합니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

C#

```
ServicePointManager.CheckCertificateRevocationList = true;
HttpWebRequest request = WebRequest.CreateHttp(uri);
using HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

`HttpClient`를 사용하는 새 코드:

C#

```
bool checkCertificateRevocationList = true;
var handler = new SocketsHttpHandler
{
    SslOptions =
    {
        CertificateRevocationCheckMode = checkCertificateRevocationList ?
X509RevocationMode.Online : X509RevocationMode.NoCheck,
    }
};
```

```
var client = new HttpClient(handler);
using var response = await client.GetAsync(uri);
```

예시: 인증서 가져오기

`RemoteCertificateValidationCallback`의 `HttpClient`에서 인증서를 가져오려면 `ServerCertificateCustomValidationCallback` 또는 `HttpClientHandler`의 `SocketsHttpHandler.SslOptions` 속성을 사용할 수 있습니다. 이 콜백을 사용하면 SSL/TLS 핸드셰이크 중에 서버의 인증서를 검사할 수 있습니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

C#

```
HttpWebRequest request = WebRequest.CreateHttp(uri);
using HttpWebResponse response = (HttpWebResponse)request.GetResponse();
X509Certificate? serverCertificate = request.ServicePoint.Certificate;
```

`HttpClient`를 사용하는 새 코드:

C#

```
X509Certificate? serverCertificate = null;
var handler = new SocketsHttpHandler
{
    SslOptions = new SslClientAuthenticationOptions
    {
        RemoteCertificateValidationCallback = (sender, certificate, chain, sslPolicyErrors) =>
        {
            serverCertificate = certificate;

            // Leave the validation as-is.
            return sslPolicyErrors == SslPolicyErrors.None;
        }
    }
};
var client = new HttpClient(handler);
using var response = await client.GetAsync("https://example.com");
```

예시: 상호 인증 사용 설정

양방향 SSL 또는 클라이언트 인증서 인증이라고도 하는 상호 인증은 클라이언트와 서버가 서로를 인증하는 보안 프로세스입니다. 이렇게 하면 두 당사자가 모두 자신이 주장하는 당사자임을 보장하여 민감한 커뮤니케이션에 대한 추가적인 보안 계층을 제공합니다. `HttpClient`에서 클라이언트 인증서를 포함하도록 `HttpClientHandler` 또는 `SocketsHttpHandler`을 구성하고 서버의 인증서를 유효성 검사하여 상호 인증을 활성화할 수 있습니다.

상호 인증을 사용하려면 다음 단계를 따르세요.

- 클라이언트 인증서를 로드합니다.
- 클라이언트 인증서를 `HttpClientHandler` 구성하거나 `SocketsHttpHandler` 포함하도록 합니다.
- 사용자 지정 유효성 검사가 필요한 경우 서버 인증서 유효성 검사 콜백을 설정합니다.

다음은 `SocketsHttpHandler` 사용하는 예제입니다.

C#

```
var handler = new SocketsHttpHandler
{
    SslOptions = new SslClientAuthenticationOptions
    {
        ClientCertificates = new X509CertificateCollection
        {
            // Load the client certificate from a file
            new X509Certificate2("path_to_certificate.pfx", "certificate_password")
        },
        RemoteCertificateValidationCallback = (sender, certificate, chain, sslPolicyErrors) =>
        {
            // Custom validation logic for the server certificate
            return sslPolicyErrors == SslPolicyErrors.None;
        }
    }
};

var client = new HttpClient(handler);
using var response = await client.GetAsync(uri);
```

헤더 속성 사용

헤더는 HTTP 통신에서 중요한 역할을 하며 요청과 응답에 대한 필수 메타데이터를 제공합니다. .NET에서 `HttpClient`로 작업할 때 다양한 헤더 속성을 설정하고 관리하여 HTTP 요청 및 응답의 동작을 제어할 수 있습니다. 이러한 헤더 속성을 효과적으로 사용하는 방법을 해석하면 애플리케이션이 웹 서비스와 효율적이고 안전하게 통신하는 데 도움이 될 수 있습니다.

요청 헤더 설정

요청 헤더는 서버에 요청에 대한 추가 정보를 제공하는 데 사용됩니다. 일반적인 사용 사례로는 콘텐츠 유형 지정, 인증 토큰 설정, 사용자 지정 헤더 추가 등이 있습니다. `DefaultRequestHeaders` 속성 또는 `HttpClient` 속성을 사용하여 `Headers`의 `HttpRequestMessage` 요청 헤더를 설정할 수 있습니다.

예시: 사용자 지정 요청 헤더 설정

에서 기본 사용자 지정 요청 헤더 설정 `HttpClient`

C#

```
var client = new HttpClient();
client.DefaultRequestHeaders.Add("Custom-Header", "value");
```

에서 사용자 지정 요청 헤더 설정 `HttpRequestMessage`

C#

```
using var request = new HttpRequestMessage(HttpMethod.Get, uri);
request.Headers.Add("Custom-Header", "value");
```

예시: 공통 요청 헤더 설정

.NET에서 `HttpRequestMessage`로 작업할 때 서버에 요청에 대한 추가 정보를 제공하려면 공통 요청 헤더를 설정하는 것이 필수적입니다. 이러한 헤더에는 인증 토큰 등이 포함될 수 있습니다. 이러한 헤더를 올바르게 구성하면 서버에서 HTTP 요청을 올바르게 처리할 수 있습니다. HttpRequestHeaders에서 사용할 수 있는 공통 속성의 전체 목록은 속성을 참조하세요.

`HttpRequestMessage`에서 공통 요청 헤더를 설정하려면 `Headers` 객체의 `HttpRequestMessage` 속성을 사용하면 됩니다. 이 속성은 필요에 따라 헤더를 추가하거나 수정할 수 있는 `HttpRequestHeaders` 컬렉션에 대한 액세스를 제공합니다.

에서 일반적인 기본 요청 헤더 설정 `HttpClient`

C#

```
using System.Net.Http.Headers;

var client = new HttpClient();
client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", "token");
```

에서 일반 요청 헤더 설정 `HttpRequestMessage`

C#

```
using System.Net.Http.Headers;

using var request = new HttpRequestMessage(HttpMethod.Get, uri);
request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", "token");
```

예시: 콘텐츠 헤더 설정

콘텐츠 헤더는 HTTP 요청 또는 응답의 본문에 대한 추가 정보를 제공하는 데 사용됩니다. .NET에서 `HttpClient`로 작업할 때 콘텐츠 헤더를 설정하여 전송 또는 수신되는 콘텐츠와 관련된 미디어 유형, 인코딩 및 기타 메타데이터를 지정할 수 있습니다. 콘텐츠 헤더를 올바르게 구성하면 서버와 클라이언트가 콘텐츠를 올바르게 해석하고 처리할 수 있습니다.

C#

```
var client = new HttpClient();
using var request = new HttpRequestMessage(HttpMethod.Post, uri);

// Create the content and set the content headers
var jsonData = "{\"key\":\"value\"}";
var content = new StringContent(jsonData, Encoding.UTF8, "application/json");

// The following headers are set automatically by `StringContent`. If you wish to override their
values, you can do it like so:
// content.Headers.ContentType = new MediaTypeHeaderValue("application/json; charset=utf-8");
```

```
// content.Headers.ContentLength = Encoding.UTF8.GetByteCount(jsonData);

// Assign the content to the request
request.Content = content;

using var response = await client.SendAsync(request);
```

예: `MaximumErrorResponseLength`에서 `HttpClient` 설정

`MaximumErrorResponseLength`을 사용하면 개발자가 핸들러가 버퍼링할 오류 응답 콘텐츠의 최대 길이를 지정할 수 있습니다. 이는 서버에서 오류 응답이 수신될 때 메모리에 읽고 저장되는 데이터의 양을 제어하는 데 유용합니다. 이 기술을 사용하면 대용량 오류 응답을 처리할 때 과도한 메모리 사용을 방지하고 애플리케이션의 성능을 개선할 수 있습니다.

이를 수행하는 방법에는 몇 가지가 있는데, 이 예제에서는 `TruncatedReadStream` 기법을 살펴보겠습니다.

C#

```
internal sealed class TruncatedReadStream(Stream innerStream, long maxSize) : Stream
{
    private long _maxRemainingLength = maxSize;
    public override bool CanRead => true;
    public override bool CanSeek => false;
    public override bool CanWrite => false;

    public override long Length => throw new NotSupportedException();
    public override long Position { get => throw new NotSupportedException(); set => throw new
NotSupportedException(); }

    public override void Flush() => throw new NotSupportedException();

    public override int Read(byte[] buffer, int offset, int count)
    {
        return Read(new Span<byte>(buffer, offset, count));
    }

    public override int Read(Span<byte> buffer)
    {
        int readBytes = innerStream.Read(buffer.Slice(0, (int)Math.Min(buffer.Length,
_maxRemainingLength)));
        _maxRemainingLength -= readBytes;
        return readBytes;
    }

    public override Task<int> ReadAsync(byte[] buffer, int offset, int count, CancellationToken
cancellationToken)
    {
        return ReadAsync(new Memory<byte>(buffer, offset, count), cancellationToken).AsTask();
    }

    public override async ValueTask<int> ReadAsync(Memory<byte> buffer, CancellationToken
cancellationToken = default)
    {
        int readBytes = await innerStream.ReadAsync(buffer.Slice(0, (int)Math.Min(buffer.Length,
_maxRemainingLength)), cancellationToken)
            .ConfigureAwait(false);
        _maxRemainingLength -= readBytes;
        return readBytes;
```

```
    }

    public override long Seek(long offset, SeekOrigin origin) => throw new
NotSupportedException();
    public override void SetLength(long value) => throw new NotSupportedException();
    public override void Write(byte[] buffer, int offset, int count) => throw new
NotSupportedException();

    public override ValueTask DisposeAsync() => innerStream.DisposeAsync();

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            innerStream.Dispose();
        }
    }
}
```

그리고 `TruncatedReadStream`의 사용 예시입니다.

C#

```
int maxErrorResponseLength = 1 * 1024; // 1 KB

HttpClient client = new HttpClient();
using HttpResponseMessage response = await client.GetAsync(Uri);

if (response.Content is not null)
{
    Stream responseReadStream = await response.Content.ReadAsStreamAsync();
    // If MaxErrorResponseLength is set and the response status code is an error code, then wrap
the response stream in a TruncatedReadStream
    if (maxErrorResponseLength >= 0 && !response.IsSuccessStatusCode)
    {
        responseReadStream = new TruncatedReadStream(responseReadStream, maxErrorResponseLength);
    }
    // Read the response stream
    Memory<byte> buffer = new byte[1024];
    int readValue = await responseReadStream.ReadAsync(buffer);
}
```

예: 헤더 적용 `CachePolicy`

> ⚠ **Warning**
>
> `HttpClient`에는 응답을 캐시하는 기본 제공 논리가 없습니다. 모든 캐싱을 직접 구현하는 것 외에는 해결 방법이 없습니다. 헤더를 설정하기만 하면 캐싱이 수행되지 않습니다.

`HttpWebRequest`에서 `HttpClient`로 마이그레이션할 때 `pragma` 및 `cache-control`와 같은 캐시 관련 헤더를 올바르게 처리하는 것이 중요합니다. 이러한 헤더는 응답이 캐시되고 검색되는 방식을 제어하여 애플리케이션이 성능 및 데이터 최신성 측면에서 예상대로 작동하도록 합니다.

`HttpWebRequest`에서 `CachePolicy` 속성을 사용하여 이러한 헤더를 설정했을 수 있습니다. 그러나 `HttpClient`에서는 요청에서 이러한 헤더를 수동으로 설정해야 합니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

C#

```
HttpWebRequest request = WebRequest.CreateHttp(uri);
request.CachePolicy = new HttpRequestCachePolicy(HttpRequestCacheLevel.NoCacheNoStore);
using HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

이전 `HttpWebRequest` API에서는 이 기능이 기본적으로 지원되었기 때문에 `CachePolicy`를 적용하는 것이 간단했습니다. 그러나 최신 `HttpClient` API는 동일한 기본 제공 기능을 제공하지 않습니다. 그럼에도 불구하고 캐시 관련 헤더를 수동으로 추가하는 `AddCacheControlHeaders`을 구현하여 유사한 동작을 얻을 수 있습니다.

`HttpClient`를 사용하는 새 코드:

C#

```
public static class CachePolicy
{
    public static void AddCacheControlHeaders(HttpRequestMessage request, RequestCachePolicy
policy)
```

구현 `AddCacheControlHeaders`은 AddCacheControlHeaders.cs 참조하세요.

`AddCacheControlHeaders` 사용법:

C#

```
static async Task AddCacheControlHeaders()
{
    HttpClient client = new HttpClient();
    HttpRequestMessage requestMessage = new HttpRequestMessage(HttpMethod.Get, Uri);
    CachePolicy.AddCacheControlHeaders(requestMessage, new
HttpRequestCachePolicy(HttpRequestCacheLevel.NoCacheNoStore));
    HttpResponseMessage response = await client.SendAsync(requestMessage);
}
```

버퍼링 속성 사용

마이그레이션할 `HttpWebRequest` `HttpClient`때는 이러한 두 API가 버퍼링을 처리하는 방식의 차이점을 이해하는 것이 중요합니다.

`HttpWebRequest`을(를) 사용하는 이전 코드:

`HttpWebRequest`에서는 `AllowWriteStreamBuffering` 및 `AllowReadStreamBuffering` 속성을 통해 버퍼링 속성을 직접 제어할 수 있습니다. 이러한 속성은 서버와 주고받는 데이터의 버퍼링을 활성화 또는 비활성화합니다.

C#

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(uri);
request.AllowReadStreamBuffering = true; // Default is `false`.
request.AllowWriteStreamBuffering = false; // Default is `true`.
```

`HttpClient`를 사용하는 새 코드:

`HttpClient`에서 `AllowWriteStreamBuffering` 및 `AllowReadStreamBuffering` 속성과 직접적으로 등가되는 속성은 없습니다.

`HttpClient`는 요청 본문을 자체적으로 버퍼링하지 않고 대신 사용되는 `HttpContent`에 그 책임을 위임합니다. 기본적으로 `StringContent`는 사용 시 버퍼링이 전혀 발생하지 않는 반면, `ByteArrayContent` 또는 `StreamContent`과 같은 콘텐츠는 이미 메모리에 버퍼링되어 있습니다. 콘텐츠를 강제로 버퍼링하려면 요청을 보내기 전에 `HttpContent.LoadIntoBufferAsync`을 호출하면 됩니다. 예를 들어 다음과 같습니다.

C#

```
HttpClient client = new HttpClient();

using HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Post, uri);
request.Content = new StreamContent(yourStream);
await request.Content.LoadIntoBufferAsync();

using HttpResponseMessage response = await client.SendAsync(request);
```

`HttpClient`에서는 읽기 버퍼링이 기본적으로 활성화되어 있습니다. 이를 방지하려면 `HttpCompletionOption.ResponseHeadersRead` 플래그를 지정하거나 `GetStreamAsync` 도우미를 사용할 수 있습니다.

C#

```
HttpClient client = new HttpClient();

using HttpResponseMessage response = await client.GetAsync(uri,
HttpCompletionOption.ResponseHeadersRead);
await using Stream responseStream = await response.Content.ReadAsStreamAsync();

// Or simply
await using Stream responseStream = await client.GetStreamAsync(uri);
```

Last updated on 2026. 02. 24.

HttpClient 클래스를 사용하여 HTTP 요청 만들기

이 문서에서는 HTTP 요청을 수행하고 `HttpClient` 클래스를 사용하여 응답을 처리하는 방법을 알아봅니다.

> **ⓘ Important**
>
> 이 문서의 모든 예제 HTTP 요청은 다음 URL 중 하나를 대상으로 합니다.
>
> - https://jsonplaceholder.typicode.com: 테스트 및 프로토타입 생성을 위한 무료 Fake API 플랫폼을 제공하는 사이트입니다.
> - https://www.example.com: 문서의 설명 예제에서 사용할 수 있는 도메인입니다.

HTTP 엔드포인트는 일반적으로 JSON(JavaScript Object Notation) 데이터를 반환하지만 항상 그런 것은 아닙니다. 편의를 위해 선택적 System.Net.Http.Json NuGet 패키지는 `HttpClient` NuGet 패키지를 사용하여 `HttpContent` 및 객체에 대해 자동 직렬화 및 역직렬화를 수행하는 몇 가지 확장 메서드를 제공합니다. 이 문서의 예제에서는 이러한 확장을 사용할 수 있는 위치에 주의를 기울입니다.

> **팁**
>
> 이 문서에서 참조하는 모든 소스 코드는 GitHub: .NET Docs 리포지토리에서 사용할 수 있습니다.

HttpClient 개체 만들기

이 문서의 대부분의 예제는 동일한 `HttpClient` 인스턴스를 다시 사용하므로 인스턴스를 한 번 구성하고 나머지 예제에 사용할 수 있습니다. `HttpClient` 개체를 만들려면 `HttpClient` 클래스 생성자를 사용합니다. 자세한 내용은 HttpClient 사용 지침을 참조하세요.

C#

```
// HttpClient lifecycle management best practices:
// https://learn.microsoft.com/dotnet/fundamentals/networking/http/httpclient-guidelines#recommended-use
private static HttpClient sharedClient = new()
{
    BaseAddress = new Uri("https://jsonplaceholder.typicode.com"),
};
```

코드는 다음 작업을 완료합니다.

- 새 `HttpClient` 인스턴스를 `static` 변수로 인스턴스화합니다. 지침에 따라 애플리케이션 수명 주기 동안 `HttpClient` 인스턴스를 다시 사용하는 것이 좋습니다.
- HttpClient.BaseAddress 속성을 `"https://jsonplaceholder.typicode.com"`으로 설정합니다.

이 `HttpClient` 인스턴스는 기본 주소를 사용하여 후속 요청을 수행합니다. 다른 구성을 적용하려면 다음 API를 고려합니다.

- HttpClient.DefaultRequestHeaders 속성을 설정합니다.
- 기본이 아닌 HttpClient.Timeout 속성을 적용합니다.
- HttpClient.DefaultRequestVersion 속성을 지정합니다.

> **팁**
>
> 또는 여러 클라이언트를 구성하고 종속성 주입 서비스로 사용할 수 있는 팩터리 패턴 접근 방식을 사용하여 `HttpClient` 인스턴스를 만들 수 있습니다. 자세한 내용은 **.NET이 포함된 HTTP 클라이언트 팩터리**를 참조하세요.

HTTP 요청 수행

HTTP 요청을 수행하려면 다음 API 메서드를 호출합니다.

테이블 확장

HTTP 메서드	응용 프로그램 인터페이스 (API)
`GET`	HttpClient.GetAsync
`GET`	HttpClient.GetByteArrayAsync
`GET`	HttpClient.GetStreamAsync
`GET`	HttpClient.GetStringAsync
`POST`	HttpClient.PostAsync
`PUT`	HttpClient.PutAsync
`PATCH`	HttpClient.PatchAsync
`DELETE`	HttpClient.DeleteAsync
†`USER SPECIFIED`	HttpClient.SendAsync

> †`USER SPECIFIED` 요청은 `SendAsync` 메서드가 유효한 HttpMethod 개체를 허용한다는 것을 나타냅니다.

> **⚠ Warning**
>
> HTTP 요청 만들기는 네트워크 I/O 바인딩된 작업으로 간주됩니다. 동기 **HttpClient.Send** 메서드가 있지만, 그럴 만한 이유가 없다면 비동기 API를 대신 사용하는 것이 좋습니다.

> **ⓘ 참고 항목**
>
> Android 디바이스(예: .NET MAUI 개발)를 대상으로 하는 동안 `android:usesCleartextTraffic="true"` 파일의 `<application></application>` 섹션에 정의를 추가해야 합니다. 이 설정을 사용하면 HTTP 요청과 같은 텍스트 지우기 트래픽을 사용할 수 있으며, 그렇지 않으면 Android 보안 정책으로 인해 기본적으로 사용하지 않도록 설정됩니다. 다음 XML 설정 예시를 살펴보세요:
>
> XML
>
> ```xml
> <?xml version="1.0" encoding="utf-8"?>
> <manifest xmlns:android="http://schemas.android.com/apk/res/android">
>   <application android:usesCleartextTraffic="true"></application>
>   <!-- omitted for brevity -->
> </manifest>
> ```
>
> 자세한 내용은 **로컬호스트 도메인에 일반 텍스트 네트워크 트래픽 활성화하기**를 참조하세요.

HTTP 콘텐츠 이해

HttpContent 형식은 HTTP 엔터티 본문 및 해당 콘텐츠 헤더를 나타내는 데 사용됩니다. 본문(`POST`, `PUT`, `PATCH`)이 필요한 HTTP 메서드(또는 요청 메서드)의 경우 HttpContent 클래스를 사용하여 요청 본문을 지정합니다. 대부분의 예제에서는 JSON 페이로드를 사용하여 StringContent 서브클래스를 준비하는 방법을 보여 주지만 다른 콘텐츠(MIME) 형식에 대한 다른 서브클래스도 존재합니다.

- ByteArrayContent: 바이트 배열에 따라 HTTP 콘텐츠를 제공합니다.
- FormUrlEncodedContent: `"application/x-www-form-urlencoded"` MIME 형식을 사용하여 인코딩된 이름/값 튜플에 대한 HTTP 콘텐츠를 제공합니다.
- JsonContent: JSON에 따라 HTTP 콘텐츠를 제공합니다.

- MultipartContent: `"multipart/*"` MIME 형식 사양을 사용하여 serialize되는 HttpContent 개체의 컬렉션을 제공합니다.
- MultipartFormDataContent: `"multipart/form-data"` MIME 형식을 사용하여 인코딩된 콘텐츠에 대한 컨테이너를 제공합니다.
- ReadOnlyMemoryContent: ReadOnlyMemory<T> 값을 기반으로 HTTP 콘텐츠를 제공합니다.
- StreamContent: 스트림에 따라 HTTP 콘텐츠를 제공합니다.
- StringContent: 문자열에 따라 HTTP 콘텐츠를 제공합니다.

`HttpContent` 클래스는 HttpResponseMessage 속성에서 액세스할 수 있는 HttpResponseMessage.Content 클래스의 응답 본문을 나타내는 데도 사용됩니다.

HTTP GET 요청 사용

`GET` 요청은 본문을 보내면 안 됩니다. 이 요청은 메서드 이름에서 알 수 있듯이 리소스에서 데이터를 검색(또는 가져오기)하는 데 사용됩니다. `GET` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.GetAsync 메서드를 사용합니다.

C#

```
static async Task GetAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await httpClient.GetAsync("todos/3");

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output:
    //   GET https://jsonplaceholder.typicode.com/todos/3 HTTP/1.1
    //   {
    //     "userId": 1,
    //     "id": 3,
    //     "title": "fugiat veniam minus",
    //     "completed": false
    //   }
}
```

코드는 다음 작업을 완료합니다.

- `GET` 요청을 `"https://jsonplaceholder.typicode.com/todos/3"` 엔드포인트로 합니다.
- 응답이 성공하는지 확인합니다.
- 콘솔에 요청 세부 정보를 씁니다.
- 응답 본문을 문자열로 읽습니다.

- 콘솔에 JSON 응답 본문을 씁니다.

`WriteRequestToConsole` 메서드는 프레임워크의 일부가 아닌 사용자 지정 확장입니다. 구현에 대해 궁금한 경우 다음 C# 코드를 고려하세요.

C#

```
static class HttpResponseMessageExtensions
{
    internal static void WriteRequestToConsole(this HttpResponseMessage response)
    {
        if (response is null)
        {
            return;
        }

        var request = response.RequestMessage;
        Console.Write($"{request?.Method} ");
        Console.Write($"{request?.RequestUri} ");
        Console.WriteLine($"HTTP/{request?.Version}");
    }
}
```

이 기능은 다음 형식으로 콘솔에 요청 세부 정보를 작성하는 데 사용됩니다.

`<HTTP Request Method> <Request URI> <HTTP/Version>`

예를 들어 `GET` 엔드포인트에 대한 `"https://jsonplaceholder.typicode.com/todos/3"` 요청은 다음 메시지를 출력합니다.

출력

```
GET https://jsonplaceholder.typicode.com/todos/3 HTTP/1.1
```

JSON에서 HTTP GET 요청 만들기

https://jsonplaceholder.typicode.com/todos 엔드포인트는 `Todo` 개체의 JSON 배열을 반환합니다. 해당 JSON 구조는 다음 형식과 유사합니다.

JSON

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "example title",
    "completed": false
  },
```

```
  {
    "userId": 1,
    "id": 2,
    "title": "another example title",
    "completed": true
  },
]
```

C# `Todo` 개체는 다음과 같이 정의됩니다.

C#

```
public record class Todo(
    int? UserId = null,
    int? Id = null,
    string? Title = null,
    bool? Completed = null);
```

`record class` 형식이며 선택적 `Id` `Title`, `Completed` 및 `UserId` 속성이 있습니다. `record` 형식에 대한 자세한 내용은 C#의 레코드 형식 소개를 참조하세요. `GET` 요청을 강력한 형식의 C# 개체로 자동으로 역직렬화하려면 GetFromJsonAsync NuGet 패키지의 일부인 📦 확장 메서드를 사용합니다.

C#

```
static async Task GetFromJsonAsync(HttpClient httpClient)
{
    var todos = await httpClient.GetFromJsonAsync<List<Todo>>(
        "todos?userId=1&completed=false");

    Console.WriteLine("GET https://jsonplaceholder.typicode.com/todos?
userId=1&completed=false HTTP/1.1");
    todos?.ForEach(Console.WriteLine);
    Console.WriteLine();

    // Expected output:
    //   GET https://jsonplaceholder.typicode.com/todos?userId=1&completed=false
HTTP/1.1
    //   Todo { UserId = 1, Id = 1, Title = delectus aut autem, Completed = False }
    //   Todo { UserId = 1, Id = 2, Title = quis ut nam facilis et officia qui,
Completed = False }
    //   Todo { UserId = 1, Id = 3, Title = fugiat veniam minus, Completed = False
}
    //   Todo { UserId = 1, Id = 5, Title = laboriosam mollitia et enim quasi
adipisci quia provident illum, Completed = False }
    //   Todo { UserId = 1, Id = 6, Title = qui ullam ratione quibusdam voluptatem
quia omnis, Completed = False }
    //   Todo { UserId = 1, Id = 7, Title = illo expedita consequatur quia in,
Completed = False }
    //   Todo { UserId = 1, Id = 9, Title = molestiae perspiciatis ipsa, Completed
= False }
```

```
    //   Todo { UserId = 1, Id = 13, Title = et doloremque nulla, Completed = False
}
    //   Todo { UserId = 1, Id = 18, Title = dolorum est consequatur ea mollitia in
culpa, Completed = False }
}
```

코드는 다음 작업을 완료합니다.

- `GET` 요청을 `"https://jsonplaceholder.typicode.com/todos?userId=1&completed=false"`에 만듭니다.

  쿼리 문자열은 요청에 대한 필터링 조건을 나타냅니다. 명령이 성공하면 응답이 자동으로 `List<Todo>` 개체로 역직렬화됩니다.

- 각 `Todo` 개체와 함께 요청 세부 정보를 콘솔에 씁니다.

HTTP POST 요청 사용

`POST` 요청은 처리를 위해 데이터를 서버로 보냅니다. 요청의 `Content-Type` 헤더는 본문이 보내는 MIME 형식을 의미합니다. `POST` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.PostAsync 메서드를 사용합니다.

C#

```
static async Task PostAsync(HttpClient httpClient)
{
    using StringContent jsonContent = new(
        JsonSerializer.Serialize(new
        {
            userId = 77,
            id = 1,
            title = "write code sample",
            completed = false
        }),
        Encoding.UTF8,
        "application/json");

    using HttpResponseMessage response = await httpClient.PostAsync(
        "todos",
        jsonContent);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output:
    //   POST https://jsonplaceholder.typicode.com/todos HTTP/1.1
```

```
    //   {
    //     "userId": 77,
    //     "id": 201,
    //     "title": "write code sample",
    //     "completed": false
    //   }
}
```

코드는 다음 작업을 완료합니다.

- 요청의 JSON 본문(StringContentMIME 형식)을 사용하여 `"application/json"` 인스턴스를 준비합니다.
- `POST` 요청을 `"https://jsonplaceholder.typicode.com/todos"` 엔드포인트로 합니다.
- 응답이 성공하는지 확인하고 요청 세부 정보를 콘솔에 씁니다.
- 응답 본문을 콘솔에 문자열로 씁니다.

HTTP POST 요청을 JSON으로 만들기

요청 인수를 자동으로 직렬화 `POST` 하고 응답을 강력한 형식의 C# 개체로 역직렬화하려면 PostAsJsonAsync NuGet 패키지의 일부인 확장 메서드와 ReadFromJsonAsync 확장 메서드를 각각 사용합니다.

C#

```
static async Task PostAsJsonAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await httpClient.PostAsJsonAsync(
        "todos",
        new Todo(UserId: 9, Id: 99, Title: "Show extensions", Completed: false));

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var todo = await response.Content.ReadFromJsonAsync<Todo>();
    Console.WriteLine($"{todo}\n");

    // Expected output:
    //   POST https://jsonplaceholder.typicode.com/todos HTTP/1.1
    //   Todo { UserId = 9, Id = 201, Title = Show extensions, Completed = False }
}
```

코드는 다음 작업을 완료합니다.

- `Todo` 인스턴스를 JSON으로 직렬화하고 `POST` 엔드포인트에 `"https://jsonplaceholder.typicode.com/todos"` 요청을 합니다.
- 응답이 성공하는지 확인하고 요청 세부 정보를 콘솔에 씁니다.
- 응답 본문을 `Todo` 인스턴스로 역직렬화하고 `Todo` 개체를 콘솔에 씁니다.

HTTP PUT 요청 사용

`PUT` 요청 메서드는 기존 리소스를 대체하거나 요청 본문 페이로드를 사용하여 새 리소스를 만듭니다. `PUT` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.PutAsync 메서드를 사용합니다.

C#

```
static async Task PutAsync(HttpClient httpClient)
{
    using StringContent jsonContent = new(
        JsonSerializer.Serialize(new
        {
            userId = 1,
            id = 1,
            title = "foo bar",
            completed = false
        }),
        Encoding.UTF8,
        "application/json");

    using HttpResponseMessage response = await httpClient.PutAsync(
        "todos/1",
        jsonContent);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output:
    //   PUT https://jsonplaceholder.typicode.com/todos/1 HTTP/1.1
    //   {
    //     "userId": 1,
    //     "id": 1,
    //     "title": "foo bar",
    //     "completed": false
    //   }
}
```

코드는 다음 작업을 완료합니다.

- 요청의 JSON 본문(StringContentMIME 형식)을 사용하여 `"application/json"` 인스턴스를 준비합니다.
- `PUT` 요청을 `"https://jsonplaceholder.typicode.com/todos/1"` 엔드포인트로 합니다.
- 응답이 성공하는지 확인하고 JSON 응답 본문을 사용하여 요청 세부 정보를 콘솔에 씁니다.

HTTP PUT 요청을 JSON으로 만들기

요청 인수를 자동으로 직렬화 `PUT` 하고 응답을 강력한 형식의 C# 개체로 역직렬화하려면 PutAsJsonAsync NuGet 패키지의 일부인 확장 메서드와 ReadFromJsonAsync 확장 메서드를 각각 사용합니다.

C#

```
static async Task PutAsJsonAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await httpClient.PutAsJsonAsync(
        "todos/5",
        new Todo(Title: "partially update todo", Completed: true));

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var todo = await response.Content.ReadFromJsonAsync<Todo>();
    Console.WriteLine($"{todo}\n");

    // Expected output:
    //   PUT https://jsonplaceholder.typicode.com/todos/5 HTTP/1.1
    //   Todo { UserId = , Id = 5, Title = partially update todo, Completed = True
}
}
```

코드는 다음 작업을 완료합니다.

- `Todo` 인스턴스를 JSON으로 직렬화하고 `PUT` 엔드포인트에 `"https://jsonplaceholder.typicode.com/todos/5"` 요청을 합니다.
- 응답이 성공하는지 확인하고 요청 세부 정보를 콘솔에 씁니다.
- 응답 본문을 `Todo` 인스턴스로 역직렬화하고 `Todo` 개체를 콘솔에 씁니다.

HTTP PATCH 요청 사용

`PATCH` 요청은 기존 리소스에 대한 부분 업데이트입니다. 이 요청은 새 리소스를 만들지 않으며 기존 리소스를 대체하기 위한 것이 아닙니다. 대신 이 메서드는 리소스를 부분적으로만 업데이트합니다. `PATCH` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.PatchAsync 메서드를 사용합니다.

C#

```
static async Task PatchAsync(HttpClient httpClient)
{
    using StringContent jsonContent = new(
        JsonSerializer.Serialize(new
        {
            completed = true
        }),
```

```
        Encoding.UTF8,
        "application/json");

    using HttpResponseMessage response = await httpClient.PatchAsync(
        "todos/1",
        jsonContent);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output
    //   PATCH https://jsonplaceholder.typicode.com/todos/1 HTTP/1.1
    //   {
    //     "userId": 1,
    //     "id": 1,
    //     "title": "delectus aut autem",
    //     "completed": true
    //   }
}
```

코드는 다음 작업을 완료합니다.

- 요청의 JSON 본문(StringContentMIME 형식)을 사용하여 `"application/json"` 인스턴스를 준비합니다.
- `PATCH` 요청을 `"https://jsonplaceholder.typicode.com/todos/1"` 엔드포인트로 합니다.
- 응답이 성공하는지 확인하고 JSON 응답 본문을 사용하여 요청 세부 정보를 콘솔에 씁니다.

`PATCH` NuGet 패키지에는 `System.Net.Http.Json` 요청에 대한 확장 메서드가 존재하지 않습니다.

HTTP DELETE 요청 사용

`DELETE` 요청은 기존 리소스를 제거하고, 요청은 멱등적이지만 안전하지는 않습니다. 동일한 리소스에 대한 여러 `DELETE` 요청은 동일한 결과를 생성하지만 요청은 리소스의 상태에 영향을 줍니다. `DELETE` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.DeleteAsync 메서드를 사용합니다.

C#

```
static async Task DeleteAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await httpClient.DeleteAsync("todos/1");

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
```

```
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output
    //   DELETE https://jsonplaceholder.typicode.com/todos/1 HTTP/1.1
    //   {}
}
```

코드는 다음 작업을 완료합니다.

- `DELETE` 요청을 `"https://jsonplaceholder.typicode.com/todos/1"` 엔드포인트로 합니다.
- 응답이 성공하는지 확인하고 요청 세부 정보를 콘솔에 씁니다.

> **팁**
>
> `DELETE` 요청과 마찬가지로 `PUT` 요청에 대한 응답은 본문을 포함하거나 포함하지 않을 수 있습니다.

HTTP HEAD 요청 살펴보기

`HEAD` 요청은 `GET` 요청과 유사합니다. 이 요청은 리소스를 반환하는 대신 리소스와 연결된 헤더만 반환합니다. `HEAD` 요청에 대한 응답은 본문을 반환하지 않습니다. `HEAD` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.SendAsync 형식이 HttpMethod설정된 `HttpMethod.Head` 메서드를 사용합니다.

C#

```
static async Task HeadAsync(HttpClient httpClient)
{
    using HttpRequestMessage request = new(
        HttpMethod.Head,
        "https://www.example.com");

    using HttpResponseMessage response = await httpClient.SendAsync(request);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    foreach (var header in response.Headers)
    {
        Console.WriteLine($"{header.Key}: {string.Join(", ", header.Value)}");
    }
    Console.WriteLine();

    // Expected output:
    //   HEAD https://www.example.com/ HTTP/1.1
    //   Accept-Ranges: bytes
    //   Age: 550374
```

```
    //   Cache-Control: max-age=604800
    //   Date: Wed, 10 Aug 2022 17:24:55 GMT
    //   ETag: "3147526947"
    //   Server: ECS, (cha / 80E2)
    //   X-Cache: HIT
}
```

코드는 다음 작업을 완료합니다.

- `HEAD` 요청을 `"https://www.example.com/"` 엔드포인트로 합니다.
- 응답이 성공하는지 확인하고 요청 세부 정보를 콘솔에 씁니다.
- 모든 응답 헤더를 반복하고 각 헤더를 콘솔에 씁니다.

HTTP OPTIONS 요청 살펴보기

`OPTIONS` 요청은 서버 또는 엔드포인트가 지원하는 HTTP 메서드를 식별하는 데 사용됩니다. `OPTIONS` 인스턴스 및 `HttpClient` 개체가 지정된 HTTP Uri 요청을 만들려면 HttpClient.SendAsync 형식이 HttpMethod설정된 `HttpMethod.Options` 메서드를 사용합니다.

C#

```
static async Task OptionsAsync(HttpClient httpClient)
{
    using HttpRequestMessage request = new(
        HttpMethod.Options,
        "https://www.example.com");

    using HttpResponseMessage response = await httpClient.SendAsync(request);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    foreach (var header in response.Content.Headers)
    {
        Console.WriteLine($"{header.Key}: {string.Join(", ", header.Value)}");
    }
    Console.WriteLine();

    // Expected output
    //   OPTIONS https://www.example.com/ HTTP/1.1
    //   Allow: OPTIONS, GET, HEAD, POST
    //   Content-Type: text/html; charset=utf-8
    //   Expires: Wed, 17 Aug 2022 17:28:42 GMT
    //   Content-Length: 0
}
```

코드는 다음 작업을 완료합니다.

- `OPTIONS` 엔드포인트에 `"https://www.example.com/"` HTTP 요청을 보냅니다.

- 응답이 성공하는지 확인하고 요청 세부 정보를 콘솔에 씁니다.
- 모든 응답 콘텐츠 헤더를 반복하고 각 헤더를 콘솔에 씁니다.

HTTP TRACE 요청 살펴보기

`TRACE` 요청은 요청 메시지의 애플리케이션 수준 루프백을 제공하기 때문에 디버깅에 유용할 수 있습니다. HTTP `TRACE` 요청을 만들려면 HttpRequestMessage 형식을 사용하여 `HttpMethod.Trace` 만듭니다.

C#

```
using HttpRequestMessage request = new(
    HttpMethod.Trace,
    "{ValidRequestUri}");
```

> ⊗ **주의**
>
> 모든 HTTP 서버가 `TRACE` HTTP 메서드를 지원하지는 않습니다. 이 메서드는 현명하지 않게 사용하는 경우 보안 취약성을 노출할 수 있습니다. 자세한 내용은 **OWASP(Open Web Application Security Project): 교차 사이트 추적** 을 참조하세요.

HTTP 응답 처리

HTTP 응답을 처리할 때 HttpResponseMessage 형식과 상호 작용합니다. 응답의 유효성을 평가하는 데 여러 멤버가 사용됩니다. HTTP 상태 코드는 HttpResponseMessage.StatusCode 속성에서 사용할 수 있습니다.

클라이언트 인스턴스가 지정된 요청을 전송한다고 가정합니다.

C#

```
using HttpResponseMessage response = await httpClient.SendAsync(request);
```

`response` `OK`(HTTP 상태 코드 200)되도록 하려면 다음 예제와 같이 값을 평가할 수 있습니다.

C#

```
if (response is { StatusCode: HttpStatusCode.OK })
{
    // Omitted for brevity...
}
```

`CREATED`(HTTP 상태 코드 201), `ACCEPTED`(HTTP 상태 코드 202), `NO CONTENT`(HTTP 상태 코드 204) 및 `RESET CONTENT`(HTTP 상태 코드 205)와 같은 성공적인 응답을 나타내는 다른 HTTP 상태 코드가 있습니다. HttpResponseMessage.IsSuccessStatusCode 속성을 사용하여 이러한 코드도 평가할 수 있습니다. 그러면 응답 상태 코드가 200-299 범위 내에 있는지 확인할 수 있습니다.

C#

```
if (response.IsSuccessStatusCode)
{
    // Omitted for brevity...
}
```

프레임워크에서 HttpRequestException 오류를 발생시켜야 한다면, HttpResponseMessage.EnsureSuccessStatusCode() 메서드를 호출할 수 있습니다.

C#

```
response.EnsureSuccessStatusCode();
```

응답 상태 코드가 200-299 범위 내에 있지 않으면 이 코드는 `HttpRequestException` 오류를 throw합니다.

HTTP 유효한 콘텐츠 응답 살펴보기

유효한 응답을 사용하면 Content 속성을 사용하여 응답 본문에 액세스할 수 있습니다. 본문은 스트림, 바이트 배열 또는 문자열로 본문에 액세스하는 데 사용할 수 있는 HttpContent 인스턴스로 사용할 수 있습니다.

다음 코드는 `responseStream` 개체를 사용하여 응답 본문을 읽습니다.

C#

```
await using Stream responseStream =
    await response.Content.ReadAsStreamAsync();
```

다른 개체를 사용하여 응답 본문을 읽을 수 있습니다. `responseByteArray` 개체를 사용하여 응답 본문을 읽습니다.

C#

```
byte[] responseByteArray = await response.Content.ReadAsByteArrayAsync();
```

`responseString` 개체를 사용하여 응답 본문을 읽습니다.

C#

```
string responseString = await response.Content.ReadAsStringAsync();
```

HTTP 엔드포인트가 JSON을 반환하는 것을 알고 있는 경우 System.Net.Http.Json NuGet 패키지를 사용하여 응답 본문을 유효한 C# 개체로 역직렬화할 수 있습니다.

C#

```
T? result = await response.Content.ReadFromJsonAsync<T>();
```

이 코드에서 `result` 값은 형식 `T`로 응답 본문이 역직렬화된 것입니다.

HTTP 오류 처리 사용

HTTP 요청이 실패하면 시스템에서 HttpRequestException 개체를 발생시킵니다. 예외를 처리하는 것만으로는 충분하지 않을 수 있습니다. 처리를 고려할 수 있는 다른 잠재적인 예외가 발생할 수 있습니다. 예를 들어 호출 코드는 요청이 완료되기 전에 취소된 취소 토큰을 사용할 수 있습니다. 이 시나리오에서는 TaskCanceledException 오류를 잡을 수 있습니다.

C#

```
using var cts = new CancellationTokenSource();
try
{
    // Assuming:
    //   httpClient.Timeout = TimeSpan.FromSeconds(10)

    using var response = await httpClient.GetAsync(
        "http://localhost:5001/sleepFor?seconds=100", cts.Token);
}
catch (OperationCanceledException ex) when (cts.IsCancellationRequested)
{
    // When the token has been canceled, it is not a timeout.
    Console.WriteLine($"Canceled: {ex.Message}");
}
```

마찬가지로 HTTP 요청을 수행할 때 HttpClient.Timeout 값을 초과하기 전에 서버가 응답하지 않으면 동일한 예외가 throw됩니다. 이 시나리오에서는 Exception.InnerException 오류를 포착할 때 TaskCanceledException 속성을 평가하여 시간 초과가 발생했는지 식별할 수 있습니다.

C#

```
using var cts = new CancellationTokenSource();
try
{
```

```
    // Assuming:
    //   httpClient.Timeout = TimeSpan.FromSeconds(10)

    using var response = await httpClient.GetAsync(
        "http://localhost:5001/sleepFor?seconds=100", cts.Token);
}
catch (OperationCanceledException ex) when (ex.InnerException is TimeoutException
tex)
{
    // when the time-out occurred. Here the cancellation token has not been
canceled.
    Console.WriteLine($"Timed out: {ex.Message}, {tex.Message}");
}
```

코드에서 내부 예외가 TimeoutException 형식인 경우 시간 초과가 발생하고 취소 토큰이 요청을 취소하지 않습니다.

HttpRequestException 개체를 catch할 때 HTTP 상태 코드를 평가하려면 HttpRequestException.StatusCode 속성을 평가할 수 있습니다.

C#

```
try
{
    // Assuming:
    //   httpClient.Timeout = TimeSpan.FromSeconds(10)

    using var response = await httpClient.GetAsync(
        "http://localhost:5001/doesNotExist");

    response.EnsureSuccessStatusCode();
}
catch (HttpRequestException ex) when (ex is { StatusCode: HttpStatusCode.NotFound
})
{
    // Handle 404
    Console.WriteLine($"Not found: {ex.Message}");
}
```

코드에서 EnsureSuccessStatusCode() 메서드는 응답이 성공하지 못한 경우 예외를 throw하기 위해 호출됩니다. 그런 다음, HttpRequestException.StatusCode 속성이 평가되어 응답이 `404`(HTTP 상태 코드 404)인지 확인합니다. `HttpClient` 개체에는 사용자 대신 `EnsureSuccessStatusCode` 메서드를 암시적으로 호출하는 몇 가지 도우미 메서드가 있습니다.

HTTP 오류 전달의 경우 다음 API를 고려합니다.

- HttpClient.GetByteArrayAsync 메서드
- HttpClient.GetStreamAsync 메서드
- HttpClient.GetStringAsync 메서드

> **팁**
>
> `HttpClient` 형식을 반환하지 않는 HTTP 요청을 만드는 데 사용되는 모든 `HttpResponseMessage` 메서드는 사용자 대신 `EnsureSuccessStatusCode` 메서드를 암시적으로 호출합니다.

이러한 메서드를 호출할 때 `HttpRequestException` 개체를 처리하고 HttpRequestException.StatusCode 속성을 평가하여 응답의 HTTP 상태 코드를 확인할 수 있습니다.

C#

```
try
{
    // These methods will throw HttpRequestException
    // with StatusCode set when the HTTP response status code isn't 2xx:
    //
    //   GetByteArrayAsync
    //   GetStreamAsync
    //   GetStringAsync

    using var stream = await httpClient.GetStreamAsync(
        "https://localhost:5001/doesNotExists");
}
catch (HttpRequestException ex) when (ex is { StatusCode: HttpStatusCode.NotFound
})
{
    // Handle 404
    Console.WriteLine($"Not found: {ex.Message}");
}
```

코드에서 HttpRequestException 개체를 던져야 하는 상황이 있을 수 있습니다. HttpRequestException() 생성자는 공용이며 사용자 지정 메시지와 함께 예외를 throw하는 데 사용할 수 있습니다.

C#

```
try
{
    using var response = await httpClient.GetAsync(
        "https://localhost:5001/doesNotExists");

    // Throw for anything higher than 400.
    if (response is { StatusCode: >= HttpStatusCode.BadRequest })
    {
        throw new HttpRequestException(
            "Something went wrong", inner: null, response.StatusCode);
    }
```

```
}
catch (HttpRequestException ex) when (ex is { StatusCode: HttpStatusCode.NotFound
})
{
    Console.WriteLine($"Not found: {ex.Message}");
}
```

HTTP 프록시 구성

HTTP 프록시는 두 가지 방법 중 하나로 구성할 수 있습니다. 기본값은 HttpClient.DefaultProxy 속성에 지정됩니다. 또는 HttpClientHandler.Proxy 속성에 프록시를 지정할 수 있습니다.

전역 기본 프록시 사용

`HttpClient.DefaultProxy` 속성은 생성자를 통해 전달된 `HttpClient` 개체에 명시적으로 설정된 프록시가 없는 경우 모든 HttpClientHandler 인스턴스에서 사용하는 기본 프록시를 결정하는 정적 속성입니다.

이 속성에서 반환되는 기본 인스턴스는 플랫폼에 따라 다른 규칙 집합에 따라 초기화됩니다.

- **Windows**: 환경 변수에서 프록시 구성을 읽거나 변수가 정의되지 않은 경우 사용자 프록시 설정에서 읽습니다.
- **macOS**: 환경 변수에서 프록시 구성을 읽거나 변수가 정의되지 않은 경우 시스템 프록시 설정에서 읽습니다.
- **Linux**: 환경 변수에서 프록시 구성을 읽거나 변수가 정의되지 않은 경우 모든 주소를 무시하도록 구성되지 않은 인스턴스를 초기화합니다.

Windows에서 프록시 **설정에서 설정을 자동으로 검색** 하도록 설정하면 시스템은 일반적으로 WPAD(웹 프록시 자동 검색) 프로토콜을 사용합니다. 가장 일반적인 구성에서 WPAD는 DNS에서 명명된 `wpad` 호스트를 쿼리한 다음(DNS 검색 접미사를 기반으로 변형을 시도할 수 있음) 종종 같은 `http://wpad/wpad.dat` URL에서 PAC(프록시 자동 구성) 파일을 다운로드합니다. 네트워크 관리자는 DHCP 또는 사용자 지정 PAC URL을 통해 WPAD를 구성할 수도 있으므로 정확한 검색 단계 및 PAC URL은 네트워크 구성에 따라 달라집니다. PAC 파일은 시스템에서 각 URL에 대해 올바른 프록시를 확인하기 위해 평가하는 JavaScript 파일입니다.

Windows 및 Unix 기반 플랫폼에서 `DefaultProxy` 속성 초기화는 다음 환경 변수를 사용합니다.

- `HTTP_PROXY`: HTTP 요청에 사용되는 프록시 서버입니다.
- `HTTPS_PROXY`: HTTPS 요청에 사용되는 프록시 서버입니다.
- `ALL_PROXY`: `HTTP_PROXY` 및/또는 `HTTPS_PROXY` 변수가 정의되지 않은 경우 HTTP 및/또는 HTTPS 요청에 사용되는 프록시 서버입니다.

- `NO_PROXY`: 프록시에서 제외할 호스트 이름의 쉼표로 구분된 목록입니다. 와일드카드에는 별표가 지원되지 않습니다. 하위 도메인과 일치하려는 경우 선행 마침표(.)를 사용합니다. 예: `NO_PROXY=.example.com`(앞에 마침표 포함)은 `www.example.com`과 일치하지만 `example.com`와는 일치하지 않습니다. `NO_PROXY=example.com`(선행 기간 제외)은 `www.example.com`일치하지 않습니다. 이 동작은 나중에 다른 에코시스템과 더 잘 일치하도록 다시 검토될 수 있습니다.

환경 변수가 대/소문자를 구분하는 시스템에서 변수 이름은 모두 소문자 또는 모든 대문자일 수 있습니다. 소문자 이름이 먼저 확인됩니다.

프록시 서버는 호스트 이름 또는 IP 주소일 수 있으며, 필요에 따라 콜론 및 포트 번호가 뒤따를 수도 있고, `http` URL일 수도 있고, 필요에 따라 프록시 인증을 위한 사용자 이름 및 암호를 포함할 수도 있습니다. URL은 `http`아닌 `https`시작해야 하며 호스트 이름, IP 또는 포트 뒤의 텍스트를 포함할 수 없습니다.

클라이언트당 프록시 구성

HttpClientHandler.Proxy 속성은 인터넷 리소스에 대한 요청을 처리하는 데 사용할 WebProxy 개체를 식별합니다. 프록시를 사용하지 않도록 지정하려면 `Proxy` 속성을 GlobalProxySelection.GetEmptyWebProxy() 메서드에서 반환된 프록시 인스턴스로 설정합니다.

로컬 컴퓨터 또는 애플리케이션 구성 파일은 기본 프록시가 사용되도록 지정할 수 있습니다. `Proxy` 속성이 지정된 경우 `Proxy` 속성의 프록시 설정은 로컬 컴퓨터 또는 애플리케이션 구성 파일을 재정의하고 처리기는 지정된 프록시 설정을 사용합니다. 구성 파일에 프록시가 지정되지 않고 `Proxy` 속성이 지정되지 않은 경우 처리기는 로컬 컴퓨터에서 상속된 프록시 설정을 사용합니다. 프록시 설정이 없으면 요청이 서버로 직접 전송됩니다.

HttpClientHandler 클래스는 로컬 컴퓨터 설정에서 상속된 와일드카드 문자를 사용하여 프록시 바이패스 목록을 구문 분석합니다. 예를 들어 `HttpClientHandler` 클래스는 브라우저에서 `"nt*"`의 바이패스 목록을 `"nt.*"`의 정규식으로 구문 분석합니다. 따라서 `http://nt.com` URL은 `HttpClientHandler` 클래스를 사용하여 프록시를 무시합니다.

`HttpClientHandler` 클래스는 로컬 프록시 바이패스를 지원합니다. 클래스는 다음 조건이 충족되는 경우 대상을 로컬로 간주합니다.

- 대상에는 고정 이름(URL에 마침표(.) 없음)이 포함됩니다.
- 대상에 루프백 주소(Loopback 또는 IPv6Loopback)가 포함되거나 대상에 로컬 컴퓨터에 할당된 IPAddress 속성이 포함됩니다.
- 대상의 도메인 접미사는 DomainName 속성에 정의된 대로 로컬 컴퓨터의 도메인 접미사와 일치합니다.

프록시 구성에 대한 자세한 내용은 다음 API를 참조하세요.

- WebProxy.Address 속성
- WebProxy.BypassProxyOnLocal 속성
- WebProxy.BypassArrayList 속성

다음 단계

- .NET의 HTTP 지원
- HttpClient 사용 지침
- .NET이 포함된 HTTP 클라이언트 팩터리
- HttpClient에서 HTTP/3 사용
- HttpRepl을 사용하여 웹 API 테스트

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 05.

IHttpClientFactory의 사용과 .NET

이 문서에서는 `IHttpClientFactory` 인터페이스를 사용하여 DI(종속성 주입), 로깅 및 구성과 같은 다양한 .NET 기본 사항으로 `HttpClient` 형식을 만드는 방법을 알아봅니다. HttpClient 형식은 2012년에 릴리스된 .NET Framework 4.5에서 도입되었습니다. 즉, 그것은 한동안 존재해 왔습니다. `HttpClient`는 HTTP를 요청하고 Uri에 의해 식별된 웹 리소스에서 HTTP 응답을 처리하는 데 사용됩니다. HTTP 프로토콜은 모든 인터넷 트래픽의 대다수를 구성합니다.

모범 사례를 이끌어내는 최신 애플리케이션 개발 원칙에 따라 IHttpClientFactory는 사용자 지정 구성으로 `HttpClient` 인스턴스를 만들 수 있는 팩터리 추상화의 역할을 수행합니다. IHttpClientFactory .NET Core 2.1에서 도입되었습니다. 일반적인 HTTP 기반 .NET 워크로드는 복원력 있고 일시적인 오류 처리 타사 미들웨어를 쉽게 활용할 수 있습니다.

> ⚠ **Warning**
>
> 앱에 쿠키가 필요한 경우 사용하지 않는 IHttpClientFactory것이 좋습니다. HttpMessageHandler 인스턴스를 풀링하면 CookieContainer 개체가 공유됩니다. 예기치 CookieContainer 않은 공유는 애플리케이션의 관련 없는 부분 간에 쿠키를 누출할 수 있습니다. 또한 만료되면 HandlerLifetime 처리기가 재활용되므로 해당 처리기에 저장된 CookieContainer 모든 쿠키가 손실됩니다. 클라이언트를 관리하는 다른 방법은 **HTTP 클라이언트 사용 지침**을 참조하세요.

> ⓘ **Important**
>
> `HttpClient`에서 만든 `IHttpClientFactory` 인스턴스의 수명 관리는 수동으로 만든 인스턴스와 완전히 다릅니다. 전략은 **일시적인** 클라이언트를 `IHttpClientFactory`에서 생성하거나 **장기간 사용 가능한** 클라이언트를 `PooledConnectionLifetime`로 설정하여 사용하는 것입니다. 자세한 내용은 **HttpClient 수명 관리** 섹션 및 **HTTP 클라이언트 사용에 대한 지침**을 참조하세요.

`IHttpClientFactory` 형식

이 문서에서 제공하는 모든 샘플 소스 코드는 Microsoft.Extensions.Http NuGet 패키지를 설치해야 합니다. 이를테면, 코드 예제는 무료 JSON Placeholder API에서 사용자 개체를 검색하기 위한 HTTP 요청 사용법을 보여 줍니다.

AddHttpClient 확장 메서드 중 하나를 호출하면 `IHttpClientFactory` 및 관련 서비스를 IServiceCollection에 추가하게 됩니다. `IHttpClientFactory` 형식은 다음과 같은 이점을 제공합니다.

- `HttpClient` 클래스를 DI 준비 형식으로 노출합니다.
- 논리적 `HttpClient` 인스턴스를 구성하고 이름을 지정하기 위한 중앙 위치를 제공합니다.
- `HttpClient`에서 위임 처리기를 통해 나가는 미들웨어의 개념을 체계화합니다.
- Polly 기반 미들웨어에 대한 확장 메서드를 제공하여 `HttpClient`에서의 처리기 위임을 활용합니다.
- 기본 HttpClientHandler 인스턴스의 캐싱 및 수명을 관리합니다. 자동 관리가 `HttpClient` 수명을 수동으로 관리할 때 발생하는 일반적인 DNS(Domain Name System) 문제를 방지해 줍니다.
- 팩터리에서 만든 클라이언트를 통해 전송된 모든 요청에 대해 구성 가능한 로깅 경험(ILogger을 통해)을 추가합니다.

사용 패턴

앱에서 `IHttpClientFactory`를 사용할 수 있는 몇 가지 방법이 있습니다.

- 기본 사용량
- 명명된 클라이언트
- 형식화된 클라이언트
- 생성된 클라이언트

가장 좋은 방법은 앱의 요구 사항에 따라서 달라집니다.

기본적인 사용 방법

`IHttpClientFactory`를 등록하려면 `AddHttpClient`를 호출합니다.

C#

```
using Shared;
using BasicHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHttpClient();
builder.Services.AddTransient<TodoService>();

using IHost host = builder.Build();
```

서비스를 사용할 때 `IHttpClientFactory`를 포함한 생성자 매개변수를 사용하여 DI가 필요할 수 있습니다. 다음 코드는 `IHttpClientFactory`를 사용하여 `HttpClient` 인스턴스를 만듭니다.

C#

```
using System.Net.Http.Json;
using System.Text.Json;
using Microsoft.Extensions.Logging;
using Shared;

namespace BasicHttp.Example;

public sealed class TodoService(
    IHttpClientFactory httpClientFactory,
    ILogger<TodoService> logger)
{
    public async Task<Todo[]> GetUserTodosAsync(int userId)
    {
        // Create the client
        HttpClient client = httpClientFactory.CreateClient();

        try
        {
            // Make HTTP GET request
            // Parse JSON response deserialize into Todo types
            Todo[]? todos = await client.GetFromJsonAsync<Todo[]>(
                $"https://jsonplaceholder.typicode.com/todos?userId={userId}",
                new JsonSerializerOptions(JsonSerializerDefaults.Web));

            return todos ?? [];
        }
        catch (Exception ex)
        {
            logger.LogError("Error getting something fun to say: {Error}", ex);
        }

        return [];
    }
}
```

앞서 나온 예제에서와 같이 `IHttpClientFactory`를 사용하는 것은 기존 앱을 리팩터링하는 좋은 방법입니다. `HttpClient`가 사용되는 방식에는 어떠한 영향도 없습니다. 기존 앱에서 `HttpClient` 인스턴스가 만들어지는 위치에서 해당 코드를 CreateClient에 대한 호출로 대체합니다.

명명된 클라이언트

명명된 클라이언트는 다음과 같은 경우에 적합합니다.

- 앱에서 `HttpClient`를 서로 다른 곳에서 여러 번 사용해야 합니다.
- 많은 `HttpClient` 인스턴스에 다양한 구성이 있습니다.

명명된 `HttpClient`에 대한 구성은 `IServiceCollection`에서 등록하는 동안 지정할 수 있습니다.

C#

```
using Shared;
using NamedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

string? httpClientName = builder.Configuration["TodoHttpClientName"];
ArgumentException.ThrowIfNullOrEmpty(httpClientName);

builder.Services.AddHttpClient(
    httpClientName,
    client =>
    {
        // Set the base address of the named client.
        client.BaseAddress = new Uri("https://jsonplaceholder.typicode.com/");

        // Add a user-agent default request header.
        client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
    });
```

위의 코드에서 클라이언트가 다음을 사용하여 구성됩니다.

- `"TodoHttpClientName"`의 구성에서 가져온 이름입니다.
- 기본 주소 `https://jsonplaceholder.typicode.com/`.
- `"User-Agent"` 헤더입니다.

구성을 사용하여 HTTP 클라이언트 이름을 지정할 수 있으며, 이는 클라이언트를 추가하고 만들 때 잘못된 명명을 방지하는 데 유용합니다. 이 예제에서는 HTTP 클라이언트 이름을 만드는 데 *appsettings.json* 파일이 사용됩니다.

JSON

```
{
    "TodoHttpClientName": "JsonPlaceholderApi"
}
```

손쉽게 이 구성을 확장하고 HTTP 클라이언트의 작동 방식에 대한 자세한 정보를 저장할 수 있습니다. 자세한 내용은 .NET 구성을 참조하세요.

> ⓘ **참고 항목**
>
> 등록된 고유의 명명된 클라이언트 수는 리소스 소모로 이어질 수 있으므로 바인딩되지 않아야 합니다. 예를 들어 바인딩되지 않은 입력에서 클라이언트 이름을 파생하지 마세요.

클라이언트 만들기

CreateClient가 호출될 때마다

- `HttpClient`의 새 인스턴스가 만들어집니다.
- 구성 작업이 호출됩니다.

명명된 클라이언트를 만들려면 `CreateClient`로 이름을 전달합니다.

C#

```
using System.Net.Http.Json;
using System.Text.Json;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Shared;

namespace NamedHttp.Example;

public sealed class TodoService
{
    private readonly IHttpClientFactory _httpClientFactory = null!;
    private readonly IConfiguration _configuration = null!;
    private readonly ILogger<TodoService> _logger = null!;

    public TodoService(
        IHttpClientFactory httpClientFactory,
        IConfiguration configuration,
        ILogger<TodoService> logger) =>
        (_httpClientFactory, _configuration, _logger) =
            (httpClientFactory, configuration, logger);

    public async Task<Todo[]> GetUserTodosAsync(int userId)
    {
        // Create the client
        string? httpClientName = _configuration["TodoHttpClientName"];
        HttpClient client = _httpClientFactory.CreateClient(httpClientName ?? "");

        try
        {
            // Make HTTP GET request
            // Parse JSON response deserialize into Todo type
            Todo[]? todos = await client.GetFromJsonAsync<Todo[]>(
                $"todos?userId={userId}",
                new JsonSerializerOptions(JsonSerializerDefaults.Web));

            return todos ?? [];
        }
        catch (Exception ex)
        {
            _logger.LogError("Error getting something fun to say: {Error}", ex);
        }
```

```
        return [];
    }
}
```

위의 코드에서는 HTTP 요청이 호스트 이름을 지정할 필요가 없습니다. 클라이언트에 대해 구성된 기본 주소가 사용되었으므로 코드는 경로만 전달할 수 있습니다.

형식화된 클라이언트

형식화된 클라이언트:

- 문자열을 키로 사용할 필요가 없이 명명된 클라이언트와 동일한 기능을 제공합니다.
- 클라이언트를 사용할 때 IntelliSense 및 컴파일러 도움말을 제공합니다.
- 특정 `HttpClient`을 구성하고 상호 작용하기 위해 단일 위치를 제공합니다. 예를 들어, 다음과 같은 경우에 단일 형식화된 클라이언트를 사용할 수 있습니다.
  - 단일 백엔드 엔드포인트에 대해.
  - 엔드포인트를 처리하는 모든 로직을 캡슐화하기 위해.
- DI로 작업하고 앱에서 필요할 경우 삽입할 수 있습니다.

형식화된 클라이언트는 생성자에서 `HttpClient` 매개 변수를 받습니다.

C#

```
using System.Net.Http.Json;
using System.Text.Json;
using Microsoft.Extensions.Logging;
using Shared;

namespace TypedHttp.Example;

public sealed class TodoService(
    HttpClient httpClient,
    ILogger<TodoService> logger)
{
    public async Task<Todo[]> GetUserTodosAsync(int userId)
    {
        try
        {
            // Make HTTP GET request
            // Parse JSON response deserialize into Todo type
            Todo[]? todos = await httpClient.GetFromJsonAsync<Todo[]>(
                $"todos?userId={userId}",
                new JsonSerializerOptions(JsonSerializerDefaults.Web));

            return todos ?? [];
        }
        catch (Exception ex)
        {
            logger.LogError("Error getting something fun to say: {Error}", ex);
```

```
        }

        return [];
    }
}
```

위의 코드에서

- 구성은 형식화된 클라이언트가 서비스 컬렉션에 추가되는 경우에 설정됩니다.
- `HttpClient`는 클래스 범위 변수(필드)로 할당되고 노출된 API와 함께 사용됩니다.

`HttpClient` 기능을 노출하는 API 특정 메서드를 만들 수 있습니다. 예를 들어 `GetUserTodosAsync` 메서드는 사용자별 `Todo` 개체를 검색하는 코드를 캡슐화합니다.

다음 코드는 AddHttpClient를 호출하여 형식화된 클라이언트 클래스를 등록합니다.

C#

```
using Shared;
using TypedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHttpClient<TodoService>(
    client =>
    {
        // Set the base address of the typed client.
        client.BaseAddress = new Uri("https://jsonplaceholder.typicode.com/");

        // Add a user-agent default request header.
        client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
    });
```

형식화된 클라이언트는 DI를 사용하여 일시적으로 등록됩니다. 위의 코드에서 `AddHttpClient`는 `TodoService`를 임시 서비스로 등록합니다. 이 등록에서는 팩터리 메서드를 사용하여 다음을 수행합니다.

1. `HttpClient`의 인스턴스를 만듭니다.
2. `TodoService`의 인스턴스를 생성자에 전달하여 `HttpClient`의 인스턴스를 만듭니다.

> **ⓘ Important**
>
> 싱글톤 서비스에서 형식화된 클라이언트를 사용하는 것은 위험할 수 있습니다. 자세한 내용은 **싱글톤 서비스에서 형식화된 클라이언트 방지** 섹션을 참조하세요.

> ⓘ **참고 항목**
>
> 형식화된 클라이언트를 `AddHttpClient<TClient>` 메서드를 사용하여 등록하는 경우, 매개 변수로서 `TClient`을(를) 허용하는 생성자가 `HttpClient` 형식에 있어야 합니다. 또한, DI 컨테이너에 `TClient` 형식을 별도로 등록하면 안 됩니다. 이러한 경우 이후 등록에서 전자를 덮어쓰게 됩니다.

생성된 클라이언트

`IHttpClientFactory`는 Refit과 같은 타사 라이브러리와 함께 사용할 수 있습니다. Refit은 .NET을 위한 REST 라이브러리입니다. 선언적 REST API 정의가 인터페이스 메서드를 엔드포인트에 매핑할 수 있습니다. 인터페이스의 구현은 `RestService`를 사용하여 외부 HTTP를 호출하도록 `HttpClient`에 의해 동적으로 생성됩니다.

다음 `record` 형식을 고려해 보세요.

C#

```
namespace Shared;

public record class Todo(
    int UserId,
    int Id,
    string Title,
    bool Completed);
```

다음 예에서는 Refit.HttpClientFactory NuGet 패키지를 사용하며 간단한 인터페이스입니다.

C#

```
using Refit;
using Shared;

namespace GeneratedHttp.Example;

public interface ITodoService
{
    [Get("/todos?userId={userId}")]
    Task<Todo[]> GetUserTodosAsync(int userId);
}
```

이전 C# 인터페이스:

- `GetUserTodosAsync` 인스턴스를 반환하는 `Task<Todo[]>`라는 이름의 메서드를 정의합니다.

- 외부 API에 대한 경로 및 쿼리 문자열로 `Refit.GetAttribute` 특성을 선언합니다.

구현을 생성하기 위해 Refit를 사용하여 형식화된 클라이언트를 추가할 수 있습니다.

C#

```
using GeneratedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Refit;
using Shared;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddRefitClient<ITodoService>()
    .ConfigureHttpClient(client =>
    {
        // Set the base address of the typed client.
        client.BaseAddress = new Uri("https://jsonplaceholder.typicode.com/");

        // Add a user-agent default request header.
        client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
    });
```

DI 및 Refit에서 제공한 구현을 통해 필요한 곳에서 정의된 인터페이스를 사용할 수 있습니다.

POST, PUT 및 DELETE 요청 수행

위의 예제에서 모든 HTTP 요청은 `GET` HTTP 동사를 사용합니다. `HttpClient`는 다음을 비롯한 다른 HTTP 동사도 지원합니다.

- `POST`
- `PUT`
- `DELETE`
- `PATCH`

지원되는 HTTP 동사의 전체 목록은 HttpMethod를 참조하세요. HTTP 요청에 대한 자세한 내용은 HttpClient를 사용하여 요청 보내기를 참조하세요.

다음 예제에서는 HTTP `POST` 요청을 수행하는 방법을 보여 줍니다.

C#

```
public async Task CreateItemAsync(Item item)
{
    using StringContent json = new(
```

```
        JsonSerializer.Serialize(item, new
JsonSerializerOptions(JsonSerializerDefaults.Web)),
        Encoding.UTF8,
        MediaTypeNames.Application.Json);

    using HttpResponseMessage httpResponse =
        await httpClient.PostAsync("/api/items", json);

    httpResponse.EnsureSuccessStatusCode();
}
```

위의 코드에서 `CreateItemAsync` 메서드는 다음을 수행합니다.

- `Item`을 사용하여 `System.Text.Json`매개 변수를 JSON으로 serialize합니다. JsonSerializerOptions의 인스턴스를 사용하여 serialization 프로세스를 구성합니다.
- HTTP 요청의 본문에서 전송하기 위해 serialize된 JSON을 패키지할 StringContent의 인스턴스를 만듭니다.
- PostAsync를 호출하여 JSON 콘텐츠를 지정된 URL로 보냅니다. HttpClient.BaseAddress에 추가되는 상대 URL입니다.
- 응답 상태 코드가 성공을 나타내지 않을 경우 EnsureSuccessStatusCode를 호출하여 예외를 throw합니다.

`HttpClient`는 다른 형식의 콘텐츠도 지원합니다. 예를 들어 MultipartContent 및 StreamContent를 지정합니다. 지원되는 콘텐츠의 전체 목록은 HttpContent를 참조하세요.

다음 예제에서는 HTTP `PUT` 요청을 보여 줍니다.

C#

```
public async Task UpdateItemAsync(Item item)
{
    using StringContent json = new(
        JsonSerializer.Serialize(item, new
JsonSerializerOptions(JsonSerializerDefaults.Web)),
        Encoding.UTF8,
        MediaTypeNames.Application.Json);

    using HttpResponseMessage httpResponse =
        await httpClient.PutAsync($"/api/items/{item.Id}", json);

    httpResponse.EnsureSuccessStatusCode();
}
```

앞의 코드는 `POST` 예제와 매우 비슷합니다. `UpdateItemAsync` 메서드는 PutAsync 대신 `PostAsync`를 호출합니다.

다음 예제에서는 HTTP `DELETE` 요청을 보여 줍니다.

C#

```
public async Task DeleteItemAsync(Guid id)
{
    using HttpResponseMessage httpResponse =
        await httpClient.DeleteAsync($"/api/items/{id}");

    httpResponse.EnsureSuccessStatusCode();
}
```

위의 코드에서 `DeleteItemAsync` 메서드는 DeleteAsync를 호출합니다. HTTP `DELETE` 요청에는 일반적으로 본문이 `DeleteAsync` 없기 때문에 메서드는 인스턴스 `HttpContent`를 허용하는 오버로드를 제공하지 않습니다.

`HttpClient`에 다른 HTTP 동사를 사용하는 방법에 대한 자세한 내용은 HttpClient를 참조하세요.

`HttpClient` 수명 관리

`HttpClient`에서 `CreateClient`가 호출될 때마다 새로운 `IHttpClientFactory` 인스턴스가 반환됩니다. 클라이언트마다 HttpClientHandler 인스턴스가 하나씩 만들어집니다. 팩터리는 `HttpClientHandler` 인스턴스의 수명을 관리합니다.

`IHttpClientFactory`는 리소스 사용을 줄이기 위해 팩터리에서 만든 `HttpClientHandler` 인스턴스를 캐시합니다. 수명이 만료되지 않은 경우, 새 `HttpClientHandler` 인스턴스를 만들 때 캐시에서 `HttpClient` 인스턴스가 재사용될 수 있습니다.

일반적으로 각 처리기는 자체적인 기본 HTTP 연결 풀을 관리하므로 처리기의 캐싱이 적합합니다. 필요한 것보다 더 많은 처리기를 만들면 소켓 고갈 및 연결 지연이 발생할 수 있습니다. 또한 일부 처리기는 무한정으로 연결을 열어 놓아 처리기가 DNS 변경에 대응하는 것을 막을 수 있습니다.

기본 처리기 수명은 2분입니다. 기본값을 재정의하려면 SetHandlerLifetime에서 `IHttpClientBuilder`를 등록할 때 각 클라이언트에서 `IHttpClientFactory`를 호출하십시오.

C#

```
services.AddHttpClient("Named.Client")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

> ⓘ **Important**
>
> `HttpClient`에서 만든 `IHttpClientFactory` 인스턴스는 **단기**여야 합니다.

- 처리기가 DNS 변경에 반응하도록 보장하려면, 수명이 만료되면 `HttpMessageHandler`을 재활용하고 `IHttpClientFactory`을 다시 생성해야 합니다. `HttpClient`는 만들어질 때 특정 처리기 인스턴스에 연결되므로 클라이언트가 업데이트된 처리기를 가져올 수 있도록 새 `HttpClient` 인스턴스를 적시에 요청해야 합니다.

- 팩터리에서 만든 이러한 `HttpClient` 인스턴스**를 삭제해도** 소켓 고갈로 이어지지는 않습니다. 이 **인스턴스를 삭제해도** 의 삭제가 트리거`HttpMessageHandler`되지 않기 때문입니다. `IHttpClientFactory`는 `HttpClient` 인스턴스, 특히 `HttpMessageHandler` 인스턴스를 만드는 데 사용되는 리소스를 추적하고 삭제합니다. 이러한 인스턴스는 수명이 곧 만료되고 이러한 인스턴스를 사용하는 `HttpClient`가 더 이상 없기 때문입니다.

단일 `HttpClient` 인스턴스를 장기간 활성 상태로 유지하는 것은 에 대한 **대안**으로 `IHttpClientFactory`사용할 수 있는 일반적인 패턴입니다. 그러나 이 패턴을 사용하려면 `PooledConnectionLifetime`과 같은 추가 설정이 필요합니다. **오래 지속되는** 클라이언트를 `PooledConnectionLifetime`와 함께 사용하거나 **짧게 지속되는** 클라이언트를 `IHttpClientFactory`에 의해 생성하여 사용할 수 있습니다. 앱에서 사용할 전략에 대한 내용은 HTTP 클라이언트 사용에 대한 지침을 참조하세요.

`HttpMessageHandler`을 구성하십시오.

클라이언트가 사용하는 내부 HttpMessageHandler의 구성을 제어해야 할 수도 있습니다.

IHttpClientBuilder는 명명된 또는 형식화된 클라이언트를 추가할 때 반환됩니다. 확장 메서드는 ConfigurePrimaryHttpMessageHandler에서 호출될 수 있고, `IHttpClientBuilder`로 대리자를 전달할 수 있습니다. 대리자는 해당 클라이언트가 사용하는 기본 `HttpMessageHandler`을 만들고 구성하는 데 사용됩니다.

C#

```
.ConfigurePrimaryHttpMessageHandler(() =>
{
    return new HttpClientHandler
    {
        AllowAutoRedirect = false,
        UseDefaultCredentials = true
    };
});
```

`HttpClientHandler`를 구성하면 처리기의 다양한 속성 중에서 `HttpClient` 인스턴스에 대한 프록시를 지정할 수 있습니다. 자세한 내용은 클라이언트당 프록시를 참조하세요.

추가 구성

`IHttpClientBuilder`를 제어하기 위한 몇 가지 추가 구성 옵션이 있습니다.

테이블 확장

메서드	설명
AddHttpMessageHandler	명명된 `HttpClient`에 대한 추가 메시지 처리기를 추가합니다.
AddTypedClient	`TClient`와 연결된 명명된 `HttpClient`와 `IHttpClientBuilder` 간의 바인딩을 구성합니다.
ConfigureHttpClient	명명된 `HttpClient`를 구성하는 데 사용되는 대리자를 추가합니다.
ConfigurePrimaryHttpMessageHandler	명명된 `HttpMessageHandler`에 대해 종속성 주입 컨테이너에서 기본 `HttpClient`를 구성합니다.
RedactLoggedHeaders	로깅하기 전에 값을 수정해야 하는 HTTP 헤더 이름의 컬렉션을 설정합니다.
SetHandlerLifetime	`HttpMessageHandler` 인스턴스를 다시 사용할 수 있는 시간을 설정합니다. 명명된 클라이언트마다 고유하게 구성된 처리기 수명 값이 있을 수 있습니다.
UseSocketsHttpHandler	종속성 주입 컨테이너에서 `SocketsHttpHandler`(으)로 명명된 인스턴스에 대한 기본 처리기로 새 인스턴스 또는 이전에 추가된 `HttpClient` 인스턴스를 사용하도록 구성합니다. (.NET 5 이상에서만 사용 가능)

`IHttpClientFactory` `SocketsHttpHandler` 함께 사용하기

`SocketsHttpHandler` `HttpMessageHandler` 구현이 .NET Core 2.1에 추가되어 `PooledConnectionLifetime` 구성할 수 있습니다. 이 설정은 처리기가 DNS 변경에 반응하도록 하는 데 사용되므로 `SocketsHttpHandler` 사용은 `IHttpClientFactory` 사용의 대안으로 간주됩니다. 자세한 내용은 HTTP 클라이언트 사용 지침을 참조하세요.

그러나 `SocketsHttpHandler` `IHttpClientFactory` 구성 가능성을 개선하기 위해 함께 사용할 수 있습니다. 이러한 API를 모두 사용하면 낮은 수준(예: 동적 인증서 선택에 대해 `LocalCertificateSelectionCallback` 사용) 및 높은 수준(예: DI 통합 및 여러 클라이언트 구성 활용) 둘 다에서 구성할 수 있습니다.

두 API를 모두 사용하려면 다음을 수행합니다.

1. `SocketsHttpHandler` `PrimaryHandler` 통해 ConfigurePrimaryHttpMessageHandler 또는 UseSocketsHttpHandler(.NET 5+만 해당)으로 지정합니다.
2. DNS가 업데이트될 것으로 예상되는 간격에 따라 SocketsHttpHandler.PooledConnectionLifetime을 설정하십시오. 예를 들어, 이는 확장 메서드에서 이전에 설정된 값입니다.
3. (선택 사항) `SocketsHttpHandler`이(가) 연결 풀링 및 재생을 처리하므로 `IHttpClientFactory` 수준에서 처리기 재생이 더 이상 필요하지 않습니다. `HandlerLifetime`을 `Timeout.InfiniteTimeSpan`로 설정하여 사용하지 않도록 설정할 수 있습니다.

C#

```
services.AddHttpClient(name)
    .UseSocketsHttpHandler((handler, _) =>
        handler.PooledConnectionLifetime = TimeSpan.FromMinutes(2)) // Recreate
connection every 2 minutes
    .SetHandlerLifetime(Timeout.InfiniteTimeSpan); // Disable rotation, as it is
handled by PooledConnectionLifetime
```

위의 예제에서 2분은 기본값 `HandlerLifetime`에 맞춰 설명 목적으로 임의로 선택되었습니다. DNS 또는 기타 네트워크 변경의 예상 빈도에 따라 값을 선택해야 합니다. DNS 동작 부분과 `HttpClient` 지침의 PooledConnectionLifetime API 설명서의 설명 섹션을 참조하여 더 많은 정보를 확인하세요.

싱글톤 서비스에서는 형식이 지정된 클라이언트를 방지합니다.

명명된 클라이언트 접근 방식을 사용하는 경우 `IHttpClientFactory`는 서비스에 주입되고 `HttpClient` 인스턴스는 CreateClient가 필요할 때마다 `HttpClient`를 호출하여 생성됩니다.

그러나 *형식화된 클라이언트* 접근 방식을 사용할 경우 형식화된 클라이언트는 일반적으로 서비스에 주입되는 일시적인 개체입니다. 형식화된 클라이언트를 싱글톤 서비스에 삽입할 수 있기 때문에 문제가 발생할 수 있습니다.

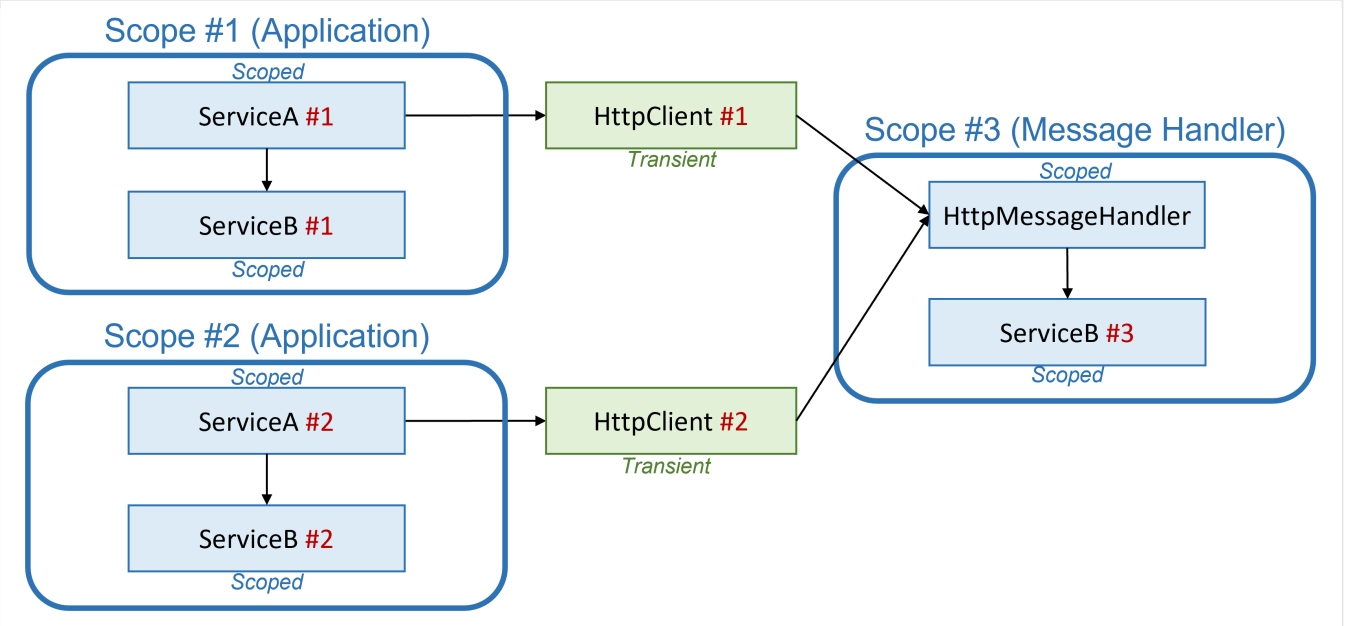
> **ⓘ Important**
>
> 형식화된 클라이언트는 에서 생성된 인스턴스와 마찬가지로 단기적인 것으로 예상됩니다(자세한 내용은 수명 관리를 참조하십시오). 형식화된 클라이언트 인스턴스를 만드는 즉시 `IHttpClientFactory`에서 제어할 수 없게 됩니다. 형식화된 클라이언트 인스턴스가 싱글톤으로 캡처되는 경우 DNS 변경에 반응하지 못하게 하여 `IHttpClientFactory`의 용도 중 하나를 무효화할 수 있습니다.

싱글톤 서비스에서 `HttpClient` 인스턴스를 사용해야 하는 경우 다음 옵션을 고려합니다.

- 대신 *명명된 클라이언트* 접근 방식을 사용하고, 필요한 경우 싱글톤 서비스에 `IHttpClientFactory`를 주입하고 `HttpClient` 인스턴스를 다시 만듭니다.
- *형식화된 클라이언트* 접근 방식이 필요한 경우 기본 처리기로 구성한 `SocketsHttpHandler`과 함께 `PooledConnectionLifetime`를 사용합니다. `SocketsHttpHandler`에서 `IHttpClientFactory`를 사용하는 방법에 대한 자세한 내용은 SocketsHttpHandler와 함께 IHttpClientFactory 사용 섹션을 참조하세요.

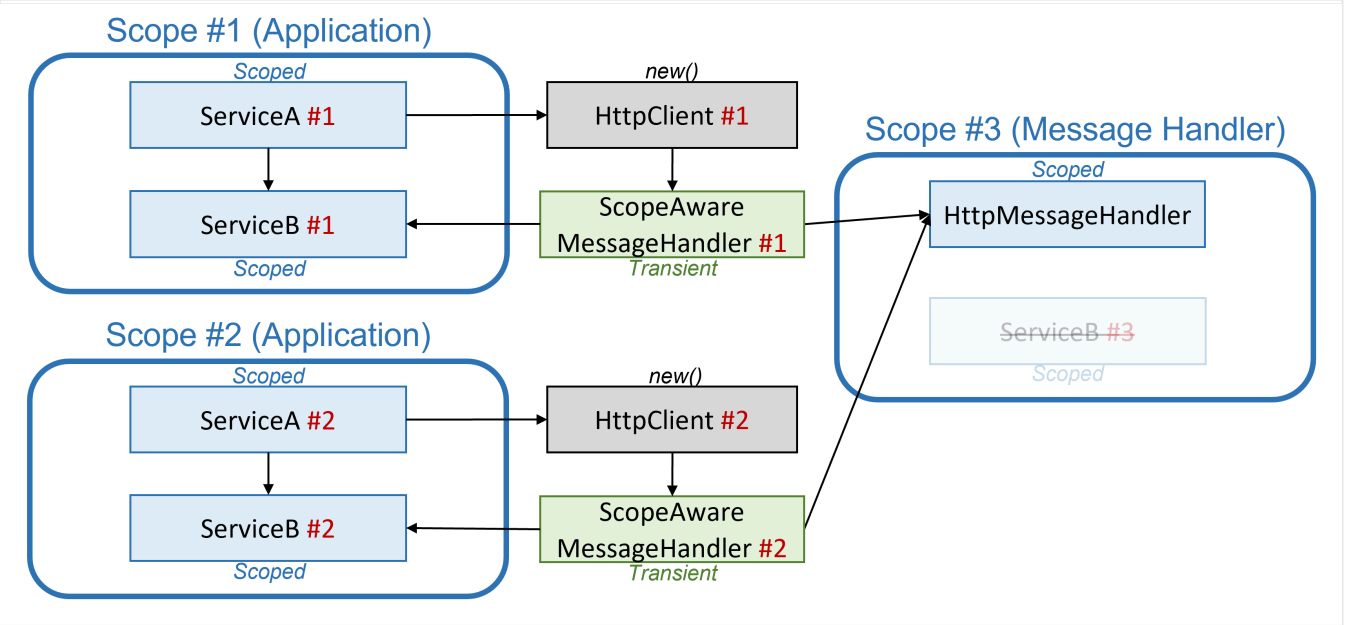
메시지 처리기 범위

`IHttpClientFactory`는 각 `HttpMessageHandler` 인스턴스당 별도의 DI 범위를 만듭니다. 이러한 DI 범위는 애플리케이션 DI 범위(예: ASP.NET 들어오는 요청 범위 또는 사용자가 만든 수동 DI 범위)에서 분리되므로 범위가 지정된 서비스 인스턴스를 공유하지 **않습니다**. 메시지 처리기 범위는 처리기 수명에 연결되며 애플리케이션 범위보다 오래 지속될 수 있습니다. 예를 들어 들어오는 여러 요청 간에 동일한 주입된 범위 종속성이 있는 동일한 `HttpMessageHandler` 인스턴스를 다시 사용할 수 있습니다.



사용자는 **인스턴스 내에서** 범위 관련 정보(예: `HttpContext`의 데이터)를 캐시하지 말고 중요한 정보 유출을 방지하지 않도록 `HttpMessageHandler`범위가 지정된 종속성을 주의해서 사용해야 합니다.

예를 들어 인증을 위해 메시지 처리기에서 앱 DI 범위에 액세스해야 하는 경우 별도의 임시 `DelegatingHandler`에 범위 인식 논리를 캡슐화하고 이를 `HttpMessageHandler` 캐시의 `IHttpClientFactory` 인스턴스에 래핑합니다. 등록된 IHttpMessageHandlerFactory.CreateHandler의 처리기 호출에 액세스하려면 을(를) 사용하세요. 이 경우 구성된 처리기를 사용하여 직접 `HttpClient` 인스턴스를 만듭니다.



다음 예에서는 범위 인식 `HttpClient`를 사용하여 `DelegatingHandler`를 만드는 방법을 보여 줍니다.

C#

```
if (scopeAwareHandlerType != null)
{
    if (!typeof(DelegatingHandler).IsAssignableFrom(scopeAwareHandlerType))
    {
        throw new ArgumentException($"""
            Scope aware HttpHandler {scopeAwareHandlerType.Name} should
            be assignable to DelegatingHandler
            """);
    }

    // Create top-most delegating handler with scoped dependencies
    scopeAwareHandler =
(DelegatingHandler)_scopeServiceProvider.GetRequiredService(scopeAwareHandlerType);
// should be transient
    if (scopeAwareHandler.InnerHandler != null)
    {
        throw new ArgumentException($"""
            Inner handler of a delegating handler {scopeAwareHandlerType.Name}
should be null.
            Scope aware HttpHandler should be registered as Transient.
            """);
    }
}

// Get or create HttpMessageHandler from HttpClientFactory
HttpMessageHandler handler = _httpMessageHandlerFactory.CreateHandler(name);

if (scopeAwareHandler != null)
{
    scopeAwareHandler.InnerHandler = handler;
    handler = scopeAwareHandler;
```

```
}

HttpClient client = new(handler);
```

추가적인 해결 방법으로 범위 인식 `DelegatingHandler`를 등록하고, 현재 앱 범위에 액세스할 수 있는 임시 서비스를 통해 기본 `IHttpClientFactory` 등록을 재정의하는 확장 메서드를 사용할 수 있습니다.

C#

```
public static IHttpClientBuilder AddScopeAwareHttpHandler<THandler>(
    this IHttpClientBuilder builder) where THandler : DelegatingHandler
{
    builder.Services.TryAddTransient<THandler>();
    if (!builder.Services.Any(sd => sd.ImplementationType ==
typeof(ScopeAwareHttpClientFactory)))
    {
        // Override default IHttpClientFactory registration
        builder.Services.AddTransient<IHttpClientFactory,
ScopeAwareHttpClientFactory>();
    }

    builder.Services.Configure<ScopeAwareHttpClientFactoryOptions>(
        builder.Name, options => options.HttpHandlerType = typeof(THandler));

    return builder;
}
```

자세한 내용은 전체 예제를 참조하세요.

"공장 초기 설정 기본 처리기에 의존하지 않도록 하십시오."

이 섹션에서 "*factory-default*" *기본 처리기*라는 용어는 기본 `IHttpClientFactory` 구현(또는 더 정확하게는 기본 `HttpMessageHandlerBuilder` 구현)이 *어떠한 방식으로도 전혀 구성되지 않았을 때 할당하는 기본 처리기*를 의미합니다.

> **참고 항목**
>
> "공장 기본값" 기본 처리기는 *구현 세부 사항*이며, 변경될 수 있습니다. ❌ "팩터리 기본값"(예: `HttpClientHandler`)으로 사용되는 특정 구현 방법에 의존하지 마세요.

특히 클래스 라이브러리에서 작업하는 경우 기본 처리기의 특정 형식을 알아야 하는 경우가 있습니다. 최종 사용자의 구성을 유지하면서, 필요에 따라 `HttpClientHandler`에 관련된

`ClientCertificates`, `UseCookies` 및 `UseProxy`과 같은 특정 속성을 업데이트할 수 있습니다. `HttpClientHandler` 을(를) "팩터리-기본값" 기본 처리기로 사용했을 때처럼, 주 처리기를 `HttpClientHandler`로 캐스팅하려는 유혹이 있을 수 있습니다. 그러나 구현 세부 사항에 의존하는 모든 코드와 마찬가지로, 이러한 해결 방법은 취약하여 깨질 가능성이 있습니다.

"팩터리 기본값" 기본 처리기를 사용하는 대신 `ConfigureHttpClientDefaults` 사용하여 "앱 수준" 기본 기본 처리기 인스턴스를 설정할 수 있습니다.

C#

```
// Contract with the end-user: Only HttpClientHandler is supported.

// --- "Pre-configure" stage ---
// The default is fixed as HttpClientHandler to avoid depending on the "factory-
default"
// Primary Handler.
services.ConfigureHttpClientDefaults(b =>
    b.ConfigurePrimaryHttpMessageHandler(() => new HttpClientHandler() { UseCookies
= false }));

// --- "End-user" stage ---
// IHttpClientBuilder builder = services.AddHttpClient("test", /* ... */);
// ...

// --- "Post-configure" stage ---
// The code can rely on the contract, and cast to HttpClientHandler only.
builder.ConfigurePrimaryHttpMessageHandler((handler, provider) =>
    {
        if (handler is not HttpClientHandler h)
        {
            throw new InvalidOperationException("Only HttpClientHandler is
supported");
        }

        h.ClientCertificates.Add(GetClientCert(provider, builder.Name));

        //X509Certificate2 GetClientCert(IServiceProvider p, string name) { ... }
    });
```

또는 기본 처리기 유형을 확인하고 잘 알려진 지원 형식(대부분 `HttpClientHandler` 및 `SocketsHttpHandler`)에서만 클라이언트 인증서와 같은 세부 정보를 구성할 수 있습니다.

C#

```
// --- "End-user" stage ---
// IHttpClientBuilder builder = services.AddHttpClient("test", /* ... */);
// ...

// --- "Post-configure" stage ---
// No contract is in place. Trying to configure main handler types supporting
```

```
client
// certs, logging and skipping otherwise.
builder.ConfigurePrimaryHttpMessageHandler((handler, provider) =>
    {
        if (handler is HttpClientHandler h)
        {
            h.ClientCertificates.Add(GetClientCert(provider, builder.Name));
        }
        else if (handler is SocketsHttpHandler s)
        {
            s.SslOptions ??= new
System.Net.Security.SslClientAuthenticationOptions();
            s.SslOptions.ClientCertificates ??= new X509CertificateCollection();
            s.SslOptions.ClientCertificates!.Add(GetClientCert(provider,
builder.Name));
        }
        else
        {
            // Log warning
        }

        //X509Certificate2 GetClientCert(IServiceProvider p, string name) { ... }
    });
```

참고하기

- 일반적인 IHttpClientFactory 사용 문제
- .NET의 종속성 주입
- .NET에서의 로깅
- .NET의 구성
- IHttpClientFactory
- IHttpMessageHandlerFactory
- HttpClient
- HttpClient를 사용하여 HTTP 요청 만들기
- 지수 백오프를 사용하여 HTTP 다시 시도 구현

Last updated on 2026. 03. 11.

.NET의 HTTP 클라이언트 대기 시간 원격 분석

2025. 10. 18.

HTTP를 통해 통신하는 애플리케이션을 빌드하는 경우 요청 성능 특성을 관찰하는 것이 중요합니다. 이 AddHttpClientLatencyTelemetry 확장을 사용하면 호출 코드를 변경하지 않고 나가는 HTTP 호출에 대한 자세한 타이밍 정보를 수집할 수 있습니다. 기존 `HttpClientFactory` 파이프라인에 연결하여 요청 수명 주기 전반에 걸쳐 스테이지 타이밍을 캡처하고, HTTP 프로토콜 세부 정보를 기록하고, 런타임이 해당 데이터를 노출하는 가비지 수집 영향을 측정하고, 성능 분석 및 튜닝에 적합한 균일한 원격 분석 셰이프를 내보낸다. 확장 메서드를 호출 `AddHttpClientLatencyTelemetry()` 하여 사용하도록 설정합니다. 기본 제공 처리기는 아웃바운드 요청마다 `ILatencyContext`를 생성하고 내부 파이프라인이 완료되는 시점까지 측정값을 집계합니다. `await base.SendAsync(...)` 나중에 위임 처리기(원격 분석 후 추가됨)에서 이를 사용하고 메트릭 백 엔드로 내보냅니다. 예제:

확장 메서드 등록:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Http.Diagnostics;
using Microsoft.Extensions.Hosting;

var builder = Host.CreateApplicationBuilder(args);

// An example of some accessor that is able to read latency context
builder.Services.AddSingleton<ILatencyContextAccessor, LatencyContextAccessor>();

// Register HTTP client latency telemetry first so its delegating handler runs
earlier.
builder.Services.AddHttpClientLatencyTelemetry();

// Register export handler (runs after telemetry; sees finalized ILatencyContext).
builder.Services.AddTransient<HttpLatencyExportHandler>();

// Register an HttpClient that will emit and export latency measures.
builder.Services
    .AddHttpClient("observed")
    .AddHttpMessageHandler<HttpLatencyExportHandler>();

var host = builder.Build();
await host.RunAsync();
```

컨텍스트에 액세스합니다.

C#

```
public sealed class HttpLatencyExportHandler : DelegatingHandler
{
    // ILatencyContextAccessor is just an example of some accessor that is able to
read latency context
    private readonly ILatencyContextAccessor _latency;

    public HttpLatencyExportHandler(ILatencyContextAccessor latency) => _latency =
latency;

    protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken ct)
    {
        var rsp = await base.SendAsync(request, ct).ConfigureAwait(false);

        var ctx = _latency.Current;
        if (ctx != null)
        {
            var data = ctx.LatencyData;
            // Record/export gc and conn with version as a dimension here.
        }
        return rsp;
    }
}
```

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Http.Diagnostics --version 9.10.0
```

자세한 내용은 .NET 애플리케이션에서 dotnet 패키지 추가 또는 패키지 종속성 관리를 참조하세요.

HTTP 클라이언트의 지연 시간 원격 분석 등록

애플리케이션에 HTTP 클라이언트 대기 시간 원격 분석을 추가하려면, 서비스를 구성할 때 AddHttpClientLatencyTelemetry 확장 메서드를 호출합니다.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add HTTP client factory
builder.Services.AddHttpClient();

// Add HTTP client latency telemetry
```

```
builder.Services.AddHttpClientLatencyTelemetry();
```

이 등록은 IHttpClientFactory을(를) 통해 생성된 모든 HTTP 클라이언트에 `DelegatingHandler`을 추가하여 각 요청에 대한 자세한 대기 시간 정보를 수집합니다.

원격 분석 옵션 구성

표준 옵션 패턴을 통해 텔레메트리 수집을 구성합니다. 대리자를 사용하거나 바인딩 구성(예 `appsettings.json`:)을 사용하여 값을 제공할 수 있습니다. 옵션 인스턴스는 처리기 파이프라인당 한 번 확인되므로 변경 내용이 새 클라이언트/처리기에 적용됩니다.

C#

```
// Configure with delegate
builder.Services.AddHttpClientLatencyTelemetry(options =>
{
    options.EnableDetailedLatencyBreakdown = true;
});

// Or configure from configuration
builder.Services.AddHttpClientLatencyTelemetry(
builder.Configuration.GetSection("HttpClientTelemetry"));
```

구성 옵션

클래스는 HttpClientLatencyTelemetryOptions 다음 설정을 제공합니다.

테이블 확장

Option	유형	Default	Description	사용하지 않도록 설정해야 하는 경우
상세한 지연 시간 분석 활성화	불리언 (Boolean)	`true`	각 HttpClient 요청(예: 연결 설정, 전송된 헤더, 첫 번째 바이트, 완료)에 대해 세분화된 단계 타이밍을 사용하도록 설정하여 총 대기 시간의 분석을 생성합니다. CPU/시간 측정 비용이 약간 추가되지만, 와이어 오버헤드는 없습니다.	`false` 최소 오버헤드가 필요하고 총 기간만으로 충분한 매우 높은 처리량 시나리오에서만 설정됩니다.

수집된 원격 분석 데이터

HTTP 클라이언트 대기 시간 원격 분석을 사용하도록 설정하면 성능 분석에 사용되는 단계 타임스탬프, 선택한 측정값(지원되는 경우) 및 프로토콜 특성을 캡처합니다.

타이밍 검사점

타임스탬프는 HTTP 요청 수명 주기의 주요 단계에 대해 기록됩니다.

테이블 확장

Phase	시작 이벤트	종료 이벤트	비고
DNS 해결	Http.NameResolutionStart	Http.NameResolutionEnd	호스트 이름 조회(캐시되어 건너뛸 수도 있음).
소켓 연결	Http.SocketConnectStart	Http.SocketConnectEnd	CP(처리기에 의해 결합된 경우 TLS 핸드셰이크).
연결 설정		Http.ConnectionEstablished (연결이 설정됨)	핸드셰이크 후 사용 가능한 연결을 표시합니다.
요청 헤더	Http.RequestHeadersStart	Http.RequestHeadersEnd	와이어/버퍼에 요청 헤더를 작성합니다.
콘텐츠 요청	Http.RequestContentStart	Http.RequestContentEnd	스트리밍 또는 버퍼링 요청 본문입니다.
응답 헤더	Http.ResponseHeadersStart	Http.ResponseHeadersEnd	헤더 구문 분석 완료에 대한 첫 번째 바이트입니다.
응답 콘텐츠	Http.ResponseContentStart	Http.ResponseContentEnd	전체 응답 본문을 읽습니다(완료 또는 폐기).

측정값(플랫폼 종속)

측정값은 원시 단계 체크포인트에서 사용할 수 없는 대기 시간 요인을 정량화합니다(GC 일시 중지 겹침, 연결 분산, 기타 축적된 수치 또는 지속 시간). 호출 `AddHttpClientLatencyTelemetry()`할 때 생성된 메모리 내 대기 시간 컨텍스트에서 수집됩니다. 아무것도 자동으로 내보내지 않습니다. 컨텍스트는 요청이 완료될 때까지 검사점, 측정값 및 태그를 누적합니다. 또한 HTTP 클라이언트 로깅 보강 `AddExtendedHttpClientLogging()`을 사용하도록 설정하면 완성된 컨텍스트가 명명 `LatencyInfo` 된 단일 구조화된 로그 필드(버전 표식, 사용 가능한 경우 서버 이름, 태그, 검사점 및 측정값 이름/값 시퀀스)로 평면화됩니다.

이 로그 필드는 유일한 기본 제공 출력 아티팩트입니다. 고유한 내보내기를 추가하지 않는 한 메트릭이나 추적이 생성되지 않습니다. 메트릭으로 표시하기 위해 요청 파이프라인이 반환된 후 컨텍스트를 읽고, GC 일시 중지 겹침을 히스토그램으로 기록하고 연결 시작을 카운터로 기록합니다. 필요에 따라 프로토콜 버전별로 차원화할 수 있습니다.

테이블 확장

이름	Description
Http.GCPauseTime	요청과 겹치는 총 GC 일시 중지 기간입니다.
Http.ConnectionInitiated	풀링된 재사용이 아닌 새 기본 연결이 만들어질 때 내보내집니다.

태그들

태그를 사용하여 원시 데이터를 다시 처리하지 않고 메트릭과 로그를 분할, 필터링 및 집계할 수 있도록 각 요청에 안정적인 범주 차원을 연결합니다. 대기 시간 컨텍스트에서 캡처된 낮은 카디널리티 분류 레이블(타이밍 아님)이며, HTTP 클라이언트 로그 보강을 사용하는 경우 단일 LatencyInfo 로그 필드로 직렬화됩니다. 애플리케이션 시작 시점에 보강 기능을 활성화하려면, 예를 들어 모든 클라이언트 또는 클라이언트별로 대기 시간 원격 분석과 함께 로깅 확장을 추가하십시오.

C#

```
var builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHttpClientLatencyTelemetry(); // enables latency context + measures/tags
builder.Services.AddExtendedHttpClientLogging();
var app = builder.Build();
```

그 후 구조적 로깅 파이프라인을 통해 기록된 아웃바운드 요청에는 평면화된 태그, 검사점 및 측정값이 포함된 `LatencyInfo` 속성이 포함됩니다. 태그에 대한 메트릭 또는 추적은 자동으로 내보내지지 않습니다. 로그 외부에 필요한 경우 직접 내보내기(예: 태그 값을 메트릭 차원 또는 `Activity` 태그로 변환).

테이블 확장

Tag	Description
HTTP 버전	협상/사용된 HTTP 프로토콜 버전(예: 1.1, 2, 3).

사용 예제

이러한 구성 요소를 사용하면 HTTP 클라이언트 요청 처리의 대기 시간을 추적하고 보고할 수 있습니다.

다음 방법을 사용하여 서비스를 등록할 수 있습니다.

C#

```
public static IServiceCollection AddHttpClientLatencyTelemetry(
    this IServiceCollection services);

public static IServiceCollection AddHttpClientLatencyTelemetry(
    this IServiceCollection services,
    IConfigurationSection section);

public static IServiceCollection AddHttpClientLatencyTelemetry(
    this IServiceCollection services,
    Action<HttpClientLatencyTelemetryOptions> configure);
```

다음은 그 예입니다.

C#

```
var builder = Host.CreateApplicationBuilder(args);

// Register IHttpClientFactory:
builder.Services.AddHttpClient();

// Register redaction services:
builder.Services.AddRedaction();

// Register latency context services:
builder.Services.AddLatencyContext();

// Register HttpClient logging enrichment & redaction services:
builder.Services.AddExtendedHttpClientLogging();

// Register HttpClient latency telemetry services:
builder.Services.AddHttpClientLatencyTelemetry();

var host = builder.Build();
```

플랫폼 고려 사항

HTTP 클라이언트 대기 시간 원격 분석은 지원되는 모든 대상(.NET 9, .NET 8, .NET Standard 2.0 및 .NET Framework 4.6.2)에서 실행됩니다. 핵심 시간 체크포인트는 항상 수집됩니다. GC 일시 중지 메트릭(Http.GCPauseTime)은 .NET 8 또는 .NET 9에서 실행되는 경우에만 내보내집니다. 구현은 런타임에 플랫폼 기능을 검색하고 추가 구성 없이 지원되는 기능을 사용하도록 설정합니다.

일반적인 `IHttpClientFactory` 사용 문제

`IHttpClientFactory`을(를) `HttpClient` 인스턴스를 만들기 위해 사용하는 경우에 발생할 수 있는 가장 일반적인 문제 중 일부를 이 문서에서 알아봅니다.

DI 컨테이너에서 여러 `IHttpClientFactory` 구성을 설정하고, 로깅을 구성하며, 복원력 전략을 설정하는 편리한 방법으로 `HttpClient`이(가) 있습니다. `IHttpClientFactory`는 또한 소켓 소모 및 DNS 변경 손실과 같은 문제를 방지하기 위해 `HttpClient` 및 `HttpMessageHandler` 인스턴스의 수명 관리를 캡슐화합니다. .NET 애플리케이션에서 `IHttpClientFactory`를 사용하는 방법에 대한 개요는 IHttpClientFactory with .NET을 참조하세요.

DI와 `IHttpClientFactory` 통합의 복잡한 특성 때문에, 잡아내기 어렵고 문제 해결이 힘든 몇 가지 문제를 겪을 수 있습니다. 잠재적인 문제를 방지하고자 사전에 적용할 수 있는 권장 사항도 이 문서에 나열된 시나리오에 포함되어 있습니다.

`HttpClient`은(는) `Scoped` 수명을 존중하지 않습니다.

예를 들어 `HttpContext` 또는 일부 범위 지정된 캐시에 접근해야 할 경우, `HttpMessageHandler` 내부에서 범위 지정된 서비스에 접근하려고 할 때 문제가 발생할 수 있습니다. 저장된 데이터는 "사라지거나" 그렇지 않은 경우 "지속"될 수 있습니다. 이는 애플리케이션 컨텍스트와 처리기 인스턴스 간의 DI(종속성 주입) **범위 불일치**로 인해 발생되며, `IHttpClientFactory`에 대한 제한 사항으로 알려져 있습니다.

`IHttpClientFactory`는 각 `HttpMessageHandler` 인스턴스당 별도의 DI 범위를 만듭니다. 이러한 처리기 범위는 범위가 지정된 서비스 인스턴스를 공유하지 않으며, 이는 애플리케이션 컨텍스트 범위(예: ASP.NET Core의 수신 요청 범위 또는 사용자에 의해 생성된 수동 DI 범위)에서 분리되기 때문입니다.

다음은 이러한 제한으로 인한 결과입니다.

- 범위가 지정된 서비스에서 "외부적으로" 캐시된 데이터는 사용 가능하지 **않습니다** `HttpMessageHandler`.
- "내부적으로" `HttpMessageHandler` 내부의 캐시된 모든 데이터는 여러 애플리케이션 DI 범위(예: 다른 들어오는 요청)에서 관찰할 수 **있으며**, 이는 동일한 처리기를 공유할 수 있기 때문입니다.

다음과 같은 권장 사항을 고려하여 알려진 제한 사항을 완화하세요.

❌ 중요한 정보가 유출되지 않도록 범위 관련 정보(`HttpContext`의 데이터 등)를 `HttpMessageHandler` 인스턴스 또는 해당 종속성 내부에 캐시하지 마세요.

❌ 쿠키를 사용하지 마십시오. `CookieContainer`이(가) 처리기와 함께 공유됩니다.

✔ 정보를 저장하지 않거나 `HttpRequestMessage` 인스턴스 내에서 해당 사항만 전달하는 것이 권장됩니다.

`HttpRequestMessage` 속성을 사용하여 HttpRequestMessage.Options와(과) 함께 임의 정보를 전달할 수 있습니다.

✔ 모든 범위 관련(인증 등) 논리를 `DelegatingHandler`에 의해 생성되지 **않은** 별도의 `IHttpClientFactory` 내에 캡슐화하는 것을 고려해야 하며, 이를 사용하여 `IHttpClientFactory`에 의해 생성된 처리기를 래핑하는 것이 권장됩니다.

오직 `HttpMessageHandler`을(를) 생성하고 `HttpClient` 없이, 등록된 모든 *명명된 클라이언트*에 대해 IHttpMessageHandlerFactory.CreateHandler을(를) 호출하세요. 그러한 경우 직접 `HttpClient` 인스턴스를 만들기 위해 결합된 처리기를 사용할 필요가 있습니다. 이 해결 방법을 위한 완전히 실행 가능한 예제를 GitHub에서 찾을 수 있습니다.

자세한 내용은 IHttpClientFactory의 메시지 핸들러 범위 섹션을 `IHttpClientFactory` 지침에서 참조하세요.

`HttpClient`은(는) DNS 변경 내용을 존중하지 않습니다.

`IHttpClientFactory`이(가) 사용되는 경우에도 부실 DNS 문제를 해결 가능합니다. 이는 일반적으로 `HttpClient` 인스턴스가 `Singleton` 서비스에서 캡처되거나, 일반적으로 지정된 `HandlerLifetime`보다 긴 기간 동안 어딘가에 저장될 때 발생 가능합니다. 싱글톤에 의해 *typed client*가 캡처되는 경우, `HttpClient`도 캡처됩니다.

✘ `HttpClient`에 의해 장기간 생성된 `IHttpClientFactory` 인스턴스는 캐시하지 마세요.

✘ *형식화된 클라이언트* 인스턴스를 `Singleton` 서비스에 삽입하지 마세요.

✔ 적시에 또는 필요할 때마다 `IHttpClientFactory`(으)로부터 클라이언트를 요청하는 것을 고려하세요. 팩터리에 의해 생성된 클라이언트는 제거해도 안전합니다.

`HttpClient`에서 만든 `IHttpClientFactory` 인스턴스는 **단기**여야 합니다.

- 처리기가 DNS 변경에 반응하도록 보장하려면, 수명이 만료되면 `HttpMessageHandler`을 재활용하고 `IHttpClientFactory`을 다시 생성해야 합니다. `HttpClient`는 만들어질 때 특정 처리기 인스턴스에 연결되므로 클라이언트가 업데이트된 처리기를 가져올 수 있도록 새 `HttpClient` 인스턴스를 적시에 요청해야 합니다.

- `HttpClient` 이러한 **공장에서 생성된 인스턴스를** 삭제해도 소켓 소진으로 이어지지 않습니다. 이는 해당 인스턴스를 삭제한다고 해서 의 삭제가 트리거되지 않기 때문입니다

`HttpMessageHandler`. `IHttpClientFactory`는 `HttpClient` 인스턴스, 특히 `HttpMessageHandler` 인스턴스를 만드는 데 사용되는 리소스를 추적하고 삭제합니다. 이러한 인스턴스는 수명이 곧 만료되고 이러한 인스턴스를 사용하는 `HttpClient`가 더 이상 없기 때문입니다.

*형식화된 클라이언트*는 **단명**할 것으로 의도되며, 이는 `HttpClient` 인스턴스가 생성자에 삽입되어 *형식화된 클라이언트*의 수명을 공유하기 때문입니다.

자세한 내용은 `IHttpClientFactory` 지침의 수명 관리 및 싱글톤 서비스에서 *형식화된 클라이언트 사용 피하기* 섹션을 참조하세요.

`HttpClient`이(가) 소켓을 너무 많이 사용합니다.

`IHttpClientFactory`이(가) 사용되는 경우에도 특정 사용 시나리오에서 소켓 소진 문제에 대해 해결 가능합니다. 기본적으로 `HttpClient`은(는) 동시 요청 수를 제한하지 않습니다. 동시에 많은 수의 HTTP/1.1 요청이 시작될 경우, 연결 풀에 사용 가능한 연결이 없으며 제한이 설정되어 있지 않기 때문에 새로운 HTTP 연결 시도를 각 요청이 트리거합니다.

❌ 동시에 많은 수의 HTTP/1.1 요청을 제한을 지정하지 않은 상태로 시작하지 마세요.

✔ HttpClientHandler.MaxConnectionsPerServer을(를) 적절한 값으로 설정하는 것을 고려하세요(또는 기본 처리기로 사용하는 경우에는 SocketsHttpHandler.MaxConnectionsPerServer). 특정 처리기 인스턴스에만 이러한 제한이 적용됩니다.

✔ 단일 TCP 연결을 통해 멀티플렉싱 요청을 허용하는 HTTP/2를 사용하는 것이 권장됩니다.

*Typed client*에 잘못된 `HttpClient`가 주입되었습니다.

`HttpClient`가 *타입화된 클라이언트*에 예기치 않게 삽입될 수 있는 다양한 상황이 존재할 수 있습니다. 근본 원인은 대부분의 경우 잘못된 구성에 있으며, 이는 DI 설계에 따라 서비스의 후속 등록이 이전 등록을 재정의하기 때문입니다.

*형식화된 클라이언트*는 "내부적으로" *명명된 클라이언트*를 사용하며, 이는 *형식화된 클라이언트*를 추가함으로써 암시적으로 등록되며 *명명된 클라이언트*에 연결됩니다. 명시적으로 제공되지 않는 한, `TClient`의 형식 이름으로 클라이언트 이름이 설정됩니다. 오버로드가 사용되는 경우, `TClient,TImplementation` 쌍에서 **첫 번째**가 됩니다.

그러므로, *형식화된 클라이언트*를 등록함으로써 다음의 두 가지 개별 작업이 수행됩니다.

1. *명명된 클라이언트*를 등록합니다(간단한 기본 경우 이름은 `typeof(TClient).Name`입니다).

2. 제공된 `Transient` 또는 `TClient`을(를) 활용하여 `TClient,TImplementation` 서비스를 등록합니다.

다음과 같은 두 가지 문장은 기술적으로 동일합니다.

C#

```
services.AddHttpClient<ExampleClient>(c => c.BaseAddress = new
Uri("http://example.com"));

// -OR-

services.AddHttpClient(nameof(ExampleClient), c => c.BaseAddress = new
Uri("http://example.com")) // register named client
    .AddTypedClient<ExampleClient>(); // link the named client to a typed client
```

간단한 경우, 다음과 유사합니다.

C#

```
services.AddHttpClient(nameof(ExampleClient), c => c.BaseAddress = new
Uri("http://example.com")); // register named client

// register plain Transient service and link it to the named client
services.AddTransient<ExampleClient>(s =>
    new ExampleClient(
        s.GetRequiredService<IHttpClientFactory>
().CreateClient(nameof(ExampleClient))));
```

형식화된 클라이언트와 *명명된* 클라이언트 간의 연결이 어떻게 끊어질 수 있는지에 관한 다음과 같은 예시를 고려하세요.

*형식화된 클라이언트*가 두 번째로 등록됨

❌ `AddHttpClient<T>` 호출에 의해 *typed client*가 이미 자동으로 등록되므로, 따로 등록할 필요가 없습니다.

일반적인 임시 서비스로서 *형식화된 클라이언트*가 두 번째로 잘못 등록되는 경우, 해당 `IHttpClientFactory`에서 추가한 등록을 덮어쓰며 *명명된 클라이언트*에 대한 링크를 끊습니다. `HttpClient`의 구성이 없는 것처럼 나타나게 됩니다. 이는 구성이 없는 `HttpClient`이(가) 대신 *타입 클라이언트*에 주입되기 때문입니다.

"잘못된" `HttpClient`이(가) 예외를 발생시키는 대신 사용되는 것이 혼란스러울 수 있습니다. 이는 "기본"으로 구성되지 않은 `HttpClient`, 즉 Options.DefaultName 이름(`string.Empty`)을 가진 클라이언트가 가장 기본적인 `IHttpClientFactory` 사용 시나리오를 가능하게 하기 위해 일시적

인 서비스로 등록되기 때문입니다. 그러므로 링크가 끊어지며 *형식화된 클라이언트*가 일반 서비스가 되는 경우, 자연스럽게 해당 생성자 매개 변수에 이 "기본값" `HttpClient`이(가) 삽입됩니다.

공통 인터페이스에 다른 *형식화된 클라이언트*가 등록됩니다.

공통 인터페이스에 두 개의 서로 다른 *형식화된 클라이언트*가 등록된 경우에는 동일한 *명명된 클라이언트*를 다시 사용합니다. 이는 첫 번째 *형식화된 클라이언트*가 두 번째 *명명된 클라이언트*를 "잘못" 삽입하는 것처럼 보일 수 있습니다.

❌ 여러 *형식화된 클라이언트*를 단일 인터페이스에 등록하지 않고, 이름을 명시적으로 지정하지 않아야 합니다.

✔ 별도로 *명명된 클라이언트*를 등록하고 구성한 뒤, 호출에서 이름을 지정하거나 `AddHttpClient<T>` 설정 중에 `AddTypedClient`을(를) 호출하여 하나 이상의 *형식화된 클라이언트*에 연결하는 것이 권장됩니다.

기본적으로 동일한 이름을 가진 *명명된 클라이언트*를 여러 번 등록하고 구성하면 기존 클라이언트 목록에 구성 작업이 추가됩니다. `IHttpClientFactory`의 이러한 동작은 명확하지 않을 수 있지만, 옵션 패턴 및 Configure와(과) 같은 구성 API에서 사용하는 것과 동일한 방식입니다.

이는 고급 처리기 구성에 주로 유용합니다. 그 예시로, 사용자 지정 처리기를 외부에 정의된 *명명된 클라이언트*에 추가하거나 테스트에 대한 기본 처리기를 모의하는 경우가 있지만 `HttpClient` 인스턴스 구성에서도 작동합니다. 예를 들어, 다음 세 가지 예는 동일한 방식으로 `HttpClient`가 구성된 결과를 보여줍니다(그리고 `BaseAddress` 및 `DefaultRequestHeaders` 둘 다 설정됩니다).

C#

```
// one configuration callback
services.AddHttpClient("example", c =>
    {
        c.BaseAddress = new Uri("http://example.com");
        c.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClient/8.0");
    });

// -OR-

// two configuration callbacks
services.AddHttpClient("example", c => c.BaseAddress = new
Uri("http://example.com"))
    .ConfigureHttpClient(c =>
c.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClient/8.0"));

// -OR-

// two configuration callbacks in separate AddHttpClient calls
services.AddHttpClient("example", c => c.BaseAddress = new
Uri("http://example.com"));
```

```
services.AddHttpClient("example")
    .ConfigureHttpClient(c =>
c.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClient/8.0"));
```

이렇게 하면 이미 정의된 *명명된 클라이언트*에 *형식화된 클라이언트*를 연결하고 단일 *명명된 클라이언트*에 여러 *형식화된 클라이언트*를 연결 가능합니다. 매개 변수 `name`와 함께 사용된 오버로드가 있을 때 더 분명해집니다.

C#

```
services.AddHttpClient("LogClient", c => c.BaseAddress = new
Uri(LogServerAddress));

services.AddHttpClient<FooLogger>("LogClient");
services.AddHttpClient<BarLogger>("LogClient");
```

동일한 작업을 수행하기 위해 *명명된 클라이언트* 구성 중에 AddTypedClient를 호출할 수도 있습니다.

C#

```
services.AddHttpClient("LogClient", c => c.BaseAddress = new Uri(LogServerAddress))
    .AddTypedClient<FooLogger>()
    .AddTypedClient<BarLogger>();
```

그러나 *동일한 명명된 클라이언트*를 다시 *사용하지 않고 동일한* 인터페이스에 클라이언트를 등록하려는 경우에도 다른 이름을 명시적으로 지정하여 이 작업을 수행할 수 있습니다.

C#

```
services.AddHttpClient<ITypedClient, ExampleClient>(nameof(ExampleClient),
    c => c.BaseAddress = new Uri("http://example.com"));
services.AddHttpClient<ITypedClient, GithubClient>(nameof(GithubClient),
    c => c.BaseAddress = new Uri("https://github.com"));
```

참고 항목

- .NET을 사용한 IHttpClientFactory
- IHttpClientFactory
- IHttpMessageHandlerFactory
- HttpClient
- HttpClient를 사용하여 HTTP 요청 만들기

Last updated on 2026. 02. 24.

`IHttpClientFactory` 키 방식 DI 지원

아티클 • 2025. 01. 31.

이 문서에서는 keyed Services와 `IHttpClientFactory` 통합하는 방법에 대해 알아봅니다.

Keyed Services(*Keyed DI*라고도 함)는 단일 서비스의 여러 구현으로 편리하게 작동할 수 있는 DI(종속성 주입) 기능입니다. 등록 시 다른 *서비스 키* 특정 구현과 연결할 수 있습니다. 런타임에 이 키는 서비스 유형과 함께 조회에 사용되므로 일치하는 키를 전달하여 특정 구현을 검색할 수 있습니다. Keyed Services 및 DI에 대한 자세한 내용은 .NET 종속성 주입참조하세요.

.NET 애플리케이션에서 `IHttpClientFactory` 사용하는 방법에 대한 개요는 .NET IHttpClientFactory를 참조하세요.

배경

`IHttpClientFactory` 및 명명된 `HttpClient` 인스턴스는 당연히 Keyed Services 아이디어와 잘 일치합니다. 역사적으로, 무엇보다도, `IHttpClientFactory` 오랫동안 누락 된 DI 기능을 극복 할 수있는 방법이었습니다. 그러나 일반적인 명명된 클라이언트는 구성된 `HttpClient`을 주입하는 대신 `IHttpClientFactory` 인스턴스를 가져오고 저장하고 쿼리해야 합니다. 이 과정은 불편할 수 있습니다. 형식화된 클라이언트는 해당 부분을 단순화하려고 시도하지만 catch가 함께 제공됩니다. 형식화된 클라이언트는 잘못 구성하고 오용하기 쉽고, 지원 인프라는 특정 시나리오(예: 모바일 플랫폼)에서 실질적인 오버헤드가 될 수도 있습니다.

.NET 9(`Microsoft.Extensions.Http` 및 `Microsoft.Extensions.DependencyInjection` 패키지 버전 `9.0.0+`)부터 `IHttpClientFactory` Keyed DI를 직접 활용하여 새로운 "Keyed DI 접근 방식"("명명됨" 및 "형식화된" 접근 방식과 반대)을 도입할 수 있습니다. "Keyed DI 접근 방식은 편리하고 매우 유연하게 구성할 수 있는 `HttpClient` 등록을 특정하게 구성된 `HttpClient` 인스턴스의 직관적인 주입과 결합합니다."

기본 사용량

.NET 9에서는 AddAsKeyed 확장 메서드를 호출하여 기능에 옵트인해야 합니다. 옵트인하면 구성을 적용하는 명명된 클라이언트가 서비스 키로 클라이언트의 이름을 사용하여 DI 컨테이너에 키 `HttpClient` 서비스로 추가되므로 표준 키 서비스 API(예: FromKeyedServicesAttribute)를 사용하여 원하는 명명된 `HttpClient` 인스턴스(`IHttpClientFactory`생성 및 구성)를 가져올 수 있습니다. 기본적으로 클라이언트는 *영역* 수명으로 등록됩니다.

다음 코드는 `IHttpClientFactory`, Keyed DI 및 ASP.NET Core 9.0 최소 API 간의 통합을 보여 줍니다.

C#

```
var builder = WebApplication.CreateBuilder(args);

// --- (1) Registration ---
builder.Services.AddHttpClient("github", c =>
    {
        c.BaseAddress = new Uri("https://api.github.com/");
        c.DefaultRequestHeaders.Add("Accept",
"application/vnd.github.v3+json");
        c.DefaultRequestHeaders.Add("User-Agent", "dotnet");
    })
    .AddAsKeyed(); // Add HttpClient as a Keyed Scoped service for
key="github"

var app = builder.Build();

// --- (2) Obtaining HttpClient instance ---
// Directly inject the Keyed HttpClient by its name
app.MapGet("/", ([FromKeyedServices("github")] HttpClient httpClient) =>
    // --- (3) Using HttpClient instance ---
    httpClient.GetFromJsonAsync<Repo>("/repos/dotnet/runtime"));

app.Run();

record Repo(string Name, string Url);
```

엔드포인트 응답:

sh

```
> ~  curl http://localhost:5000/
{"name":"runtime","url":"https://api.github.com/repos/dotnet/runtime"}
```

이 예제에서 구성된 `HttpClient` ASP.NET Core 매개 변수 바인딩에 통합된 표준 Keyed DI 인프라를 통해 요청 처리기에 삽입됩니다. ASP.NET Core에서 Keyed Services에 대한 자세한 내용은 ASP.NET Core의 종속성 주입을 참조하세요.

키드(Keyed), 네임드(Named), 타이핑(Typed) 접근 방식 비교

Basic Usage 예제의 `IHttpClientFactory`관련 코드만 고려합니다.

C#

```
services.AddHttpClient("github", /* ... */).AddAsKeyed();                //
(1)

app.MapGet("/", ([FromKeyedServices("github")] HttpClient httpClient) => //
(2)
    //httpClient.Get....                                                  //
(3)
```

이 코드 조각은 *Keyed DI 접근 방식을 사용할 때 등록* `(1)`, *구성된* `HttpClient` *인스턴스* `(2)` *가져오고, 필요에 따라 얻은 클라이언트 인스턴스를 사용하는* `(3)` 방법을 보여 줍니다.

두 가지 "이전" 접근 방식과 동일한 단계를 수행하는 방법을 비교합니다.

먼저 *명명된 접근 방식을*.

C#

```
services.AddHttpClient("github", /* ... */);                          // (1)

app.MapGet("/github", (IHttpClientFactory httpClientFactory) =>
{
    HttpClient httpClient = httpClientFactory.CreateClient("github"); // (2)
    //return httpClient.Get....                                       // (3)
});
```

둘째, *유형 접근 방식*:

C#

```
services.AddHttpClient<GitHubClient>(/* ... */);          // (1)

app.MapGet("/github", (GitHubClient gitHubClient) =>
    gitHubClient.GetRepoAsync());

public class GitHubClient(HttpClient httpClient)          // (2)
{
    private readonly HttpClient _httpClient = httpClient;

    public Task<Repo> GetRepoAsync() =>
        //_httpClient.Get....                               // (3)
}
```

세 가지 중에서 Keyed DI 접근 방식은 동일한 동작을 달성하는 가장 간결한 방법을 제공합니다.

기본 제공 DI 컨테이너 유효성 검사

특정 명명된 클라이언트에 대해 키 등록을 사용하도록 설정한 경우 기존 Keyed DI API를 사용하여 액세스할 수 있습니다. 그러나 아직 사용하도록 설정되지 않은 이름을 잘못 사용하려고 하면 표준 Keyed DI 예외가 발생합니다.

C#

```
services.AddHttpClient("keyed").AddAsKeyed();
services.AddHttpClient("not-keyed");

provider.GetRequiredKeyedService<HttpClient>("keyed"); // OK

// Throws: No service for type 'System.Net.Http.HttpClient' has been
registered.
provider.GetRequiredKeyedService<HttpClient>("not-keyed");
```

또한 클라이언트의 범위가 지정된 수명은 의존 관계의 문제를 발견하는 데 도움이 될 수 있습니다.

C#

```
services.AddHttpClient("scoped").AddAsKeyed();
services.AddSingleton<CapturingSingleton>();

// Throws: Cannot resolve scoped service 'System.Net.Http.HttpClient' from
root provider.
rootProvider.GetRequiredKeyedService<HttpClient>("scoped");

using var scope = provider.CreateScope();
scope.ServiceProvider.GetRequiredKeyedService<HttpClient>("scoped"); // OK

// Throws: Cannot consume scoped service 'System.Net.Http.HttpClient' from
singleton 'CapturingSingleton'.
public class CapturingSingleton([FromKeyedServices("scoped")] HttpClient
httpClient)
//{ ...
```

서비스 수명 선택

기본적으로 `AddAsKeyed()`는 `HttpClient`을 키 *범위가 지정된* 서비스로 등록합니다. `ServiceLifetime` 매개 변수를 `AddAsKeyed()` 메서드에 전달하여 수명을 명시적으로 지정할 수도 있습니다.

C#

```
services.AddHttpClient("explicit-scoped")
    .AddAsKeyed(ServiceLifetime.Scoped);

services.AddHttpClient("singleton")
    .AddAsKeyed(ServiceLifetime.Singleton);
```

형식화된 클라이언트 등록 내에서 `AddAsKeyed()` 호출하는 경우 기본 명명된 클라이언트만 Keyed로 등록됩니다. 형식화된 클라이언트 자체는 계속해서 일반 임시 서비스로 등록됩니다.

일시적인 HttpClient 메모리 누수 방지

> **ⓘ 중요**
>
> `HttpClient`는 `IDisposable`이므로, Keyed `HttpClient` 인스턴스에 대해서 *일시적인 수명을 피하는 것이 가장* 추천해 드립니다.
>
> 클라이언트를 Keyed Transient 서비스로 등록하면 둘 다 `IDisposable`구현하므로 `HttpClient` 및 `HttpMessageHandler` 인스턴스가 DI 컨테이너 캡처됩니다. 이로 인해 클라이언트가 Singleton 서비스 내에서 여러 번 해결되는 경우 메모리 누수가 발생할 수 있습니다.

포로 종속성 방지

> **ⓘ 중요**
>
> 다음 중 하나가 적용되면 `HttpClient`이 등록됩니다.
>
> - Keyed *싱글톤*으로서 - 또는 -
> - Keyed *로 범위 지정된* 또는 *임시*, 그리고 *장기 실행*(`HandlerLifetime`이상) 애플리케이션 범위 내에 삽입, -OR-
> - Keyed *임시*로서 *Singleton* 서비스에 삽입합니다.
>
> - `HttpClient` 인스턴스는 *포로*되며 예상 수명 `HandlerLifetime`넘길 가능성이 있습니다. `IHttpClientFactory`은 제한된 클라이언트를 제어할 수 없으므로 처리기 회전에 참여할 수 없으며, 그로 인해 DNS 변경이 **손실될 수**있습니다. 임시 서비스로 등록된 Typed 클라이언트에 대해 유사한 문제가 이미 있습니다.

클라이언트의 장수를 피할 수 없거나 의식적으로 원하는 경우(예: Keyed Singleton)의 경우 `PooledConnectionLifetime` 적절한 값으로 설정하여 `SocketsHttpHandler` 활용하는 것이 좋습니다.

C#

```
services.AddHttpClient("shared")
    .AddAsKeyed(ServiceLifetime.Singleton) // explicit singleton
    .UseSocketsHttpHandler((h, _) => h.PooledConnectionLifetime =
TimeSpan.FromMinutes(2))
    .SetHandlerLifetime(Timeout.InfiniteTimeSpan); // disable rotation
services.AddSingleton<MySingleton>();

public class MySingleton([FromKeyedServices("shared")] HttpClient shared) //
{ ...
```

범위 불일치 주의

스코프 지정 수명은 Named `HttpClient`의 경우(싱글톤 및 일시적인 문제에 비해) 훨씬 덜 문제가 되지만, 자체적인 문제가 있습니다.

> **ⓘ 중요**
>
> 특정 `HttpClient` 인스턴스의 키 범위 수명은 예상대로 확인된 "일반" 애플리케이션 범위(예: 들어오는 요청 범위)에 바인딩됩니다. 그러나 공장에서 직접 생성한 명명된 클라이언트와 동일한 방식으로 `IHttpClientFactory`에서 관리하는 기본 메시지 핸들러 체인에는 적용되지 않습니다. `HttpClient`가 *동일한* 이름을 사용하지만, 두 개의 서로 다른 범위(예: 동일한 엔드포인트에 대한 두 개의 동시 요청)에서 `HandlerLifetime` 시간 범위 내에 확인되는 경우에는, 동일한 *`HttpMessageHandler` 인스턴스를*재사용할 수 있습니다. 이 인스턴스에는 메시지 처리기 범위에 설명된 대로 고유한 별도의 범위가 있습니다.

> **ⓘ 참고**
>
> **범위 불일치** 문제는 불쾌하고 오래 지속되는 문제이며.NET 9에서는 아직 해결되지 않은 남아 있습니다. 일반 DI 인프라를 통해 주입된 서비스에서는 모든 종속성이 동일한 범위에서 충족될 것으로 예상하지만 키 범위가 지정된 `HttpClient` 인스턴스의 경우 그렇지 않습니다.

키 지정 메시지 처리기 체인

일부 고급 시나리오의 경우 `HttpClient` 개체 대신 `HttpMessageHandler` 체인에 직접 액세스할 수 있습니다. `IHttpClientFactory` 처리기를 만드는 `IHttpMessageHandlerFactory` 인터페이스를 제공합니다. Keyed DI를 사용하도록 설정하면 `HttpClient`뿐만 아니라 해당 `HttpMessageHandler` 체인도 Keyed 서비스로 등록됩니다.

C#

```
services.AddHttpClient("keyed-handler").AddAsKeyed();

var handler = provider.GetRequiredKeyedService<HttpMessageHandler>("keyed-handler");
var invoker = new HttpMessageInvoker(handler, disposeHandler: false);
```

방법: 형식화된 접근 방식에서 Keyed DI로 전환

> ⓘ **참고**
>
> 현재 형식화된 클라이언트 대신 Keyed DI 접근 방식을 사용하는 것이 좋습니다.

기존 형식화된 클라이언트에서 키 종속성으로의 최소 변경 스위치는 다음과 같습니다.

diff

```
- services.AddHttpClient<Service>(          // (1) Typed client
+ services.AddHttpClient(nameof(Service),   // (1) Named client
      c => { /* ... */ }                    // HttpClient configuration
  //).Configure....
- );
+ ).AddAsKeyed();                           // (1) + Keyed DI opt-in

+ services.AddTransient<Service>();         // (1) Plain Transient service

  public class Service(
-                                           // (2) "Hidden" Named dependency
+     [FromKeyedServices(nameof(Service))] // (2) Explicit Keyed dependency
      HttpClient httpClient) // { ...
```

예제에서는 다음을 수행합니다.

1. Typed 클라이언트 `Service`의 등록은 다음과 같이 분할됩니다.
    - 동일한 `HttpClient` 구성을 사용하여 명명된 클라이언트 `nameof(Service)` 등록하고 Keyed DI에 옵트인합니다. 그리고
    - 일반 임시 서비스 `Service`.

2. `Service`에서 `HttpClient` 종속성은 키 `nameof(Service)`가 있는 서비스에 명확하게 바인딩됩니다.

이름은 `nameof(Service)` 필요가 없지만 동작 변경을 최소화하기 위한 예제입니다. 내부적으로 형식화된 클라이언트는 명명된 클라이언트를 사용하며, 기본적으로 이러한 "숨겨진" 명명된 클라이언트는 연결된 형식화된 클라이언트의 형식 이름으로 이동합니다. 이 경우 "숨김" 이름은 `nameof(Service)`이었고, 예제에서는 이를 그대로 유지했습니다.

기술적으로는 이전에 "숨겨진" 명명된 클라이언트가 "노출"되고 종속성이 Typed 클라이언트 인프라 대신 Keyed DI 인프라를 통해 충족되도록 Typed 클라이언트를 "래프 해제"하는 예제입니다.

방법: 기본적으로 Keyed DI에 옵트인

모든 단일 클라이언트에 대해 AddAsKeyed 호출할 필요가 없습니다. ConfigureHttpClientDefaults통해 "전역적으로"(모든 클라이언트 이름에 대해) 쉽게 옵트인할 수 있습니다. Keyed Services의 관점에서 보면, 이는 KeyedService.AnyKey 등록을 의미합니다.

C#

```
services.ConfigureHttpClientDefaults(b => b.AddAsKeyed());

services.AddHttpClient("first", /* ... */);
services.AddHttpClient("second", /* ... */);
services.AddHttpClient("third", /* ... */);

public class MyController(
    [FromKeyedServices("first")] HttpClient first,
    [FromKeyedServices("second")] HttpClient second,
    [FromKeyedServices("third")] HttpClient third)
//{ ...
```

"알 수 없는" 클라이언트 주의

> **ⓘ 참고**
>
> `KeyedService.AnyKey` 등록은 *임의의* 키 값을 특정 서비스 인스턴스로 매핑을 정의합니다. 그러나 결과적으로 컨테이너 유효성 검사가 적용되지 않으며, *잘못된* 키 값이 자동으로 *잘못된 인스턴스가 삽입될* 있습니다.

> **ⓘ 중요**

Keyed `HttpClient`경우 클라이언트 이름의 실수로 인해 이름이 등록되지 않은 클라이언트인 "알 수 없는" 클라이언트가 잘못 삽입될 수 있습니다.

일반적으로 이름이 지정된 클라이언트에 대해서도 마찬가지입니다. `IHttpClientFactory`는 옵션 패턴이와 함께 작동하는 방식과 일치하여 클라이언트 이름을 명시적으로 등록할 필요가 없습니다. 공장은 알 수 없는 이름에 대해 구성되지 않았거나, 보다 정확히는 기본 구성된`HttpClient`을 제공합니다.

> ⓘ **참고**
>
> 따라서 "기본적으로 키 입력" 접근 방식은 등록된 모든 `HttpClient`뿐만 아니라 `IHttpClientFactory` 모든 클라이언트가 *만들 수*있습니다.

C#

```
services.ConfigureHttpClientDefaults(b => b.AddAsKeyed());
services.AddHttpClient("known", /* ... */);

provider.GetRequiredKeyedService<HttpClient>("known");   // OK
provider.GetRequiredKeyedService<HttpClient>("unknown"); // OK (unconfigured
instance)
```

"옵트인" 전략 고려 사항

"글로벌" 옵트인은 한 줄로 된 옵트인이지만, "기본 제공"으로 작업하는 대신 이 기능이 여전히 필요한 것은 불행한 일입니다. 해당 결정에 대한 전체 컨텍스트 및 추론은 dotnet/runtime#89755 및 dotnet/runtime#104943 참조하세요. 요컨대, "기본 설정으로 켜기"의 주요 차단기는 `ServiceLifetime` "논쟁"입니다: DI 및 `IHttpClientFactory` 구현의 현재 (`9.0.0`) 상태에서는, 가능한 모든 상황에서 모든 `HttpClient`에 대해 합리적으로 안전한 단일 `ServiceLifetime`가 존재하지 않습니다. 그러나 향후 릴리스의 주의 사항을 해결하고 전략을 "옵트인"에서 "옵트아웃"으로 전환하려는 의도가 있습니다.

방법: 키 등록에서 옵트아웃

각 클라이언트 이름별로 RemoveAsKeyed 확장 메서드를 호출하여 `HttpClient`키드 DI에서 명시적으로 옵트아웃할 수 있습니다.

C#

```
services.ConfigureHttpClientDefaults(b => b.AddAsKeyed());      // opt IN by
default
```

```
services.AddHttpClient("keyed", /* ... */);
services.AddHttpClient("not-keyed", /* ... */).RemoveAsKeyed(); // opt OUT
per name

provider.GetRequiredKeyedService<HttpClient>("keyed");     // OK
provider.GetRequiredKeyedService<HttpClient>("not-keyed"); // Throws: No
service for type 'System.Net.Http.HttpClient' has been registered.
provider.GetRequiredKeyedService<HttpClient>("unknown");   // OK
(unconfigured instance)
```

또는 ConfigureHttpClientDefaults사용하여 "전역적으로" 다음을 수행합니다.

C#

```
services.ConfigureHttpClientDefaults(b => b.RemoveAsKeyed()); // opt OUT by
default
services.AddHttpClient("keyed", /* ... */).AddAsKeyed();      // opt IN per
name
services.AddHttpClient("not-keyed", /* ... */);

provider.GetRequiredKeyedService<HttpClient>("keyed");     // OK
provider.GetRequiredKeyedService<HttpClient>("not-keyed"); // Throws: No
service for type 'System.Net.Http.HttpClient' has been registered.
provider.GetRequiredKeyedService<HttpClient>("unknown");   // Throws: No
service for type 'System.Net.Http.HttpClient' has been registered.
```

우선 순위

함께 호출되거나 두 번 이상 호출되는 경우 `AddAsKeyed()` 및 `RemoveAsKeyed()` 일반적으로 `IHttpClientFactory` 구성 및 DI 등록 규칙을 따릅니다.

1. 동일한 이름으로 호출되는 경우 마지막 설정이 우선합니다. 마지막 `AddAsKeyed()` 수명은 키 등록을 만드는 데 사용됩니다(`RemoveAsKeyed()` 마지막으로 호출되지 않은 경우 이 경우 이름은 제외됨).
2. `ConfigureHttpClientDefaults`내에서만 사용하는 경우 마지막 설정이 우선합니다.
3. `ConfigureHttpClientDefaults` 및 특정 클라이언트 이름을 모두 사용한 경우 모든 기본값은 모든 이름별 설정 전에 "발생"하는 것으로 간주됩니다. 따라서 기본값을 무시할 수 있으며 이름별 설정의 마지막이 우선합니다.

참조

- .NET 사용하여 IHttpClientFactory
- .NET에서의 종속성 주입
- IHttpClientFactory

- 일반적인 IHttpClientFactory 사용상의 문제

복원력 있는 HTTP 앱 빌드: 주요 개발 패턴

일시적인 오류를 복구할 수 있는 강력한 HTTP 앱을 빌드하는 것은 일반적인 요구 사항입니다. 이 문서에서는 전달된 핵심 개념을 확장하므로 복원력 있는 앱 개발 소개를 이미 읽었다고 가정합니다. 복원력 있는 HTTP 앱을 빌드하는 데 도움이 되도록 Microsoft.Extensions.Http.Resilience NuGet 패키지는 특히 HttpClient에 대한 복원력 메커니즘을 제공합니다. 이 NuGet 패키지는 자주 사용되는 오픈 소스 프로젝트인 `Microsoft.Extensions.Resilience` 라이브러리와 *Polly*를 사용합니다. 자세한 내용은 Polly를 참조하세요.

시작하기

HTTP 앱에서 복원력 패턴을 사용하려면 Microsoft.Extensions.Http.Resilience NuGet 패키지를 설치합니다.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Http.Resilience
```

자세한 내용은 .NET 애플리케이션에서 dotnet 패키지 추가 또는 패키지 종속성 관리를 참조하세요.

HTTP 클라이언트에 복원력 추가

HttpClient에 복원력을 추가하려면 사용 가능한 IHttpClientBuilder 메서드를 호출하여 반환되는 AddHttpClient 형식에 호출을 연결합니다. 자세한 내용은 .NET을 사용한 IHttpClientFactory를 참조하세요.

여러 가지 복원력 중심 확장을 사용할 수 있습니다. 일부는 표준이므로 다양한 업계 모범 사례를 채택하고 다른 일부는 사용자 지정이 가능합니다. 복원력을 추가할 때 복원력 처리기를 하나만 추가하고 처리기를 겹쳐서는 안 됩니다. 여러 복원력 처리기를 추가해야 하는 경우 복원력 전략을 사용자 지정할 수 있는 `AddResilienceHandler` 확장 메서드 사용을 고려해야 합니다.

> **Important**
>
> 이 문서의 모든 예제는 AddHttpClient 인스턴스를 반환하는 Microsoft.Extensions.Http 라이브러리의 IHttpClientBuilder API를 사용합니다. IHttpClientBuilder 인스턴스는

HttpClient를 구성하고 복원력 처리기를 추가하는 데 사용됩니다. DI 컨테이너 없이 또는 `static`에 `HttpClient` 복원력을 추가해야 하는 경우 **정적 클라이언트를 사용한 복원력을 참조하세요**.

표준 복원력 처리기 추가

표준 복원력 처리기는 요청을 보내고 일시적인 오류를 처리하는 기본 옵션과 함께 서로 겹쳐진 여러 복원력 전략을 사용합니다. 표준 복원력 처리기는 `AddStandardResilienceHandler` 인스턴스에서 IHttpClientBuilder 확장 메서드를 호출하여 추가됩니다.

C#

```
var services = new ServiceCollection();

var httpClientBuilder = services.AddHttpClient<ExampleClient>(
    configureClient: static client =>
    {
        client.BaseAddress = new("https://jsonplaceholder.typicode.com");
    });
```

앞의 코드가 하는 역할은 다음과 같습니다.

- ServiceCollection 인스턴스를 만듭니다.
- 서비스 컨테이너에 HttpClient 형식에 대한 `ExampleClient`를 추가합니다.
- HttpClient을 기준 주소로 사용하도록 `"https://jsonplaceholder.typicode.com"`를 구성합니다.
- 이 문서의 다른 예 전반에 걸쳐 사용되는 `httpClientBuilder`를 만듭니다.

좀 더 실제적인 예는 .NET 일반 호스트 문서에 설명된 것과 같은 호스팅에 의존하는 것입니다. Microsoft.Extensions.Hosting NuGet 패키지를 사용하여 다음 업데이트된 예를 고려합니다.

C#

```
using Http.Resilience.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

IHttpClientBuilder httpClientBuilder =
builder.Services.AddHttpClient<ExampleClient>(
    configureClient: static client =>
    {
        client.BaseAddress = new("https://jsonplaceholder.typicode.com");
    });
```

이전 코드는 수동 `ServiceCollection` 만들기 방식과 유사하지만 대신 Host.CreateApplicationBuilder()를 사용하여 서비스를 노출하는 호스트를 빌드합니다.

`ExampleClient`는 다음과 같이 정의됩니다.

C#

```
using System.Net.Http.Json;

namespace Http.Resilience.Example;

/// <summary>
/// An example client service, that relies on the <see cref="HttpClient"/>
instance.
/// </summary>
/// <param name="client">The given <see cref="HttpClient"/> instance.</param>
internal sealed class ExampleClient(HttpClient client)
{
    /// <summary>
    /// Returns an <see cref="IAsyncEnumerable{T}"/> of <see cref="Comment"/>s.
    /// </summary>
    public IAsyncEnumerable<Comment?> GetCommentsAsync()
    {
        return client.GetFromJsonAsAsyncEnumerable<Comment>("/comments");
    }
}
```

앞의 코드가 하는 역할은 다음과 같습니다.

- `ExampleClient`를 허용하는 생성자가 있는 HttpClient 형식을 정의합니다.
- GET 요청을 `GetCommentsAsync` 엔드포인트로 보내고 응답을 반환하는 `/comments` 메서드를 노출합니다.

`Comment` 형식은 다음과 같이 정의됩니다.

C#

```
namespace Http.Resilience.Example;

public record class Comment(
    int PostId, int Id, string Name, string Email, string Body);
```

IHttpClientBuilder(`httpClientBuilder`)을 만들었고 이제 `ExampleClient` 구현과 해당 `Comment` 모델을 이해했다면 다음 예를 고려해 보세요.

C#

```
httpClientBuilder.AddStandardResilienceHandler();
```

앞의 코드는 표준 복원력 처리기를 HttpClient에 추가합니다. 대부분의 복원력 API와 마찬가지로 기본 옵션과 적용된 복원력 전략을 사용자 지정할 수 있는 오버로드가 있습니다.

표준 복원력 처리기 제거

이전에 등록된 모든 복원력 처리기를 제거하는 메서드 RemoveAllResilienceHandlers 있습니다. 사용자 지정 복원력을 추가하기 위해 기존 복원력 처리기를 지워야 하는 경우에 유용합니다. 다음 예제에서는 HttpClient 메서드를 사용하여 사용자 지정 `AddHttpClient` 구성하고, 미리 정의된 모든 복원력 전략을 제거하고, 새 처리기로 바꾸는 방법을 보여 줍니다. 이 방법을 사용하면 기존 구성을 지우고 특정 요구 사항에 따라 새 구성을 정의할 수 있습니다.

C#

```
// By default, we want all HttpClient instances to include the
StandardResilienceHandler.
services.ConfigureHttpClientDefaults(builder =>
builder.AddStandardResilienceHandler());
// For a named HttpClient "custom" we want to remove the StandardResilienceHandler
and add the StandardHedgingHandler instead.
services.AddHttpClient("custom")
    .RemoveAllResilienceHandlers()
    .AddStandardHedgingHandler();
```

앞의 코드가 하는 역할은 다음과 같습니다.

- ServiceCollection 인스턴스를 만듭니다.
- 모든 HttpClient 인스턴스에 표준 복원력 처리기를 추가합니다.
- "사용자 지정" HttpClient:
  - 이전에 등록된 모든 미리 정의된 복원력 처리기를 제거합니다. 이 기능은 고유한 사용자 지정 전략을 추가하기 위해 깨끗한 상태로 시작하려는 경우에 유용합니다.
  - `StandardHedgingHandler`를 HttpClient에 추가합니다. `AddStandardHedgingHandler()` 재시도 메커니즘, 회로 차단기 또는 기타 복원력 기술과 같이 애플리케이션의 요구에 맞는 전략으로 바꿀 수 있습니다.

표준 복원력 처리기 기본값

기본 구성은 다음 순서(가장 바깥쪽부터 가장 안쪽까지)로 5개의 복원력 전략을 연결합니다.

테이블 확장

순서	전략	설명	기본값
1	속도 제한기	속도 제한기 파이프라인은 종속성으로 전송되는 최대 동시 요청 수를 제한합니다.	큐: `0` 허용: `1_000`
2	총 타임아웃	총 요청 제한 시간 파이프라인은 실행에 전체 제한 시간을 적용하여 다시 시도를 포함한 요청이 구성된 제한을 초과하지 않도록 합니다.	총 시간 제한: 30초
3	재시도	다시 시도 파이프라인은 종속성이 느리거나 일시적인 오류를 반환하는 경우 요청을 다시 시도합니다.	최대 재시도: `3` 백오프: `Exponential` 지터 사용: `true` 지연: 2초
4	회로 차단기	너무 많은 직접적인 실패나 시간 초과가 탐지되면 회로 차단기가 실행을 차단합니다.	실패율: 10% 최소 처리량: `100` 샘플링 기간: 30초 휴식 시간: 5초
5	시도 시간 제한	시도 제한 시간 파이프라인은 각 요청 시도 기간을 제한하고 초과되면 오류를 발생시킵니다.	시도 시간 제한: 10초

다시 시도 및 회로 차단기

다시 시도 및 회로 차단기 전략은 모두 특정 HTTP 상태 코드 및 예외 집합을 처리합니다. 다음 HTTP 상태 코드를 고려합니다.

- HTTP 500 이상(서버 오류)
- HTTP 408(요청 시간 초과)
- HTTP 429(요청이 너무 많음)

또한 이러한 전략은 다음 예외를 처리합니다.

- `HttpRequestException`
- `TimeoutRejectedException`

지정된 HTTP 메서드 목록에 대한 재시도 사용 안 함

기본적으로 표준 복원력 처리기는 모든 HTTP 메서드에 대해 다시 시도하도록 구성됩니다. 일부 애플리케이션의 경우 이러한 동작은 바람직하지 않거나 심지어 해로울 수 있습니다. 예를 들어 POST 요청이 데이터베이스에 새 레코드를 삽입하는 경우 이러한 요청에 대해 다시 시도하면 데이터 중복이 발생할 수 있습니다. 지정된 HTTP 메서드 목록에 대한 재시도를 사용하지 않도록

설정해야 하는 경우 DisableFor(HttpRetryStrategyOptions, HttpMethod[]) 메서드를 사용할 수 있습니다.

C#

```
httpClientBuilder.AddStandardResilienceHandler(options =>
{
    options.Retry.DisableFor(HttpMethod.Post, HttpMethod.Delete);
});
```

또는 DisableForUnsafeHttpMethods(HttpRetryStrategyOptions), `POST`, `PATCH`, `PUT` 및 `DELETE` 요청에 대한 재시도를 사용하지 않도록 설정하는 `CONNECT` 메서드를 사용할 수 있습니다. RFC 에 따르면 이러한 메서드는 안전하지 않은 것으로 간주됩니다. 의미 체계는 읽기 전용이 아닙니다.

C#

```
httpClientBuilder.AddStandardResilienceHandler(options =>
{
    options.Retry.DisableForUnsafeHttpMethods();
});
```

표준 Hedging 처리기 추가

표준 헤징 처리기는 표준 헤징 메커니즘을 사용하여 요청의 실행을 처리합니다. Hedging은 느린 요청을 병렬로 다시 시도합니다.

표준 Hedging 처리기를 사용하려면 `AddStandardHedgingHandler` 확장 메서드를 호출합니다. 다음 예에서는 표준 Hedging 처리기를 사용하도록 `ExampleClient`를 구성합니다.

C#

```
httpClientBuilder.AddStandardHedgingHandler();
```

앞의 코드는 표준 Hedging 처리기를 HttpClient에 추가합니다.

표준 헤징 처리기 기본값

표준 헤징에서는 비정상적인 엔드포인트가 헤징되지 않도록 회로 차단기 풀을 사용하여 보장합니다. 기본적으로 풀의 선택은 URL 권한(구성표 + 호스트 + 포트)을 기반으로 합니다.

팁

고급 시나리오의 경우 `StandardHedgingHandlerBuilderExtensions.SelectPipelineByAuthority` 또는 `StandardHedgingHandlerBuilderExtensions.SelectPipelineBy`를 호출하여 전략이 선택되는 방식을 구성하는 것이 좋습니다.

앞의 코드는 표준 Hedging 처리기를 IHttpClientBuilder에 추가합니다. 기본 구성은 다음 순서(가장 바깥쪽부터 가장 안쪽까지)로 5개의 복원력 전략을 연결합니다.

테이블 확장

순서	전략	설명	기본값
1	총 요청 시간 초과	총 요청 제한 시간 파이프라인은 실행에 전체 제한 시간을 적용하여 Hedging 시도를 포함한 요청이 구성된 제한을 초과하지 않도록 합니다.	총 시간 제한: 30초
2	위험 회피	Hedging 전략은 종속성이 느리거나 일시적인 오류를 반환하는 경우 여러 엔드포인트에 대해 요청을 실행합니다. 라우팅은 옵션이며 기본적으로 원래 HttpRequestMessage에서 제공한 URL을 헤지합니다.	최소 시도 횟수: `1` 최대 시도 횟수: `10` 지연: 2초
3	속도 제한기(엔드포인트당)	속도 제한기 파이프라인은 종속성으로 전송되는 최대 동시 요청 수를 제한합니다.	큐: `0` 허용: `1_000`
4	회로 차단기(엔드포인트당)	너무 많은 직접적인 실패나 시간 초과가 탐지되면 회로 차단기가 실행을 차단합니다.	실패율: 10% 최소 처리량: `100` 샘플링 기간: 30초 휴식 시간: 5초
5	엔드포인트별 시도 시간 제한	시도 제한 시간 파이프라인은 각 요청 시도 기간을 제한하고 초과되면 오류를 발생시킵니다.	시간 제한: 10초

헤징 처리기 경로 선택을 사용자 지정하기

표준 Hedging 처리기를 사용할 때 `IRoutingStrategyBuilder` 형식에 대한 다양한 확장을 호출하여 요청 엔드포인트가 선택되는 방식을 사용자 지정할 수 있습니다. 이는 요청의 일부를 다른 엔드포인트로 라우팅하려는 A/B 테스트와 같은 시나리오에 유용할 수 있습니다.

C#

```
httpClientBuilder.AddStandardHedgingHandler(static (IRoutingStrategyBuilder
builder) =>
{
    // Hedging allows sending multiple concurrent requests
    builder.ConfigureOrderedGroups(static options =>
    {
        options.Groups.Add(new UriEndpointGroup()
        {
            Endpoints =
            {
                // Imagine a scenario where 3% of the requests are
                // sent to the experimental endpoint.
                new() { Uri = new("https://example.net/api/experimental"), Weight =
3 },
                new() { Uri = new("https://example.net/api/stable"), Weight = 97 }
            }
        });
    });
});
```

앞의 코드가 하는 역할은 다음과 같습니다.

- IHttpClientBuilder에 Hedging 처리기를 추가합니다.
- `IRoutingStrategyBuilder` 메서드를 사용하여 순서가 지정된 그룹을 구성하도록 `ConfigureOrderedGroups`를 구성합니다.
- 요청의 3%를 `EndpointGroup` 엔드포인트로 라우팅하고 요청의 97%를 `orderedGroup` 엔드포인트로 라우팅하는 `https://example.net/api/experimental`에 `https://example.net/api/stable`을 추가합니다.
- `IRoutingStrategyBuilder`를 `ConfigureWeightedGroups` 메서드를 사용하여 구성합니다.

가중치 적용 그룹을 구성하려면 `ConfigureWeightedGroups` 형식에서 `IRoutingStrategyBuilder` 메서드를 호출합니다. 다음 예에서는 `IRoutingStrategyBuilder` 메서드를 사용하여 가중치 적용 그룹을 구성하도록 `ConfigureWeightedGroups`를 구성합니다.

C#

```
httpClientBuilder.AddStandardHedgingHandler(static (IRoutingStrategyBuilder
builder) =>
{
    // Hedging allows sending multiple concurrent requests
    builder.ConfigureWeightedGroups(static options =>
    {
        options.SelectionMode = WeightedGroupSelectionMode.EveryAttempt;

        options.Groups.Add(new WeightedUriEndpointGroup()
        {
            Endpoints =
```

```
            {
                // Imagine A/B testing
                new() { Uri = new("https://example.net/api/a"), Weight = 33 },
                new() { Uri = new("https://example.net/api/b"), Weight = 33 },
                new() { Uri = new("https://example.net/api/c"), Weight = 33 }
            }
        });
    });
});
```

앞의 코드가 하는 역할은 다음과 같습니다.

- IHttpClientBuilder에 Hedging 처리기를 추가합니다.
- 가중치 적용 그룹을 구성하기 위해 `IRoutingStrategyBuilder` 메서드를 사용하도록 `ConfigureWeightedGroups`를 구성합니다.
- `SelectionMode`를 `WeightedGroupSelectionMode.EveryAttempt`으로 설정합니다.
- 요청의 33%를 `WeightedEndpointGroup` 엔드포인트로, 요청의 33%를 `weightedGroup` 엔드포인트로, 요청의 33%를 `https://example.net/api/a` 엔드포인트로 라우팅하는 `https://example.net/api/b`에 `https://example.net/api/c`을 추가합니다.

> **팁**
>
> 최대 Hedging 시도 횟수는 구성된 그룹 수와 직접적으로 연관됩니다. 예를 들어, 그룹이 2개 있는 경우 최대 시도 횟수는 2입니다.

자세한 내용은 Polly 문서: 헤징 복원력 전략을 참조하세요.

순서가 지정된 그룹이나 가중치가 부여된 그룹을 구성하는 것이 일반적이지만 둘 다 구성하는 것도 유효합니다. 순서가 지정된 가중치 그룹을 사용하는 것은 A/B 테스트와 같이 요청의 일부를 다른 엔드포인트로 보내려는 시나리오에서 유용합니다.

사용자 지정 복원력 처리기 추가

더 효과적으로 제어하려면 `AddResilienceHandler` API를 사용하여 복원력 처리기를 사용자 지정할 수 있습니다. 이 메서드는 복원력 전략을 만드는 데 사용되는 `ResiliencePipelineBuilder<HttpResponseMessage>` 인스턴스를 구성하는 대리자를 허용합니다.

명명된 복원력 처리기를 구성하려면 처리기 이름으로 `AddResilienceHandler` 확장 메서드를 호출합니다. 다음 예에서는 `"CustomPipeline"`이라는 명명된 복원력 처리기를 구성합니다.

C#

```
httpClientBuilder.AddResilienceHandler(
    "CustomPipeline",
    static builder =>
{
    // See: https://www.pollydocs.org/strategies/retry.html
    builder.AddRetry(new HttpRetryStrategyOptions
    {
        // Customize and configure the retry logic.
        BackoffType = DelayBackoffType.Exponential,
        MaxRetryAttempts = 5,
        UseJitter = true
    });

    // See: https://www.pollydocs.org/strategies/circuit-breaker.html
    builder.AddCircuitBreaker(new HttpCircuitBreakerStrategyOptions
    {
        // Customize and configure the circuit breaker logic.
        SamplingDuration = TimeSpan.FromSeconds(10),
        FailureRatio = 0.2,
        MinimumThroughput = 3,
        ShouldHandle = static args =>
        {
            return ValueTask.FromResult(args is
            {
                Outcome.Result.StatusCode:
                    HttpStatusCode.RequestTimeout or
                        HttpStatusCode.TooManyRequests
            });
        }
    });

    // See: https://www.pollydocs.org/strategies/timeout.html
    builder.AddTimeout(TimeSpan.FromSeconds(5));
});
```

앞의 코드가 하는 역할은 다음과 같습니다.

- 이름이 `"CustomPipeline"`인 복원 처리기를 `pipelineName`으로 서비스 컨테이너에 추가합니다.
- "복원력 빌더에 지수 백오프, 5회 재시도, 및 지터 선택 옵션이 포함된 재시도 전략을 추가합니다."
- 샘플링 기간 10초, 실패율 0.2(20%), 최소 처리량 3, `RequestTimeout` 및 `TooManyRequests` HTTP 상태 코드를 처리하는 조건자를 포함하는 회로 차단기 전략을 복원력 작성기에 추가합니다.
- 복원력 빌더에 5초 시간 제한을 두는 시간 제한 전략을 추가합니다.

각 복원력 전략에 사용할 수 있는 다양한 옵션이 있습니다. 자세한 내용은 Polly 문서: 전략 을 참조하세요. `ShouldHandle` 대리자 구성에 대한 자세한 내용은 Polly 문서: 대응 전략의 오류 처리 를 참조하세요.

> ⚠ **Warning**
>
> 재시도 및 시간 제한 전략을 모두 사용하고 재시도 전략의 `ShouldHandle` 대리자를 구성하려는 경우, Polly의 시간 제한 예외를 처리해야 하는지 여부를 고려해야 합니다. Polly는 표준 `TimeoutRejectedException`이 아닌 **Exception**을 throw하며, 이는 **TimeoutException**로부터 상속됩니다.

동적 새로 고침

Polly는 구성된 복원력 전략의 동적 재로드를 지원합니다. 즉, 런타임에 복원력 전략의 구성을 변경할 수 있습니다. 동적 다시 로드를 사용하도록 설정하려면 `AddResilienceHandler`를 노출하는 적절한 `ResilienceHandlerContext` 오버로드를 사용합니다. 컨텍스트가 주어지면 해당 복원력 전략 옵션의 `EnableReloads`를 호출합니다.

C#

```
httpClientBuilder.AddResilienceHandler(
    "AdvancedPipeline",
    static (ResiliencePipelineBuilder<HttpResponseMessage> builder,
        ResilienceHandlerContext context) =>
    {
        // Enable reloads whenever the named options change
        context.EnableReloads<HttpRetryStrategyOptions>("RetryOptions");

        // Retrieve the named options
        var retryOptions =
            context.GetOptions<HttpRetryStrategyOptions>("RetryOptions");

        // Add retries using the resolved options
        builder.AddRetry(retryOptions);
    });
```

앞의 코드가 하는 역할은 다음과 같습니다.

- 이름이 `"AdvancedPipeline"`인 복원 처리기를 `pipelineName`으로 서비스 컨테이너에 추가합니다.
- 명명된 `"AdvancedPipeline"` 옵션이 변경될 때마다 `RetryStrategyOptions` 파이프라인을 다시 로드할 수 있습니다.
- IOptionsMonitor<TOptions> 서비스에서 명명된 옵션을 검색합니다.
- 복원력 설정 도구에 검색된 옵션이 포함된 다시 시도 전략을 추가합니다.

자세한 내용은 Polly 문서: 고급 종속성 주입을 참조하세요.

이 예는 *appsettings.json* 파일과 같이 변경 가능한 옵션 섹션을 사용합니다. 다음 *appsettings.json* 파일을 고려하세요.

JSON

```
{
    "RetryOptions": {
        "Retry": {
            "BackoffType": "Linear",
            "UseJitter": false,
            "MaxRetryAttempts": 7
        }
    }
}
```

이제 이러한 옵션이 앱 구성에 바인딩되어 `HttpRetryStrategyOptions`를 `"RetryOptions"` 섹션에 바인딩했다고 상상해 보세요.

C#

```
var section = builder.Configuration.GetSection("RetryOptions");

builder.Services.Configure<HttpStandardResilienceOptions>(section);
```

자세한 내용은 .NET의 옵션 패턴을 참조하세요.

예제 사용

앱은 종속성 주입을 사용하여 `ExampleClient` 및 해당 HttpClient를 해결합니다. 코드는 IServiceProvider를 빌드하고 `ExampleClient`를 해결합니다.

C#

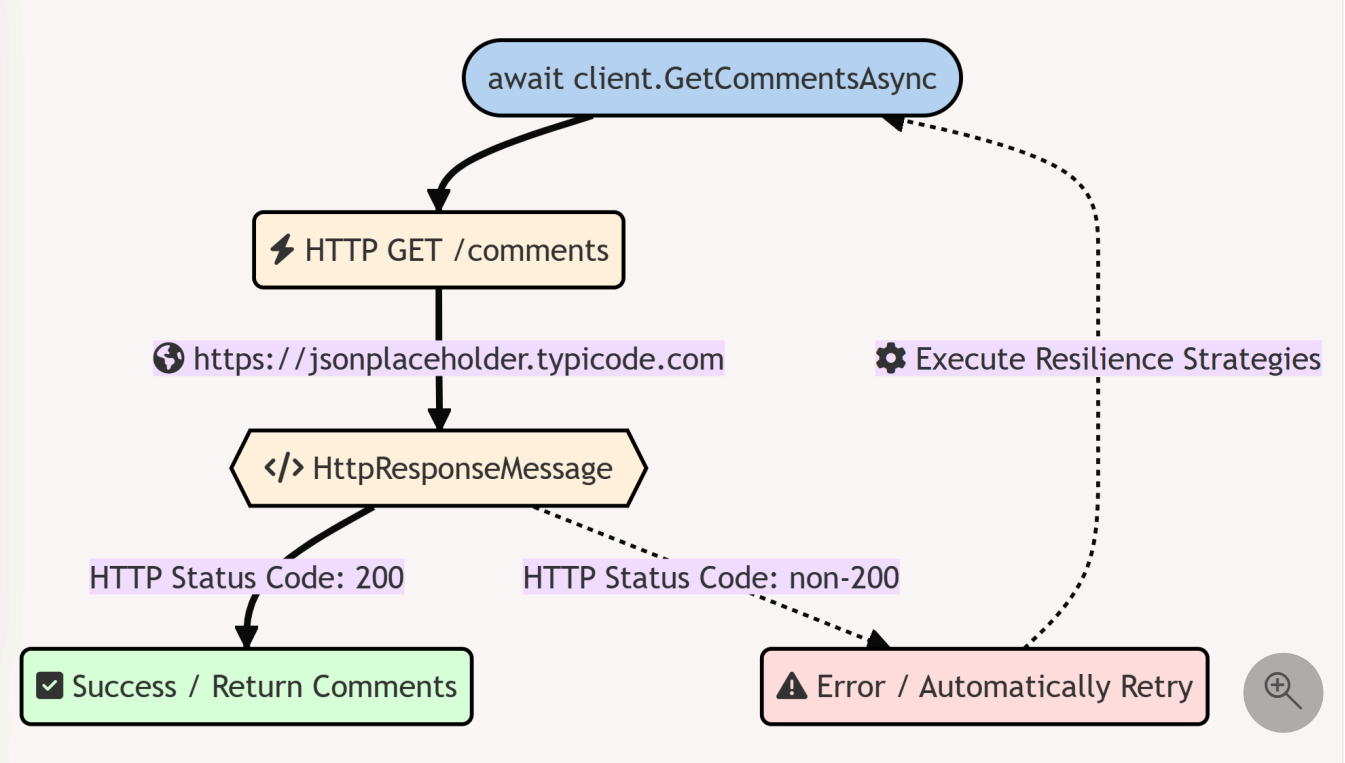
```
IHost host = builder.Build();

ExampleClient client = host.Services.GetRequiredService<ExampleClient>();

await foreach (Comment? comment in client.GetCommentsAsync())
{
    Console.WriteLine(comment);
}
```

앞의 코드가 하는 역할은 다음과 같습니다.

- IServiceProvider에서 ServiceCollection를 빌드합니다.
- `ExampleClient`에서 IServiceProvider를 해결합니다.

- 주석을 가져오기 위해 `GetCommentsAsync`에서 `ExampleClient` 메서드를 호출합니다.
- 각 주석을 콘솔에 씁니다.

네트워크가 다운되거나 서버가 응답하지 않는 상황을 상상해 보세요. 다음 다이어그램은 `ExampleClient` 및 `GetCommentsAsync` 메서드를 고려하여 복원력 전략이 상황을 처리하는 방법을 보여 줍니다.



앞의 다이어그램은 다음을 보여 줍니다.

- `ExampleClient`는 HTTP GET 요청을 `/comments` 엔드포인트로 보냅니다.
- HttpResponseMessage가 평가됩니다.
  - 응답이 성공하면(HTTP 200) 응답이 반환됩니다.
  - 응답이 실패하면(HTTP 200이 아님) 복원력 파이프라인은 구성된 복원력 전략을 사용합니다.

이는 간단한 예이지만 복원력 전략을 사용하여 일시적인 오류를 처리하는 방법을 보여 줍니다. 자세한 내용은 Polly 문서: 전략을 참조하세요.

알려진 문제

다음 섹션에서는 알려진 다양한 문제에 대해 자세히 알아봅니다.

`Grpc.Net.ClientFactory` 패키지와의 호환성

`Grpc.Net.ClientFactory` 버전 `2.63.0` 이하를 사용하는 경우, gRPC 클라이언트에 대해 표준 복원력 또는 hedging 처리기를 사용하도록 설정하면 런타임 예외가 발생할 수 있습니다. 특히 다음 코드 샘플을 고려합니다.

C#

```
services
    .AddGrpcClient<Greeter.GreeterClient>()
    .AddStandardResilienceHandler();
```

위의 코드는 다음과 같은 예외를 발생시킵니다.

Output

```
System.InvalidOperationException: The ConfigureHttpClient method isn't supported
when creating gRPC clients. Unable to create client with name 'GreeterClient'.
```

이 문제를 해결하려면 `Grpc.Net.ClientFactory` 버전 `2.64.0` 이상으로 업그레이드하는 것이 좋습니다.

빌드 시간 확인을 통해 `Grpc.Net.ClientFactory` 버전 `2.63.0` 이하를 사용하고 있는지 확인하고, 사용 중인 경우 컴파일 경고가 생성됩니다. 프로젝트 파일에서 다음 속성을 설정하여 경고를 표시하지 않을 수 있습니다.

XML

```
<PropertyGroup>

<SuppressCheckGrpcNetClientFactoryVersion>true</SuppressCheckGrpcNetClientFactoryVe
rsion>
</PropertyGroup>
```

.NET Application Insights와의 호환성

.NET Application Insights 버전 **2.22.0** 이하를 사용하는 경우 애플리케이션에서 복원력 기능을 사용하도록 설정하면 모든 Application Insights 원격 분석이 누락될 수 있습니다. 이 문제는 Application Insights 서비스 전에 복원력 기능이 등록될 때 발생합니다. 문제를 일으키는 다음 샘플을 고려해 보세요.

C#

```
// At first, we register resilience functionality.
services.AddHttpClient().AddStandardResilienceHandler();

// And then we register Application Insights. As a result, Application Insights
```

```
doesn't work.
services.AddApplicationInsightsTelemetry();
```

.NET Application Insights를 버전 **2.23.0** 이상으로 업데이트하여 문제를 해결할 수 있습니다. 업데이트할 수 없는 경우 아래와 같이 복원력 기능 전에 Application Insights 서비스를 등록하면 문제가 해결됩니다.

C#

```
// We register Application Insights first, and now it is working correctly.
services.AddApplicationInsightsTelemetry();
services.AddHttpClient().AddStandardResilienceHandler();
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 24.

HttpClient에서 HTTP/3 사용

2025. 06. 17.

HTTP/3 은 세 번째이자 최근에 표준화된 HTTP 주 버전입니다. HTTP/3은 HTTP/1.1 및 HTTP/2와 동일한 의미 체계를 사용하기 때문에 동일한 요청 메서드, 상태 코드 및 메시지 필드가 모든 버전에 적용됩니다. 기본 전송에 차이가 있습니다. HTTP/1.1과 HTTP/2 모두 TCP를 전송으로 사용합니다. HTTP/3은 QUIC 라는 HTTP/3과 함께 개발된 전송 기술을 사용합니다.

HTTP/3 및 QUIC는 HTTP/1.1 및 HTTP/2에 비해 몇 가지 이점이 있습니다.

- 첫 번째 요청에 대한 응답 시간이 빨라집니다. QUIC와 HTTP/3은 클라이언트와 서버 간의 왕복을 줄이면 연결을 협상합니다. 첫 번째 요청이 서버에 더 빨리 도달합니다.
- 연결 패킷 손실이 있을 때의 환경이 향상되었습니다. HTTP/2는 하나의 TCP 연결을 통해 여러 요청을 멀티플렉싱합니다. 연결의 패킷 손실은 모든 요청에 영향을 미칩니다. 이 문제를 'HOC(head-of-line) 블로킹'이라고 합니다. QUIC는 네이티브 멀티플렉싱을 제공하므로 손실된 패킷은 데이터가 손실된 요청에만 영향을 줍니다.
- 네트워크 간 전환을 지원합니다. 이 기능은 모바일 디바이스가 위치를 변경함에 따라 WIFI와 셀룰러 네트워크 간에 전환하는 것이 일반적인 모바일 디바이스에 유용합니다. 현재 네트워크를 전환할 때 오류와 함께 HTTP/1.1 및 HTTP/2 연결이 실패합니다. 앱 또는 웹 브라우저는 실패한 HTTP 요청을 다시 시도해야 합니다. HTTP/3을 사용하면 네트워크가 변경될 때도 앱 또는 웹 브라우저가 원활하게 지속될 수 있습니다. HttpClient 및 Kestrel은 .NET 7에서 네트워크 전환을 지원하지 않습니다. 향후 릴리스에서는 지원될 수 있을 것입니다.

> **ⓘ 중요**
>
> HTTP/3을 활용하도록 구성된 앱은 HTTP/1.1 및 HTTP/2도 지원하도록 디자인되어야 합니다. HTTP/3에서 문제가 식별되면 이후 .NET 릴리스에서 문제가 해결될 때까지 HTTP/3을 사용하지 않도록 설정하는 것이 좋습니다.

HttpClient 설정

HTTP 버전은 3.0으로 설정 `HttpRequestMessage.Version` 하여 구성할 수 있습니다. 그러나 모든 라우터, 방화벽 및 프록시가 HTTP/3을 제대로 지원하지 않으므로 HTTP/1.1 및 HTTP/2와 함께 HTTP/3을 구성하는 것이 좋습니다. 에서 `HttpClient` 다음을 지정하여 이 작업을 수행할 수 있습니다.

- HttpRequestMessage.Version 에서 1.1로
- HttpRequestMessage.VersionPolicy가 HttpVersionPolicy.RequestVersionOrHigher로 변경되었습니다.

플랫폼 종속성

HTTP/3은 QUIC를 전송 프로토콜로 사용합니다. HTTP/3의 .NET 구현에서는 MsQuic 를 사용하여 QUIC 기능을 제공합니다. 따라서 HTTP/3의 .NET 지원은 MsQuic 플랫폼 요구 사항에 따라 달라집니다. **MsQuic**를 설치하는 방법에 대한 자세한 내용은 QUIC 플랫폼 종속성을 참조하세요. HttpClient가 실행되는 플랫폼에 HTTP/3에 대한 모든 요구 사항이 없는 경우 사용하지 않도록 설정됩니다.

HttpClient 사용

다음 코드 예제에서는 최상위 문을 사용하고 요청에서 HTTP3을 지정하는 방법을 보여 줍니다.

C#

```
// See https://aka.ms/new-console-template for more information
using System.Net;

using var client = new HttpClient
{
    DefaultRequestVersion =  HttpVersion.Version30,
    DefaultVersionPolicy = HttpVersionPolicy.RequestVersionExact
};

Console.WriteLine("--- localhost:5001 ---");

HttpResponseMessage resp = await client.GetAsync("https://localhost:5001/");
string body = await resp.Content.ReadAsStringAsync();

Console.WriteLine(
    $"status: {resp.StatusCode}, version: {resp.Version}, " +
    $"body: {body.Substring(0, Math.Min(100, body.Length))}");
```

.NET 6의 HTTP/3 지원

.NET 6에서는 HTTP/3 사양이 아직 확정되지 않았기 때문에 HTTP/3을 *미리 보기 기능* 으로 사용할 수 있습니다. 동작 또는 성능 문제는 .NET 6이 있는 HTTP/3에 있을 수 있습니다. 미리 보기 기능에 대한 자세한 내용은 미리 보기 기능 사양을 참조하세요.

.NET 6에서 HTTP/3 지원을 활성화하려면 HTTP/3을 사용하도록 설정하는 데 필요한 노드 `RuntimeHostConfigurationOption` 및 `HttpClient`을 프로젝트 파일에 포함하세요.

XML

```
<ItemGroup>
    <RuntimeHostConfigurationOption Value="true"
```

```
        Include="System.Net.SocketsHttpHandler.Http3Support" />
</ItemGroup>
```

또는 앱 코드에서 호출 System.AppContext.SetSwitch 하거나 환경 변수 `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP3SUPPORT`를 `true` .로 설정할 수 있습니다. 자세한 내용은 .NET 환경 변수(DOTNET_SYSTEM_NET_HTTP_*)를 참조하세요.

HTTP/3에 대한 구성 플래그를 요구하는 이유는 버전 정책을 `RequestVersionOrHigher`사용할 때 향후 중단으로부터 앱을 보호하기 위한 것입니다. 현재 HTTP/1.1 및 HTTP/2를 사용하는 서버를 호출할 때 서버가 나중에 HTTP/3으로 업그레이드하는 경우 클라이언트는 HTTP/3을 사용하려고 시도하며 표준이 최종적이지 않으므로 호환되지 않을 수 있으므로 .NET 6이 릴리스된 후에 변경될 수 있습니다.

.NET 6은 libmsquic의 1.9.x 버전과만 호환됩니다. 라이브러리의 호환성이 손상되는 변경으로 인해 Libmsquic 2.x는 .NET 6과 호환되지 않습니다. Libmsquic는 보안 픽스를 통합하는 데 필요한 경우 1.9.x에 대한 업데이트를 받습니다.

HTTP/3 서버

ASP.NET은 Kestrel 서버를 통해 HTTP/3을 지원하며, .NET 6에서는 미리 보기로, .NET 7에서는 완전히 지원됩니다. 자세한 내용은 ASP.NET Core Kestrel 웹 서버에서 HTTP/3 사용을 참조하세요.

공용 테스트 서버

Cloudflare는 클라이언트를 테스트 https://cloudflare-quic.com 하는 데 사용할 수 있는 HTTP/3 사이트를 호스트합니다.

참고하십시오

- HttpClient
- Kestrel의 HTTP/3 지원
- .NET의 QUIC 지원

.NET의 HTTP 처리기 속도 제한

2025. 06. 17.

이 문서에서는 전송하는 요청 수를 제한하는 클라이언트 쪽 HTTP 처리기를 만드는 방법을 알아봅니다. HttpClient가 `"www.example.com"` 리소스를 액세스하는 것을 볼 수 있습니다. 리소스는 리소스를 사용하는 앱에서 사용되며, 앱이 단일 리소스에 대해 너무 많은 요청을 하면 *리소스 경합*이 발생할 수 있습니다. 리소스 경합은 리소스가 너무 많은 앱에서 사용되고 리소스가 리소스를 요청하는 모든 앱을 제공할 수 없는 경우에 발생합니다. 이로 인해 사용자 환경이 저하될 수 있으며 경우에 따라 DoS(서비스 거부) 공격으로 이어질 수도 있습니다. DoS에 대한 자세한 내용은 OWASP: 서비스 거부를 참조하세요.

속도 제한이란?

속도 제한은 리소스에 액세스할 수 있는 양을 제한하는 개념입니다. 예를 들어 앱이 액세스하는 데이터베이스가 분당 1,000개의 요청을 안전하게 처리할 수 있지만 그 이상을 처리하지 못할 수 있습니다. 1분마다 1,000개의 요청만 허용하고 데이터베이스에 액세스하기 전에 더 이상 요청을 거부하는 속도 제한기를 앱에 넣을 수 있습니다. 따라서 데이터베이스의 요청 속도를 제한하여 앱이 안전한 수의 요청을 처리할 수 있도록 합니다. 이는 분산 시스템의 일반적인 패턴으로, 여러 개의 앱 인스턴스가 실행 중일 수 있으며 모든 인스턴스가 동시에 데이터베이스에 액세스하지 않도록 하려고 합니다. 요청 흐름을 제어하는 여러 가지 속도 제한 알고리즘이 있습니다.

.NET에서 속도 제한을 사용하려면 System.Threading.RateLimiting NuGet 패키지를 참조합니다.

`DelegatingHandler` 하위 클래스 구현

요청 흐름을 제어하려면 사용자 지정 DelegatingHandler 하위 클래스를 구현합니다. 서버로 전송되기 전에 요청을 가로채고 처리할 수 있는 형식 HttpMessageHandler 입니다. 호출자에게 반환되기 전에 응답을 가로채고 처리할 수도 있습니다. 이 예제에서는 단일 리소스로 보낼 수 있는 요청 수를 제한하는 사용자 지정 `DelegatingHandler` 서브클래스를 구현합니다. 다음 사용자 지정 `ClientSideRateLimitedHandler` 클래스를 고려합니다.

C#

```
internal sealed class ClientSideRateLimitedHandler(
    RateLimiter limiter)
    : DelegatingHandler(new HttpClientHandler()), IAsyncDisposable
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        using RateLimitLease lease = await limiter.AcquireAsync(
```

```
            permitCount: 1, cancellationToken);

        if (lease.IsAcquired)
        {
            return await base.SendAsync(request, cancellationToken);
        }

        var response = new HttpResponseMessage(HttpStatusCode.TooManyRequests);
        if (lease.TryGetMetadata(
                MetadataName.RetryAfter, out TimeSpan retryAfter))
        {
            response.Headers.Add(
                "Retry-After",
                ((int)retryAfter.TotalSeconds).ToString(
                    NumberFormatInfo.InvariantInfo));
        }

        return response;
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        await limiter.DisposeAsync().ConfigureAwait(false);

        Dispose(disposing: false);
        GC.SuppressFinalize(this);
    }

    protected override void Dispose(bool disposing)
    {
        base.Dispose(disposing);

        if (disposing)
        {
            limiter.Dispose();
        }
    }
}
```

앞의 C# 코드는 다음과 같습니다.

- `DelegatingHandler` 형식을 상속합니다.
- IAsyncDisposable 인터페이스를 구현합니다.
- `RateLimiter` 생성자에서 할당된 필드를 정의합니다.
- 메서드 `SendAsync`를 재정의하여 서버로 전송되기 전에 요청을 가로채고 처리합니다.
- DisposeAsync() 메서드를 재정의하여 `RateLimiter` 인스턴스를 삭제합니다.

메서드를 좀 더 자세히 `SendAsync` 살펴보면 다음을 확인할 수 있습니다.

- `RateLimiter` 인스턴스를 통해 `RateLimitLease`에서 `AcquireAsync`를 획득합니다.
- 속성이 `lease.IsAcquired` 있으면 `true`요청이 서버로 전송됩니다.

- 그렇지 않으면 HttpResponseMessage가 `429` 상태 코드와 함께 반환되고, `lease`에 `RetryAfter` 값이 포함되어 있으면 `Retry-After` 헤더가 해당 값으로 설정됩니다.

여러 동시 요청 에뮬레이트

이 사용자 지정 `DelegatingHandler` 하위 클래스를 테스트에 배치하려면 여러 동시 요청을 에뮬레이트하는 콘솔 앱을 만듭니다. 이 `Program` 클래스는 사용자 지정HttpClient을 사용하여 `ClientSideRateLimitedHandler` 만듭니다.

C#

```
var options = new TokenBucketRateLimiterOptions
{
    TokenLimit = 8,
    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
    QueueLimit = 3,
    ReplenishmentPeriod = TimeSpan.FromMilliseconds(1),
    TokensPerPeriod = 2,
    AutoReplenishment = true
};

// Create an HTTP client with the client-side rate limited handler.
using HttpClient client = new(
    handler: new ClientSideRateLimitedHandler(
        limiter: new TokenBucketRateLimiter(options)));

// Create 100 urls with a unique query string.
var oneHundredUrls = Enumerable.Range(0, 100).Select(
    i => $"https://example.com?iteration={i:0#}");

// Flood the HTTP client with requests.
var floodOneThroughFortyNineTask = Parallel.ForEachAsync(
    source: oneHundredUrls.Take(0..49),
    body: (url, cancellationToken) => GetAsync(client, url, cancellationToken));

var floodFiftyThroughOneHundredTask = Parallel.ForEachAsync(
    source: oneHundredUrls.Take(^50..),
    body: (url, cancellationToken) => GetAsync(client, url, cancellationToken));

await Task.WhenAll(
    floodOneThroughFortyNineTask,
    floodFiftyThroughOneHundredTask);

static async ValueTask GetAsync(
    HttpClient client, string url, CancellationToken cancellationToken)
{
    using var response =
        await client.GetAsync(url, cancellationToken);

    Console.WriteLine(
        $"URL: {url}, HTTP status code: {response.StatusCode}
```

```
({(int)response.StatusCode})");
}
```

이전 콘솔 앱에서:

- `TokenBucketRateLimiterOptions` 토큰 제한 `8`및 큐 처리 순서`OldestFirst`, 큐 제한`3`(밀리초) 및 보충 기간`1`, 기간 값`2`당 토큰 및 자동 보충 값`true`으로 구성됩니다.
- `HttpClient`은 `ClientSideRateLimitedHandler`로 구성되어 `TokenBucketRateLimiter`을 만들어냅니다.
- 100개의 Enumerable.Range 요청을 에뮬레이트하려면 각각 고유한 쿼리 문자열 매개 변수가 있는 100개의 URL을 만듭니다.
- 두 Task 개체는 Parallel.ForEachAsync 메서드에 의해 할당되어 URL을 두 그룹으로 분할합니다.
- 각 `HttpClient` URL에 `GET` 요청을 보내는 데 사용되며 응답은 콘솔에 기록됩니다.
- Task.WhenAll 는 두 작업이 모두 완료되기를 기다립니다.

`HttpClient`가 `ClientSideRateLimitedHandler`로 구성되어 있기 때문에 모든 요청이 서버 리소스에 도달하는 것은 아닙니다. 콘솔 앱을 실행하여 이 어설션을 테스트할 수 있습니다. 총 요청 수의 일부만 서버로 전송되고 나머지는 HTTP 상태 코드 `429`로 거부됩니다. 서버로 전송되는 요청 수가 어떻게 변경되는지 확인하기 위해 `options`를 만드는 데 사용되는 `TokenBucketRateLimiter` 개체를 변경해 보십시오.

다음 예제 출력을 고려합니다.

Output

```
URL: https://example.com?iteration=06, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=60, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=55, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=59, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=57, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=11, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=63, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=13, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=62, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=65, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=64, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=67, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=14, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=68, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=16, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=69, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=70, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=71, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=17, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=18, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=72, HTTP status code: TooManyRequests (429)
```

```
URL: https://example.com?iteration=73, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=74, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=19, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=75, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=76, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=79, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=77, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=21, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=78, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=81, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=22, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=80, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=20, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=82, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=83, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=23, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=84, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=24, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=85, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=86, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=25, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=87, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=26, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=88, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=89, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=27, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=90, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=28, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=91, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=94, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=29, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=93, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=96, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=92, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=95, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=31, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=30, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=97, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=98, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=99, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=32, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=33, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=34, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=35, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=36, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=37, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=38, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=39, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=40, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=41, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=42, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=43, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=44, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=45, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=46, HTTP status code: TooManyRequests (429)
```

```
URL: https://example.com?iteration=47, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=48, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=15, HTTP status code: OK (200)
URL: https://example.com?iteration=04, HTTP status code: OK (200)
URL: https://example.com?iteration=54, HTTP status code: OK (200)
URL: https://example.com?iteration=08, HTTP status code: OK (200)
URL: https://example.com?iteration=00, HTTP status code: OK (200)
URL: https://example.com?iteration=51, HTTP status code: OK (200)
URL: https://example.com?iteration=10, HTTP status code: OK (200)
URL: https://example.com?iteration=66, HTTP status code: OK (200)
URL: https://example.com?iteration=56, HTTP status code: OK (200)
URL: https://example.com?iteration=52, HTTP status code: OK (200)
URL: https://example.com?iteration=12, HTTP status code: OK (200)
URL: https://example.com?iteration=53, HTTP status code: OK (200)
URL: https://example.com?iteration=07, HTTP status code: OK (200)
URL: https://example.com?iteration=02, HTTP status code: OK (200)
URL: https://example.com?iteration=01, HTTP status code: OK (200)
URL: https://example.com?iteration=61, HTTP status code: OK (200)
URL: https://example.com?iteration=05, HTTP status code: OK (200)
URL: https://example.com?iteration=09, HTTP status code: OK (200)
URL: https://example.com?iteration=03, HTTP status code: OK (200)
URL: https://example.com?iteration=58, HTTP status code: OK (200)
URL: https://example.com?iteration=50, HTTP status code: OK (200)
```

첫 번째 로깅된 항목은 항상 즉시 반환되는 429개의 응답이며, 마지막 항목은 항상 200개의 응답입니다. 이는 속도 제한이 클라이언트 쪽에 발생하고 서버에 대한 HTTP 호출을 방지하기 때문입니다. 이는 서버가 요청으로 넘쳐나지 않는다는 것을 의미하기 때문에 좋은 일입니다. 또한 속도 제한이 모든 클라이언트에서 일관되게 적용됨을 의미합니다.

또한 각 URL의 쿼리 문자열은 고유합니다. 매개 변수를 `iteration` 검사하여 각 요청에 대해 하나씩 증가되는지 확인합니다. 이 매개 변수는 429 응답이 첫 번째 요청의 응답이 아니라 속도 제한에 도달한 후에 수행되는 요청의 응답임을 보여 주는 데 도움이 됩니다. 200개의 응답은 나중에 도착하지만 이러한 요청은 제한에 도달하기 전에 이전에 이루어졌습니다.

다양한 속도 제한 알고리즘을 더 잘 이해하려면 다른 RateLimiter 구현을 허용하도록 이 코드를 다시 작성해 보세요. 다음 TokenBucketRateLimiter 외에도 다음을 시도할 수 있습니다.

- ConcurrencyLimiter
- FixedWindowRateLimiter
- PartitionedRateLimiter
- SlidingWindowRateLimiter

요약

이 문서에서는 사용자 지정 `ClientSideRateLimitedHandler`을 구현하는 방법을 알아보았습니다. 이 패턴은 API 제한이 있는 리소스에 대해 속도 제한 HTTP 클라이언트를 구현하는 데 사용할 수 있습니다. 이러한 방식으로 클라이언트 앱이 서버에 불필요한 요청을 하지 못하게 하고 서버에

의해 앱이 차단되지 않도록 방지합니다. 또한 메타데이터를 사용하여 재시도 타이밍 값을 저장하면 자동 재시도 논리를 구현할 수도 있습니다.

참고하십시오

- .NET에 대한 속도 제한 발표
- ASP.NET Core의 속도 제한 미들웨어
- Azure 아키텍처: 속도 제한 패턴
- .NET의 자동 재시도 논리

HTTP 요청에서 SNI 사용자 지정

2025. 06. 17.

클라이언트와 서버가 HTTPS 연결을 협상하는 경우 먼저 TLS 연결을 설정해야 합니다. TLS 핸드셰이크의 일부로 클라이언트는 TLS 확장 중 하나에서 연결하는 서버의 도메인 이름을 보냅니다. 여러 (가상) 서버가 동일한 컴퓨터에서 호스트되는 경우 TLS 프로토콜의 이 기능을 통해 클라이언트는 연결 중인 서버를 구분하고 서버 인증서와 같은 TLS 설정을 구성할 수 있습니다.

HTTP 요청을 사용하는 `HttpClient` 경우 구현은 클라이언트가 연결하는 URL에 따라 SNI(서버 이름 표시) 확장에 대한 값을 자동으로 선택합니다. 확장을 더 수동으로 제어해야 하는 시나리오의 경우 다음 방법 중 하나를 사용할 수 있습니다.

호스트 헤더

호스트 HTTP 헤더는 TLS에서 SNI 확장과 유사한 함수를 수행합니다. 이를 통해 대상 서버는 단일 IP 주소에서 여러 호스트 이름에 대한 요청을 구분할 수 있습니다. `HttpClient` 는 요청 URI를 사용하여 호스트 헤더를 자동으로 채웁니다. 그러나 해당 값을 수동으로 설정할 수도 있으며 `HttpClient` SNI 확장에서 새 값도 사용합니다. 이 효과를 사용 `HttpRequestMessage.Headers.Host` 하거나 `HttpClient.DefaultRequestHeaders.Host` 사용할 수 있습니다.

C#

```
using HttpClient client = new();

client.DefaultRequestHeaders.Host = "www.microsoft.com";

using var response = await client.GetAsync("https://127.0.0.1:5001/");

System.Console.WriteLine(response);
```

> ⓘ **참고**
>
> 이 메서드를 사용하면 호스트 이름을 사용하여 URL에 연결할 때 SNI를 완전히 보내지 않도록 방지할 수 없습니다. 헤더가 빈 문자열 `HttpClient` 로 설정된 경우 URL의 호스트 이름을 대신 사용합니다.

> ⓘ **참고**

호스트 헤더를 사용자 지정하면 서버 인증서 유효성 검사에 영향을 줍니다. 기본적으로 클라이언트는 서버 인증서가 호스트 헤더의 호스트 이름과 일치해야 합니다.

ConnectCallback을 통한 수동 SslStream 인증

더 복잡하지만 더 강력한 옵션은 `SocketsHttpHandler.ConnectCallback`를 사용하는 것입니다. .NET 7부터 인증된 `SslStream` 항목을 반환하여 TLS 연결 설정 방법을 사용자 지정할 수 있습니다. 콜백 내에서 임의의 `SslClientAuthenticationOptions` 옵션을 사용하여 클라이언트 쪽 인증을 수행할 수 있습니다.

C#

```
var handler = new SocketsHttpHandler
{
    ConnectCallback = async (context, cancellationToken) =>
    {
        var socket = new Socket(SocketType.Stream, ProtocolType.Tcp) { NoDelay =
true };
        try
        {
            await socket.ConnectAsync(context.DnsEndPoint, cancellationToken);

            var sslStream = new SslStream(new NetworkStream(socket, ownsSocket:
true));

            // When using HTTP/2, you must also keep in mind to set options like
ApplicationProtocols
            await sslStream.AuthenticateAsClientAsync(new
SslClientAuthenticationOptions
            {
                TargetHost = context.DnsEndPoint.Host,

            }, cancellationToken);

            return sslStream;
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
};

using HttpClient client = new(handler);

using var response = await client.GetAsync("https://www.microsoft.com");

System.Console.WriteLine(response);
```

System.Net.Http.HttpClient 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스 인스턴스는 HttpClient HTTP 요청을 보내는 세션 역할을 합니다. HttpClient 인스턴스는 해당 인스턴스에서 실행된 모든 요청에 적용되는 설정 컬렉션입니다. 또한 모든 HttpClient 인스턴스는 자체 연결 풀을 사용하여 다른 HttpClient 인스턴스에서 실행된 요청에서 해당 요청을 격리합니다.

인스턴스화 (인스턴스를 생성하거나 사용하는 과정)

HttpClient 는 애플리케이션의 수명 동안 한 번 인스턴스화되고 다시 사용되도록 의도된 것입니다. .NET Core 및 .NET 5 이상 `HttpClient` 에서는 처리기 인스턴스 내의 연결을 풀하고 여러 요청에서 연결을 다시 사용합니다. 모든 요청에 대해 클래스를 `HttpClient` 인스턴스화하면 부하가 많은 상태에서 사용할 수 있는 소켓 수가 소진됩니다. 이 고갈로 인해 오류가 발생합니다 SocketException .

생성자의 일부로 "처리기"를 전달하여 추가 옵션을 구성할 수 있으며, 예를 들어 HttpClientHandler 또는 .NET Core 2.1 이상에서는 SocketsHttpHandler을 사용할 수 있습니다. 요청이 제출된 후에는 처리기의 연결 속성을 변경할 수 없으므로 새 `HttpClient` 인스턴스를 만드는 한 가지 이유는 연결 속성을 변경해야 하는 경우입니다. 각각 다른 요청이 다른 설정을 필요로 하는 경우, 이는 또한 애플리케이션이 여러 HttpClient 인스턴스를 갖게 될 수 있으며, 각 인스턴스는 적절하게 구성되어 있고, 그런 다음 관련 클라이언트에서 요청이 발급됩니다.

`HttpClient`는 연결을 설정할 때 DNS 항목만 확인합니다. DNS 서버에서 지정한 TTL(Time to Live) 기간을 추적하지 않습니다. 일부 컨테이너 시나리오에서 발생할 수 있는 DNS 항목이 정기적으로 변경되는 경우 클라이언트는 이러한 업데이트를 존중하지 않습니다. 이 문제를 해결하려면 연결을 바꿀 때 DNS 조회가 필요하도록 속성을 설정 SocketsHttpHandler.PooledConnectionLifetime 하여 연결의 수명을 제한할 수 있습니다.

C#

```
public class GoodController : ApiController
{
    private static readonly HttpClient httpClient;

    static GoodController()
    {
        var socketsHandler = new SocketsHttpHandler
        {
            PooledConnectionLifetime = TimeSpan.FromMinutes(2)
```

```
        };

        httpClient = new HttpClient(socketsHandler);
    }
}
```

하나의 `HttpClient` 인스턴스를 생성하는 대신, IHttpClientFactory을 사용하여 `HttpClient` 인스턴스를 관리할 수도 있습니다. 자세한 내용은 HttpClient 사용 지침을 참조하세요.

파생

또한 더 HttpClient 구체적인 HTTP 클라이언트에 대한 기본 클래스로도 작동합니다. 예를 들어, Facebook 웹 서비스에 특정한 추가 메서드를 제공하는 `FacebookHttpClient`가 있을 수 있습니다(예: `GetFriends` 메서드). 파생 클래스는 클래스의 가상 메서드를 재정의해서는 안 됩니다. 대신 사전 요청 또는 사후 요청 처리를 구성하는 데 허용하는 HttpMessageHandler 생성자 오버로드를 사용합니다.

운송

이 HttpClient API는 실행되는 각 플랫폼에서 사용할 수 있는 하위 수준 기능을 래핑하는 상위 수준 API입니다.

각 플랫폼에서 HttpClient 사용 가능한 최상의 전송을 사용하려고 합니다.

테이블 확장

호스트/런타임	백 엔드
Windows/.NET Framework	HttpWebRequest
Windows/Mono	HttpWebRequest
Windows/UWP	Windows 네이티브 WinHttpHandler (HTTP 2.0 지원)
Windows/.NET Core 1.0-2.0	Windows 네이티브 WinHttpHandler (HTTP 2.0 지원)
macOS/Mono	HttpWebRequest
macOS/.NET Core 1.0-2.0	`libcurl` 기반 HTTP 전송 프로토콜(HTTP 2.0 지원)
Linux/Mono	HttpWebRequest
Linux/.NET Core 1.0-2.0	`libcurl` 기반 HTTP 전송 프로토콜(HTTP 2.0 지원)
.NET Core 2.1 이상	System.Net.Http.SocketsHttpHandler

사용자는 HttpClient을(를) 받는 HttpClient 생성자를 호출해서 특정 전송 HttpMessageHandler을 구성할 수도 있습니다.

.NET Framework 및 Mono

기본적으로 .NET Framework 및 Mono HttpWebRequest 는 서버에 요청을 보내는 데 사용됩니다. 이 동작은 HttpMessageHandler 매개 변수를 사용하는 생성자 오버로드 중 하나에서 다른 처리기를 지정하여 수정할 수 있습니다. 인증 또는 캐싱과 같은 기능이 필요한 경우 설정을 구성하는 데 사용할 WebRequestHandler 수 있으며 인스턴스를 생성자에 전달할 수 있습니다. 반환된 처리기를 HttpMessageHandler 매개 변수를 가진 생성자 오버로드에 전달할 수 있습니다.

.NET 코어

.NET Core 2.1 System.Net.Http.SocketsHttpHandler 부터 클래스는 HttpClientHandler 같은 상위 수준의 HTTP 네트워킹 클래스에서 사용하는 구현을 HttpClient대신 제공합니다. 사용 SocketsHttpHandler 은 다음과 같은 여러 가지 이점을 제공합니다.

- 이전 구현보다 월등히 향상된 성능.
- 배포 및 서비스를 간소화하는 플랫폼 종속성을 제거합니다. 예를 들어 `libcurl` 더 이상 macOS용 .NET Core 및 Linux용 .NET Core에 대한 종속성이 없습니다.
- 모든 .NET 플랫폼에서 일관된 동작.

이 변경이 바람직하지 않은 경우 Windows에서는 WinHttpHandler 참조하고 HttpClient의 생성자에 수동으로 전달하여 계속 사용할 수 있습니다.

런타임 구성 옵션을 사용하여 동작 구성

동작의 HttpClient특정 측면은 런타임 구성 옵션을 통해 사용자 지정할 수 있습니다. 그러나 이러한 스위치의 동작은 .NET 버전마다 다릅니다. 예를 들어 .NET Core 2.1 - 3.1에서 기본적으로 사용되는지 여부를 SocketsHttpHandler 구성할 수 있지만 해당 옵션은 .NET 5부터 더 이상 사용할 수 없습니다.

연결 풀링 (Connection Pooling)

`HttpClient` 는 가능한 경우 HTTP 연결을 풀로 만들고 둘 이상의 요청에 사용합니다. 연결 핸드셰이크가 한 번만 수행되므로 이는 특히 HTTPS 요청의 경우 상당한 성능 이점을 얻을 수 있습니다.

연결 풀 속성은 HttpClientHandler 또는 SocketsHttpHandler에 구성될 수 있으며, 생성 중에 전달되는 MaxConnectionsPerServer, PooledConnectionIdleTimeout, 및 PooledConnectionLifetime를 포함합니다.

인스턴스를 삭제하면 `HttpClient` 열려 있는 연결이 닫히고 보류 중인 요청이 취소됩니다.

> ⓘ **참고 항목**
>
> 동일한 서버에 HTTP/1.1 요청을 동시에 보내는 경우 새 연결을 만들 수 있습니다. 인스턴스를 `HttpClient` 다시 사용하는 경우에도 요청 속도가 높거나 방화벽 제한 사항이 있는 경우 기본 TCP 정리 타이머로 인해 사용 가능한 소켓을 소모할 수 있습니다. 동시 연결 수를 제한하려면 속성을 설정할 `MaxConnectionsPerServer` 수 있습니다. 기본적으로 동시 HTTP/1.1 연결 수는 무제한입니다.

버퍼링 및 요청 수명

기본적으로 `HttpClient` 메서드(제외 GetStreamAsync)는 비동기 결과를 반환하기 전에 모든 응답 본문을 메모리로 읽어 서버의 응답을 버퍼링합니다. 이러한 요청은 다음 중 하나가 발생할 때까지 계속됩니다.

- Task<TResult>이 성공하여 결과를 반환합니다.
- 경우 Timeout에 이르면 Task<TResult> 취소될 것입니다.
- CancellationToken 일부 메서드 오버로드가 실행됩니다.
- CancelPendingRequests()을 호출합니다.
- HttpClient가 삭제됩니다.

일부 메서드 오버로드에서 사용할 수 있는 매개 변수를 사용하여 HttpCompletionOption 요청별로 버퍼링 동작을 변경할 수 있습니다. 이 인수는 응답 헤더만 읽은 후 또는 응답 콘텐츠를 읽고 버퍼링한 후에 완료로 간주되어야 하는지 여부를 Task<TResult> 지정하는 데 사용할 수 있습니다.

네임스페이스의 HttpClient 사용 및 관련 클래스를 사용하는 System.Net.Http 앱이 대량의 데이터(50MB 이상)를 다운로드하려는 경우 앱은 해당 다운로드를 스트리밍하고 기본 버퍼링을 사용하지 않아야 합니다. 기본 버퍼링을 사용하는 경우 클라이언트 메모리 사용량이 매우 커져 성능이 크게 저하될 수 있습니다.

스레드 안전성

다음 메서드는 스레드로부터 안전합니다.

- CancelPendingRequests
- DeleteAsync
- GetAsync
- GetByteArrayAsync

- GetStreamAsync
- GetStringAsync
- PostAsync
- PutAsync
- SendAsync

프록시

기본적으로 `HttpClient` 플랫폼에 따라 환경 변수 또는 사용자/시스템 설정에서 프록시 구성을 읽습니다. 이 동작을 변경하려면 먼저 WebProxy 또는 IWebProxy을 우선 순위에 따라 전달하십시오.

- Proxy 생성 중에 전달된 `HttpClientHandler`의 `HttpClient` 속성
- DefaultProxy 정적 속성(모든 인스턴스에 영향을 줍니다.)

를 사용하여 UseProxy프록시를 사용하지 않도록 설정할 수 있습니다. Windows 사용자의 기본 설정은 네트워크 검색 기능을 사용하여 프록시를 자동으로 감지하는 것이며, 이는 속도가 느려질 수 있습니다. 프록시가 필요하지 않다고 알려진 높은 처리량 애플리케이션의 경우 프록시를 사용하지 않도록 설정해야 합니다.

프록시 설정(예: Credentials)은 `HttpClient`첫 번째 요청이 사용되기 전에 변경해야 합니다. 처음 사용한 `HttpClient` 후의 변경 내용은 후속 요청에 반영되지 않을 수 있습니다.

타임아웃

Timeout를 사용하여 `HttpClient` 인스턴스의 모든 HTTP 요청에 대한 기본 시간 제한을 설정할 수 있습니다. 시간 제한은 요청/응답을 시작하는 xxxAsync 메서드에만 적용됩니다. 시간 제한에 도달 Task<TResult> 하면 해당 요청에 대한 요청이 취소됩니다.

SocketsHttpHandler 인스턴스를 전달하여 `HttpClient` 개체를 생성할 때 몇 가지 추가 시간 제한을 설정할 수 있습니다.

테이블 확장

재산	설명
ConnectTimeout	요청에 새 TCP 연결을 만들어야 하는 경우 사용되는 시간 제한을 지정합니다. 시간 제한이 발생하면 요청 Task<TResult> 이 취소됩니다.
PooledConnectionLifetime	연결 풀의 각 연결에 사용할 시간 제한을 지정합니다. 연결이 유휴 상태이면 연결이 즉시 닫힙니다. 그렇지 않으면 현재 요청이 끝날 때 연결이 닫힙니다.

재산	설명
PooledConnectionIdleTimeout	연결 풀의 연결이 오랫동안 유휴 상태이면 연결이 닫힙니다.
Expect100ContinueTimeout	요청에 "Expect: 100-continue" 헤더가 있는 경우 시간 제한까지 또는 "100-continue" 응답이 수신될 때까지 콘텐츠 전송이 지연됩니다.

`HttpClient` 만 연결을 만들 때 DNS 항목을 확인합니다. DNS 서버에서 지정한 TTL(Time to Live) 기간을 추적하지 않습니다. 일부 컨테이너 시나리오에서 발생할 수 있는 DNS 항목이 정기적으로 변경되는 경우 연결을 바꿀 때 DNS 조회가 필요하도록 연결의 수명을 제한하는 데 사용할 PooledConnectionLifetime 수 있습니다.

Last updated on 2026. 02. 12.

System.Net.Http.HttpClientHandler 클래스

2025. 06. 22.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

파생된 클래스 및 클래스를 통해 개발자는 HttpClientHandler 프록시에서 인증에 이르는 다양한 옵션을 구성할 수 있습니다.

.NET Core의 HttpClientHandler

.NET Core 2.1부터 클래스의 `HttpClientHandler` 구현은 클래스에서 사용하는 System.Net.Http.SocketsHttpHandler 플랫폼 간 HTTP 프로토콜 스택에 따라 변경되었습니다. .NET Core 2.1 이전에는 `HttpClientHandler` 클래스가 오래된 HTTP 프로토콜 스택을 사용했습니다. Windows에서는 WinHttpHandler를, Linux에서는 네이티브 `CurlHandler` 구성 요소 위에 구현된 내부 클래스인 `libcurl`를 사용했습니다.

System.Net.HttpListener 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스를 HttpListener 사용하여 HTTP 요청에 응답하는 간단한 HTTP 프로토콜 수신기를 만들 수 있습니다. 수신기는 개체의 HttpListener 수명 동안 활성 상태이며 해당 사용 권한으로 애플리케이션 내에서 실행됩니다.

HttpListener을 사용하려면 HttpListener 생성자를 사용하여 클래스의 새 인스턴스를 만들고, Prefixes 속성을 사용하여 HttpListener가 처리해야 할 URI(Uniform Resource Identifier) 접두사를 지정하는 문자열을 포함하는 컬렉션에 액세스합니다.

URI 접두사 문자열은 스키마(http 또는 https), 호스트, 선택적 포트 및 선택적 경로로 구성됩니다. 전체 접두사 문자열의 예는 .입니다 `http://www.contoso.com:8080/customerData/`. 접두사는 슬래시("/")로 끝나야 합니다. HttpListener 요청된 URI와 가장 밀접하게 일치하는 접두사를 가진 개체가 요청에 응답합니다. 여러 HttpListener 개체가 동일한 접두사를 추가할 수 없습니다. 이미 사용 중인 접두사를 추가하면 Win32Exception 예외가 발생합니다.

포트를 지정하면 요청된 URI가 다른 접두사와 일치하지 않는 경우 호스트 요소를 "*"로 바꿔 포트에 전송된 요청을 수락함을 나타낼 HttpListener 수 있습니다. 예를 들어 요청된 URI가 처리 HttpListener되지 않을 때 포트 8080으로 전송된 모든 요청을 받으려면 접두사는 *http://*:8080/* 입니다. 마찬가지로 포트에 전송된 HttpListener 모든 요청을 수락하도록 지정하려면 호스트 요소를 "+" 문자로 바꿉니다. 예: *https://+:8080*. "*" 및 "+" 문자는 경로를 포함하는 접두사에 있을 수 있습니다.

URI 접두사에서 관리하는 HttpListener 개체는 와일드카드 하위 도메인을 지원합니다. 와일드카드 하위 도메인을 지정하려면 URI 접두사에서 호스트 이름의 일부로 "*" 문자를 사용합니다. 예 : *http://*.foo.com/*. 메서드의 인수로 Add를 전달합니다.

> ⚠ **Warning**
>
> 최상위 와일드카드 바인딩(*http://*:8080/* 및 *http://+:8080*)을 **사용하면 안 됩니다** . 최상위 와일드카드 바인딩은 보안 취약점에 앱을 노출시킬 수 있습니다. 강력한 와일드카드와 약한 와일드카드 모두에 적용됩니다. 와일드카드보다는 명시적 호스트 이름을 사용합니다. 전체 부모 도메인을 제어하는 경우 하위 도메인 와일드카드 바인딩(예: `*.mysub.com`)에는 이러한 보안 위험이 없습니다(취약한 `*.com`과 반대임). 자세한 내용은 **rfc7230 섹션-5.4** 를 참조하세요.

클라이언트의 요청 수신 대기를 시작하려면 컬렉션에 URI 접두사를 추가하고 메서드를 호출합니다 Start . HttpListener 는 클라이언트 요청을 처리하기 위한 동기 모델과 비동기 모델을 모두 제공합니다. HttpListenerContext 메서드 또는 비동기 대응 메서드인 GetContext 및 BeginGetContext 메서드에 의해 반환된 EndGetContext 개체를 사용하여 요청 및 관련 응답에 액세스합니다.

동기 모델은 클라이언트 요청을 기다리는 동안 애플리케이션이 차단되어야 하고 한 번에 하나의 요청만 처리하려는 경우에 적합합니다. 동기 모델을 사용하여 클라이언트가 GetContext 요청을 보낼 때까지 기다리는 메서드를 호출합니다. 이 메서드는 어떤 일이 발생하면 처리를 위해 HttpListenerContext 개체를 반환합니다.

더 복잡한 비동기 모델에서는 요청을 기다리는 동안 애플리케이션이 차단되지 않으며 각 요청은 자체 실행 스레드에서 처리됩니다. 들어오는 BeginGetContext 각 요청에 대해 호출할 애플리케이션 정의 메서드를 지정하려면 이 메서드를 사용합니다. 해당 메서드 내에서 메서드를 EndGetContext 호출하여 요청을 가져오고, 처리하고, 응답합니다.

두 모델에서 들어오는 요청은 HttpListenerContext.Request 속성을 사용하여 접근되며 HttpListenerRequest 개체로 표현됩니다. 마찬가지로 응답은 HttpListenerContext.Response 속성을 통해 액세스되며, HttpListenerResponse 객체로 표시됩니다. 이러한 개체는 일부 기능을 HttpWebRequest 및 HttpWebResponse 개체와 공유하지만, 후자의 개체는 클라이언트 동작을 구현하므로 HttpListener와 함께 사용할 수 없습니다.

HttpListener은 클라이언트 인증을 요구할 수 있습니다. 인증에 사용할 특정 체계를 지정하거나 사용할 스키마를 결정하는 대리자를 지정할 수 있습니다. 클라이언트의 ID에 대한 정보를 얻으려면 어떤 형태의 인증이 필요합니다. 자세한 내용은 User, AuthenticationSchemes, 및 AuthenticationSchemeSelectorDelegate 속성을 참조하세요.

> ⓘ **참고 항목**
>
> https를 사용하여 **HttpListener**를 만들 경우, 해당 수신기에 대해 서버 인증서를 선택해야 합니다. 그렇지 않으면 연결이 예기치 않게 닫히면 이에 **HttpListener** 대한 요청이 실패합니다.

> ⓘ **참고 항목**
>
> 네트워크 셸(netsh.exe)을 사용하여 서버 인증서 및 기타 수신기 옵션을 구성할 수 있습니다. 자세한 내용은 **네트워크 셸(Netsh)** 을 참조하세요. 실행 파일은 Windows Server 2008 및 Windows Vista와 함께 배송되기 시작했습니다.

> ⓘ **참고 항목**

여러 인증 체계를 HttpListener지정하는 경우 수신기는 클라이언트에 다음 순서 `Negotiate NTLM Digest Basic`로 이의를 제기합니다.

HTTP.sys

클래스는 Windows의 모든 HTTP 트래픽을 처리하는 커널 모드 수신기인 HttpListener 위에 구축됩니다. `HTTP.sys` 는 연결 관리, 대역폭 제한 및 웹 서버 로깅을 제공합니다. HttpCfg.exe 도구를 사용하여 SSL 인증서를 추가합니다.

.NET 11부터 Windows `HTTP.sys` 구현은 HttpListener 커널 수준 응답 버퍼링을 사용하도록 설정합니다. 사용하도록 설정하면 클라이언트로 `HTTP.sys` 전송되기 전에 응답 데이터가 버퍼링되므로 대기 시간이 긴 연결에 대한 처리량을 향상시킬 수 있습니다. 이 설정은 다음과 같이 메서드를 AppContext.SetSwitch 호출하여 사용할 수 있습니다.

C#

```
AppContext.SetSwitch("System.Net.HttpListener.EnableKernelResponseBuffering",
true);
```

Last updated on 2026. 03. 16.

.NET의 소켓

네임스페이스 System.Net.Sockets에는 관리형 플랫폼 간의 소켓 네트워킹 구현이 포함됩니다. 네임스페이스의 System.Net 다른 모든 네트워크 액세스 클래스는 이 소켓 구현을 기반으로 빌드됩니다.

클래스는 Socket Linux, macOS 또는 Windows와의 네이티브 상호 운용성에 의존하여 제공되는 소켓 서비스의 관리 코드 버전입니다. 대부분의 경우 클래스 메서드는 데이터를 네이티브 메서드로 변환하여 마샬링하고 필요한 보안 검사를 처리합니다.

클래스는 `Socket` 동기 모드와 비동기 모드의 두 가지 기본 모드를 지원합니다. 동기 모드에서 네트워크 작업(예: Send 및 Receive)을 수행하는 함수에 대한 호출은 작업이 완료될 때까지 기다렸다가 제어를 호출 프로그램에 반환합니다. 비동기 모드에서 이러한 호출은 즉시 반환됩니다.

참고하십시오

- 소켓을 사용하여 데이터 보내기 및 받기
- .NET의 네트워킹
- System.Net.Sockets
- Socket

Last updated on 2026. 02. 24.

소켓을 사용하여 TCP를 통해 데이터 보내기 및 받기

소켓을 사용하여 원격 디바이스와 통신하려면 먼저 프로토콜 및 네트워크 주소 정보를 사용하여 소켓을 초기화해야 합니다. 클래스의 Socket 생성자에는 소켓이 연결을 만드는 데 사용하는 주소 패밀리, 소켓 유형 및 프로토콜 형식을 지정하는 매개 변수가 있습니다. 클라이언트 소켓을 서버 소켓에 연결할 때 클라이언트는 개체를 `IPEndPoint` 사용하여 서버의 네트워크 주소를 지정합니다.

IP 엔드포인트 만들기

System.Net.Sockets로 작업할 때 네트워크 엔드포인트를 IPEndPoint 개체로 나타냅니다. `IPEndPoint`는 IPAddress 및 해당 포트 번호로 생성됩니다. Socket을 통해 대화를 시작하려면 먼저 앱과 원격 대상 간에 데이터 파이프를 만듭니다.

TCP/IP는 네트워크 주소와 서비스 포트 번호를 사용하여 서비스를 고유하게 식별합니다. 네트워크 주소는 특정 네트워크 대상을 식별하고, 포트 번호는 연결할 해당 디바이스의 특정 서비스를 식별합니다. 네트워크 주소와 서비스 포트의 조합을 엔드포인트가라고 하며, .NET에서는 EndPoint 클래스로 표현됩니다. `EndPoint`의 하위 항목이 지원되는 각 주소 패밀리에 대해 정의되고, IP 주소 패밀리에 대한 클래스는 IPEndPoint입니다.

Dns 클래스는 TCP/IP 인터넷 서비스를 사용하는 앱에 도메인 이름 서비스를 제공합니다. GetHostEntryAsync 메서드는 DNS 서버를 쿼리하여 친숙한 도메인 이름(예: "host.contoso.com")을 숫자 인터넷 주소(예: `192.168.1.1`)에 매핑합니다. `GetHostEntryAsync`는 대기 시 요청된 이름에 대한 주소 및 별칭 목록이 들어 있는 `Task<IPHostEntry>`를 반환합니다. 대부분의 경우 AddressList 배열에 반환된 첫 번째 주소를 사용할 수 있습니다. 다음 코드에서는 IPAddress 서버의 IP 주소가 포함된 `host.contoso.com`를 가져옵니다.

C#

```
IPHostEntry ipHostInfo = await Dns.GetHostEntryAsync("host.contoso.com");
IPAddress ipAddress = ipHostInfo.AddressList[0];
```

팁

수동 테스트 및 디버깅 목적으로 일반적으로 GetHostEntryAsync 값의 결과 호스트 이름과 함께 Dns.GetHostName() 메서드를 사용하여 로컬 호스트 이름을 IP 주소로 확인할 수 있습니다. 다음 코드 조각을 살펴봅니다.

C#

```
var hostName = Dns.GetHostName();
IPHostEntry localhost = await Dns.GetHostEntryAsync(hostName);
// This is the IP address of the local machine
IPAddress localIpAddress = localhost.AddressList[0];
```

IANA(Internet Assigned Numbers Authority)는 공통 서비스의 포트 번호를 정의합니다. 자세한 내용은 IANA: 서비스 이름 및 전송 프로토콜 포트 번호 레지스트리를 참조하세요. 다른 서비스에는 1,024 ~ 65,535 범위의 등록된 포트 번호가 있을 수 있습니다. 다음 코드에서는 `host.contoso.com`의 IP 주소를 포트 번호와 결합하여 연결에 대한 원격 엔드포인트를 만듭니다.

C#

```
IPEndPoint ipEndPoint = new(ipAddress, 11_000);
```

원격 디바이스의 주소를 결정하고 연결에 사용할 포트를 선택하면 앱이 원격 디바이스에 대한 연결을 설정할 수 있습니다.

`Socket` 클라이언트 만들기

개체를 `endPoint` 만든 상태에서 서버에 연결할 클라이언트 소켓을 만듭니다. 소켓이 연결되면 서버 소켓 연결에서 데이터를 보내고 받을 수 있습니다.

C#

```
using Socket client = new(
    ipEndPoint.AddressFamily,
    SocketType.Stream,
    ProtocolType.Tcp);

await client.ConnectAsync(ipEndPoint);
while (true)
{
    // Send message.
    var message = "Hi friends 👋!<|EOM|>";
    var messageBytes = Encoding.UTF8.GetBytes(message);
    _ = await client.SendAsync(messageBytes, SocketFlags.None);
    Console.WriteLine($"Socket client sent message: \"{message}\"");

    // Receive ack.
    var buffer = new byte[1_024];
    var received = await client.ReceiveAsync(buffer, SocketFlags.None);
    var response = Encoding.UTF8.GetString(buffer, 0, received);
    if (response == "<|ACK|>")
    {
        Console.WriteLine(
            $"Socket client received acknowledgment: \"{response}\"");
        break;
```

```
    }
    // Sample output:
    //     Socket client sent message: "Hi friends 👋!<|EOM|>"
    //     Socket client received acknowledgment: "<|ACK|>"
}

client.Shutdown(SocketShutdown.Both);
```

앞의 C# 코드는 다음과 같습니다.

- 주어진 `Socket` 인스턴스 주소 패밀리, `endPoint`, 및 SocketType.Stream를 사용하여 새 ProtocolType.Tcp 객체를 인스턴스화합니다.
- 인스턴스를 Socket.ConnectAsync 사용하여 메서드를 `endPoint` 인수로 호출합니다.
- 루프에서 `while` :
  - 를 사용하여 Socket.SendAsync메시지를 인코딩하고 서버로 보냅니다.
  - 보낸 메시지를 콘솔에 씁니다.
  - 를 사용하여 Socket.ReceiveAsync서버에서 데이터를 수신하도록 버퍼를 초기화합니다.
  - `response` 승인이면 콘솔에 기록되고 루프가 종료됩니다.
- 마지막으로 `client`을 제공하여 지정된 Socket.Shutdown 소켓이 SocketShutdown.Both을 호출하며, 이는 송신 및 수신 작업을 모두 종료시킵니다.

`Socket` 서버 만들기

서버 소켓을 만들려면 개체가 `endPoint` IP 주소에서 들어오는 연결을 수신 대기할 수 있지만 포트 번호를 지정해야 합니다. 소켓이 만들어지면 서버는 들어오는 연결을 수락하고 클라이언트와 통신할 수 있습니다.

C#

```
using Socket listener = new(
    ipEndPoint.AddressFamily,
    SocketType.Stream,
    ProtocolType.Tcp);

listener.Bind(ipEndPoint);
listener.Listen(100);

var handler = await listener.AcceptAsync();
while (true)
{
    // Receive message.
    var buffer = new byte[1_024];
    var received = await handler.ReceiveAsync(buffer, SocketFlags.None);
    var response = Encoding.UTF8.GetString(buffer, 0, received);
```

```
    var eom = "<|EOM|>";
    if (response.IndexOf(eom) > -1 /* is end of message */)
    {
        Console.WriteLine(
            $"Socket server received message: \"{response.Replace(eom, "")}\"");

        var ackMessage = "<|ACK|>";
        var echoBytes = Encoding.UTF8.GetBytes(ackMessage);
        await handler.SendAsync(echoBytes, 0);
        Console.WriteLine(
            $"Socket server sent acknowledgment: \"{ackMessage}\"");

        break;
    }
    // Sample output:
    //    Socket server received message: "Hi friends 👋!"
    //    Socket server sent acknowledgment: "<|ACK|>"
}
```

앞의 C# 코드는 다음과 같습니다.

- 주어진 `Socket` 인스턴스 주소 패밀리, `endPoint`, 및 SocketType.Stream를 사용하여 새 ProtocolType.Tcp 객체를 인스턴스화합니다.
- 메서드 `listener` 를 Socket.Bind 인스턴스와 함께 `endPoint` 인수로 호출하여 소켓을 네트워크 주소와 연결합니다.
- 들어오는 Socket.Listen() 연결을 수신 대기하기 위해 메서드가 호출됩니다.
- `listener`는 Socket.AcceptAsync 소켓에서 들어오는 연결을 수락하기 위해 `handler` 메서드를 호출합니다.
- 루프에서 `while` :
  - 클라이언트에서 데이터를 수신하기 위한 호출 Socket.ReceiveAsync 입니다.
  - 데이터를 받으면 디코딩되어 콘솔에 기록됩니다.
  - `response` 메시지가 `<|EOM|>`로 끝나면 Socket.SendAsync를 사용하여 클라이언트에게 확인 응답이 전송됩니다.

샘플 클라이언트 및 서버 실행

먼저 서버 애플리케이션을 시작한 다음 클라이언트 애플리케이션을 시작합니다.

.NET CLI

```
dotnet run --project socket-server
Socket server starting...
Found: 172.23.64.1 available on port 9000.
Socket server received message: "Hi friends 👋!"
```

```
Socket server sent acknowledgment: "<|ACK|>"
Press ENTER to continue...
```

클라이언트 애플리케이션은 서버에 메시지를 보내고 서버는 승인으로 응답합니다.

.NET CLI

```
dotnet run --project socket-client
Socket client starting...
Found: 172.23.64.1 available on port 9000.
Socket client sent message: "Hi friends 👋!<|EOM|>"
Socket client received acknowledgment: "<|ACK|>"
Press ENTER to continue...
```

참고하십시오

- .NET에서의 소켓
- .NET의 네트워킹
- System.Net.Sockets
- Socket

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

TCP 개요

> **ⓘ Important**
>
> 고급 사용자에게는 Socket 및 `TcpClient` 대신 `TcpListener` 클래스가 매우 권장됩니다.

TCP(Transmission Control Protocol)를 사용하려면 두 가지 옵션이 있습니다. 하나는 제어 및 성능을 극대화하기 위해 Socket을 사용하는 것이고, 다른 하나는 TcpClient 및 TcpListener 도우미 클래스를 사용하는 것입니다. TcpClient 및 TcpListener는 System.Net.Sockets.Socket 클래스 위에 빌드되며, 사용 편의를 위해 데이터 전송 관련 세부 정보를 처리합니다.

프로토콜 클래스는 기본 `Socket` 클래스를 사용하여 상태 정보를 유지 관리하거나 프로토콜 관련 소켓 설정의 세부 정보를 알아야 하는 오버헤드 없이 네트워크 서비스에 대한 쉬운 액세스를 제공합니다. 비동기 `Socket` 메서드를 사용하려면 NetworkStream 클래스에서 제공하는 비동기 메서드를 사용할 수 있습니다. 프로토콜 클래스에 의해 노출되지 않는 `Socket` 클래스의 기능에 액세스하려면 `Socket` 클래스를 사용해야 합니다.

`TcpClient` 및 `TcpListener`는 `NetworkStream` 클래스를 사용하여 네트워크를 나타냅니다. GetStream 메서드를 사용하여 네트워크 스트림을 반환한 다음 스트림의 NetworkStream.ReadAsync 및 NetworkStream.WriteAsync 메서드를 호출합니다. `NetworkStream`은 프로토콜 클래스의 기본 소켓을 소유하지 않으므로 닫아도 소켓에 영향을 주지 않습니다.

`TcpClient` 및 `TcpListener` 사용

TcpClient 클래스는 TCP를 사용하여 인터넷 리소스의 데이터를 요청합니다. `TcpClient`의 메서드 및 속성은 TCP를 사용하여 데이터를 요청 및 수신하는 Socket을 만들기 위한 세부 정보를 추상화합니다. 원격 디바이스에 대한 연결은 스트림으로 표현되므로 .NET Framework 스트림 처리 기법을 사용하여 데이터를 읽고 쓸 수 있습니다.

TCP 프로토콜은 원격 엔드포인트에 연결한 후 해당 연결을 사용하여 데이터 패킷을 주고받습니다. TCP는 데이터 패킷이 엔드포인트로 전송되고 도착 시 올바른 순서로 어셈블되도록 합니다.

IP 엔드포인트 만들기

System.Net.Sockets로 작업할 때 네트워크 엔드포인트를 IPEndPoint 개체로 나타냅니다. `IPEndPoint`는 IPAddress 및 해당 포트 번호로 생성됩니다. Socket을 통해 대화를 시작하려면 먼저 앱과 원격 대상 간에 데이터 파이프를 만듭니다.

TCP/IP는 네트워크 주소와 서비스 포트 번호를 사용하여 서비스를 고유하게 식별합니다. 네트워크 주소는 특정 네트워크 대상을 식별하고, 포트 번호는 연결할 해당 디바이스의 특정 서비스를

식별합니다. 네트워크 주소와 서비스 포트의 조합을 엔드포인트가라고 하며, .NET에서는 EndPoint 클래스로 표현됩니다. `EndPoint`의 하위 항목이 지원되는 각 주소 패밀리에 대해 정의되고, IP 주소 패밀리에 대한 클래스는 IPEndPoint입니다.

Dns 클래스는 TCP/IP 인터넷 서비스를 사용하는 앱에 도메인 이름 서비스를 제공합니다. GetHostEntryAsync 메서드는 DNS 서버를 쿼리하여 친숙한 도메인 이름(예: "host.contoso.com")을 숫자 인터넷 주소(예: `192.168.1.1`)에 매핑합니다. `GetHostEntryAsync`는 대기 시 요청된 이름에 대한 주소 및 별칭 목록이 들어 있는 `Task<IPHostEntry>`를 반환합니다. 대부분의 경우 AddressList 배열에 반환된 첫 번째 주소를 사용할 수 있습니다. 다음 코드에서는 IPAddress 서버의 IP 주소가 포함된 `host.contoso.com`를 가져옵니다.

C#

```
IPHostEntry ipHostInfo = await Dns.GetHostEntryAsync("host.contoso.com");
IPAddress ipAddress = ipHostInfo.AddressList[0];
```

팁

수동 테스트 및 디버깅 목적으로 일반적으로 GetHostEntryAsync 값의 결과 호스트 이름과 함께 Dns.GetHostName() 메서드를 사용하여 로컬 호스트 이름을 IP 주소로 확인할 수 있습니다. 다음 코드 조각을 살펴봅니다.

C#

```
var hostName = Dns.GetHostName();
IPHostEntry localhost = await Dns.GetHostEntryAsync(hostName);
// This is the IP address of the local machine
IPAddress localIpAddress = localhost.AddressList[0];
```

IANA(Internet Assigned Numbers Authority)는 공통 서비스의 포트 번호를 정의합니다. 자세한 내용은 IANA: 서비스 이름 및 전송 프로토콜 포트 번호 레지스트리를 참조하세요. 다른 서비스에는 1,024 ~ 65,535 범위의 등록된 포트 번호가 있을 수 있습니다. 다음 코드에서는 `host.contoso.com`의 IP 주소를 포트 번호와 결합하여 연결에 대한 원격 엔드포인트를 만듭니다.

C#

```
IPEndPoint ipEndPoint = new(ipAddress, 11_000);
```

원격 디바이스의 주소를 결정하고 연결에 사용할 포트를 선택하면 앱이 원격 디바이스에 대한 연결을 설정할 수 있습니다.

인증 요청을 처리하는 데 사용하는 `TcpClient`

`TcpClient` 클래스는 `Socket` 클래스보다 높은 추상화 수준에서 TCP 서비스를 제공합니다. `TcpClient`는 원격 호스트에 대한 클라이언트 연결을 만드는 데 사용됩니다. `IPEndPoint`를 가져오는 방법을 알고 있으므로 원하는 포트 번호와 페어링할 `IPAddress`가 있다고 가정해 보겠습니다. 다음 예제에서는 TCP 포트 13에서 시간 서버에 연결하도록 `TcpClient`를 설정하는 방법을 보여 줍니다.

C#

```
var ipEndPoint = new IPEndPoint(ipAddress, 13);

using TcpClient client = new();
await client.ConnectAsync(ipEndPoint);
await using NetworkStream stream = client.GetStream();

var buffer = new byte[1_024];
int received = await stream.ReadAsync(buffer);

var message = Encoding.UTF8.GetString(buffer, 0, received);
Console.WriteLine($"Message received: \"{message}\"");
// Sample output:
//     Message received: "📅 8/22/2022 9:07:17 AM 🕛"
```

위의 C# 코드에서:

- 알려진 `IPEndPoint` 및 포트에서 `IPAddress`를 만듭니다.
- 새 `TcpClient` 개체를 인스턴스화합니다.
- `client`를 사용하여 포트 13의 원격 TCP 시간 서버에 TcpClient.ConnectAsync를 연결합니다.
- NetworkStream을 사용하여 원격 호스트에서 데이터를 읽습니다.
- `1_024` 바이트의 읽기 버퍼를 선언합니다.
- `stream`에서 읽기 버퍼로 데이터를 읽습니다.
- 결과를 콘솔에 문자열로 씁니다.

클라이언트는 메시지가 작다는 것을 알고 있으므로 하나의 작업으로 전체 메시지를 읽기 버퍼로 읽을 수 있습니다. 메시지 크기가 크거나 길이가 확정되지 않은 메시지의 경우 클라이언트는 버퍼를 더 적절하게 사용하고 `while` 루프에서 읽어야 합니다.

> **ⓘ Important**
>
> 메시지를 보내고 받을 때 Encoding은 서버와 클라이언트 모두에 미리 알려야 합니다. 예를 들어 서버가 ASCIIEncoding을 사용하여 통신하지만 클라이언트가 UTF8Encoding을 사용하려고 하면 메시지 형식이 잘못됩니다.

인증 요청을 처리하는 데 사용하는 `TcpListener`

TcpListener는 TCP 포트에서 들어오는 요청을 모니터링한 다음, 클라이언트에 대한 연결을 관리하는 `Socket` 또는 `TcpClient`를 만드는 데 사용됩니다. Start 메서드는 수신 대기를 사용하도록 설정하고, Stop 메서드는 포트에서 수신 대기를 사용하지 않도록 설정합니다. AcceptTcpClientAsync 메서드는 들어오는 연결 요청을 허용하고 `TcpClient`를 만들어 요청을 처리하며, AcceptSocketAsync 메서드는 들어오는 연결 요청을 허용하고 `Socket`을 만들어 요청을 처리합니다.

다음 예제에서는 `TcpListener`를 사용해서 네트워크 시간 서버를 만들어 TCP 포트 13을 모니터링하는 방법을 보여 줍니다. 들어오는 연결 요청이 허용되면 시간 서버가 호스트 서버의 현재 날짜 및 시간을 사용하여 응답합니다.

C#

```
var ipEndPoint = new IPEndPoint(IPAddress.Any, 13);
TcpListener listener = new(ipEndPoint);

try
{
    listener.Start();

    using TcpClient handler = await listener.AcceptTcpClientAsync();
    await using NetworkStream stream = handler.GetStream();

    var message = $"📅 {DateTime.Now} 🕛";
    var dateTimeBytes = Encoding.UTF8.GetBytes(message);
    await stream.WriteAsync(dateTimeBytes);

    Console.WriteLine($"Sent message: \"{message}\"");
    // Sample output:
    //     Sent message: "📅 8/22/2022 9:07:17 AM 🕛"
}
finally
{
    listener.Stop();
}
```

위의 C# 코드에서:

- `IPEndPoint` 및 포트를 사용하여 IPAddress.Any를 만듭니다.
- 새 `TcpListener` 개체를 인스턴스화합니다.
- Start 메서드를 호출하여 포트에서 수신 대기를 시작합니다.
- `TcpClient` 메서드의 AcceptTcpClientAsync를 사용하여 들어오는 연결 요청을 수락합니다.
- 현재 날짜 및 시간을 문자열 메시지로 인코딩합니다.
- NetworkStream을 사용하여 연결된 클라이언트에 데이터를 씁니다.
- 보낸 메시지를 콘솔에 씁니다.

- 마지막으로 Stop 메서드를 호출하여 포트의 수신 대기를 중지합니다.

`Socket` 클래스를 사용하는 유한 TCP 컨트롤

`TcpClient` 및 `TcpListener` 둘 다 내부적으로 `Socket` 클래스에 의존합니다. 즉, 이러한 클래스로 수행할 수 있는 모든 작업은 소켓을 직접 사용하여 수행할 수 있습니다. 이 섹션에서는 여러 `TcpClient` 및 `TcpListener` 사용 사례와 기능적으로 동일한 해당 `Socket` 사용 사례를 보여 줍니다.

클라이언트 소켓 만들기

`TcpClient`의 기본 생성자는 *Socket(SocketType, ProtocolType)* 생성자를 통해 이중 스택 소켓을 만들려고 합니다. 이 생성자는 IPv6이 지원되면 이중 스택 소켓을 만들고, 그렇지 않으면 IPv4로 대체됩니다.

다음 TCP 클라이언트 코드를 고려합니다.

C#

```
using var client = new TcpClient();
```

이전 TCP 클라이언트 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
```

TcpClient(AddressFamily) 생성자

이 생성자는 세 개의 `AddressFamily` 값만 허용합니다. 그렇지 않으면 ArgumentException을 throw합니다. 유효한 값은 다음과 같습니다.

- AddressFamily.InterNetwork: IPv4 소켓의 경우.
- AddressFamily.InterNetworkV6: IPv6 소켓의 경우.
- AddressFamily.Unknown: 기본 생성자와 유사하게 이중 스택 소켓을 만들려고 시도합니다.

다음 TCP 클라이언트 코드를 고려합니다.

C#

```
using var client = new TcpClient(AddressFamily.InterNetwork);
```

이전 TCP 클라이언트 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
```

TcpClient(IPEndPoint) 생성자

소켓을 만들 때 이 생성자는 제공된 **로컬**에도 `IPEndPoint`됩니다. IPEndPoint.AddressFamily 속성은 소켓의 주소 패밀리를 결정하는 데 사용됩니다.

다음 TCP 클라이언트 코드를 고려합니다.

C#

```
var endPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5001);
using var client = new TcpClient(endPoint);
```

이전 TCP 클라이언트 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
// Example IPEndPoint object
var endPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5001);
using var socket = new Socket(endPoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
socket.Bind(endPoint);
```

TcpClient(String, Int32) 생성자

이 생성자는 기본 생성자와 유사한 이중 스택을 만들고 및 쌍으로 정의된 `hostname` DNS 엔드포인트에 `port`하려고 합니다.

다음 TCP 클라이언트 코드를 고려합니다.

C#

```
using var client = new TcpClient("www.example.com", 80);
```

이전 TCP 클라이언트 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
socket.Connect("www.example.com", 80);
```

서버에 연결

`Connect`의 `ConnectAsync` 모든 `BeginConnect`, `EndConnect` 및 `TcpClient` 오버로드는 해당 `Socket` 메서드와 기능적으로 동일합니다.

다음 TCP 클라이언트 코드를 고려합니다.

C#

```
using var client = new TcpClient();
client.Connect("www.example.com", 80);
```

위의 `TcpClient` 코드는 다음 소켓 코드와 동일합니다.

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
socket.Connect("www.example.com", 80);
```

서버 소켓 만들기

원시 `TcpClient` 해당 항목과 기능적으로 동등한 `Socket` 인스턴스처럼 이 섹션에서는 `TcpListener` 생성자를 해당 소켓 코드에 매핑합니다. 고려해야 할 첫 번째 생성자는 `TcpListener(IPAddress localaddr, int port)`입니다.

C#

```
var listener = new TcpListener(IPAddress.Loopback, 5000);
```

이전 TCP 수신기 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
var ep = new IPEndPoint(IPAddress.Loopback, 5000);
using var socket = new Socket(ep.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
```

서버에서 수신 대기 시작

Start() 메서드는 `Socket`의 Bind 및 Listen() 기능을 결합하는 래퍼입니다.

다음 TCP 수신기 코드를 고려해 보세요.

C#

```
var listener = new TcpListener(IPAddress.Loopback, 5000);
listener.Start(10);
```

이전 TCP 수신기 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
var endPoint = new IPEndPoint(IPAddress.Loopback, 5000);
using var socket = new Socket(endPoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
socket.Bind(endPoint);
try
{
    socket.Listen(10);
}
catch (SocketException)
{
    socket.Dispose();
}
```

서버 연결 허용

내부적으로, 들어오는 TCP 연결은 수락되면 항상 새 소켓을 만듭니다. `TcpListener`는 Socket 인스턴스를 직접 수락할 수 있습니다(AcceptSocket() 또는 AcceptSocketAsync()를 통해). 또는 TcpClient을 수락할 수 있습니다(AcceptTcpClient() 및 AcceptTcpClientAsync()를 통해).

다음 `TcpListener` 코드를 생각해 볼 수 있습니다.

C#

```
var listener = new TcpListener(IPAddress.Loopback, 5000);
using var acceptedSocket = await listener.AcceptSocketAsync();

// Synchronous alternative.
// var acceptedSocket = listener.AcceptSocket();
```

이전 TCP 수신기 코드는 다음 소켓 코드와 기능적으로 동일합니다.

C#

```
var endPoint = new IPEndPoint(IPAddress.Loopback, 5000);
using var socket = new Socket(endPoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
using var acceptedSocket = await socket.AcceptAsync();
```

```
// Synchronous alternative
// var acceptedSocket = socket.Accept();
```

데이터를 보내고 받을 `NetworkStream` 만들기

`TcpClient`를 사용하는 경우 데이터를 보내고 받을 수 있도록 NetworkStream을 GetStream() 메서드로 인스턴스화해야 합니다. `Socket`을 사용하면 `NetworkStream` 만들기를 수동으로 수행해야 합니다.

다음 `TcpClient` 코드를 생각해 볼 수 있습니다.

C#

```
using var client = new TcpClient();
using NetworkStream stream = client.GetStream();
```

이것은 다음 소켓 코드와 동일합니다.

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);

// Be aware that transferring the ownership means that closing/disposing the stream
will also close the underlying socket.
using var stream = new NetworkStream(socket, ownsSocket: true);
```

> **팁**
>
> 코드에 Stream 인스턴스를 사용할 필요가 없는 경우 `Socket`를 만드는 대신, Send의 Send/Receive 메서드(SendAsync, Receive, ReceiveAsync 및 NetworkStream)를 직접 사용할 수 있습니다.

추가 정보

- 소켓을 사용하여 TCP를 통해 데이터 보내기 및 받기
- .NET의 네트워킹
- Socket
- TcpClient
- TcpListener
- NetworkStream

Last updated on 2025. 10. 20.

System.Net.Sockets.Socket 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 클래스는 Socket 네트워크 통신을 위한 다양한 메서드 및 속성 집합을 제공합니다. 이 Socket 클래스를 사용하면 열거형에 나열된 ProtocolType 통신 프로토콜을 사용하여 동기 및 비동기 데이터 전송을 모두 수행할 수 있습니다.

클래스는 Socket 비동기 메서드에 대한 .NET 명명 패턴을 따릅니다. 예를 들어 동기 Receive 메서드는 비동 ReceiveAsync 기 변형에 해당합니다.

동기 작업 모드에는 다음 메서드를 사용합니다.

- TCP와 같은 연결 지향 프로토콜을 사용하는 경우 서버는 이 메서드를 사용하여 Listen 연결을 수신 대기할 수 있습니다. 이 메서드는 Accept 들어오는 연결 요청을 처리하고 원격 호스트와 데이터를 통신하는 데 사용할 수 있는 것을 반환 Socket 합니다. 반환된 Socket을(를) 사용하여 Send 또는 Receive 메서드를 호출합니다. 로컬 IP 주소와 포트 번호를 지정하려면 Bind 메서드를 호출하기 전에 Listen 메서드를 호출하십시오. 기본 서비스 공급자가 무료 포트를 할당하려면 포트 번호 0을 사용합니다. 수신 대기 호스트에 연결하려면 메서드를 호출합니다 Connect . 데이터를 통신하려면 Send 메서드 또는 Receive 메서드를 호출합니다.
- UDP와 같은 연결 없는 프로토콜을 사용하는 경우 연결을 전혀 수신 대기할 필요가 없습니다. 들어오는 데이터그램을 ReceiveFrom 수락하려면 메서드를 호출합니다. 이 메서드를 SendTo 사용하여 원격 호스트에 데이터그램을 보냅니다.

통신을 비동기적으로 처리하려면 다음 메서드를 사용합니다.

- TCP와 같은 연결 지향 프로토콜을 사용하는 경우 수신 대기 호스트와 연결하는 데 사용합니다 ConnectAsync . SendAsync 또는 ReceiveAsync을 사용하여 데이터를 비동기적으로 통신합니다. 들어오는 연결 요청은 .를 사용하여 AcceptAsync처리할 수 있습니다.
- UDP와 같은 연결 없는 프로토콜을 사용하는 경우 데이터그램을 보내고 데이터그램 SendToAsync을 받는 데 사용할 ReceiveFromAsync 수 있습니다.

소켓에서 여러 비동기 작업을 수행하는 경우 시작 순서대로 완료되지는 않습니다.

데이터 송수신을 마쳤으면 이 메서드를 Shutdown 사용하여 . Socket를 사용하지 않도록 설정합니다. Shutdown을(를) 호출한 후, Close에 연결된 모든 리소스를 해제하기 위해 Socket 메서드를 호출하세요.

Socket 클래스는 Socket 메서드를 사용하여 SetSocketOption을 구성할 수 있도록 합니다. 메서드를 사용하여 이러한 설정을 검색합니다 GetSocketOption .

.NET의 WebSockets 지원

아티클 • 2025. 01. 31.

WebSocket 프로토콜을 사용하면 클라이언트와 원격 호스트 간의 양방향 통신이 가능합니다. System.Net.WebSockets.ClientWebSocket은 여는 핸드셰이크를 통해 WebSocket 연결을 설정할 수 있는 기능을 제공하며, `ConnectAsync` 메서드에 의해 생성되고 전송됩니다.

HTTP/1.1 및 HTTP/2 WebSocket의 차이점

HTTP/1.1을 통한 WebSockets는 단일 TCP 연결을 사용하므로 연결 전체 헤더를 통해 관리됩니다. 자세한 내용은 RFC 6455 참조하세요. HTTP/1.1을 통해 WebSocket을 설정하는 방법의 다음 예제를 고려하세요.

```
c#

Uri uri = new("ws://corefx-net-
http11.azurewebsites.net/WebSocket/EchoWebSocket.ashx");

using ClientWebSocket ws = new();
await ws.ConnectAsync(uri, default);

var bytes = new byte[1024];
var result = await ws.ReceiveAsync(bytes, default);
string res = Encoding.UTF8.GetString(bytes, 0, result.Count);

await ws.CloseAsync(WebSocketCloseStatus.NormalClosure, "Client closed",
default);
```

멀티플렉싱 특성으로 인해 HTTP/2에서 다른 방법을 사용해야 합니다. WebSocket은 스트림별로 설정됩니다. 자세한 내용은 rfC 8441 참조하세요. HTTP/2를 사용하면 일반 HTTP 스트림과 함께 여러 웹 소켓 스트림에 대해 하나의 연결을 사용하고 HTTP/2의 보다 효율적인 네트워크 사용을 WebSockets로 확장할 수 있습니다. 기존 풀링된 연결을 재사용할 수 있도록 HttpMessageInvoker을 허용하는 특별한 ConnectAsync(Uri, HttpMessageInvoker, CancellationToken) 오버로드가 있습니다.

```
c#

using SocketsHttpHandler handler = new();
using ClientWebSocket ws = new();
await ws.ConnectAsync(uri, new HttpMessageInvoker(handler),
cancellationToken);
```

HTTP 버전 및 정책 설정

기본적으로 `ClientWebSocket` HTTP/1.1을 사용하여 여는 핸드셰이크를 보내고 다운그레이드를 허용합니다. .NET 7에서는 HTTP/2를 통한 웹 소켓을 사용할 수 있습니다. `ConnectAsync`호출하기 전에 변경할 수 있습니다.

```c#
using SocketsHttpHandler handler = new();
using ClientWebSocket ws = new();

ws.Options.HttpVersion = HttpVersion.Version20;
ws.Options.HttpVersionPolicy = HttpVersionPolicy.RequestVersionOrHigher;

await ws.ConnectAsync(uri, new HttpMessageInvoker(handler),
cancellationToken);
```

호환되지 않는 옵션

`ClientWebSocket` 연결이 설정되기 전에 사용자가 설정할 수 System.Net.WebSockets.ClientWebSocketOptions 속성이 있습니다. 그러나 `HttpMessageInvoker` 제공되면 이러한 속성도 있습니다. 모호성을 방지하려면 이 경우 속성은 `HttpMessageInvoker`설정해야 하며 `ClientWebSocketOptions` 기본값이 있어야 합니다. 그렇지 않으면, `ClientWebSocketOptions`이 변경되었을 때 `ConnectAsync` 오버로드가 ArgumentException를 발생시킵니다.

```c#
using HttpClientHandler handler = new()
{
    CookieContainer = cookies;
    UseCookies = cookies != null;
    ServerCertificateCustomValidationCallback =
remoteCertificateValidationCallback;
    Credentials = useDefaultCredentials
        ? CredentialCache.DefaultCredentials
        : credentials;
};
if (proxy is null)
{
    handler.UseProxy = false;
}
else
{
    handler.Proxy = proxy;
}
if (clientCertificates?.Count > 0)
```

```
{
    handler.ClientCertificates.AddRange(clientCertificates);
}
HttpMessageInvoker invoker = new(handler);
using ClientWebSocket cws = new();
await cws.ConnectAsync(uri, invoker, cancellationToken);
```

압축

WebSocket 프로토콜은 RFC 7692 정의된 대로 메시지별 수축을 지원합니다. System.Net.WebSockets.ClientWebSocketOptions.DangerousDeflateOptions의해 제어됩니다. 있는 경우 옵션은 핸드셰이크 단계 중에 서버로 전송됩니다. 서버에서 메시지별 deflate를 지원하고 옵션이 허용되는 경우 모든 메시지에 대해 기본적으로 압축을 사용하도록 설정된 `ClientWebSocket` 인스턴스가 만들어집니다.

c#

```
using ClientWebSocket ws = new()
{
    Options =
    {
        DangerousDeflateOptions = new WebSocketDeflateOptions()
        {
            ClientMaxWindowBits = 10,
            ServerMaxWindowBits = 10
        }
    }
};
```

> ⓘ **중요**
>
> 압축을 사용하기 전에 애플리케이션을 사용하도록 설정하면 범죄/위반 유형의 공격이 적용된다는 점에 유의하세요. 자세한 내용은 CRIME 및 위반 참조하세요. 이러한 메시지에 대한 `DisableCompression` 플래그를 지정하여 비밀이 포함된 데이터를 보낼 때 압축을 해제하는 것이 좋습니다.

Keep-Alive 전략

.NET 8 및 그 이전 버전에서는 유일하게 사용 가능한 Keep-Alive 전략이 원치 않는 PONG입니다. 이 전략은 기본 TCP 연결이 유휴 상태가 되지 않도록 하기에 충분합니다. 그러나 원격 호스트가 응답하지 않는 경우(예: 원격 서버 크래시) 원치 않는 PONG에서 이러한 상황을 감지하는 유일한 방법은 TCP 시간 제한에 의존하는 것입니다.

.NET 9 기존 `KeepAliveInterval` 설정을 새 `KeepAliveTimeout` 설정으로 보완하여 오랫동안 원하는 *PING/PONG* Keep-Alive 전략을 도입했습니다. .NET 9부터 Keep-Alive 전략이 다음과 같이 선택됩니다.

1. Keep-Alive는 **OFF**의 경우
    - `KeepAliveInterval` `TimeSpan.Zero` 또는 `Timeout.InfiniteTimeSpan`
2. **요청되지 않은 PONG**,
    - `KeepAliveInterval`은 양수이자 유한한 `TimeSpan`이며, -AND-
    - `KeepAliveTimeout` `TimeSpan.Zero` 또는 `Timeout.InfiniteTimeSpan`
3. **PING/PONG**(있는 경우)
    - `KeepAliveInterval` 양수 유한 `TimeSpan`, -AND-
    - `KeepAliveTimeout`는 양의 유한 `TimeSpan`입니다.

기본 `KeepAliveTimeout` 값은 `Timeout.InfiniteTimeSpan`. 따라서 기본 Keep-Alive 동작은 .NET 버전 간에 일관성을 유지합니다.

`ClientWebSocket`사용하는 경우 기본 ClientWebSocketOptions.KeepAliveInterval 값은 WebSocket.DefaultKeepAliveInterval(일반적으로 30초)입니다. 즉, `ClientWebSocket`은 기본적으로 Keep-Alive이 ON 상태이며, 기본 전략은 요청되지 않은 PONG입니다.

PING/PONG 전략으로 전환하려면 ClientWebSocketOptions.KeepAliveTimeout을 재정의하면 충분합니다.

C#

```
var ws = new ClientWebSocket();
ws.Options.KeepAliveTimeout = TimeSpan.FromSeconds(20);
await ws.ConnectAsync(uri, cts.Token);
```

기본 `WebSocket`경우 Keep-Alive 기본적으로 OFF입니다. PING/PONG 전략을 사용하려면 WebSocketCreationOptions.KeepAliveInterval 및 WebSocketCreationOptions.KeepAliveTimeout 모두 설정해야 합니다.

C#

```
var options = new WebSocketCreationOptions()
{
    KeepAliveInterval = WebSocket.DefaultKeepAliveInterval,
    KeepAliveTimeout = TimeSpan.FromSeconds(20)
```

```
};
var ws = WebSocket.CreateFromStream(stream, options);
```

원치 않는 PONG 전략을 사용하는 경우 PONG 프레임은 단방향 하트비트로 사용됩니다. 원격 엔드포인트가 통신하는지 여부에 관계없이 `KeepAliveInterval` 간격으로 정기적으로 전송됩니다.

PING/PONG 전략이 활성화된 경우 *마지막 통신이 원격 엔드포인트에서* 이후 경과된 `KeepAliveInterval` 시간 후에 PING 프레임이 전송됩니다. 각 PING 프레임에는 예상된 PONG 응답과 쌍을 이루는 정수 토큰이 포함되어 있습니다. `KeepAliveTimeout` 경과한 후 PONG 응답이 도착하지 않으면 원격 엔드포인트가 응답하지 않는 것으로 간주되고 WebSocket 연결이 자동으로 중단됩니다.

C#

```
var ws = new ClientWebSocket();
ws.Options.KeepAliveInterval = TimeSpan.FromSeconds(10);
ws.Options.KeepAliveTimeout = TimeSpan.FromSeconds(10);
await ws.ConnectAsync(uri, cts.Token);

// NOTE: There must be an outstanding read at all times to ensure
// incoming PONGs are processed
var result = await _webSocket.ReceiveAsync(buffer, cts.Token);
```

시간 제한이 경과하면 미해결인 `ReceiveAsync`이(가) `OperationCanceledException`예외를 발생시킵니다.

txt

```
System.OperationCanceledException: Aborted
 ---> System.AggregateException: One or more errors occurred. (The WebSocket
didn't receive a Pong frame in response to a Ping frame within the
configured KeepAliveTimeout.) (Unable to read data from the transport
connection: The I/O operation has been aborted because of either a thread
exit or an application request..)
 ---> System.Net.WebSockets.WebSocketException (0x80004005): The WebSocket
didn't receive a Pong frame in response to a Ping frame within the
configured KeepAliveTimeout.
   at System.Net.WebSockets.ManagedWebSocket.KeepAlivePingHeartBeat()
...
```

PONG 처리를 위해 계속 읽기

ⓘ 참고

현재 `WebSocket`는 `ReceiveAsync`이 보류 중인 경우에만 들어오는 프레임을 처리합니다.

> **ⓘ 중요**
>
> Keep-Alive 제한 시간을 사용하려면 PONG 응답이 *중요하게*즉시 처리되어야 합니다. 원격 엔드포인트가 활성 상태이며 PONG 응답을 올바르게 전송함에도 불구하고, `WebSocket`이 들어오는 프레임을 처리하지 않을 경우, Keep-Alive 메커니즘은 "오탐" 중단을 실행할 수 있습니다. 이 문제는 시간 제한이 경과하기 전에 전송 스트림에서 PONG 프레임을 선택하지 않는 경우에 발생할 수 있습니다.

양호한 연결이 끊어지지 않도록 하기 위해, Keep-Alive 타임아웃이 구성된 모든 WebSocket에서 읽기 작업을 보류 상태로 유지하는 것이 좋습니다.

TLS/SSL 모범 사례

2025. 06. 17.

TLS(전송 계층 보안)는 인터넷을 통해 두 컴퓨터 간의 통신을 보호하도록 설계된 암호화 프로토콜입니다. .NET에서 SslStream 클래스를 통해 TLS 프로토콜이 노출됩니다.

이 문서에서는 클라이언트와 서버 간의 보안 통신을 설정하는 모범 사례를 제시하고 .NET을 사용하는 것으로 가정합니다. .NET Framework의 모범 사례는 .NET Framework를 사용한 TLS(전송 계층 보안) 모범 사례를 참조하세요.

TLS 버전 선택

속성을 통해 EnabledSslProtocols 사용 가능한 TLS 프로토콜의 버전을 지정할 수 있지만, None 값을 사용하여 운영 체제 설정에 따르는 것이 좋습니다(이것이 기본값임).

결정을 OS로 연기하면 사용 가능한 최신 버전의 TLS가 자동으로 사용되며 OS 업그레이드 후 애플리케이션이 변경 내용을 선택할 수 있습니다. 운영 체제는 더 이상 안전한 것으로 간주되지 않는 TLS 버전을 사용하지 못할 수도 있습니다.

암호 그룹 선택

`SslStream` 에서는 사용자가 클래스를 통해 CipherSuitesPolicy TLS 핸드셰이크로 협상할 수 있는 암호 그룹을 지정할 수 있습니다. TLS 버전과 마찬가지로 OS에서 협상하기에 가장 적합한 암호화 제품군을 결정할 수 있도록 하는 것이 좋습니다. 따라서 사용하지 CipherSuitesPolicy않는 것이 좋습니다.

> ⓘ **참고**
>
> CipherSuitesPolicy 는 Windows에서 지원되지 않으며 인스턴스화를 시도하면 NotSupportedException 예외가 발생합니다.

서버 인증서 지정

서버 SslStream 로 인증하는 경우 인스턴스가 X509Certificate2 필요합니다. 프라이빗 키도 포함하는 인스턴스를 X509Certificate2 항상 사용하는 것이 좋습니다.

서버 인증서를 전달할 수 있는 방법에는 여러 가지가 있습니다.SslStream

- SslStream.AuthenticateAsServerAsync에 대한 매개변수로 직접 또는 SslServerAuthenticationOptions.ServerCertificate 속성을 통해

- SslServerAuthenticationOptions.ServerCertificateSelectionCallback 속성의 선택 콜백에서
- 속성에 SslStreamCertificateContext를 전달하여 SslServerAuthenticationOptions.ServerCertificateContext

권장되는 방법은 속성을 사용하는 것입니다 SslServerAuthenticationOptions.ServerCertificateContext . 다른 두 가지 방법 중 하나로 인증서를 획득하면 SslStreamCertificateContext 구현체가 내부적으로 SslStream 인스턴스를 생성합니다. SslStreamCertificateContext을 만들기 위해서는 CPU 집약적인 작업인 X509Chain를 빌드해야 합니다. 한 번 만들고 SslStreamCertificateContext 여러 SslStream 인스턴스에 다시 사용하는 것이 더 효율적입니다.

인스턴스를 SslStreamCertificateContext 다시 사용하면 Linux 서버에서 TLS 세션 재개 와 같은 추가 기능도 사용할 수 있습니다.

사용자 지정 `X509Certificate` 유효성 검사

기본 인증서 유효성 검사 절차가 적절하지 않고 일부 사용자 지정 유효성 검사 논리가 필요한 특정 시나리오가 있습니다. 유효성 검사 논리의 일부를 지정 SslClientAuthenticationOptions.CertificateChainPolicy 하거나 SslServerAuthenticationOptions.CertificateChainPolicy지정하여 사용자 지정할 수 있습니다. 또는 System.Net.Security.SslClientAuthenticationOptions.RemoteCertificateValidationCallback< 속성을 통해 >완전히 사용자 지정 논리를 제공할 수 있습니다. 자세한 내용은 사용자 지정 인증서 트러스트를 참조하세요.

사용자 지정 인증서 신뢰

컴퓨터에서 신뢰하는 인증 기관에서 발급하지 않은 인증서(자체 서명된 인증서 포함)가 발견되면 기본 인증서 유효성 검사 절차가 실패합니다. 이 문제를 해결할 수 있는 한 가지 방법은 컴퓨터의 신뢰할 수 있는 저장소에 필요한 발급자 인증서를 추가하는 것입니다. 그러나 이는 시스템의 다른 애플리케이션에 영향을 줄 수 있으며 항상 가능한 것은 아닙니다.

대체 솔루션은 .를 통해 신뢰할 수 있는 사용자 지정 루트 인증서를 지정하는 것입니다 X509ChainPolicy. 유효성 검사 중에 시스템 신뢰 목록 대신 사용할 사용자 지정 신뢰 목록을 지정하려면 다음 예제를 고려하세요.

C#

```
SslClientAuthenticationOptions clientOptions = new();

clientOptions.CertificateChainPolicy = new X509ChainPolicy()
{
    TrustMode = X509ChainTrustMode.CustomRootTrust,
    CustomTrustStore =
```

```
    {
        customIssuerCert
    }
};
```

이전 정책으로 구성된 클라이언트는 신뢰할 수 있는 인증서만 허용합니다 `customIssuerCert`.

특정 유효성 검사 오류 무시

영구 클록이 없는 IoT 디바이스를 고려합니다. 전원을 켜면 디바이스의 시계가 몇 년 전에 시작되므로 모든 인증서는 "아직 유효하지 않음"으로 간주됩니다. 유효성 검사 기간 위반을 무시하는 유효성 검사 콜백 구현을 보여 주는 다음 코드를 고려합니다.

C#

```
static bool CustomCertificateValidationCallback(
    object sender,
    X509Certificate? certificate,
    X509Chain? chain,
    SslPolicyErrors sslPolicyErrors)
{
    // Anything that would have been accepted by default is OK
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        return true;
    }

    // If there is something wrong other than a chain processing error, don't
trust it.
    if (sslPolicyErrors != SslPolicyErrors.RemoteCertificateChainErrors)
    {
        return false;
    }

    Debug.Assert(chain is not null);

    // If the reason for RemoteCertificateChainError is that the chain built
empty, don't trust it.
    if (chain.ChainStatus.Length == 0)
    {
        return false;
    }

    foreach (X509ChainStatus status in chain.ChainStatus)
    {
        // If an error other than `NotTimeValid` (or `NoError`) is present, don't
trust it.
        if ((status.Status & ~X509ChainStatusFlags.NotTimeValid) !=
X509ChainStatusFlags.NoError)
        {
            return false;
```

```
        }
    }

    return true;
}
```

인증서 고정

사용자 지정 인증서 유효성 검사가 필요한 또 다른 상황은 클라이언트가 서버에서 특정 인증서 또는 알려진 인증서의 작은 집합의 인증서를 사용해야 하는 경우입니다. 이 방법을 인증서 고정이라고 합니다. 다음 코드 조각은 서버가 알려진 특정 공개 키가 있는 인증서를 제공하는지 확인하는 유효성 검사 콜백을 보여 줍니다.

C#

```
static bool CustomCertificateValidationCallback(
    object sender,
    X509Certificate? certificate,
    X509Chain? chain,
    SslPolicyErrors sslPolicyErrors)
{
    // If there is something wrong other than a chain processing error, don't
trust it.
    if ((sslPolicyErrors & ~SslPolicyErrors.RemoteCertificateChainErrors) != 0)
    {
        return false;
    }

    Debug.Assert(certificate is not null);

    const string ExpectedPublicKey =
        "3082010A0282010100C204ECF88CEE04C2B3D850D57058CC9318EB5C" +
        "A86849B022B5F9959EB12B2C763E6CC04B604C4CEAB2B4C00F80B6B0" +
        "F972C98602F95C415D132B7F71C44BBCE9942E5037A6671C618CF641" +
        "42C546D31687279F74EB0A9D11522621736C844C7955E4D16BE8063D" +
        "481552ADB328DBAAFF6EFF60954A776B39F124D131B6DD4DC0C4FC53" +
        "B96D42ADB57CFEAEF515D23348E72271C7C2147A6C28EA374ADFEA6C" +
        "B572B47E5AA216DC69B15744DB0A12ABDEC30F47745C4122E19AF91B" +
        "93E6AD2206292EB1BA491C0C279EA3FB8BF7407200AC9208D98C5784" +
        "538105CBE6FE6B5498402785C710BB7370EF6918410745557CF9643F" +
        "3D2CC3A97CEB931A4C86D1CA850203010001";

    return certificate.GetPublicKeyString().Equals(ExpectedPublicKey);
}
```

클라이언트 인증서 유효성 검사에 대한 고려 사항

서버 애플리케이션은 클라이언트 인증서를 요구하고 유효성을 검사할 때 주의해야 합니다. 인증서에는 발급자 인증서를 다운로드할 수 있는 위치를 지정하는 AIA(기관 정보 액세스) 확장이 포함될 수 있습니다. 따라서 서버는 클라이언트 인증서를 빌드할 때 외부 서버에서 발급자 인증서를 X509Chain 다운로드하려고 시도할 수 있습니다. 마찬가지로 서버는 외부 서버에 연결하여 클라이언트 인증서가 해지되지 않았는지 확인해야 할 수 있습니다.

외부 서버를 빌드하고 유효성을 X509Chain 검사할 때 외부 서버에 연결해야 하는 경우 외부 서버가 응답 속도가 느린 경우 애플리케이션이 서비스 거부 공격에 노출될 수 있습니다. 따라서 서버 애플리케이션은 X509Chain의 빌드 동작을 CertificateChainPolicy을 사용하여 구성해야 합니다.

인증 문제 해결 `SslStream`

2025. 06. 17.

이 문서에서는 .NET에서 암호화 및 보안 관련 기능을 사용할 SslStream 때 가장 자주 발생하는 인증 문제를 OS API(예: Windows의 Schannel) 또는 하위 수준 시스템 라이브러리(예: Linux의 OpenSSL)와 상호 운용하여 구현합니다. 따라서 예외 메시지 및 오류 코드를 포함한 .NET 애플리케이션의 동작은 실행되는 플랫폼에 따라 변경됩니다.

Wireshark 또는 tcpdump 와 같은 도구를 사용하여 유선으로 교환되는 실제 메시지를 관찰하여 일부 문제를 보다 쉽게 조사하고 해결할 수 있습니다. 이러한 도구를 사용하여 광고된 지원 TLS 버전 및 허용되고 협상된 암호화 스위트, 그리고 인증을 위해 교환된 인증서를 포함한 `ClientHello`, `ServerHello`, 기타 메시지를 검사할 수 있습니다.

중간 인증서가 전송되지 않음

TLS 핸드셰이크 중에 서버(및 클라이언트 인증이 요청된 경우 클라이언트도)는 인증서를 전송하여 해당 ID를 클라이언트에 증명합니다. 인증서의 신뢰성을 확인하려면 인증서 체인을 빌드하고 확인해야 합니다. 체인의 루트는 신뢰할 수 있는 루트 CA(인증 기관) 중 하나여야 하며, 인증서는 컴퓨터 인증서 저장소에 저장됩니다.

신뢰할 수 있는 CA 중 하나에서 피어 인증서를 발급하지 않은 경우 인증서 체인을 빌드하려면 중간 CA 인증서가 필요합니다. 그러나 중간 인증서를 사용할 수 없는 경우 인증서의 유효성을 검사할 수 없으며 TLS 핸드셰이크가 실패할 수 있습니다.

이 문제는 Windows에서 가장 자주 발생합니다. 애플리케이션에서 인증 옵션을 통해 중간 인증서를 제공했지만 Windows 인증서 저장소에 저장되지 않는 한 피어로 전송되지 않습니다. 이 제한은 실제 TLS 핸드셰이크가 애플리케이션 프로세스 외부에서 발생하기 때문입니다.

서버 애플리케이션의 경우 SslStreamCertificateContext를 SslServerAuthenticationOptions.ServerCertificateContext로 전달할 수 있습니다. 인스턴스를 생성할 SslStreamCertificateContext 때 추가 중간 인증서를 전달할 수 있으며 인증서 저장소에 임시로 추가됩니다.

아쉽게도 클라이언트 애플리케이션의 유일한 솔루션은 인증서 저장소에 인증서를 수동으로 추가하는 것입니다.

임시 키로 핸드셰이크 실패

Windows에서는 임시 키와 `(0x8009030E): No credentials are available in the security package` 함께 인증서를 사용하려고 할 때 오류 메시지가 표시될 수 있습니다. 이 동작은 기본 OS

API(Schannel)의 버그 때문입니다. 관련 정보 및 해결 방법은 관련 GitHub 문제 에서 찾을 수 있습니다.

클라이언트 및 서버에 공통 알고리즘이 없습니다.

`ClientHello`와 `ServerHello` 메시지를 검사할 때, 클라이언트와 서버 모두에서 제공하는 암호 제품군이 없거나, 명시적으로 구성된 경우에도 일부 암호 제품군이 제공되지 않는다는 것을 알 수 있습니다(CipherSuitesPolicy는 Linux에서만 사용 가능). 기본 TLS 라이브러리는 안전하지 않은 것으로 간주되는 TLS 버전 및 암호 그룹을 사용하지 않도록 설정할 수 있습니다.

많은 Linux 배포판에서 관련 구성 파일은 .에 `/etc/ssl/openssl.cnf`있습니다.

Windows에서는 Enable-TlsCipherSuiteDisable-TlsCipherSuite 암호 그룹을 구성하는 데 PowerShell cmdlet을 사용할 수 있습니다. 레지스트리 키를 구성 `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols\TLS <version>\{Client|Server}\Enabled` 하여 개별 TLS 버전을 사용하거나 사용하지 않도록 설정할 수 있습니다.

.NET Framework에서 .NET으로 SslStream 코드 마이그레이션

2025. 06. 22.

.NET Core는 SslStream 작동 방식에 많은 개선 사항과 호환성을 깨는 변경 사항을 가져왔습니다. 네트워크 보안과 관련된 가장 중요한 변경 사항은 클래스 ServicePointManager가 더 이상 사용되지 않으며, 레거시 WebRequest 인터페이스에만 영향을 준다는 것입니다.

각 SslStream 인스턴스마다 허용된 TLS 프로토콜과 인증서 유효성 검사 콜백을 SslServerAuthenticationOptions 또는 SslClientAuthenticationOptions를 통해 각각 개별적으로 구성해야 합니다. HTTPS에서 HttpClient사용되는 네트워크 보안 옵션을 구성하려면 기본 처리기에서 보안 옵션을 구성해야 합니다. HttpClient에서 사용되는 기본 처리기는 SocketsHttpHandler이며, 이 처리기에는 SslOptions을(를) 허용하는 SslClientAuthenticationOptions 속성이 있습니다.

사용자 지정 인증서 유효성 검사 콜백을 HttpClient 사용하여 만드는 방법을 보여 주는 다음 예제를 고려합니다.

C#

```
bool CustomCertificateValidator(
    object sender,
    X509Certificate? certificate,
    X509Chain? chain,
    SslPolicyErrors sslPolicyErrors)
{
    // TODO: Always returns false.
    // Need to implement certificate evaluation logic.
    return false;
}

HttpClient httpClient = new(
    new SocketsHttpHandler
    {
        SslOptions =
        {
            RemoteCertificateValidationCallback = CustomCertificateValidator
        }
    });
```

다음 표에서는 TLS와 관련된 개별 ServicePointManager 속성을 마이그레이션하는 방법을 보여 줍니다.

테이블 확장

원본 API	대상 API
CheckCertificateRevocationList	X509RevocationMode에 적절한 SslClientAuthenticationOptions.CertificateRevocationCheckMode 설정을 적용합니다.
EncryptionPolicy	SslClientAuthenticationOptions.EncryptionPolicy을 사용합니다.
SecurityProtocol	SslClientAuthenticationOptions.EnabledSslProtocols을 사용합니다.
ServerCertificateValidationCallback	SslClientAuthenticationOptions.RemoteCertificateValidationCallback을 사용합니다.

QUIC 프로토콜

QUIC는 RFC 9000으로 표준화된 네트워크 전송 계층 프로토콜입니다. UDP를 기본 프로토콜로 사용하며 TLS 1.3 사용을 의무화하므로 기본적으로 안전합니다. 자세한 내용은 RFC 9001을 참조하세요. TCP 및 UDP와 같은 잘 알려진 전송 프로토콜의 또 다른 흥미로운 차이점은 전송 계층에 스트림 멀티플렉싱이 기본 제공된다는 것입니다. 이렇게 하면 서로 영향을 주지 않는 여러 개의 동시 독립 데이터 스트림을 가질 수 있습니다.

QUIC 자체는 전송 프로토콜이기 때문에 교환된 데이터에 대한 의미 체계를 정의하지 않습니다. 대신 애플리케이션 계층 프로토콜(예: HTTP/3 또는 QUIC를 통한 SMB)에서 사용됩니다. 사용자 지정 정의 프로토콜에도 사용할 수 있습니다.

프로토콜은 TLS를 사용하는 TCP에 비해 많은 이점을 제공하는데, 몇 가지는 다음과 같습니다.

- TCP 위에 TLS가 있는 경우보다 왕복 횟수가 적어 연결 설정이 더 빠릅니다.
- 하나의 패킷 손실이 다른 모든 스트림의 데이터를 차단하지 않는 헤드 오브 라인 차단 문제를 방지합니다.

반면에 QUIC를 사용할 때 고려해야 할 잠재적인 단점이 있습니다. 최신 프로토콜이기 때문에 도입은 여전히 증가하고 있지만 제한적입니다. 그 외에도 일부 네트워킹 구성 요소에 의해 QUIC 트래픽이 차단될 수도 있습니다.

.NET의 QUIC

QUIC 구현은 .NET 5에서 `System.Net.Quic` 라이브러리로 도입되었습니다. 그러나 .NET 7까지 라이브러리는 엄밀히 말해 내부용이었고 HTTP/3의 구현으로만 사용되었습니다. .NET 7을 사용하면 라이브러리가 공개되어 API가 노출되었습니다.

> ⓘ **참고 항목**
>
> .NET 7.0 및 8.0에서는 API가 **미리 보기 기능**으로 게시되었습니다. .NET 9부터 이러한 API는 더 이상 미리 보기 기능으로 간주되지 않으며 이제 안정적인 것으로 간주됩니다.

구현 관점에서 `System.Net.Quic`는 QUIC 프로토콜의 네이티브 구현인 MsQuic에 따라 달라집니다. 따라서 `System.Net.Quic` 플랫폼 지원 및 종속성은 MsQuic에서 상속되고 플랫폼 종속성 섹션에 설명되어 있습니다. 즉, MsQuic 라이브러리는 Windows용 .NET의 일부로 제공됩니다. 그러나 Linux의 경우 적절한 패키지 관리자를 통해 `libmsquic`를 수동으로 설치해야 합니다. 다른 플랫폼의 경우 여전히 SChannel 또는 OpenSSL에 대해 MsQuic를 수동으로 빌드하고 `System.Net.Quic`와 함께 사용할 수 있습니다. 그러나 이러한 시나리오는 테스트 매트릭스의 일부가 아니며 예기치 않은 문제가 발생할 수 있습니다.

플랫폼 종속성

다음 섹션에서는 .NET의 QUIC에 대한 플랫폼 종속성에 대해 설명합니다.

윈도우즈

- Windows 11, Windows Server 2022 이상 (이전 Windows 버전에는 QUIC를 지원하는 데 필요한 암호화 API가 없습니다.)

Windows에서 msquic.dll .NET 런타임의 일부로 배포되며 설치하는 데 다른 단계가 필요하지 않습니다.

리눅스

> ⓘ **참고 항목**
>
> .NET 7 이상은 libmsquic의 2.2 이상 버전과만 호환됩니다.

`libmsquic` 패키지는 Linux에 필요합니다. 이 패키지는 Microsoft의 공식 Linux 패키지 리포지토리에 게시되며 Alpine https://packages.microsoft.com Packages - libmsquic와 같은 일부 공식 리포지토리에서도 사용할 수 있습니다.

`libmsquic` Microsoft의 공식 Linux 패키지 리포지토리에서 설치

패키지를 설치하기 전에 패키지 관리자에 이 리포지토리를 추가해야 합니다. 자세한 내용은 Microsoft 제품용 Linux 소프트웨어 리포지토리를 참조하세요.

> ⊗ **주의**
>
> 배포의 리포지토리에서 .NET 및 기타 Microsoft 패키지를 제공하는 경우 Microsoft 패키지 리포지토리를 추가하면 배포의 리포지토리와 충돌할 수 있습니다. 패키지 혼합을 방지하거나 문제를 해결하려면 **Linux에서 누락된 파일과 관련된 .NET 오류 문제**를 검토합니다.

예제

패키지 관리자를 사용하여 `libmsquic`를 설치하는 몇 가지 예는 다음과 같습니다.

- APT

  ```Bash
  sudo apt-get install libmsquic
  ```

- APK

  ```Bash
  sudo apk add libmsquic
  ```

- DNF

  ```Bash
  sudo dnf install libmsquic
  ```

- zypper

  ```Bash
  ```

```
sudo zypper install libmsquic
```

- YUM

```Bash
sudo yum install libmsquic
```

`libmsquic` 배포 패키지 리포지토리에서 설치

`libmsquic` 배포 패키지 저장소에서 설치할 수도 있지만, 현재는 `Alpine` 전용입니다.

예제

패키지 관리자를 사용하여 `libmsquic`를 설치하는 몇 가지 예는 다음과 같습니다.

- **Alpine 3.21 이상**

```Bash
apk add libmsquic
```

- **Alpine 3.20 이상**

```Bash
# Get libmsquic from community repository edge branch.
apk add --repository=http://dl-cdn.alpinelinux.org/alpine/edge/community/ libmsquic
```

libmsquic의 종속성

다음 종속성은 모두 `libmsquic` 패키지 매니페스트에 명시되며 패키지 관리자가 자동으로 설치합니다.

- OpenSSL 3 이상 또는 1.1 - 배포 버전의 기본 OpenSSL 버전(예: Ubuntu 22 의 경우 OpenSSL 3 및 Ubuntu 20 의 경우 OpenSSL 1.1)에 따라 달라집니다.
- libnuma1

macOS

이제 QUIC는 몇 가지 제한 사항이 있는 비표준 Homebrew 패키지 관리자를 통해 macOS에서 부분적으로 지원됩니다. 다음 명령을 사용하여 Homebrew를 사용하여 macOS에 설치 `libmsquic` 할 수 있습니다.

```Bash
brew install libmsquic
```

사용하는 `libmsquic`.NET 애플리케이션을 실행하려면 환경 변수를 실행하기 전에 설정해야 합니다. 이렇게 하면 애플리케이션이 런타임 중의 동적 로드 동안 라이브러리 `libmsquic`를 찾을 수 있습니다. 기본 명령 앞에 다음 명령을 추가하여 이 작업을 수행할 수 있습니다.

Bash

```
DYLD_FALLBACK_LIBRARY_PATH=$DYLD_FALLBACK_LIBRARY_PATH:$(brew --prefix)/lib dotnet run
```

또는

Bash

```
DYLD_FALLBACK_LIBRARY_PATH=$DYLD_FALLBACK_LIBRARY_PATH:$(brew --prefix)/lib ./binaryname
```

또는 다음을 사용하여 환경 변수를 설정할 수 있습니다.

Bash

```
export DYLD_FALLBACK_LIBRARY_PATH=$DYLD_FALLBACK_LIBRARY_PATH:$(brew --prefix)/lib
```

그런 다음, 기본 명령을 실행합니다.

Bash

```
./binaryname
```

API 개요

System.Net.Quic는 QUIC 프로토콜을 사용할 수 있도록 하는 세 가지 주요 클래스를 제공합니다.

- QuicListener - 들어오는 연결을 수락하기 위한 서버 쪽 클래스입니다.
- QuicConnection - RFC 9000 섹션 5에 해당하는 QUIC 연결입니다.
- QuicStream - RFC 9000 섹션 2에 해당하는 QUIC 스트림입니다.

그러나 이러한 클래스를 사용하기 전에 `libmsquic`가 누락되었거나 TLS 1.3이 지원되지 않을 수 있기 때문에 코드는 QUIC가 현재 지원되는지 여부를 검사해야 합니다. 이를 위해 `QuicListener`와 `QuicConnection`은 모두 정적 속성 `IsSupported`를 노출합니다.

C#

```
if (QuicListener.IsSupported)
{
    // Use QuicListener
}
else
{
    // Fallback/Error
}

if (QuicConnection.IsSupported)
{
    // Use QuicConnection
```

```
}
else
{
    // Fallback/Error
}
```

이러한 속성은 동일한 값을 보고하지만 나중에 변경될 수 있습니다. 서버 시나리오의 경우 IsSupported를, 클라이언트 시나리오의 경우 IsSupported를 검사하는 것이 좋습니다.

`QuicListener`

QuicListener는 클라이언트에서 들어오는 연결을 허용하는 서버 쪽 클래스를 나타냅니다. 수신기가 생성되고 정적 메서드 ListenAsync(QuicListenerOptions, CancellationToken)로 시작됩니다. 이 메서드는 수신기를 시작하고 들어오는 연결을 수락하는 데 필요한 모든 설정을 사용하여 QuicListenerOptions 클래스의 인스턴스를 허용합니다. 그 후 수신기는 AcceptConnectionAsync(CancellationToken)를 통해 연결을 전달할 준비를 갖추게 됩니다. 이 메서드에서 반환된 연결은 항상 완전히 연결되므로 TLS 핸드셰이크가 완료되고 연결을 사용할 준비가 된 것입니다. 마지막으로, 수신 대기를 중지하고 모든 리소스를 해제하려면 DisposeAsync()를 호출해야 합니다.

다음 `QuicListener` 예제 코드를 참조하세요.

C#

```
using System.Net.Quic;

// First, check if QUIC is supported.
if (!QuicListener.IsSupported)
{
    Console.WriteLine("QUIC is not supported, check for presence of libmsquic and support of TLS 1.3.");
    return;
}

// Share configuration for each incoming connection.
// This represents the minimal configuration necessary.
var serverConnectionOptions = new QuicServerConnectionOptions
{
    // Used to abort stream if it's not properly closed by the user.
    // See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
    DefaultStreamErrorCode = 0x0A, // Protocol-dependent error code.

    // Used to close the connection if it's not done by the user.
    // See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
    DefaultCloseErrorCode = 0x0B, // Protocol-dependent error code.

    // Same options as for server side SslStream.
    ServerAuthenticationOptions = new SslServerAuthenticationOptions
    {
        // Specify the application protocols that the server supports. This list must be a subset of the protocols specified in QuicListenerOptions.ApplicationProtocols.
        ApplicationProtocols = [new SslApplicationProtocol("protocol-name")],
        // Server certificate, it can also be provided via ServerCertificateContext or ServerCertificateSelectionCallback.
        ServerCertificate = serverCertificate
    }
};

// Initialize, configure the listener and start listening.
var listener = await QuicListener.ListenAsync(new QuicListenerOptions
```

```
{
    // Define the endpoint on which the server will listen for incoming connections. The port
number 0 can be replaced with any valid port number as needed.
    ListenEndPoint = new IPEndPoint(IPAddress.Loopback, 0),
    // List of all supported application protocols by this listener.
    ApplicationProtocols = [new SslApplicationProtocol("protocol-name")],
    // Callback to provide options for the incoming connections, it gets called once per each
connection.
    ConnectionOptionsCallback = (_, _, _) => ValueTask.FromResult(serverConnectionOptions)
});

// Accept and process the connections.
while (isRunning)
{
    // Accept will propagate any exceptions that occurred during the connection establishment,
    // including exceptions thrown from ConnectionOptionsCallback, caused by invalid
QuicServerConnectionOptions or TLS handshake failures.
    var connection = await listener.AcceptConnectionAsync();

    // Process the connection...
}

// When finished, dispose the listener.
await listener.DisposeAsync();
```

`QuicListener` 설계 방법에 대한 자세한 내용은 API 제안 을 참조하세요.

`QuicConnection`

QuicConnection는 서버 및 클라이언트 쪽 QUIC 연결에 사용되는 클래스입니다. 서버 쪽 연결은 수신기에 의해 내부에서 만들어지고 AcceptConnectionAsync(CancellationToken)를 통해 전달됩니다. 클라이언트 쪽 연결을 열고 서버에 연결해야 합니다. 수신기와 마찬가지로 연결을 인스턴스화하고 연결하는 정적 메서드 ConnectAsync(QuicClientConnectionOptions, CancellationToken)가 있습니다. QuicClientConnectionOptions와 유사한 클래스의 QuicServerConnectionOptions인스턴스를 허용합니다. 그 후에는 클라이언트와 서버 간에 연결 작업이 달라지지 않습니다. 나가는 스트림을 열고 들어오는 스트림을 수락할 수 있습니다. 또한 연결에 대한 정보(예: LocalEndPoint, RemoteEndPoint 또는 RemoteCertificate)가 포함된 속성을 제공합니다.

연결 작업이 완료되면 연결을 닫고 삭제해야 합니다. 즉시 닫기 위해 애플리케이션 계층 코드를 사용하는 QUIC 프로토콜은 RFC 9000 섹션 10.2 를 참조하세요. 이를 위해 애플리케이션 계층 코드가 포함된 CloseAsync(Int64, CancellationToken)를 호출할 수 있으며, 호출하지 않을 경우 DisposeAsync()는 DefaultCloseErrorCode에 제공된 코드를 사용합니다. 어느 쪽이든 연결된 모든 리소스를 완전히 해제하려면 연결 작업이 끝날 때 DisposeAsync()를 호출해야 합니다.

다음 `QuicConnection` 예제 코드를 참조하세요.

C#

```
using System.Net.Quic;

// First, check if QUIC is supported.
if (!QuicConnection.IsSupported)
{
    Console.WriteLine("QUIC is not supported, check for presence of libmsquic and support of TLS
1.3.");
    return;
}
```

```
// This represents the minimal configuration necessary to open a connection.
var clientConnectionOptions = new QuicClientConnectionOptions
{
    // End point of the server to connect to.
    RemoteEndPoint = listener.LocalEndPoint,

    // Used to abort stream if it's not properly closed by the user.
    // See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
    DefaultStreamErrorCode = 0x0A, // Protocol-dependent error code.

    // Used to close the connection if it's not done by the user.
    // See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
    DefaultCloseErrorCode = 0x0B, // Protocol-dependent error code.

    // Optionally set limits for inbound streams.
    MaxInboundUnidirectionalStreams = 10,
    MaxInboundBidirectionalStreams = 100,

    // Same options as for client side SslStream.
    ClientAuthenticationOptions = new SslClientAuthenticationOptions
    {
        // List of supported application protocols.
        ApplicationProtocols = [new SslApplicationProtocol("protocol-name")],
        // The name of the server the client is trying to connect to. Used for server certificate
validation.
        TargetHost = ""
    }
};

// Initialize, configure and connect to the server.
var connection = await QuicConnection.ConnectAsync(clientConnectionOptions);

Console.WriteLine($"Connected {connection.LocalEndPoint} --> {connection.RemoteEndPoint}");

// Open a bidirectional (can both read and write) outbound stream.
// Opening a stream reserves it but does not notify the peer or send any data. If you don't send
data, the peer
// won't be informed about the stream, which can cause AcceptInboundStreamAsync() to hang. To
avoid this, ensure
// you send data on the stream to properly initiate communication.
var outgoingStream = await connection.OpenOutboundStreamAsync(QuicStreamType.Bidirectional);

// Work with the outgoing stream ...

// To accept any stream on a client connection, at least one of MaxInboundBidirectionalStreams or
MaxInboundUnidirectionalStreams of QuicConnectionOptions must be set.
while (isRunning)
{
    // Accept an inbound stream.
    var incomingStream = await connection.AcceptInboundStreamAsync();

    // Work with the incoming stream ...
}

// Close the connection with the custom code.
await connection.CloseAsync(0x0C);

// Dispose the connection.
await connection.DisposeAsync();
```

`QuicConnection` 설계 방법에 대한 자세한 내용은 API 제안 을 참조하세요.

`QuicStream`

QuicStream은 QUIC 프로토콜에서 데이터를 보내고 받는 데 사용되는 실제 형식입니다. 일반 Stream에서 파생되며 이와 같이 사용할 수 있지만 QUIC 프로토콜과 관련된 몇 가지 기능도 제공합니다. 첫째, QUIC 스트림은 단방향일 수도 있고 양방향일 수도 있습니다. RFC 9000 섹션 2.1을 참조하세요. 양방향 스트림은 양쪽에서 데이터를 보내고 받을 수 있는 반면 단방향 스트림은 시작 쪽에서만 쓰고 수락된 스트림에서 읽을 수 있습니다. 각 피어는 각 형식의 동시 스트림이 허용할 수를 제한할 수 있습니다. MaxInboundBidirectionalStreams 및 MaxInboundUnidirectionalStreams를 참조하세요.

QUIC 스트림의 또 다른 특성은 스트림 작업 중에 쓰기 쪽을 명시적으로 닫는 기능입니다. CompleteWrites() 인수를 사용한 WriteAsync(ReadOnlyMemory<Byte>, Boolean, CancellationToken) 또는 `completeWrites` 오버로드를 참조하세요. 쓰기 쪽을 닫으면 피어가 더 이상 데이터가 도착하지 않음을 알 수 있지만 피어는 양방향 스트림의 경우 계속 전송할 수 있습니다. 이는 클라이언트가 요청을 보내고 쓰기 쪽을 닫아 서버가 요청 콘텐츠의 끝임을 알리는 HTTP 요청/응답 교환과 같은 시나리오에서 유용합니다. 서버는 그 후에도 응답을 보낼 수 있지만 클라이언트에서 더 이상 데이터가 도착하지 않는다는 것을 알고 있습니다. 오류가 발생한 경우 스트림의 쓰기 또는 읽기 단계를 중단할 수 있습니다. Abort(QuicAbortDirection, Int64)를 참조하세요.

> **참고 항목**
>
> 스트림을 열면 데이터를 보내지 않고 스트림만 예약됩니다. 이 접근 방식은 거의 빈 프레임의 전송을 방지하여 네트워크 사용량을 최적화하도록 설계되었습니다. 실제 데이터가 전송될 때까지 피어에 알림이 전송되지 않으므로, 피어의 입장에서는 스트림이 비활성 상태로 유지됩니다. 데이터를 보내지 않으면 피어가 스트림을 인식하지 못하므로 의미 있는 스트림을 기다리는 동안 `AcceptInboundStreamAsync()`이(가) 중단될 수 있습니다. 적절한 통신을 보장하려면 스트림을 연 후 데이터를 보내야 합니다.

각 스트림 유형에 대한 개별 메서드의 동작은 다음 표에 요약되어 있습니다(클라이언트와 서버 모두 스트림을 열고 수락할 수 있음).

테이블 확장

메서드	피어 스트림 개시	피어 수락 스트림
`CanRead`	*양방향*: `true` *단방향*: `false`	`true`
`CanWrite`	`true`	*양방향*: `true` *단방향*: `false`
`ReadAsync`	*양방향*: 데이터 읽기 *단방향*: `InvalidOperationException`	데이터 읽기
`WriteAsync`	데이터 보내기 => 피어 읽기를 통해 데이터 반환	*양방향*: 데이터 보내기 => 피어 읽기를 통해 데이터 반환 *단방향*: `InvalidOperationException`
`CompleteWrites`	쓰기 측면을 닫습니다=> 피어 읽기 작업이 0을 반환합니다	*양방향*: 쓰기 쪽 닫기 => 피어 읽기를 통해 0 반환 *단방향*: no-op
`Abort(QuicAbortDirection.Read)`	*양방향*: STOP_SENDING => 피어 쓰기를 통해 `QuicException(QuicError.OperationAborted)`	STOP_SENDING => 피어 쓰기 예외 발생 `QuicException(QuicError.OperationAborted)`

메서드	피어 스트림 개시	피어 수락 스트림
	throw *단방향*: 동작 없음	
`Abort(QuicAbortDirection.Write)`	RESET_STREAM => 피어 읽기 오류 발생 `QuicException(QuicError.OperationAborted)`	*양방향*: RESET_STREAM => 피어 읽기를 통해 `QuicException(QuicError.OperationAborted)` throw *단방향*: no-op

이러한 메서드를 기반으로 `QuicStream`은 스트림의 읽기 또는 쓰기 쪽이 닫혀 있을 때마다 알림을 받을 수 있는 두 가지 특수 속성인 ReadsClosed 및 WritesClosed를 제공합니다. 둘 다 성공 여부와 관계없이 해당하는 쪽이 닫히는 상태로 완료되는 `Task`를 반환하며, 이 경우 적절한 예외가 `Task`에 포함됩니다. 이러한 속성은 사용자 코드가 `ReadAsync` 또는 `WriteAsync` 호출을 실행하지 않고 스트림 쪽이 닫히는 것을 인식해야 하는 경우에 유용합니다.

마지막으로 스트림 작업이 완료되면 DisposeAsync()를 사용하여 삭제해야 합니다. 삭제 시 스트림 유형에 따라 읽기 및/또는 쓰기 쪽이 모두 닫혀 있는지 확인합니다. 스트림이 끝까지 제대로 읽히지 않은 경우, 처리에서는 `Abort(QuicAbortDirection.Read)`와 동등한 작업을 수행합니다. 그러나 스트림 쓰기 쪽이 닫혀 있지 않으면 `CompleteWrites`를 사용한 경우처럼 정상적으로 닫힙니다. 이러한 차이가 발생하는 이유는 일반 `Stream`을 사용하는 시나리오가 예상대로 동작하여 성공적인 경로로 이어지도록 하기 위한 것입니다. 다음 예제를 참조하세요.

C#

```
// Work done with all different types of streams.
async Task WorkWithStreamAsync(Stream stream)
{
    // This will dispose the stream at the end of the scope.
    await using (stream)
    {
        // Simple echo, read data and send them back.
        byte[] buffer = new byte[1024];
        int count = 0;
        // The loop stops when read returns 0 bytes as is common for all streams.
        while ((count = await stream.ReadAsync(buffer)) > 0)
        {
            await stream.WriteAsync(buffer.AsMemory(0, count));
        }
    }
}

// Open a QuicStream and pass to the common method.
var quicStream = await connection.OpenOutboundStreamAsync(QuicStreamType.Bidirectional);
await WorkWithStreamAsync(quicStream);
```

클라이언트 시나리오의 `QuicStream` 샘플 사용:

C#

```
// Consider connection from the connection example, open a bidirectional stream.
await using var stream = await connection.OpenOutboundStreamAsync(QuicStreamType.Bidirectional,
cancellationToken);

// Send some data.
await stream.WriteAsync(data, cancellationToken);
```

```
await stream.WriteAsync(data, cancellationToken);

// End the writing-side together with the last data.
await stream.WriteAsync(data, completeWrites: true, cancellationToken);
// Or separately.
stream.CompleteWrites();

// Read data until the end of stream.
while (await stream.ReadAsync(buffer, cancellationToken) > 0)
{
    // Handle buffer data...
}

// DisposeAsync called by await using at the top.
```

서버 시나리오의 `QuicStream` 사용 샘플:

C#

```
// Consider connection from the connection example, accept a stream.
await using var stream = await connection.AcceptInboundStreamAsync(cancellationToken);

if (stream.Type != QuicStreamType.Bidirectional)
{
    Console.WriteLine($"Expected bidirectional stream, got {stream.Type}");
    return;
}

// Read the data.
while (stream.ReadAsync(buffer, cancellationToken) > 0)
{
    // Handle buffer data...

    // Client completed the writes, the loop might be exited now without another ReadAsync.
    if (stream.ReadsCompleted.IsCompleted)
    {
        break;
    }
}

// Listen for Abort(QuicAbortDirection.Read) from the client.
var writesClosedTask = WritesClosedAsync(stream);
async ValueTask WritesClosedAsync(QuicStream stream)
{
    try
    {
        await stream.WritesClosed;
    }
    catch (Exception ex)
    {
        // Handle peer aborting our writing side ...
    }
}

// DisposeAsync called by await using at the top.
```

`QuicStream` 설계 방법에 대한 자세한 내용은 API 제안을 참조하세요.

참고 항목

- .NET의 네트워킹
- HTTP/3와 HttpClient
- System.Net.Quic
- QuicListener
- QuicConnection
- QuicStream

Last updated on 2026. 02. 24.

QUIC 구성 옵션

아티클 • 2025. 01. 22.

System.Net.Quic 라이브러리는 옵션 클래스를 사용하여 생성 및 초기화 전에 프로토콜 개체(QuicListener 및 QuicConnection)를 구성합니다. 이렇게 하려면 세 가지 옵션 클래스가 있습니다.

- QuicListenerOptions: QuicListener.ListenAsync(QuicListenerOptions, CancellationToken)를 시작하기 전에 QuicListener을 구성합니다.
- QuicClientConnectionOptions: QuicConnection.ConnectAsync(QuicClientConnectionOptions, CancellationToken)를 통해 설정하기 전에 나가는 QuicConnection을 구성하다
- QuicServerConnectionOptions: QuicListener.AcceptConnectionAsync(CancellationToken)로 전달되기 전에 들어오는 QuicConnection을 구성합니다.

모든 옵션 클래스를 증분 방식으로 설정할 수 있습니다. 즉, 생성자를 통해 속성을 초기화할 필요가 없으며 독립적으로 설정할 수 있습니다. 그러나 새 수신기나 연결을 구성하기 위해 사용되는 순간, 옵션의 유효성이 검사되며 필수 값이 누락되었거나 값이 잘못 구성된 경우에 적절한 유형의 ArgumentException가 던져집니다. 예를 들어, 필수 QuicConnectionOptions.DefaultStreamErrorCode가 설정되지 않은 경우 ConnectAsync(QuicClientConnectionOptions, CancellationToken)을 호출하면 ArgumentOutOfRangeException가 발생합니다.

퀵리스너옵션

새 QuicListener를 시작할 때 QuicListener.ListenAsync(QuicListenerOptions, CancellationToken)에서 QuicListenerOptions이 사용됩니다. 개별 구성 속성은 다음과 같습니다.

- 애플리케이션 프로토콜
- ConnectionOptionsCallback
- ListenBacklog
- ListenEndPoint

애플리케이션 프로토콜

ApplicationProtocols 서버(RFC 7301 - ALPN)에서 허용하는 애플리케이션 프로토콜을 정의합니다. 관련이 없는 여러 프로토콜에 대한 여러 값을 포함할 수 있습니다. 새 연결을 수락하는 과정에서 수신기는 들어오는 각 연결에 대해 하나의 특정 프로토콜을 좁히거나

선택할 수 있습니다. QuicListenerOptions.ConnectionOptionsCallback참조하세요. **이 속성은 필수이며 하나 이상의 값을 포함해야 합니다.**

커넥션옵션콜백

ConnectionOptionsCallback은(는) 들어오는 연결에 대해 QuicServerConnectionOptions을(를) 선택하는 대리자입니다. 이 함수는 클라이언트에서 요청한 서버 이름(RFC 6066 - SNI)을 포함하는 QuicConnection 및 SslClientHelloInfo 부분적으로 초기화된 인스턴스가 제공됩니다. 들어오는 각 연결에 대해 대리자가 호출됩니다. 제공된 클라이언트 정보에 따라 다른 옵션을 반환하거나 매번 동일한 옵션 인스턴스를 안전하게 반환할 수 있습니다. 대리자의 용도와 형태는 의도적으로 SslStream.AuthenticateAsServerAsync(ServerOptionsSelectionCallback, Object, CancellationToken)에서 사용된 ServerOptionsSelectionCallback와 비슷하게 설정되었습니다. **이 속성은 필수입니다.**

듣기미완료목록

ListenBacklog 추가 연결이 거부되기 전에 수신기가 보유할 수 있는 들어오는 연결 수를 결정합니다. 연결이 실패하거나 큐에서 대기하는 동안 연결이 종료되는 경우에도 연결을 설정하려는 모든 시도가 중요합니다. 이 제한에 대한 새 연결 수를 설정하는 프로세스도 진행 중입니다. 연결 또는 연결 시도는 QuicListener.AcceptConnectionAsync(CancellationToken)을 통해 검색될 때까지 집계됩니다. 백로그 제한의 목적은 서버가 처리할 수 있는 것보다 더 많은 들어오는 연결에 의해 과부하되는 것을 방지하는 것입니다. **이 속성은 선택 사항이며 기본값은 512입니다.**

리스닝 엔드포인트

ListenEndPoint 수신기가 새 연결을 수락할 IP 주소 및 포트를 포함합니다. 기본 구현으로 인해 수신기인 `MsQuic` 여기에 지정된 항목에 관계없이 항상 이중 스택 와일드카드 소켓에 바인딩됩니다. 이로 인해 특히 HTTP/1.1 및 HTTP/2 사례와 같은 일반 TCP 소켓과 비교하여 예기치 않은 동작이 발생할 수 있습니다. 자세한 내용은 QUIC 문제 해결 가이드참조하세요. **이 속성은 필수입니다.**

퀵 연결 옵션 (QuicConnectionOptions)

QuicConnectionOptions 옵션은 QuicClientConnectionOptionsQuicServerConnectionOptions간에 공유됩니다. 추상 기본 클래스이며 자체적으로 사용할 수 없습니다. 여기에는 다음 속성이 포함됩니다.

- defaultCloseErrorCode

- DefaultStreamErrorCode
- 핸드셰이크시간초과
- 유휴 시간 초과
- 초기 수신 창 크기
- KeepAlive간격
- MaxInboundBidirectionalStreams
- 최대수신단방향스트림
- StreamCapacityCallback

DefaultCloseErrorCode (기본 닫기 오류 코드)

DefaultCloseErrorCode QuicConnection.CloseAsync(Int64, CancellationToken)호출하지 않고 연결이 삭제될 때 사용됩니다. 연결을 닫는 애플리케이션 수준 이유를 제공하기 위해 QUIC 프로토콜이 필요합니다(RFC 9000 - 연결 닫기). QuicConnection 연결을 삭제하기 전에 애플리케이션 코드가 CloseAsync(Int64, CancellationToken) 호출하도록 강제할 방법이 없습니다. 이 경우 연결에서 사용할 오류 코드를 알아야 합니다. **이 속성은 필수입니다.**

DefaultStreamErrorCode (기본 스트림 오류 코드)

DefaultStreamErrorCode 모든 데이터를 읽기 전에 스트림이 삭제될 때 사용됩니다. QUIC 스트림을 통해 데이터를 수신하는 경우 애플리케이션은 모든 데이터를 사용하거나 그렇지 않은 경우 읽기 쪽을 중단해야 합니다. 연결 닫기와 마찬가지로 QUIC 프로토콜에는 읽기 쪽을 중단하는 애플리케이션 수준 이유가 필요합니다(rfC 9000 - 보내기 중지). **이 속성은 필수입니다.**

핸드셰이크타임아웃

HandshakeTimeout 연결을 완전히 설정해야 하는 시간 제한을 설정합니다. 그렇지 않으면 중단됩니다. 이 값을 InfiniteTimeSpan 설정할 수 있지만 권장되지 않습니다. 연결 시도가 무기한 멈춰 있을 수 있으며 QuicListener을 중지하는 방법 외에는 이를 취소할 방법이 없어. **이 속성은 선택 사항이며 기본값은 10초입니다.**

IdleTimeout

지정된 IdleTimeout이상 연결이 비활성 상태이면 연결이 끊어집니다. 이 옵션은 QUIC 프로토콜 사양(RFC 9000 - 유휴 시간 제한)의 일부이며 연결 핸드셰이크 중에 피어로 전송됩니다. 그런 다음 연결은 자신의 시간 제한과 피어의 유휴 시간 제한 중 더 작은 값을 선택하여 사용합니다. 따라서 유휴 시간 제한 시 이 옵션이 설정된 것보다 빨리 연결을 닫을 수 있습니다. **이 속성은 선택 사항이며 기본값은 30초인 MsQuic을 기반으로 합니다.**

초기 수신 창 크기들

InitialReceiveWindowSizes 처음에는 연결 및/또는 스트림에서 수신할 수 있는 데이터의 양을 제한하는 값 집합을 지정합니다. QUIC 프로토콜은 개별 스트림을 통해 전송할 수 있는 데이터의 양을 제한하고 전체 연결(rfC 9000- 데이터 흐름 제어)에 대해 누적적으로 전송할 수 있는 양을 제한하는 메커니즘을 정의합니다. 이러한 제한은 애플리케이션이 데이터 사용을 시작하기 전에만 적용됩니다. 그 후 `MsQuic` 애플리케이션에서 읽는 속도에 따라 수신 창의 크기를 지속적으로 조정합니다. 이 속성은 다음 옵션을 포함하는 QuicReceiveWindowSizes 형식입니다.

- Connection: 이 연결에 속하는 모든 스트림에서 수신된 데이터에 대한 누적 제한입니다.
- LocallyInitiatedBidirectionalStream: 나가는 양방향 스트림에서 수신된 데이터에 대한 제한입니다.
- RemotelyInitiatedBidirectionalStream: 들어오는 양방향 스트림에서 수신된 데이터에 대한 제한입니다.
- UnidirectionalStream: 들어오는 단방향 스트림에서 수신된 데이터에 대한 제한입니다.

이러한 값은 2의 거듭제곱인 음수가 아닌 정수여야 합니다. 이는 `MsQuic`에서 기인한 제한 사항입니다. 이러한 값을 0으로 설정하면 기본적으로 특정 스트림 또는 연결 전체에서 데이터를 수신하지 않습니다. **이 속성은 선택 사항이며, 기본값은 스트림의 경우 64KB, 연결의 경우 64MB입니다.**

KeepAliveInterval

KeepAliveInterval 연결을 활성 상태로 유지하고 IdleTimeout 닫히는 것을 방지하기 위해 PING 프레임이 전송되는지 여부와 빈도를 결정합니다(RFC 9000 - PING 프레임). 이 속성을 설정할 때는 RFC 9000 - 유휴 시간 제한 지연에 대한 권장 사항을 고려하십시오. 값을 너무 낮게 설정하면 성능에 부정적인 영향을 미칠 수 있습니다. 또한 유휴 시간 제한에 너무 가깝게 속성을 설정하면 연결이 닫힐 수 있습니다. **이 속성은 선택 사항이며 기본값은 InfiniteTimeSpan PING가 전송되지 않음을 의미합니다.**

MaxInboundBidirectionalStreams

MaxInboundBidirectionalStreams 연결에서 허용할 최대 동시 활성 양방향 스트림 수를 결정합니다. 이는 QUIC 사양이 동시성 처리(RFC 9000 - 동시성 제어)를 정의하는 방식과 다릅니다. QUIC 프로토콜은 연결 수명 동안 누적 스트림 수를 계산하고, 계속 증가하는 제한을 사용하여 이미 닫힌 스트림(rfC 9000 -MAX_STREAMS 프레임)을 포함하여 연결에서 허용하는 전체 스트림 수를 결정합니다. 이 속성은 애플리케이션에서 동시 스트림 제한만 지정하고 `MsQuic` 이 제한을 해당 `MAX_STREAMS` 프레임으로 변환하도록 간소화합

니다. **이 속성은 선택 사항이며, 기본값은 클라이언트 연결의 경우 0이고 서버 연결의 경우 100입니다.**

MaxInboundUnidirectionalStreams

MaxInboundUnidirectionalStreams 연결에서 허용할 최대 동시 활성 단방향 스트림 수를 결정합니다. 이는 QUIC 사양에서 처리 스트림 동시성(rfC 9000- 동시성 제어)을 정의하는 방법과 다릅니다. QUIC 프로토콜은 연결 수명 동안 누적 스트림 수를 계산하고, 계속 증가하는 제한을 사용하여 이미 닫힌 스트림(rfC 9000 -MAX_STREAMS 프레임)을 포함하여 연결에서 허용하는 전체 스트림 수를 결정합니다. 이 속성은 애플리케이션에서 동시 스트림 제한만 지정하고 `MsQuic` 이 제한을 해당 `MAX_STREAMS` 프레임으로 변환하도록 간소화합니다. **이 속성은 선택 사항이며, 기본값은 클라이언트 연결의 경우 0이고 서버 연결의 경우 10입니다.**

스트림 용량 콜백

StreamCapacityCallback은 피어가 `MAX_STREAMS`을 통해 새로운 스트림 용량을 해제할 때마다 호출되는 콜백으로, 그 결과 현재 용량이 0을 초과하게 됩니다. 콜백 인수에 제공된 값은 용량의 증가량을 나타냅니다. 따라서 콜백에서 제공된 모든 값의 합계는 `MAX_STREAMS`로부터 받은 마지막 값(RFC 9000 - MAX_STREAMS 프레임)에 해당합니다. 이 콜백은 SocketsHttpHandler.EnableMultipleHttp3Connections 기능을 지원하도록 설계되었으며 다음과 같은 몇 가지 주의 사항이 있습니다.

- 언제든 실제 용량을 알 수 있도록 열려 있는 모든 스트림과 열린 스트림을 추적하는 것은 애플리케이션의 책임입니다.
- 콜백은 병렬로 호출될 수 있으므로 스트림 계산을 중심으로 동기화를 제대로 처리하는 것은 애플리케이션의 맡입니다.
- 첫 번째 호출(초기 용량 포함)은 QuicConnection 인스턴스가 QuicConnection.ConnectAsync(QuicClientConnectionOptions, CancellationToken) 또는 QuicListener.AcceptConnectionAsync(CancellationToken)통해 전달되기 전에 발생할 수 있습니다.

다음 간소화된 시나리오는 스트림 열기 및 콜백의 동작을 캡처합니다.

1. 클라이언트는 다음을 통해 서버에 대한 연결을 시작합니다.

```
C#

var client = await QuicConnection.ConnectAsync(new
QuicClientConnectionOptions
{
    ...
    StreamCapacityCallback = (connection, args) =>
```

```
        Console.WriteLine($"{connection} stream capacity increased by:
unidi += {args.UnidirectionalIncrement}, bidi +=
{args.BidirectionalIncrement}")
};
```

2. 서버는 단방향 스트림에 대한 스트림 제한 `2` 및 양방향 스트림에 대한 스트림 제한 `0`을 클라이언트에 초기 설정으로 보냅니다.

3. 클라이언트의 `StreamCapacityCallback` 호출되고 인쇄됩니다.

```
text

[conn][0x58575BF805B0] stream capacity increased by: unidi += 2, bidi
+= 0
```

4. `ConnectAsync`에 대한 클라이언트 호출은 `[conn][0x58575BF805B0]` 연결로 반환됩니다.

5. 클라이언트는 몇 가지 스트림을 열려고 시도합니다.

```
C#

var stream1 = await
connection.OpenOutboundStreamAsync(QuicStreamType.Unidirectional);
var stream2 = await
connection.OpenOutboundStreamAsync(QuicStreamType.Unidirectional);
// The following call will get suspended because the stream's limit has
been reached.
var taskStream3 =
connection.OpenOutboundStreamAsync(QuicStreamType.Unidirectional);
```

6. 클라이언트는 처음 두 스트림을 완료하고 닫습니다.

```
C#

await stream1.WriteAsync(data, completeWrites: true);
await stream1.DisposeAsync();
await stream2.WriteAsync(data, completeWrites: true);
await stream2.DisposeAsync();
Console.WriteLine($"Stream 3 {(taskStream3.IsCompleted ? "opened" :
"pending")}");
```

7. 클라이언트는 다음을 인쇄합니다.

```
text
```

```
Stream 3 pending
```

8. 서버는 처음 두 스트림을 처리한 후 `2` 추가 용량을 제공합니다.

9. 클라이언트에서 두 가지 일이 발생합니다. 먼저 세 번째 스트림이 열립니다.

```C#
var stream3 = await taskStream3;
```

그런 다음 클라이언트의 `StreamCapacityCallback` 다시 호출되고 인쇄됩니다.

```text
[conn][0x58575BF805B0] stream capacity increased by: unidi += 2, bidi
+= 0
```

이 속성은 선택 사항입니다.

QuicServerConnectionOptions

QuicServerConnectionOptions 옵션은 서버 쪽 연결에 따라 다릅니다.
QuicConnectionOptions상속된 속성 외에도 다음이 포함됩니다.

- 서버 인증 옵션

서버 인증 옵션

ServerAuthenticationOptions 서버 연결에 대한 TLS 설정을 포함합니다. 옵션은 SslStream.AuthenticateAsServer(SslServerAuthenticationOptions) 및 SslStream.AuthenticateAsServerAsync(SslServerAuthenticationOptions, CancellationToken)사용되는 것과 동일합니다. QUIC 서버의 경우 다음과 같은 경우 SslServerAuthenticationOptions 유효합니다.

- 다음 속성 중 하나 이상이 유효한 인증서를 반환합니다. ServerCertificateSelectionCallback, ServerCertificateContext, ServerCertificate.
- ApplicationProtocols에서 하나 이상의 애플리케이션 프로토콜이 정의됩니다.
- 변경된 경우 EncryptionPolicy가 NoEncryption로 설정되지 않습니다(기본값은 RequireEncryption).
- 설정된 경우, CipherSuitesPolicy에는 TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256 중 하나 이상이 포

함됩니다(기본값은 `null`이고, `MsQuic`는 OS에서 지원하는 모든 QUIC 호환 암호화 스위트를 사용할 수 있습니다).

이 속성은 필수이며 나열된 조건을 충족해야 합니다.

QuicClientConnectionOptions

QuicClientConnectionOptions 옵션은 클라이언트 쪽 연결과 관련이 있습니다. QuicConnectionOptions상속된 속성 외에도 다음이 포함됩니다.

- ClientAuthenticationOptions
- 로컬 엔드포인트
- RemoteEndPoint

클라이언트 인증 옵션

ClientAuthenticationOptions 클라이언트 연결에 대한 TLS 설정을 포함합니다. 옵션은 SslStream.AuthenticateAsClient(SslClientAuthenticationOptions) 및 SslStream.AuthenticateAsClientAsync(SslClientAuthenticationOptions, CancellationToken)사용되는 것과 동일합니다. QUIC 클라이언트의 경우 다음과 같은 경우 SslClientAuthenticationOptions 유효합니다.

- ApplicationProtocols에서 적어도 하나의 애플리케이션 프로토콜이 정의됩니다.
- 변경된 경우 EncryptionPolicy가 NoEncryption로 설정되지 않습니다(기본값은 RequireEncryption).
- 설정된 경우, CipherSuitesPolicy에는 TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256 중 적어도 하나가 포함됩니다. (기본값은 `null`이며, `MsQuic`가 OS에서 지원하는 모든 QUIC 호환 암호 그룹을 사용할 수 있도록 합니다.)

이 속성은 필수이며 나열된 조건을 충족해야 합니다.

LocalEndPoint (로컬 엔드포인트)

LocalEndPoint 클라이언트 연결이 바인딩할 IP 주소 및 포트를 포함합니다. 지정하지 않으면 OS는 IP 주소와 포트를 할당합니다. **이 속성은 선택 사항입니다.**

RemoteEndPoint

RemoteEndPoint는 연결이 설정되는 피어의 DnsEndPoint일 수도 있고, IPEndPoint일 수도 있습니다. DnsEndPoint일 경우, Dns.GetHostAddressesAsync(String,

CancellationToken)이 반환한 첫 번째 IP 주소가 사용됩니다. **이 속성은 필수입니다.**

.NET에서 QUIC 문제 해결 방법

2025. 06. 17.

이 문서에서는 .NET에서 QUIC와 관련된 가장 일반적인 문제를 진단하는 방법을 알아봅니다.

라이브러리는 `System.Net.Quic` 오픈 소스 QUIC 구현 MsQuic를 기반으로합니다. 이 때문에 동작은 일반적인 소켓과 다르며 때로는 의도적으로 다릅니다. 또한 UDP 프로토콜을 기반으로 하며 TCP와 정확히 동일한 환경을 제공하지 않습니다.

수신기가 실행 중이지만 데이터를 수신하지 않음

수신기가 실행 중이지만 데이터를 수신하지 않는 경우 동일한 포트에서 수신 대기하는 다른 프로세스로 인해 발생할 수 있습니다. 실행되는 포트를 사용하는 프로세스를 확인하려면 다음을 수행합니다.

```Bash
sudo ss -tulpw
```

이 동작은 `MsQuic`을 사용하여 더 나은 성능을 달성하도록 `SO_REUSEPORT`가 설계된 것입니다. 자세한 내용은 ListenerStart 설명서 및 원래 문제 dotnet/runtime#59382를 참조하세요.

> **참고**
>
> 이 문제는 MsQuic이 포트 예약을 시도하는 Windows에서는 발생하지 않습니다. 이렇게 하면 동일한 포트에서 두 번째 수신기를 열려는 애플리케이션이 시작되지 않습니다.

QuicListener는 항상 ANY 주소에서 수신 대기합니다.

속성을 통해 `ListenEndpoint` 특정 주소가 제공되더라도 QuicListener는 여전히 이중 스택 와일드카드 소켓을 엽니다. 이 동작은 `MsQuic`에 의해 설계되었습니다. 수신 대기 IP 주소는 여전히 내부에서 `MsQuic`필터링을 수행하는 데 사용되고 있습니다. 자세한 내용은 ListenerStart 설명서 및 원래 문제 [dotnet/runtime#92812]를 참조하세요.

클라이언트가 예기치 않은 ALPN 오류를 수신합니다.

클라이언트는 연결을 시도하지만 서버와 동일한 ALPN을 사용했음에도 불구하고 수신합니다 `Application layer protocol negotiation error was encountered` .

애플리케이션이 지정한 내용에 관계없이 수신기는 항상 이중 모드 와일드카드 주소에 바인딩됩니다. 그런 다음 IP 주소 및 ALPN을 기준으로 들어오는 연결 요청과 일치시킵니다. 일치하는 항목이 없으면 위에서 언급한 오류를 보고합니다. 따라서 수신 대기 IP 주소와 연결이 일치하지 않으면 ALPN 오류가 발생합니다.

이 오류를 방지하려면 수신기가 시작된 주소와 동일한 주소에 연결해야 합니다. 예를 들어 수신기의 수신 대기 주소를 인쇄합니다.

C#

```
await using var listener = QuicListener.ListenAsync(new() /* appropriate options
*/);
Console.WriteLine(listener.LocalEndPoint);
```

이 문제는 이 문제 dotnet/runtime#85412와 같은 여러 다른 시나리오에서 발생할 수 있습니다. 서버가 `Loopback` 주소(`127.0.0.1`)로 시작되었으며, 동일한 컴퓨터에서 실행될 때 모든 것이 작동했습니다. 그러나 클라이언트가 다른 주소에서 연결하려고 하면 주소가 서버 루프백 주소와 일치하지 않으며 ALPN 오류로 거부되었습니다.

> **참고**
>
> 수신기가 MsQuic을 사용하는 경우(예: .NET `QuicListener`을 통해) 발생합니다.

수신기가 비활성화되었음에도 불구하고 IPv6에 대해 시작하는 데 성공함

IPv6을 사용하지 않도록 설정 `QuicListener.ListenAsync` 했음에도 불구하고 IPv6 주소로 성공합니다. 이는 와일드카드 주소에 바인딩되므로 `MsQuic` 이전 문제와 관련이 있으므로 성공적으로 수행됩니다. 따라서 수신기가 시작되지만 연결할 수 없습니다. 이러한 경우 throw되는 소켓의 동작 차이입니다.

예를 들어 Linux에서 IPv6 모듈의 상태를 확인하여 IPv6이 사용하도록 설정되었는지 확인하는 방법에는 여러 가지가 있습니다.

Bash

```
cat /sys/module/ipv6/parameters/disable

# 0 - IPv6 is enable
# 1 - IPv6 is disabled
```

위에서 설명한 대로 여기서 MsQuic 동작은 의도적입니다. 그러나 이 특정 문제는 나중에 .NET 쪽에서 완화될 수 있습니다. 자세한 내용은 dotnet/runtime#75343 을 참조하세요.

수신기가 `QUIC_STATUS_ADDRESS_IN_USE` 오류로 인해 시작하지 못함

Windows 관련 문제입니다. 수신기는 특정 주소 및 포트에서 다른 프로세스가 실행되지 않음에도 불구하고 `QuicException` 오류와 함께 `QUIC_STATUS_ADDRESS_IN_USE`를 throw합니다. 이 오류는 수신기가 수신 대기하는 것과 동일한 포트에 대해 정의된 포트 제외 범위로 인해 발생합니다. 제외 범위를 확인하려면 다음을 실행합니다.

```batch
netsh.exe int ip show excludedportrange protocol=udp
```

제외된 범위의 포트에 바인딩을 시도했을 때 실패하는 것은 예상되는 동작입니다. 이러한 이유로 이 문제를 해결할 즉각적인 계획은 없습니다. 자세한 내용은 dotnet/runtime#71518 에서 찾을 수 있습니다.

참고하십시오

- MsQuic 문제 해결 가이드

.NET의 네트워크 원격 측정

아티클 • 2025. 02. 01.

.NET 네트워킹 스택은 다양한 계층에서 계측됩니다. .NET은 메트릭, 분산 추적, 이벤트 카운터 및 이벤트를 사용하여 HTTP 요청의 수명 동안 정확한 타이밍을 수집하는 옵션을 제공합니다.

- **네트워킹 메트릭**: .NET 8부터 최신 System.Diagnostics.Metrics API사용하여 HTTP 및 DNS(이름 확인) 구성 요소를 계측합니다. 이러한 메트릭은 OpenTelemetry 협력하여 설계되었으며 다양한 모니터링 도구로 내보낼 수 있습니다.
- **분산 추적**: `HttpClient` 계측되어 분산 추적 활동(범위라고도 함)을 내보낸다.
- **네트워킹 이벤트**: 이벤트는 정확한 타임스탬프로 디버그 및 추적 정보를 제공합니다.
- **네트워킹 이벤트 카운터**: 모든 네트워킹 구성 요소는 EventCounters API를 사용하여 실시간 성능 메트릭을 게시하도록 계측됩니다.

.NET의 네트워킹 메트릭

측정지표는 시간에 따라 보고되는 수치적 측정값입니다. 일반적으로 앱의 상태를 모니터링하고 경고를 생성하는 데 사용됩니다.

.NET 8부터 `System.Net.Http` 및 `System.Net.NameResolution` 구성 요소가 계측되어, .NET의 새로운 System.Diagnostics.Metrics API를 사용하여 메트릭을 게시합니다. 이러한 메트릭은 OpenTelemetry 협력하여 표준과 일치하고 Prometheus 및 Grafana 같은 인기 있는 도구와 잘 작동하도록 설계되었습니다. 또한 다차원 , 즉 측정값은 태그(특성 또는 레이블이라고도 함)라는 키-값 쌍과 연결됩니다. 태그를 사용하면 측정값을 분류하여 분석에 도움이 됩니다.

> **팁**
>
> 모든 기본 제공 도구와 그 특성의 포괄적인 목록은 **System.Net 메트릭**을 참조하세요.

System.Net 메트릭 수집

기본 제공 메트릭 계측을 활용하려면 이러한 메트릭을 수집하도록 .NET 앱을 구성해야 합니다. 이는 일반적으로 외부 스토리지 및 분석을 위해 모니터링 시스템으로 변환하는 것을 의미합니다.

.NET에서 네트워킹 메트릭을 수집하는 방법에는 여러 가지가 있습니다.

- 간단하고 독립적인 예제를 사용하여 빠른 개요를 보려면 dotnet-counters 메트릭 수집을 참조하세요.
- **프로덕션 시간** 메트릭 수집 및 모니터링의 경우 OpenTelemetry 및 Prometheus Grafana를 사용하거나 Azure Monitor Application Insights 수 있습니다. 그러나 이러한 도구는 복잡성 때문에 개발 시 사용하기가 불편할 수 있습니다.
- **개발 시간** 메트릭 수집 및 문제 해결의 경우 애플리케이션에서 메트릭 및 분산 추적을 시작하고 로컬에서 문제를 진단하는 간단하지만 확장 가능한 방법을 제공하는 Aspire를 사용하는 것이 좋습니다.
- 또한 Aspire 오케스트레이션 없이 Aspire 서비스 기본 프로젝트를 재사용하여 ASP.NET 프로젝트에 OpenTelemetry 구성 추적 및 메트릭 API를 도입하는 편리한 방법입니다.

dotnet-counters를 사용하여 메트릭 수집

dotnet-counters .NET 메트릭의 임시 검사 및 첫 번째 수준 성능 조사를 위한 플랫폼 간 명령줄 도구입니다.

이 자습서를 위해 다양한 엔드포인트에 HTTP 요청을 병렬로 보내는 앱을 만듭니다.

.NET CLI

```
dotnet new console -o HelloBuiltinMetrics
cd ..\HelloBuiltinMetrics
```

`Program.cs` 내용을 다음 샘플 코드로 바꿉니다.

C#

```
using System.Net;

string[] uris = ["http://example.com", "http://httpbin.org/get",
"https://example.com", "https://httpbin.org/get"];
using HttpClient client = new()
{
    DefaultRequestVersion = HttpVersion.Version20
};

Console.WriteLine("Press any key to start.");
Console.ReadKey();

while (!Console.KeyAvailable)
{
    await Parallel.ForAsync(0, Random.Shared.Next(20), async (_, ct) =>
    {
        string uri = uris[Random.Shared.Next(uris.Length)];
        try
        {
            byte[] bytes = await client.GetByteArrayAsync(uri, ct);
            await Console.Out.WriteLineAsync($"{uri} - received {bytes.Length}
bytes.");
        }
        catch { await Console.Out.WriteLineAsync($"{uri} - failed."); }
    });
}
```

`dotnet-counters` 설치되어 있는지 확인합니다.

.NET CLI

```
dotnet tool install --global dotnet-counters
```

HelloBuiltinMetrics 앱을 시작합니다.

.NET CLI

```
dotnet run -c Release
```

별도의 CLI 창에서 `dotnet-counters` 시작하고 감시할 프로세스 이름과 미터를 지정한 다음 HelloBuiltinMetrics 앱에서 키를 눌러 요청 보내기를 시작합니다. 측정값이 착륙하기 시작하자마자 `dotnet-counters` 계속해서 최신 숫자로 콘솔을 새로 고칩니다.

콘솔

```
dotnet-counters monitor --counters System.Net.Http,System.Net.NameResolution -n
HelloBuiltinMetrics
```

dotnet-counters 출력

```
Press p to pause, r to resume, q to quit.
    Status: Running
Warning: Histogram tracking limit (10) reached. Not all data is being shown.
The limit can be changed with --maxHistograms but will use more memory in the target process.

[System.Net.Http]
    http.client.active_requests ({request})
        http.request.method=GET,server.address=example.com,url.s          0
        http.request.method=GET,server.address=example.com,url.s          0
        http.request.method=GET,server.address=httpbin.org,url.s          2
        http.request.method=GET,server.address=httpbin.org,url.s          2
    http.client.open_connections ({connection})
        http.connection.state=active,network.peer.address=::ffff          2
        http.connection.state=active,network.peer.address=::ffff          1
        http.connection.state=active,network.peer.address=2600:1          0
        http.connection.state=active,network.peer.address=2600:1          0
        http.connection.state=idle,network.peer.address=::ffff:5          5
        http.connection.state=idle,network.peer.address=::ffff:5          0
        http.connection.state=idle,network.peer.address=2600:140          7
        http.connection.state=idle,network.peer.address=2600:140          1
    http.client.request.time_in_queue (s)
        http.request.method=GET,network.protocol.version=1.1,ser          0.166
        http.request.method=GET,network.protocol.version=1.1,ser          0.166
        http.request.method=GET,network.protocol.version=1.1,ser          0.166
        http.request.method=GET,network.protocol.version=1.1,ser          0.147
        http.request.method=GET,network.protocol.version=1.1,ser          0.147
        http.request.method=GET,network.protocol.version=1.1,ser          0.147
[System.Net.NameResolution]
    dns.lookup.duration (s)
        dns.question.name=example.com,Percentile=50                       0.002
        dns.question.name=example.com,Percentile=95                       0.002
        dns.question.name=example.com,Percentile=99                       0.002
        dns.question.name=httpbin.org,Percentile=50                       0.027
        dns.question.name=httpbin.org,Percentile=95                       0.028
        dns.question.name=httpbin.org,Percentile=99                       0.028
```

dotnet-counters output

Aspire를 사용하여 메트릭 수집

ASP.NET 애플리케이션에서 추적 및 메트릭을 수집하는 간단한 방법은 Aspire를 사용하는 것입니다. Aspire는 분산 애플리케이션을 쉽게 만들고 사용할 수 있도록 .NET에 대한 확장 집합입니다. Aspire를 사용할 때의 이점 중 하나는 .NET용 OpenTelemetry 라이브러리를 사용하여 원격 분석이 기본 제공된다는 것입니다.

Aspire의 기본 프로젝트 템플릿에는 프로젝트가 포함되어 있습니다 `ServiceDefaults` . Aspire 솔루션의 각 서비스에는 서비스 기본값 프로젝트에 대한 참조가 있습니다. 서비스는 OTel을 설정하고 구성하는 데 사용합니다.

서비스 기본값 프로젝트 템플릿에는 OTel SDK, ASP.NET, HttpClient 및 런타임 계측 패키지가 포함됩니다. 이러한 계측 구성 요소는 Extensions.cs 파일에 구성됩니다. Aspire 대시보드에서 원격 분석 시각화를 지원하기 위해 서비스 기본값 프로젝트에는 기본적으로 OTLP 내보내기도 포함됩니다.

Aspire 대시보드는 로컬 디버그 주기에 원격 분석 관찰을 제공하도록 설계되어 개발자가 애플리케이션에서 원격 분석을 생성하도록 할 수 있습니다. 원격 분석 시각화는 이러한 애플리케이션을 로컬로 진단하는 데도 도움이 됩니다. 서비스 간의 호출을 관찰할 수 있다는 것은 프로덕션에서처럼 디버그 시 유용합니다. Aspire 대시보드는 Visual Studio의 Project 또는 `AppHost` 명령줄에서 `dotnet run` 프로젝트를 `AppHost`로 실행할 때 자동으로 시작됩니다.

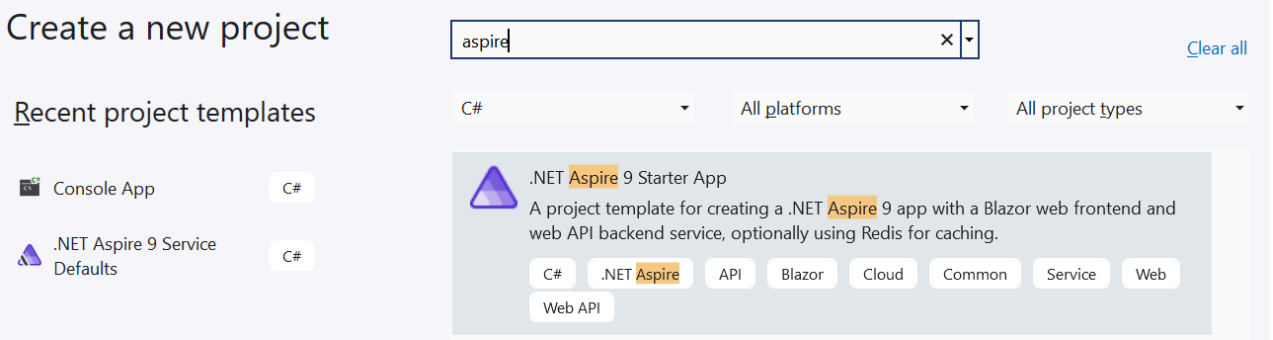
빠른 가이드

1. **Aspire 9 Starter 앱**을 `dotnet new`를 사용하여 만듭니다.

   .NET CLI

   ```
   dotnet new aspire-starter-9 --output AspireDemo
   ```

   또는 Visual Studio에서 새 프로젝트를 만들고 **Aspire 9 Starter 앱** 템플릿을 선택합니다.

2. `Extensions.cs` 프로젝트에서 `ServiceDefaults` 열고 `ConfigureOpenTelemetry` 메서드로 스크롤합니다. 네트워킹 미터를 구독하는 `AddHttpClientInstrumentation()` 호출을 확인합니다.

   C#

   ```
   .WithMetrics(metrics =>
   {
       metrics.AddAspNetCoreInstrumentation()
           .AddHttpClientInstrumentation()
           .AddRuntimeInstrumentation();
   })
   ```

.NET 8 및 이후 버전에서는 `AddHttpClientInstrumentation()`를 수동 미터 구독으로 바꿀 수 있음을 유의하십시오.

C#

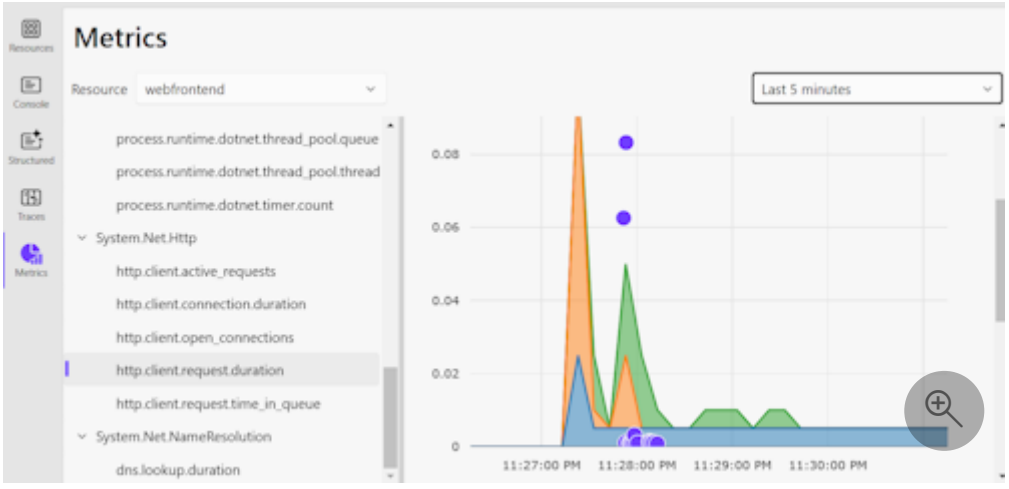
```
.WithMetrics(metrics =>
{
    metrics.AddAspNetCoreInstrumentation()
        .AddMeter("System.Net.Http")
        .AddMeter("System.Net.NameResolution")
        .AddRuntimeInstrumentation();
})
```

3. `AppHost` 프로젝트를 실행합니다. 이는 Aspire 대시보드를 실행합니다.

4. `webfrontend` 앱의 날씨 페이지로 이동하여 `HttpClient`대한 `apiservice` 요청을 생성합니다. 여러 요청을 보내려면 페이지를 여러 번 새로 고치세요.

5. 대시보드로 돌아가서 **메트릭** 페이지로 이동하고 `webfrontend` 리소스를 선택합니다. 아래로 스크롤하면 기본 제공 `System.Net` 메트릭을 찾아볼 수 있습니다.

   Aspire 대시보드의 네트워킹 메트릭



Aspire에 대한 자세한 내용은 다음을 참조하세요.

- 열망 개요
- Aspire 원격 분석
- Aspire 대시보드

Aspire의 오케스트레이션을 사용하지 않고 서비스 기본값 프로젝트를 재사용

Aspire Service Defaults 프로젝트는 오케스트레이션에 AppHost와 같은 *나머지 Aspire를 사용하지 않더라도* ASP.NET 프로젝트에 OTel을 쉽게 구성할 수 있는 방법을 제공합니다. 서비스 기본

값 프로젝트는 Visual Studio 또는 `dotnet new`통해 프로젝트 템플릿으로 사용할 수 있습니다. OTel을 구성하고 OTLP 내보내기를 설정합니다. 그런 다음 OTel 환경 변수 사용하여 원격 분석을 보내고 애플리케이션에 대한 리소스 속성을 제공하도록 OTLP 엔드포인트를 구성할 수 있습니다.

Aspire 외부에서 *ServiceDefaults*를 사용하는 단계는 다음과 같습니다.

1. Visual Studio에서 새 프로젝트 추가를 사용하여 *ServiceDefaults* 프로젝트를 솔루션에 추가하거나 `dotnet new`사용합니다.

   .NET CLI

   ```
   dotnet new aspire-servicedefaults --output ServiceDefaults
   ```

2. ASP.NET 애플리케이션에서 *ServiceDefaults* 프로젝트를 참조합니다. Visual Studio에서 >선택하고 **ServiceDefaults** 프로젝트"를 선택합니다.

3. 애플리케이션 작성기 초기화의 일부로 OpenTelemetry 설치 함수 `ConfigureOpenTelemetry()` 호출합니다.

   C#

   ```
   var builder = WebApplication.CreateBuilder(args)
   builder.ConfigureOpenTelemetry(); // Extension method from ServiceDefaults.
   var app = builder.Build();
   app.MapGet("/", () => "Hello World!");
   app.Run();
   ```

전체 단계별 설명은 예제: OpenTelemetry와 OTLP 사용 및 단독 Aspire 대시보드를 참조하세요.

OpenTelemetry 및 Prometheus를 사용하여 Grafana에서 메트릭 보기

예제 앱을 Prometheus 및 Grafana와 연결하는 방법을 보려면 Prometheus, Grafana 및 JaegerOpenTelemetry 사용 연습에 따릅니다.

다양한 엔드포인트에 병렬 요청을 전송하여 `HttpClient` 강조하려면 다음 엔드포인트를 사용하여 예제 앱을 확장합니다.

C#

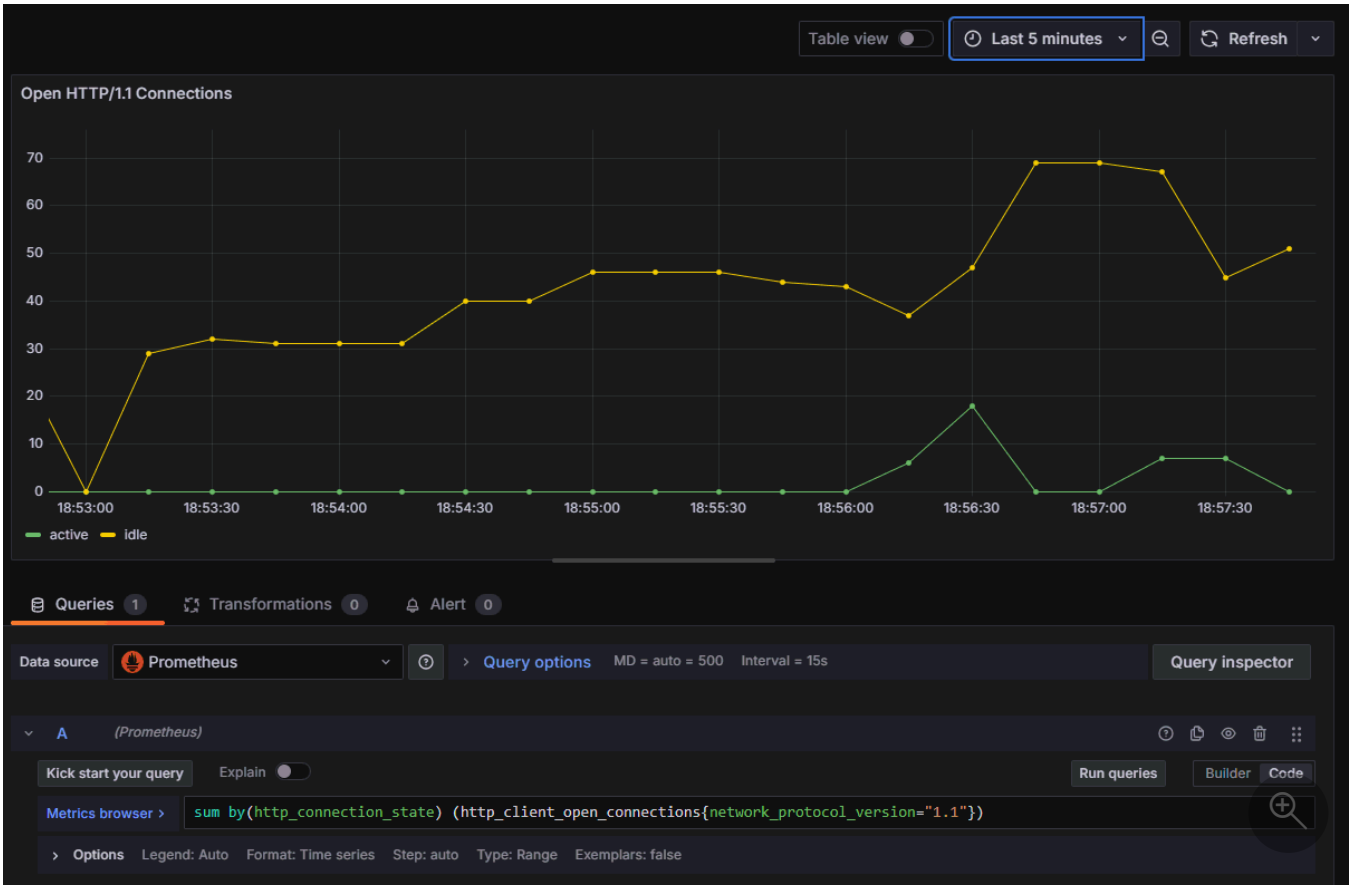
```
app.MapGet("/ClientStress", async Task<string> (ILogger<Program> logger, HttpClient
client) =>
{
    string[] uris = ["http://example.com", "http://httpbin.org/get",
```

```
"https://example.com", "https://httpbin.org/get"];
    await Parallel.ForAsync(0, 50, async (_, ct) =>
    {
        string uri = uris[Random.Shared.Next(uris.Length)];

        try
        {
            await client.GetAsync(uri, ct);
            logger.LogInformation($"{uri} - done.");
        }
        catch { logger.LogInformation($"{uri} - failed."); }
    });
    return "Sent 50 requests to example.com and httpbin.org.";
});
```

위쪽 도구 모음에서 + 아이콘을 선택한 다음 **대시보드**선택하여 Grafana 대시보드를 만듭니다. 표시되는 대시보드 편집기에서 **제목** 상자에 **HTTP/1.1 연결 열기**를 입력하고, PromQL 식 필드에 다음 쿼리를 입력합니다.

```
sum by(http_connection_state)
(http_client_open_connections{network_protocol_version="1.1"})
```

적용 선택하여 새 대시보드를 저장하고 봅니다. 풀에서 활성 및 유휴 HTTP/1.1 연결 수를 표시합니다.



농축

보강 메트릭에 사용자 지정 태그(특성 또는 레이블이라고도 함)를 추가하는 것입니다. 이는 앱이 메트릭을 사용하여 빌드된 대시보드 또는 경고에 사용자 지정 분류를 추가하려는 경우에 유용합니다. http.client.request.duration 계측기는 콜백을 HttpMetricsEnrichmentContext을 통해 등록하여 강화를 지원합니다. 이 API는 하위 수준 API이며 각 `HttpRequestMessage`대해 별도의 콜백 등록이 필요합니다.

단일 위치에서 콜백 등록을 수행하는 간단한 방법은 사용자 지정 DelegatingHandler구현하는 것입니다. 이렇게 하면 내부 처리기로 전달되고 서버로 전송되기 전에 요청을 가로채고 수정할 수 있습니다.

C#

```
using System.Net.Http.Metrics;

using HttpClient client = new(new EnrichmentHandler() { InnerHandler = new
HttpClientHandler() });

await client.GetStringAsync("https://httpbin.org/response-headers?Enrichment-
Value=A");
await client.GetStringAsync("https://httpbin.org/response-headers?Enrichment-
Value=B");

sealed class EnrichmentHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request, CancellationToken cancellationToken)
    {
        HttpMetricsEnrichmentContext.AddCallback(request, static context =>
        {
            if (context.Response is not null) // Response is null when an exception
occurs.
            {
                // Use any information available on the request or the response to
emit custom tags.
                string? value = context.Response.Headers.GetValues("Enrichment-
Value").FirstOrDefault();
                if (value != null)
                {
                    context.AddCustomTag("enrichment_value", value);
                }
            }
        });
        return base.SendAsync(request, cancellationToken);
    }
}
```

IHttpClientFactory으로 작업하는 경우, AddHttpMessageHandler을 사용하여 `EnrichmentHandler`를 등록할 수 있습니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using System.Net.Http.Metrics;

ServiceCollection services = new();
services.AddHttpClient(Options.DefaultName).AddHttpMessageHandler(() => new
EnrichmentHandler());

ServiceProvider serviceProvider = services.BuildServiceProvider();
HttpClient client = serviceProvider.GetRequiredService<HttpClient>();

await client.GetStringAsync("https://httpbin.org/response-headers?Enrichment-
Value=A");
await client.GetStringAsync("https://httpbin.org/response-headers?Enrichment-
Value=B");
```

> ⓘ **참고 항목**
>
> 성능상의 이유로 보강 콜백은 `http.client.request.duration` 계측이 활성화된 경우에만 호출됩니다. 즉, 메트릭을 수집해야 합니다. `dotnet-monitor`, Prometheus 내보내기, MeterListener또는 `MetricCollector<T>`일 수 있습니다.

`IMeterFactory` 및 `IHttpClientFactory` 통합

HTTP 메트릭은 격리 및 테스트 가능성을 염두에 두고 설계되었습니다. 이러한 측면은 IMeterFactory사용하여 지원되며, 이를 통해 미터를 서로 격리하기 위해 사용자 지정 Meter 인스턴스에서 메트릭을 게시할 수 있습니다. 기본적으로 전역 Meter 모든 메트릭을 내보내는 데 사용됩니다. 이 Meter은(는) `System.Net.Http` 라이브러리 내에 있습니다. 사용자 지정 IMeterFactory 인스턴스를 SocketsHttpHandler.MeterFactory 또는 HttpClientHandler.MeterFactory할당하여 이 동작을 재정의할 수 있습니다.

> ⓘ **참고 항목**
>
> Meter.Name는 `System.Net.Http`와 `HttpClientHandler`이 내보내는 모든 메트릭에 대해 `SocketsHttpHandler`입니다.

.NET 8 이상에서 Microsoft.Extensions.Http 및 IHttpClientFactory 사용하는 경우 기본 `IHttpClientFactory` 구현은 `IMeterFactory` 등록된 IServiceCollection 인스턴스를 자동으로 선택하고 내부적으로 만드는 기본 처리기에 할당합니다.

> ⓘ **참고 항목**
>
> .NET 8부터 AddHttpClient 메서드는 자동으로 AddMetrics 호출하여 메트릭 서비스를 초기화하고 기본 IMeterFactory 구현을 IServiceCollection등록합니다. 기본 IMeterFactory은 이름으로 Meter 인스턴스를 캐시합니다. 즉, Meter마다 이름이 `System.Net.Http` 인 IServiceCollection 하나가 있습니다.

테스트 메트릭

다음 예제에서는 `IHttpClientFactory` NuGet 패키지에서 xUnit, `MetricCollector<T>` 및 Microsoft.Extensions.Diagnostics.Testing 사용하여 단위 테스트에서 기본 제공 메트릭의 유효성을 검사하는 방법을 보여 줍니다.

C#

```
[Fact]
public async Task RequestDurationTest()
{
    // Arrange
    ServiceCollection services = new();
    services.AddHttpClient();
    ServiceProvider serviceProvider = services.BuildServiceProvider();
    var meterFactory = serviceProvider.GetService<IMeterFactory>();
    var collector = new MetricCollector<double>(meterFactory,
        "System.Net.Http", "http.client.request.duration");
    var client = serviceProvider.GetRequiredService<HttpClient>();

    // Act
    await client.GetStringAsync("http://example.com");

    // Assert
    await collector.WaitForMeasurementsAsync(minCount:
1).WaitAsync(TimeSpan.FromSeconds(5));
    Assert.Collection(collector.GetMeasurementSnapshot(),
        measurement =>
        {
            Assert.Equal("http", measurement.Tags["url.scheme"]);
            Assert.Equal("GET", measurement.Tags["http.request.method"]);
        });
}
```

메트릭과 EventCounters 비교

메트릭은 EventCounters보다 기능이 풍부한 , 특히 다차원 특성 때문입니다. 이 다차원을 사용하면 Prometheus와 같은 도구에서 정교한 쿼리를 만들고 EventCounters에서 불가능한 수준에서 인사이트를 얻을 수 있습니다.

그럼에도 불구하고 .NET 8부터는 메트릭을 사용하여 `System.Net.Http` 및 `System.Net.NameResolutions` 구성 요소만 계측됩니다. 즉, `System.Net.Sockets` 또는 `System.Net.Security`같은 스택의 하위 수준에서 카운터가 필요한 경우 EventCounters를 사용해야 합니다.

또한 메트릭과 일치하는 EventCounters 간에는 몇 가지 의미 체계 차이가 있습니다. 예를 들어 `HttpCompletionOption.ResponseContentRead`사용하는 경우 current-requests EventCounter 요청 본문의 마지막 바이트를 읽은 순간까지 요청이 활성화된 것으로 간주합니다. http.client.active_requests 해당 메트릭에는 활성 요청을 계산할 때 응답 본문을 읽는 데 소요된 시간이 포함되지 않습니다.

더 많은 메트릭이 필요하세요?

메트릭을 통해 노출될 수 있는 다른 유용한 정보에 대한 제안이 있는 경우 dotnet/runtime 이슈 을 만드세요.

Last updated on 2026. 02. 24.

System.Net 라이브러리의 분산 주적

2025. 10. 11.

분산 추적 엔지니어가 애플리케이션 내의 오류 및 성능 문제, 특히 여러 컴퓨터 또는 프로세스에 분산된 문제를 지역화하는 데 도움이 되는 진단 기술입니다. 이 기술은 서로 다른 구성 요소에서 수행하는 작업을 상호 연결하고 애플리케이션이 동시 요청에 대해 수행할 수 있는 다른 작업과 분리하여 애플리케이션을 통해 요청을 추적합니다. 예를 들어 일반적인 웹 서비스에 대한 요청은 먼저 부하 분산 장치에서 받은 다음 웹 서버 프로세스로 전달된 다음 데이터베이스에 여러 쿼리를 수행합니다. 분산 추적을 사용하면 엔지니어가 이러한 단계 중 실패한 단계와 각 단계의 소요 기간을 구분할 수 있습니다. 실행될 때 각 단계에서 생성된 메시지를 기록할 수도 있습니다.

.NET의 추적 시스템은 OTel(OpenTelemetry)과 작동하도록 설계되었으며 OTel을 사용하여 데이터를 모니터링 시스템으로 내보냅니다. .NET의 추적은 System.Diagnostics API를 사용하여 구현됩니다. 여기서 작업 단위는 OTel System.Diagnostics.Activity해당하는 클래스로 표시됩니다. OpenTelemetry는 의미 체계 규칙이라고 하는 특성(태그)과 함께 범위(활동)에 대한 업계 전반의 표준 명명 체계를 정의합니다. .NET 원격 분석은 가능한 경우 기존 의미 체계 규칙을 사용합니다.

> **참고**
>
> *용어는* 및 *활동* 이 문서의 동의어입니다. .NET 코드의 컨텍스트에서 System.Diagnostics.Activity 인스턴스를 참조합니다. OTel 범위를 System.Span<T>혼동하지 마세요.

> **팁**
>
> 태그/특성과 함께 모든 기본 제공 활동의 포괄적인 목록은 .NET 기본 제공 활동을 참조하세요.

계측

추적을 내보내기 위해 System.Net 라이브러리는 기본 제공 원본을 사용하여 ActivitySource 수행된 작업을 추적하는 Activity 개체를 만듭니다. 활동은 ActivitySource구독하는 수신기가 있는 경우에만 만들어집니다.

기본 제공 계측은 .NET 버전으로 발전했습니다.

- .NET 8 이하에서는 계측이 빈 HTTP 클라이언트 요청 작업만들기로 제한됩니다. 즉, 사용자는 OpenTelemetry.Instrumentation.Http 라이브러리를 사용하여 유용한 추적을 내보내

는 데 필요한 정보(예: 태그)로 활동을 채워야 합니다.

- .NET 9는 HTTP 클라이언트 요청 활동에서 OTel HTTP 클라이언트 의미 체계 규칙에 따라 이름, 상태, 예외 정보, 그리고 가장 중요한 태그를 내보내어 계측을 확장했습니다. 즉, `OpenTelemetry.Instrumentation.Http` 같은 고급 기능이 필요하지 않으면 .NET 9 이상에서 종속성을 생략할 수 있습니다.
- .NET 9는 또한 실험적 연결 추적도입하여 `System.Net` 라이브러리에 새로운 활동을 추가하여 연결 문제 진단을 지원합니다.

System.Net 추적 수집

가장 낮은 수준추적 컬렉션은 사용자 정의 논리를 포함하는 AddActivityListener 개체를 등록하는 ActivityListener 메서드를 통해 지원됩니다.

그러나 애플리케이션 개발자는 OpenTelemetry .NET SDK 제공된 기능을 기반으로 구축된 풍부한 에코시스템을 사용하여 추적을 수집, 내보내기 및 모니터링하는 것이 좋습니다.

- OTel을 사용하여 추적 컬렉션에 대한 기본적인 이해를 얻으려면 OpenTelemetry 사용하여 추적을 수집하는가이드를 참조하세요.
- **프로덕션 시간** 추적 수집 및 모니터링을 위해 OpenTelemetry를 Prometheus, Grafana, Jaeger와 함께 사용하거나, Azure Monitor 및 Application Insights와 함께 사용할 수 있습니다. 그러나 이러한 도구는 매우 복잡하며 개발 시 사용하기가 불편할 수 있습니다.
- **개발 시간** 추적 수집 및 모니터링의 경우 애플리케이션에서 분산 추적을 시작하고 로컬에서 문제를 진단하는 간단하지만 확장 가능한 방법을 제공하는 Aspire를 사용하는 것이 좋습니다.
- 또한 Aspire 오케스트레이션 없이 Aspire Service 기본 설정 프로젝트를 재사용할 수도 있습니다. 이는 ASP.NET 프로젝트에서 OpenTelemetry 추적 및 메트릭을 소개하고 구성하는 편리한 방법입니다.

Aspire를 사용하여 추적 수집

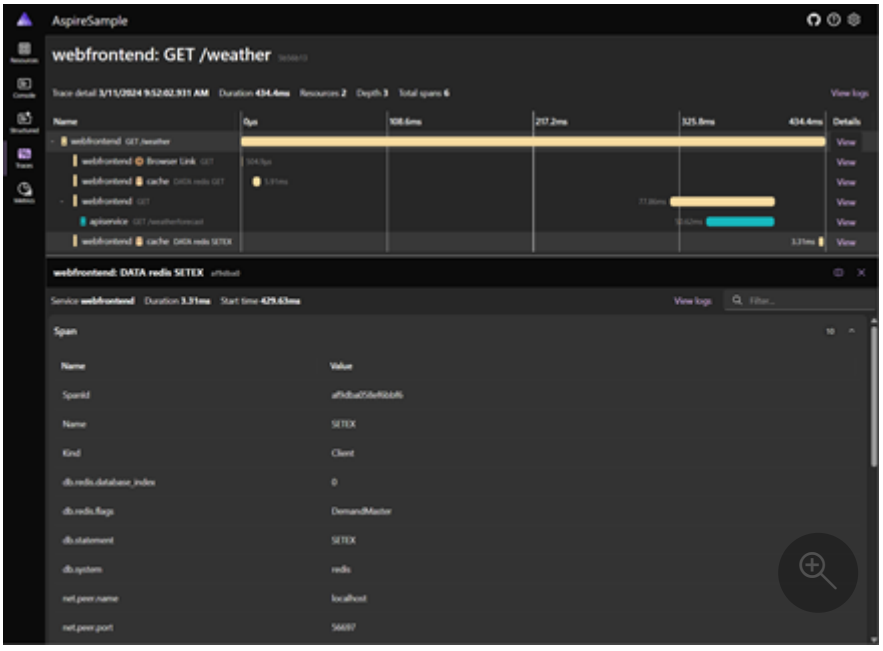
ASP.NET 애플리케이션에서 추적 및 메트릭을 수집하는 간단한 방법은 Aspire를 사용하는 것입니다. Aspire는 분산 애플리케이션을 쉽게 만들고 사용할 수 있도록 .NET에 대한 확장 집합입니다. Aspire를 사용할 때의 이점 중 하나는 .NET용 OpenTelemetry 라이브러리를 사용하여 원격 분석이 기본 제공된다는 것입니다.

Aspire의 기본 프로젝트 템플릿에는 프로젝트가 포함되어 있습니다 `ServiceDefaults` . Aspire 솔루션의 각 서비스에는 서비스 기본값 프로젝트에 대한 참조가 있습니다. 서비스는 OTel을 설정하고 구성하는 데 사용합니다.

서비스 기본값 프로젝트 템플릿에는 OTel SDK, ASP.NET, HttpClient 및 런타임 계측 패키지가 포함됩니다. 이러한 계측 구성 요소는 Extensions.cs 파일에 구성됩니다. Aspire 대시보드에서 원

격 분석 시각화를 지원하기 위해 서비스 기본값 프로젝트에는 기본적으로 OTLP 내보내기도 포함됩니다.

Aspire 대시보드는 로컬 디버그 주기에 원격 분석 관찰을 제공하도록 설계되어 개발자가 애플리케이션에서 원격 분석을 생성하도록 할 수 있습니다. 원격 분석 시각화는 이러한 애플리케이션을 로컬로 진단하는 데도 도움이 됩니다. 서비스 간의 호출을 관찰할 수 있다는 것은 프로덕션에서처럼 디버그 시 유용합니다. Aspire 대시보드는 Visual Studio의 Project 또는 `AppHost` 명령줄에서 `dotnet run` 프로젝트를 `AppHost`로 실행할 때 자동으로 시작됩니다.



Aspire에 대한 자세한 내용은 다음을 참조하세요.

- 포부 개요
- Aspire 원격 분석
- Aspire 대시보드

서비스 기본값 프로젝트를 Aspire 오케스트레이션 없이 다시 사용

Aspire Service Defaults 프로젝트는 오케스트레이션에 AppHost와 같은 *나머지 Aspire를 사용하지 않더라도* ASP.NET 프로젝트에 OTel을 쉽게 구성할 수 있는 방법을 제공합니다. 서비스 기본값 프로젝트는 Visual Studio 또는 `dotnet new`통해 프로젝트 템플릿으로 사용할 수 있습니다. OTel을 구성하고 OTLP 내보내기를 설정합니다. 그런 다음 OTel 환경 변수 사용하여 원격 분석을 보내고 애플리케이션에 대한 리소스 속성을 제공하도록 OTLP 엔드포인트를 구성할 수 있습니다.

Aspire 외부에서 *ServiceDefaults*를 사용하는 단계는 다음과 같습니다.

1. Visual Studio에서 새 프로젝트 추가를 사용하여 *ServiceDefaults* 프로젝트를 솔루션에 추가하거나 `dotnet new`사용합니다.

   .NET CLI

   ```
   dotnet new aspire-servicedefaults --output ServiceDefaults
   ```

2. ASP.NET 애플리케이션에서 *ServiceDefaults* 프로젝트를 참조합니다. Visual Studio에서 >선택하고 **ServiceDefaults** 프로젝트"를 선택합니다.

3. 애플리케이션 작성기 초기화의 일부로 OpenTelemetry 설치 함수 `ConfigureOpenTelemetry()` 호출합니다.

   C#

   ```
   var builder = WebApplication.CreateBuilder(args)
   builder.ConfigureOpenTelemetry(); // Extension method from ServiceDefaults.
   var app = builder.Build();
   app.MapGet("/", () => "Hello World!");
   app.Run();
   ```

전체 설명은 예제: OTLP와 함께 OpenTelemetry 사용 및 독립 실행형 Aspire 대시보드를에서 확인하십시오.

실험적 연결 추적

`HttpClient` 문제 또는 병목 상태를 해결할 때 HTTP 요청을 보낼 때 소요되는 시간을 확인하는 것이 중요할 수 있습니다. 일반적으로 DNS 조회, TCP 연결 및 TLS 핸드셰이크로 분류되는 HTTP 연결 설정 중에 문제가 발생합니다.

.NET 9에서는 연결 설정의 DNS, TCP 및 TLS 단계를 나타내는 3개의 자식 범위가 있는 `HTTP connection setup` 범위를 추가하는 실험적 연결 추적을 도입했습니다. 연결 추적의 HTTP 부분은 SocketsHttpHandler내에서 구현됩니다. 즉, 활동 모델은 기본 연결 풀링 동작을 준수해야 합니다.

> **참고**
>
> SocketsHttpHandler연결 및 요청에는 독립적인 수명 주기가 있습니다. **풀된 연결** 오랜 시간 동안 유지될 수 있으며 많은 요청을 처리할 수 있습니다. 요청을 수행할 때 연결 풀에서 즉시 사용할 수 있는 연결이 없는 경우 요청이 요청 큐에 추가되어 사용 가능한 연결을 기다립니다. 대기 요청과 연결 사이에는 직접적인 관계가 없습니다. 다른 연결을 사용할 수 있게 되었을 때 연결 프로세스가 시작되었을 수 있습니다. 이 경우 해제된 연결이 사용됩니다. 따

라서 `HTTP connection setup` 범위는 `HTTP client request` 범위의 자식으로 모델링되지 않습니다. 대신 범위 링크가 사용됩니다.

.NET 9에서는 자세한 연결 정보를 수집할 수 있도록 다음 범위를 도입했습니다.

테이블 확장

이름	ActivitySource	묘사
HTTP wait_for_connection	`Experimental.System.Net.Http.Connections`	요청 큐에서 사용 가능한 연결을 기다리는 시간 간격을 나타내는 `HTTP client request` 범위의 하위 범위입니다.
HTTP connection_setup	`Experimental.System.Net.Http.Connections`	HTTP 연결의 설치를 나타냅니다. 자체 `TraceId`있는 별도의 추적 루트 범위입니다. `HTTP client request` 범위에는 `HTTP connection_setup`대한 링크가 포함될 수 있습니다.
DNS lookup	`Experimental.System.Net.NameResolution`	Dns 클래스에서 수행하는 DNS 조회입니다.
socket connect	`Experimental.System.Net.Sockets`	Socket 연결 설정
TLS handshake	`Experimental.System.Net.Security`	TLS 클라이언트 또는 서버 핸드셰이크가 SslStream에 의해 수행됩니다.

참고

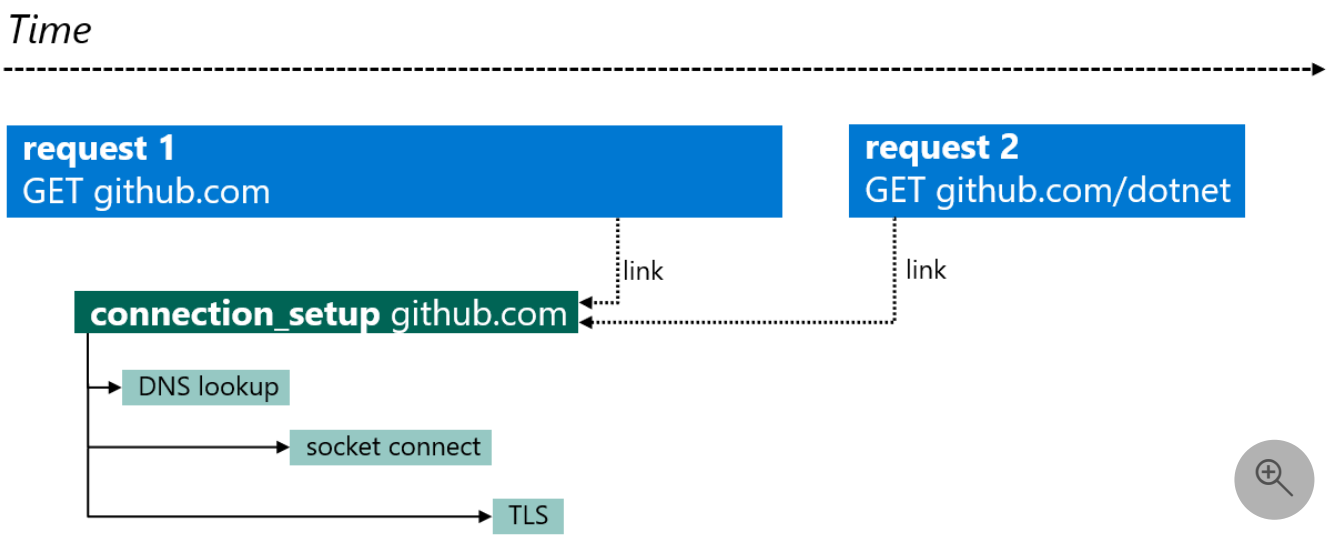
해당 `ActivitySource` 이름은 `Experimental`접두사로 시작합니다. 이러한 범위는 프로덕션 환경에서 얼마나 잘 작동하는지에 대해 자세히 알아보면서 이후 버전에서 변경될 수 있습니다.

이러한 범위는 워크로드가 많은 프로덕션 시나리오에서 24x7을 사용하기에는 너무 자세한 정보입니다. 이는 시끄럽고 일반적으로 이 수준의 계측이 필요하지 않습니다. 그러나 연결 문제를 진단하거나 네트워크 및 연결 대기 시간이 서비스에 미치는 영향을 자세히 이해하려는 경우 다른 수단으로 수집하기 어려운 인사이트를 제공합니다.

`Experimental.System.Net.Http.Connections` ActivitySource를 사용하도록 설정하면 `HTTP client request`제공하는 연결에 해당하는 `HTTP connection_setup` 범위에 대한 링크가 포함됩니다. HTTP 연결은 수명이 길어질 수 있으므로 각 요청 활동에서 연결 범위에 대한 많은 링크가 발생할 수 있습니다. 일부 APM 모니터링 도구는 보기 화면을 구축하기 위해 범위 간 링크를 적극적

으로 탐색하므로, 이 범위를 포함하면 많은 링크를 처리할 수 없도록 설계된 경우 문제가 발생할 수 있습니다.

다음 다이어그램은 범위 및 해당 관계의 동작을 보여 줍니다.



.NET 9에서 실험적 연결 추적 사용 안내

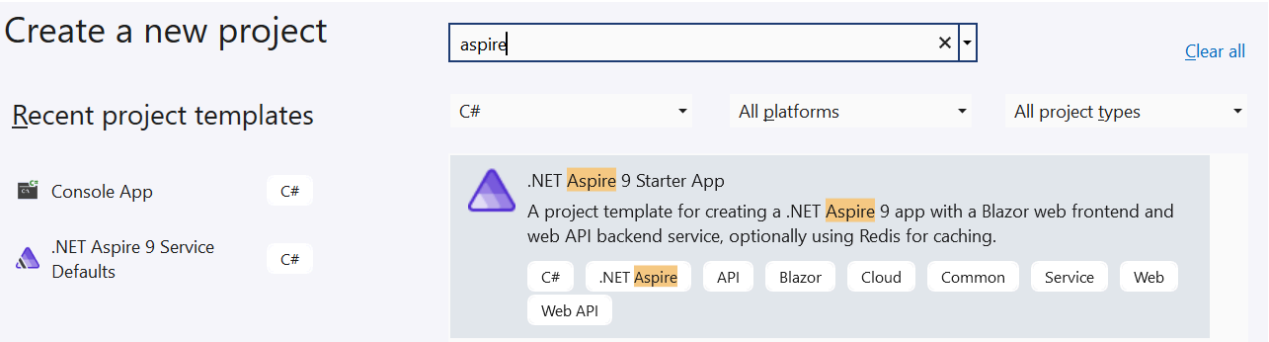
이 연습에서는 .NET 9 Aspire Starter 앱을 사용하여 연결 추적을 시연합니다. 하지만 다른 모니터링 도구 를 사용해도 쉽게 설정할 수 있습니다. 주요 단계는 ActivitySources를 사용하도록 설정하는 것입니다.

1. **Aspire 9 Starter 앱**을 `dotnet new`를 사용하여 만듭니다.

   .NET CLI

   ```
   dotnet new aspire-starter-9 --output ConnectionTracingDemo
   ```

   또는 Visual Studio에서 다음을 수행합니다.

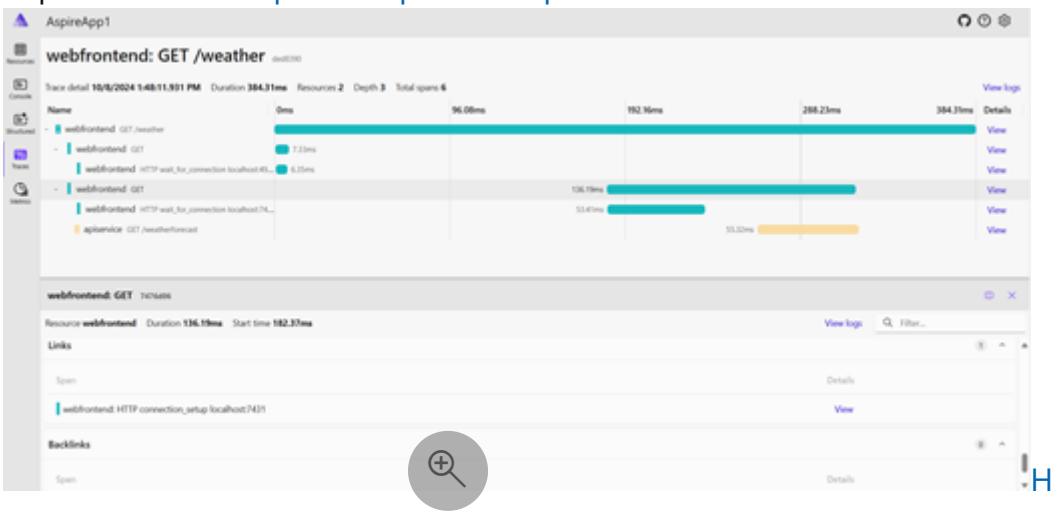
2. `Extensions.cs` 프로젝트에서 `ServiceDefaults`을 열고, 추적 구성 콜백에서 연결에 필요한 ActivitySources를 추가하도록 `ConfigureOpenTelemetry` 메서드를 편집합니다.

   C#

```
.WithTracing(tracing =>
{
    tracing.AddAspNetCoreInstrumentation()
        // Instead of using .AddHttpClientInstrumentation()
        // .NET 9 allows to add the ActivitySources directly.
        .AddSource("System.Net.Http")
        // Add the experimental connection tracking ActivitySources using a
wildcard.
        .AddSource("Experimental.System.Net.*");
});
```

3. 솔루션을 시작합니다. 이는 Aspire 대시보드를 열어야 합니다.
4. `webfrontend` 앱의 날씨 페이지로 이동하여 `HttpClient`쪽으로 `apiservice` 요청을 생성합니다.
5. 대시보드로 돌아가서 **흔적** 페이지로 이동합니다. `webfrontend: GET /weather` 추적을 엽니다.

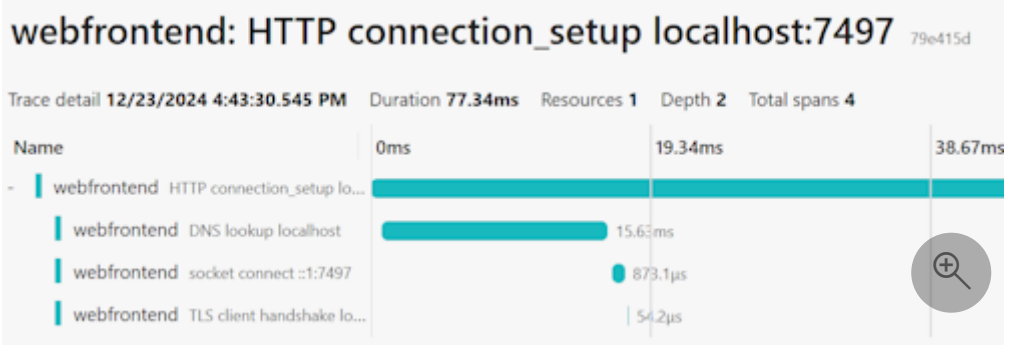
Aspire 대시보드HttpClient Spans in Aspire Dashboard



HttpClient Spans in Aspire DashboardHttpClient 범위

연결 계측을 사용하도록 설정된 상태에서 HTTP 요청을 수행하면 클라이언트 요청 범위에 다음과 같은 변경 내용이 표시됩니다.

- 연결을 설정해야 하거나 앱이 연결 풀에서 연결을 기다리는 경우 추가 HTTP wait_for_connection 범위가 표시되며 이는 연결이 만들어질 때까지 기다리는 지연을 나타냅니다. 이렇게 하면 코드에서 수행되는 `HttpClient` 요청과 요청 처리가 실제로 시작되는 시점 사이의 지연을 이해하는 데 도움이 됩니다. 이전 이미지에서:
  - 선택한 범위는 HttpClient 요청입니다.
  - 아래 범위는 요청이 연결이 설정되기를 기다리는 데 소요되는 시간을 나타냅니다.
  - 노란색의 마지막 범위는 요청을 처리하는 대상에서 가져옵니다.
- HttpClient 범위에는 요청에 사용되는 HTTP 연결을 만드는 활동을 나타내는 HTTP connection_setup 범위에 대한 링크가 있습니다.



에 포함됩니다.

앞서설명한 대로 HTTP connection_setup 범위는 고유한 `TraceId`있는 별도의 범위이며 수명은 각 개별 클라이언트 요청과 독립적입니다. 이 범위에는 일반적으로 자식 범위 DNS lookup, (TCP) socket connect및 TLS client handshake가 있습니다.

농축

경우에 따라 기존 `System.Net` 추적 기능을 보강해야 합니다. 일반적으로 이는 기본 제공 작업에 추가 태그/특성을 삽입하는 것을 의미합니다. 이것을 *보강*이라 부릅니다.

OpenTelemetry 계측 라이브러리의 보강 API

HTTP 클라이언트 요청 활동에 태그/특성을 추가하려면 가장 간단한 방법은 OpenTelemetry HttpClient 및 HttpWebRequest 계측 라이브러리의 `HttpClient` 보강 API를 사용하는 것입니다. 이렇게 하려면 OpenTelemetry.Instrumentation.Http 패키지에 종속되어야 합니다.

수동 보강

`HTTP client request` 활동의 강화를 수동으로 수행할 수 있습니다. 이를 위해 작업이 완료되기 전에 요청 활동의 범위에서 실행되는 코드의 Activity.Current 액세스해야 합니다. 이 작업은 `IObserver<DiagnosticListener>` 구현하고 AllListeners 구독하여 네트워킹 작업이 발생하는 경우에 대한 콜백을 가져와서 수행할 수 있습니다. 실제로 OpenTelemetry HttpClient 및 HttpWebRequest 계측 라이브러리는 이렇게 구현되었습니다. 코드 예제는 DiagnosticSourceSubscriber.cs 구독 코드와 알림이 위임된 HttpHandlerDiagnosticListener.cs 기본 구현을 참조하세요.

더 많은 추적이 필요하세요?

추적을 통해 노출될 수 있는 다른 유용한 정보에 대한 제안이 있는 경우 dotnet/runtime 이슈을 만듭니다.

.NET의 HTTP 클라이언트 로깅

HTTP 요청을 만드는 애플리케이션을 빌드할 때 구조적 로깅을 사용하면 나가는 트래픽을 모니터링, 문제 해결 및 분석할 수 있습니다. AddExtendedHttpClientLogging 확장은 `IHttpClientFactory`으로 생성된 모든 HTTP 클라이언트에 대해 향상된 로깅을 제공하며, 다음과 같은 기본 제공 지원을 포함합니다.

- **구성 가능한 로깅** - 기록되는 항목 제어(헤더, 본문, 쿼리 매개 변수, 경로 매개 변수)
- **데이터 편집** - 개인 정보 및 규정 준수를 위해 데이터 분류 및 수정 정책 적용
- **로그 보강** - HTTP 요청 로그에 사용자 지정 컨텍스트 추가
- **최소 오버헤드** - 성능에 민감한 기본값을 사용하는 옵트인 기능

확장 메서드를 호출하여 향상된 HTTP 클라이언트 로깅을 AddExtendedHttpClientLogging 사용하도록 설정합니다. 이렇게 하면 기본 HTTP 클라이언트 로거가 세분화된 구성 및 데이터 준수 기능을 지원하는 확장 로거로 바뀝니다.

패키지 추가

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Http.Diagnostics
```

또는 .NET 10+ SDK를 사용하는 경우:

.NET CLI

```
dotnet package add Microsoft.Extensions.Http.Diagnostics
```

자세한 내용은 .NET 애플리케이션에서 dotnet 패키지 추가 또는 패키지 종속성 관리를 참조하세요.

HTTP 클라이언트의 로깅 사용 활성화

애플리케이션에 향상된 HTTP 클라이언트 로깅을 추가하려면 서비스를 구성할 때 확장 메서드를 호출 AddExtendedHttpClientLogging 합니다.

C#

```
using Microsoft.Extensions.DependencyInjection;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

// Register redaction services (required)
builder.Services.AddRedaction();

// Add enhanced HTTP client logging
builder.Services.AddExtendedHttpClientLogging();

// Add your HTTP clients
```

```
builder.Services.AddHttpClient<MyApiClient>();

using IHost host = builder.Build();

await host.RunAsync();
```

이 등록은 기본 로거를 고급 구성을 지원하는 로거로 대체하여 만든 IHttpClientFactory모든 HTTP 클라이언트에 향상된 로거를 추가합니다.

> ⓘ **Important**
>
> `AddExtendedHttpClientLogging()`는 `AddDefaultLogger()`로 등록된 기본 로거를 포함한 모든 다른 로거를 제거합니다. 이렇게 하면 모든 HTTP 클라이언트에서 일관된 로깅 동작이 보장됩니다.

> ⓘ **Important**
>
> 패키지 `Microsoft.Extensions.Compliance.Redaction`에서 AddRedaction을(를) 호출하여 편집 서비스를 등록해야 합니다. HTTP 클라이언트 로깅 기능은 기본적으로 중요한 정보(예: 경로 매개변수)를 편집하며 종속성 주입 컨테이너에 편집기 제공자가 필요합니다.

특정 HTTP 클라이언트에 로깅 적용

확장 메서드를 사용하여 특정 명명되거나 형식화된 HTTP 클라이언트에 향상된 로깅을 `IHttpClientBuilder` 적용할 수도 있습니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

// Register redaction services (required)
builder.Services.AddRedaction();

// Apply to a named client
builder.Services.AddHttpClient("MyNamedClient")
    .AddExtendedHttpClientLogging();

// Apply to a typed client with configuration
builder.Services.AddHttpClient<MyApiClient>()
    .AddExtendedHttpClientLogging(options =>
    {
        options.LogBody = true;
        options.ResponseBodyContentTypes.Add("application/json");
    });

using IHost host = builder.Build();

await host.RunAsync();
```

오버로드를 `IHttpClientBuilder` 사용하는 경우 로깅은 전역이 아닌 특정 클라이언트에만 적용됩니다.

보강 로그 데이터 보기

향상된 HTTP 클라이언트 로깅은 *보강*을 사용하여 로그에 정보를 추가합니다. 즉, 데이터가 콘솔 메시지가 아닌 구조적 로그에 태그로 추가됩니다. 보강된 정보를 보려면 구조화된 로그를 지원하는 로깅 공급자를 사용합니다.

빠른 테스트를 위해 구조적 로그를 콘솔에 인쇄하는 데 사용합니다 AddJsonConsole .

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddJsonConsole();

builder.Services.AddRedaction();

builder.Services.AddHttpClient("MyClient")
    .AddExtendedHttpClientLogging(options =>
    {
        options.LogBody = true;
        options.ResponseBodyContentTypes.Add("application/json");
    });
```

프로덕션 시나리오의 경우 보강된 태그 데이터를 쿼리하고 시각화할 수 있는 Application Insights, Seq 또는 Elasticsearch와 같은 구조적 로깅 공급자를 사용하는 것이 좋습니다.

로깅 옵션 구성

LoggingOptions 사용하여 클래스를 통해 HTTP 클라이언트 로깅을 구성합니다. 코드에서 옵션을 구성하거나 구성에서 바인딩할 수 있습니다(예: `appsettings.json`).

대리자를 사용하여 구성

C#

```
builder.Services.AddExtendedHttpClientLogging(options =>
{
    // Log request headers with data classification
    options.RequestHeadersDataClasses.Add("User-Agent", DataClassification.None);
    options.RequestHeadersDataClasses.Add("Authorization", DataClassification.Unknown);

    // Log response headers
    options.ResponseHeadersDataClasses.Add("Content-Type", DataClassification.None);

    // Enable request/response body logging (use carefully in production)
    options.LogBody = true;
    options.BodySizeLimit = 4096; // Limit to 4KB
    options.RequestBodyContentTypes.Add("application/json");
    options.ResponseBodyContentTypes.Add("application/json");
});
```

appsettings.json에서 구성

JSON

```
{
  "HttpClientLogging": {
```

```
    "LogRequestStart": false,
    "LogBody": false,
    "BodySizeLimit": 32768,
    "BodyReadTimeout": "00:00:01",
    "RequestHeadersDataClasses": {
      "User-Agent": "None",
      "Content-Type": "None"
    },
    "ResponseHeadersDataClasses": {
      "Content-Type": "None"
    },
    "RequestPathLoggingMode": "Formatted",
    "RequestPathParameterRedactionMode": "Strict"
  }
}
```

C#

```
builder.Services.AddExtendedHttpClientLogging(
    builder.Configuration.GetSection("HttpClientLogging"));
```

구성 옵션

클래스는 LoggingOptions 다음과 같은 구성 옵션을 제공합니다.

기본 로깅 옵션

테이블 확장

Option	유형	Default	Description
`LogRequestStart`	`bool`	`false`	처리 `true`가 시작되기 전에 요청을 기록합니다. 이 경우 `false`요청 및 응답 데이터를 모두 사용하여 단일 항목을 기록합니다.
`LogBody`	`bool`	`false`	HTTP 요청 및 응답 본문의 로깅을 사용하도록 설정합니다. **경고**: 중요한 정보가 누출될 수 있으므로 프로덕션 환경에서 사용하도록 설정하지 마세요.
`BodySizeLimit`	`int`	32,768(≈32KB)	요청 또는 응답 본문에서 읽을 최대 바이트 수입니다. 큰 개체 힙 할당을 방지하려면 85,000바이트 미만으로 유지합니다.
`BodyReadTimeout`	`TimeSpan`	1초	요청 또는 응답 본문을 읽을 때 대기할 최대 시간입니다. 1밀리초에서 1분 사이여야 합니다.
`RequestBodyContentTypes`	`ISet<string>`	비어 있음	HTTP 요청 콘텐츠 형식은 텍스트로 간주되고 serialization에 적합합니다.
`ResponseBodyContentTypes`	`ISet<string>`	비어 있음	HTTP 응답 콘텐츠 형식은 텍스트로 간주되고 serialization에 적합합니다.

헤더 및 쿼리 매개 변수 로깅

테이블 확장

Option	유형	Default	Description
`RequestHeadersDataClasses`	`IDictionary<string, DataClassification>`	비어 있음	수정할 데이터 분류와 함께 로그하기 위한 HTTP 요청 헤더입니다. 비어 있으면 요청 헤더가 기록되지 않습니다.
`ResponseHeadersDataClasses`	`IDictionary<string, DataClassification>`	비어 있음	데이터 삭제를 위한 데이터 분류와 함께 로그할 HTTP 응답 헤더입니다. 비어 있으면 응답 헤더가 기록되지 않습니다.
`RequestQueryParametersDataClasses`	`IDictionary<string, DataClassification>`	비어 있음	데이터 분류와 함께 기록할 HTTP 요청 쿼리 매개변수입니다. 비어 있으면 쿼리 매개 변수가 기록되지 않습니다. 실험적 기능.
`LogContentHeaders`	`bool`	`false`	HTTP 콘텐츠 헤더(예 `Content-Type`:)의 로깅을 사용하도록 설정합니다. 콘텐츠 헤더가 있는 `RequestHeadersDataClasses` `ResponseHeadersDataClasses` 경우에만 사용하도록 설정합니다. 실험적 기능.

경로 및 경로 매개 변수 로깅

테이블 확장

Option	유형	Default	Description
`RequestPathLoggingMode`	`OutgoingPathLoggingMode`	`Formatted`	나가는 HTTP 요청 경로를 기록하는 방법입니다. 이 아닌 `RequestPathParameterRedactionMode` 경우에만 `None` 적용됩니다.
`RequestPathParameterRedactionMode`	`HttpRouteParameterRedactionMode`	`Strict`	나가는 HTTP 요청에서 경로 매개 변수를 수정하는 방법입니다.
`RouteParameterDataClasses`	`IDictionary<string, DataClassification>`	비어 있음	해당 데이터 분류를 사용하여 수정할 매개 변수를 라우팅합니다.

로그 확장 도구 추가

HTTP 클라이언트 로그 보강자를 사용하면 요청, 응답 및 예외에 따라 HTTP 클라이언트 로그에 사용자 지정 컨텍스트 정보를 추가할 수 있습니다. 범용 로그 보강자와 달리 HTTP 클라이언트 보강자는 특히 나가는 HTTP 요청을 대상으로 하며 HTTP 관련 컨텍스트에 액세스할 수 있습니다.

다음은 빠른 예제입니다.

C#

```
public class CustomHttpLogEnricher : IHttpClientLogEnricher
{
    public void Enrich(IEnrichmentTagCollector collector,
        HttpRequestMessage request, HttpResponseMessage? response, Exception? exception)
    {
        // Add tags based on the exception (if available)
```

```
        if (exception is not null)
        {
            collector.Add("error_type", exception.GetType().Name);
        }
    }
}

// Register the enricher
builder.Services.AddHttpClientLogEnricher<CustomHttpLogEnricher>();
```

모범 사례 및 고급 시나리오를 포함하여 HTTP 클라이언트 로그 보강자를 만들고 사용하는 방법에 대한 자세한 내용은 HTTP 클라이언트 로그 보강자를 참조하세요.

데이터 분류 및 편집

HTTP 클라이언트 로깅은 데이터 편집 및 데이터 분류 기능과 통합되어 중요한 정보를 보호합니다. 헤더, 쿼리 매개 변수 또는 경로 매개 변수에 대해 지정 `DataClassification` 하는 경우 편집 시스템은 구성된 편집기에 따라 적절한 편집을 적용합니다.

예: 중요한 헤더 수정

C#

```
using Microsoft.Extensions.Compliance.Classification;

builder.Services.AddExtendedHttpClientLogging(options =>
{
    // Public headers - no redaction
    options.RequestHeadersDataClasses.Add("User-Agent", DataClassification.None);
    options.RequestHeadersDataClasses.Add("Content-Type", DataClassification.None);

    // Sensitive headers - apply redaction
    options.RequestHeadersDataClasses.Add("Authorization", DataClassification.Unknown);
    options.RequestHeadersDataClasses.Add("X-API-Key", DataClassification.Unknown);
});
```

편집기 구성에 대한 자세한 내용은 .NET의 데이터 편집을 참조하세요.

경로 및 경로 매개 변수 편집

기본적으로 HTTP 클라이언트 로깅은 요청 및 응답 경로를 수정하여 개인 정보를 보호합니다. HTTP 요청 경로를 로깅할 때 다음 옵션을 사용하여 `RequestPathParameterRedactionMode` 경로 매개 변수를 수정하는 방법을 제어할 수 있습니다.

테이블 확장

모드	Description	예시
`None`	수정 안 됨, 전체 경로 기록됨	`/api/users/12345/orders/67890`
`Strict`	모든 경로 매개 변수가 수정됨	`/api/users/{userId}/orders/{orderId}`
`Loose`	매개 `RouteParameterDataClasses` 변수만 수정됩니다.	`/api/users/12345/orders/{orderId}`

이 `RequestPathLoggingMode` 옵션은 기록된 경로의 형식을 제어합니다.

테이블 확장

모드	Description	예시
`Formatted`	매개 변수 이름과 함께 경로 템플릿 형식 사용	`/api/users/{userId}/orders/{orderId}`
`Structured`	경로 및 매개 변수를 별도로 기록합니다.	경로: `/api/users/*/orders/*`, 매개 변수: `{userId, orderId}`

예: 경로 편집 사용 안 함

전체, 수정되지 않은 경로(예: 개발 환경)를 기록하려면 다음으로 `RequestPathParameterRedactionMode`설정합니다`None`.

C#

```
builder.Services.AddExtendedHttpClientLogging(options =>
{
    // Disable redaction of request/response paths
    options.RequestPathParameterRedactionMode = HttpRouteParameterRedactionMode.None;
});
```

> ⊗ **주의**
>
> 경로 편집을 사용하지 않도록 설정하면 로그에 중요한 정보가 노출될 수 있습니다. 개발 시 또는 특정 경로에 중요한 데이터가 포함되지 않은 경우에만 사용합니다 `HttpRouteParameterRedactionMode.None` .

성능 고려 사항

향상된 HTTP 클라이언트 로깅은 성능을 염두에 두고 설계되었지만 고려해야 할 단점이 있습니다.

- **본문 로깅**: 사용하도록 설정하면 `LogBody` 버퍼링 및 읽기 요청/응답 본문에 대한 오버헤드가 추가됩니다. `BodySizeLimit` 및 `BodyReadTimeout`를 사용하여 리소스 사용을 제어합니다.
- **헤더 로깅**: 필요한 로그 헤더만. `RequestHeadersDataClasses` 및 `ResponseHeadersDataClasses`가 비어 있어 헤더 처리 오버헤드를 방지합니다.
- **보강자**: 등록된 각 보강자는 모든 요청에 대해 호출됩니다. 강화 논리를 경량으로 유지하십시오.
- **수정**: 데이터 분류 및 편집은 최소한의 오버헤드를 추가하지만 분류된 항목 수로 크기를 조정합니다.

> 💡 **팁**
>
> 프로덕션 환경에서는 최소 로깅(본문 없음, 제한된 헤더 없음)으로 시작하고 문제 해결에 대해서만 추가 로깅을 사용하도록 설정합니다.

참고하십시오

- .NET의 HTTP 클라이언트 팩터리

- HTTP 클라이언트 로그 강화기
- HTTP 클라이언트 성능 모니터링 및 분석
- .NET의 데이터 편집
- .NET의 데이터 분류
- .NET의 로깅

Last updated on 2026. 02. 13.

.NET의 네트워킹 이벤트

이벤트는 다음 항목에 대한 액세스 권한을 제공합니다.

- 작업의 기간 동안 정확한 타임스탬프를 제공합니다. 예를 들어 서버에 연결하는 데 걸린 시간과 HTTP 요청이 응답 헤더를 받는 데 걸린 시간 등이 있습니다.
- 다른 방법으로 얻을 수 없는 디버그/추적 정보. 예를 들어 연결 풀에서 어떤 종류의 결정을 내렸는지와 그 이유가 있습니다.

계측은 EventSource를 기준으로 하므로 프로세스 내부 및 외부에서 이 정보를 수집할 수 있습니다.

이벤트 공급자

네트워킹 정보는 다음 이벤트 공급자 간에 분할됩니다.

- `System.Net.Http`(`HttpClient` 및 `SocketsHttpHandler`)
- `System.Net.NameResolution` (`Dns`)
- `System.Net.Security` (`SslStream`)
- `System.Net.Sockets`
- `Microsoft.AspNetCore.Hosting`
- `Microsoft-AspNetCore-Server-Kestrel`

원격 분석을 활성화할 때는 일부 성능 부하가 발생하므로 관심 있는 이벤트 공급자만 구독해야 합니다.

인프로세스에서 이벤트 처리

보다 쉬운 이벤트 상관 관계 파악 및 분석을 위해 가능한 경우 In-Process 수집을 선호합니다.

이벤트 리스너

EventListener는 EventSource 이벤트를 생성한 동일한 프로세스 내에서 이 이벤트를 수신 대기할 수 있는 API입니다.

C#

```
using System.Diagnostics.Tracing;

using var listener = new MyListener();

using var client = new HttpClient();
await client.GetStringAsync("https://httpbin.org/get");
```

```
public sealed class MyListener : EventListener
{
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        if (eventSource.Name == "System.Net.Http")
        {
            EnableEvents(eventSource, EventLevel.Informational);
        }
    }

    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        Console.WriteLine($"{DateTime.UtcNow:ss:fff} {eventData.EventName}: " +
            string.Join(' ',
eventData.PayloadNames!.Zip(eventData.Payload!).Select(pair => $"{pair.First}=
{pair.Second}")));
    }
}
```

앞의 코드는 다음과 유사한 출력을 표시합니다.

출력

```
00:598 RequestStart: scheme=https host=httpbin.org port=443 pathAndQuery=/get
versionMajor=1 versionMinor=1 versionPolicy=0
01:470 ConnectionEstablished: versionMajor=1 versionMinor=1
01:474 RequestLeftQueue: timeOnQueueMilliseconds=470,6214 versionMajor=1
versionMinor=1
01:476 RequestHeadersStart:
01:477 RequestHeadersStop:
01:592 ResponseHeadersStart:
01:593 ResponseHeadersStop:
01:633 ResponseContentStart:
01:635 ResponseContentStop:
01:635 RequestStop:
01:637 ConnectionClosed: versionMajor=1 versionMinor=1
```

Yarp.Telemetry.Consumption

위에서 설명한 `EventListener` 접근 방식은 빠른 실험 및 디버깅에 유용하지만 API는 강력한 형식이 지정되지 않았으며 계측된 라이브러리의 구현 세부 정보를 강제로 사용하도록 합니다.

이 문제를 해결하기 위해 .NET은 In Process에서 네트워킹 이벤트를 쉽게 사용할 수 있는 라이브러리 Yarp.Telemetry.Consumption 을 만들었습니다. 패키지는 현재 YARP 프로젝트의 일부로 유지 관리되지만 모든 .NET 애플리케이션에서 사용할 수 있습니다.

이를 사용하려면 관심 있는 인터페이스 및 메서드(이벤트)를 구현합니다.

C#

```
public sealed class MyTelemetryConsumer : IHttpTelemetryConsumer,
INetSecurityTelemetryConsumer
{
    public void OnRequestStart(DateTime timestamp, string scheme, string host, int
port, string pathAndQuery, int versionMajor, int versionMinor, HttpVersionPolicy
versionPolicy)
    {
        Console.WriteLine($"Request to {host} started at {timestamp}");
    }

    public void OnHandshakeStart(DateTime timestamp, bool isServer, string
targetHost)
    {
        Console.WriteLine($"Starting TLS handshake with {targetHost}");
    }
}
```

그런 후 DI 컨테이너에 구현을 등록합니다.

C#

```
services.AddTelemetryConsumer<MyTelemetryConsumer>();
```

라이브러리는 다음과 같은 강력한 형식의 인터페이스를 제공합니다.

- IHttpTelemetryConsumer
- INameResolutionTelemetryConsumer
- INetSecurityTelemetryConsumer
- ISocketsTelemetryConsumer
- IKestrelTelemetryConsumer

이러한 콜백은 계측된 작업의 일부로 호출되므로 일반 로깅 지침이 적용됩니다. 비용이 많이 드는 계산을 콜백의 일부로 차단하거나 수행하지 않아야 합니다. 기본 작업에 대기 시간을 추가하지 않도록 사후 처리 작업을 다른 스레드로 오프로드합니다.

프로세스 외부에서 이벤트 소비

dotnet-trace

dotnet-trace는 네이티브 프로파일러 없이 실행 중인 프로세스의 .NET Core 추적을 수집할 수 있도록 하는 플랫폼 간 도구입니다.

콘솔

```
dotnet tool install --global dotnet-trace
```

콘솔

```
dotnet-trace collect --providers
System.Net.Http,System.Net.Security,System.Threading.Tasks.TplEventSource:0x80:4 --
process-id 1234
```

사용 가능한 모든 명령 및 매개 변수는 dotnet-trace 문서를 참조하세요.

Visual Studio 또는 PerfView에서 캡처된 파일을 분석할 수 있습니다. 자세한 내용은 dotnet-trace 분석 문서를 참조하세요.

PerfView

PerfView는 무료 고급 성능 분석 도구입니다. Windows에서 실행되지만 Linux에서 캡처된 추적도 분석할 수 있습니다.

캡처할 이벤트 목록을 구성하려면 `Advanced Options / Additional Providers` 아래에 지정합니다.

txt

```
*System.Net.Sockets,*System.Net.NameResolution,*System.Net.Http,*System.Net.Securit
y
```

추적 이벤트

TraceEvent는 다양한 프로세스의 이벤트를 실시간으로 사용할 수 있는 라이브러리입니다. `dotnet-trace` 및 `PerfView` 둘 다 이 라이브러리를 사용합니다.

이벤트를 프로그래밍 방식으로 실시간으로 처리하려면 TraceEvent 문서를 참조하세요.

이벤트 시작 및 중지

더 큰 작업은 종종 `Start` 이벤트로 시작하고 `Stop` 이벤트로 끝납니다. 예를 들어 `RequestStart`의 `RequestStop` `System.Net.Http` 이벤트 또는 `ConnectStart`의 `ConnectStop` `System.Net.Sockets` 이벤트를 볼 수 있습니다.

이와 같은 대규모 작업은 종종 `Stop` 이벤트가 항상 존재한다고 보장하지만 항상 존재하는 것은 아닙니다. 예를 들어 `RequestHeadersStart`에서 `System.Net.Http` 이벤트를 본다고 해서 `RequestHeadersStop`가 기록될 것이라고 보장할 수는 없습니다.

이벤트 상관 관계

이제 이러한 이벤트를 관찰하는 방법을 알게 되었으므로 이벤트를 함께, 일반적으로는 원래 HTTP 요청과 상호 연결해야 하는 경우가 많습니다.

보다 쉬운 이벤트 상관 관계 파악 및 분석을 위해 가능한 경우 In-Process 수집을 선호합니다.

공정 내 상관 관계

네트워킹에서 비동기 I/O를 사용하므로 지정된 요청을 완료한 스레드가 해당 요청을 시작한 스레드이기도 하다고 가정할 수 없습니다. 즉, `[ThreadLocal]` 정적 변수를 사용하여 이벤트를 상호 연결할 수는 없지만 AsyncLocal을 사용할 수 있습니다. AsyncLocal 형식은 여러 스레드에서 작업을 상호 연결하는 데 핵심적이므로 숙지하도록 합니다.

`AsyncLocal`을 사용하면 작업의 비동기 흐름에 따라 동일한 상태에 더 깊이 액세스할 수 있습니다. `AsyncLocal` 값은 아래로만 흐르고(비동기 호출 스택까지 깊게) 변경 내용을 호출자에게 전파하지 않습니다.

다음과 같은 예제를 참조하세요.

C#

```
AsyncLocal<int> asyncLocal = new();
asyncLocal.Value = 1;

await WorkAsync();
Console.WriteLine($"Value after WorkAsync: {asyncLocal.Value}");

async Task WorkAsync()
{
    Console.WriteLine($"Value in WorkAsync: {asyncLocal.Value}");
    asyncLocal.Value = 2;
    Console.WriteLine($"Value in WorkAsync: {asyncLocal.Value}");
}
```

이 코드는 다음 출력을 생성합니다.

출력

```
Value in WorkAsync: 1
Value in WorkAsync: 2
Value after WorkAsync: 1
```

`1` 값은 호출자에서 `WorkAsync`로 전달되었지만 `WorkAsync(2)`의 수정 내용은 호출자에게 다시 전달되지 않았습니다.

원격 분석 이벤트(및 해당 콜백)는 기본 작업의 일부로 발생하므로 요청을 시작한 호출자와 동일한 비동기 범위 내에서 발생합니다. 즉, 콜백 내에서 `asyncLocal.Value`를 관찰할 수 있지만 콜백에서 값을 설정하면 스택까지 더 자세히 관찰할 수 없습니다.

다음 단계에서는 일반적인 패턴을 보여 줍니다.

1. 이벤트 콜백 내부에서 업데이트할 수 있는 변경 가능한 클래스를 만듭니다.

```C#
public sealed class RequestInfo
{
    public DateTime StartTime, HeadersSent;
}
```

2. 주 작업 `AsyncLocal.Value` *전에* 상태가 작업으로 흐르도록 설정합니다.

```C#
private static readonly AsyncLocal<RequestInfo> _requestInfo = new();

public async Task SendRequestAsync(string url)
{
    var info = new RequestInfo();
    _requestInfo.Value = info;

    info.StartTime = DateTime.UtcNow;
    await _client.GetStringAsync(url);
```

3. 이벤트 콜백 내에서 공유 상태를 사용할 수 있는지 확인하고 업데이트합니다. 구성 요소가 처음에 `AsyncLocal.Value`를 설정하지 않고 요청을 보낸 경우, `AsyncLocal.Value`는 `null`입니다.

```C#
public void OnRequestHeadersStop(DateTime timestamp)
{
    if (_requestInfo.Value is { } info) info.HeadersSent = timestamp;
}
```

4. 작업을 완료한 후 수집된 정보를 처리합니다.

```C#
await _client.GetStringAsync(url);
```

```
Log($"Time until headers were sent {url} was {info.HeadersSent -
info.StartTime}");
```

이 작업을 수행하는 방법에 대한 자세한 내용은 샘플 섹션을 참조하세요.

프로세스 외부의 상관 관계

모든 이벤트에는 ActivityID라는 데이터가 연결되어 있습니다. 이 ID는 이벤트가 생성될 때 비동기 계층 구조를 인코딩합니다.

PerfView에서 추적 파일을 보면 다음과 같은 이벤트가 표시됩니다.

txt

```
System.Net.Http/Request/Start            ActivityID="/#14396/1/1/"
System.Net.Http/RequestHeaders/Start     ActivityID="/#14396/1/1/2/"
System.Net.Http/RequestHeaders/Stop      ActivityID="/#14396/1/1/2/"
System.Net.Http/ResponseHeaders/Start    ActivityID="/#14396/1/1/3/"
System.Net.Http/ResponseHeaders/Stop     ActivityID="/#14396/1/1/3/"
System.Net.Http/Request/Stop             ActivityID="/#14396/1/1/"
```

이러한 이벤트는 `/#14396/1/1/` 접두사를 공유하기 때문에 동일한 요청에 속한다는 것을 알 수 있습니다.

수동 조사를 수행할 때 유용한 방법은 관심 있는 요청의 `System.Net.Http/Request/Start` 이벤트를 찾은 다음, `ActivityID`를 `Text Filter` 텍스트 상자에 붙여넣는 것입니다. 이제 사용 가능한 모든 공급자를 선택하면 해당 요청의 일부로 생성된 모든 이벤트 목록이 표시됩니다.

PerfView는 자동으로 `ActivityID`를 수집하지만 `dotnet-trace`와 같은 도구를 사용하는 경우 `System.Threading.Tasks.TplEventSource:0x80:4` 공급자를 명시적으로 사용하도록 설정해야 합니다(위의 `dotnet-trace` 예제 참조).

HttpClient 요청 및 연결 수명

.NET 6 이후부터 HTTP 요청은 더 이상 특정 연결에 연결되지 않습니다. 대신, 모든 연결을 사용할 수 있게 되는 즉시, 요청이 처리됩니다.

즉, 다음과 같은 이벤트 순서가 표시될 수 있습니다.

1. 요청 시작
2. Dns 시작
3. 정차 요청
4. Dns 중지

이것은 요청이 DNS 확인을 트리거했지만 DNS 호출이 완료되기 전에 다른 연결에 의해 처리되었음을 나타냅니다. 소켓 연결 또는 TLS 핸드셰이크도 마찬가지입니다. 이러한 작업을 수행하기 전에 원래 요청이 완료될 수 있습니다.

이러한 이벤트는 별도로 고려해야 합니다. 특정 요청의 타임라인에 연결하지 않고 DNS 확인 또는 TLS 핸드셰이크를 자체적으로 모니터링합니다.

내부 진단

.NET의 일부 구성 요소는 내부적으로 발생하는 상황에 대한 자세한 인사이트를 제공하는 추가 디버그 수준 이벤트로 계측됩니다. 이러한 이벤트는 고성능 오버헤드를 유발하며 해당 상황은 지속적으로 변경됩니다. 이름에서 알 수 있듯이 퍼블릭 API의 일부가 아니므로 해당 동작이나 존재에 의존하지 않아야 합니다. 또한 수정되지 않으며 PII를 포함할 수 있습니다.

그럼에도 불구하고 이러한 이벤트는 다른 모든 것이 실패할 때 많은 인사이트를 제공할 수 있습니다. `System.Net` 스택은 `Private.InternalDiagnostics.System.Net.*` 네임스페이스에서 이러한 이벤트를 내보냅니다.

위 `EventListener` 예제의 조건을 `eventSource.Name.Contains("System.Net")`으로 변경하면 스택의 여러 계층에서 100개 이상의 이벤트가 표시됩니다. 자세한 내용은 전체 예제를 참조하세요.

프로세스 외부에서 이들을 사용하려면, 예를 들어, `dotnet-trace`를 사용하십시오.

콘솔

```
dotnet-trace collect --providers Private.InternalDiagnostics.System.Net.Http:0xf --
process-id 1234
```

샘플

- 지정된 엔드포인트에 대한 DNS 확인 측정
- HttpClient를 사용하는 경우 헤더까지의 시간 측정
- Kestrel을 실행하는 ASP.NET Core에서 요청을 처리하는 시간
- .NET 역방향 프록시의 대기 시간 측정

DNS 해상도를 지정된 엔드포인트에 대해 측정하기

C#

```
services.AddTelemetryConsumer(new DnsMonitor("httpbin.org"));
```

```
public sealed class DnsMonitor : INameResolutionTelemetryConsumer
{
    private static readonly AsyncLocal<DateTime?> _startTimestamp = new();
    private readonly string _hostname;

    public DnsMonitor(string hostname) => _hostname = hostname;

    public void OnResolutionStart(DateTime timestamp, string hostNameOrAddress)
    {
        if (hostNameOrAddress.Equals(_hostname,
StringComparison.OrdinalIgnoreCase))
        {
            _startTimestamp.Value = timestamp;
        }
    }

    public void OnResolutionStop(DateTime timestamp)
    {
        if (_startTimestamp.Value is { } start)
        {
            Console.WriteLine($"DNS resolution for {_hostname} took {(timestamp -
start).TotalMilliseconds} ms");
        }
    }
}
```

HttpClient를 사용하는 경우 헤더까지의 시간 측정

C#

```
var info = RequestState.Current; // Initialize the AsyncLocal's value ahead of time

var response = await client.GetStringAsync("http://httpbin.org/get");

var requestTime = (info.RequestStop - info.RequestStart).TotalMilliseconds;
var serverLatency = (info.HeadersReceived - info.HeadersSent).TotalMilliseconds;
Console.WriteLine($"Request took {requestTime:N2} ms, server request latency was
{serverLatency:N2} ms");

public sealed class RequestState
{
    private static readonly AsyncLocal<RequestState> _asyncLocal = new();
    public static RequestState Current => _asyncLocal.Value ??= new();

    public DateTime RequestStart;
    public DateTime HeadersSent;
    public DateTime HeadersReceived;
    public DateTime RequestStop;
}

public sealed class TelemetryConsumer : IHttpTelemetryConsumer
{
```

```
    public void OnRequestStart(DateTime timestamp, string scheme, string host, int
port, string pathAndQuery, int versionMajor, int versionMinor, HttpVersionPolicy
versionPolicy) =>
        RequestState.Current.RequestStart = timestamp;

    public void OnRequestStop(DateTime timestamp) =>
        RequestState.Current.RequestStop = timestamp;

    public void OnRequestHeadersStop(DateTime timestamp) =>
        RequestState.Current.HeadersSent = timestamp;

    public void OnResponseHeadersStop(DateTime timestamp) =>
        RequestState.Current.HeadersReceived = timestamp;
}
```

Kestrel을 실행하는 ASP.NET Core에서 요청을 처리하는 시간

이것은 현재, 지정된 요청의 기간을 측정하는 가장 정확한 방법입니다.

C#

```
public sealed class KestrelTelemetryConsumer : IKestrelTelemetryConsumer
{
    private static readonly AsyncLocal<DateTime?> _startTimestamp = new();
    private readonly ILogger<KestrelTelemetryConsumer> _logger;

    public KestrelTelemetryConsumer(ILogger<KestrelTelemetryConsumer> logger) =>
_logger = logger;

    public void OnRequestStart(DateTime timestamp, string connectionId, string
requestId, string httpVersion, string path, string method)
    {
        _startTimestamp.Value = timestamp;
    }

    public void OnRequestStop(DateTime timestamp, string connectionId, string
requestId, string httpVersion, string path, string method)
    {
        if (_startTimestamp.Value is { } startTime)
        {
            var elapsed = timestamp - startTime;
            _logger.LogInformation("Request {requestId} to {path} took {elapsedMs}
ms", requestId, path, elapsed.TotalMilliseconds);
        }
    }
}
```

.NET 역방향 프록시의 대기 시간 측정

이 샘플은 Kestrel을 통해 인바운드 요청을 수신하고 HttpClient를 통해 아웃바운드 요청을 수행하는 역방향 프록시가 있는 경우에 적용됩니다(예: YARP).

이 샘플은 요청 헤더를 수신할 때부터 요청 헤더가 백 엔드 서버로 전송될 때까지의 시간을 측정합니다.

C#

```
public sealed class InternalLatencyMonitor : IKestrelTelemetryConsumer,
IHttpTelemetryConsumer
{
    private record RequestInfo(DateTime StartTimestamp, string RequestId, string
Path);

    private static readonly AsyncLocal<RequestInfo> _requestInfo = new();
    private readonly ILogger<InternalLatencyMonitor> _logger;

    public InternalLatencyMonitor(ILogger<InternalLatencyMonitor> logger) =>
_logger = logger;

    public void OnRequestStart(DateTime timestamp, string connectionId, string
requestId, string httpVersion, string path, string method)
    {
        _requestInfo.Value = new RequestInfo(timestamp, requestId, path);
    }

    public void OnRequestHeadersStop(DateTime timestamp)
    {
        if (_requestInfo.Value is { } requestInfo)
        {
            var elapsed = (timestamp -
requestInfo.StartTimestamp).TotalMilliseconds;
            _logger.LogInformation("Internal latency for {requestId} to {path} was
{duration} ms", requestInfo.RequestId, requestInfo.Path, elapsed);
        }
    }
}
```

더 많은 원격 분석이 필요하세요?

이벤트 또는 메트릭을 통해 노출될 수 있는 다른 유용한 정보에 대한 제안 사항이 있는 경우 dotnet/runtime 문제 를 만듭니다.

Yarp.Telemetry.Consumption 라이브러리를 사용하고 있으며 제안 사항이 있는 경우 microsoft/reverse-proxy 문제 를 만듭니다.

Last updated on 2026. 02. 24.

.NET의 네트워킹 이벤트 카운터

2025. 06. 17.

EventCounters 는 경량, 플랫폼 간 및 거의 실시간 성능 메트릭 컬렉션에 사용되는 .NET API입니다.

네트워킹 구성 요소는 EventCounters를 사용하여 기본 진단 정보를 게시하도록 계측됩니다. 여기에는 다음과 같은 정보가 포함됩니다.

- `System.Net.Http` > `requests-started`
- `System.Net.Http` > `requests-failed`
- `System.Net.Http` > `http11-connections-current-total`
- `System.Net.Security` > `all-tls-sessions-open`
- `System.Net.Sockets` > `outgoing-connections-established`
- `System.Net.NameResolution` > `dns-lookups-duration`

> **팁**
>
> 전체 목록은 **잘 알려진 카운터를 참조하세요**.

> **팁**
>
> .NET 8 이상을 대상으로 하는 프로젝트에서는 EventCounters 대신 기능이 풍부한 최신 **네트워킹 메트릭을** 사용하는 것이 좋습니다.

공급자

네트워킹 정보는 다음 공급자 간에 분할됩니다.

- `System.Net.Http` (`HttpClient` 및 `SocketsHttpHandler`)
- `System.Net.NameResolution` (`Dns`)
- `System.Net.Security` (`SslStream`)
- `System.Net.Sockets`
- `Microsoft.AspNetCore.Hosting`
- `Microsoft-AspNetCore-Server-Kestrel`

원격 분석에는 사용하도록 설정할 때 성능 오버헤드가 일부 있으므로 실제로 관심이 있는 공급자만 구독해야 합니다.

프로세스 외부에서 이벤트 카운터 모니터링

dotnet-counters (명령줄 도구)

dotnet-counters 는 임시 상태 모니터링 및 첫 번째 수준 성능 조사를 위한 플랫폼 간 성능 모니터링 도구입니다.

.NET CLI

```
dotnet tool install --global dotnet-counters
```

.NET CLI

```
dotnet-counters monitor --counters System.Net.Http,System.Net.Security --process-id 1234
```

이 명령은 최신 숫자로 콘솔을 지속적으로 새로 고칩니다.

txt

```
[System.Net.Http]
    Current Http 1.1 Connections                       3
    Current Http 2.0 Connections                       1
    Current Http 3.0 Connections                       0
    Current Requests                                   4
    HTTP 1.1 Requests Queue Duration (ms)              0
    HTTP 2.0 Requests Queue Duration (ms)              0
    HTTP 3.0 Requests Queue Duration (ms)              0
    Requests Failed                                    0
    Requests Failed Rate (Count / 1 sec)               0
    Requests Started                                 470
    Requests Started Rate (Count / 1 sec)             18
```

사용 가능한 모든 명령 및 매개 변수는 dotnet-counter 문서를 참조하세요.

Application Insights (애플리케이션 인사이트)

Application Insights는 기본적으로 이벤트 카운터를 수집하지 않습니다. 관심 있는 카운터 집합을 사용자 지정하는 방법에 대한 자세한 내용은 AppInsights EventCounters 문서를 참조하세요.

다음은 그 예입니다.

C#

```
services.ConfigureTelemetryModule<EventCounterCollectionModule>((module, options)
=>
{
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Http",
"current-requests"));
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Http",
"requests-failed"));
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Http",
"http11-connections-current-total"));
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Security",
"all-tls-sessions-open"));
});
```

여러 런타임 및 ASP.NET 이벤트 카운터를 구독하는 방법의 예는 RuntimeEventCounters 샘플을 참조하세요. 각 항목에 `EventCounterCollectionRequest`를 추가하기만 하면 됩니다.

C#

```
foreach (var (eventSource, counters) in RuntimeEventCounters.EventCounters)
{
    foreach (string counter in counters)
    {
        module.Counters.Add(new EventCounterCollectionRequest(eventSource,
counter));
    }
}
```

프로세스 내 이벤트 카운터 사용

라이브러리를 Yarp.Telemetry.Consumption 사용하면 프로세스 내에서 이벤트 카운터를 쉽게 사용할 수 있습니다. 패키지는 현재 YARP 프로젝트의 일부로 유지 관리되지만 모든 .NET 애플리케이션에서 사용할 수 있습니다.

이 인터페이스를 사용하려면 다음 인터페이스를 구현합니다 `IMetricsConsumer<TMetrics>` .

C#

```
public sealed class MyMetricsConsumer : IMetricsConsumer<SocketsMetrics>
{
    public void OnMetrics(SocketsMetrics previous, SocketsMetrics current)
    {
        var elapsedTime = (current.Timestamp -
previous.Timestamp).TotalMilliseconds;
        Console.WriteLine($"Received {current.BytesReceived -
previous.BytesReceived} bytes in the last {elapsedTime:N2} ms");
    }
}
```

그런 다음, 구현을 DI 컨테이너에 등록합니다.

C#

```
services.AddSingleton<IMetricsConsumer<SocketsMetrics>, MyMetricsConsumer>();
services.AddTelemetryListeners();
```

라이브러리는 다음과 같은 강력한 형식의 메트릭 형식을 제공합니다.

- HttpMetrics
- NameResolutionMetrics
- NetSecurityMetrics
- SocketsMetrics
- KestrelMetrics

더 많은 원격 분석이 필요하세요?

이벤트 또는 메트릭을 통해 노출될 수 있는 다른 유용한 정보에 대한 제안이 있는 경우 dotnet/런타임 문제를 만듭니다.

Yarp.Telemetry.Consumption 라이브러리를 사용하고 제안 사항이 있는 경우 microsoft/역방향 프록시 이슈를 등록하세요.

.NET에서 파일 와일드카드 사용

2025. 06. 24.

이 문서에서는 Microsoft.Extensions.FileSystemGlobbing NuGet 패키지로 파일 와일드카드를 사용하는 방법을 알아봅니다. *GLOB*은 와일드카드를 기반으로 파일 및 디렉터리 이름의 일치를 확인하기 위한 패턴을 정의하는 데 사용되는 용어입니다. 와일드카드 사용은 하나 이상의 GLOB 패턴을 정의하고 포함 또는 제외 일치에서 파일을 생성하는 동작입니다.

패턴

사용자 정의 패턴을 기반으로 파일 시스템에서 파일의 일치를 확인하려면 Matcher 개체를 인스턴스화하는 것으로 시작합니다. `Matcher`를 매개 변수 없이 인스턴스화할 수도 있고, 패턴을 파일 이름과 비교하기 위해 내부적으로 사용되는 System.StringComparison 매개 변수를 사용하여 인스턴스화할 수도 있습니다. `Matcher`는 다음과 같은 추가 메서드를 노출합니다.

- Matcher.AddExclude
- Matcher.AddInclude

결과에서 제외하거나 포함할 다양한 파일 이름 패턴을 추가하기 위해 `AddExclude` 및 `AddInclude` 메서드를 여러 번 호출할 수 있습니다. `Matcher`를 인스턴스화하고 패턴을 추가하면 Matcher.Execute 메서드를 사용하여 시작 디렉터리에서 일치 항목을 평가하는 데 사용됩니다.

확장 메서드

`Matcher` 개체에는 여러 확장 메서드가 있습니다.

다중 제외

다중 제외 패턴을 추가하려면 다음을 사용할 수 있습니다.

C#

```
Matcher matcher = new();
matcher.AddExclude("*.txt");
matcher.AddExclude("*.asciidoc");
matcher.AddExclude("*.md");
```

또는 MatcherExtensions.AddExcludePatterns(Matcher, IEnumerable<String>[])을 사용하여 단일 호출에서 다중 제외 패턴을 추가할 수 있습니다.

C#

```
Matcher matcher = new();
matcher.AddExcludePatterns(new [] { "*.txt", "*.asciidoc", "*.md" });
```

이 확장 메서드는 사용자를 대신하여 AddExclude를 호출하는 제공된 모든 패턴을 반복합니다.

다중 포함

다중 포함 패턴을 추가하려면 다음을 사용할 수 있습니다.

C#

```
Matcher matcher = new();
matcher.AddInclude("*.txt");
matcher.AddInclude("*.asciidoc");
matcher.AddInclude("*.md");
```

또는 MatcherExtensions.AddIncludePatterns(Matcher, IEnumerable<String>[])을 사용하여 단일 호출에 다중 포함 패턴을 추가할 수 있습니다.

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });
```

이 확장 메서드는 사용자를 대신하여 AddInclude를 호출하는 제공된 모든 패턴을 반복합니다.

일치하는 모든 파일 가져오기

일치하는 모든 파일을 가져오려면 직접 또는 간접적으로 Matcher.Execute(DirectoryInfoBase)를 호출해야 합니다. 직접 호출하려면 검색 디렉터리를 사용해야 합니다.

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });

string searchDirectory = "../starting-folder/";

PatternMatchingResult result = matcher.Execute(
    new DirectoryInfoWrapper(
        new DirectoryInfo(searchDirectory)));

// Use result.HasMatches and results.Files.
// The files in the results object are file paths relative to the search
directory.
```

위의 C# 코드에서:

- Matcher 개체를 인스턴스화합니다.
- AddIncludePatterns(Matcher, IEnumerable<String>[])을 호출하여 포함할 여러 파일 이름 패턴을 추가합니다.
- 검색 디렉터리 값을 선언하고 할당합니다.
- 지정된 DirectoryInfo에서 `searchDirectory`를 인스턴스화합니다.
- 래핑하는 DirectoryInfoWrapper에서 `DirectoryInfo`를 인스턴스화합니다.
- `Execute` 인스턴스가 제공되면 `DirectoryInfoWrapper`를 호출하여 PatternMatchingResult 개체를 생성합니다.

> **참고**
>
> `DirectoryInfoWrapper` 형식은 `Microsoft.Extensions.FileSystemGlobbing.Abstractions` 네임스페이스에 정의되고 `DirectoryInfo` 형식은 `System.IO` 네임스페이스에 정의됩니다. 불필요한 `using` 지시문을 방지하려면 제공된 확장 메서드를 사용할 수 있습니다.

일치하는 파일을 나타내는 `IEnumerable<string>`을 생성하는 또 다른 확장 메서드가 있습니다.

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });

string searchDirectory = "../starting-folder/";

IEnumerable<string> matchingFiles = matcher.GetResultsInFullPath(searchDirectory);

// Use matchingFiles if there are any found.
// The files in this collection are fully qualified file system paths.
```

위의 C# 코드에서:

- Matcher 개체를 인스턴스화합니다.
- AddIncludePatterns(Matcher, IEnumerable<String>[])을 호출하여 포함할 여러 파일 이름 패턴을 추가합니다.
- 검색 디렉터리 값을 선언하고 할당합니다.
- `GetResultsInFullPath` 값이 제공되면 `searchDirectory`를 호출하여 일치하는 모든 파일을 `IEnumerable<string>`으로 생성합니다.

오버로드 일치

PatternMatchingResult 개체는 FilePatternMatch 인스턴스의 컬렉션을 나타내며 결과에 일치 항목, `boolean`가 있는지를 나타내는 PatternMatchingResult.HasMatches 값을 노출합니다.

`Matcher` 인스턴스를 사용하면 다양한 `Match` 오버로드를 호출하여 패턴 일치 결과를 얻을 수 있습니다. `Match` 메서드는 일치를 평가할 파일 또는 파일 컬렉션을 제공하는 호출자의 책임을 반전합니다. 즉, 호출자는 일치를 확인할 파일을 전달해야 합니다.

> ⓘ **중요**
>
> `Match` 오버로드를 사용하는 경우 파일 시스템 I/O가 관련되지 않습니다. 모든 파일 와일드카드 사용은 `matcher` 인스턴스의 포함 및 제외 패턴을 사용하여 메모리에서 수행됩니다. `Match` 오버로드의 매개 변수는 정규화된 경로일 필요가 없습니다. 지정되지 않은 경우 현재 디렉터리(Directory.GetCurrentDirectory())가 사용됩니다.

단일 파일의 일치를 확인하려면:

```
C#

Matcher matcher = new();
matcher.AddInclude("**/*.md");

PatternMatchingResult result = matcher.Match("file.md");
```

위의 C# 코드에서:

- 임의의 디렉터리 깊이에서 파일 확장명이 *.md*인 모든 파일의 일치를 확인합니다.
- *file.md*라는 이름의 파일이 현재 디렉터리의 하위 디렉터리에 있는 경우:
  - `result.HasMatches`는 `true`가 됩니다.
  - `result.Files`의 일치 항목은 하나가 됩니다.

추가 `Match` 오버로드는 비슷한 방식으로 작동합니다.

include/exclude의 순차적인 평가

기본적으로 선택기는 **모든** 포함 패턴을 먼저 평가한 다음 추가한 순서에 관계없이 **모든** 제외 패턴을 적용합니다. 즉, 이전에 제외된 파일을 다시 포함할 수 없습니다.

Microsoft.Extensions.FileSystemGlobbing 패키지 버전 10부터 *순서가 지정된* 평가를 옵트인할 수 있습니다. 여기서 포함 및 제외는 추가된 순서대로 정확하게 처리됩니다.

```
C#
```

```
using Microsoft.Extensions.FileSystemGlobbing;

// Preserve the order of patterns when matching.
Matcher matcher = new(preserveFilterOrder: true);

matcher.AddInclude("**/*");                // include everything
matcher.AddExclude("logs/**/*");           // exclude logs
matcher.AddInclude("logs/important/**/*"); // re-include important logs

var result = matcher.Execute(new DirectoryInfoWrapper(new DirectoryInfo(root)));
foreach (var file in result.Files)
{
    Console.WriteLine(file.Path);
}
```

이 모드에서는 패턴이 하나씩 적용됩니다.

- `**/*` 는 모든 파일을 추가합니다.
- `logs/**/*`에서 모든 것을 제거합니다`logs/`.
- `logs/important/**/*`에서 `logs/important/`의 하위 파일만 다시 추가합니다.

기본 생성자를 사용하는 기존 코드는 원래 "모든 포함, 모든 제외" 동작으로 계속 실행됩니다.

패턴 형식

`AddExclude` 및 `AddInclude` 메서드에 지정된 패턴은 다음 형식을 사용하여 여러 파일 또는 디렉터리의 일치를 확인할 수 있습니다.

- 정확한 디렉터리 또는 파일 이름
  - `some-file.txt`
  - `path/to/file.txt`
- 구분 문자를 포함하지 않는 0부터 여러 개의 문자를 나타내는 파일 및 디렉터리 이름의 와일드카드 `*`.

테이블 확장

값	설명
`*.txt`	파일 확장명이 *.txt*인 모든 파일.
`*.*`	확장이 있는 모든 파일.
`*`	최상위 디렉터리에 있는 모든 파일.
`.*`	'.'으로 시작하는 파일 이름.

값	설명
`*word*`	파일 이름에 'word'가 있는 모든 파일.
`readme.*`	임의의 파일 확장자명과 함께 'readme'라는 이름의 모든 파일.
`styles/*.css`	'styles/' 디렉터리에서 확장명이 '.css'인 모든 파일.
`scripts/*/*`	'scripts/'의 모든 파일 또는 'scripts/' 아래 한 수준 하위 디렉터리의 모든 파일.
`images*/*`	이름이 'images'이거나 'images'로 시작하는 폴더의 모든 파일.

- 임의의 디렉터리 깊이(`/**/`).

테이블 확장

값	설명
`**/*`	임의의 하위 디렉터리에 있는 모든 파일.
`dir/`	'dir/' 아래 임의의 하위 디렉터리에 있는 모든 파일.
`dir/**/*`	'dir/' 아래 임의의 하위 디렉터리에 있는 모든 파일.

- 상대 경로.

  형제 수준에서 "shared"라는 디렉터리의 모든 파일에 대해 Matcher.Execute(DirectoryInfoBase)에 지정된 기본 디렉터리와 일치를 확인하려면 `../shared/*`를 사용합니다.

예제

다음 예제 디렉터리와 해당 폴더 내의 각 파일을 고려해 보겠습니다.

Directory

```
📂 parent
│    file.md
│    README.md
│
└──📂 child
    │    file.MD
    │    index.js
    │    more.md
    │    sample.mtext
    │
    ├──📂 assets
    │        image.png
```

```
    │       image.svg
    │
    └──📁 grandchild
            file.md
            style.css
            sub.text
```

> **팁**
>
> 일부 파일 확장명은 대문자이고 나머지 파일 확장명은 소문자입니다. 기본적으로 StringComparer.OrdinalIgnoreCase가 사용됩니다. 서로 다른 문자열 비교 동작을 지정하려면 Matcher.Matcher(StringComparison) 생성자를 사용합니다.

대/소문자에 관계없이 파일 확장명이 *.md* 또는 *.mtext*인 모든 markdown 파일을 가져오려면:

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "**/*.md", "**/*.mtext" });

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

애플리케이션을 실행하면 다음과 유사한 결과가 출력됩니다.

Console

```
C:\app\parent\file.md
C:\app\parent\README.md
C:\app\parent\child\file.MD
C:\app\parent\child\more.md
C:\app\parent\child\sample.mtext
C:\app\parent\child\grandchild\file.md
```

임의의 깊이에서 *assets* 디렉터리에 있는 파일을 가져오려면:

C#

```
Matcher matcher = new();
matcher.AddInclude("**/assets/**/*");

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
```

```
    Console.WriteLine(file);
}
```

애플리케이션을 실행하면 다음과 유사한 결과가 출력됩니다.

Console

```
C:\app\parent\child\assets\image.png
C:\app\parent\child\assets\image.svg
```

임의의 깊이에서 디렉터리 이름에 *child*라는 단어가 포함되어 있고 파일 확장명이 *.md*, *.text* 또는 *.mtext*가 아닌 파일을 가져오려면:

C#

```
Matcher matcher = new();
matcher.AddInclude("**/*child/**/*");
matcher.AddExcludePatterns(
    new[]
    {
        "**/*.md", "**/*.text", "**/*.mtext"
    });

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

애플리케이션을 실행하면 다음과 유사한 결과가 출력됩니다.

Console

```
C:\app\parent\child\index.js
C:\app\parent\child\assets\image.png
C:\app\parent\child\assets\image.svg
C:\app\parent\child\grandchild\style.css
```

참고 항목

- 런타임 라이브러리 개요
- 파일 및 스트림 I/O

기본 형식: .NET용 확장 라이브러리

아티클 • 2023. 03. 18.

이 문서에서는 Microsoft.Extensions.Primitives 라이브러리에 관해 알아봅니다. 이 문서의 기본 형식은 BCL의 .NET 기본 형식 또는 C# 언어의 기본 형식과 혼동되지 '않습니다'. 대신 기본 라이브러리 내의 형식은 다음과 같은 일부 주변 .NET NuGet 패키지의 구성 요소 역할을 합니다.

- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.FileExtensions
- Microsoft.Extensions.FileProviders.Composite
- Microsoft.Extensions.FileProviders.Physical
- Microsoft.Extensions.Logging.EventSource
- Microsoft.Extensions.Options
- System.Text.Json

변경 알림

변경이 발생하는 경우 알림을 전파하는 것은 프로그래밍의 기본 개념입니다. 개체의 관찰된 상태는 자주 변경될 수 있습니다. 변경이 발생하는 경우 Microsoft.Extensions.Primitives.IChangeToken 인터페이스의 구현은 사용하여 관련 대상에 해당 변경을 알리는 데 사용됩니다. 사용 가능한 구현은 다음과 같습니다.

- CancellationChangeToken
- CompositeChangeToken

개발자는 고유한 형식을 자유롭게 구현할 수도 있습니다. IChangeToken 인터페이스는 다음과 같은 몇 가지 속성을 정의합니다.

- IChangeToken.HasChanged: 변경이 발생했는지 나타내는 값을 가져옵니다.
- IChangeToken.ActiveChangeCallbacks: 토큰이 사전 예방적으로 콜백을 발생시키는지 나타냅니다. `false`인 경우 토큰 소비자는 `HasChanged`를 폴링하여 변경 내용을 감지해야 합니다.

인스턴스 기반 기능

`CancellationChangeToken`의 다음 예제 사용을 참조하세요.

```
C#
```

```
CancellationTokenSource cancellationTokenSource = new();
CancellationChangeToken cancellationChangeToken =
new(cancellationTokenSource.Token);

Console.WriteLine($"HasChanged: {cancellationChangeToken.HasChanged}");

static void callback(object? _) =>
    Console.WriteLine("The callback was invoked.");

using (IDisposable subscription =
    cancellationChangeToken.RegisterChangeCallback(callback, null))
{
    cancellationTokenSource.Cancel();
}

Console.WriteLine($"HasChanged: {cancellationChangeToken.HasChanged}\n");

// Outputs:
//     HasChanged: False
//     The callback was invoked.
//     HasChanged: True
```

앞의 예제에서 CancellationTokenSource가 인스턴스화되고 Token이 CancellationChangeToken 생성자에 전달됩니다. `HasChanged`의 초기 상태는 콘솔에 기록됩니다. 콜백이 콘솔에 호출될 때 기록되는 `Action<object?> callback`이 생성됩니다. `callback`이 제공되면 토큰의 RegisterChangeCallback(Action<Object>, Object) 메서드가 호출됩니다. `using` 문 내에서 `cancellationTokenSource`가 취소됩니다. 그러면 콜백이 트리거되고 `HasChanged`의 상태가 다시 콘솔에 기록됩니다.

여러 변경 내용 소스에서 조치를 취해야 하는 경우 CompositeChangeToken을 사용합니다. 이 구현은 하나 이상의 변경 토큰을 집계하고, 변경이 트리거되는 횟수와 관계없이 등록된 각 콜백을 정확히 한 번 발생시킵니다. 다음과 같은 예제를 참조하세요.

C#

```
CancellationTokenSource firstCancellationTokenSource = new();
CancellationChangeToken firstCancellationChangeToken =
new(firstCancellationTokenSource.Token);

CancellationTokenSource secondCancellationTokenSource = new();
CancellationChangeToken secondCancellationChangeToken =
new(secondCancellationTokenSource.Token);

CancellationTokenSource thirdCancellationTokenSource = new();
CancellationChangeToken thirdCancellationChangeToken =
new(thirdCancellationTokenSource.Token);

var compositeChangeToken =
    new CompositeChangeToken(
```

```
        new IChangeToken[]
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken,
            thirdCancellationChangeToken
        });

static void callback(object? state) =>
    Console.WriteLine($"The {state} callback was invoked.");

// 1st, 2nd, 3rd, and 4th.
compositeChangeToken.RegisterChangeCallback(callback, "1st");
compositeChangeToken.RegisterChangeCallback(callback, "2nd");
compositeChangeToken.RegisterChangeCallback(callback, "3rd");
compositeChangeToken.RegisterChangeCallback(callback, "4th");

// It doesn't matter which cancellation source triggers the change.
// If more than one trigger the change, each callback is only fired once.
Random random = new();
int index = random.Next(3);
CancellationTokenSource[] sources = new[]
{
    firstCancellationTokenSource,
    secondCancellationTokenSource,
    thirdCancellationTokenSource
};
sources[index].Cancel();

Console.WriteLine();

// Outputs:
//     The 4th callback was invoked.
//     The 3rd callback was invoked.
//     The 2nd callback was invoked.
//     The 1st callback was invoked.
```

이전 C# 코드에서 세 개의 CancellationTokenSource 개체 인스턴스가 만들어지고 해당 CancellationChangeToken 인스턴스와 쌍을 이룹니다. 복합 토큰은 토큰의 배열을 CompositeChangeToken 생성자에 전달하여 인스턴스화됩니다. `Action<object?>` `callback`이 만들어지지만 이번에는 `state` 개체가 사용되고 형식이 지정된 메시지로 콘솔에 기록됩니다. 콜백은 각각 약간 다른 상태 개체 인수를 사용하여 네 번 등록됩니다. 코드는 의사 난수 생성기를 사용하여 변경 토큰 소스 중 하나를 선택하고(어느 것이든 관계없음) 해당 Cancel() 메서드를 호출합니다. 이렇게 하면 변경이 트리거되어 등록된 각 콜백이 정확히 한 번 호출됩니다.

대체 `static` 접근 방식

`RegisterChangeCallback`을 호출하는 대신 Microsoft.Extensions.Primitives.ChangeToken 정적 클래스를 사용할 수 있습니다. 다음 소비 패턴을 고려합니다.

C#

```
CancellationTokenSource cancellationTokenSource = new();
CancellationChangeToken cancellationChangeToken =
new(cancellationTokenSource.Token);

IChangeToken producer()
{
    // The producer factory should always return a new change token.
    // If the token's already fired, get a new token.
    if (cancellationTokenSource.IsCancellationRequested)
    {
        cancellationTokenSource = new();
        cancellationChangeToken = new(cancellationTokenSource.Token);
    }

    return cancellationChangeToken;
}

void consumer() => Console.WriteLine("The callback was invoked.");

using (ChangeToken.OnChange(producer, consumer))
{
    cancellationTokenSource.Cancel();
}

// Outputs:
//     The callback was invoked.
```

이전 예제와 마찬가지로 `changeTokenProducer`에서 생성되는 `IChangeToken`의 구현이 필요합니다. 생산자는 `Func<IChangeToken>`로 정의되며 호출할 때마다 새 토큰을 반환해야 합니다. `consumer`는 `state`를 사용하지 않는 경우 `Action`이고, 제네릭 `TState` 형식이 변경 알림을 통해 이동하는 경우 `Action<TState>`입니다.

문자열 토크나이저, 세그먼트 및 값

문자열과 상호 작용하는 것은 애플리케이션 개발에서 일반적입니다. 문자열의 다양한 표현은 구문 분석, 분할 또는 반복됩니다. 기본 형식 라이브러리는 문자열과의 상호 작용을 보다 최적화하고 효율적으로 만드는 데 도움이 되는 몇 가지 선택 형식을 제공합니다. 다음 형식을 고려합니다.

- StringSegment: 부분 문자열의 최적화된 표현입니다.
- StringTokenizer: `string`을 `StringSegment` 인스턴스로 토큰화합니다.
- StringValues: `null`, 0, 하나 또는 여러 개의 문자열을 효율적인 방식으로 나타냅니다.

`StringSegment` 형식

이 섹션에서는 형식이라고 하는 부분 문자열의 최적화된 표현에 대해 알아봅니다 StringSegment`struct` . 일부 `StringSegment` 속성 및 `AsSpan` 메서드를 보여 주는 다음 C# 코드 예제를 살펴봅니다.

C#

```
var segment =
    new StringSegment(
        "This a string, within a single segment representation.",
        14, 25);

Console.WriteLine($"Buffer: \"{segment.Buffer}\"");
Console.WriteLine($"Offset: {segment.Offset}");
Console.WriteLine($"Length: {segment.Length}");
Console.WriteLine($"Value: \"{segment.Value}\"");

Console.Write("Span: \"");
foreach (char @char in segment.AsSpan())
{
    Console.Write(@char);
}
Console.Write("\"\n");

// Outputs:
//     Buffer: "This a string, within a single segment representation."
//     Offset: 14
//     Length: 25
//     Value: " within a single segment "
//     " within a single segment "
```

앞의 코드는 `string` 값, `offset`, `length`가 제공된 경우 `StringSegment`를 인스턴스화합니다. StringSegment.Buffer는 원래 문자열 인수이고 StringSegment.Value는 StringSegment.Offset 및 StringSegment.Length 값을 기반으로 하는 부분 문자열입니다.

`StringSegment` 구조체는 세그먼트와 상호 작용하기 위한 많은 메서드를 제공합니다.

`StringTokenizer` 형식

StringTokenizer개체는 `string`를 `StringSegment` 인스턴스로 토큰화하는 구조체 형식입니다. 일반적으로 많은 문자열을 토큰화하려면 문자열을 분할하고 반복해야 합니다. 따라서 String.Split을 고려할 수 있습니다. 이러한 API는 비슷하지만 일반적으로 StringTokenizer가 더 나은 성능을 제공합니다. 먼저 다음 예제를 살펴봅니다.

C#

```
var tokenizer =
    new StringTokenizer(
        s_nineHundredAutoGeneratedParagraphsOfLoremIpsum,
        new[] { ' ' });

foreach (StringSegment segment in tokenizer)
{
    // Interact with segment
}
```

이전 코드에서 Lorem Ipsum 텍스트의 자동 생성된 단락 900개와 공백 문자 `' '`의 단일 값을 제공하면 `StringTokenizer` 형식의 인스턴스가 생성됩니다. 토크나이저 내의 각 값은 `StringSegment`로 표시됩니다. 이 코드는 세그먼트를 반복하여 소비자가 각 `segment`와 상호 작용할 수 있도록 합니다.

`StringTokenizer`를 `string.Split`에 비교하는 벤치마크

문자열을 조각화 및 분할하는 다양한 방법을 사용하여 두 가지 방법을 벤치마크와 비교하는 것이 적절합니다. BenchmarkDotNet NuGet 패키지를 사용하여 다음 두 개의 벤치마크 방법을 살펴봅니다.

1. StringTokenizer **사용**:

```C#
StringBuilder buffer = new();

var tokenizer =
    new StringTokenizer(
        s_nineHundredAutoGeneratedParagraphsOfLoremIpsum,
        new[] { ' ', '.' });

foreach (StringSegment segment in tokenizer)
{
    buffer.Append(segment.Value);
}
```

2. String.Split **사용**:

```C#
StringBuilder buffer = new();

string[] tokenizer =
    s_nineHundredAutoGeneratedParagraphsOfLoremIpsum.Split(
        new[] { ' ', '.' });
```

```
foreach (string segment in tokenizer)
{
    buffer.Append(segment);
}
```

두 방법은 모두 API 노출 영역에서 비슷하며 큰 문자열을 청크로 분할할 수 있습니다. 아래 벤치마크 결과는 `StringTokenizer` 접근 방식이 거의 3배 더 빠르지만 '결과가 다를 수 있음'을 보여 줍니다. 모든 성능 고려 사항과 마찬가지로 특정 사용 사례를 평가해야 합니다.

메서드	평균	오류	StdDev	비율
토크나이저	3.315ms	0.0659ms	0.0705ms	0.32
분할	10.257ms	0.2018ms	0.2552ms	1.00

범례

- 평균: 모든 측정의 산술 평균
- 오차: 99.9% 신뢰 간격의 절반
- 표준 편차: 모든 측정의 표준 편차
- 중앙값: 모든 측정의 상위 절반을 구분하는 값(50번째 백분위 수)
- 비율: 비율 분포의 평균(현재/기준)
- 비율 표준 편차: 비율 분포의 표준 편차(현재/기준)
- 1밀리초: 1밀리초(0.001초)

.NET을 사용한 벤치마킹에 관한 자세한 내용은 BenchmarkDotNet 을 참조하세요.

`StringValues` 형식

StringValues 개체는 효율적인 방식으로 `null`, 0, 하나 또는 여러 개의 문자열을 나타내는 `struct` 형식입니다. `StringValues` 형식은 `string?` 또는 `string?[]?` 구문 중 하나를 사용하여 생성할 수 있습니다. 이전 예제의 텍스트를 사용하여 다음 C# 코드를 살펴봅니다.

C#

```
StringValues values =
    new(s_nineHundredAutoGeneratedParagraphsOfLoremIpsum.Split(
        new[] { '\n' }));

Console.WriteLine($"Count = {values.Count:#,#}");

foreach (string? value in values)
{
    // Interact with the value
}
```

```
// Outputs:
//     Count = 1,799
```

앞의 코드는 문자열 값의 배열이 제공된 경우 `StringValues` 개체를 인스턴스화합니다. 는 StringValues.Count 콘솔에 기록됩니다.

`StringValues` 형식은 다음 컬렉션 형식의 구현입니다.

- `IList<string>`
- `ICollection<string>`
- `IEnumerable<string>`
- `IEnumerable`
- `IReadOnlyList<string>`
- `IReadOnlyCollection<string>`

따라서 반복될 수 있으며 필요에 따라 각 `value`가 상호 작용할 수 있습니다.

참고 항목

- .NET의 옵션 패턴
- .NET의 구성
- .NET의 로깅 공급자

.NET 애플리케이션 전역화 및 지역화

2025. 06. 17.

하나 이상의 언어로 지역화할 수 있는 애플리케이션을 포함하여 세계화가 가능한 애플리케이션을 개발하는 데는 세계화, 지역화 가능성 검토 및 지역화의 세 가지 단계가 포함됩니다.

세계화

이 단계에서는 문화권 중립적이고 언어 중립적이며 모든 사용자에 대해 지역화된 사용자 인터페이스 및 지역 데이터를 지원하는 애플리케이션을 디자인하고 코딩하는 작업이 포함됩니다. 여기에는 문화권별 가정을 기반으로 하지 않는 디자인 및 프로그래밍 결정이 포함됩니다. 전역화된 애플리케이션은 지역화되지 않지만, 이후에 비교적 쉽게 하나 이상의 언어로 지역화될 수 있도록 설계되고 작성됩니다.

지역화 가능성 검토

이 단계에서는 애플리케이션의 코드 및 디자인을 검토하여 쉽게 지역화할 수 있는지 확인하고 지역화에 대한 잠재적인 장애물을 식별하고 애플리케이션의 실행 코드가 해당 리소스와 분리되어 있는지 확인합니다. 세계화 단계가 유효하면 지역화 검토에서 세계화 중에 선택한 디자인 및 코딩을 확인합니다. 지역화 단계에서는 지역화 단계에서 애플리케이션의 소스 코드를 수정할 필요가 없도록 나머지 문제를 식별할 수도 있습니다.

지역화

이 단계에서는 특정 문화권 또는 지역에 대한 애플리케이션을 사용자 지정합니다. 세계화 및 지역화 단계가 올바르게 수행된 경우 지역화는 주로 사용자 인터페이스 번역으로 구성됩니다.

다음 세 단계를 수행하면 두 가지 이점이 있습니다.

- 이를 통해 미국 영어와 같은 단일 문화를 지원하도록 설계된 애플리케이션을 개조하여 추가 문화를 지원할 필요가 없습니다.
- 이로 인해 지역화된 애플리케이션이 더 안정적이고 버그가 적습니다.

.NET은 세계적 준비 및 지역화된 애플리케이션 개발을 광범위하게 지원합니다. 특히 .NET 클래스 라이브러리의 많은 형식 멤버는 현재 사용자의 문화권 또는 지정된 문화권의 규칙을 반영하는 값을 반환하여 세계화를 지원합니다. 또한 .NET은 위성 어셈블리를 지원하여 애플리케이션을 지역화하는 프로세스를 용이하게 합니다.

이 부분에서는

세계화

문화권 중립적이고 언어 중립적인 애플리케이션을 디자인하고 코딩하는 세계적 지원 애플리케이션을 만드는 첫 번째 단계에 대해 설명합니다.

.NET 세계화 및 ICU

.NET 세계화에서 ICU(International Components for Unicode)를 사용하는 방법을 설명합니다.

지역화 가능성 검토

지역화에 대한 잠재적인 장애물을 식별하는 지역화된 애플리케이션을 만드는 두 번째 단계를 설명합니다.

지역화

특정 지역 또는 문화권에 대한 애플리케이션의 사용자 인터페이스를 사용자 지정하는 지역화된 애플리케이션을 만드는 마지막 단계에 대해 설명합니다.

문화권을 구분하지 않는 문자열 작업

기본적으로 문화권에 민감한 .NET 메서드 및 클래스를 사용하여 문화권에 둔감한 결과를 얻는 방법을 설명합니다.

세계적 지원 애플리케이션 개발을 위한 모범 사례

세계화, 지역화 및 세계화가 가능한 ASP.NET 애플리케이션 개발을 위해 따라야 하는 모범 사례를 설명합니다.

참고 문헌

- System.Globalization 네임스페이스

  언어, 국가/지역, 사용 중인 달력, 날짜, 통화 및 숫자에 대한 형식 패턴 및 문자열의 정렬 순서를 포함하여 문화권 관련 정보를 정의하는 클래스를 포함합니다.

- System.Resources 네임스페이스

  리소스를 만들고 조작하고 사용하기 위한 클래스를 제공합니다.

- System.Text 네임스페이스

  ASCII, ANSI, 유니코드 및 기타 문자 인코딩을 나타내는 클래스를 포함합니다.

- Resgen.exe(리소스 파일 생성기)

  Resgen.exe 사용하여 .txt 파일 및 XML 기반 리소스 형식(.resx) 파일을 공용 언어 런타임 이진 .resources 파일로 변환하는 방법을 설명합니다.

- Winres.exe(Windows Forms 리소스 편집기)

  Winres.exe 사용하여 Windows Forms 양식을 지역화하는 방법을 설명합니다.

세계화

2025. 06. 17.

세계화에는 여러 문화권의 사용자를 위해 지역화된 인터페이스 및 지역 데이터를 지원하는 세계화가 가능한 앱을 디자인하고 개발하는 작업이 포함됩니다. 디자인 단계를 시작하기 전에 앱에서 지원할 문화권을 결정해야 합니다. 앱은 단일 문화권 또는 지역을 기본값으로 대상으로 하지만 다른 문화권 또는 지역의 사용자로 쉽게 확장할 수 있도록 디자인하고 작성할 수 있습니다.

개발자로서 우리 모두는 문화권에 의해 형성된 사용자 인터페이스 및 데이터에 대한 가정을 가지고 있습니다. 예를 들어 미국에서 영어를 구사하는 개발자의 경우 날짜 및 시간 데이터를 형식 `MM/dd/yyyy hh:mm:ss` 의 문자열로 직렬화하는 것이 매우 합리적입니다. 그러나 다른 문화권의 시스템에서 해당 문자열을 역직렬화하면 예외가 FormatException 발생하거나 부정확한 데이터가 생성됩니다. 세계화를 사용하면 이러한 문화권별 가정을 식별하고 앱의 디자인이나 코드에 영향을 미치지 않도록 할 수 있습니다.

이 문서에서는 고려해야 할 몇 가지 주요 문제와 전역화된 앱에서 문자열, 날짜 및 시간 값 및 숫자 값을 처리할 때 수행할 수 있는 모범 사례를 설명합니다.

현악기들

각 문화권 또는 지역에서 서로 다른 문자와 문자 집합을 사용하고 다르게 정렬할 수 있으므로 문자 및 문자열 처리는 세계화의 중심입니다. 이 섹션에서는 세계화된 앱에서 문자열을 사용하기 위한 권장 사항을 제공합니다.

내부적으로 유니코드 사용

기본적으로 .NET은 유니코드 문자열을 사용합니다. 유니코드 문자열은 각각 UTF-16 코드 단위를 나타내는 0개, 하나 이상의 Char 개체로 구성됩니다. 전 세계에서 사용 중인 모든 문자 집합의 거의 모든 문자에 대한 유니코드 표현이 있습니다.

Windows 운영 체제를 비롯한 많은 애플리케이션 및 운영 체제는 코드 페이지를 사용하여 문자 집합을 나타낼 수도 있습니다. 코드 페이지는 일반적으로 0x00 0x7F 표준 ASCII 값을 포함하고 다른 문자를 0x80 0xFF 나머지 값에 매핑합니다. 0x80 0xFF 값의 해석은 특정 코드 페이지에 따라 달라집니다. 이 때문에 가능하면 전역화된 앱에서 코드 페이지를 사용하지 않아야 합니다.

다음 예제에서는 시스템의 기본 코드 페이지가 데이터를 저장한 코드 페이지와 다를 때 코드 페이지 데이터를 해석하는 위험을 보여 줍니다. (이 시나리오를 시뮬레이션하기 위해 예제에서는 다른 코드 페이지를 명시적으로 지정합니다.) 먼저 이 예제에서는 그리스어 알파벳의 대문자로 구성된 배열을 정의합니다. 코드 페이지 737(MS-DOS 그리스어라고도 함)을 사용하여 바이트 배열로 인코딩하고 바이트 배열을 파일에 저장합니다. 파일이 검색되고 해당 바이트 배열이 코드 페이지 737을 사용하여 디코딩되면 원래 문자가 복원됩니다. 그러나 파일이 검색되고 바이트 배

열이 코드 페이지 1252(또는 라틴 알파벳의 문자를 나타내는 Windows-1252)를 사용하여 디코딩되는 경우 원래 문자는 손실됩니다.

C#

```
using System;
using System.IO;
using System.Text;

public class Example
{
    public static void CodePages()
    {
        // Represent Greek uppercase characters in code page 737.
        char[] greekChars =
        {
            'Α', 'Β', 'Γ', 'Δ', 'Ε', 'Ζ', 'Η', 'Θ',
            'Ι', 'Κ', 'Λ', 'Μ', 'Ν', 'Ξ', 'Ο', 'Π',
            'Ρ', 'Σ', 'Τ', 'Υ', 'Φ', 'Χ', 'Ψ', 'Ω'
        };

        Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);

        Encoding cp737 = Encoding.GetEncoding(737);
        int nBytes = cp737.GetByteCount(greekChars);
        byte[] bytes737 = new byte[nBytes];
        bytes737 = cp737.GetBytes(greekChars);
        // Write the bytes to a file.
        FileStream fs = new FileStream(@".\\CodePageBytes.dat", FileMode.Create);
        fs.Write(bytes737, 0, bytes737.Length);
        fs.Close();

        // Retrieve the byte data from the file.
        fs = new FileStream(@".\\CodePageBytes.dat", FileMode.Open);
        byte[] bytes1 = new byte[fs.Length];
        fs.Read(bytes1, 0, (int)fs.Length);
        fs.Close();

        // Restore the data on a system whose code page is 737.
        string data = cp737.GetString(bytes1);
        Console.WriteLine(data);
        Console.WriteLine();

        // Restore the data on a system whose code page is 1252.
        Encoding cp1252 = Encoding.GetEncoding(1252);
        data = cp1252.GetString(bytes1);
        Console.WriteLine(data);
    }
}

// The example displays the following output:
//       ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ
```

```
//       €,ƒ„…†‡ˆ‰Š‹ŒŽ‘’""•––
```

유니코드를 사용하면 동일한 코드 단위가 항상 동일한 문자에 매핑되고 동일한 문자가 항상 동일한 바이트 배열에 매핑됩니다.

리소스 파일 사용

단일 문화권 또는 지역을 대상으로 하는 앱을 개발하는 경우에도 리소스 파일을 사용하여 사용자 인터페이스에 표시되는 문자열 및 기타 리소스를 저장해야 합니다. 코드에 직접 추가해서는 안 됩니다. 리소스 파일을 사용하는 경우 다음과 같은 여러 가지 이점이 있습니다.

- 모든 문자열은 단일 위치에 있습니다. 특정 언어 또는 문화권에 대해 수정할 문자열을 식별하기 위해 소스 코드 전체에서 검색할 필요가 없습니다.
- 문자열을 복제할 필요가 없습니다. 리소스 파일을 사용하지 않는 개발자는 종종 여러 소스 코드 파일에서 동일한 문자열을 정의합니다. 이 중복은 문자열이 수정될 때 하나 이상의 인스턴스가 간과될 확률을 높입니다.
- 이미지 또는 이진 데이터와 같은 문자열이 아닌 리소스를 별도의 독립 실행형 파일에 저장하는 대신 리소스 파일에 포함할 수 있으므로 쉽게 검색할 수 있습니다.

리소스 파일을 사용하는 경우 지역화된 앱을 만드는 경우 특히 이점이 있습니다. 위성 어셈블리에 리소스를 배포할 때, 공용 언어 런타임은 사용자의 현재 UI 문화권을 CultureInfo.CurrentUICulture 속성에 따라 정의하고, 문화를 적절히 반영하는 리소스를 자동으로 선택합니다. 적절한 문화권별 리소스를 제공하고 개체를 올바르게 인스턴스화 ResourceManager 하거나 강력한 형식의 리소스 클래스를 사용하는 한 런타임은 적절한 리소스 검색의 세부 정보를 처리합니다.

리소스 파일을 만드는 방법에 대한 자세한 내용은 리소스 파일 만들기를 참조하세요. 위성 어셈블리를 만들고 배포하는 방법에 대한 자세한 내용은 위성 어셈블리 만들기 및 리소스 패키지 및 배포를 참조하세요.

문자열 검색 및 비교

가능하면 문자열을 일련의 개별 문자로 처리하는 대신 전체 문자열로 처리해야 합니다. 이는 결합된 문자 구문 분석과 관련된 문제를 방지하기 위해 부분 문자열을 정렬하거나 검색할 때 특히 중요합니다.

> **팁**
>
> 클래스를 사용하여 문자열의 StringInfo 개별 문자가 아닌 텍스트 요소를 사용할 수 있습니다.

문자열 검색 및 비교에서 일반적인 실수는 문자열을 각 개체가 나타내는 Char 문자 컬렉션으로 처리하는 것입니다. 실제로 단일 문자는 하나, 둘 이상의 Char 개체로 형성될 수 있습니다. 이러한 문자는 알파벳이 유니코드 기본 라틴 문자 범위(U+0021~ U+007E) 외부의 문자로 구성된 문화권의 문자열에서 가장 자주 발견됩니다. 다음 예제에서는 문자열에서 라틴 대문자 A 위에 물음표가 있는 문자(U+00C0)의 인덱스를 찾습니다. 그러나 이 문자는 단일 코드 단위(U+00C0) 또는 복합 문자(U+0041 및 U+0300의 두 코드 단위)로 두 가지 방법으로 나타낼 수 있습니다. 이 경우 문자는 문자열 인스턴스에서 U+0041 및 U+0300이라는 두 개체 Char 로 표시됩니다. 예제 코드는 문자열 인스턴스에서 String.IndexOf(Char) 이 문자의 위치를 찾기 위해 오버로드 및 String.IndexOf(String) 오버로드를 호출하지만 다른 결과를 반환합니다. 첫 번째 메서드 호출에는 인수가 Char 있으므로 서수 비교를 수행하므로 일치 항목을 찾을 수 없습니다. 두 번째 호출에는 인수가 String 있습니다. 이 호출은 문화권 구분 비교를 수행하므로 일치 항목을 찾습니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example17
{
    public static void Main17()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("pl-PL");
        string composite = "\u0041\u0300";
        Console.WriteLine($"Comparing using Char:
{composite.IndexOf('\u00C0')}");
        Console.WriteLine($"Comparing using String:
{composite.IndexOf("\u00C0")}");
    }
}

// The example displays the following output:
//       Comparing using Char:   -1
//       Comparing using String: 0
```

StringComparison 매개변수를 포함한 오버로드, 예를 들어 String.IndexOf(String, StringComparison) 또는 String.LastIndexOf(String, StringComparison) 메서드를 호출하면, 서로 다른 결과를 반환하는 유사한 오버로드 두 개를 호출할 때 이 예제의 모호성을 방지할 수 있습니다.

그러나 검색이 항상 문화권을 구분하는 것은 아닙니다. 검색의 목적이 보안 결정을 내리거나 일부 리소스에 대한 액세스를 허용하거나 허용하지 않는 경우 다음 섹션에서 설명한 대로 비교가 서수여야 합니다.

문자열이 같은지 테스트

정렬 순서에서 비교하는 방법을 결정하는 대신 두 문자열을 같음으로 테스트하려면 문자열 비교 메서드(예: String.Equals 또는 String.Compare) 대신 메서드를 사용합니다 CompareInfo.Compare.

같음 비교는 일반적으로 일부 리소스에 조건부로 액세스하기 위해 수행됩니다. 예를 들어 같음 비교를 수행하여 암호를 확인하거나 파일이 있는지 확인할 수 있습니다. 이러한 비언어적 비교는 항상 문화적 민감성을 반영하기보다는 서수적이어야 합니다. 일반적으로 String.Equals(String, StringComparison) 값을 사용하여 암호와 같은 문자열에 대해 인스턴스 String.Equals(String, String, StringComparison) 메서드를 호출하고, 파일 이름 또는 URI와 같은 문자열에 대해 StringComparison.Ordinal 값을 사용하여 정적 StringComparison.OrdinalIgnoreCase 메서드를 호출해야 합니다.

같음 비교에는 메서드에 대한 호출이 아닌 검색 또는 부분 문자열 비교가 String.Equals 포함되는 경우가 있습니다. 경우에 따라 부분 문자열 검색을 사용하여 해당 부분 문자열이 다른 문자열과 같은지 여부를 확인할 수 있습니다. 이 비교의 목적이 비언어적이라면, 검색은 문화에 민감한 것보다 서수이어야 합니다.

다음 예제에서는 비언어적 데이터에 대한 문화권 구분 검색의 위험을 보여 줍니다. 이 `AccessesFileSystem` 메서드는 부분 문자열 "FILE"로 시작하는 URI에 대한 파일 시스템 액세스를 금지하도록 설계되었습니다. URI의 시작 부분을 문자열 "FILE"과 문화권에 민감하되 대/소문자 구분 없이 비교하는 작업을 수행합니다. 파일 시스템에 액세스하는 URI는 "FILE:" 또는 "file:"로 시작할 수 있으므로 암시적 가정은 "i"(U+0069)가 항상 "I"(U+0049)와 같은 소문자라는 것입니다. 그러나 터키어와 아제르바이잔어에서는 "i"의 대문자 버전이 "1"(U+0130)입니다. 이러한 불일치로 인해 문화권 구분 비교를 사용하면 금지해야 하는 경우 파일 시스템에 액세스할 수 있습니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example10
{
    public static void Main10()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("tr-TR");
        string uri = @"file:\\c:\users\username\Documents\bio.txt";
        if (!AccessesFileSystem(uri))
            // Permit access to resource specified by URI
            Console.WriteLine("Access is allowed.");
        else
            // Prohibit access.
            Console.WriteLine("Access is not allowed.");
    }
```

```
    private static bool AccessesFileSystem(string uri)
    {
        return uri.StartsWith("FILE", true, CultureInfo.CurrentCulture);
    }
}

// The example displays the following output:
//         Access is allowed.
```

다음 예제와 같이 대/소문자를 무시하는 서수 비교를 수행하여 이 문제를 방지할 수 있습니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example11
{
    public static void Main11()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("tr-TR");
        string uri = @"file:\\c:\users\username\Documents\bio.txt";
        if (!AccessesFileSystem(uri))
            // Permit access to resource specified by URI
            Console.WriteLine("Access is allowed.");
        else
            // Prohibit access.
            Console.WriteLine("Access is not allowed.");
    }

    private static bool AccessesFileSystem(string uri)
    {
        return uri.StartsWith("FILE", StringComparison.OrdinalIgnoreCase);
    }
}

// The example displays the following output:
//         Access is not allowed.
```

문자열 순서 및 정렬

일반적으로 사용자 인터페이스에 표시할 순서가 지정된 문자열은 문화권에 따라 정렬되어야 합니다. 대부분의 경우 문자열과 같은 Array.SortList<T>.Sort문자열을 정렬하는 메서드를 호출할 때 .NET에서 이러한 문자열 비교를 암시적으로 처리합니다. 기본적으로 문자열은 현재 문화권의 정렬 규칙을 사용하여 정렬됩니다. 다음 예제에서는 영어(미국) 문화권과 스웨덴어(스웨덴) 문화권의 규칙을 사용하여 문자열 배열을 정렬할 때의 차이를 보여 줍니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example18
{
    public static void Main18()
    {
        string[] values = { "able", "ångström", "apple", "Æble",
                            "Windows", "Visual Studio" };
        // Change thread to en-US.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        // Sort the array and copy it to a new array to preserve the order.
        Array.Sort(values);
        string[] enValues = (String[])values.Clone();

        // Change culture to Swedish (Sweden).
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("sv-SE");
        Array.Sort(values);
        string[] svValues = (String[])values.Clone();

        // Compare the sorted arrays.
        Console.WriteLine("{0,-8} {1,-15} {2,-15}\n", "Position", "en-US", "sv-
SE");
        for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
            Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues[ctr],
svValues[ctr]);
    }
}

// The example displays the following output:
//       Position en-US           sv-SE
//
//       0        able            able
//       1        Æble            Æble
//       2        ångström        apple
//       3        apple           Windows
//       4        Visual Studio   Visual Studio
//       5        Windows         ångström
```

문화에 민감한 문자열 비교는 각 문화권의 속성 CompareInfo에 의해 반환되는 개체 CultureInfo.CompareInfo로 정의됩니다. 메서드 오버로드를 사용하는 String.Compare 문화권 구분 문자열 비교는 CompareInfo 개체도 사용합니다.

.NET에서는 테이블을 사용하여 문자열 데이터에 대해 문화권 구분 정렬을 수행합니다. 정렬 가중치 및 문자열 정규화에 대한 데이터를 포함하는 이러한 테이블의 내용은 특정 버전의 .NET에서 구현된 유니코드 표준 버전에 따라 결정됩니다. 다음 표에서는 지정된 버전의 .NET에서 구현

한 유니코드 버전을 나열합니다. 지원되는 유니코드 버전 목록은 문자 비교 및 정렬에만 적용됩니다. 범주별 유니코드 문자 분류에는 적용되지 않습니다. 자세한 내용은 문서의 "문자열 및 유니코드 표준" 섹션을 String 참조하세요.

테이블 확장

.NET Framework 버전	운영 체제	유니코드 버전
.NET Framework 2.0	모든 운영 체제	유니코드 4.1
.NET Framework 3.0	모든 운영 체제	유니코드 4.1
.NET Framework 3.5	모든 운영 체제	유니코드 4.1
.NET Framework 4	모든 운영 체제	유니코드 5.0
.NET Framework 4.5 이상	Windows 7	유니코드 5.0
.NET Framework 4.5 이상	Windows 8 이상 운영 체제	유니코드 6.3.0
.NET Core 및 .NET 5 이상		기본 OS에서 지원하는 유니코드 표준의 버전에 따라 달라집니다.

.NET Framework 4.5부터 모든 버전의 .NET Core 및 .NET 5 이상에서 문자열 비교 및 정렬은 운영 체제에 따라 달라집니다. Windows 7에서 실행되는 .NET Framework 4.5 이상은 유니코드 5.0을 구현하는 자체 테이블에서 데이터를 검색합니다. Windows 8 이상에서 실행되는 .NET Framework 4.5 이상은 유니코드 6.3을 구현하는 운영 체제 테이블에서 데이터를 검색합니다. .NET Core 및 .NET 5 이상에서 지원되는 유니코드 버전은 기본 운영 체제에 따라 달라집니다. 문화권에 민감한 정렬된 데이터를 직렬화하는 경우 클래스를 사용하여 SortVersion .NET 및 운영 체제의 정렬 순서와 일치하도록 직렬화된 데이터를 정렬해야 하는 시기를 결정할 수 있습니다. 예제를 보시려면 SortVersion 클래스 항목을 참조하세요.

앱이 문화권별로 문자열 데이터를 정렬해야 하는 경우, SortKey 클래스를 사용하여 문자열을 비교할 수 있습니다. 정렬 키는 특정 문자열의 알파벳, 대/소문자 및 발음 가중치를 포함하여 문화권별 정렬 가중치를 반영합니다. 정렬 키를 사용하는 비교는 이진이므로 개체를 암시적으로 또는 명시적으로 사용하는 CompareInfo 비교보다 빠릅니다. 메서드에 문자열 CompareInfo.GetSortKey 을 전달하여 특정 문자열에 대한 문화권별 정렬 키를 만듭니다.

다음 예제는 이전 예제와 비슷합니다. 그러나 메서드 Array.Sort(Array)를 암시적으로 호출하는 CompareInfo.Compare 메서드를 호출하는 대신, 정렬 키를 비교하기 위한 System.Collections.Generic.IComparer<T> 구현을 정의하고 이를 인스턴스화하여 Array.Sort<T>(T[], IComparer<T>) 메서드에 전달합니다.

C#

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Threading;

public class SortKeyComparer : IComparer<String>
{
    public int Compare(string? str1, string? str2)
    {
        return (str1, str2) switch
        {
            (null, null) => 0,
            (null, _) => -1,
            (_, null) => 1,
            (var s1, var s2) => SortKey.Compare(
                CultureInfo.CurrentCulture.CompareInfo.GetSortKey(s1),
                CultureInfo.CurrentCulture.CompareInfo.GetSortKey(s1))
        };
    }
}

public class Example19
{
    public static void Main19()
    {
        string[] values = { "able", "ångström", "apple", "Æble",
                            "Windows", "Visual Studio" };
        SortKeyComparer comparer = new SortKeyComparer();

        // Change thread to en-US.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        // Sort the array and copy it to a new array to preserve the order.
        Array.Sort(values, comparer);
        string[] enValues = (String[])values.Clone();

        // Change culture to Swedish (Sweden).
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("sv-SE");
        Array.Sort(values, comparer);
        string[] svValues = (String[])values.Clone();

        // Compare the sorted arrays.
        Console.WriteLine("{0,-8} {1,-15} {2,-15}\n", "Position", "en-US", "sv-
SE");
        for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
            Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues[ctr],
svValues[ctr]);
    }
}

// The example displays the following output:
//       Position en-US           sv-SE
//
```

```
//       0        able            able
//       1        Æble            Æble
//       2        ångström        apple
//       3        apple           Windows
//       4        Visual Studio   Visual Studio
//       5        Windows         ångström
```

문자열 연결 방지

가능한 경우 연결된 구에서 런타임에 빌드된 복합 문자열을 사용하지 마세요. 복합 문자열은 다른 지역화된 언어에 적용되지 않는 앱의 원래 언어로 문법 순서를 가정하는 경우가 많기 때문에 지역화하기가 어렵습니다.

날짜 및 시간 처리

날짜 및 시간 값을 처리하는 방법은 사용자 인터페이스에 표시되는지 또는 유지되는지에 따라 달라집니다. 이 섹션에서는 두 가지 사용량을 모두 살펴봅니다. 또한 날짜 및 시간을 사용할 때 표준 시간대 차이 및 산술 연산을 처리하는 방법에 대해서도 설명합니다.

날짜 및 시간 표시

일반적으로 날짜 및 시간이 사용자 인터페이스에 표시될 때, CultureInfo.CurrentCulture 속성인 사용자 문화권의 서식 규칙과 DateTimeFormatInfo 속성에 의해 반환되는 `CultureInfo.CurrentCulture.DateTimeFormat` 개체를 사용해야 합니다. 현재 문화권의 서식 지정 규칙은 다음 메서드를 사용하여 날짜 서식을 지정할 때 자동으로 사용됩니다.

- 매개 변수가 없는 DateTime.ToString() 메서드입니다.
- DateTime.ToString(String) 형식 문자열을 포함하는 메서드입니다.
- 매개 변수가 없는 DateTimeOffset.ToString() 메서드입니다.
- 형식 문자열을 포함하는 DateTimeOffset.ToString(String)입니다.
- 복합 서식 지정 기능은 날짜와 함께 사용할 때 유용합니다.

다음은 2012년 10월 11일의 일출 및 일몰 데이터를 두 번 표시하는 예제입니다. 먼저 현재 문화를 크로아티아어(크로아티아)로 설정한 다음 영어(영국)로 설정합니다. 각 경우에 날짜와 시간은 해당 문화권에 적합한 형식으로 표시됩니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example3
{
```

```
    static DateTime[] dates = { new DateTime(2012, 10, 11, 7, 06, 0),
                        new DateTime(2012, 10, 11, 18, 19, 0) };

    public static void Main3()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("hr-HR");
        ShowDayInfo();
        Console.WriteLine();
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
        ShowDayInfo();
    }

    private static void ShowDayInfo()
    {
        Console.WriteLine($"Date: {dates[0]:D}");
        Console.WriteLine($"   Sunrise: {dates[0]:T}");
        Console.WriteLine($"   Sunset:  {dates[1]:T}");
    }
}

// The example displays the following output:
//       Date: 11. listopada 2012.
//          Sunrise: 7:06:00
//          Sunset:  18:19:00
//
//       Date: 11 October 2012
//          Sunrise: 07:06:00
//          Sunset:  18:19:00
```

날짜 및 시간 유지

문화권에 따라 달라질 수 있는 형식으로 날짜 및 시간 데이터를 유지해서는 안 됩니다. 이는 손상된 데이터 또는 런타임 예외를 초래하는 일반적인 프로그래밍 오류입니다. 다음 예제에서는 영어(미국) 문화권의 서식 규칙을 사용하여 2013년 1월 9일과 2013년 8월 18일의 두 날짜를 문자열로 직렬화합니다. 영어(미국) 문화권의 규칙을 사용하여 데이터를 검색하고 구문 분석하면 성공적으로 복원됩니다. 그러나 영어(영국) 문화권의 규칙을 사용하여 검색 및 구문 분석할 때 첫 번째 날짜는 9월 1일로 잘못 해석되고 두 번째 날짜는 그레고리오력에 18번째 달이 없기 때문에 구문 분석에 실패합니다.

C#

```
using System;
using System.IO;
using System.Globalization;
using System.Threading;

public class Example4
{
```

```
    public static void Main4()
    {
        // Persist two dates as strings.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        DateTime[] dates = { new DateTime(2013, 1, 9),
                             new DateTime(2013, 8, 18) };
        StreamWriter sw = new StreamWriter("dateData.dat");
        sw.Write("{0:d}|{1:d}", dates[0], dates[1]);
        sw.Close();

        // Read the persisted data.
        StreamReader sr = new StreamReader("dateData.dat");
        string dateData = sr.ReadToEnd();
        sr.Close();
        string[] dateStrings = dateData.Split('|');

        // Restore and display the data using the conventions of the en-US
culture.
        Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, out restoredDate))
                Console.WriteLine($"The date is {restoredDate:D}");
            else
                Console.WriteLine($"ERROR: Unable to parse {dateStr}");
        }
        Console.WriteLine();

        // Restore and display the data using the conventions of the en-GB
culture.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
        Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, out restoredDate))
                Console.WriteLine($"The date is {restoredDate:D}");
            else
                Console.WriteLine($"ERROR: Unable to parse {dateStr}");
        }
    }
}

// The example displays the following output:
//       Current Culture: English (United States)
//       The date is Wednesday, January 09, 2013
//       The date is Sunday, August 18, 2013
//
//       Current Culture: English (United Kingdom)
```

```
//       The date is 01 September 2013
//       ERROR: Unable to parse 8/18/2013
```

다음 세 가지 방법 중에서 이 문제를 방지할 수 있습니다.

- 날짜와 시간을 문자열이 아닌 이진 형식으로 직렬화합니다.
- 사용자의 문화권에 관계없이 동일한 사용자 지정 형식 문자열을 사용하여 날짜 및 시간의 문자열 표현을 저장하고 구문 분석합니다.
- 고정 문화권의 서식 규칙을 사용하여 문자열을 저장합니다.

다음 예제에서는 마지막 방법을 보여 줍니다. 정적 CultureInfo.InvariantCulture 속성이 반환하는 불변 문화권의 서식 지정 규칙을 사용합니다.

C#

```
using System;
using System.IO;
using System.Globalization;
using System.Threading;

public class Example5
{
    public static void Main5()
    {
        // Persist two dates as strings.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        DateTime[] dates = { new DateTime(2013, 1, 9),
                             new DateTime(2013, 8, 18) };
        StreamWriter sw = new StreamWriter("dateData.dat");
        sw.Write(String.Format(CultureInfo.InvariantCulture,
                               "{0:d}|{1:d}", dates[0], dates[1]));
        sw.Close();

        // Read the persisted data.
        StreamReader sr = new StreamReader("dateData.dat");
        string dateData = sr.ReadToEnd();
        sr.Close();
        string[] dateStrings = dateData.Split('|');

        // Restore and display the data using the conventions of the en-US
culture.
        Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, CultureInfo.InvariantCulture,
                                  DateTimeStyles.None, out restoredDate))
                Console.WriteLine($"The date is {restoredDate:D}");
            else
```

```
                Console.WriteLine($"ERROR: Unable to parse {dateStr}");
        }
        Console.WriteLine();

        // Restore and display the data using the conventions of the en-GB
culture.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
        Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, CultureInfo.InvariantCulture,
                                  DateTimeStyles.None, out restoredDate))
                Console.WriteLine($"The date is {restoredDate:D}");
            else
                Console.WriteLine($"ERROR: Unable to parse {dateStr}");
        }
    }
}

// The example displays the following output:
//       Current Culture: English (United States)
//       The date is Wednesday, January 09, 2013
//       The date is Sunday, August 18, 2013
//
//       Current Culture: English (United Kingdom)
//       The date is 09 January 2013
//       The date is 18 August 2013
```

직렬화 및 표준 시간대 인식

날짜 및 시간 값은 일반 시간("2013년 1월 2일 오전 9:00에 매장 오픈")에서 특정 시간("생년월일: 2013년 1월 2일 오전 6:32:00")에 이르기까지 여러 해석을 가질 수 있습니다. 시간 값이 특정 시간을 나타내고 직렬화된 값에서 복원하는 경우 사용자의 지리적 위치 또는 표준 시간대에 관계없이 동일한 시간을 나타내는지 확인해야 합니다.

다음 예제에서는 이 문제를 보여 줍니다. 단일 로컬 날짜 및 시간 값을 세 가지 표준 형식의 문자열로 저장합니다.

- "G"는 일반적인 날짜와 긴 시간 형식을 나타냅니다.
- 정렬 가능한 날짜/시간을 나타내는 "s"입니다.
- 왕복 날짜/시간에 대한 「o」입니다.

C#

```
using System;
using System.IO;
```

```
public class Example6
{
    public static void Main6()
    {
        DateTime dateOriginal = new DateTime(2023, 3, 30, 18, 0, 0);
        dateOriginal = DateTime.SpecifyKind(dateOriginal, DateTimeKind.Local);

        // Serialize a date.
        if (!File.Exists("DateInfo.dat"))
        {
            StreamWriter sw = new StreamWriter("DateInfo.dat");
            sw.Write("{0:G}|{0:s}|{0:o}", dateOriginal);
            sw.Close();
            Console.WriteLine("Serialized dates to DateInfo.dat");
        }
        Console.WriteLine();

        // Restore the date from string values.
        StreamReader sr = new StreamReader("DateInfo.dat");
        string datesToSplit = sr.ReadToEnd();
        string[] dateStrings = datesToSplit.Split('|');
        foreach (var dateStr in dateStrings)
        {
            DateTime newDate = DateTime.Parse(dateStr);
            Console.WriteLine($"'{dateStr}' --> {newDate} {newDate.Kind}");
        }
    }
}
```

데이터가 직렬화된 시스템과 동일한 표준 시간대의 시스템에서 복원되면 역직렬화된 날짜 및 시간 값은 출력에 표시된 대로 원래 값을 정확하게 반영합니다.

콘솔

```
'3/30/2013 6:00:00 PM' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00.0000000-07:00' --> 3/30/2013 6:00:00 PM Local
```

그러나 다른 표준 시간대의 시스템에서 데이터를 복원하는 경우 "o"(왕복) 표준 형식 문자열로 형식이 지정된 날짜 및 시간 값만 표준 시간대 정보를 유지하므로 동일한 인스턴트 시간을 나타냅니다. 로맨스 표준 시간대의 시스템에서 날짜 및 시간 데이터가 복원되는 경우의 출력은 다음과 같습니다.

콘솔

```
'3/30/2023 6:00:00 PM' --> 3/30/2023 6:00:00 PM Unspecified
'2023-03-30T18:00:00' --> 3/30/2023 6:00:00 PM Unspecified
'2023-03-30T18:00:00.0000000-07:00' --> 3/31/2023 3:00:00 AM Local
```

데이터가 역직렬화되는 시스템의 표준 시간대에 관계없이 단일 시간을 나타내는 날짜 및 시간 값을 정확하게 반영하려면 다음 중 하나라도 수행할 수 있습니다.

- "o"(왕복) 표준 형식 문자열을 사용하여 값을 문자열로 저장합니다. 그런 다음 대상 시스템에서 역직렬화합니다.
- "r"(RFC1123) 표준 형식 문자열을 사용하여 UTC로 변환하고 문자열로 저장합니다. 그런 다음 대상 시스템에서 역직렬화하고 현지 시간으로 변환합니다.
- UTC로 변환하고 "u"(범용 정렬 가능) 표준 형식 문자열을 사용하여 문자열로 저장합니다. 그런 다음 대상 시스템에서 역직렬화하고 현지 시간으로 변환합니다.

다음 예제에서는 각 기술을 보여 줍니다.

C#

```
using System;
using System.IO;

public class Example9
{
    public static void Main9()
    {
        // Serialize a date.
        DateTime dateOriginal = new DateTime(2023, 3, 30, 18, 0, 0);
        dateOriginal = DateTime.SpecifyKind(dateOriginal, DateTimeKind.Local);

        // Serialize the date in string form.
        if (!File.Exists("DateInfo2.dat"))
        {
            StreamWriter sw = new StreamWriter("DateInfo2.dat");
            sw.Write("{0:o}|{1:r}|{1:u}", dateOriginal,
                                          dateOriginal.ToUniversalTime());
            sw.Close();
        }

        // Restore the date from string values.
        StreamReader sr = new StreamReader("DateInfo2.dat");
        string datesToSplit = sr.ReadToEnd();
        string[] dateStrings = datesToSplit.Split('|');
        for (int ctr = 0; ctr < dateStrings.Length; ctr++)
        {
            DateTime newDate = DateTime.Parse(dateStrings[ctr]);
            if (ctr == 1)
            {
                Console.WriteLine($"'{dateStrings[ctr]}' --> {newDate}
{newDate.Kind}");
            }
            else
            {
                DateTime newLocalDate = newDate.ToLocalTime();
                Console.WriteLine($"'{dateStrings[ctr]}' --> {newLocalDate}
{newLocalDate.Kind}");
            }
```

```
        }
    }
}
```

데이터가 Pacific Standard 표준 시간대의 시스템에서 직렬화되고 로맨스 표준 시간대의 시스템에서 역직렬화되는 경우 예제에서는 다음 출력을 표시합니다.

콘솔

```
'2023-03-30T18:00:00.0000000-07:00' --> 3/31/2023 3:00:00 AM Local
'Sun, 31 Mar 2023 01:00:00 GMT' --> 3/31/2023 3:00:00 AM Local
'2023-03-31 01:00:00Z' --> 3/31/2023 3:00:00 AM Local
```

자세한 내용은 표준 시간대 간 시간 변환을 참조하세요.

날짜 및 시간 산술 연산 수행

DateTime 두 DateTimeOffset 형식 모두 산술 연산을 지원합니다. 두 날짜 값 간의 차이를 계산하거나 날짜 값에 특정 시간 간격을 추가하거나 뺄 수 있습니다. 그러나 날짜 및 시간 값에 대한 산술 연산은 표준 시간대 및 표준 시간대 조정 규칙을 고려하지 않습니다. 이로 인해 시간 단위를 나타내는 값에 대한 날짜 및 시간 산술 연산은 부정확한 결과를 반환할 수 있습니다.

예를 들어 태평양 표준시에서 태평양 일광 절약 시간으로의 전환은 2013년 3월 10일인 3월 둘째 일요일에 발생합니다. 다음 예제에서 볼 수 있듯이 태평양 표준시 표준 시간대의 시스템에서 2013년 3월 9일 오전 10시 30분에 48시간 이후의 날짜 및 시간을 계산하는 경우 결과는 2013년 3월 11일 오전 10시 30분에 수행되는 시간 조정을 고려하지 않습니다.

C#

```
using System;

public class Example7
{
    public static void Main7()
    {
        DateTime date1 = DateTime.SpecifyKind(new DateTime(2013, 3, 9, 10, 30, 0),
                                              DateTimeKind.Local);
        TimeSpan interval = new TimeSpan(48, 0, 0);
        DateTime date2 = date1 + interval;
        Console.WriteLine($"{date1:g} + {interval.TotalHours:N1} hours =
{date2:g}");
    }
}

// The example displays the following output:
//       3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 10:30 AM
```

날짜 및 시간 값에 대한 산술 연산이 정확한 결과를 생성하도록 하려면 다음 단계를 수행합니다.

1. 원본 표준 시간대의 시간을 UTC로 변환합니다.
2. 산술 연산을 수행합니다.
3. 결과가 날짜 및 시간 값인 경우 UTC에서 원본 표준 시간대의 시간으로 변환합니다.

다음 예제는 2013년 3월 9일 오전 10시 30분에 48시간을 올바르게 추가하기 위해 다음 세 단계를 수행한다는 점을 제외하고 이전 예제와 비슷합니다.

C#

```
using System;

public class Example8
{
    public static void Main8()
    {
        TimeZoneInfo pst = TimeZoneInfo.FindSystemTimeZoneById("Pacific Standard
Time");
        DateTime date1 = DateTime.SpecifyKind(new DateTime(2013, 3, 9, 10, 30, 0),
                                              DateTimeKind.Local);
        DateTime utc1 = date1.ToUniversalTime();
        TimeSpan interval = new TimeSpan(48, 0, 0);
        DateTime utc2 = utc1 + interval;
        DateTime date2 = TimeZoneInfo.ConvertTimeFromUtc(utc2, pst);
        Console.WriteLine($"{date1:g} + {interval.TotalHours:N1} hours =
{date2:g}");
    }
}

// The example displays the following output:
//        3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 11:30 AM
```

자세한 내용은 날짜 및 시간을 사용하여 산술 연산 수행을 참조하세요.

날짜 요소에 문화권 구분 이름 사용

앱에서 월 또는 요일의 이름을 표시해야 할 수 있습니다. 이렇게 하려면 다음과 같은 코드가 일반적입니다.

C#

```
using System;

public class Example12
{
   public static void Main12()
   {
      DateTime midYear = new DateTime(2013, 7, 1);
```

```
      Console.WriteLine($"{midYear:d} is a {GetDayName(midYear)}.");
   }

   private static string GetDayName(DateTime date)
   {
      return date.DayOfWeek.ToString("G");
   }
}

// The example displays the following output:
//        7/1/2013 is a Monday.
```

그러나 이 코드는 항상 요일의 이름을 영어로 반환합니다. 월의 이름을 추출하는 코드는 종종 훨씬 더 유연하지 않습니다. 특정 언어로 된 월 이름이 있는 12개월 달력을 자주 가정합니다.

사용자 지정 날짜 및 시간 형식 문자열 또는 개체의 DateTimeFormatInfo 속성을 사용하면 다음 예제와 같이 사용자 문화권에서 요일 또는 월의 이름을 반영하는 문자열을 쉽게 추출할 수 있습니다. 현재 언어 설정을 프랑스어(프랑스)로 변경하고, 2013년 7월 1일의 요일과 달 이름을 표시합니다.

C#

```
using System;
using System.Globalization;

public class Example13
{
    public static void Main13()
    {
        // Set the current culture to French (France).
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR");

        DateTime midYear = new DateTime(2013, 7, 1);
        Console.WriteLine($"{midYear:d} is a
{DateUtilities.GetDayName(midYear)}.");
        Console.WriteLine($"{midYear:d} is a
{DateUtilities.GetDayName((int)midYear.DayOfWeek)}.");
        Console.WriteLine($"{midYear:d} is in
{DateUtilities.GetMonthName(midYear)}.");
        Console.WriteLine($"{midYear:d} is in
{DateUtilities.GetMonthName(midYear.Month)}.");
    }
}

public static class DateUtilities
{
    public static string GetDayName(int dayOfWeek)
    {
        if (dayOfWeek < 0 | dayOfWeek >
DateTimeFormatInfo.CurrentInfo.DayNames.Length)
            return String.Empty;
        else
```

```
            return DateTimeFormatInfo.CurrentInfo.DayNames[dayOfWeek];
    }

    public static string GetDayName(DateTime date)
    {
        return date.ToString("dddd");
    }

    public static string GetMonthName(int month)
    {
        if (month < 1 | month > DateTimeFormatInfo.CurrentInfo.MonthNames.Length -
1)
            return String.Empty;
        else
            return DateTimeFormatInfo.CurrentInfo.MonthNames[month - 1];
    }

    public static string GetMonthName(DateTime date)
    {
        return date.ToString("MMMM");
    }
}

// The example displays the following output:
//       01/07/2013 is a lundi.
//       01/07/2013 is a lundi.
//       01/07/2013 is in juillet.
//       01/07/2013 is in juillet.
```

숫자 값

숫자 처리는 사용자 인터페이스에 표시되는지 또는 유지되는지에 따라 달라집니다. 이 섹션에서는 두 가지 사용량을 모두 살펴봅니다.

> ⓘ **참고**
>
> 구문 분석 및 서식 지정 작업에서 .NET은 기본 라틴 문자 0~9(U+0030~U+0039)만 숫자 숫자로 인식합니다.

숫자 값 표시

일반적으로 사용자 인터페이스에 숫자가 표시되는 경우, CultureInfo.CurrentCulture 속성과 NumberFormatInfo 속성에서 반환되는 `CultureInfo.CurrentCulture.NumberFormat` 개체에 의해 정의된 사용자 문화권의 서식 규칙을 사용해야 합니다. 현재 문화권의 서식 지정 규칙은 다음과 같은 방법으로 날짜 서식을 지정할 때 자동으로 사용됩니다.

- 모든 숫자 형식의 매개 변수 없는 `ToString` 메서드 사용
- 형식 문자열을 `ToString(String)` 인수로 포함하는 숫자 형식의 메서드를 사용합니다.
- 복합 서식을 사용하여 숫자 값을 표시합니다.

다음은 프랑스 파리의 월 평균 온도를 표시하는 예제입니다. 먼저 데이터를 표시하기 전에 현재 문화권을 프랑스어(프랑스)로 설정한 다음 영어(미국)로 설정합니다. 각 경우에 월 이름과 온도는 해당 문화권에 적합한 형식으로 표시됩니다. 두 문화권은 온도 값에 서로 다른 소수 구분 기호를 사용합니다. 또한 이 예제에서는 "MMMM" 사용자 지정 날짜 및 시간 형식 문자열을 사용하여 전체 월 이름을 표시하고 배열에서 가장 긴 월 이름의 길이를 결정하여 결과 문자열의 월 이름에 DateTimeFormatInfo.MonthNames 적절한 양의 공간을 할당합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example14
{
    public static void Main14()
    {
        DateTime dateForMonth = new DateTime(2013, 1, 1);
        double[] temperatures = {  3.4, 3.5, 7.6, 10.4, 14.5, 17.2,
                                  19.9, 18.2, 15.9, 11.3, 6.9, 5.3 };

        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");
        // Build the format string dynamically so we allocate enough space for the
month name.
        string fmtString = "{0,-" + GetLongestMonthNameLength().ToString() +
":MMMM}     {1,4}";
        for (int ctr = 0; ctr < temperatures.Length; ctr++)
            Console.WriteLine(fmtString,
                              dateForMonth.AddMonths(ctr),
                              temperatures[ctr]);

        Console.WriteLine();

        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");
        fmtString = "{0,-" + GetLongestMonthNameLength().ToString() + ":MMMM}
{1,4}";
        for (int ctr = 0; ctr < temperatures.Length; ctr++)
            Console.WriteLine(fmtString,
                              dateForMonth.AddMonths(ctr),
                              temperatures[ctr]);
    }
```

```
    private static int GetLongestMonthNameLength()
    {
        int length = 0;
        foreach (var nameOfMonth in DateTimeFormatInfo.CurrentInfo.MonthNames)
            if (nameOfMonth.Length > length) length = nameOfMonth.Length;

        return length;
    }
}

// The example displays the following output:
//    Current Culture: French (France)
//       janvier        3,4
//       février        3,5
//       mars           7,6
//       avril         10,4
//       mai           14,5
//       juin          17,2
//       juillet       19,9
//       août          18,2
//       septembre     15,9
//       octobre       11,3
//       novembre       6,9
//       décembre       5,3
//
//       Current Culture: English (United States)
//       January        3.4
//       February       3.5
//       March          7.6
//       April         10.4
//       May           14.5
//       June          17.2
//       July          19.9
//       August        18.2
//       September     15.9
//       October       11.3
//       November       6.9
//       December       5.3
```

숫자 값 유지

숫자 데이터를 문화권별 형식으로 유지해서는 안 됩니다. 이는 손상된 데이터 또는 런타임 예외를 초래하는 일반적인 프로그래밍 오류입니다. 다음 예제에서는 10개의 임의 부동 소수점 숫자를 생성한 다음 영어(미국) 문화권의 서식 규칙을 사용하여 문자열로 serialize합니다. 영어(미국) 문화권의 규칙을 사용하여 데이터를 검색하고 구문 분석하면 성공적으로 복원됩니다. 그러나 프랑스(프랑스) 문화권의 규칙을 사용하여 검색 및 구문 분석하는 경우 문화권이 서로 다른 소수 구분 기호를 사용하기 때문에 숫자를 구문 분석할 수 없습니다.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example15
{
    public static void Main15()
    {
        // Create ten random doubles.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        double[] numbers = GetRandomNumbers(10);
        DisplayRandomNumbers(numbers);

        // Persist the numbers as strings.
        StreamWriter sw = new StreamWriter("randoms.dat");
        for (int ctr = 0; ctr < numbers.Length; ctr++)
            sw.Write("{0:R}{1}", numbers[ctr], ctr < numbers.Length - 1 ? "|" :
"");

        sw.Close();

        // Read the persisted data.
        StreamReader sr = new StreamReader("randoms.dat");
        string numericData = sr.ReadToEnd();
        sr.Close();
        string[] numberStrings = numericData.Split('|');

        // Restore and display the data using the conventions of the en-US
culture.
        Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
        foreach (var numberStr in numberStrings)
        {
            double restoredNumber;
            if (Double.TryParse(numberStr, out restoredNumber))
                Console.WriteLine(restoredNumber.ToString("R"));
            else
                Console.WriteLine($"ERROR: Unable to parse '{numberStr}'");
        }
        Console.WriteLine();

        // Restore and display the data using the conventions of the fr-FR
culture.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
        foreach (var numberStr in numberStrings)
        {
            double restoredNumber;
            if (Double.TryParse(numberStr, out restoredNumber))
                Console.WriteLine(restoredNumber.ToString("R"));
```

```
            else
                Console.WriteLine($"ERROR: Unable to parse '{numberStr}'");
        }
    }

    private static double[] GetRandomNumbers(int n)
    {
        Random rnd = new Random();
        double[] numbers = new double[n];
        for (int ctr = 0; ctr < n; ctr++)
            numbers[ctr] = rnd.NextDouble() * 1000;
        return numbers;
    }

    private static void DisplayRandomNumbers(double[] numbers)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
            Console.WriteLine(numbers[ctr].ToString("R"));
        Console.WriteLine();
    }
}

// The example displays output like the following:
//       487.0313743534644
//       674.12000879371533
//       498.72077885024288
//       42.3034229512808
//       970.57311049223563
//       531.33717716268131
//       587.82905693530529
//       562.25210175023039
//       600.7711019370571
//       299.46113717717174
//
//       Current Culture: English (United States)
//       487.0313743534644
//       674.12000879371533
//       498.72077885024288
//       42.3034229512808
//       970.57311049223563
//       531.33717716268131
//       587.82905693530529
//       562.25210175023039
//       600.7711019370571
//       299.46113717717174
//
//       Current Culture: French (France)
//       ERROR: Unable to parse '487.0313743534644'
//       ERROR: Unable to parse '674.12000879371533'
//       ERROR: Unable to parse '498.72077885024288'
//       ERROR: Unable to parse '42.3034229512808'
//       ERROR: Unable to parse '970.57311049223563'
//       ERROR: Unable to parse '531.33717716268131'
//       ERROR: Unable to parse '587.82905693530529'
//       ERROR: Unable to parse '562.25210175023039'
```

```
//       ERROR: Unable to parse '600.7711019370571'
//       ERROR: Unable to parse '299.46113717717174'
```

이 문제를 방지하려면 다음 기술 중 하나를 사용할 수 있습니다.

- 사용자 문화권에 관계없이 동일한 사용자 지정 형식 문자열을 사용하여 숫자의 문자열 표현을 저장하고 구문 분석합니다.
- CultureInfo.InvariantCulture 속성에서 반환된 고정 문화권의 서식 규칙을 사용하여 숫자를 문자열로 저장합니다.

통화 값을 직렬화하는 것은 특별한 경우입니다. 통화 값은 표현되는 통화 단위에 따라 달라지므로 독립적인 숫자 값으로 취급하는 것은 의미가 없습니다. 그러나 통화 값을 통화 기호를 포함하는 형식이 지정된 문자열로 저장하는 경우 다음 예제와 같이 기본 문화권이 다른 통화 기호를 사용하는 시스템에서 역직렬화할 수 없습니다.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example1
{
   public static void Main1()
   {
      // Display the currency value.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-
US");
      Decimal value = 16039.47m;
      Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");
      Console.WriteLine($"Currency Value: {value:C2}");

      // Persist the currency value as a string.
      StreamWriter sw = new StreamWriter("currency.dat");
      sw.Write(value.ToString("C2"));
      sw.Close();

      // Read the persisted data using the current culture.
      StreamReader sr = new StreamReader("currency.dat");
      string currencyData = sr.ReadToEnd();
      sr.Close();

      // Restore and display the data using the conventions of the current
culture.
      Decimal restoredValue;
      if (Decimal.TryParse(currencyData, out restoredValue))
         Console.WriteLine(restoredValue.ToString("C2"));
      else
         Console.WriteLine($"ERROR: Unable to parse '{currencyData}'");
```

```
      Console.WriteLine();

      // Restore and display the data using the conventions of the en-GB culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-
GB");
      Console.WriteLine($"Current Culture:
{Thread.CurrentThread.CurrentCulture.DisplayName}");
      if (Decimal.TryParse(currencyData, NumberStyles.Currency, null, out
restoredValue))
         Console.WriteLine(restoredValue.ToString("C2"));
      else
         Console.WriteLine($"ERROR: Unable to parse '{currencyData}'");
      Console.WriteLine();
   }
}
// The example displays output like the following:
//       Current Culture: English (United States)
//       Currency Value: $16,039.47
//       ERROR: Unable to parse '$16,039.47'
//
//       Current Culture: English (United Kingdom)
//       ERROR: Unable to parse '$16,039.47'
```

대신 값과 해당 통화 기호를 현재 문화권과 독립적으로 역직렬화할 수 있도록 문화권 이름과 같은 일부 문화권 정보와 함께 숫자 값을 직렬화해야 합니다. 다음 예제에서는 `CurrencyValue` 구조체를 정의하여 두 멤버, 즉 Decimal 값과 그 값이 속한 문화권의 이름을 포함합니다.

C#

```
using System;
using System.Globalization;
using System.Text.Json;
using System.Threading;

public class Example2
{
    public static void Main2()
    {
        // Display the currency value.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Decimal value = 16039.47m;
        Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");
        Console.WriteLine($"Currency Value: {value:C2}");

        // Serialize the currency data.
        CurrencyValue data = new()
        {
            Amount = value,
            CultureName = CultureInfo.CurrentCulture.Name
        };
```

```
        string serialized = JsonSerializer.Serialize(data);
        Console.WriteLine();

        // Change the current culture.
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB");
        Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");

        // Deserialize the data.
        CurrencyValue restoredData = JsonSerializer.Deserialize<CurrencyValue>
(serialized);

        // Display the round-tripped value.
        CultureInfo culture =
CultureInfo.CreateSpecificCulture(restoredData.CultureName);
        Console.WriteLine($"Currency Value: {restoredData.Amount.ToString("C2",
culture)}");
    }
}

internal struct CurrencyValue
{
    public decimal Amount { get; set; }
    public string CultureName { get; set; }
}

// The example displays the following output:
//       Current Culture: English (United States)
//       Currency Value: $16,039.47
//
//       Current Culture: English (United Kingdom)
//       Currency Value: $16,039.47
```

문화별 설정 작업

.NET에서 클래스는 CultureInfo 특정 문화권 또는 지역을 나타냅니다. 일부 속성은 문화권의 일부 측면에 대한 특정 정보를 제공하는 개체를 반환합니다.

- 이 속성은 CultureInfo.CompareInfo 문화권이 문자열을 비교하고 정렬하는 방법에 대한 정보가 포함된 개체를 반환 CompareInfo 합니다.
- 이 속성은 CultureInfo.DateTimeFormat 날짜 및 시간 데이터의 서식 지정에 사용되는 문화권별 정보를 제공하는 개체를 반환 DateTimeFormatInfo 합니다.
- CultureInfo.NumberFormat 속성은 숫자 데이터의 서식 지정을 위해 문화권별 정보를 제공하는 NumberFormatInfo 개체를 반환합니다.
- 이 CultureInfo.TextInfo 속성은 문화권의 문자 체계에 대한 정보를 제공하는 TextInfo 개체를 반환합니다.

일반적으로 특정 CultureInfo 속성 및 관련 개체의 값에 대해 어떠한 가정도 하지 마세요. 대신 다음과 같은 이유로 문화권별 데이터를 변경될 수 있는 것으로 확인해야 합니다.

- 개별 속성 값은 시간이 지남에 따라 변경 및 수정될 수 있으며, 데이터가 수정되거나, 더 나은 데이터를 사용할 수 있게 되거나, 문화권별 규칙이 변경됩니다.
- 개별 속성 값은 .NET 버전 또는 운영 체제 버전에 따라 다를 수 있습니다.
- .NET은 대체 문화권을 지원합니다. 이렇게 하면 기존 표준 문화권을 보완하거나 기존 표준 문화권을 완전히 대체하는 새 사용자 지정 문화권을 정의할 수 있습니다.
- Windows 시스템에서 사용자는 제어판의 **지역 및 언어** 앱을 사용하여 문화권별 설정을 사용자 지정할 수 있습니다. 개체를 CultureInfo 인스턴스화할 때 생성자를 호출 CultureInfo(String, Boolean) 하여 이러한 사용자 지정을 반영하는지 여부를 확인할 수 있습니다. 일반적으로 최종 사용자 앱의 경우 사용자가 예상한 형식으로 데이터를 표시할 수 있도록 사용자 기본 설정을 준수해야 합니다.

참고하십시오

- 세계화 및 지역화
- 문자열 사용하기 위한 모범 사례

.NET 세계화 및 ICU

아티클 • 2024. 12. 20.

.NET 5 이전에는 .NET 세계화 API가 서로 다른 플랫폼에서 서로 다른 기본 라이브러리를 사용했습니다. Unix에서는 ICU(International Components for Unicode)를 사용했고, Windows에서는 NLS(National Language Support)를 사용했습니다. 이로 인해 서로 다른 플랫폼에서 애플리케이션을 실행할 때 일부 세계화 API의 동작이 약간 달랐습니다. 다음 영역에서는 동작이 확실히 달랐습니다.

- 문화권 및 문화권 데이터
- 문자열 대/소문자 구분
- 문자열 정렬 및 검색
- 정렬 키
- 문자열 정규화
- IDN(다국어 도메인 이름) 지원
- Linux의 표준 시간대 표시 이름

.NET 5부터는 사용되는 기본 라이브러리에 대한 개발자의 제어가 향상되면서 애플리케이션의 플랫폼 간 차이를 방지할 수 있게 되었습니다.

> **참고**
>
> ICU 라이브러리의 동작을 구동하는 문화권 데이터는 일반적으로 런타임이 아닌 CLDR(Common Locale Data Repository)에서 유지 관리됩니다.

Windows의 ICU

이제 Windows는 미리 설치된 icu.dll 버전을 세계화 작업에 자동으로 적용되는 기능의 일부로 통합합니다. 이 수정을 통해 .NET은 이 ICU 라이브러리를 세계화 지원에 사용할 수 있습니다. 이전 Windows 버전의 경우와 마찬가지로 ICU 라이브러리를 사용할 수 없거나 로드할 수 없는 경우 .NET 5 및 후속 버전은 NLS 기반 구현을 사용하도록 되돌아갑니다.

다음 표에서는 여러 Windows 클라이언트 및 서버 버전에서 ICU 라이브러리를 로드할 수 있는 .NET 버전을 보여 줍니다.

테이블 확장

.NET 버전	Windows 버전
.NET 5 또는 .NET 6	Windows 클라이언트 10 버전 1903 이상

.NET 버전	Windows 버전
.NET 5 또는 .NET 6	Windows Server 2022 이상
.NET 7 이상	Windows 클라이언트 10 버전 1703 이상
.NET 7 이상	Windows Server 2019 이상

> ⓘ **참고**
>
> .NET 7 이상 버전에는 .NET 6 및 .NET 5와 달리 이전 Windows 버전에서 ICU를 로드하는 기능이 있습니다.

> ⓘ **참고**
>
> ICU를 사용하는 경우에도 `CurrentCulture`, `CurrentUICulture`, `CurrentRegion` 멤버는 여전히 Windows 운영 체제 API를 사용하여 사용자 설정을 적용합니다.

동작의 차이

.NET 5 이상을 대상으로 앱을 업그레이드하는 경우 세계화 기능을 사용하고 있다는 사실을 모르더라도 앱에 변경 내용이 표시될 수 있습니다. 다음 섹션에서는 발생할 수 있는 몇 가지 동작 변경 내용을 나열합니다.

문자열 정렬 및 `System.Globalization.CompareOptions`

`CompareOptions`는 두 문자열을 비교하는 방법에 영향을 주기 위해 `String.Compare`에 전달될 수 있는 옵션 열거형입니다.

문자열이 같은지 비교하고 정렬 순서를 결정하는 방법은 NLS와 ICU 간에 다릅니다. 특히 다음 사항에 주의하십시오.

- 기본 문자열 정렬 순서는 다르므로 `CompareOptions`를 직접 사용하지 않더라도 명확하게 나타납니다 ICU를 사용하는 경우 `None` 기본 옵션은 `StringSort`와 동일한 작업을 수행합니다. `StringSort`는 영숫자가 아닌 문자를 영숫자 앞에 정렬합니다(예: "bill's"는 "bill" 앞에 나옴). 이전 `None` 기능을 복원하려면 NLS 기반 구현을 사용해야 합니다.
- 합자 문자의 기본 처리는 다릅니다. NLS에서는 합자와 비합자(예: "oeuf" 및 "œuf")가 동일한 것으로 간주되지만 .NET의 ICU에서는 그렇지 않습니다. 이것은 두 구현

간의 데이터 정렬 강도가 다르기 때문입니다. ICU를 사용할 때 NLS 동작을 복원하려면 `CompareOptions.IgnoreNonSpace` 값을 사용합니다.

String.IndexOf

문자열에서 null 문자 String.IndexOf(String) 인덱스를 찾기 위해 `\0`(을)를 호출하는 다음 코드를 고려합니다.

C#

```
const string greeting = "Hel\0lo";
Console.WriteLine($"{greeting.IndexOf("\0")}");
Console.WriteLine($"{greeting.IndexOf("\0",
StringComparison.CurrentCulture)}");
Console.WriteLine($"{greeting.IndexOf("\0", StringComparison.Ordinal)}");
```

- Windows의 .NET Core 3.1 이전 버전에서 코드 조각은 세 줄 각각에 `3`(을)를 인쇄합니다.
- Windows 섹션 테이블의 ICU에 나열된 Windows 버전에서 실행되는 .NET 5 및 최신 버전의 경우 코드 조각은 서수 검색을 위해 `0`, `0` 및 `3`(을)를 인쇄합니다.

기본적으로 String.IndexOf(String)(이)가 문화권 인식 언어 검색을 수행합니다. ICU는 null 문자 `\0`(을)를 *0 가중치 문자*로 간주하므로 .NET 5 이상에서 언어 검색을 사용하는 경우 문자열에서 문자를 찾을 수 없습니다. 그러나 NLS는 null 문자 `\0`(을)를 0 가중치 문자로 간주하지 않으며 .NET Core 3.1 이하에서 언어 검색을 통해 위치 3에서 문자를 찾습니다. 서수 검색은 모든 .NET 버전에서 위치 3의 문자를 찾습니다.

코드 분석 규칙 CA1307:명확성을 위해 StringComparison 지정 및 CA1309: 서수 StringComparison 사용을 실행하여 코드에서 문자열 비교가 지정되지 않았거나 서수가 아닌 호출 사이트를 찾을 수 있습니다.

자세한 내용은 .NET 5+에서 문자열 비교 시 동작 변경을 참조하세요.

String.EndsWith

C#

```
const string foo = "abc";

Console.WriteLine(foo.EndsWith("\0"));
Console.WriteLine(foo.EndsWith("c"));
Console.WriteLine(foo.EndsWith("\0", StringComparison.CurrentCulture));
Console.WriteLine(foo.EndsWith("\0", StringComparison.Ordinal));
Console.WriteLine(foo.EndsWith('\0'));
```

> ⓘ 중요
>
> Windows 테이블의 ICU에 나열된 Windows 버전에서 실행되는 .NET 5 이상에서는 위의 코드 조각이 인쇄됩니다.
>
> Output
>
> ```
> True
> True
> True
> False
> False
> ```

이 동작을 방지하려면 `char` 매개 변수 오버로드 또는 `StringComparison.Ordinal`(을)를 사용합니다.

String.StartsWith

C#

```
const string foo = "abc";

Console.WriteLine(foo.StartsWith("\0"));
Console.WriteLine(foo.StartsWith("a"));
Console.WriteLine(foo.StartsWith("\0", StringComparison.CurrentCulture));
Console.WriteLine(foo.StartsWith("\0", StringComparison.Ordinal));
Console.WriteLine(foo.StartsWith('\0'));
```

> ⓘ 중요
>
> Windows 테이블의 ICU에 나열된 Windows 버전에서 실행되는 .NET 5 이상에서는 위의 코드 조각이 인쇄됩니다.
>
> Output
>
> ```
> True
> True
> True
> False
> False
> ```

이 동작을 방지하려면 `char` 매개 변수 오버로드 또는 `StringComparison.Ordinal`(을)를 사용합니다.

TimeZoneInfo.FindSystemTimeZoneById

ICU는 애플리케이션이 Windows에서 실행되는 경우에도 TimeZoneInfo표준 시간대 ID를 사용하여 인스턴스를 유연하게 만들 수 있습니다. 마찬가지로 Windows 이외의 플랫폼에서 실행되는 경우에도 Windows 표준 시간대 ID를 사용하여 TimeZoneInfo 인스턴스를 만들 수 있습니다. 그러나 NLS 모드를 사용하거나 세계화 고정 모드를 사용하는 경우 이 기능을 사용할 수 없다는 점에 유의해야 합니다.

요일의 약어

DateTimeFormatInfo.GetShortestDayName(DayOfWeek) 메서드는 지정된 요일에 해당하는 가장 짧은 약식 요일 이름을 가져옵니다.

- Windows의 .NET Core 3.1 및 이전 버전에서 이러한 요일 약어는 "Su"와 같이 두 문자로 구성되었습니다.
- .NET 5 이상 버전에서 이러한 요일 약어는 "S"와 같이 하나의 문자로만 구성됩니다.

ICU 종속 API

.NET은 ICU에 종속된 API를 도입했습니다. 이러한 API는 ICU를 사용하는 경우에만 성공할 수 있습니다. 다음 몇 가지 예를 참조하세요.

- TryConvertIanaIdToWindowsId(String, String)
- TryConvertWindowsIdToIanaId

Windows의 ICU 섹션 표에 나열된 Windows 버전에서는 언급된 API가 성공합니다. 그러나 이전 버전의 Windows에서는 이러한 API가 실패합니다. 이러한 경우 앱 로컬 ICU 기능을 사용하도록 설정하여 이러한 API의 성공을 보장할 수 있습니다. Windows가 아닌 플랫폼에서 이러한 API는 버전에 관계없이 항상 성공합니다.

또한 앱이 세계화 고정 모드 또는 NLS 모드로 실행되지 않도록 하여 이러한 API의 성공을 보장하는 것이 중요합니다.

ICU 대신 NLS 사용

NLS 대신 ICU를 사용하면 일부 세계화 관련 작업에서 동작 차이가 발생할 수 있습니다. NLS를 다시 사용하려면 ICU 구현을 옵트아웃하면 됩니다. 애플리케이션은 다음 방법으로 NLS 모드를 사용하도록 설정할 수 있습니다.

- 프로젝트 파일에서:

```
XML
```

```
<ItemGroup>
  <RuntimeHostConfigurationOption Include="System.Globalization.UseNls"
Value="true" />
</ItemGroup>
```

- `runtimeconfig.json` 파일에서:

```JSON
{
  "runtimeOptions": {
     "configProperties": {
       "System.Globalization.UseNls": true
      }
  }
}
```

- 환경 변수 `DOTNET_SYSTEM_GLOBALIZATION_USENLS`를 `true` 또는 `1`으로 설정.

> ⓘ **참고**
>
> 프로젝트 또는 `runtimeconfig.json` 파일에 설정된 값은 환경 변수보다 우선적으로 적용됩니다.

자세한 내용은 런타임 구성 설정을 참조하세요.

앱에서 ICU를 사용하고 있는지 확인

다음 코드 조각은 앱이 NLS가 아닌 ICU 라이브러리를 사용하여 실행 중인지 확인하는 데 도움이 될 수 있습니다.

```C#
public static bool ICUMode()
{
    SortVersion sortVersion =
CultureInfo.InvariantCulture.CompareInfo.Version;
    byte[] bytes = sortVersion.SortId.ToByteArray();
    int version = bytes[3] << 24 | bytes[2] << 16 | bytes[1] << 8 |
bytes[0];
    return version != 0 && version == sortVersion.FullVersion;
}
```

.NET 버전을 확인하려면 RuntimeInformation.FrameworkDescription(을)를 사용합니다.

앱 로컬 ICU

각 ICU 릴리스에는 버그 수정과 세계 언어를 설명하는 업데이트된 CLDR(Common Locale Data Repository) 데이터가 포함될 수 있습니다. ICU 버전 간 이동은 세계화 관련 작업과 관련하여 앱 동작에 미묘한 영향을 줄 수 있습니다. 애플리케이션 개발자가 모든 배포에서 일관성을 유지할 수 있도록 .NET 5 이상 버전에서는 Windows 및 Unix의 앱이 자체 ICU 복사본을 갖고 사용할 수 있습니다.

애플리케이션은 다음 방법 중 하나로 앱 로컬 ICU 구현 모드를 옵트인할 수 있습니다.

- 프로젝트 파일에서 적절한 `RuntimeHostConfigurationOption` 값을 설정합니다.

  XML

  ```
  <ItemGroup>
    <RuntimeHostConfigurationOption
  Include="System.Globalization.AppLocalIcu" Value="<suffix>:<version> or
  <version>" />
  </ItemGroup>
  ```

- 또는 *runtimeconfig.json* 파일에서 적절한 `runtimeOptions.configProperties` 값을 설정합니다.

  JSON

  ```
  {
    "runtimeOptions": {
       "configProperties": {
         "System.Globalization.AppLocalIcu": "<suffix>:<version> or
  <version>"
       }
    }
  }
  ```

- 또는 환경 변수 `DOTNET_SYSTEM_GLOBALIZATION_APPLOCALICU`(을)를 `<suffix>:<version>` 또는 `<version>`값으로 설정합니다.

  `<suffix>`: 공용 ICU 패키징 규칙에 따라 길이가 36자 미만인 선택적 접미사입니다. 사용자 지정 ICU를 빌드할 때 lib 이름을 생성하고 내보낸 기호 이름이 접미사를 포함하도록(예: `libicuucmyapp`, 여기서 `myapp`은 접미사) 사용자 지정할 수 있습니다.

  `<version>`: 유효한 ICU 버전(예: 67.1)입니다. 이 버전은 이진 파일을 로드하고 내보낸 기호를 가져오는 데 사용됩니다.

이러한 옵션 중 하나가 설정되면 구성된 에 해당하는 프로젝트에 `PackageReference` `version`를 추가할 수 있습니다.

또는 앱-로컬 스위치가 설정되면 ICU를 로드하기 위해 .NET은 여러 경로를 검색하는 NativeLibrary.TryLoad 메서드를 사용합니다. 메서드는 먼저 `NATIVE_DLL_SEARCH_DIRECTORIES` 속성에서 라이브러리를 찾으려고 시도합니다. 이 라이브러리는 앱의 `deps.json` 파일에 기반한 dotnet 호스트에서 만들어집니다. 자세한 내용은 기본 검색을 참조하세요.

자체 포함 앱의 경우 ICU가 앱 디렉터리에 있는지 확인하는 것 외에 특별한 작업이 필요하지 않습니다(자체 포함 앱의 경우 작업 디렉터리의 기본값은 `NATIVE_DLL_SEARCH_DIRECTORIES`).

NuGet 패키지를 통해 사용하는 ICU는 프레임워크 종속 애플리케이션에서 작동합니다. NuGet은 네이티브 자산을 확인하여 `deps.json` 파일 및 `runtimes` 디렉터리 아래에 있는 애플리케이션의 출력 디렉터리에 포함합니다. .NET은 여기에서 로드합니다.

ICU가 로컬 빌드에서 사용되는 프레임워크 종속 앱(자체 포함 아님)의 경우 추가 단계를 수행해야 합니다. .NET SDK에는 "느슨한" 네이티브 이진 파일을 `deps.json`에 통합하는 기능이 아직 없습니다(이 SDK 문제 참조). 대신 애플리케이션의 프로젝트 파일에 추가 정보를 추가하여 이 기능을 사용하도록 설정할 수 있습니다. 예시:

XML

```
<ItemGroup>
  <IcuAssemblies Include="icu\*.so*" />
  <RuntimeTargetsCopyLocalItems Include="@(IcuAssemblies)"
AssetType="native" CopyLocal="true"
    DestinationSubDirectory="runtimes/linux-x64/native/"
DestinationSubPath="%(FileName)%(Extension)"
    RuntimeIdentifier="linux-x64"
NuGetPackageId="System.Private.Runtime.UnicodeData" />
</ItemGroup>
```

지원되는 런타임의 모든 ICU 이진 파일에 이 작업을 수행해야 합니다. 또한 `NuGetPackageId` 항목 그룹의 `RuntimeTargetsCopyLocalItems` 메타데이터는 프로젝트가 실제로 참조하는 NuGet 패키지와 일치해야 합니다.

Linux에서 특정 ICU 버전 로드

기본적으로 Linux에서 ICU를 사용하는 경우 .NET은 시스템에서 설치된 최신 버전의 ICU를 로드하려고 시도합니다. 그러나 `DOTNET_ICU_VERSION_OVERRIDE` 환경 변수를 설정하여 로드할 특정 버전의 ICU를 지정할 수 있습니다.

예를 들어 환경 변수가 `67.1`같은 특정 버전 번호로 설정된 경우 .NET은 해당 버전의 ICU를 로드하려고 시도합니다. 예를 들어, .NET은 `libicuuc.so.67.1` 및 `libicui18n.so.67.1` 라이브러리를 찾습니다.

> ⓘ **참고**
>
> 이 환경 변수는 Microsoft에서 제공하는 .NET 빌드에서만 지원되며 Linux 배포판에서 제공하는 빌드에서는 지원되지 않습니다. .NET 10 이전 버전의 경우 환경 변수를 `CLR_ICU_VERSION_OVERRIDE`호출합니다.

지정된 버전을 찾을 수 없는 경우 .NET은 시스템에서 설치된 가장 높은 ICU 버전을 로드하는 것으로 돌아갑니다.

이 구성은 ICU 버전 사용량을 유연하게 제어하여 애플리케이션별 또는 시스템 제공 ICU 버전과의 호환성을 보장합니다.

macOS 동작

macOS는 `Mach-O` 파일에 지정된 부하 명령에서 종속 동적 라이브러리를 확인하는 동작이 Linux 로더와 다릅니다. Linux 로더에서 .NET은 ICU 종속성 그래프를 충족하기 위해 `libicudata`, `libicuuc`, `libicui18n`을 (이 순서로) 시도할 수 있습니다. 하지만 macOS에서는 이것이 작동하지 않습니다. macOS에서 ICU를 빌드하는 경우 기본적으로 `libicuuc`에서 이러한 load 명령을 사용하여 동적 라이브러리를 가져옵니다. 다음 코드 조각은 예제를 보여 줍니다.

```sh
~/ % otool -L /Users/santifdezm/repos/icu-
build/icu/install/lib/libicuuc.67.1.dylib
/Users/santifdezm/repos/icu-build/icu/install/lib/libicuuc.67.1.dylib:
 libicuuc.67.dylib (compatibility version 67.0.0, current version 67.1.0)
 libicudata.67.dylib (compatibility version 67.0.0, current version 67.1.0)
 /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
1281.100.1)
 /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version
902.1.0)
```

이러한 명령은 ICU의 다른 구성 요소에 대한 종속 라이브러리의 이름을 참조합니다. 로더는 `dlopen` 규칙에 따라 검색을 수행합니다. 그러려면 시스템 디렉터리에 이러한 라이브러리가 있거나 `LD_LIBRARY_PATH` 환경 변수를 설정하거나 앱 수준 디렉터리에 ICU가 있어야 합니다. `LD_LIBRARY_PATH`를 설정할 수 없거나 ICU 바이너리가 앱 수준 디렉터리에 있는지 확인할 수 없다면 몇 가지 추가 작업을 수행해야 합니다.

`@loader_path`와 같이 해당 load 명령을 사용하여 이진 파일과 동일한 디렉터리에서 해당 종속성을 검색하도록 로더에게 지시하는 로더를 위한 몇 가지 지시문이 있습니다. 두 가지 방법으로 이 작업을 수행할 수 있습니다.

- `install_name_tool -change`

  다음 명령을 실행합니다.

  ```
  Bash

  install_name_tool -change "libicudata.67.dylib"
  "@loader_path/libicudata.67.dylib" /path/to/libicuuc.67.1.dylib
  install_name_tool -change "libicudata.67.dylib"
  "@loader_path/libicudata.67.dylib" /path/to/libicui18n.67.1.dylib
  install_name_tool -change "libicuuc.67.dylib"
  "@loader_path/libicuuc.67.dylib" /path/to/libicui18n.67.1.dylib
  ```

- `@loader_path`로 설치 이름을 생성하도록 ICU 패치

  autoconf(`./runConfigureICU`)를 실행하기 전에 이 줄을 다음으로 변경합니다.

  ```

  LD_SONAME = -Wl,-compatibility_version -Wl,$(SO_TARGET_VERSION_MAJOR) -
  Wl,-current_version -Wl,$(SO_TARGET_VERSION) -install_name
  @loader_path/$(notdir $(MIDDLE_SO_TARGET))
  ```

WebAssembly의 ICU

WebAssembly 워크로드 전용인 ICU 버전을 사용할 수 있습니다. 이 버전은 데스크톱 프로필과의 세계화 호환성을 제공합니다. ICU 데이터 파일 크기를 24MB에서 1.4MB(또는 Brotli로 압축된 경우 0.3MB 이하)로 줄이기 위해 이 워크로드에는 몇 가지 제한 사항이 있습니다.

다음 API는 지원되지 않습니다.

- CultureInfo.EnglishName
- CultureInfo.NativeName
- DateTimeFormatInfo.NativeCalendarName
- RegionInfo.NativeName

다음 API는 제한적으로 지원됩니다.

- String.Normalize(NormalizationForm) 및 String.IsNormalized(NormalizationForm)는 거의 사용되지 않는 FormKC 및 FormKD 형식을 지원하지 않습니다.

- RegionInfo.CurrencyNativeName는 RegionInfo.CurrencyEnglishName와 동일한 값을 반환합니다.

또한 지원되는 로캘은 더 적습니다. 지원되는 목록은 dotnet/icu 리포지토리에서 찾을 수 있습니다.

.NET 앱의 세계화 설정

.NET 세계화 초기화는 적절한 세계화 라이브러리를 로드하고, 문화권 데이터를 설정하고, 세계화 설정을 구성하는 복잡한 프로세스입니다. 다음 섹션에서는 다양한 플랫폼에서 세계화 초기화가 작동하는 방식을 설명합니다.

Windows

Windows에서 .NET은 다음 단계에 따라 세계화를 초기화합니다.

- 세계화 불변 모드가 활성화되어 있는지 확인합니다. 이 모드가 활성화되면 .NET은 ICU 라이브러리 로드를 무시하고 NLS API를 사용하지 않습니다. 대신 기본 제공 고정 문화권 데이터를 사용하여 동작이 운영 체제 및 ICU 라이브러리와 완전히 독립적으로 유지되도록 합니다.

- NLS 모드 사용하도록 설정되어 있는지 확인합니다. 사용하도록 설정하면 .NET은 ICU 라이브러리 로드를 건너뛰고 대신 세계화 지원을 위해 Windows NLS API를 사용합니다.

- 앱 로컬 ICU 기능이 활성화되어 있는지 확인합니다. 이 경우 .NET은 지정된 버전을 라이브러리 이름에 추가하여 애플리케이션 디렉터리에서 ICU 라이브러리를 로드하려고 시도합니다. 예를 들어 버전이 72.1인 경우 .NET은 먼저 `icuuc72.dll`, `icuin72.dll` 및 `icudt72.dll`로드하려고 합니다. 이러한 라이브러리를 로드할 수 없는 경우 `icuuc72.1.dll`, `icuin72.1.dll` 및 `icudt72.1.dll`로드하려고 시도합니다. 라이브러리를 찾을 수 없는 경우 프로세스는 다음과 같은 오류 메시지(예: `Failed to load app-local ICU: {library name}`)로 종료됩니다.

- 이전 조건이 충족되지 않으면 .NET은 시스템 디렉터리에서 ICU 라이브러리를 로드하려고 시도합니다. 먼저 `icu.dll`로드하려고 시도합니다. 이 라이브러리를 사용할 수 없는 경우 시스템 디렉터리에서 `icuuc.dll` 및 `icuin.dll`을 로드하려고 시도합니다. 이러한 라이브러리를 찾을 수 없는 경우 런타임은 세계화 지원을 위해 NLS API를 사용하는 것으로 대체됩니다.

> **참고**

NLS API는 항상 모든 Windows 버전에서 사용할 수 있으므로 .NET은 항상 세계화 지원을 위해 다시 사용할 수 있습니다.

Linux

- 세계화 불변 모드가 활성화되어 있는지 확인합니다. 이 모드가 활성화되면 .NET은 ICU 라이브러리 로드를 무시합니다. 대신 기본 제공 고정 문화권 데이터를 사용하여 동작이 운영 체제 및 ICU 라이브러리와 완전히 독립적으로 유지되도록 합니다.
- 앱 로컬 ICU 기능이 활성화되어 있는지 확인합니다. 이 경우 .NET은 지정된 버전을 라이브러리 이름에 추가하여 애플리케이션 디렉터리에서 ICU 라이브러리를 로드하려고 시도합니다. 예를 들어 버전이 68.2.0.9인 경우 .NET은 `libicuuc.so.68.2.0.9`와 `libicui18n.so.68.2.0.9`을 로드하려고 시도합니다. 라이브러리를 찾을 수 없는 경우 프로세스는 다음과 같은 오류 메시지(예: `Failed to load app-local ICU: {library name}`)로 종료됩니다.
- `DOTNET_ICU_VERSION_OVERRIDE` 환경 변수가 설정되어 있는지 확인합니다. 이 경우 .NET은 에서 설명된 대로특정 ICU 버전을 Linux에서 로드하려고 시도합니다.
- 이전 조건이 충족되지 않으면 .NET은 시스템에서 설치된 ICU 라이브러리의 가장 높은 버전을 로드하려고 시도합니다. 시스템에 설치된 가장 높은 ICU 버전인 `libicuuc.so.[version]`에 따라 `libicui18n.so.[version]` 및 `[version]`라이브러리를 로드하려고 시도합니다. 라이브러리를 찾을 수 없으면 프로세스는 다음과 같은 오류 메시지(예: `Failed to load system ICU: {library name}`)로 종료됩니다.

macOS

- 세계화 불변 모드가 활성화되어 있는지 확인합니다. 이 모드가 활성화되면 .NET은 ICU 라이브러리 로드를 무시합니다. 대신 기본 제공 고정 문화권 데이터를 사용하여 동작이 운영 체제 및 ICU 라이브러리와 완전히 독립적으로 유지되도록 합니다.
- 앱 로컬 ICU 기능이 활성화되어 있는지 확인합니다. 이 경우 .NET은 지정된 버전을 라이브러리 이름에 추가하여 애플리케이션 디렉터리에서 ICU 라이브러리를 로드하려고 시도합니다. 예를 들어 버전이 68.2.0.9인 경우 .NET은 `libicuuc68.2.0.9.dylib`와 `libicui18n68.2.0.9.dylib`을 로드하려고 시도합니다. 라이브러리를 찾을 수 없는 경우 프로세스는 다음과 같은 오류 메시지(예: `Failed to load app-local ICU: {library name}`)로 종료됩니다.
- 위의 조건이 충족되지 않으면, .NET은 macOS 동작에 설명된 것처럼 설치된 ICU 라이브러리 버전을 로드하려고 시도합니다.

문화에 민감하지 않은 문자열 작업 수행

2025. 06. 17.

문화권별 문자열 작업은 문화권별로 사용자에게 결과를 표시하도록 설계된 애플리케이션을 만드는 경우에 유용합니다. 기본적으로 문화권 구분 메서드는 현재 스레드의 CurrentCulture 속성에서 사용할 문화권을 가져옵니다.

때때로 문화 민감성을 고려한 문자열 처리가 원하는 결과가 아닐 때도 있습니다. 결과가 문화권과 독립적이어야 하는 경우 문화권에 민감한 작업을 사용하면 사용자 지정 사례 매핑 및 정렬 규칙이 있는 문화권에서 애플리케이션 코드가 실패할 수 있습니다. 현재 문화권을 사용하는 문자열 비교 섹션을 문자열 사용에 대한 모범 사례에서 참조하세요.

문자열 작업이 문화권을 구분해야 하는지 또는 문화권을 구분하지 않아야 하는지 여부는 애플리케이션에서 결과를 사용하는 방법에 따라 달라집니다. 사용자에게 결과를 표시하는 문자열 작업은 일반적으로 문화에 맞게 조정해야 합니다. 예를 들어 애플리케이션이 목록 상자에 지역화된 문자열의 정렬된 목록을 표시하는 경우 애플리케이션은 문화권 구분 정렬을 수행해야 합니다.

내부적으로 사용되는 문자열 작업의 결과는 일반적으로 문화에 대해 민감하지 않아야 합니다. 일반적으로 애플리케이션이 사용자에게 표시되지 않는 파일 이름, 지속성 형식 또는 기호 정보로 작업하는 경우 문자열 작업의 결과는 문화권에 따라 달라지지 않아야 합니다. 예를 들어 애플리케이션이 문자열을 비교하여 인식된 XML 태그인지 여부를 확인하는 경우 비교는 문화권을 구분하지 않아야 합니다. 또한 보안 결정이 문자열 비교 또는 대/소문자 변경 작업의 결과를 기반으로 하는 경우, 결과가 CurrentCulture값에 영향을 받지 않도록 하기 위해 문화권에 구애받지 않아야 합니다.

기본적으로 문화권 구분 문자열 작업을 수행하는 대부분의 .NET 메서드는 문화권을 구분하지 않는 결과를 보장할 수 있는 오버로드도 제공합니다. 인수를 사용하는 CultureInfo 이러한 오버로드를 사용하면 매핑 및 정렬 규칙의 경우 문화권 변형을 제거할 수 있습니다. 문화권을 구분하지 않는 문자열 작업의 경우 문화권을 .로 CultureInfo.InvariantCulture지정합니다.

이 부분에서는

이 섹션의 문서에서는 기본적으로는 문화권에 따라 달라지는 .NET 메서드를 사용하여 문화권에 영향을 받지 않는 문자열 작업을 수행하는 방법을 보여 줍니다.

문화권을 구분하지 않는 문자열 비교 수행
String.Compare 및 String.CompareTo 메서드를 사용하여 문화에 관계없이 문자열 비교를 수행하는 방법을 설명합니다.

문화권을 구분하지 않는 대소문자 변경 수행
문화권에 영향을 받지 않는 대/소문자 변경을 수행하기 위해 String.ToUpper, String.ToLower,

Char.ToUpper, 및 Char.ToLower 메서드를 사용하는 방법을 설명합니다.

컬렉션에서 문화권을 구분하지 않는 문자열 작업 수행
CaseInsensitiveComparer, CaseInsensitiveHashCodeProvider 클래스, SortedList, ArrayList.Sort, CollectionsUtil.CreateCaseInsensitiveHashtable를 사용하여 컬렉션에서 문화권을 구분하지 않는 작업을 수행하는 방법을 설명합니다.

배열에서 문화권을 구분하지 않는 문자열 작업 수행
배열에서 문화권을 구분하지 않는 작업을 수행하기 위해 Array.Sort와 Array.BinarySearch 메서드를 사용하는 방법을 설명합니다.

참고하십시오

- 가중치 테이블 정렬(Windows 시스템의 .NET용)
- 기본 유니코드 데이터 정렬 요소 테이블(Linux 및 macOS의 .NET Core용)

문화에 민감하지 않은 문자열 비교 실행

아티클 • 2025. 04. 01.

기본적으로 String.Compare 메서드는 문화권 구분 및 대/소문자 구분 비교를 수행합니다. 이 메서드에는 사용할 문화권을 지정할 수 있는 `culture` 매개 변수를 제공하는 여러 오버로드와 사용할 비교 규칙을 지정할 수 있는 `comparisonType` 매개 변수도 포함됩니다. 기본 오버로드 대신 이러한 메서드를 호출하면 특정 메서드 호출에 사용되는 규칙에 대한 모호성이 제거되고 특정 비교가 문화권에 민감하거나 문화권을 구분하지 않는지 명확하게 알 수 있습니다.

> **참고**
>
> String.CompareTo 메서드의 오버로드는 모두 문화권 구분 및 대/소문자 구분 비교를 수행합니다. 문화권을 구분하지 않는 비교를 수행하려면 이 메서드를 사용할 수 없습니다. 코드 명확성을 위해 대신 String.Compare 메서드를 사용하는 것이 좋습니다.

문화권 구분 작업의 경우 StringComparison.CurrentCulture 또는 StringComparison.CurrentCultureIgnoreCase 열거형 값을 `comparisonType` 매개 변수로 지정합니다. 현재 문화권 이외의 지정된 문화권을 사용하여 문화권 구분 비교를 수행하려면 해당 문화권을 나타내는 CultureInfo 개체를 `culture` 매개 변수로 지정합니다.

String.Compare 메서드가 지원하는 문화권에 민감하지 않은 문자열 비교는 언어적 비교(고정 문화권의 정렬 규칙에 근거) 또는 비언어적 비교(문자열의 문자 서수 값을 기준으로)입니다. 대부분의 문화권을 구분하지 않는 문자열 비교는 비언어적입니다. 이러한 비교의 경우 StringComparison.Ordinal 또는 StringComparison.OrdinalIgnoreCase 열거형 값을 `comparisonType` 매개 변수로 지정합니다. 예를 들어 보안 결정(예: 사용자 이름 또는 암호 비교)이 문자열 비교의 결과를 기반으로 하는 경우 특정 문화권 또는 언어의 규칙에 의해 결과가 영향을 받지 않도록 하려면 문화권을 구분하지 않고 비언어적이어야 합니다.

여러 문화권의 언어 관련 문자열을 일관된 방식으로 처리하려면 문화권을 구분하지 않는 언어 문자열 비교를 사용합니다. 예를 들어 애플리케이션이 목록 상자에 여러 문자 집합을 사용하는 단어를 표시하는 경우 현재 문화권에 관계없이 동일한 순서로 단어를 표시할 수 있습니다. 문화권을 구분하지 않는 언어 비교의 경우 .NET은 영어의 언어 규칙을 기반으로 하는 고정 문화권을 정의합니다. 문화권을 구분하지 않는 언어 비교를 수행하려면 StringComparison.InvariantCulture 또는 StringComparison.InvariantCultureIgnoreCase `comparisonType` 매개 변수로 지정합니다.

다음 예제에서는 두 가지 문화권을 구분하지 않는 비언어적 문자열 비교를 수행합니다. 첫 번째는 대/소문자를 구분하지만 두 번째는 그렇지 않습니다.

C#

```
using System;

public class CompareSample
{
    public static void Main()
    {
        string string1 = "file";
        string string2 = "FILE";
        int compareResult = 0;

        compareResult = String.Compare(string1, string2,
                                       StringComparison.Ordinal);
        Console.WriteLine($"{StringComparison.Ordinal} comparison of
'{string1}' and '{string2}': {compareResult}");

        compareResult = String.Compare(string1, string2,
                                       StringComparison.OrdinalIgnoreCase);
        Console.WriteLine($"{StringComparison.OrdinalIgnoreCase} comparison
of '{string1}' and '{string2}': {compareResult}");
    }
}
// The example displays the following output:
//    Ordinal comparison of 'file' and 'FILE': 32
//    OrdinalIgnoreCase comparison of 'file' and 'FILE': 0
```

정렬 가중치 테이블, Windows 운영 체제의 정렬 및 비교 작업에 사용되는 문자 가중치에 대한 정보가 포함된 텍스트 파일 집합 및 Linux 및 macOS용 정렬 가중치 테이블인 기본 유니코드 데이터 정렬 요소 테이블 다운로드할 수 있습니다.

참고하십시오

- String.Compare
- String.CompareTo
- 문화에 구애받지 않는 문자열 작업을 수행하기
- 문자열 사용하기 위한 모범 사례

문화권에 구애받지 않고 대소문자 변환을 수행

String.ToUpper, String.ToLower, Char.ToUpper 및 Char.ToLower 메서드에는 어떤 매개변수도 허용하지 않는 오버로드가 있습니다. 기본적으로 매개 변수가 없는 이러한 오버로드는 값 CultureInfo.CurrentCulture에 따라 대/소문자를 변경합니다. 그 결과는 문화권에 따라 달라질 수 있는 대소문자 구분을 생성합니다. 대/소문자 변경을 문화에 민감하게 할지, 아니면 민감하지 않게 할지를 명확하게 하려면, 매개 변수를 명시적으로 지정해야 하는 이러한 메서드의 오버로드를 `culture` 사용해야 합니다. 문화권에 민감한 대/소문자 변경의 경우, `culture` 매개변수로 `CultureInfo.CurrentCulture`를 지정하십시오. 문화에 구애받지 않는 대/소문자 변경의 경우 `culture` 매개 변수에 `CultureInfo.InvariantCulture`을 지정하십시오.

나중에 더 쉽게 조회할 수 있도록 문자열이 표준 사례로 변환되는 경우가 많습니다. 이러한 방식으로 문자열을 사용하는 경우, `culture` 매개 변수에 대해 `CultureInfo.InvariantCulture`를 지정해야 합니다. 이는 대/소문자 변경 시간과 조회가 발생하는 시간 사이에 Thread.CurrentCulture의 값이 잠재적으로 변경될 수 있기 때문입니다.

보안 결정이 대소문자 변경 작업을 기반으로 하는 경우, 해당 작업은 문화에 의존하지 않아 결과가 `CultureInfo.CurrentCulture` 값에 의해 영향을 받지 않도록 해야 합니다. 문자열 사용 모범 사례 문서의 "현재 문화권을 사용하는 문자열 비교" 섹션을 참조하여, 문화에 민감한 문자열 작업이 어떻게 일관되지 않은 결과를 초래할 수 있는지 알아보세요.

String.ToUpper 및 String.ToLower

코드의 명확성을 위해서는, 항상 `String.ToUpper` 및 `String.ToLower` 메서드의 오버로드를 사용하여 문화권을 명시적으로 지정하는 것이 좋습니다. 예를 들어 다음 코드는 식별자 조회를 수행합니다. 이 `key.ToLower` 작업은 기본적으로 문화에 민감하게 반응하지만, 이로 인해 코드에서 이 동작이 명확하지 않습니다.

예시

C#

```
static object LookupKey(string key)
{
    return internalHashtable[key.ToLower()];
}
```

작업을 문화에 구애받지 않게 하려면 다음과 같이 앞의 예제를 변경하여 대/소문자를 변경할 때 명시적으로 `CultureInfo.InvariantCulture`을(를) 사용하십시오.

C#

```
static object LookupKey(string key)
{
    return internalHashtable[key.ToLower(CultureInfo.InvariantCulture)];
}
```

Char.ToUpper 및 Char.ToLower

`Char.ToUpper` 메서드와 `Char.ToLower` 메서드의 특성 `String.ToUpper` `String.ToLower` 이 동일하지만 영향을 받는 문화권은 터키어(튀르키예)와 아제르바이잔어(라틴어, 아제르바이잔)뿐입니다. 단일 문자 대/소문자 구분 차이가 있는 유일한 두 문화권입니다. 이 고유한 대소문자 매핑에 대한 자세한 내용은 String 클래스 문서의 "대소문자" 섹션을 참조하세요. 코드 명확성을 위해 일관된 결과를 보장하려면 항상 매개 변수를 허용하는 이러한 메서드의 오버로드를 CultureInfo 사용하는 것이 좋습니다.

참고하십시오

- String.ToUpper
- String.ToLower
- Char.ToUpper
- Char.ToLower
- CA1311: 문화권 지정 또는 고정 버전 사용
- .NET의 변경 사례
- 문화에 구애받지 않는 문자열 작업을 수행하기

Last updated on 2026. 03. 26.

컬렉션에서 문화에 민감하지 않은 문자열 작업 수행

2025. 06. 17.

네임스페이스 System.Collections에는 기본적으로 지역 문화에 민감한 동작을 제공하는 클래스와 멤버가 있습니다. CaseInsensitiveComparer 및 CaseInsensitiveHashCodeProvider 클래스에 대한 매개 변수가 없는 생성자는 속성 Thread.CurrentCulture을 사용하여 새 인스턴스를 초기화합니다. 모든 CollectionsUtil.CreateCaseInsensitiveHashtable 메서드의 오버로드는 기본적으로 Hashtable 속성을 사용하여 `Thread.CurrentCulture` 클래스의 새 인스턴스를 만듭니다. 메서드의 ArrayList.Sort 오버로드는 기본적으로 `Thread.CurrentCulture`를 사용하여 문화권 구분 정렬을 수행합니다. SortedList에서 문자열을 키로 사용할 때, 조회 및 정렬이 `Thread.CurrentCulture`에 의해 영향을 받을 수 있습니다. 이 섹션에 제공된 사용 권장 사항을 따라 네임스페이스 `Collections`에서 이러한 클래스 및 메서드의 문화에 민감하지 않은 결과를 얻으십시오.

> ⓘ **참고**
>
> 비교 메서드에 전달 CultureInfo.InvariantCulture 하면 문화권을 구분하지 않는 비교가 수행됩니다. 그러나 파일 경로, 레지스트리 키 및 환경 변수와 같이 비언어적 비교는 발생하지 않습니다. 비교 결과에 따라 보안 결정을 지원하지도 않습니다. 비언어적 비교나 결과 기반 보안 결정을 지원하려면, 응용 프로그램은 StringComparison 값을 허용하는 비교 메서드를 사용해야 합니다. 그 후에는 애플리케이션이 StringComparison를 전달해야 합니다.

`CaseInsensitiveComparer` 및 `CaseInsensitiveHashCodeProvider` 클래스 사용

매개변수가 없는 생성자는 `CaseInsensitiveHashCodeProvider`, `CaseInsensitiveComparer` 클래스를 새 인스턴스로 초기화하여 `Thread.CurrentCulture`를 사용해 문화에 민감한 동작을 생성합니다. 다음 코드 예제는 매개변수가 없는 `Hashtable` 및 `CaseInsensitiveHashCodeProvider` 생성자를 사용하여 문화권 구분이 가능한 `CaseInsensitiveComparer` 생성자를 보여줍니다.

C#

```
internalHashtable = new Hashtable(CaseInsensitiveHashCodeProvider.Default,
CaseInsensitiveComparer.Default);
```

문화에 구애받지 않는 `Hashtable`를 만들고자 한다면, `CaseInsensitiveComparer` 및 `CaseInsensitiveHashCodeProvider` 클래스를 사용하고, `culture` 매개 변수를 허용하는 생성자를 통해 이러한 클래스의 새 인스턴스를 초기화하세요. `culture` 매개 변수에 대해

CultureInfo.InvariantCulture를 지정합니다. 다음 코드 예제에서는 문화권을 구분 `Hashtable`하지 않는 생성자를 보여 줍니다.

```
C#

internalHashtable = new Hashtable(new CaseInsensitiveHashCodeProvider
    (CultureInfo.InvariantCulture),
    new CaseInsensitiveComparer(CultureInfo.InvariantCulture));
```

`CollectionsUtil.CreateCaseInsensitiveHashTable` 메서드 사용

이 `CollectionsUtil.CreateCaseInsensitiveHashTable` 메서드는 문자열의 대소문자를 무시하여 `Hashtable` 클래스의 새 인스턴스를 생성하는 데 유용한 바로가기입니다. 그러나 `CollectionsUtil.CreateCaseInsensitiveHashTable` 메서드의 모든 오버로드는 `Thread.CurrentCulture` 속성을 사용하기 때문에 문화에 민감합니다. 이 방법을 사용하여 문화에 민감하지 않은 `Hashtable`를 생성할 수 없습니다. 문화에 구애받지 않는 `Hashtable`를 만들려면 `Hashtable` 매개 변수를 허용하는 `culture` 생성자를 사용합니다. `culture` 매개 변수에 대해 `CultureInfo.InvariantCulture`를 지정합니다. 다음 코드 예제에서는 문화권을 구분 `Hashtable` 하지 않는 생성자를 보여 줍니다.

```
C#

internalHashtable = new Hashtable(new CaseInsensitiveHashCodeProvider
    (CultureInfo.InvariantCulture),
    new CaseInsensitiveComparer(CultureInfo.InvariantCulture));
```

`SortedList` 클래스 사용

A `SortedList` 는 키별로 정렬되고 키 및 인덱스별로 액세스할 수 있는 키 및 값 쌍의 컬렉션을 나타냅니다. 문자열이 키인 `SortedList` 위치를 사용하면 정렬 및 조회가 속성의 `Thread.CurrentCulture` 영향을 받을 수 있습니다. 문화에 민감하지 않은 동작을 `SortedList`에서 얻으려면, `SortedList` 매개변수를 허용하는 생성자 중 하나를 사용하여 `comparer`을(를) 만드세요. `comparer` 매개 변수는 키를 비교할 때 사용할 IComparer 구현을 지정합니다. 매개 변수의 경우 키를 비교하는 데 사용하는 `CultureInfo.InvariantCulture` 사용자 지정 비교자 클래스를 지정합니다. 다음 예제는 생성자에 `comparer` 매개 변수로 지정할 수 있는 사용자 지정 문화에 무관한 비교자 클래스를 설명합니다.

```
C#
```

```
using System;
using System.Collections;
using System.Globalization;

internal class InvariantComparer : IComparer
{
    private CompareInfo _compareInfo;
    internal static readonly InvariantComparer Default = new
        InvariantComparer();

    internal InvariantComparer()
    {
        _compareInfo = CultureInfo.InvariantCulture.CompareInfo;
    }

    public int Compare(Object a, Object b)
    {
        if (a is string sa && b is string sb)
            return _compareInfo.Compare(sa, sb);
        else
            return Comparer.Default.Compare(a,b);
    }
}
```

일반적으로, 사용자 지정 불변 비교자를 명시하지 않고 `SortedList`를 문자열에 사용할 경우, 목록이 채워진 후 `Thread.CurrentCulture`로 변경하면 목록이 무효화될 수 있습니다.

`ArrayList.Sort` 메서드 사용

`ArrayList.Sort` 메서드의 오버로드는 기본적으로 `Thread.CurrentCulture` 속성을 사용하여 문화권 구분 정렬을 수행합니다. 정렬 순서가 다르기 때문에 문화권에 따라 결과가 달라질 수 있습니다. 문화권에 민감한 동작을 제거하려면 `IComparer` 구현을 허용하는 이 메서드의 오버로드를 사용하십시오. 매개변수 `comparer`에 대한 사용자 지정 불변 비교자 클래스를 `CultureInfo.InvariantCulture` 사용하는 것으로 지정합니다. SortedList 클래스 사용 항목에서 사용자 지정 고정 비교자 클래스의 예가 제공됩니다.

참고하십시오

- CaseInsensitiveComparer
- CaseInsensitiveHashCodeProvider
- ArrayList.Sort
- SortedList
- Hashtable
- IComparer

- 문화에 구애받지 않는 문자열 작업을 수행하기
- CollectionsUtil.CreateCaseInsensitiveHashtable

배열에서 문화의 차이를 고려하지 않는 문자열 작업 수행

2025. 06. 17.

Array.Sort 및 Array.BinarySearch 메서드의 오버로드는 기본적으로 Thread.CurrentCulture 속성을 사용하여 문화권별 정렬을 수행합니다. 이러한 메서드에서 반환하는 문화권 구분 결과는 정렬 순서의 차이로 인해 문화권에 따라 달라질 수 있습니다. 문화권에 민감한 동작을 제거하려면 매개 변수를 허용하는 `comparer` 이 메서드의 오버로드 중 하나를 사용합니다. 매개 변수는 `comparer` 배열의 IComparer 요소를 비교할 때 사용할 구현을 지정합니다. 매개 변수에 대해 CultureInfo.InvariantCulture를 사용하는 사용자 지정 고정 비교자 클래스를 지정하세요. 사용자 지정 고정 비교자 클래스의 예는 컬렉션 항목에서 문화권을 구분하지 않는 문자열 작업 수행 의 "SortedList 클래스 사용" 하위 항목에 제공됩니다.

> ⓘ **참고**
>
> **CultureInfo.InvariantCulture**를 비교 메서드에 전달하면 문화권을 구분하지 않는 비교가 수행됩니다. 그러나 파일 경로, 레지스트리 키 및 환경 변수와 같이 비언어적 비교는 발생하지 않습니다. 비교 결과에 따라 보안 결정을 지원하지도 않습니다. 비언어적 비교나 결과 기반 보안 결정을 지원하려면, 응용 프로그램은 StringComparison 값을 허용하는 비교 메서드를 사용해야 합니다. 그 후에는 애플리케이션이 Ordinal를 전달해야 합니다.

참고하십시오

- Array.Sort
- Array.BinarySearch
- IComparer
- 문화에 구애받지 않는 문자열 작업을 수행하기

세계적 지원 애플리케이션 개발을 위한 모범 사례

2025. 06. 17.

이 섹션에서는 세계적 지원 애플리케이션을 개발할 때 따라야 할 모범 사례에 대해 설명합니다.

세계화 모범 사례

1. 애플리케이션을 내부적으로 유니코드로 만듭니다.

2. 네임스페이스에서 제공하는 문화권 인식 클래스를 System.Globalization 사용하여 데이터를 조작하고 서식을 지정합니다.

    - 정렬을 위해 SortKey 클래스와 CompareInfo 클래스를 사용하세요.
    - 문자열 비교의 경우 클래스를 CompareInfo 사용합니다.
    - 날짜 및 시간 서식의 경우 클래스를 DateTimeFormatInfo 사용합니다.
    - 숫자 서식 지정의 경우 클래스를 NumberFormatInfo 사용합니다.
    - 그레고리오력 및 그레고리오력이 아닌 달력의 경우 클래스 또는 특정 달력 구현 중 하나를 사용합니다 Calendar .

3. 적절한 상황에서 클래스에서 System.Globalization.CultureInfo 제공하는 문화권 속성 설정을 사용합니다. CultureInfo.CurrentCulture 날짜 및 시간 또는 숫자 서식 지정과 같은 작업의 서식 지정에 이 속성을 사용합니다. 속성을 CultureInfo.CurrentUICulture 사용하여 리소스를 검색합니다. `CurrentCulture` 및 `CurrentUICulture` 속성은 각 스레드마다 설정할 수 있습니다.

4. 네임스페이스의 인코딩 클래스를 사용하여 애플리케이션이 다양한 인코딩에서 데이터를 읽고 쓸 수 있도록 합니다 System.Text . ASCII 데이터를 가정하지 마세요. 사용자가 텍스트를 입력할 수 있는 모든 위치에 국제 문자가 제공된다고 가정합니다. 예를 들어 애플리케이션은 서버 이름, 디렉터리, 파일 이름, 사용자 이름 및 URL에서 국제 문자를 허용해야 합니다.

5. 보안상의 이유로 클래스를 UTF8Encoding 사용하는 경우 이 클래스에서 제공하는 오류 검색 기능을 사용합니다. 오류 검색 기능을 활성화하려면 `throwOnInvalidBytes` 매개변수를 사용하는 생성자를 통해 클래스의 인스턴스를 만들고, 해당 매개변수의 값을 `true`로 설정합니다.

6. 가능하면 문자열을 일련의 개별 문자가 아닌 전체 문자열로 처리합니다. 이는 부분 문자열을 정렬하거나 검색할 때 특히 중요합니다. 이렇게 하면 결합된 문자 구문 분석과 관련된 문제가 방지됩니다. 클래스를 사용하여 단일 문자가 아닌 텍스트 단위로 작업할 System.Globalization.StringInfo 수도 있습니다.

7. 네임스페이스에서 제공하는 System.Drawing 클래스를 사용하여 텍스트를 표시합니다.

8. 운영 체제 간의 일관성을 위해 사용자 설정이 CultureInfo를 재정의하지 않도록 하십시오. `CultureInfo` 매개변수를 수락하고 `useUserOverride`에 설정하는 `false` 생성자를 사용합니다.

9. 국제 데이터를 사용하여 국제 운영 체제 버전에서 애플리케이션 기능을 테스트합니다.

10. 보안 결정이 문자열 비교 또는 대/소문자 변경 작업의 결과를 기반으로 하는 경우 문화권을 구분하지 않는 문자열 작업을 사용합니다. 이렇게 하면 결과가 값의 `CultureInfo.CurrentCulture`영향을 받지 않습니다. 문자열 사용에 대한 모범 사례의 "현재 문화권을 사용하는 문화 민감한 문자열 비교" 섹션을 참조하십시오. 문화 민감한 문자열 비교가 일관성 없는 결과를 어떻게 생성할 수 있는지를 보여주는 예제가 있습니다.

11. 상호 교환(예: API 호출의 JSON 문서의 필드) 또는 저장에 사용되는 요소의 경우 CultureInfo를 사용합니다. 추가적으로, 왕복 변환 형식(예: "O""o" 날짜 및 시간 형식 지정자)을 명시적으로 지정해야 합니다. 고정 문화권의 형식 문자열은 안정적이고 변경될 가능성이 낮지만 명시적 형식 문자열을 지정하면 코드의 의도를 명확히 하는 데 도움이 됩니다.

    - 날짜/시간 요소의 경우 귀중한 통찰력을 공유하는 Noda Time 저자 Jon Skeet의 조언과 관찰을 고려합니다. 자세한 내용은 Jon Skeet: *UTC 저장은 만병통치약이 아닙니다*를 참고하시기 바랍니다.

12. 세계화 데이터는 안정적이지 않으므로 이를 염두에 두고 애플리케이션 및 해당 테스트를 작성해야 합니다. 지원되는 모든 플랫폼에서 호스트 OS 채널을 통해 1년마다 여러 번 업데이트됩니다. 이 데이터는 일반적으로 런타임과 함께 분산되지 않습니다.

지역화 모범 사례

1. 지역화 가능한 모든 리소스를 별도의 리소스 전용 DLL로 이동합니다. 지역화 가능한 리소스에는 문자열, 오류 메시지, 대화 상자, 메뉴 및 포함된 개체 리소스와 같은 사용자 인터페이스 요소가 포함됩니다.

2. 문자열 또는 사용자 인터페이스 리소스를 하드 코딩하지 마세요.

3. 지역화할 수 없는 리소스를 리소스 전용 DLL에 넣지 마세요. 이는 번역가들을 혼란스럽게 합니다.

4. 연결된 구에서 런타임에 빌드된 복합 문자열을 사용하지 마세요. 복합 문자열은 일부 언어에 적용되지 않는 영어 문법 순서를 가정하기 때문에 지역화하기가 어렵습니다.

5. 문자열 구성 요소의 문법 역할에 따라 문자열을 다르게 변환할 수 있는 "빈 폴더"와 같은 모호한 구문을 사용하지 마세요. 예를 들어 "empty"는 동사 또는 형용사일 수 있으며, 이는 이탈리아어 또는 프랑스어와 같은 언어로 다른 번역으로 이어질 수 있습니다.

6. 애플리케이션에 텍스트가 포함된 이미지와 아이콘을 사용하지 마세요. 지역화하는 데 비용이 많이 듭니다.

7. 사용자 인터페이스에서 문자열 길이를 확장할 충분한 공간을 허용합니다. 일부 언어에서 구에는 다른 언어에 필요한 것보다 50~75% 더 많은 공간이 필요할 수 있습니다.

8. 클래스를 System.Resources.ResourceManager 사용하여 문화권에 따라 리소스를 검색합니다.

9. Visual Studio를 사용하여 Windows Forms 리소스 편집기(Winres.exe)를 사용하여 지역화할 수 있도록 Windows Forms 대화 상자를 만듭니다. Windows Forms 대화 상자를 직접 코딩하지 마세요.

10. 전문 번역 서비스를 준비합니다.

11. 리소스를 만들고 지역화하는 방법에 대한 자세한 내용은 .NET 앱의 리소스를 참조하세요.

ASP.NET 및 기타 서버 애플리케이션에 대한 세계화 모범 사례

> **팁**
>
> 다음 모범 사례는 ASP.NET Framework 앱에 대한 것입니다. ASP.NET Core 앱은 **ASP.NET Core의 세계화 및 지역화를 참조하세요**.

1. 애플리케이션에서 CurrentUICulture 속성과 CurrentCulture 속성을 명시적으로 설정합니다. 기본값을 사용하지 마세요.

2. ASP.NET 애플리케이션은 관리되는 애플리케이션이므로 문화권에 따라 정보를 검색, 표시 및 조작하기 위해 다른 관리되는 애플리케이션과 동일한 클래스를 사용할 수 있습니다.

3. ASP.NET 다음 세 가지 유형의 인코딩을 지정할 수 있습니다.

   - `requestEncoding` 는 클라이언트의 브라우저에서 받은 인코딩을 지정합니다.
   - `responseEncoding` 는 클라이언트 브라우저로 보낼 인코딩을 지정합니다. 대부분의 경우 이 인코딩은 지정된 인코딩과 `requestEncoding`같아야 합니다.
   - fileEncoding은 *.aspx*, *.asmx* 및 *.asax* 파일 구문 분석의 기본 인코딩을 지정합니다.

4. ASP.NET 애플리케이션에서 다음 세 위치에 있는 `requestEncoding`, `responseEncoding`, `fileEncoding`, `culture`, 및 `uiCulture` 특성의 값을 지정합니다.

- *Web.config* 파일의 국제화 섹션에서 이 파일은 ASP.NET 애플리케이션 외부에 있습니다. 자세한 내용은 세계화< 요소를 참조>하세요.
- 페이지 지시문에서 애플리케이션이 페이지에 있는 경우 파일이 이미 읽혀졌습니다. 따라서 fileEncoding 및 requestEncoding을 지정하기에는 너무 늦었습니다. `uiCulture`, `culture`, 그리고 `responseEncoding`만 페이지 지시문에 지정할 수 있습니다.
- 애플리케이션 코드에서 프로그래밍 방식으로. 이 설정은 요청에 따라 달라질 수 있습니다. 페이지 지시문과 마찬가지로 애플리케이션의 코드에 도달하면 `fileEncoding` 및 `requestEncoding`를 지정하기에는 너무 늦습니다. 애플리케이션 코드에서는 `uiCulture`, `culture`, 및 `responseEncoding`만 지정할 수 있습니다.

5. uiCulture 값은 브라우저 수락 언어로 설정할 수 있습니다.

6. 분산된 애플리케이션의 경우, 무중단 업데이트(예: Azure Container Apps)를 허용하거나, 서로 다른 형식 규칙 또는 문화권 데이터, 특히 표준 시간대 규칙을 사용하는 애플리케이션 인스턴스가 여러 개 있을 수 있는 상황에 대한 계획이 필요합니다.

   - 애플리케이션 배포에 데이터베이스가 포함된 경우 데이터베이스에는 자체 세계화 규칙이 있어야 합니다. 대부분의 경우 데이터베이스에서 세계화 관련 함수를 수행하지 않아야 합니다.
   - 애플리케이션 배포에 클라이언트 세계화 리소스를 사용하는 클라이언트 애플리케이션 또는 웹 프런트 엔드가 포함된 경우 클라이언트 리소스가 서버에서 사용할 수 있는 리소스와 다르다고 가정합니다. 클라이언트에서만 세계화 함수를 수행하는 것이 좋습니다.

강력한 테스트를 위한 권장 사항

1. 종속성을 보다 명시적이고 테스트가 더 쉽고 병렬화할 수 있도록 하려면 매개 변수와 같은 문화권 관련 설정을 형식 지정을 수행하는 메서드 및 날짜 및 `CultureInfo` 시간을 사용하는 메서드에 명시적으로 전달하는 것이 `TimeZoneInfo`. *시간을 검색할 때도* 사용하거나 유사한 형식을 사용해야 TimeProvider 합니다.

2. 대부분의 테스트에서는 지정된 서식 작업의 정확한 출력 또는 표준 시간대의 정확한 오프셋의 유효성을 명시적으로 검사 *해서는 안 됩니다*. 서식 및 표준 시간대 데이터는 언제든지 변경될 수 있으며 운영 체제의 두 인스턴스(및 동일한 컴퓨터의 잠재적으로 다른 프로세스)가 다를 수 있습니다. 정확한 값에 의존하면 테스트가 취약해집니다.

   - 일반적으로 일부 출력이 수신되었는지 확인하는 것으로 충분합니다(예: 서식을 지정할 때 비어있지 않은 문자열).
   - 일부 데이터 요소 및 형식의 경우 데이터가 입력 값으로 구문 분석되는지 확인하는 대신(왕복) 사용할 수 있습니다. 필드가 삭제되는 경우(예: 일부 날짜 관련 필드의 경우 연도) 또는 잘리거나 반올림된 값(예: 부동 소수점 출력)에 주의해야 합니다.

- 모든 지역화된 형식 출력의 유효성을 검사하기 위한 명시적 요구 사항이 있는 경우 테스트 설정 중에 사용자 지정 문화권을 만들고 사용하는 *것이 좋습니다* . 대부분의 간단한 경우 생성자를 `CultureInfo` 사용하여 개체를 `new CultureInfo(..)` 인스턴스화하고 및 `DateTimeFormat` 속성을 설정 `NumberFormat` 하여 이 작업을 수행할 수 있습니다. 더 복잡한 경우 형식을 서브클래싱하면 추가 속성을 재정의할 수 있습니다. 리소스 파일을 사용하여 pseudolocalization 을 사용하도록 설정하는 것과 같은 추가적인 이점이 있을 수 있습니다.
- 모든 날짜/시간 작업의 결과의 유효성을 검사하기 위한 명시적 요구 사항이 있는 경우 테스트 설정 중에 사용자 지정 인스턴스를 만들고 사용하는 `TimeZoneInfo`. 특정 에지 사례(예: DST 규칙 변경)를 안정적으로 테스트할 수 있도록 설정하는 등 이에 대한 추가적인 이점이 있을 수 있습니다.

참고하십시오

- 전역화 및 지역화
- .NET 앱의 리소스

지역화 가능성 검토

2025. 06. 17.

지역화 가능성 검토는 세계적 지원 애플리케이션 개발의 중간 단계입니다. 전역화된 애플리케이션이 지역화할 준비가 되었으며 특별한 처리가 필요한 코드 또는 사용자 인터페이스의 모든 측면을 식별합니다. 또한 이 단계는 지역화 프로세스가 애플리케이션에 기능적 결함을 도입하지 않도록 하는 데 도움이 됩니다. 지역화 가능성 검토로 인해 발생한 모든 문제가 해결되면 애플리케이션이 지역화할 준비가 된 것입니다. 지역화 가능성 검토가 철저한 경우 지역화 프로세스 중에 소스 코드를 수정할 필요가 없습니다.

지역화 가능성 검토는 다음 세 가지 검사로 구성됩니다.

- ✔ 세계화 권장 사항이 구현되었나요?
- ✔ 문화권에 민감한 기능이 올바르게 처리되었나요?
- ✔ 국제 데이터로 애플리케이션을 테스트했나요?

세계화 권장 사항 구현

지역화를 염두에 두고 애플리케이션을 설계하고 개발했으며 세계화 문서에서 설명한 권장 사항을 따른 경우 지역화 가능성 검토는 주로 품질 보증 통과가 됩니다. 그렇지 않으면 이 단계에서 세계화 에 대한 권장 사항을 검토하고 구현하고 지역화를 방지하는 소스 코드의 오류를 수정해야 합니다.

문화적으로 민감한 기능을 처리하다

.NET은 문화권에 따라 크게 다른 여러 영역에서 프로그래밍 방식으로 지원을 제공하지 않습니다. 대부분의 경우 다음과 같은 기능 영역을 처리하는 사용자 지정 코드를 작성해야 합니다.

- 주소
- 전화 번호
- 용지 크기
- 길이, 가중치, 영역, 볼륨 및 온도에 사용되는 측정 단위

  .NET은 측정 단위 간 변환에 대한 기본 제공 지원을 제공하지 않지만 다음 예제와 같이 특정 국가 또는 지역에서 메트릭 시스템을 사용하는지 여부를 결정하는 데 이 속성을 사용할 RegionInfo.IsMetric 수 있습니다.

  ```
  C#
  ```

```
string[] cultureNames = { "en-US", "en-GB", "fr-FR",
                          "ne-NP", "es-BO", "ig-NG" };
foreach (string cultureName in cultureNames)
{
    RegionInfo region = new(cultureName);
    string usesMetric = region.IsMetric ? "uses" : "does not use";
    Console.WriteLine($"{region.EnglishName} {usesMetric} the metric
system.");
}

// The example displays the following output:
//       United States does not use the metric system.
//       United Kingdom uses the metric system.
//       France uses the metric system.
//       Nepal uses the metric system.
//       Bolivia uses the metric system.
//       Nigeria uses the metric system.
```

애플리케이션 테스트

애플리케이션을 지역화하기 전에 운영 체제의 국제 버전에서 국제 데이터를 사용하여 테스트해야 합니다. 이 시점에서 대부분의 사용자 인터페이스는 지역화되지 않지만 다음과 같은 문제를 감지할 수 있습니다.

- 운영 체제 버전 간에 올바르게 역직렬화되지 않는 직렬화된 데이터입니다.
- 현재 문화권의 규칙을 반영하지 않는 숫자 데이터입니다. 예를 들어 부정확한 그룹 구분 기호, 소수 구분 기호 또는 통화 기호를 사용하여 숫자를 표시할 수 있습니다.
- 현재 문화권의 규칙을 반영하지 않는 날짜 및 시간 데이터입니다. 예를 들어 월과 요일을 나타내는 숫자가 잘못된 순서로 표시되거나 날짜 구분 기호가 잘못되거나 표준 시간대 정보가 올바르지 않을 수 있습니다.
- 애플리케이션의 기본 문화권을 식별하지 않았기 때문에 찾을 수 없는 리소스입니다.
- 특정 문화권에 대해 비정상적인 순서로 표시되는 문자열입니다.
- 예기치 않은 결과를 초래하는 문자열 비교 또는 동등성 비교.

애플리케이션을 개발할 때 세계화 권장 사항을 따르고, 문화권에 민감한 기능을 올바르게 처리하고, 테스트 중에 발생한 지역화 문제를 식별하고 해결한 경우 다음 단계인 지역화를 진행할 수 있습니다.

참고하십시오

- 전역화 및 지역화
- 지역화
- 세계화
- .NET 앱의 리소스

.NET의 지역화

지역화는 애플리케이션의 리소스를 애플리케이션이 지원할 각 문화권에 대해 지역화된 버전으로 변환하는 프로세스입니다. 지역화 가능성 검토 단계를 완료한 후에만 지역화 단계를 진행하여 세계화된 애플리케이션이 지역화할 준비가 되었는지 확인해야 합니다.

지역화할 준비가 된 애플리케이션은 모든 사용자 인터페이스 요소를 포함하는 블록과 실행 코드가 포함된 블록이라는 두 가지 개념 블록으로 구분됩니다. 사용자 인터페이스 블록에는 중립 문화권에 대한 문자열, 오류 메시지, 대화 상자, 메뉴, 포함된 개체 리소스 등과 같은 지역화 가능한 사용자 인터페이스 요소만 포함됩니다. 코드 블록에는 지원되는 모든 문화권에서 사용할 애플리케이션 코드만 포함됩니다. 공용 언어 런타임은 애플리케이션의 실행 코드를 해당 리소스와 분리하는 위성 어셈블리 리소스 모델을 지원합니다. 이 모델을 구현하는 방법에 대한 자세한 내용은 .NET의 리소스를 참조하세요.

애플리케이션의 지역화된 각 버전에 대해 대상 문화권에 적합한 언어로 번역된 지역화된 사용자 인터페이스 블록을 포함하는 새 위성 어셈블리를 추가합니다. 모든 문화권의 코드 블록은 동일하게 유지되어야 합니다. 사용자 인터페이스 블록의 지역화된 버전을 코드 블록과 조합하면 지역화된 버전의 애플리케이션이 생성됩니다.

이 기사에서는 IStringLocalizer<T> 및 IStringLocalizerFactory 구현을 사용하는 방법을 배웁니다. 이 문서의 모든 예제 소스 코드는 Microsoft.Extensions.Localization NuGet 패키지와 Microsoft.Extensions.Hosting NuGet 패키지를 사용합니다. 호스팅에 대한 자세한 내용은 .NET 제네릭 호스트를 참조하세요.

리소스 파일

지역화 가능한 문자열을 격리하기 위한 기본 메커니즘은 **리소스 파일을** 사용하는 것입니다. 리소스 파일은 *.resx* 파일 확장자를 가진 XML 파일입니다. 리소스 파일은 소비하는 애플리케이션을 실행하기 전에 번역됩니다. 즉, 미사용 시 번역된 콘텐츠를 나타냅니다. 리소스 파일 이름은 가장 일반적으로 로캘 식별자를 포함하며 다음 형식을 사용합니다.

`<FullTypeName><.Locale>.resx`

위치:

- 특정 `<FullTypeName>` 형식에 대한 지역화 가능한 리소스를 나타냅니다.
- 선택적 `<.Locale>` 리소스 파일 내용의 로캘을 나타냅니다.

로캘 지정

로캘은 최소한 언어를 정의해야 하지만 문화권(지역 언어) 및 국가 또는 지역을 정의할 수도 있습니다. 이러한 세그먼트는 일반적으로 문자로 `-` 구분됩니다. 문화권의 구체성이 추가되면 최적의 일치를 우선적으로 고려하여 "문화 대체" 규칙이 적용됩니다. 로캘은 잘 알려진 언어 태그에 매핑되어야 합니다. 자세한 내용은 CultureInfo.Name를 참조하세요.

문화권 대체 시나리오

지역화된 앱이 다양한 세르비아어 로캘을 지원하고 다음과 같은 리소스 파일이 있다고 `MessageService` 상상해 보세요.

테이블 확장

파일	지역 언어	국가 코드
MessageService.sr-Cyrl-RS.resx	(키릴 자모, 세르비아)	알에스
MessageService.sr-Cyrl.resx	키릴 문자체	
MessageService.sr-Latn-BA.resx	(라틴 문자, 보스니아 및 헤르체고비나)	학사
MessageService.sr-Latn-ME.resx	(라틴어, 몬테네그로)	저
MessageService.sr-Latn-RS.resx	(라틴어, 세르비아)	알에스
MessageService.sr-Latn.resx	라틴어	
MessageService.sr.resx	† 라틴어	
MessageService.resx		

† *언어의 기본 지역 언어입니다.*

앱이 CultureInfo.CurrentCulture으로 실행되며 문화 `"sr-Cyrl-RS"`으로 지역화를 시도할 때, 파일을 다음 순서로 확인하려고 시도합니다.

1. *MessageService.sr-Cyrl-RS.resx*
2. *MessageService.sr-Cyrl.resx*
3. *MessageService.sr.resx*
4. *MessageService.resx*

앱이 CultureInfo.CurrentCulture로 실행 중이지만 `"sr-Latn-BA"` 문화 설정이 있는 경우, 다음 순서로 파일을 확인하려고 시도합니다.

1. *MessageService.sr-Latn-BA.resx*
2. *MessageService.sr-Latn.resx*
3. *MessageService.sr.resx*
4. *MessageService.resx*

"문화권 대체" 규칙은 일치하는 항목이 없을 때 로캘을 무시합니다. 즉, 일치 항목을 찾을 수 없는 경우 리소스 파일 번호 4가 선택됩니다. 문화권이 설정된 `"fr-FR"`경우 지역화는 문제가 될 수 있는 *MessageService.resx* 파일로 떨어지게 됩니다. 자세한 내용은 리소스 대체 프로세스를 참조하세요.

리소스 조회

리소스 파일은 조회 루틴의 일부로 자동으로 확인됩니다. 프로젝트 파일 이름이 프로젝트의 루트 네임스페이스와 다른 경우 어셈블리 이름이 다를 수 있습니다. 이렇게 하면 리소스 조회가 성공하지 못할 수 있습니다. 이 불일치를 해결하려면 지역화 서비스에 힌트를 제공하는 데 사용합니다 RootNamespaceAttribute . 제공된 경우 리소스 조회 중에 사용됩니다.

예제 프로젝트는 *example.csproj*로 이름이 지정되어 있으며 *example.dll*와 *example.exe*를 생성하지만, `Localization.Example` 네임스페이스가 사용됩니다. `assembly` 수준 특성을 적용하여 이 불일치를 수정합니다.

C#

```
[assembly: RootNamespace("Localization.Example")]
```

지역화 서비스 등록

지역화 서비스를 등록하려면 서비스를 구성하는 동안 확장 메서드 중 AddLocalization 하나를 호출합니다. 이렇게 하면 다음 형식의 DI(종속성 주입)가 활성화됩니다.

- Microsoft.Extensions.Localization.IStringLocalizer<T>
- Microsoft.Extensions.Localization.IStringLocalizerFactory

지역화 옵션 구성

오버로드는 AddLocalization(IServiceCollection, Action<LocalizationOptions>) 형식의 `setupAction` 매개 변수를 `Action<LocalizationOptions>` 허용합니다. 이를 통해 지역화 옵션을 구성할 수 있습니다.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddLocalization(options =>
{
    options.ResourcesPath = "Resources";
});

// Omitted for brevity.
```

자원 파일은 프로젝트의 어느 곳에나 살 수 있지만 성공한 것으로 입증된 일반적인 사례가 있습니다. 더 자주는 아니지만, 최소 저항의 경로가 따릅니다. 앞의 C# 코드는 다음과 같습니다.

- 기본 호스트 앱 작성기를 만듭니다.
- 서비스 컬렉션 호출 `AddLocalization`에서 LocalizationOptions.ResourcesPath을 `"Resources"`로 지정합니다.

이로 인해 지역화 서비스가 *Resources* 디렉터리에서 리소스 파일을 찾게 됩니다.

ResourcesPath 및 공유 리소스 클래스

`ResourcesPath`가 구성되면, `IStringLocalizerFactory.Create(Type)`는 형식의 상대 이름을 계산하여 (루트 네임스페이스를 기준으로), `ResourcesPath`를 앞에 추가함으로써 리소스 파일 경로를 확인합니다. 즉, 프로젝트 루트 네임스페이스를 기준으로 공유 리소스 클래스의 배치가 중요합니다.

예를 들어 여러 구성 요소에서 문자열을 통합하는 데 사용되는 공유 리소스 클래스를 고려합니다. 설정되어 있고 공유 리소스 클래스 `ResourcesPath = "Resources"`가 네임스페이스의 `SharedResource` 폴더 *내에 위치하는* 경우, (정규화된 형식 이름`MyApp.Resources.SharedResource`) 팩터리는 루트 네임스페이스 `MyApp`에 상대적으로 기본 이름을 `Resources.SharedResource`으로 확인하고 앞에 `ResourcesPath`를 추가하여 실제 파일 위치와 일치하지 않는 경로 `Resources/Resources/SharedResource.resx`를 생성합니다.

이를 방지하기 위한 두 가지 옵션이 있습니다.

- **공유 리소스 클래스를 루트 네임스페이스에 유지합니다.** 폴더 내부가 아닌 프로젝트 루트 (네임스페이스`SharedResource.cs`)에 배치 `MyApp` 합니다`Resources`. 그런 다음 팩토리는 기본 이름을 `SharedResource`로 해석하고, `ResourcesPath = "Resources"`을 사용하여 `Resources/SharedResource.resx`를 올바르게 찾습니다.

- **`factory.Create(string baseName, string location)`을 사용합니다.** 기본 이름 및 어셈블리 이름을 명시적으로 제공하면 리소스 파일 확인을 직접 제어할 수 있습니다.

    ```
    ```

C#

```
var assemblyName = typeof(SharedResource).Assembly.GetName().Name;
IStringLocalizer localizer = factory.Create("SharedResource", assemblyName);
```

이 방법은 프로젝트의 위치에 `SharedResource.cs` 관계없이 작동합니다.

> ⓘ **참고 항목**
>
> 내부적으로 사용하는 `IStringLocalizerFactory` 프레임워크와 통합할 때도 동일한 고려 사항이 적용됩니다. 예를 들어 ASP.NET Core는 `AddDataAnnotationsLocalization` 이 팩터리를 공유 리소스 클래스와 함께 사용합니다. `ResourcesPath` 이(가) 구성된 경우, `factory.Create(string, string)` 오버로드를 사용하십시오. 또는 공유 리소스 클래스가 루트 네임스페이스에 있는지 확인합니다.

`IStringLocalizer<T>` 및 `IStringLocalizerFactory` 사용

지역화 서비스를 등록 (및 필요에 따라 구성)한 후 DI에서 다음 형식을 사용할 수 있습니다.

- IStringLocalizer<T>
- IStringLocalizerFactory

지역화된 문자열을 반환할 수 있는 메시지 서비스를 만들려면 다음 `MessageService`을 고려합니다.

C#

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.Extensions.Localization;

namespace Localization.Example;

public sealed class MessageService(IStringLocalizer<MessageService> localizer)
{
    [return: NotNullIfNotNull(nameof(localizer))]
    public string? GetGreetingMessage()
    {
        LocalizedString localizedString = localizer["GreetingMessage"];

        return localizedString;
    }
}
```

위의 C# 코드에서:

- `IStringLocalizer<MessageService> localizer` 필드가 선언됩니다.
- 기본 생성자는 매개 변수를 `IStringLocalizer<MessageService>` 정의하고 인수로 `localizer` 캡처합니다.
- 메서드는 `GetGreetingMessage`을 호출하여 IStringLocalizer.Item[String]을 인수로 전달합니다.

`IStringLocalizer` 매개 변수가 있는 문자열 리소스도 지원합니다. 다음 `ParameterizedMessageService`을 고려합니다.

C#

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.Extensions.Localization;

namespace Localization.Example;

public class ParameterizedMessageService(IStringLocalizerFactory factory)
{
    private readonly IStringLocalizer _localizer =
        factory.Create(typeof(ParameterizedMessageService));

    [return: NotNullIfNotNull(nameof(_localizer))]
    public string? GetFormattedMessage(DateTime dateTime, double dinnerPrice)
    {
        LocalizedString localizedString = _localizer["DinnerPriceFormat", dateTime,
dinnerPrice];

        return localizedString;
    }
}
```

위의 C# 코드에서:

- `IStringLocalizer _localizer` 필드가 선언됩니다.
- 기본 생성자는 `IStringLocalizerFactory` 매개 변수를 받아 `IStringLocalizer` 유형에서 `ParameterizedMessageService`을 생성하고 `_localizer` 필드에 할당합니다.
- `GetFormattedMessage` 메서드는 IStringLocalizer.Item[String, Object[]]를 호출하여 `"DinnerPriceFormat"`, `dateTime` 객체, 그리고 `dinnerPrice`를 인수로 전달합니다.

> ⓘ **Important**
>
> 필수 `IStringLocalizerFactory` 는 아닙니다. 대신, 서비스를 사용할 때 IStringLocalizer<T>을 요구하는 것이 좋습니다.

두 인덱서 모두 암시적 변환이 가능한 IStringLocalizer.Item[]을 반환하며, 이는 LocalizedString에서 로 변환될 수 있습니다.

전부 합치세요

지역화 및 리소스 파일과 함께 메시지 서비스를 모두 사용하여 앱을 예로 들려면 다음 *Program.cs* 파일을 고려합니다.

C#

```
using System.Globalization;
using Localization.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Localization;
using Microsoft.Extensions.Logging;
using static System.Console;
using static System.Text.Encoding;

[assembly: RootNamespace("Localization.Example")]

OutputEncoding = Unicode;

if (args is [var cultureName])
{
    CultureInfo.CurrentCulture =
        CultureInfo.CurrentUICulture =
            CultureInfo.GetCultureInfo(cultureName);
}

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddLocalization();
builder.Services.AddTransient<MessageService>();
builder.Services.AddTransient<ParameterizedMessageService>();
builder.Logging.SetMinimumLevel(LogLevel.Warning);

using IHost host = builder.Build();

IServiceProvider services = host.Services;

ILogger logger =
    services.GetRequiredService<ILoggerFactory>()
        .CreateLogger("Localization.Example");

MessageService messageService =
    services.GetRequiredService<MessageService>();
logger.LogWarning(
    "{Msg}",
    messageService.GetGreetingMessage());
```

```
ParameterizedMessageService parameterizedMessageService =
    services.GetRequiredService<ParameterizedMessageService>();
logger.LogWarning(
    "{Msg}",
    parameterizedMessageService.GetFormattedMessage(
        DateTime.Today.AddDays(-3), 37.63));

await host.RunAsync();
```

위의 C# 코드에서:

- RootNamespaceAttribute은 `"Localization.Example"`을 루트 네임스페이스로 설정합니다.
- Console.OutputEncoding는 Encoding.Unicode에 할당됩니다.
- 단일 인수가 `args`에 전달되면, CultureInfo.CurrentCulture이(가) 주어진 상황에서 CultureInfo.CurrentUICulture과 CultureInfo.GetCultureInfo(String)에 `arg[0]`의 결과가 할당됩니다.
- Host은 기본 값으로 만듭니다.
- DI를 위해 `MessageService`에 등록되는 지역화 서비스, `ParameterizedMessageService` 및 `IServiceCollection`입니다.
- 노이즈를 제거하기 위해 로깅은 경고보다 낮은 로그 수준을 무시하도록 구성됩니다.
- `MessageService`는 `IServiceProvider` 인스턴스로부터 해결되고 생성된 메시지가 기록됩니다.
- `ParameterizedMessageService`는 `IServiceProvider` 인스턴스로부터 해결되어 그 결과로 형식이 지정된 메시지가 기록됩니다.

각 `*MessageService` 클래스는 각각 단일 항목을 가진 *.resx* 파일의 집합을 정의합니다. 다음은 `MessageService`부터 시작하는 리소스 파일에 대한 예제 콘텐츠입니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Hi friends, the ".NET" developer community is excited to see you here!
</value>
  </data>
</root>
```

MessageService.sr-Cyrl-RS.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
```

```
    <value>Здраво пријатељи, ".NET" девелопер заједница је узбуђена што вас види
овде!</value>
  </data>
</root>
```

MessageService.sr-Latn.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Zdravo prijatelji, ".NET" developer zajednica je uzbuđena što vas vidi
ovde!</value>
  </data>
</root>
```

다음은 `ParameterizedMessageService`부터 시작하는 리소스 파일에 대한 예제 콘텐츠입니다.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>On {0:D} my dinner cost {1:C}.</value>
  </data>
</root>
```

ParameterizedMessageService.sr-Cyrl-RS.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>У {0:D} моја вечера је коштала {1:C}.</value>
  </data>
</root>
```

ParameterizedMessageService.sr-Latn.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>U {0:D} moja večera je koštala {1:C}.</value>
```

```
  </data>
</root>
```

> **팁**
>
> 모든 리소스 파일 XML 주석, 스키마 및 `<resheader>` 요소는 간결하게 하기 위해 의도적으로 생략됩니다.

예제 실행

다음 예제 실행에서는 지정된 대상 로캘에 따라 다양한 지역화된 출력을 보여 줍니다.

다음을 고려합니다. `"sr-Latn"`

.NET CLI

```
dotnet run --project .\example\example.csproj sr-Latn

warn: Localization.Example[0]
      Zdravo prijatelji, ".NET" developer zajednica je uzbuđena što vas vidi ovde!
warn: Localization.Example[0]
      U utorak, 03. avgust 2021. moja večera je koštala 37,63 ¤.
```

.NET CLI를 사용하여 프로젝트를 실행할 때, 인수가 생략되면 기본 시스템 문화권이 사용됩니다. 이 경우 `"en-US"`:

.NET CLI

```
dotnet run --project .\example\example.csproj

warn: Localization.Example[0]
      Hi friends, the ".NET" developer community is excited to see you here!
warn: Localization.Example[0]
      On Tuesday, August 3, 2021 my dinner cost $37.63.
```

전달 `"sr-Cryl-RS"`하면 올바른 해당 리소스 파일이 발견되고 지역화가 적용됩니다.

.NET CLI

```
dotnet run --project .\example\example.csproj sr-Cryl-RS

warn: Localization.Example[0]
      Здраво пријатељи, ".NET" девелопер заједница је узбуђена што вас види овде!
warn: Localization.Example[0]
      У уторак, 03. август 2021. моја вечера је коштала 38 RSD.
```

샘플 애플리케이션은 리소스 파일을 `"fr-CA"`제공하지 않지만 해당 문화권으로 호출될 때 지역화되지 않은 리소스 파일이 사용됩니다.

> **⚠ Warning**
>
> 문화권을 찾았지만 올바른 리소스 파일은 없으므로 서식이 적용되면 부분 지역화로 끝납니다.

.NET CLI

```
dotnet run --project .\example\example.csproj fr-CA

warn: Localization.Example[0]
      Hi friends, the ".NET" developer community is excited to see you here!
warn: Localization.Example[0]
      On mardi 3 août 2021 my dinner cost 37,63 $.
```

참고하십시오

- .NET 애플리케이션 전역화 및 지역화
- .NET 앱에서 리소스 패키지 및 배포
- Microsoft.Extensions.Localization
- .NET에서 종속성 주입
- .NET의 로깅
- ASP.NET Core 지역화

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 04. 03.

CompareInfo 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

데이터 비교 및 정렬에 대한 규칙은 문화권마다 다릅니다. 예를 들어, 정렬 순서는 발음 또는 문자의 시각적 표현을 기반으로 할 수 있습니다. 동아시아 언어에서는 한자가 부수와 획에 따라 정렬됩니다. 언어와 문화권이 알파벳에 사용하는 순서에 따라 정렬 순서가 달라지기도 합니다. 예를 들어 덴마크어 알파벳의 "Æ" 문자는 "Z" 다음에 옵니다. 또한 비교는 대소문자 구분 여부에 따라 다를 수 있으며, 대소문자 규칙도 문화에 따라 달라질 수 있습니다. 클래스는 CompareInfo 이 문화권 구분 문자열 비교 데이터를 유지 관리하고 문화권 구분 문자열 작업을 수행합니다.

일반적으로 CompareInfo 개체는 직접 인스턴스화할 필요가 없습니다. 메서드 String.Compare 호출을 포함하여 모든 비서수 문자열 비교 작업에서 암시적으로 사용되기 때문입니다. 그러나 개체 CompareInfo를 검색하려는 경우 다음 방법 중 하나로 할 수 있습니다.

- 특정 문화권의 CultureInfo.CompareInfo 속성 값을 검색합니다.
- 문화권 이름을 사용하여 정적 GetCompareInfo 메서드를 호출합니다. 이렇게 하면 런타임에 CompareInfo 개체에 바인딩된 액세스가 허용됩니다.

무시된 검색 값

문자 집합에는 언어 또는 문화권 구분 비교를 수행할 때 고려되지 않는 문자인 무시 가능한 문자가 포함됩니다. 문화권 구분 비교를 수행할 때와 같은 IndexOf 비교 메서드는 LastIndexOf 이러한 문자를 고려하지 않습니다. 무시할 수 있는 문자는 다음과 같습니다.

- String.Empty; 문화권 구분 비교 메서드는 항상 검색되는 문자열의 시작 부분(인덱스 0)에 빈 문자열을 찾습니다.
- 비교 옵션으로 인해 작업에서 고려되지 않는 코드 포인트가 있는 문자로 구성된 문자 또는 문자열입니다. 특히 CompareOptions.IgnoreNonSpace 및 CompareOptions.IgnoreSymbols 옵션은 기호와 간격이 없는 결합 문자가 무시되는 검색을 생성합니다.
- 언어적 의미가 없는 코드 포인트가 있는 문자열입니다. 예를 들어 소프트 하이픈(U+00AD)은 문화권 구분 문자열 비교에서 항상 무시됩니다.

보안 고려 사항

보안 결정이 문자열 비교 또는 대/소문자 변경에 따라 달라지는 경우 운영 체제의 문화권 설정에 관계없이 동작이 일관되도록 속성을 사용해야 InvariantCulture 합니다.

ⓘ 참고

가능하면 형식 CompareOptions 의 매개 변수가 있는 문자열 비교 메서드를 사용하여 예상되는 비교 종류를 지정해야 합니다. 일반적으로 사용자 인터페이스에 표시된 문자열을 비교하고 보안 비교를 위해 OrdinalOrdinalIgnoreCase 언어 옵션(현재 문화권 사용)을 사용합니다.

CompareOptions enum

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 CompareOptions 옵션은 대/소문자 형식을 무시해야 하는 대/소문자 구분 또는 필요성을 나타냅니다.

.NET에서는 단어 정렬, 문자열 정렬 및 서수 정렬의 세 가지 고유한 정렬 방법을 사용합니다. 단어 정렬은 문자열의 문화권 구분 비교를 수행합니다. 특정 무수 문자에는 특수 가중치가 할당되어 있을 수 있습니다. 예를 들어 하이픈("-")에는 매우 작은 가중치가 할당되어 정렬된 목록에서 "coop" 및 "co-op"가 나란히 표시될 수 있습니다. 문자열 정렬은 특수한 경우가 없다는 점을 제외하고 단어 정렬과 유사합니다. 따라서 모든 무수 기호는 모든 영숫자 문자 앞에 옵니다. 서수 정렬은 문자열의 각 요소에 대한 유니코드 값을 기준으로 문자열을 비교합니다. Windows 운영 체제의 정렬 및 비교 작업에 사용되는 문자 가중치 정보를 포함한 텍스트 파일을 다운로드하려면 정렬 가중치 테이블을 참조하세요. Linux 및 macOS에 대한 정렬 가중치 테이블은 기본 유니코드 데이터 정렬 요소 테이블을 참조하세요. Linux 및 macOS에서 정렬 가중치 테이블의 특정 버전은 시스템에 설치된 유니코드용 International Components 라이브러리의 버전에 따라 달라집니다. ICU 버전 및 구현하는 유니코드 버전에 대한 자세한 내용은 ICU 다운로드를 참조하세요.

값은 `StringSort` 및 CompareInfo.Compare와 함께 CompareInfo.GetSortKey만 사용할 수 있습니다. ArgumentException는 StringSort 값이 CompareInfo.IsPrefix, CompareInfo.IsSuffix, CompareInfo.IndexOf 또는 CompareInfo.LastIndexOf와 함께 사용될 때 throw됩니다.

> **참고**
>
> 가능하면 CompareOptions 값을 허용하는 문자열 비교 메서드를 사용하여 기대하는 비교 종류를 지정해야 합니다. 일반적으로 사용자 지향 비교는 현재 문화에 맞춘 언어 옵션을 사용하는 것이 가장 좋습니다. 반면, 보안 비교는 `Ordinal` 또는 `OrdinalIgnoreCase`를 명시해야 합니다.

문화권 구분 정렬

> **참고**
>
> **Linux 및 macOS 시스템에서만 실행되는 .NET Core:** C 및 Posix 문화권에 대한 데이터 정렬 동작은 이러한 문화권에서 예상되는 유니코드 데이터 정렬 순서를 사용하지 않으므로 항상

대/소문자를 구분합니다. C 또는 Posix가 아닌 다른 문화를 사용하여 문화에 민감하면서 대소문자를 구분하지 않는 정렬 작업을 수행할 것을 권장합니다.

CultureAndRegionInfoBuilder 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> **참고**
>
> 이 CultureAndRegionInfoBuilder 클래스는 Windows 운영 체제에만 유용합니다. 생성된 *.nlp* 파일은 비 Windows 운영 체제에서 지원되지 않습니다. 또한 Windows에서도 생성된 *.nlp* 파일은 .NET Framework(또는 NLS 세계화 모드를 사용하는 경우 .NET Core)에서만 지원됩니다.

클래스는 CultureInfo 관련 언어, 하위 언어, 국가/지역, 달력 및 문화권 규칙과 같은 문화권별 정보를 보유합니다. 이 클래스는 또한 대/소문자 변경, 날짜 및 숫자 형식화 및 구문 분석, 문자열 비교와 같은 문화권별 작업에 필요한 DateTimeFormatInfo, NumberFormatInfo, CompareInfo, 및 TextInfo 클래스의 문화권별 인스턴스를 제공합니다.

기본적으로 .NET은 CultureInfo 미리 정의된 문화권 집합을 나타내는 개체를 지원합니다. Windows 시스템에서 사용할 수 있는 이러한 문화권 목록은 Windows에서 **지원하는 언어/지역 이름 목록에서 언어**태그 열을 참조하세요. 문화권 이름은 BCP 47 에 정의된 표준을 따릅니다. 클래스 CultureAndRegionInfoBuilder 를 사용하면 완전히 새로운 사용자 지정 문화권을 만들거나 미리 정의된 문화권을 재정의할 수 있습니다. 사용자 지정 문화권이 설치되어 특정 컴퓨터에 등록되면 미리 정의된 CultureInfo 개체와 구별할 수 없게 되며 이러한 개체와 마찬가지로 인스턴스화하고 사용할 수 있습니다.

> **중요**
>
> CultureAndRegionInfoBuilder 클래스는 *sysglobl.dll*이라는 어셈블리에서 발견됩니다. 이 형식을 사용하는 코드를 성공적으로 컴파일하려면 *sysglobl.dll*대한 참조를 추가해야 합니다.

사용자 지정 문화권은 해당 컴퓨터에 대한 관리 권한이 있는 사용자만 컴퓨터에 등록할 수 있습니다. 따라서 앱은 일반적으로 사용자 지정 문화권을 만들고 설치하지 않습니다. 대신 클래스를 CultureAndRegionInfoBuilder 사용하여 관리자가 사용자 지정 문화권을 만들고, 설치하고, 등록하는 데 사용할 수 있는 특수한 용도의 도구를 만들 수 있습니다. 사용자 지정 문화권이 컴퓨터에 등록되면 미리 정의된 문화권과 마찬가지로 앱의 클래스를 사용하여 CultureInfo 사용자 지정 문화권의 인스턴스를 만들 수 있습니다.

사용자 지정 문화권에 대해 생성된 날짜 및 시간 문자열을 구문 분석하는 경우, 구문 분석 작업이 성공할 확률을 높이기 위해 DateTime.ParseExact 또는 DateTime.TryParseExact 메서드를 사용

해야 하며, DateTime.Parse 또는 DateTime.TryParse 메서드 대신 사용해야 합니다. 사용자 지정 문화권의 날짜 및 시간 문자열은 복잡하므로 구문 분석하기가 어려울 수 있습니다. Parse 및 TryParse 메서드는 여러 암시적 구문 분석 패턴을 사용하여 문자열을 구문 분석하려고 시도하지만, 이들 모두는 실패할 수 있습니다. 반면 이 TryParseExact 메서드를 사용하려면 애플리케이션에서 성공할 가능성이 있는 하나 이상의 정확한 구문 분석 패턴을 명시적으로 지정해야 합니다.

사용자 지정 문화권 정의 및 만들기

클래스를 CultureAndRegionInfoBuilder 사용하여 사용자 지정 문화권을 정의하고 이름을 지정합니다. 사용자 지정 문화권은 완전히 새로운 문화권, 기존 문화권(즉, 추가 문화권)을 기반으로 하는 새로운 문화권 또는 기존 .NET 문화권을 대체하는 문화권일 수 있습니다. 각각의 경우 기본 단계는 동일합니다.

1. CultureAndRegionInfoBuilder 개체를 CultureAndRegionInfoBuilder(String, CultureAndRegionModifiers) 생성자를 호출하여 인스턴스화합니다. 기존 문화권을 바꾸려면 해당 문화권의 이름과 CultureAndRegionModifiers.Replacement 열거형 값을 생성자에 전달합니다. 새 문화권 또는 추가 문화권을 만들려면 고유한 문화권 이름과 CultureAndRegionModifiers.NeutralCultureAndRegionModifiers.None 열거형 값을 전달합니다.

   > ⓘ **참고**
   >
   > CultureAndRegionModifiers.Replacement 개체를 열거형 값을 사용하여 인스턴스화하는 경우, CultureAndRegionInfoBuilder 개체의 속성은 CultureAndRegionInfoBuilder 개체의 값으로 자동으로 채워집니다.

2. 새 문화나 추가 문화를 만드는 경우에:

   - CultureAndRegionInfoBuilder 개체의 속성을 채우려면, LoadDataFromCultureInfo 메서드를 호출하고 속성 값이 새 개체와 유사한 CultureInfo 개체를 전달하세요.
   - CultureAndRegionInfoBuilder 객체의 지역 속성을 채우려면 LoadDataFromRegionInfo 메서드를 호출하고 사용자 지정 문화권의 영역을 나타내는 RegionInfo 객체를 전달하십시오.

3. 필요에 따라 개체의 CultureAndRegionInfoBuilder 속성을 수정합니다.

4. 사용자 지정 문화권을 별도의 루틴에 등록하려는 경우 메서드를 호출합니다 Save . 이렇게 하면 별도의 사용자 지정 문화권 설치 루틴에서 로드하고 등록할 수 있는 XML 파일이 생성됩니다.

사용자 지정 문화권 등록

문화권을 만드는 애플리케이션과 별개인 사용자 지정 문화권에 대한 등록 애플리케이션을 개발하는 경우 메서드를 호출 CreateFromLdml 하여 사용자 지정 문화권의 정의가 포함된 XML 파일을 로드하고 개체를 CultureAndRegionInfoBuilder 인스턴스화합니다. 등록을 처리하려면 메서드를 호출합니다 Register . 등록이 성공하려면 사용자 지정 문화권을 등록하는 애플리케이션이 대상 시스템에서 관리자 권한으로 실행되어야 합니다. 그렇지 않으면 Register 호출이 UnauthorizedAccessException 예외를 발생시킵니다.

> **⚠ 경고**
>
> 문화권 데이터는 시스템 간에 다를 수 있습니다. 클래스 **CultureAndRegionInfoBuilder**를 사용하여 여러 시스템에서 일관된 사용자 지정 문화권을 만들고, 기존 **CultureInfo** 및 **RegionInfo** 객체에서 데이터를 로드하고 이를 사용자 지정하여 사용자 지정 문화권을 만드는 경우 두 가지 다른 유틸리티를 개발해야 합니다. 첫 번째는 사용자 지정 문화권을 만들고 XML 파일에 저장합니다. 두 번째 메서드를 **CreateFromLdml** 사용 하 여 XML 파일에서 사용자 지정 문화권을 로드 하 고 대상 컴퓨터에 등록 합니다.

등록 프로세스는 다음 작업을 수행합니다.

- 개체에 정의된 정보를 포함하는 CultureAndRegionInfoBuilder 파일을 만듭니다.
- 대상 컴퓨터의 %windir%\Globalization 시스템 디렉터리에 *.nlp* 파일을 저장합니다. 이렇게 하면 사용자 지정 문화권의 설정이 세션 간에 유지됩니다. CultureAndRegionInfoBuilder *(.nlp* 파일이 시스템 디렉터리에 저장되므로 메서드에 관리 권한이 필요합니다.)
- 다음에 새 사용자 지정 문화권을 만들라는 요청이 있을 때 내부 캐시 대신 %windir%\Globalization 시스템 디렉터리를 검색하도록 .NET을 준비합니다.

사용자 지정 문화권이 성공적으로 등록되면 .NET에서 미리 정의된 문화권과 구별할 수 없습니다. 메서드를 호출하여 로컬 컴퓨터에서 CultureAndRegionInfoBuilder 파일을 제거할 때까지 사용자 지정 문화를 사용할 수 있습니다.

사용자 지정 문화권 인스턴스화

다음 방법 중 하나로 사용자 지정 문화권의 인스턴스를 만들 수 있습니다.

- CultureInfo.CultureInfo 생성자를 문화권 이름으로 호출합니다.
- 문화권 이름을 사용하여 CultureInfo.CreateSpecificCulture 메서드를 호출합니다.
- 문화권 이름을 사용하여 CultureInfo.GetCultureInfo 메서드를 호출합니다.

또한 메서드에서 반환 CultureInfo 되는 개체의 CultureInfo.GetCultures 배열에는 사용자 지정 문화권이 포함됩니다.

System.Globalization.CultureAndRegionInfoBuilder 생성자

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 문서는 생성자와 CultureAndRegionInfoBuilder(String, CultureAndRegionModifiers) 관련이 있습니다.

매개 변수는 `cultureName` 새 CultureAndRegionInfoBuilder 개체의 이름을 지정합니다.

매개 `flags` 변수는 새 개체가 새 CultureAndRegionModifiers 사용자 지정 문화권인지 아니면 기존 중립 문화권, 특정 문화권 또는 Windows 로캘을 대체하는지 여부를 지정하는 값에 사용됩니다CultureAndRegionInfoBuilder.

매개 변수가 `cultureName` Windows 로캘 CultureAndRegionInfoBuilder 에서 생성된 기존 .NET 문화권, 등록된 사용자 지정 문화권 또는 문화권을 지정하는 경우 생성자는 새 개체를 문화권 및 국가/지역 정보로 자동으로 채웁니다 CultureAndRegionInfoBuilder .

CultureAndRegionInfoBuilder 및 LoadDataFromCultureInfo 메서드를 호출하여 새 LoadDataFromRegionInfo 개체에 문화권 및 국가/지역 정보를 채운다.

사용자 지정 문화권 이름

새 사용자 지정 문화권에 대한 매개 변수의 `cultureName` 기본 형식은 "[`prefix`-][-]`language`[-`region`[-`suffix`[-…]]]"입니다. 여기서 `language` 구성 요소는 필수이며 `prefix`, `region` `suffix`구성 요소는 선택 사항입니다. 각 구성 요소의 최대 길이는 8자이고 전체 `cultureName` 매개 변수의 최대 길이는 84자입니다.

구성 `prefix` 요소는 IANA(Internet Assigned Numbers Authority) ID입니다. IANA에 등록된 문화권 이름에 대해 "i-" 또는 "I-"를 지정하거나, 비공개로 예약된 문화권 이름의 경우 "x-" 또는 "X-"를 지정합니다. 그렇지 않으면 접두사는 필요하지 않습니다. 자세한 내용은 RFC 4646, "언어 식별 태그"를 참조하세요.

`language` 매개 변수의 `cultureName` 구성 요소는 ISO 639-1에서 파생된 소문자 두 글자 코드를 지정하고 `region` ISO 3166에서 파생된 대문자 2자 코드를 지정합니다. 예를 들어 en-US 미국에서 말한 대로 영어를 의미합니다. 구성 요소의 부재는 `region` 중립 문화권을 의미합니다.

.NET에 포함된 문화권의 이름과 동일한 A `cultureName` 는 대체(재정의) 문화권을 의미합니다. 대체 문화권의 속성에 할당할 수 있는 값은 제한됩니다. 이러한 제한 사항에 대한 자세한 내용은 각 속성에 대한 예외를 참조하세요.

애플리케이션은 구성 요소를 사용하여 `suffix` 유사한 문화권을 구분합니다. 예를 들어 ABC와 XYZ라는 두 회사는 새로운 ASP.NET 웹 서비스를 만들고 공유하여 전 세계 여러 시장에서 제품을 홍보합니다. 서비스의 웹 페이지에는 사용자의 문화권에 따라 각 회사의 지역 로고 및 지역 전화 번호와 같은 정보가 표시됩니다. 각 웹 페이지의 문화권별 콘텐츠는 문화권 이름으로 식별되고 회사 이름으로 한정된 별도의 리소스 파일에 있습니다. 예를 들어 en-US 및 ja-JP 문화권에 대한 리소스 파일의 이름은 en-US-ABC, en-US-XYZ, ja-JP-ABC 및 ja-JP-XYZ입니다. "ABC" 및 "XYZ" 접미사를 사용하면 웹 서비스에서 동일한 애플리케이션 논리를 사용하여 다른 시장별 정보를 표시할 수 있습니다.

`suffix` 구성 요소는 하위 구성 요소로 구성될 수 있습니다. 여기서 각 하위 구성 요소는 하이픈으로 구분되고 각 하위 구성 요소의 최대 길이는 8자입니다. 예를 들어 "en-US-honda-cars"가 매개 변수인 `cultureName` 경우 "-honda-cars"가 구성 요소입니다 `suffix` .

CultureInfo 클래스

아티클 • 2025. 03. 23.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

클래스는 CultureInfo 언어, 하위 언어, 국가/지역, 달력 및 특정 문화권과 관련된 규칙과 같은 문화권별 정보를 제공합니다. 이 클래스는 또한 DateTimeFormatInfo, NumberFormatInfo, CompareInfo 및 TextInfo 개체의 문화권별 인스턴스에 대한 액세스를 제공합니다. 이러한 개체는 대/소문자 처리, 날짜 및 숫자 형식 지정, 문자열 비교 등 문화권별 작업에 필요한 정보를 포함하고 있습니다. CultureInfo 클래스는 String, DateTime, DateTimeOffset 및 숫자 형식과 같은 문화권별 데이터를 형식화하거나 구문 분석하거나 조작하는 클래스에서 직접 또는 간접적으로 사용됩니다.

문화권 이름 및 식별자

클래스는 CultureInfo RFC 4646을 기반으로 각 문화권에 대해 고유한 이름을 지정합니다. 이름은 언어와 연결된 ISO 639 2자 또는 3자 소문자 문화권 코드와 국가 또는 지역과 연결된 ISO 3166 2자 대문자 하위 문화권 코드의 조합입니다. 또한 Windows 10 이상에서 실행되는 앱의 경우 유효한 BCP-47 언어 태그에 해당하는 문화권 이름이 지원됩니다.

> **참고**
>
> 문화권 이름이 클래스 생성자 또는 메서드(예: 또는CreateSpecificCulture)에 CultureInfo 전달되는 경우 해당 대/소문자는 중요하지 않습니다.

RFC 4646을 기반으로 하는 문화권 이름의 형식은 *languagecode2*-*country/regioncode2* *languagecode2*두 글자 언어 코드이며 *country/regioncode2* 두 글자 하위 문화권 코드입니다. 예를 들어 `ja-JP` 일본어(일본) 및 `en-US` 영어(미국)가 있습니다. 두 글자 언어 코드를 사용할 수 없는 경우 ISO 639-3에 정의된 세 글자 코드가 사용됩니다.

일부 문화권 이름은 ISO 15924 스크립트도 지정합니다. 예를 들어 Cyrl은 키릴 자모 스크립트를 지정하고 Latn은 라틴어 스크립트를 지정합니다. 스크립트를 포함하는 문화권 이름은 패턴을 *languagecode2*-*scripttag*-*country/regioncode2*사용합니다. 이러한 유형의 문화 이름의 `uz-Cyrl-UZ` 예로는 우즈벡어(키릴 문자, 우즈베키스탄)가 있습니다. Windows Vista 이전의 Windows 운영 체제에서 스크립트를 포함하는 문화권 이름은 *languagecode2*-*country/regioncode2*-*scripttag* 패턴을 사용합니다. 예를 들어, 우즈베크어(키릴 자모, 우즈베키스탄)의 경우 `uz-UZ-Cyrl`입니다.

중립 문화권은 두 글자 소문자 언어 코드로만 지정됩니다. 예를 들어 프랑스어 `fr` 의 중립 문화권을 지정하고 `de` 독일어의 중립 문화권을 지정합니다.

> **참고**
>
> 이 규칙과 모순되는 두 개의 문화권 이름이 있습니다. 중국어(간체) 및 중국어(번체)라는 `zh-Hans zh-Hant` 문화권은 중립 문화권입니다. 문화권 이름은 현재 표준을 나타내며 이전 이름과 `zh-CHS zh-CHT` 이름을 사용할 이유가 없는 한 사용해야 합니다.

문화권 식별자는 표준 국제 숫자 약어이며 설치된 문화권 중 하나를 고유하게 식별하는 데 필요한 구성 요소가 있습니다. 애플리케이션은 미리 정의 된 문화권 식별자를 사용 하거나 사용자 지정 식별자를 정의할 수 있습니다.

미리 정의된 특정 문화권 이름 및 식별자는 이 클래스와 네임스페이스의 다른 클래스에서 System.Globalization 사용됩니다. Windows 시스템에 대한 자세한 문화권 정보는 Windows에서 **지원하는** 언어/지역 이름 목록에서 언어 태그 열을 참조하세요. 문화권 이름은 BCP 47 에 정의된 표준을 따릅니다.

문화권 이름 및 식별자는 특정 컴퓨터에서 찾을 수 있는 문화권의 하위 집합만 나타냅니다. Windows 버전 또는 서비스 팩은 사용 가능한 문화권을 변경할 수 있습니다. 애플리케이션은 CultureAndRegionInfoBuilder 클래스를 사용하여 사용자 지정 문화권을 추가할 수 있습니다. 사용자는 Microsoft Locale Builder 도구를 사용하여 고유한 사용자 지정 문화권을 추가할 수 있습니다. Microsoft Locale Builder는 `CultureAndRegionInfoBuilder` 클래스를 사용하여 관리 코드로 작성된 것입니다.

여러 고유 이름은 문화권, 특히 다음 클래스 멤버와 연결된 이름과 밀접하게 연관되어 있습니다.

- CultureInfo.ToString
- CultureInfo.Name
- CompareInfo.Name

고정, 중립, 특정 문화권

문화권은 일반적으로 고정 문화권, 중립 문화권 및 특정 문화권의 세 집합으로 그룹화됩니다.

불변 문화는 문화에 민감하지 않습니다. 애플리케이션은 빈 문자열("") 또는 식별자를 사용하여 불변 문화권을 지정합니다. InvariantCulture 는 고정 문화권의 인스턴스를 정의합니다. 영어와 연결되지만 국가/지역과는 연결되지 않습니다. 문화권이 필요한 네임스페이스의 `Globalization` 거의 모든 메서드에서 사용됩니다.

중립 문화권은 언어와 연결되지만 국가/지역과는 관련이 없는 문화권입니다. 특정 문화권은 언어 및 국가/지역과 연결된 문화권입니다. 예를 들어, `fr`은 프랑스 문화에 대한 중립적 이름이고, `fr-FR`은 특정 프랑스(프랑스) 문화권의 이름입니다. 중국어(간체) 및 중국어(번체)도 중립 문화권으로 간주됩니다.

포함된 데이터가 임의 CompareInfo 이므로 중립 문화권에 대한 클래스 인스턴스를 만드는 것은 권장되지 않습니다. 데이터를 표시하고 정렬하려면 언어와 지역을 모두 지정합니다. Name 또한 중립 문화권에 대해 만든 개체의 CompareInfo 속성은 국가만 반환하며 지역을 포함하지 않습니다.

정의된 문화권에는 특정 문화권의 부모가 중립 문화권이고 중립 문화권의 부모가 고정 문화권인 계층 구조가 있습니다. 속성에는 Parent 특정 문화권과 연결된 중립 문화권이 포함됩니다. 사용자 지정 문화는 이 패턴에 따라서 Parent 속성을 정의해야 합니다.

운영 체제에서 특정 문화권에 대한 리소스를 사용할 수 없는 경우 연결된 중립 문화권에 대한 리소스가 사용됩니다. 중립 문화권에 대한 리소스를 사용할 수 없는 경우 주 어셈블리에 포함된 리소스가 사용됩니다. 리소스 대체 프로세스에 대한 자세한 내용은 리소스 패키징 및 배포를 참조 하세요.

Windows API의 로캘 목록은 .NET에서 지원하는 문화권 목록과 약간 다릅니다. 예를 들어 p/invoke 메커니즘을 통해 Windows와의 상호 운용성이 필요한 경우 애플리케이션은 운영 체제에 대해 정의된 특정 문화권을 사용해야 합니다. 특정 문화권을 사용하면 동일한 Windows 로캘과 일관성을 유지할 수 있습니다. 이 로캘은 동일한 LCID로캘 식별자로 식별됩니다.

DateTimeFormatInfo 또는 NumberFormatInfo는 불변 문화권이나 특정 문화권에 대해서만 만들 수 있으며, 중립 문화권을 위한 것은 만들 수 없습니다.

DateTimeFormatInfo.Calendar이 TaiwanCalendar이지만, Thread.CurrentCulture이 `zh-TW`로 설정되지 않은 경우, DateTimeFormatInfo.NativeCalendarName, DateTimeFormatInfo.GetEraName, 및 DateTimeFormatInfo.GetAbbreviatedEraName은 빈 문자열("")을 반환합니다.

사용자 지정 문화권

Windows에서 사용자 지정 로캘을 만들 수 있습니다. 자세한 내용은 사용자 지정 로캘을 참조하세요.

CultureInfo 및 문화 데이터

.NET은 구현, 플랫폼 및 버전에 따라 다양한 원본 중 하나에서 해당 문화 데이터를 파생합니다.

- Unix 플랫폼 또는 Windows 10 이상 버전에서 실행되는 모든 버전의 .NET(Core)에서는 ICU(International Components for Unicode) 라이브러리에서 문화권 데이터를 제공합니다. ICU 라이브러리의 특정 버전은 개별 운영 체제에 따라 달라집니다.
- Windows 9 및 이전 버전에서 실행되는 모든 버전의 .NET(Core)에서는 Windows 운영 체제에서 문화권 데이터를 제공합니다.
- .NET Framework 4 이상 버전에서는 Windows 운영 체제에서 문화권 데이터를 제공합니다.

따라서 특정 .NET 구현, 플랫폼 또는 버전에서 사용할 수 있는 문화권은 다른 .NET 구현, 플랫폼 또는 버전에서 사용할 수 없습니다.

일부 `CultureInfo` 개체는 기본 플랫폼에 따라 다릅니다. 특히 `zh-CN`중국어(간체, 중국) 및 `zh-TW`중국어(번체, 대만)는 Windows 시스템에서 사용할 수 있는 문화권이지만 Unix 시스템에서는 별칭이 지정된 문화권입니다. "zh-CN"은 "zh-Hans-CN" 문화권의 별칭이며 "zh-TW"는 "zh-Hant-TW" 문화권의 별칭입니다. 별명이 지정된 문화권은 GetCultures 메서드 호출에서 반환되지 않으며, 이는 Windows 대응 문화권과 다른 Parent 문화권 및 속성 값을 가질 수 있습니다. 문화권 `zh-CN` 의 `zh-TW` 경우 이러한 차이점은 다음과 같습니다.

- Windows 시스템에서 "zh-CN" 문화권의 부모 문화권은 "zh-Hans"이고 " zh-TW" 문화권의 부모 문화권은 "zh-Hant"입니다. 이러한 두 문화권의 부모 문화권은 "zh"입니다. Unix 시스템에서 두 문화권의 부모는 "zh"입니다. 즉, "zh-CN" 또는 "zh-TW" 문화권에 문화권별 리소스를 제공하지 않고 중립 "zh-Hans" 또는 "zh-Hant" 문화권에 대한 리소스를 제공하는 경우 애플리케이션은 Unix가 아닌 Windows에서 중립 문화권에 대한 리소스를 로드합니다. Unix 시스템에서는 스레드 CurrentUICulture 를 "zh-Hans" 또는 "zh-Hant"로 명시적으로 설정해야 합니다.

- Windows 시스템에서는 "zh-CN" 문화권을 나타내는 인스턴스를 호출 CultureInfo.Equals 하고 "zh-Hans-CN" 인스턴스를 전달하면 반환됩니다 `true`. Unix 시스템에서 메서드 호출이 반환됩니다 `false`. 이 동작은 "zh-TW" Equals 인스턴스를 호출 CultureInfo 하고 "zh-Hant-Tw" 인스턴스를 전달하는 데에도 적용됩니다.

동적 문화 데이터

고정 문화권을 제외하고 문화권 데이터는 동적입니다. 미리 정의된 문화권에서도 마찬가지입니다. 예를 들어 국가 또는 지역은 새 통화를 채택하거나, 단어의 철자를 변경하거나, 선호하는 일정을 변경하고, 문화 정의가 변경되어 이를 추적합니다. 사용자 지정 문화권은 예고 없이 변경될 수 있으며 특정 문화권은 사용자 지정 대체 문화권에 의해 재정의될 수 있습니다. 또한 아래에서 설명한 대로 개별 사용자가 문화적 선호를 재정의할 수 있습니다. 애플리케이션은 항상 실행 중에 문화권 데이터를 가져와야 합니다.

> **⊗ 주의**

데이터를 저장할 때 애플리케이션은 변경 불가능한 문화권이나 이진 형식, 또는 특정 문화권에 독립적인 형식을 사용해야 합니다. 고정 문화권 이외의 특정 문화권과 연결된 현재 값에 따라 저장된 데이터는 읽을 수 없게 되거나 문화권이 변경될 경우 의미가 변경될 수 있습니다.

현재 문화권 및 현재 사용자 인터페이스 문화권

.NET 애플리케이션의 모든 스레드에는 현재 문화와 현재 UI 문화가 있습니다. 현재 문화권은 날짜, 시간, 숫자 및 통화 값에 대한 서식 지정 규칙, 텍스트의 정렬 순서, 대/소문자 규칙 및 문자열을 비교하는 방법을 결정합니다. 현재 UI 문화 설정은 런타임에 문화권별 리소스를 자동으로 불러오는 데 사용됩니다.

> **참고**
>
> 현재 및 현재 UI 문화권이 스레드별로 결정되는 방법에 대한 자세한 내용은 문화권 및 스레드 섹션을 **참조하세요** . 새 애플리케이션 도메인에서 실행되는 스레드와 애플리케이션 도메인 경계를 넘는 스레드에서 현재 문화권과 UI 문화권이 어떻게 결정되는지에 대해 알고 싶다면, **문화권 및 애플리케이션 도메인** 섹션을 참조하세요. 작업 기반 비동기 작업을 수행하는 스레드에서 현재 및 현재 UI 문화권이 결정되는 방법에 대한 자세한 내용은 Culture 및 작업 기반 비동기 작업 섹션을 참조 **하세요** .

현재 문화권에 대한 자세한 내용은 속성을 참조하세요 CultureInfo.CurrentCulture . 현재 UI 문화권에 대한 자세한 내용은 CultureInfo.CurrentUICulture 속성 항목을 참조하십시오.

현재 문화 및 UI 문화 검색

다음 두 가지 방법 중 하나를 사용하여 현재 문화권을 나타내는 개체CultureInfo를 가져올 수 있습니다.

- CultureInfo.CurrentCulture 속성의 값을 검색하여.
- Thread.CurrentThread.CurrentCulture 속성의 값을 검색합니다.

다음 예제에서는 두 속성 값을 모두 검색하고 비교하여 동일한지 표시하고 현재 문화권의 이름을 표시합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class CurrentCultureEx
```

```
{
    public static void Main()
    {
        CultureInfo culture1 = CultureInfo.CurrentCulture;
        CultureInfo culture2 = Thread.CurrentThread.CurrentCulture;
        Console.WriteLine($"The current culture is {culture1.Name}");
        Console.WriteLine($"The two CultureInfo objects are equal: {culture1
== culture2}");
    }
}
// The example displays output like the following:
//     The current culture is en-US
//     The two CultureInfo objects are equal: True
```

다음 두 가지 방법 중 하나를 사용하여 현재 UI 문화권을 나타내는 CultureInfo 개체를 가져올 수 있습니다.

- CultureInfo.CurrentUICulture 속성 값을 검색하여 가져옵니다.
- Thread.CurrentThread.CurrentUICulture 속성의 값을 검색합니다.

다음 예제에서는 두 속성 값을 모두 검색하고 비교하여 동일한 것을 표시하고 현재 UI 문화권의 이름을 표시합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class CurrentUIEx
{
    public static void Main()
    {
        CultureInfo uiCulture1 = CultureInfo.CurrentUICulture;
        CultureInfo uiCulture2 = Thread.CurrentThread.CurrentUICulture;
        Console.WriteLine($"The current UI culture is {uiCulture1.Name}");
        Console.WriteLine($"The two CultureInfo objects are equal:
{uiCulture1 == uiCulture2}");
    }
}
// The example displays output like the following:
//     The current UI culture is en-US
//     The two CultureInfo objects are equal: True
```

현재 및 현재 UI 문화권 설정

스레드의 문화권 및 UI 문화권을 변경하려면 다음을 수행합니다.

1. 클래스 생성자를 호출 CultureInfo 하고 문화권의 이름을 전달하여 해당 문화권을 나타내는 개체를 인스턴스화 CultureInfo 합니다. CultureInfo(String) 생성자는 새 문화권이 현재 Windows 문화권과 동일한 경우 사용자 재정의를 반영하는 CultureInfo 개체를 인스턴스화합니다. CultureInfo(String, Boolean) 생성자를 사용하면 새 문화권이 현재 Windows 문화권과 동일한 경우 새로 인스턴스화된 CultureInfo 개체가 사용자 재정의를 반영하는지 여부를 지정할 수 있습니다.

2. CultureInfo 개체를 .NET Core 및 .NET Framework 4.6 이상 버전의 CultureInfo.CurrentCulture 또는 CultureInfo.CurrentUICulture 속성에 할당합니다.

다음 예제에서는 현재 문화권을 검색합니다. 프랑스(프랑스) 문화권이 아닌 다른 문화권인 경우 현재 문화권을 프랑스어(프랑스)로 변경합니다. 그렇지 않으면 현재 문화권을 프랑스어(룩셈부르크)로 변경합니다.

C#

```
using System;
using System.Globalization;

public class ChangeEx1
{
    public static void Main()
    {
        CultureInfo current = CultureInfo.CurrentCulture;
        Console.WriteLine($"The current culture is {current.Name}");
        CultureInfo newCulture;
        if (current.Name.Equals("fr-FR"))
            newCulture = new CultureInfo("fr-LU");
        else
            newCulture = new CultureInfo("fr-FR");

        CultureInfo.CurrentCulture = newCulture;
        Console.WriteLine($"The current culture is now
{CultureInfo.CurrentCulture.Name}");
    }
}
// The example displays output like the following:
//     The current culture is en-US
//     The current culture is now fr-FR
```

다음 예제에서는 현재 문화를 가져옵니다. 슬로베니아 (슬로베니아) 문화가 아닌 경우, 현재의 문화를 슬로베니아 (슬로베니아) 문화로 변경합니다. 그렇지 않으면 현재의 문화 설정을 크로아티아(크로아티아)의 문화로 변경합니다.

C#

```
using System;
using System.Globalization;
```

```
public class ChangeUICultureEx
{
    public static void Main()
    {
        CultureInfo current = CultureInfo.CurrentUICulture;
        Console.WriteLine($"The current UI culture is {current.Name}");
        CultureInfo newUICulture;
        if (current.Name.Equals("sl-SI"))
            newUICulture = new CultureInfo("hr-HR");
        else
            newUICulture = new CultureInfo("sl-SI");

        CultureInfo.CurrentUICulture = newUICulture;
        Console.WriteLine($"The current UI culture is now
{CultureInfo.CurrentUICulture.Name}");
    }
}
// The example displays output like the following:
//     The current UI culture is en-US
//     The current UI culture is now sl-SI
```

모든 문화권 가져오기

메서드를 호출하여 특정 범주의 문화권 또는 로컬 컴퓨터에서 사용할 수 있는 모든 문화권의 배열을 검색할 GetCultures 수 있습니다. 예를 들어 사용자 지정 문화권, 특정 문화권 또는 중립 문화권을 단독으로 또는 함께 검색할 수 있습니다.

다음 예제에서는 메서드를 GetCultures 두 번 호출하고, 먼저 System.Globalization.CultureTypes 열거형 멤버를 사용하여 모든 사용자 지정 문화권을 검색한 다음 System.Globalization.CultureTypes , 열거형 멤버를 사용하여 모든 대체 문화권을 검색합니다.

C#

```
using System;
using System.Globalization;

public class GetCulturesEx
{
    public static void Main()
    {
        // Get all custom cultures.
        CultureInfo[] custom =
CultureInfo.GetCultures(CultureTypes.UserCustomCulture);
        if (custom.Length == 0)
        {
            Console.WriteLine("There are no user-defined custom cultures.");
        }
        else
        {
```

```
            Console.WriteLine("Custom cultures:");
            foreach (var culture in custom)
                Console.WriteLine($"   {culture.Name} --
{culture.DisplayName}");
        }
        Console.WriteLine();

        // Get all replacement cultures.
        CultureInfo[] replacements =
CultureInfo.GetCultures(CultureTypes.ReplacementCultures);
        if (replacements.Length == 0)
        {
            Console.WriteLine("There are no replacement cultures.");
        }
        else
        {
            Console.WriteLine("Replacement cultures:");
            foreach (var culture in replacements)
                Console.WriteLine($"   {culture.Name} --
{culture.DisplayName}");
        }
        Console.WriteLine();
    }
}
// The example displays output like the following:
//     Custom cultures:
//        x-en-US-sample -- English (United States)
//        fj-FJ -- Boumaa Fijian (Viti)
//
//     There are no replacement cultures.
```

문화 및 주제

새 애플리케이션 스레드가 시작될 때, 현재 문화권 및 현재 UI 문화권은 현재 스레드 문화권이 아니라 현재 시스템 문화권에 의해 정의됩니다. 다음 예제에서 차이점을 보여 줍니다. 애플리케이션 스레드의 현재 문화권과 현재 UI 문화권을 프랑스 (프랑스) 문화권 (fr-FR)으로 설정합니다. 현재 문화권이 이미 fr-FR인 경우 예제에서는 영어(미국) 문화권(en-US)으로 설정합니다. 세 개의 난수를 통화 값으로 표시한 다음 새 스레드를 만들어, 그 스레드에서 다시 세 개의 난수를 통화 값으로 표시합니다. 하지만 예제의 출력에서 알 수 있듯이, 새 스레드에서 표시되는 통화 값은 프랑스(프랑스) 문화의 서식 관례를 반영하지 않아, 기본 애플리케이션 스레드의 출력과 다릅니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class DefaultThreadEx
```

```
{
    static Random rnd = new Random();

    public static void Main()
    {
        if (Thread.CurrentThread.CurrentCulture.Name != "fr-FR")
        {
            // If current culture is not fr-FR, set culture to fr-FR.
            Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
            Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        }
        else
        {
            // Set culture to en-US.
            Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
            Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture("en-US");
        }
        ThreadProc();

        Thread worker = new Thread(ThreadProc);
        worker.Name = "WorkerThread";
        worker.Start();
    }

    private static void DisplayThreadInfo()
    {
        Console.WriteLine($"\nCurrent Thread Name:
'{Thread.CurrentThread.Name}'");
        Console.WriteLine($"Current Thread Culture/UI Culture:
{Thread.CurrentThread.CurrentCulture.Name}/{Thread.CurrentThread.CurrentUICu
lture.Name}");
    }

    private static void DisplayValues()
    {
        // Create new thread and display three random numbers.
        Console.WriteLine("Some currency values:");
        for (int ctr = 0; ctr <= 3; ctr++)
            Console.WriteLine($"   {rnd.NextDouble() * 10:C2}");
    }

    private static void ThreadProc()
    {
        DisplayThreadInfo();
        DisplayValues();
    }
}
// The example displays output similar to the following:
//       Current Thread Name: ''
//       Current Thread Culture/UI Culture: fr-FR/fr-FR
//       Some currency values:
```

```
//          8,11 €
//          1,48 €
//          8,99 €
//          9,04 €
//
//       Current Thread Name: 'WorkerThread'
//       Current Thread Culture/UI Culture: en-US/en-US
//       Some currency values:
//          $6.72
//          $6.35
//          $2.90
//          $7.72
```

해당 문화권을 나타내는 개체를 할당하여 CultureInfo 및 DefaultThreadCurrentUICulture, 애플리케이션 도메인의 모든 스레드에 대한 문화권 및 UI 문화권을 설정할 수 있습니다. 다음 예제에서는 동일한 문화권을 공유 하는 기본 애플리케이션 도메인의 모든 스레드를 확인 하려면 이러한 속성을 사용 합니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class SetThreadsEx
{
    static Random rnd = new Random();

    public static void Main()
    {
        if (Thread.CurrentThread.CurrentCulture.Name != "fr-FR")
        {
            // If current culture is not fr-FR, set culture to fr-FR.
            CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
            CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        }
        else
        {
            // Set culture to en-US.
            CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
            CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.CreateSpecificCulture("en-US");
        }
        ThreadProc();

        Thread worker = new Thread(SetThreadsEx.ThreadProc);
        worker.Name = "WorkerThread";
        worker.Start();
    }
```

```
    private static void DisplayThreadInfo()
    {
        Console.WriteLine($"\nCurrent Thread Name:
'{Thread.CurrentThread.Name}'");
        Console.WriteLine($"Current Thread Culture/UI Culture:
{Thread.CurrentThread.CurrentCulture.Name}/{Thread.CurrentThread.CurrentUICu
lture.Name}");
    }

    private static void DisplayValues()
    {
        // Create new thread and display three random numbers.
        Console.WriteLine("Some currency values:");
        for (int ctr = 0; ctr <= 3; ctr++)
            Console.WriteLine($"   {rnd.NextDouble() * 10:C2}");
    }

    private static void ThreadProc()
    {
        DisplayThreadInfo();
        DisplayValues();
    }
}
// The example displays output similar to the following:
//       Current Thread Name: ''
//       Current Thread Culture/UI Culture: fr-FR/fr-FR
//       Some currency values:
//          6,83 €
//          3,47 €
//          6,07 €
//          1,70 €
//
//       Current Thread Name: 'WorkerThread'
//       Current Thread Culture/UI Culture: fr-FR/fr-FR
//       Some currency values:
//          9,54 €
//          9,50 €
//          0,58 €
//          6,91 €
```

> **⚠ 경고**
>
> 하지만 DefaultThreadCurrentCulture 및 DefaultThreadCurrentUICulture 속성은 정적 멤버이지만, 이러한 속성 값이 설정될 때의 현재 애플리케이션 도메인에 대해서만 기본 문화권 및 기본 UI 문화권을 정의합니다. 자세한 내용은 다음 섹션 **문화권 및 애플리케이션 도메인**을 참조하세요.

값을 DefaultThreadCurrentCulture 속성과 DefaultThreadCurrentUICulture 속성에 할당할 때, 명시적으로 문화권이 할당되지 않은 경우 애플리케이션 도메인의 스레드의 문화권과

UI 문화권도 변경됩니다. 그러나 이러한 스레드는 현재 애플리케이션 도메인에서 실행 하는 동안에 새 culture 설정을 반영 합니다. 이러한 스레드를 다른 애플리케이션 도메인에서 실행 하는 경우 해당 문화권에 해당 애플리케이션 도메인에 대해 정의 된 기본 문화권이 됩니다. 결과적으로, DefaultThreadCurrentCulture 및 DefaultThreadCurrentUICulture 속성에 의존하지 않고 항상 주 애플리케이션 스레드의 문화권을 설정할 것을 권장합니다.

문화 및 응용 분야

DefaultThreadCurrentCulture 및 DefaultThreadCurrentUICulture 하는 속성 값을 설정 하거나 검색할 때 현재 애플리케이션 도메인에 대해서만 기본 문화권을 명시적으로 정의 하는 정적 속성입니다. 다음 예제에서는 기본 애플리케이션 도메인에서 기본 문화권과 기본 UI 문화권을 프랑스(프랑스어)로 설정합니다. 그런 다음 AppDomainSetup 클래스와 AppDomainInitializer 대리자를 사용하여 새 애플리케이션 도메인에서 기본 문화권과 UI 문화권을 러시아(러시아어)로 설정합니다. 그런 다음 단일 스레드는 각 애플리케이션 도메인에서 두 메서드를 실행합니다. 주의하세요 스레드의 문화권과 UI 문화권은 명시적으로 설정되지 않으며, 스레드가 실행 중인 애플리케이션 도메인의 기본 문화권 및 UI 문화권에서 파생됩니다. 또한 DefaultThreadCurrentCulture 및 DefaultThreadCurrentUICulture 속성은 메서드 호출 시 현재 애플리케이션 도메인의 기본 CultureInfo 값을 반환합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        // Set the default culture and display the current date in the
current application domain.
        Info info1 = new Info();
        SetAppDomainCultures("fr-FR");

        // Create a second application domain.
        AppDomainSetup setup = new AppDomainSetup();
        setup.AppDomainInitializer = SetAppDomainCultures;
        setup.AppDomainInitializerArguments = new string[] { "ru-RU" };
        AppDomain domain = AppDomain.CreateDomain("Domain2", null, setup);
        // Create an Info object in the new application domain.
        Info info2 =
(Info)domain.CreateInstanceAndUnwrap(typeof(Example).Assembly.FullName,
                                                              "Info");

        // Execute methods in the two application domains.
        info2.DisplayDate();
        info2.DisplayCultures();
```

```
        info1.DisplayDate();
        info1.DisplayCultures();
    }

    public static void SetAppDomainCultures(string[] names)
    {
        SetAppDomainCultures(names[0]);
    }

    public static void SetAppDomainCultures(string name)
    {
        try
        {
            CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture(name);
            CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.CreateSpecificCulture(name);
        }
        // If an exception occurs, we'll just fall back to the system
default.
        catch (CultureNotFoundException)
        {
            return;
        }
        catch (ArgumentException)
        {
            return;
        }
    }
}

public class Info : MarshalByRefObject
{
    public void DisplayDate()
    {
        Console.WriteLine($"Today is {DateTime.Now:D}");
    }

    public void DisplayCultures()
    {
        Console.WriteLine($"Application domain is
{AppDomain.CurrentDomain.Id}");
        Console.WriteLine($"Default Culture:
{CultureInfo.DefaultThreadCurrentCulture}");
        Console.WriteLine($"Default UI Culture:
{CultureInfo.DefaultThreadCurrentUICulture}");
    }
}
// The example displays the following output:
//       Today is 14 октября 2011 г.
//       Application domain is 2
//       Default Culture: ru-RU
//       Default UI Culture: ru-RU
//       Today is vendredi 14 octobre 2011
```

```
//       Application domain is 1
//       Default Culture: fr-FR
//       Default UI Culture: fr-FR
```

문화권 및 애플리케이션 도메인에 대한 자세한 내용은 애플리케이션 도메인 항목의 "애플리케이션 도메인 및 스레드" 섹션을 참조하세요 .

문화 및 작업 기반 비동기 작업

작업 기반 비동기 프로그래밍 패턴은 및 개체를 사용하여 스레드 풀 스레드에서 대리자를 비동기적으로 실행합니다 . 특정 작업이 실행되는 특정 스레드는 사전에 알려지지 않았지만 런타임에만 결정됩니다.

.NET Framework 4.6 이상 버전을 대상으로 하는 앱의 경우 문화는 비동기 작업 컨텍스트의 일부입니다. 즉, 기본적으로 비동기 작업은 시작된 스레드의 CurrentCulture 값과 CurrentUICulture 속성을 상속합니다. 현재 문화권 또는 현재 UI 문화권이 시스템 문화권과 다른 경우 현재 문화권은 스레드 경계를 넘어 비동기 작업을 실행하는 스레드 풀 스레드의 현재 문화권이 됩니다.

다음 예제에서는 간단한 설명을 제공합니다. 이 예제에서는 일부 숫자를 통화 형식으로 반환하는 Func<TResult> 델리게이트, `formatDelegate`,를 정의합니다. 이 예제에서는 현재 시스템 문화권을 프랑스어(프랑스) 또는 프랑스어(프랑스)가 이미 현재 문화권인 경우 영어(미국)로 변경합니다. 그런 다음:

- 주 앱 스레드에서 동기적으로 실행되도록 대리자를 직접 호출합니다.
- 스레드 풀 스레드에서 대리자를 비동기적으로 실행하는 작업을 만듭니다.
- 메서드 Task.RunSynchronously를 호출하여 주 앱 스레드에서 대리자를 동기적으로 실행하는 작업을 생성합니다.

예제의 출력과 같이 현재 문화권이 프랑스어(프랑스)로 변경되면 작업이 비동기적으로 호출되는 스레드의 현재 문화권이 해당 비동기 작업의 현재 문화권이 됩니다.

C#

```
using System;
using System.Globalization;
using System.Threading;
using System.Threading.Tasks;

public class AsyncCultureEx1
{
    public static void Main()
    {
        decimal[] values = { 163025412.32m, 18905365.59m };
        string formatString = "C2";
```

```
        string FormatDelegate()
        {
            string output = $"Formatting using the
{CultureInfo.CurrentCulture.Name} " +
            "culture on thread {Thread.CurrentThread.ManagedThreadId}.\n";
            foreach (decimal value in values)
                output += $"{value.ToString(formatString)}   ";

            output += Environment.NewLine;
            return output;
        }

        Console.WriteLine($"The example is running on thread
{Thread.CurrentThread.ManagedThreadId}");
        // Make the current culture different from the system culture.
        Console.WriteLine($"The current culture is
{CultureInfo.CurrentCulture.Name}");
        if (CultureInfo.CurrentCulture.Name == "fr-FR")
            Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
        else
            Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

        Console.WriteLine($"Changed the current culture to
{CultureInfo.CurrentCulture.Name}.\n");

        // Execute the delegate synchronously.
        Console.WriteLine("Executing the delegate synchronously:");
        Console.WriteLine(FormatDelegate());

        // Call an async delegate to format the values using one format
string.
        Console.WriteLine("Executing a task asynchronously:");
        var t1 = Task.Run(FormatDelegate);
        Console.WriteLine(t1.Result);

        Console.WriteLine("Executing a task synchronously:");
        var t2 = new Task<string>(FormatDelegate);
        t2.RunSynchronously();
        Console.WriteLine(t2.Result);
    }
}
// The example displays the following output:
//         The example is running on thread 1
//         The current culture is en-US
//         Changed the current culture to fr-FR.
//
//         Executing the delegate synchronously:
//         Formatting using the fr-FR culture on thread 1.
//         163 025 412,32 €   18 905 365,59 €
//
//         Executing a task asynchronously:
//         Formatting using the fr-FR culture on thread 3.
//         163 025 412,32 €   18 905 365,59 €
//
//         Executing a task synchronously:
```

```
//          Formatting using the fr-FR culture on thread 1.
//          163 025 412,32 €   18 905 365,59 €
```

DefaultThreadCurrentCulture 및 DefaultThreadCurrentUICulture은 앱별 도메인 속성으로, 이는 특정 애플리케이션 도메인에서 문화권이 명시적으로 할당되지 않은 모든 스레드에 대해 기본 문화권을 설정합니다. 그러나 .NET Framework 4.6 이상을 대상으로 하는 앱의 경우 태스크가 앱 도메인 경계를 넘더라도 호출 스레드의 문화권은 비동기 작업 컨텍스트의 일부로 유지됩니다.

CultureInfo 개체 직렬화

CultureInfo 개체가 직렬화될 때 실제로 저장되는 것은 Name 및 UseUserOverride 뿐입니다. 동일한 의미를 가지는 환경에서만 Name가 성공적으로 역직렬화됩니다. 다음 세 가지 예제에서는 항상 그렇지 않은 이유를 보여 줍니다.

- CultureTypes 속성 값이 CultureTypes.InstalledWin32Cultures이고, 해당 문화권이 Windows 운영 체제의 특정 버전에서 처음 도입된 경우, 이전 버전의 Windows에서는 이를 역직렬화할 수 없습니다. 예를 들어 Windows 10에서 문화권이 도입된 경우 Windows 8에서는 역직렬화할 수 없습니다.

- CultureTypes 값이 CultureTypes.UserCustomCulture이고, 이를 역직렬화하는 컴퓨터에 이 사용자 지정 문화권이 설치되어 있지 않으면, 값을 역직렬화할 수 없습니다.

- CultureTypes 값이 CultureTypes.ReplacementCultures이고 역직렬화가 이루어지는 컴퓨터에 이 대체 문화권이 없다면, 같은 이름으로 역직렬화되지만, 모든 특성이 동일하지는 않습니다. 예를 들어 컴퓨터 A의 대체 문화권이 컴퓨터 B가 아닌 대체 문화권이고 이 문화권을 참조하는 개체가 컴퓨터 A에서 직렬화되고 컴퓨터 B에서 역직렬화되는 경우 `en-US` CultureInfo 문화권의 사용자 지정 특성이 전송되지 않습니다. 문화는 성공적으로 역직렬화되지만 의미가 다르게 변합니다.

제어판 설정 무시

사용자는 제어판 지역 및 언어 옵션 부분을 통해 Windows의 현재 문화권과 연결된 일부 값을 재정의하도록 선택할 수 있습니다. 예를 들어 사용자는 다른 형식으로 날짜를 표시하거나 문화권의 기본값이 아닌 통화를 사용하도록 선택할 수 있습니다. 일반적으로 귀하의 애플리케이션은 사용자 재정의를 존중해야 합니다.

UseUserOverride 지정된 문화권이 Windows `true` 의 현재 문화권과 일치하는 경우, CultureInfo는 DateTimeFormat 속성에서 반환된 DateTimeFormatInfo 인스턴스 및 NumberFormat 속성에서 반환된 NumberFormatInfo 인스턴스의 속성에 대한 사용자 설정을 포함하여 해당 재정의를 사용합니다. 사용자 설정이 연결된 문화권과 CultureInfo호

환되지 않는 경우(예: 선택한 달력이 하나 OptionalCalendars도 아닌 경우) 메서드의 결과와 속성 값이 정의되지 않습니다.

대체 정렬 순서

일부 문화권은 둘 이상의 정렬 순서를 지원합니다. 예시:

- 스페인어(스페인) 문화권에는 기본 국제 정렬 순서와 전통적인 정렬 순서의 두 가지 정렬 순서가 있습니다. 문화권 이름을 사용하여 개체 CultureInfo 를 `es-ES` 인스턴스화하면 국가별 정렬 순서가 사용됩니다. 문화권 이름을 사용하여 개체를 CultureInfo `es-ES-tradnl` 인스턴스화하면 기존 정렬 순서가 사용됩니다.
- (중국어(간체, PRC) 문화권은 `zh-CN` 발음(기본값) 및 스트로크 수의 두 가지 정렬 순서를 지원합니다. 문화권 이름을 사용하여 개체 CultureInfo 를 `zh-CN` 인스턴스화하면 기본 정렬 순서가 사용됩니다. 0x00020804 로컬 식별자를 사용하여 개체를 인스턴스화 CultureInfo 하면 문자열이 스트로크 수별로 정렬됩니다.

다음 표에서는 대체 정렬 순서를 지원하는 문화권과 기본 및 대체 정렬 순서의 식별자를 나열합니다.

테이블 확장

문화권 이름	문화	기본 정렬 이름 및 식별자	대체 정렬 이름 및 식별자
es-ES	스페인어(스페인)	국제: 0x00000C0A	전통적인: 0x0000040A
zh-TW	중국어(대만)	스트로크 수: 0x00000404	보포모포: 0x00030404
zh-CN	중국어(중화인민공화국)	발음: 0x00000804	스트로크 수: 0x00020804
zh-HK	중국어(홍콩 특별행정구)	스트로크 수: 0x00000c04	스트로크 수: 0x00020c04
zh-SG	중국어(싱가포르)	발음: 0x00001004	스트로크 수: 0x00021004
zh-MO	중국어(마카오 특별행정구)	발음: 0x00001404	스트로크 수: 0x00021404
ja-JP	일본어(일본)	기본값: 0x00000411	유니코드: 0x00010411
ko-KR	한국어(대한민국)	기본값: 0x00000412	한국어 Xwansung - 유니코드: 0x00010412
de-DE	독일어(독일)	사전: 0x00000407	전화 번호부 정렬 DIN: 0x00010407

문화권 이름	문화	기본 정렬 이름 및 식별자	대체 정렬 이름 및 식별자
hu-HU	헝가리어(헝가리)	기본값: 0x0000040e	기술 정렬: 0x0001040e
ka-GE	조지아어(조지아)	전통적: 0x00000437	최신 정렬: 0x00010437

현재 문화 및 UWP 앱

UWP(유니버설 Windows 플랫폼) 앱에서는 CurrentCulture와 CurrentUICulture 속성이 .NET Framework 및 .NET Core 앱에서와 마찬가지로 읽기/쓰기가 가능합니다. 그러나 UWP 앱은 단일 문화권을 인식합니다. CurrentCulture 속성 및 CurrentUICulture 속성은 Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages 컬렉션의 첫 번째 값에 매핑됩니다.

.NET 앱에서 현재 문화권은 스레드별 설정이며 CurrentCulture 속성은 CurrentUICulture 현재 스레드의 문화권 및 UI 문화권만 반영합니다. UWP 앱에서 현재 문화 설정은 전역 설정인 Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages 컬렉션에 매핑됩니다. CurrentCulture 또는 CurrentUICulture 속성을 설정하면 전체 앱의 문화권이 변경됩니다. 문화권은 스레드별로 설정할 수 없습니다.

System.Globalization.CultureInfo.CurrentCulture 속성

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

CurrentCulture 속성 및 관련 개체에서 반환되는 CultureInfo 개체는 날짜, 시간, 숫자 및 통화 값의 기본 형식, 텍스트 정렬 순서, 대/소문자 구분 규칙 및 문자열 비교를 결정합니다.

현재 문화는 실행 중인 스레드의 속성입니다. 이 속성을 새 문화권을 나타내는 CultureInfo 개체로 설정하면 `Thread.CurrentThread.CurrentCulture` 속성의 값도 변경됩니다. 그러나 항상 CultureInfo.CurrentCulture 속성을 사용하여 현재 문화권을 검색하고 설정하는 것이 좋습니다.

이 속성이 반환하는 `CultureInfo` 개체는 읽기 전용입니다. 즉, `DateTimeFormat` 변경하여 기존 개체를 변경할 수 없습니다. 날짜-시간 형식 또는 현재 문화권의 다른 측면을 변경하려면 새 `CultureInfo` 개체를 만들고 속성에 할당합니다.

스레드 문화 결정 방법

스레드가 시작되면 처음에는 다음과 같이 해당 문화권이 결정됩니다.

- 속성 값이 `null`이 아닌 경우, 스레드가 실행 중인 애플리케이션 도메인의 DefaultThreadCurrentCulture 속성에 의해 지정된 문화를 검색합니다.
- 스레드가 작업 기반 비동기 작업을 실행하는 스레드 풀 스레드인 경우 해당 문화권은 호출 스레드의 문화권에 따라 결정됩니다. 다음은 현재 문화권을 포르투갈어(브라질)로 변경하고 각각 스레드 ID, 작업 ID 및 현재 문화권을 표시하는 6개의 작업을 시작하는 예제입니다. 각 작업(및 스레드)은 호출 스레드의 컬처를 상속받았습니다.

```
C#

using System;
using System.Collections.Generic;
using System.Globalization;
using System.Runtime.Versioning;
using System.Threading;
using System.Threading.Tasks;

public class Example14
{
    public static async Task Main()
```

```
    {
        var tasks = new List<Task>();
        Console.WriteLine($"The current culture is
{Thread.CurrentThread.CurrentCulture.Name}");
        Thread.CurrentThread.CurrentCulture = new CultureInfo("pt-BR");
        // Change the current culture to Portuguese (Brazil).
        Console.WriteLine($"Current culture changed to
{Thread.CurrentThread.CurrentCulture.Name}");
        Console.WriteLine($"Application thread is thread
{Thread.CurrentThread.ManagedThreadId}");
        // Launch six tasks and display their current culture.
        for (int ctr = 0; ctr <= 5; ctr++)
            tasks.Add(Task.Run(() =>
            {
                Console.WriteLine($"Culture of task {Task.CurrentId} on
thread {Thread.CurrentThread.ManagedThreadId} is
{Thread.CurrentThread.CurrentCulture.Name}");
            }));

        await Task.WhenAll(tasks.ToArray());
    }
}
// The example displays output like the following:
//     The current culture is en-US
//     Current culture changed to pt-BR
//     Application thread is thread 9
//     Culture of task 2 on thread 11 is pt-BR
//     Culture of task 1 on thread 10 is pt-BR
//     Culture of task 3 on thread 11 is pt-BR
//     Culture of task 5 on thread 11 is pt-BR
//     Culture of task 6 on thread 11 is pt-BR
//     Culture of task 4 on thread 10 is pt-BR
```

자세한 내용은 Culture 및 작업 기반 비동기 작업참조하세요.

- Windows에서 `GetUserDefaultLocaleName` 함수를 호출하고, Unix와 유사한 시스템에서는 ICU 에서 범주 `LC_MESSAGES`를 사용하여 POSIX `setlocale` 함수를 호출하는 `uloc_getDefault` 함수를 호출합니다.

시스템 설치 문화권 또는 사용자의 기본 문화권과 다른 특정 문화권을 설정하고 애플리케이션이 여러 스레드를 시작하는 경우 스레드가 실행되는 애플리케이션 도메인의 DefaultThreadCurrentCulture 속성에 문화권을 할당하지 않는 한 해당 스레드의 현재 문화권은 `GetUserDefaultLocaleName` 함수에서 반환되는 문화권이 됩니다.

스레드 문화권이 결정되는 방법에 대한 자세한 내용은 CultureInfo 참조 페이지의 "문화권 및 스레드" 섹션을 참조하세요.

현재 문화 설정 가져오기

CultureInfo.CurrentCulture 속성은 스레드별 설정입니다. 즉, 각 스레드에는 고유한 문화권이 있을 수 있습니다. 다음 예제와 같이 CultureInfo.CurrentCulture 속성의 값을 검색하여 현재 스레드의 문화권을 가져옵니다.

C#

```
using System;
using System.Globalization;

public class Example5
{
   public static void Main()
   {
      CultureInfo culture = CultureInfo.CurrentCulture;
      Console.WriteLine($"The current culture is {culture.NativeName}
[{culture.Name}]");
   }
}
// The example displays output like the following:
//       The current culture is English (United States) [en-US]
```

CurrentCulture 속성을 명시적으로 설정

기존 스레드에서 사용하는 문화권을 변경하려면 CultureInfo.CurrentCulture 속성을 새 문화권으로 설정합니다. 이러한 방식으로 스레드의 문화권을 명시적으로 변경하는 경우 스레드가 애플리케이션 도메인 경계를 넘으면 해당 변경 내용이 유지됩니다. 다음은 현재 스레드 문화권을 네덜란드어(네덜란드)로 변경하는 예제입니다. 또한 현재 스레드가 애플리케이션 도메인 경계를 넘으면 현재 문화권이 변경된 상태로 유지됨을 보여 줍니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Info11 : MarshalByRefObject
{
    public void ShowCurrentCulture()
    {
        Console.WriteLine($"Culture of {Thread.CurrentThread.Name} in
application domain {AppDomain.CurrentDomain.FriendlyName}:
{CultureInfo.CurrentCulture.Name}");
    }
}

public class Example11
{
    public static void Main()
    {
```

```
        Info11 inf = new Info11();
        // Set the current culture to Dutch (Netherlands).
        Thread.CurrentThread.Name = "MainThread";
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("nl-
NL");
        inf.ShowCurrentCulture();

        // Create a new application domain.
        AppDomain ad = AppDomain.CreateDomain("Domain2");
        Info11 inf2 =
(Info11)ad.CreateInstanceAndUnwrap(typeof(Info11).Assembly.FullName,
"Info11");
        inf2.ShowCurrentCulture();
    }
}
// The example displays the following output:
//       Culture of MainThread in application domain ChangeCulture1.exe: nl-
NL
//       Culture of MainThread in application domain Domain2: nl-NL
```

ⓘ 참고

CultureInfo.CurrentCulture 속성을 이용해 문화를 변경하려면 ControlThread 값이 설정된 SecurityPermission 권한이 필요합니다. 스레드 조작은 스레드와 연결된 보안 상태 때문에 위험합니다. 따라서 이 권한은 신뢰할 수 있는 코드에만 부여된 다음 필요에 따라 지정해야 합니다. 반 신뢰할 수 있는 코드에서는 스레드 문화권을 변경할 수 없습니다.

.NET Framework 4부터 현재 스레드 문화권을 특정 문화권(예: 프랑스어(캐나다) 또는 중립 문화권(예: 프랑스어)으로 명시적으로 변경할 수 있습니다. CultureInfo 개체가 중립 문화권을 나타내는 경우 Calendar, CompareInfo, DateTimeFormat, NumberFormat및 TextInfo 같은 CultureInfo 속성 값은 중립 문화권과 연결된 특정 문화권을 반영합니다. 예를 들어 영어 중립 문화권의 주요 문화권은 영어(미국)입니다. 독일 문화권의 지배적인 문화는 독일(독일)입니다. 다음 예제에서는 현재 문화권이 특정 문화권, 프랑스어(캐나다) 및 중립 문화권(프랑스어)으로 설정된 경우 서식의 차이를 보여 줍니다.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example12
{
   public static void Main()
   {
      double value = 1634.92;
      CultureInfo.CurrentCulture = new CultureInfo("fr-CA");
```

```
      Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.Name}");
      Console.WriteLine($"{value:C2}\n");

      Thread.CurrentThread.CurrentCulture = new CultureInfo("fr");
      Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.Name}");
      Console.WriteLine($"{value:C2}");
   }
}
// The example displays the following output:
//       Current Culture: fr-CA
//       1 634,92 $
//
//       Current Culture: fr
//       1 634,92 €
```

다음 예제와 같이 CultureInfo.CurrentCulture 속성을 HttpRequest.UserLanguages 속성과 함께 사용하여 ASP.NET 애플리케이션의 CurrentCulture 속성을 사용자의 기본 문화권으로 설정할 수도 있습니다.

C#

```
CultureInfo.CurrentCulture =
CultureInfo.CreateSpecificCulture(Request13.UserLanguages[0]);
```

현재 문화 및 사용자 설정 재정의

Windows를 사용하면 제어판에서 **국가 및 언어 옵션** 사용하여 CultureInfo 개체 및 관련 개체의 표준 속성 값을 재정의할 수 있습니다. CurrentCulture 속성에서 반환된 CultureInfo 개체는 다음과 같은 경우에 이러한 사용자 재정의를 반영합니다.

- 현재 스레드 문화권이 Windows `GetUserDefaultLocaleName` 함수에 의해 암시적으로 설정된 경우
- DefaultThreadCurrentCulture 속성에서 정의한 현재 스레드 문화권이 현재 Windows 시스템 문화권에 해당하는 경우
- 현재 스레드 문화권이 CreateSpecificCulture 메서드에서 반환된 문화권으로 명시적으로 설정되고 해당 문화권이 현재 Windows 시스템 문화권에 해당하는 경우
- 현재 스레드 문화권이 CultureInfo(String) 생성자에 의해 인스턴스화된 문화권으로 명시적으로 설정된 경우 해당 문화권은 현재 Windows 시스템 문화권에 해당합니다.

일부 경우, 특히 서버 애플리케이션의 경우, 현재 문화권을 사용자 설정을 반영하는 CultureInfo 개체로 설정하는 것은 바람직하지 않을 수 있습니다. 대신 다음과 같은 방법으로 사용자 재정의를 반영하지 않는 CultureInfo 개체로 현재 문화권을 설정할 수 있습니다.

- `useUserOverride` 인수에 대한 `false` 값으로 CultureInfo(String, Boolean) 생성자를 호출합니다.
- 캐시된 읽기 전용 CultureInfo 개체를 반환하는 GetCultureInfo 메서드를 호출합니다.

현재 문화 및 UWP 앱

UWP(유니버설 Windows 플랫폼) 앱에서 CurrentCulture 속성은 .NET Framework 및 .NET Core 앱과 마찬가지로 읽기/쓰기가 가능합니다. 현재 문화권을 가져와서 설정하는 데 둘 다 사용할 수 있습니다. 그러나 UWP 앱은 현재 문화권과 현재 UI 문화권을 구분하지 않습니다. CurrentCulture 속성 및 CurrentUICulture 속성은 Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages 컬렉션의 첫 번째 값에 매핑됩니다.

.NET Framework 및 .NET Core 앱에서 현재 문화권은 스레드별 설정이며 CurrentCulture 속성은 현재 스레드의 문화권만 반영합니다. UWP 앱에서 현재 문화권은 글로벌 설정인 Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages 속성에 매핑됩니다. CurrentCulture 속성을 설정하면 전체 앱의 문화권이 변경됩니다. 문화권은 스레드별로 설정할 수 없습니다.

System.Globalization.CultureInfo.CurrentUICulture 속성

아티클 • 2025. 04. 04.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 속성은 CurrentUICulture 스레드당 속성입니다. 즉, 각 스레드에는 고유한 현재 UI 문화권이 있습니다. 이 속성은 CultureInfo 개체를 `System.Threading.Thread.CurrentThread.CurrentUICulture` 속성에 할당하여 검색하거나 설정하는 것과 동일합니다. 스레드가 시작되면 해당 UI 문화권은 처음에 다음과 같이 결정됩니다.

- 속성 값이 DefaultThreadCurrentUICulture이 아닌 경우, 스레드가 실행 중인 애플리케이션 도메인의 `null` 속성에 의해 지정된 문화를 검색합니다.
- 스레드가 작업 기반 비동기 작업을 실행하는 스레드 풀 스레드이고 앱이 .NET Framework 4.6 이상 버전의 .NET Framework를 대상으로 하는 경우 해당 UI 문화권은 호출 스레드의 UI 문화권에 의해 결정됩니다. 다음은 현재 UI 문화권을 포르투갈어(브라질)로 변경하고 각각 스레드 ID, 작업 ID 및 현재 UI 문화권을 표시하는 6개의 작업을 시작하는 예제입니다. 각 작업(및 스레드)은 호출 스레드의 UI 문화권을 상속했습니다.

C#

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Runtime.Versioning;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main()
    {
        var tasks = new List<Task>();
        Console.WriteLine($"The current UI culture is
{Thread.CurrentThread.CurrentUICulture.Name}");
        Thread.CurrentThread.CurrentUICulture = new CultureInfo("pt-
BR");
        // Change the current UI culture to Portuguese (Brazil).
        Console.WriteLine($"Current UI culture changed to
{Thread.CurrentThread.CurrentUICulture.Name}");
        Console.WriteLine($"Application thread is thread
{Thread.CurrentThread.ManagedThreadId}");
        // Launch six tasks and display their current culture.
```

```
        for (int ctr = 0; ctr <= 5; ctr++)
            tasks.Add(Task.Run(() =>
            {
                Console.WriteLine($"UI Culture of task {Task.CurrentId}
on thread {Thread.CurrentThread.ManagedThreadId} is
{Thread.CurrentThread.CurrentUICulture.Name}");
            }));

        await Task.WhenAll(tasks.ToArray());
    }
}
// The example displays output like the following:
//     The current UI culture is en-US
//     Current UI culture changed to pt-BR
//     Application thread is thread 9
//     UI Culture of task 2 on thread 11 is pt-BR
//     UI Culture of task 1 on thread 10 is pt-BR
//     UI Culture of task 3 on thread 11 is pt-BR
//     UI Culture of task 5 on thread 11 is pt-BR
//     UI Culture of task 6 on thread 11 is pt-BR
//     UI Culture of task 4 on thread 10 is pt-BR
```

자세한 내용은 설명서의 "문화권 및 작업 기반 비동기 작업" 섹션을 CultureInfo 참조하세요.

- Windows `GetUserDefaultUILanguage` 함수를 호출합니다.

스레드에서 사용하는 사용자 인터페이스 문화권을 변경하려면 속성을 새 문화권으로 설정합니다 Thread.CurrentUICulture . 이러한 방식으로 스레드의 UI 문화권을 명시적으로 변경하는 경우 스레드가 애플리케이션 도메인 경계를 넘으면 해당 변경 내용이 유지됩니다.

> **참고**
>
> 속성 값을 새 문화권을 나타내는 개체로 CultureInfo 설정하면 속성 값 `Thread.CurrentThread.CurrentCulture` 도 변경됩니다.

현재 UI 문화 설정 가져오기

속성은 CultureInfo.CurrentUICulture 스레드별 설정입니다. 즉, 각 스레드에는 자체 UI 문화권이 있을 수 있습니다. 속성 값을 검색하여 현재 스레드의 UI 문화권을 가져오는 방법은 다음 예제가 잘 보여줍니다.

C#

```
using System;
using System.Globalization;

public class Example2
{
    public static void Main()
    {
        CultureInfo culture = CultureInfo.CurrentUICulture;
        Console.WriteLine($"The current UI culture is {culture.NativeName}
[{culture.Name}]");
    }
}
// The example displays output like the following:
//       The current UI culture is English (United States) [en-US]
```

속성Thread.CurrentUICulture에서 현재 스레드의 UI 문화 설정 값을 조회할 수도 있습니다.

현재 UI 문화권을 명시적으로 설정

.NET Framework 4.6부터 속성에 새 문화권을 나타내는 개체를 할당하여 CultureInfo 현재 UI 문화 CultureInfo.CurrentUICulture 권을 변경할 수 있습니다. 현재 UI 문화권은 특정 문화권(예: en-US 또는 de-DE) 또는 중립 문화권(예: en 또는 de)으로 설정할 수 있습니다. 다음은 현재 UI 문화권을 fr-FR 또는 프랑스어(프랑스)로 설정하는 예제입니다.

C#

```
using System;
using System.Globalization;

public class Example1
{
    public static void Main()
    {
        Console.WriteLine($"The current UI culture:
{CultureInfo.CurrentUICulture.Name}");

        CultureInfo.CurrentUICulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        Console.WriteLine($"The current UI culture:
{CultureInfo.CurrentUICulture.Name}");
    }
}
// The example displays output like the following:
//       The current UI culture: en-US
//       The current UI culture: fr-FR
```

다중 스레드 애플리케이션에서 해당 문화권을 나타내는 개체를 스레드의 속성에 할당하여 CultureInfo 스레드의 Thread.CurrentUICulture UI 문화권을 명시적으로 설정할 수 있습니다. 설정하려는 문화권이 현재 스레드인 스레드인 경우 속성에 새 문화권을 할당할 CultureInfo.CurrentUICulture 수 있습니다. 스레드의 UI 문화권이 명시적으로 설정되면 해당 스레드는 애플리케이션 도메인 경계를 넘어 다른 애플리케이션 도메인에서 코드를 실행하더라도 동일한 문화권을 유지합니다.

현재 UI 문화권을 암시적으로 설정

기본 애플리케이션 스레드를 포함한 스레드가 처음 만들어지면 기본적으로 현재 UI 문화권은 다음과 같이 설정됩니다.

- 속성 값이 `null`가 아닌 경우, 현재 애플리케이션 도메인에 대해 DefaultThreadCurrentUICulture 속성으로 정의된 문화권을 사용합니다.
- 시스템의 기본 문화권을 사용하여 Windows 운영 체제를 사용하는 시스템에서 공용 언어 런타임은 Windows `GetUserDefaultUILanguage` 함수를 호출하여 현재 UI 문화권을 설정합니다. `GetUserDefaultUILanguage` 는 사용자가 설정한 기본 UI 문화권을 반환합니다. 사용자가 기본 UI 언어를 설정하지 않은 경우 시스템에 원래 설치된 언어 또는 문화 설정을 반환합니다.

스레드가 애플리케이션 경계를 넘어 다른 애플리케이션 도메인에서 코드를 실행하는 경우 해당 문화권은 새로 만든 스레드의 문화권과 동일한 방식으로 결정됩니다.

시스템이 설치한 UI 문화권 또는 사용자가 선호하는 UI 문화권과 다른 특정 UI 문화권을 설정하고 애플리케이션이 여러 스레드를 시작하는 경우 스레드가 실행되는 애플리케이션 도메인의 속성에 `GetUserDefaultUILanguage` 문화권을 할당하지 않는 한 해당 스레드의 현재 UI 문화권은 함수에서 반환되는 문화권 DefaultThreadCurrentUICulture 이 됩니다.

보안 고려 사항

현재 스레드의 문화권을 변경하려면 SecurityPermission 권한이 필요하며, ControlThread 값이 설정되어 있어야 합니다.

> **⊗ 주의**
>
> 스레드 조작은 스레드와 연결된 보안 상태 때문에 위험합니다. 따라서 이 권한은 신뢰할 수 있는 코드에만 부여된 다음 필요에 따라 지정해야 합니다. 반 신뢰할 수 있는 코드에서는 스레드 문화권을 변경할 수 없습니다.

현재의 UI 언어 설정과 UWP 앱

UWP(유니버설 Windows 플랫폼) 앱에서 CurrentUICulture 속성은 .NET Framework 및 .NET Core 앱과 마찬가지로 읽기/쓰기가 가능합니다. 현재 문화권을 가져와서 설정하는 데 둘 다 사용할 수 있습니다. 그러나 UWP 앱은 현재 문화권과 현재 UI 문화권을 구분하지 않습니다. CurrentCulture 속성 및 CurrentUICulture 속성은 Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages 컬렉션의 첫 번째 값에 매핑됩니다.

.NET Framework 및 .NET Core 앱에서 현재 UI 문화권은 스레드별 설정이며 CurrentUICulture 속성은 현재 스레드의 UI 문화권만 반영합니다. UWP 앱에서 현재 문화권은 글로벌 설정인 Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages 속성에 매핑됩니다. CurrentCulture 속성을 설정하면 전체 앱의 문화권이 변경됩니다. 문화권은 스레드별로 설정할 수 없습니다.

System.Globalization.CultureInfo.InvariantCulture 속성

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

불변 문화는 문화적 차이에 민감하지 않습니다. 이는 영어와 관련이 있지만, 특정 국가나 지역과는 연관이 없습니다. CultureInfo 인스턴스화 메서드 호출에서 빈 문자열("")을 사용하여 이름으로 고정 문화권을 지정합니다. CultureInfo.InvariantCulture이 속성은 고정 문화권의 인스턴스도 검색합니다. 문화권이 필요한 System.Globalization 네임스페이스의 거의 모든 메서드에서 사용할 수 있습니다. CompareInfo, DateTimeFormat및 NumberFormat 같은 속성에서 반환되는 개체는 고정 문화권의 문자열 비교 및 서식 규칙도 반영합니다.

사용자 사용자 지정 또는 .NET Framework 또는 운영 체제 업데이트에 의해 변경될 수 있는 문화권에 민감한 데이터와 달리 고정 문화권 데이터는 시간이 지남에 따라 설치된 문화권 간에 안정적이며 사용자가 사용자 지정할 수 없습니다. 불변 문화권은 서식이 지정된 데이터의 유지에 필요한 서식 지정 및 구문 분석 작업이나 문화권에 상관없이 데이터를 고정된 순서로 정렬하고 배열해야 하는 작업처럼 문화권 독립적인 결과가 필요한 작업에 특히 유용합니다.

문자열 작업

문화에 민감한 문자열 작업을 수행할 때 특정 문화권의 규칙에 영향을 받지 않고, 문화권 간에 일관성을 유지하는 고정 문화권을 사용할 수 있습니다. 예를 들어 정렬된 데이터를 고정된 순서로 표시하거나 현재 문화권에 관계없이 문자열에 대/소문자 규칙의 표준 집합을 적용할 수 있습니다. 이렇게 하려면 InvariantCulture 개체를 Compare(String, String, Boolean, CultureInfo) 및 ToUpper(CultureInfo)같은 CultureInfo 매개 변수가 있는 메서드에 전달합니다.

데이터 지속성

InvariantCulture 속성을 사용하여 문화권 독립적 형식으로 데이터를 유지할 수 있습니다. 이는 변경되지 않고 문화권 간에 데이터를 직렬화하고 역직렬화하는 데 사용할 수 있는 알려진 형식을 제공합니다. 데이터가 역직렬화되면 현재 사용자의 문화권 규칙에 따라 적절하게 서식을 지정할 수 있습니다.

예를 들어 문자열 형식으로 날짜 및 시간 데이터를 유지하도록 선택한 경우 InvariantCulture 개체를 DateTime.ToString(String, IFormatProvider) 또는

DateTimeOffset.ToString(IFormatProvider) 메서드에 전달하여 문자열을 만들 수 있으며 InvariantCulture 개체를 DateTime.Parse(String, IFormatProvider) 또는 DateTimeOffset.Parse(String, IFormatProvider, DateTimeStyles) 메서드에 전달하여 문자열을 날짜 및 시간 값으로 다시 변환할 수 있습니다. 이 기술을 사용하면 다른 문화권의 사용자가 데이터를 읽거나 쓸 때 기본 날짜 및 시간 값이 변경되지 않습니다.

다음 예제에서는 고정 문화를 사용하여 DateTime 값을 문자열로 저장합니다. 그런 다음 문자열을 구문 분석하고 프랑스(프랑스) 및 독일(독일) 문화권의 서식 규칙을 사용하여 해당 값을 표시합니다.

C#

```
using System;
using System.IO;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      // Persist the date and time data.
      StreamWriter sw = new StreamWriter(@".\DateData.dat");

      // Create a DateTime value.
      DateTime dtIn = DateTime.Now;
      // Retrieve a CultureInfo object.
      CultureInfo invC = CultureInfo.InvariantCulture;

      // Convert the date to a string and write it to a file.
      sw.WriteLine(dtIn.ToString("r", invC));
      sw.Close();

      // Restore the date and time data.
      StreamReader sr = new StreamReader(@".\DateData.dat");
      String input;
      while ((input = sr.ReadLine()) != null)
      {
         Console.WriteLine($"Stored data: {input}\n");

         // Parse the stored string.
         DateTime dtOut = DateTime.Parse(input, invC,
DateTimeStyles.RoundtripKind);

         // Create a French (France) CultureInfo object.
         CultureInfo frFr = new CultureInfo("fr-FR");
         // Displays the date formatted for the "fr-FR" culture.
         Console.WriteLine($"Date formatted for the {frFr.Name} culture:
{dtOut.ToString("f", frFr)}");

         // Creates a German (Germany) CultureInfo object.
         CultureInfo deDe= new CultureInfo("de-De");
         // Displays the date formatted for the "de-DE" culture.
```

```
            Console.WriteLine($"Date formatted for {deDe.Name} culture:
{dtOut.ToString("f", deDe)}");
         }
         sr.Close();
      }
}
// The example displays the following output:
//    Stored data: Tue, 15 May 2012 16:34:16 GMT
//
//    Date formatted for the fr-FR culture: mardi 15 mai 2012 16:34
//    Date formatted for de-DE culture: Dienstag, 15. Mai 2012 16:34
```

보안 결정

문자열 비교 또는 대/소문자 변경의 결과에 따라 보안 결정(예: 시스템 리소스에 대한 액세스를 허용할지 여부)을 결정하는 경우 고정 문화권을 사용하면 안 됩니다. 대신 StringComparison 매개 변수를 포함하는 메서드를 호출하고 인수로 StringComparison.Ordinal 또는 StringComparison.OrdinalIgnoreCase를 사용하여 대소문자 구분 여부에 따른 서수 비교를 수행해야 합니다. 문화권 구분 문자열 작업을 수행하는 코드는 현재 문화권이 변경되거나 코드를 실행하는 컴퓨터의 문화권이 코드를 테스트하는 데 사용되는 문화권과 다른 경우 보안 취약성을 일으킬 수 있습니다. 반면 서수 비교는 비교된 문자의 이진 값에만 의존합니다.

DateTimeFormatInfo 클래스

아티클 • 2023. 12. 30.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

클래스의 DateTimeFormatInfo 속성에는 다음과 같은 날짜 및 시간 값의 서식을 지정하거나 구문 분석하기 위한 문화권별 정보가 포함됩니다.

- 날짜 값의 서식을 지정하는 데 사용되는 패턴입니다.
- 시간 값의 서식을 지정하는 데 사용되는 패턴입니다.
- 요일의 이름입니다.
- 해당 연도의 월 이름입니다.
- 시간 값에 사용되는 오전 및 오후 지정자입니다.
- 날짜가 표현되는 달력입니다.

DateTimeFormatInfo 개체 인스턴스화

개체는 DateTimeFormatInfo 고정 문화권, 특정 문화권, 중립 문화권 또는 현재 문화권의 서식 규칙을 나타낼 수 있습니다. 이 섹션에서는 각 개체 형식 DateTimeFormatInfo 을 인스턴스화하는 방법을 설명합니다.

고정 문화권에 대한 DateTimeFormatInfo 개체 인스턴스화

고정 문화권은 문화권을 구분하지 않는 문화권을 나타냅니다. 그것은 영어에 따라, 하지만 어떤 특정 영어권 국가/지역에. 특정 문화권의 데이터는 동적일 수 있으며 새로운 문화권 규칙 또는 사용자 기본 설정을 반영하도록 변경할 수 있지만 고정 문화권의 데이터는 변경되지 않습니다. 다음과 같은 방법으로 고정 문화권의 서식 규칙을 나타내는 개체를 인스턴스화 DateTimeFormatInfo 할 수 있습니다.

- 속성 값을 InvariantInfo 검색합니다. 반환된 DateTimeFormatInfo 개체가 읽기 전용입니다.
- 매개 변수가 없는 DateTimeFormatInfo 생성자를 호출합니다. 반환 DateTimeFormatInfo 된 개체가 읽기/쓰기입니다.
- 속성에서 반환되는 개체에서 속성 값을 DateTimeFormat 검색합니다 CultureInfo.InvariantCulture.CultureInfo 반환된 DateTimeFormatInfo 개체가 읽기 전용입니다.

다음 예제에서는 이러한 각 메서드를 사용하여 고정 문화권을 나타내는 개체를 인스턴스화 DateTimeFormatInfo 합니다. 그런 다음 개체가 읽기 전용인지 여부를 나타냅니다.

```
C#
```

```
System.Globalization.DateTimeFormatInfo dtfi;

dtfi = System.Globalization.DateTimeFormatInfo.InvariantInfo;
Console.WriteLine(dtfi.IsReadOnly);

dtfi = new System.Globalization.DateTimeFormatInfo();
Console.WriteLine(dtfi.IsReadOnly);

dtfi = System.Globalization.CultureInfo.InvariantCulture.DateTimeFormat;
Console.WriteLine(dtfi.IsReadOnly);
// The example displays the following output:
//       True
//       False
//       True
```

특정 문화권에 대한 DateTimeFormatInfo 개체 인스턴스화

특정 문화권은 특정 국가/지역에서 사용되는 언어를 나타냅니다. 예를 들어 en-US는 미국 사용되는 영어를 나타내는 특정 문화권이며 en-CA는 캐나다에서 사용되는 영어를 나타내는 특정 문화권입니다. 다음과 같은 방법으로 특정 문화권의 서식 규칙을 나타내는 개체를 인스턴스화 DateTimeFormatInfo 할 수 있습니다.

- 메서드를 CultureInfo.GetCultureInfo(String) 호출하고 반환 CultureInfo 된 개체 CultureInfo.DateTimeFormat 의 속성 값을 검색합니다. 반환된 DateTimeFormatInfo 개체가 읽기 전용입니다.
- 정적 GetInstance 메서드 CultureInfo 를 전달하여 검색할 개체의 문화권을 DateTimeFormatInfo 나타내는 개체입니다. 반환 DateTimeFormatInfo 된 개체가 읽기/쓰기입니다.
- 정적 CultureInfo.CreateSpecificCulture 메서드를 호출하고 반환 CultureInfo 된 개체 CultureInfo.DateTimeFormat 의 속성 값을 검색합니다. 반환 DateTimeFormatInfo 된 개체가 읽기/쓰기입니다.
- 클래스 생성자를 호출 CultureInfo.CultureInfo 하고 반환 CultureInfo 된 개체 CultureInfo.DateTimeFormat 의 속성 값을 검색합니다. 반환 DateTimeFormatInfo 된 개체가 읽기/쓰기입니다.

다음 예제에서는 개체를 인스턴스화하는 DateTimeFormatInfo 이러한 각 방법을 보여 줍니다. 결과 개체가 읽기 전용인지 여부를 나타냅니다.

C#

```
System.Globalization.CultureInfo ci = null;
System.Globalization.DateTimeFormatInfo dtfi = null;
```

```
// Instantiate a culture using CreateSpecificCulture.
ci = System.Globalization.CultureInfo.CreateSpecificCulture("en-US");
dtfi = ci.DateTimeFormat;
Console.WriteLine("{0} from CreateSpecificCulture: {1}", ci.Name,
dtfi.IsReadOnly);

// Instantiate a culture using the CultureInfo constructor.
ci = new System.Globalization.CultureInfo("en-CA");
dtfi = ci.DateTimeFormat;
Console.WriteLine("{0} from CultureInfo constructor: {1}", ci.Name,
dtfi.IsReadOnly);

// Retrieve a culture by calling the GetCultureInfo method.
ci = System.Globalization.CultureInfo.GetCultureInfo("en-AU");
dtfi = ci.DateTimeFormat;
Console.WriteLine("{0} from GetCultureInfo: {1}", ci.Name, dtfi.IsReadOnly);

// Instantiate a DateTimeFormatInfo object by calling
DateTimeFormatInfo.GetInstance.
ci = System.Globalization.CultureInfo.CreateSpecificCulture("en-GB");
dtfi = System.Globalization.DateTimeFormatInfo.GetInstance(ci);
Console.WriteLine("{0} from GetInstance: {1}", ci.Name, dtfi.IsReadOnly);

// The example displays the following output:
//       en-US from CreateSpecificCulture: False
//       en-CA from CultureInfo constructor: False
//       en-AU from GetCultureInfo: True
//       en-GB from GetInstance: False
```

중립 문화권에 대한 DateTimeFormatInfo 개체 인스턴스화

중립 문화권은 국가/지역과 독립적인 문화권 또는 언어를 나타냅니다. 일반적으로 하나 이상의 특정 문화권의 부모입니다. 예를 들어 Fr은 프랑스어와 fr-FR 문화권의 부모에 대한 중립 문화권입니다. 특정 문화권의 서식 규칙을 나타내는 개체를 만드는 DateTimeFormatInfo 것과 동일한 방식으로 중립 문화권의 서식 규칙을 나타내는 개체를 인스턴스화 DateTimeFormatInfo 할 수 있습니다. 또한 특정 문화권의 DateTimeFormatInfo 속성에서 중립 문화권을 검색하고 해당 속성에서 반환된 개체를 검색하여 중립 문화권 DateTimeFormatInfo 의 CultureInfo.Parent 개체를 검색할 CultureInfo.DateTimeFormat 수 있습니다. 부모 문화권이 고정 문화권을 나타내지 않는 한 반환 DateTimeFormatInfo 된 개체는 읽기/쓰기가 가능합니다. 다음 예제에서는 중립 문화권을 나타내는 개체를 DateTimeFormatInfo 인스턴스화하는 이러한 방법을 보여 줍니다.

C#

```
System.Globalization.CultureInfo specific, neutral;
System.Globalization.DateTimeFormatInfo dtfi;
```

```
// Instantiate a culture by creating a specific culture and using its Parent
property.
specific = System.Globalization.CultureInfo.GetCultureInfo("fr-FR");
neutral = specific.Parent;
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from Parent property: {1}", neutral.Name,
dtfi.IsReadOnly);

dtfi = System.Globalization.CultureInfo.GetCultureInfo("fr-
FR").Parent.DateTimeFormat;
Console.WriteLine("{0} from Parent property: {1}", neutral.Name,
dtfi.IsReadOnly);

// Instantiate a neutral culture using the CultureInfo constructor.
neutral = new System.Globalization.CultureInfo("fr");
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from CultureInfo constructor: {1}", neutral.Name,
dtfi.IsReadOnly);

// Instantiate a culture using CreateSpecificCulture.
neutral = System.Globalization.CultureInfo.CreateSpecificCulture("fr");
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from CreateSpecificCulture: {1}", neutral.Name,
dtfi.IsReadOnly);

// Retrieve a culture by calling the GetCultureInfo method.
neutral = System.Globalization.CultureInfo.GetCultureInfo("fr");
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from GetCultureInfo: {1}", neutral.Name,
dtfi.IsReadOnly);

// Instantiate a DateTimeFormatInfo object by calling GetInstance.
neutral = System.Globalization.CultureInfo.CreateSpecificCulture("fr");
dtfi = System.Globalization.DateTimeFormatInfo.GetInstance(neutral);
Console.WriteLine("{0} from GetInstance: {1}", neutral.Name,
dtfi.IsReadOnly);

// The example displays the following output:
//       fr from Parent property: False
//       fr from Parent property: False
//       fr from CultureInfo constructor: False
//       fr-FR from CreateSpecificCulture: False
//       fr from GetCultureInfo: True
//       fr-FR from GetInstance: False
```

그러나 중립 문화권은 특정 국가/지역과 독립적이기 때문에 문화권별 서식 지정 정보가 부족합니다. .NET은 개체를 DateTimeFormatInfo 제네릭 값으로 채우는 대신 중립 문화권의 자식인 특정 문화권의 서식 규칙을 반영하는 개체를 반환 DateTimeFormatInfo 합니다. 예를 들어 DateTimeFormatInfo 중립 en 문화권의 개체는 en-US 문화권의 서식 규칙을 반영하고 DateTimeFormatInfo fr 문화권의 개체는 fr-FR 문화권의 서식 규칙을 반영합니다.

다음과 같은 코드를 사용하여 중립 문화권이 나타내는 특정 문화권의 서식 지정 규칙을 결정할 수 있습니다. 이 예제에서는 리플렉션을 사용하여 중립 문화권의 속성을 특정 자식 문화권의 속성과 비교 DateTimeFormatInfo 합니다. 두 달력이 동일한 달력 유형인 경우 동일한 것으로 간주하고, 양력의 경우 속성에 동일한 값이 있는 경우 GregorianCalendar.CalendarType 동일한 것으로 간주합니다.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.Reflection;

public class InstantiateEx6
{
    public static void Main()
    {
        // Get all the neutral cultures
        List<String> names = new List<String>();
        Array.ForEach(CultureInfo.GetCultures(CultureTypes.NeutralCultures),
                      culture => names.Add(culture.Name));
        names.Sort();
        foreach (var name in names)
        {
            // Ignore the invariant culture.
            if (name == "") continue;

            ListSimilarChildCultures(name);
        }
    }

    private static void ListSimilarChildCultures(String name)
    {
        // Create the neutral DateTimeFormatInfo object.
        DateTimeFormatInfo dtfi =
CultureInfo.GetCultureInfo(name).DateTimeFormat;
        // Retrieve all specific cultures of the neutral culture.
        CultureInfo[] cultures =
Array.FindAll(CultureInfo.GetCultures(CultureTypes.SpecificCultures),
                                  culture => culture.Name.StartsWith(name +
"-", StringComparison.OrdinalIgnoreCase));
        // Create an array of DateTimeFormatInfo properties
        PropertyInfo[] properties =
typeof(DateTimeFormatInfo).GetProperties(BindingFlags.Instance |
BindingFlags.Public);
        bool hasOneMatch = false;

        foreach (var ci in cultures)
        {
            bool match = true;
            // Get the DateTimeFormatInfo for a specific culture.
            DateTimeFormatInfo specificDtfi = ci.DateTimeFormat;
```

```
            // Compare the property values of the two.
            foreach (var prop in properties)
            {
                // We're not interested in the value of IsReadOnly.
                if (prop.Name == "IsReadOnly") continue;

                // For arrays, iterate the individual elements to see if
they are the same.
                if (prop.PropertyType.IsArray)
                {
                    IList nList = (IList)prop.GetValue(dtfi, null);
                    IList sList = (IList)prop.GetValue(specificDtfi, null);
                    if (nList.Count != sList.Count)
                    {
                        match = false;
                        Console.WriteLine("   Different n in {2} array for
{0} and {1}", name, ci.Name, prop.Name);
                        break;
                    }

                    for (int ctr = 0; ctr < nList.Count; ctr++)
                    {
                        if (!nList[ctr].Equals(sList[ctr]))
                        {
                            match = false;
                            Console.WriteLine("   {0} value different for
{1} and {2}", prop.Name, name, ci.Name);
                            break;
                        }
                    }

                    if (!match) break;
                }
                // Get non-array values.
                else
                {
                    Object specificValue = prop.GetValue(specificDtfi);
                    Object neutralValue = prop.GetValue(dtfi);

                    // Handle comparison of Calendar objects.
                    if (prop.Name == "Calendar")
                    {
                        // The cultures have a different calendar type.
                        if (specificValue.ToString() !=
neutralValue.ToString())
                        {
                            Console.WriteLine("   Different calendar types
for {0} and {1}", name, ci.Name);
                            match = false;
                            break;
                        }

                        if (specificValue is GregorianCalendar)
                        {
                            if
```

```
(((GregorianCalendar)specificValue).CalendarType !=
((GregorianCalendar)neutralValue).CalendarType)
                            {
                                Console.WriteLine("   Different Gregorian
calendar types for {0} and {1}", name, ci.Name);
                                match = false;
                                break;
                            }
                        }
                    }
                    else if (!specificValue.Equals(neutralValue))
                    {
                        match = false;
                        Console.WriteLine("   Different {0} values for {1}
and {2}", prop.Name, name, ci.Name);
                        break;
                    }
                }
            }
            if (match)
            {
                Console.WriteLine("DateTimeFormatInfo object for '{0}'
matches '{1}'",
                                  name, ci.Name);
                hasOneMatch = true;
            }
        }
        if (!hasOneMatch)
            Console.WriteLine("DateTimeFormatInfo object for '{0}' --> No
Match", name);

        Console.WriteLine();
    }
}
```

현재 문화권에 대한 DateTimeFormatInfo 개체 인스턴스화

다음과 같은 방법으로 현재 문화권의 서식 규칙을 나타내는 개체를 인스턴스화 DateTimeFormatInfo 할 수 있습니다.

- 속성 값을 CurrentInfo 검색합니다. 반환된 DateTimeFormatInfo 개체가 읽기 전용입니다.
- 속성에서 반환되는 개체에서 속성 값을 DateTimeFormat 검색합니다 CultureInfo.CurrentCulture.CultureInfo 반환된 DateTimeFormatInfo 개체가 읽기 전용입니다.
- 현재 문화권을 GetInstance 나타내는 개체를 사용하여 CultureInfo 메서드를 호출합니다. 반환된 DateTimeFormatInfo 개체가 읽기 전용입니다.

다음 예제에서는 이러한 각 메서드를 사용하여 현재 문화권의 서식 규칙을 나타내는 개체를 인스턴스화 DateTimeFormatInfo 합니다. 그런 다음 개체가 읽기 전용인지 여부를 나타냅니다.

C#

```
DateTimeFormatInfo dtfi;

dtfi = DateTimeFormatInfo.CurrentInfo;
Console.WriteLine(dtfi.IsReadOnly);

dtfi = CultureInfo.CurrentCulture.DateTimeFormat;
Console.WriteLine(dtfi.IsReadOnly);

dtfi = DateTimeFormatInfo.GetInstance(CultureInfo.CurrentCulture);
Console.WriteLine(dtfi.IsReadOnly);
// The example displays the following output:
//     True
//     True
//     True
```

다음 방법 중 하나로 현재 문화권의 규칙을 나타내는 쓰기 가능한 DateTimeFormatInfo 개체를 만들 수 있습니다.

- 이전의 세 가지 방법 중에서 DateTimeFormatInfo 개체를 검색하고 반환 DateTimeFormatInfo 된 개체에서 Clone 메서드를 호출합니다. 이렇게 하면 원래 개체의 복사본이 DateTimeFormatInfo 만들어지지만 해당 IsReadOnly 속성은 다음과 같습니다 `false`.

- 메서드를 CultureInfo.CreateSpecificCulture 호출하여 현재 문화권을 나타내는 개체를 만든 CultureInfo 다음 해당 CultureInfo.DateTimeFormat 속성을 사용하여 개체를 검색합니다 DateTimeFormatInfo .

다음 예제에서는 읽기/쓰기 DateTimeFormatInfo 개체를 인스턴스화하는 각 방법을 보여 줍니다. 해당 IsReadOnly 속성의 값을 표시합니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx1
{
    public static void Main()
    {
        DateTimeFormatInfo current1 = DateTimeFormatInfo.CurrentInfo;
        current1 = (DateTimeFormatInfo)current1.Clone();
        Console.WriteLine(current1.IsReadOnly);
```

```
        CultureInfo culture2 =
CultureInfo.CreateSpecificCulture(CultureInfo.CurrentCulture.Name);
        DateTimeFormatInfo current2 = culture2.DateTimeFormat;
        Console.WriteLine(current2.IsReadOnly);
    }
}
// The example displays the following output:
//       False
//       False
```

에서는 Windows에서 사용자를 재정의할 수 중 일부는 DateTimeFormatInfo 서식 지정 및 구문 분석을 통해 작업에 사용 된 속성 값을 **국가 및 언어** 제어판 애플리케이션입니다. 예를 들어 문화권이 영어(미국)인 사용자는 기본 12시간 클록(h:mm:ss tt 형식) 대신 24시간 클록(HH:mm:ss 형식)을 사용하여 장시간 값을 표시하도록 선택할 수 있습니다. 앞에서 설명한 방식으로 검색된 개체는 DateTimeFormatInfo 모두 이러한 사용자 재정의를 반영합니다. 바람직하지 않은 경우 생성자를 호출 CultureInfo.CultureInfo(String, Boolean) 하고 인수에 대한 `useUserOverride` 값을 `false` 제공하여 사용자 재정의를 반영하지 않고 읽기 전용이 아닌 읽기/쓰기인 개체를 만들 NumberFormatInfo 수 있습니다. 다음 예제에서는 현재 문화권이 영어(미국)이고 긴 시간 패턴이 h:mm:ss tt의 기본값에서 HH:mm:ss로 변경된 시스템에 대해 이를 보여 줍니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx3
{
    public static void Main()
    {
        CultureInfo culture;
        DateTimeFormatInfo dtfi;

        culture = CultureInfo.CurrentCulture;
        dtfi = culture.DateTimeFormat;
        Console.WriteLine("Culture Name:      {0}", culture.Name);
        Console.WriteLine("User Overrides:    {0}",
culture.UseUserOverride);
        Console.WriteLine("Long Time Pattern: {0}\n",
culture.DateTimeFormat.LongTimePattern);

        culture = new CultureInfo(CultureInfo.CurrentCulture.Name, false);
        Console.WriteLine("Culture Name:      {0}", culture.Name);
        Console.WriteLine("User Overrides:    {0}",
culture.UseUserOverride);
        Console.WriteLine("Long Time Pattern: {0}\n",
culture.DateTimeFormat.LongTimePattern);
    }
}
// The example displays the following output:
```

```
//       Culture Name:      en-US
//       User Overrides:    True
//       Long Time Pattern: HH:mm:ss
//
//       Culture Name:      en-US
//       User Overrides:    False
//       Long Time Pattern: h:mm:ss tt
```

DateTimeFormatInfo 및 동적 데이터

클래스에서 제공하는 DateTimeFormatInfo 날짜 및 시간 값의 서식을 지정하기 위한 문화권별 데이터는 클래스에서 제공하는 CultureInfo 문화권 데이터와 마찬가지로 동적입니다. 특정 CultureInfo 개체와 연결된 개체에 대한 값의 안정성을 DateTimeFormatInfo 가정해서는 안 됩니다. 고정 문화권 및 관련 DateTimeFormatInfo 개체에서 제공하는 데이터만 안정적입니다. 다른 데이터는 애플리케이션 세션 간에 또는 애플리케이션을 실행 하는 동안에 변경할 수 있습니다. 다음과 같은 네 가지 주요 변경 요소가 있습니다.

- 시스템 업데이트 기본 달력 또는 사용자 지정 날짜 및 시간 형식과 같은 문화권 기본 설정은 시간에 따라 변경됩니다. 이 경우 Windows 업데이트 특정 문화권의 DateTimeFormatInfo 속성 값에 대한 변경 내용을 포함합니다.

- 대체 문화권. 클래스를 CultureAndRegionInfoBuilder 사용하여 기존 문화권의 데이터를 바꿀 수 있습니다.

- 속성 값에 대한 연속 변경 내용입니다. 여러 문화권 관련 속성은 런타임에 변경되어 데이터가 변경됩니다 DateTimeFormatInfo . 예를 들어 현재 문화권은 프로그래밍 방식으로 또는 사용자 작업을 통해 변경할 수 있습니다. 이 경우 속성에서 DateTimeFormatInfo 반환된 개체가 CurrentInfo 현재 문화권과 연결된 개체로 변경됩니다. 마찬가지로 문화권의 달력이 변경될 수 있으므로 다양한 DateTimeFormatInfo 속성 값이 변경될 수 있습니다.

- 사용자 기본 설정. 애플리케이션 사용자는 제어판에서 국가 및 언어 옵션을 통해 현재 시스템 문화권과 연관 된 값의 일부를 재정의할 수도 있습니다. 예를 들어 사용자는 날짜를 다른 형식으로 표시하도록 선택할 수 있습니다. 속성이 CultureInfo.UseUserOverride 설정된 `true` 경우 개체의 DateTimeFormatInfo 속성도 사용자 설정에서 검색됩니다. 사용자 설정이 개체와 연결된 문화권과 CultureInfo 호환되지 않는 경우(예: 선택한 달력이 속성에 OptionalCalendars 표시된 달력 중 하나가 아닌 경우) 메서드의 결과와 속성 값이 정의되지 않습니다.

일치하지 않는 데이터의 가능성을 최소화하기 위해 개체를 만들 때 개체의 모든 사용자 재정의 DateTimeFormatInfo 가능한 속성이 초기화됩니다. 개체 만들기나 사용자 재정의 프로세스가 원자성이 없고 개체를 만드는 동안 관련 값이 변경 될 수 있으므로 여전히 불일치가 발생할 수 있습니다. 그러나 이 상황은 극히 드물어야 합니다.

사용자 재정의가 시스템 문화권과 동일한 문화권을 나타내는 개체에 DateTimeFormatInfo 반영되는지 여부를 제어할 수 있습니다. 다음 표에서는 개체를 DateTimeFormatInfo 검색할 수 있는 방법을 나열하고 결과 개체가 사용자 재정의를 반영하는지 여부를 나타냅니다.

테이블 확장

CultureInfo 및 DateTimeFormatInfo 개체의 원본	사용자 재정의 반영
`CultureInfo.CurrentCulture.DateTimeFormat` 속성	예
DateTimeFormatInfo.CurrentInfo 속성	예
CultureInfo.CreateSpecificCulture 메서드	예
CultureInfo.GetCultureInfo 메서드	아니요
CultureInfo.CultureInfo(String) 생성자	예
CultureInfo.CultureInfo(String, Boolean) 생성자	매개 변수 값 `useUserOverride` 에 따라 다름

사용 하는 경우 사용자 재정의가 없는 중요 한 이유가 없는 고려해 야 합니다 DateTimeFormatInfo 서식을 지정 하 고 사용자 입력을 구문 분석 하거나 데이터를 표시 하도록 클라이언트 애플리케이션에서 개체입니다. 서버 애플리케이션 또는 무인된 애플리케이션에 대 한 없습니다 다음을 수행 해야합니다. 그러나 개체를 DateTimeFormatInfo 명시적으로 또는 암시적으로 사용하여 날짜 및 시간 데이터를 문자열 형식으로 유지하는 경우 고정 문화권의 서식 규칙을 반영하는 개체를 사용 DateTimeFormatInfo 하거나 문화권에 관계없이 사용하는 사용자 지정 날짜 및 시간 서식 문자열을 지정해야 합니다.

날짜 및 시간 서식 지정

DateTimeFormatInfo 개체는 모든 날짜 및 시간 서식 지정 작업에서 암시적 또는 명시적으로 사용됩니다. 여기에는 다음 메서드에 대한 호출이 포함됩니다.

- 모든 날짜 및 시간 서식 지정 메서드(예: DateTime.ToString() 및 DateTimeOffset.ToString(String).
- 주 복합 서식 지정 메서드입니다 String.Format.
- 기타 복합 서식 지정 메서드(예: Console.WriteLine(String, Object[]) 및 StringBuilder.AppendFormat(String, Object[]).

모든 날짜 및 시간 서식 지정 작업은 구현을 IFormatProvider 사용합니다. 인터페이스에는 IFormatProvider 단일 메서드가 IFormatProvider.GetFormat(Type)포함됩니다. 이 콜백 메서드는 서식 정보를 제공하는 데 필요한 형식을 나타내는 개체로 전달 Type 됩니다. 메서드는 해당 형식의 인스턴스를 반환하거나 `null` 형식의 인스턴스를 제공할 수 없는 경

우 반환합니다. .NET에는 날짜 및 시간을 서식 지정하기 위한 두 가지 IFormatProvider 구현이 포함되어 있습니다.

- CultureInfo 특정 문화권(또는 특정 국가/지역의 특정 언어)을 나타내는 클래스입니다. 날짜 및 시간 서식 지정 작업에서 메서드는 CultureInfo.GetFormat 해당 속성과 연결된 개체를 반환 DateTimeFormatInfo 합니다 CultureInfo.DateTimeFormat .
- DateTimeFormatInfo 연결된 문화권의 서식 규칙에 대한 정보를 제공하는 클래스입니다. 메서드는 DateTimeFormatInfo.GetFormat 자체 인스턴스를 반환합니다.

형식 지정 메서드에 IFormatProvider 구현이 명시적으로 CultureInfo 제공되지 않으면 현재 문화권을 나타내는 속성에서 CultureInfo.CurrentCulture 반환된 개체가 사용됩니다.

다음 예제에서는 형식 지정 작업에서 인터페이스와 DateTimeFormatInfo 클래스 간의 IFormatProvider 관계를 보여 줍니다. 서식 지정 작업에서 요청한 개체의 형식을 표시하는 사용자 지정 IFormatProvider 구현 GetFormat 을 정의합니다. 개체를 DateTimeFormatInfo 요청하는 경우 메서드는 현재 문화권에 DateTimeFormatInfo 대한 개체를 제공합니다. 예제의 출력에서 알 Decimal.ToString(IFormatProvider) 수 있듯이 메서드는 개체를 요청 DateTimeFormatInfo 하여 형식 지정 정보를 제공하는 반면 String.Format(IFormatProvider, String, Object[]) 메서드는 구현뿐만 아니라 요청 NumberFormatInfo 및 DateTimeFormatInfo 개체를 ICustomFormatter 제공합니다.

C#

```
using System;
using System.Globalization;

public class CurrentCultureFormatProvider : IFormatProvider
{
    public Object GetFormat(Type formatType)
    {
        Console.WriteLine("Requesting an object of type {0}",
                          formatType.Name);
        if (formatType == typeof(NumberFormatInfo))
            return NumberFormatInfo.CurrentInfo;
        else if (formatType == typeof(DateTimeFormatInfo))
            return DateTimeFormatInfo.CurrentInfo;
        else
            return null;
    }
}

public class FormatProviderEx1
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 5, 28, 13, 30, 0);
        string value = dateValue.ToString("F", new
CurrentCultureFormatProvider());
        Console.WriteLine(value);
```

```
        Console.WriteLine();
        string composite = String.Format(new CurrentCultureFormatProvider(),
                                         "Date: {0:d}   Amount: {1:C}
Description: {2}",
                                         dateValue, 1264.03m, "Service
Charge");
        Console.WriteLine(composite);
        Console.WriteLine();
    }
}
// The example displays output like the following:
//       Requesting an object of type DateTimeFormatInfo
//       Tuesday, May 28, 2013 1:30:00 PM
//
//       Requesting an object of type ICustomFormatter
//       Requesting an object of type DateTimeFormatInfo
//       Requesting an object of type NumberFormatInfo
//       Date: 5/28/2013   Amount: $1,264.03   Description: Service Charge
```

문자열 및 DateTimeFormatInfo 속성 서식 지정

이 개체에는 DateTimeFormatInfo 날짜 및 시간 값을 사용하여 서식 지정 작업에 사용되는 세 가지 종류의 속성이 포함됩니다.

- 일정 관련 속성입니다. 와 MonthNamesAbbreviatedMonthNamesDayNames같은 AbbreviatedDayNames속성은 속성에 의해 Calendar 정의된 문화권에서 사용하는 달력과 연결됩니다. 이러한 속성은 긴 날짜 및 시간 형식에 사용됩니다.

- 표준 정의 결과 문자열을 생성하는 속성입니다. 및 SortableDateTimePatternUniversalSortableDateTimePattern 속성에는 RFC1123Pattern국제 표준에 의해 정의된 결과 문자열을 생성하는 사용자 지정 형식 문자열이 포함됩니다. 이러한 속성은 읽기 전용이며 수정할 수 없습니다.

- 문화권 구분 결과 문자열을 정의하는 속성입니다. 일부 속성(예: FullDateTimePattern 및ShortDatePattern)에는 결과 문자열의 형식을 지정하는 사용자 지정 형식 문자열이 포함되어 있습니다. AMDesignatorTimeSeparatorPMDesignatorDateSeparator결과 문자열에 포함할 수 있는 문화권 구분 기호 또는 부분 문자열을 정의합니다.

표준 날짜 및 시간 형식 문자열(예: "d", "D", "f" 및 "F")은 특정 DateTimeFormatInfo 형식 패턴 속성에 해당하는 별칭입니다. 대부분의 사용자 지정 날짜 및 시간 형식 문자열은 서식 지정 작업이 결과 스트림에 삽입하는 문자열 또는 부분 문자열과 관련이 있습니다. 다음 표에서는 표준 및 사용자 지정 날짜 및 시간 형식 지정자와 관련 DateTimeFormatInfo 속성을 나열합니다. 이러한 형식 지정자를 사용하는 방법에 대한 자세한 내용은 표준 날짜 및 시간 형식 문자열 및 사용자 지정 날짜 및 시간 형식 문자열을 참조하세요. 각 표준 서식 문자열은 DateTimeFormatInfo 값이 사용자 지정 날짜 및 시간 형식 문자열인 속성

에 해당합니다. 이 사용자 지정 형식 문자열의 개별 지정자는 다른 DateTimeFormatInfo 속성에 해당합니다. 표에는 표준 서식 문자열이 별칭인 속성만 DateTimeFormatInfo 나열되며, 별칭이 지정된 속성에 할당된 사용자 지정 형식 문자열에서 액세스할 수 있는 속성은 나열되지 않습니다. 또한 테이블에는 속성에 해당하는 사용자 지정 형식 지정자만 나열됩니다 DateTimeFormatInfo .

테이블 확장

형식 지정자	연결된 속성
"d"(짧은 날짜, 표준 형식 문자열)	ShortDatePattern- 결과 문자열의 전체 형식을 정의합니다.
"D"(긴 날짜, 표준 형식 문자열)	LongDatePattern- 결과 문자열의 전체 형식을 정의합니다.
"f"(전체 날짜/짧은 시간, 표준 형식 문자열)	LongDatePattern- 결과 문자열의 날짜 구성 요소 형식을 정의합니다. ShortTimePattern- 결과 문자열의 시간 구성 요소 형식을 정의합니다.
"F"(전체 날짜/긴 시간, 표준 형식 문자열)	LongDatePattern- 결과 문자열의 날짜 구성 요소 형식을 정의합니다. LongTimePattern- 결과 문자열의 시간 구성 요소 형식을 정의합니다.
"g"(일반 날짜/짧은 시간, 표준 형식 문자열)	ShortDatePattern- 결과 문자열의 날짜 구성 요소 형식을 정의합니다. ShortTimePattern- 결과 문자열의 시간 구성 요소 형식을 정의합니다.
"G"(일반 날짜/긴 시간, 표준 형식 문자열)	ShortDatePattern- 결과 문자열의 날짜 구성 요소 형식을 정의합니다. LongTimePattern- 결과 문자열의 시간 구성 요소 형식을 정의합니다.
"M", "m"(월/일, 표준 형식 문자열)	MonthDayPattern- 결과 문자열의 전체 형식을 정의합니다.
"O", "o"(왕복 날짜/시간, 표준 형식 문자열)	없음.
"R", "r"(RFC1123; 표준 형식 문자열)	RFC1123Pattern- RFC 1123 표준을 준수하는 결과 문자열을 정의합니다. 속성이 읽기 전용입니다.
"s"(정렬 가능한 날짜/시간, 표준 형식 문자열)	SortableDateTimePatternISO 8601 표준을 준수하는 결과 문자열을 정의합니다. 속성이 읽기 전용입니다.
"t"(짧은 시간, 표준 형식 문자열)	ShortTimePattern- 결과 문자열의 전체 형식을 정의합니다.
"T"(long time; standard format string)	LongTimePattern- 결과 문자열의 전체 형식을 정의합니다.

형식 지정자	연결된 속성
"u"(범용 정렬 가능 날짜/시간, 표준 형식 문자열)	UniversalSortableDateTimePattern는 조정된 범용 시간에 대한 ISO 8601 표준을 준수하는 결과 문자열을 정의합니다. 속성이 읽기 전용입니다.
"U"(범용 전체 날짜/시간, 표준 형식 문자열)	FullDateTimePattern- 결과 문자열의 전체 형식을 정의합니다.
"Y", "y"(연도 월, 표준 형식 문자열)	YearMonthPattern- 결과 문자열의 전체 형식을 정의합니다.
"ddd"(사용자 지정 형식 지정자)	AbbreviatedDayNames- 결과 문자열에 요일의 축약된 이름을 포함합니다.
"g", "gg"(사용자 지정 형식 지정자)	메서드를 GetEraName 호출하여 결과 문자열에 연대 이름을 삽입합니다.
"MMM"(사용자 지정 형식 지정자)	AbbreviatedMonthNames- 결과 문자열에 축약된 월 이름을 포함합니다.
"MMMM"(사용자 지정 형식 지정자)	MonthNames 또는 MonthGenitiveNames결과 문자열에 전체 월 이름을 포함할 수 있습니다.
"t"(사용자 지정 형식 지정자)	AMDesignator 또는 PMDesignator결과 문자열에 AM/PM 지정자의 첫 번째 문자를 포함합니다.
"tt"(사용자 지정 형식 지정자)	AMDesignator 또는 PMDesignator결과 문자열에 전체 AM/PM 지정자를 포함합니다.
":" (사용자 지정 형식 지정자)	TimeSeparator- 결과 문자열에 시간 구분 기호를 포함합니다.
"/"(사용자 지정 형식 지정자)	DateSeparator- 결과 문자열에 날짜 구분 기호를 포함합니다.

DateTimeFormatInfo 속성 수정

쓰기 가능한 개체의 연결된 속성을 수정하여 날짜 및 시간 형식 문자열에 의해 생성된 결과 문자열을 변경할 수 있습니다 DateTimeFormatInfo . 개체가 DateTimeFormatInfo 쓰기 가능한지 확인하려면 속성을 사용합니다 IsReadOnly . 이러한 방식으로 개체를 사용자 지정하려면 다음을 DateTimeFormatInfo 수행합니다.

1. 서식 규칙을 수정하려는 개체의 DateTimeFormatInfo 읽기/쓰기 복사본을 만듭니다.
2. 원하는 결과 문자열을 생성하는 데 사용되는 속성 또는 속성을 수정합니다. (서식 지정 메서드가 속성을 사용하여 DateTimeFormatInfo 결과 문자열을 정의하는 방법에

대한 자세한 내용은 이전 섹션인 형식 문자열 및 DateTimeFormatInfo 속성을 참조하세요.)

3. 만든 사용자 지정 DateTimeFormatInfo 개체를 IFormatProvider 형식 지정 메서드 호출에서 인수로 사용합니다.

결과 문자열의 형식을 변경하는 다른 두 가지 방법이 있습니다.

- 클래스를 CultureAndRegionInfoBuilder 사용하여 사용자 지정 문화권(고유한 이름을 가지며 기존 문화권을 보완하는 문화권) 또는 대체 문화권(특정 문화권 대신 사용되는 문화권)을 정의할 수 있습니다. .NET에서 지원하는 모든 CultureInfo 개체와 마찬가지로 프로그래밍 방식으로 이 문화권을 저장하고 액세스할 수 있습니다.
- 결과 문자열이 문화권을 구분하지 않고 미리 정의된 형식을 따르지 않는 경우 사용자 지정 날짜 및 시간 형식 문자열을 사용할 수 있습니다. 예를 들어 날짜 및 시간 데이터를 YYYYMMDDHHmmss 형식으로 serialize하는 경우 사용자 지정 형식 문자열을 메서드에 전달하여 결과 문자열 DateTime.ToString(String) 을 생성할 수 있으며, 메서드를 호출 DateTime.ParseExact 하여 결과 문자열을 값으로 다시 변환할 DateTime 수 있습니다.

짧은 날짜 패턴 변경

다음 예제에서는 "d"(짧은 날짜) 표준 형식 문자열에 의해 생성된 결과 문자열의 형식을 변경합니다. en-US 또는 영어(미국) 문화권의 연결된 ShortDatePattern 속성을 기본값인 "M/d/yyyy"에서 "yyyy'-"MM"-"dd"로 변경하고 "d" 표준 형식 문자열을 사용하여 속성이 변경되기 전과 후에 ShortDatePattern 날짜를 표시합니다.

C#

```
using System;
using System.Globalization;

public class Example1
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 8, 18);
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;

        Console.WriteLine("Before modifying DateTimeFormatInfo object: ");
        Console.WriteLine("{0}: {1}\n", dtfi.ShortDatePattern,
                                        dateValue.ToString("d", enUS));

        // Modify the short date pattern.
        dtfi.ShortDatePattern = "yyyy-MM-dd";
        Console.WriteLine("After modifying DateTimeFormatInfo object: ");
```

```
        Console.WriteLine("{0}: {1}", dtfi.ShortDatePattern,
                                       dateValue.ToString("d", enUS));
    }
}
// The example displays the following output:
//       Before modifying DateTimeFormatInfo object:
//       M/d/yyyy: 8/18/2013
//
//       After modifying DateTimeFormatInfo object:
//       yyyy-MM-dd: 2013-08-18
```

날짜 구분 기호 문자 변경

다음은 fr-FR 문화권의 서식 규칙을 나타내는 개체의 날짜 구분 기호 문자를 DateTimeFormatInfo 변경하는 예제입니다. 이 예제에서는 "g" 표준 형식 문자열을 사용하여 속성이 변경되기 전과 후에 DateSeparator 날짜를 표시합니다.

C#

```
using System;
using System.Globalization;

public class Example3
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 08, 28);
        CultureInfo frFR = CultureInfo.CreateSpecificCulture("fr-FR");
        DateTimeFormatInfo dtfi = frFR.DateTimeFormat;

        Console.WriteLine("Before modifying DateSeparator property: {0}",
                          dateValue.ToString("g", frFR));

        // Modify the date separator.
        dtfi.DateSeparator = "-";
        Console.WriteLine("After modifying the DateSeparator property: {0}",
                          dateValue.ToString("g", frFR));
    }
}
// The example displays the following output:
//       Before modifying DateSeparator property: 28/08/2013 00:00
//       After modifying the DateSeparator property: 28-08-2013 00:00
```

날짜 이름 약어 및 긴 날짜 패턴 변경

경우에 따라 일반적으로 전체 날짜 및 월 이름을 월 및 연도의 일 수와 함께 표시하는 긴 날짜 패턴이 너무 길 수 있습니다. 다음 예제에서는 en-US 문화권의 긴 날짜 패턴을 줄여 1자 또는 2자 일 이름 약어와 일 번호, 월 이름 약어 및 연도를 반환합니다. 이 작업은 배열

에 더 짧은 일 이름 약어를 AbbreviatedDayNames 할당하고 속성에 할당된 LongDatePattern 사용자 지정 형식 문자열을 수정하여 수행합니다. 이는 "D" 및 "f" 표준 형식 문자열에서 반환된 결과 문자열에 영향을 줍니다.

C#

```
using System;
using System.Globalization;

public class Example2
{
    public static void Main()
    {
        DateTime value = new DateTime(2013, 7, 9);
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;
        String[] formats = { "D", "F", "f" };

        // Display date before modifying properties.
        foreach (var fmt in formats)
            Console.WriteLine("{0}: {1}", fmt, value.ToString(fmt, dtfi));

        Console.WriteLine();

        // We don't want to change the FullDateTimePattern, so we need to
save it.
        String originalFullDateTimePattern = dtfi.FullDateTimePattern;

        // Modify day name abbreviations and long date pattern.
        dtfi.AbbreviatedDayNames = new String[] { "Su", "M", "Tu", "W",
"Th", "F", "Sa" };
        dtfi.LongDatePattern = "ddd dd-MMM-yyyy";
        dtfi.FullDateTimePattern = originalFullDateTimePattern;
        foreach (var fmt in formats)
            Console.WriteLine("{0}: {1}", fmt, value.ToString(fmt, dtfi));
    }
}
// The example displays the following output:
//       D: Tuesday, July 9, 2013
//       F: Tuesday, July 9, 2013 12:00:00 AM
//       f: Tuesday, July 9, 2013 12:00 AM
//
//       D: Tu 09-Jul-2013
//       F: Tuesday, July 9, 2013 12:00:00 AM
//       f: Tu 09-Jul-2013 12:00 AM
```

일반적으로 속성 변경 LongDatePattern 은 속성에도 영향을 FullDateTimePattern 줍니다. 이 속성은 "F" 표준 형식 문자열에서 반환된 결과 문자열을 정의합니다. 원래 전체 날짜 및 시간 패턴을 유지하기 위해 예제에서는 속성이 수정된 후 속성에 FullDateTimePattern 할당된 원래 사용자 지정 형식 문자열을 LongDatePattern 다시 할당합니다.

12시간 시계에서 24시간 시계로 변경

.NET의 많은 문화권에서 시간은 12시간 시계와 AM/PM 지정자를 사용하여 표현됩니다. 다음 예제에서는 12시간 시계를 사용하는 모든 시간 형식을 24시간 클록을 사용하는 형식으로 바꾸는 메서드를 정의 `ReplaceWith24HourClock` 합니다.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example5
{
    public static void Main()
    {
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;

        Console.WriteLine("Original Property Values:");
        Console.WriteLine("ShortTimePattern: " + dtfi.ShortTimePattern);
        Console.WriteLine("LongTimePattern: " + dtfi.LongTimePattern);
        Console.WriteLine("FullDateTimePattern: " +
dtfi.FullDateTimePattern);
        Console.WriteLine();

        dtfi.LongTimePattern = ReplaceWith24HourClock(dtfi.LongTimePattern);
        dtfi.ShortTimePattern =
ReplaceWith24HourClock(dtfi.ShortTimePattern);

        Console.WriteLine("Modififed Property Values:");
        Console.WriteLine("ShortTimePattern: " + dtfi.ShortTimePattern);
        Console.WriteLine("LongTimePattern: " + dtfi.LongTimePattern);
        Console.WriteLine("FullDateTimePattern: " +
dtfi.FullDateTimePattern);
    }

    private static string ReplaceWith24HourClock(string fmt)
    {
        string pattern = @"^(?<openAMPM>\s*t+\s*)? " +
                         @"(?(openAMPM) h+(?<nonHours>[^ht]+)$ " +
                         @"| \s*h+(?<nonHours>[^ht]+)\s*t+)";
        return Regex.Replace(fmt, pattern, "HH${nonHours}",
                             RegexOptions.IgnorePatternWhitespace);
    }
}
// The example displays the following output:
//       Original Property Values:
//       ShortTimePattern: h:mm tt
//       LongTimePattern: h:mm:ss tt
//       FullDateTimePattern: dddd, MMMM dd, yyyy h:mm:ss tt
//
//       Modififed Property Values:
```

```
//       ShortTimePattern: HH:mm
//       LongTimePattern: HH:mm:ss
//       FullDateTimePattern: dddd, MMMM dd, yyyy HH:mm:ss
```

이 예제에서는 정규식을 사용하여 형식 문자열을 수정합니다. 정규식 패턴 `@"^(?<openAMPM>\s*t+\s*)? (?(openAMPM) h+(?<nonHours>[^ht]+)$ | \s*h+(?<nonHours>[^ht]+)\s*t+)` 은 다음과 같이 정의됩니다.

테이블 확장

패턴	설명
`^`	문자열의 시작 부분에서 검색을 시작합니다.
`(?<openAMPM>\s*t+\s*)?`	공백 문자가 0개 또는 1개 이상인 다음 문자 "t"를 한 번 이상 찾은 다음 0개 이상의 공백 문자를 찾습니다. 이 캡처링 그룹의 이름은 `openAMPM` 다음과 같습니다.
`(?(openAMPM) h+(?<nonHours>[^ht]+)$`	`openAMPM` 그룹에 일치하는 문자가 있는 경우 문자 "h"를 한 번 이상 일치시킨 다음 " h" 또는 "t"가 없는 하나 이상의 문자를 잇습니다. 일치 항목은 문자열의 끝에 끝납니다. "h" 후에 캡처된 모든 문자는 캡처 `nonHours` 링 그룹에 포함됩니다.
`| \s*h+(?<nonHours>[^ht]+)\s*t+)`	`openAMPM` 그룹에 일치하는 문자가 없으면 문자 "h"를 한 번 이상 일치시킨 다음 " h" 또는 "t"가 아닌 하나 이상의 문자와 0개 이상의 공백 문자를 찾습니다. 마지막으로 하나 이상의 문자 "t"를 일치시킬 수 있습니다. "h" 뒤와 공백 앞에 캡처된 모든 문자와 "t"는 명명 `nonHours`된 캡처링 그룹에 포함됩니다.

캡처링 그룹에는 `nonHours` 사용자 지정 날짜 및 시간 형식 문자열의 분 및 두 번째 구성 요소와 시간 구분 기호가 포함됩니다. 대체 패턴 `HH${nonHours}` 은 이러한 요소에 부분 문자열 "HH"를 추가합니다.

날짜에 연대 표시 및 변경

다음 예제에서는 en-US 문화권의 서식 규칙을 나타내는 개체의 속성에 "g" 사용자 지정 서식 지정자를 LongDatePattern 추가합니다. 이 추가는 다음 세 가지 표준 형식 문자열에 영향을 줍니다.

- 속성에 직접 LongDatePattern 매핑되는 "D"(긴 날짜) 표준 형식 문자열입니다.
- "f"(전체 날짜/짧은 시간) 표준 형식 문자열로, 및 속성에 의해 생성된 부분 문자열을 연결하는 결과 문자열을 LongDatePatternShortTimePattern 생성합니다.

- 속성에 직접 FullDateTimePattern 매핑되는 "F"(전체 날짜/긴 시간) 표준 서식 문자열입니다. 이 속성 값을 명시적으로 설정하지 않았기 때문에 속성과 LongTimePattern 속성을 연결 LongDatePattern 하여 동적으로 생성됩니다.

이 예제에서는 달력에 단일 연대가 있는 문화권의 연대 이름을 변경하는 방법도 보여 줍니다. 이 경우 en-US 문화권은 개체로 표현되는 그레고리오력(Gregorian Calendar)을 GregorianCalendar 사용합니다. 이 클래스는 GregorianCalendar A.D.(안노 도미니)라는 이름의 단일 시대를 지원합니다. 이 예제에서는 속성에 할당된 FullDateTimePattern 형식 문자열의 "g" 사용자 지정 형식 지정자를 리터럴 문자열로 바꿔 연대 이름을 C.E.(Common Era)로 변경합니다. 연대 이름은 일반적으로 .NET 또는 운영 체제에서 제공하는 문화권 테이블의 프라이빗 데이터에서 메서드에 의해 GetEraName 반환되므로 리터럴 문자열을 사용해야 합니다.

C#

```
using System;
using System.Globalization;

public class Example4
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 5, 18, 13, 30, 0);
        String[] formats = { "D", "f", "F" };

        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;
        String originalLongDatePattern = dtfi.LongDatePattern;

        // Display the default form of three long date formats.
        foreach (var fmt in formats)
            Console.WriteLine(dateValue.ToString(fmt, dtfi));

        Console.WriteLine();

        // Modify the long date pattern.
        dtfi.LongDatePattern = originalLongDatePattern + " g";
        foreach (var fmt in formats)
            Console.WriteLine(dateValue.ToString(fmt, dtfi));

        Console.WriteLine();

        // Change A.D. to C.E. (for Common Era)
        dtfi.LongDatePattern = originalLongDatePattern + @" 'C.E.'";
        foreach (var fmt in formats)
            Console.WriteLine(dateValue.ToString(fmt, dtfi));
    }
}
// The example displays the following output:
//       Saturday, May 18, 2013
//       Saturday, May 18, 2013 1:30 PM
```

```
//       Saturday, May 18, 2013 1:30:00 PM
//
//       Saturday, May 18, 2013 A.D.
//       Saturday, May 18, 2013 A.D. 1:30 PM
//       Saturday, May 18, 2013 A.D. 1:30:00 PM
//
//       Saturday, May 18, 2013 C.E.
//       Saturday, May 18, 2013 C.E. 1:30 PM
//       Saturday, May 18, 2013 C.E. 1:30:00 PM
```

날짜 및 시간 문자열 구문 분석

구문 분석에서는 날짜 및 시간의 문자열 표현을 값으로 DateTimeDateTimeOffset 변환합니다. 이러한 형식에는 구문 분석 작업을 지원하는 메서드, 및 `TryParseExact` 메서드가 모두 포함`ParseExact` `Parse` `TryParse`됩니다. 및 메서드는 `Parse` 다양한 형식을 가질 수 있는 문자열을 변환하는 `TryParseExact` 반면 `ParseExact` 문자열에는 정의된 형식 또는 형식이 있어야 `TryParse` 합니다. 구문 분석 작업이 실패 `Parse` 하고 `ParseExact` 예외를 throw하는 반면 `TryParse` `TryParseExact` 에 .`false`

구문 분석 메서드는 암시적 또는 명시적으로 열거형 값을 사용하여 DateTimeStyles 구문 분석할 문자열에 있을 수 있는 스타일 요소(예: 선행, 후행 또는 내부 공백)와 구문 분석된 문자열 또는 누락된 요소를 해석하는 방법을 결정합니다. 또는 `TryParse` 메서드를 DateTimeStyles 호출 `Parse` 할 때 값을 제공하지 않으면 기본값은 DateTimeStyles.AllowWhiteSpaces, 플래그 및 DateTimeStyles.AllowInnerWhite 플래그를 포함하는 DateTimeStyles.AllowLeadingWhiteDateTimeStyles.AllowTrailingWhite복합 스타일입니다. 및 `TryParseExact` 메서드의 `ParseExact` 경우 기본값은 입력 문자열이 DateTimeStyles.None특정 사용자 지정 날짜 및 시간 형식 문자열에 정확하게 해당해야 합니다.

구문 분석 메서드는 구문 분석할 문자열에서 발생할 수 있는 특정 기호 및 패턴을 정의하는 개체를 암시적으로 또는 명시적으로 사용합니다 DateTimeFormatInfo . 개체 DateTimeFormatInfo 를 DateTimeFormatInfo 제공하지 않으면 현재 문화권의 개체가 기본적으로 사용됩니다. 날짜 및 시간 문자열 구문 분석에 대한 자세한 내용은 개별 구문 분석 메서드(예: DateTime.Parse, DateTime.TryParseDateTimeOffset.ParseExact및 DateTimeOffset.TryParseExact)를 참조하세요.

다음 예제에서는 구문 분석 날짜 및 시간 문자열의 문화권 구분 특성을 보여 줍니다. en-US, en-GB, fr-FR 및 fi-FI 문화권의 규칙을 사용하여 두 날짜 문자열을 구문 분석하려고 합니다. en-US 문화권에서 2014년 8월 18일로 해석되는 날짜는 18이 월 번호로 해석되기 때문에 다른 세 문화권에서 예외를 throw FormatException 합니다. 2015년 1월 2일은 en-US 문화에서 첫 번째 달의 둘째 날로 구문 분석되지만 다시 기본 문화권에서 두 번째 달의 첫째 날로 구문 분석됩니다.

C#

```
using System;
using System.Globalization;

public class ParseEx1
{
    public static void Main()
    {
        string[] dateStrings = { "08/18/2014", "01/02/2015" };
        string[] cultureNames = { "en-US", "en-GB", "fr-FR", "fi-FI" };

        foreach (var cultureName in cultureNames)
        {
            CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
            Console.WriteLine("Parsing strings using the {0} culture.",
                              culture.Name);
            foreach (var dateStr in dateStrings)
            {
                try
                {
                    Console.WriteLine(String.Format(culture,
                                      "   '{0}' --> {1:D}", dateStr,
                                      DateTime.Parse(dateStr, culture)));
                }
                catch (FormatException)
                {
                    Console.WriteLine("   Unable to parse '{0}'", dateStr);
                }
            }
        }
    }
}
// The example displays the following output:
//       Parsing strings using the en-US culture.
//          '08/18/2014' --> Monday, August 18, 2014
//          '01/02/2015' --> Friday, January 02, 2015
//       Parsing strings using the en-GB culture.
//          Unable to parse '08/18/2014'
//          '01/02/2015' --> 01 February 2015
//       Parsing strings using the fr-FR culture.
//          Unable to parse '08/18/2014'
//          '01/02/2015' --> dimanche 1 février 2015
//       Parsing strings using the fi-FI culture.
//          Unable to parse '08/18/2014'
//          '01/02/2015' --> 1. helmikuuta 2015
```

날짜 및 시간 문자열은 일반적으로 두 가지 이유로 구문 분석됩니다.

- 사용자 입력을 날짜 및 시간 값으로 변환합니다.

- 날짜 및 시간 값을 왕복하려면 즉, 이전에 문자열로 serialize된 날짜 및 시간 값을 역직렬화합니다.

다음 섹션에서는 이러한 두 작업에 대해 자세히 설명합니다.

사용자 문자열 구문 분석

사용자가 입력한 날짜 및 시간 문자열을 구문 분석할 때는 사용자가 만들었을 수 있는 사용자 지정을 포함하여 사용자의 문화권 설정을 반영하는 개체를 항상 인스턴스화 DateTimeFormatInfo 해야 합니다. 그렇지 않으면 날짜 및 시간 개체에 잘못된 값이 있을 수 있습니다. 사용자 문화권 사용자 지정을 반영하는 개체를 DateTimeFormatInfo 인스턴스화하는 방법에 대한 자세한 내용은 DateTimeFormatInfo 및 동적 데이터 섹션을 참조하세요.

다음 예제에서는 사용자 문화권 설정을 반영하는 구문 분석 작업과 그렇지 않은 구문 분석 작업의 차이점을 보여 줍니다. 이 경우 기본 시스템 문화권은 en-US이지만 사용자는 제어판, **지역 및 언어**를 사용하여 짧은 날짜 패턴을 기본값인 "M/d/yyyy"에서 "yy/MM/dd"로 변경했습니다. 사용자가 사용자 설정을 반영하는 문자열을 입력하고 사용자 설정(재정의)도 반영하는 개체에 의해 DateTimeFormatInfo 문자열을 구문 분석하면 구문 분석 작업이 올바른 결과를 반환합니다. 그러나 문자열이 표준 en-US culture 설정을 반영하는 개체에 의해 DateTimeFormatInfo 구문 분석되는 경우 구문 분석 메서드는 14를 연도의 마지막 두 자리가 아닌 월 수로 해석하기 때문에 예외를 throw FormatException 합니다.

C#

```
using System;
using System.Globalization;

public class ParseEx2
{
    public static void Main()
    {
        string inputDate = "14/05/10";

        CultureInfo[] cultures = { CultureInfo.GetCultureInfo("en-US"),
                                   CultureInfo.CreateSpecificCulture("en-US")
};

        foreach (var culture in cultures)
        {
            try
            {
                Console.WriteLine("{0} culture reflects user overrides:
{1}",
                                  culture.Name, culture.UseUserOverride);
                DateTime occasion = DateTime.Parse(inputDate, culture);
```

```
                Console.WriteLine("'{0}' --> {1}", inputDate,
                                  occasion.ToString("D",
CultureInfo.InvariantCulture));
            }
            catch (FormatException)
            {
                Console.WriteLine("Unable to parse '{0}'", inputDate);
            }
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       en-US culture reflects user overrides: False
//       Unable to parse '14/05/10'
//
//       en-US culture reflects user overrides: True
//       '14/05/10' --> Saturday, 10 May 2014
```

날짜 및 시간 데이터 직렬화 및 역직렬화

직렬화된 날짜 및 시간 데이터는 왕복할 것으로 예상됩니다. 즉, 직렬화되고 역직렬화된 모든 값은 동일해야 합니다. 날짜 및 시간 값이 시간에서 한 순간을 나타내는 경우 역직렬화된 값은 복원된 시스템의 문화권 또는 표준 시간대에 관계없이 동일한 시간을 나타내야 합니다. 왕복 날짜 및 시간 데이터를 성공적으로 사용하려면 속성에서 반환 InvariantInfo 하는 고정 문화권의 규칙을 사용하여 데이터를 생성하고 구문 분석해야 합니다. 서식 지정 및 구문 분석 작업은 기본 문화권의 규칙을 반영해서는 안 됩니다. 기본 문화권 설정을 사용하는 경우 데이터의 이식성이 엄격하게 제한됩니다. 문화권별 설정이 serialize된 스레드의 설정과 동일한 스레드에서만 역직렬화할 수 있습니다. 경우에 따라 동일한 시스템에서 데이터를 성공적으로 직렬화하고 역직렬화할 수 없다는 의미입니다.

날짜 및 시간 값의 시간 구성 요소가 중요한 경우 UTC로 변환하고 "o" 또는 "r" 표준 형식 문자열을 사용하여 serialize해야 합니다. 그런 다음 구문 분석 메서드를 호출하고 고정 문화권과 함께 적절한 형식 문자열을 인수로 `provider` 전달하여 시간 데이터를 복원할 수 있습니다.

다음 예제에서는 날짜 및 시간 값을 라운드트립하는 프로세스를 보여 줍니다. 미국 태평양 시간을 관찰하고 현재 문화권이 en-US인 시스템에서 날짜와 시간을 직렬화합니다.

C#

```
using System;
using System.Globalization;
using System.IO;

public class SerializeEx1
{
```

```
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\DateData.dat");
        // Define a date and time to serialize.
        DateTime originalDate = new DateTime(2014, 08, 18, 08, 16, 35);
        // Display information on the date and time.
        Console.WriteLine("Date to serialize: {0:F}", originalDate);
        Console.WriteLine("Current Culture:   {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Time Zone:         {0}",
                          TimeZoneInfo.Local.DisplayName);
        // Convert the date value to UTC.
        DateTime utcDate = originalDate.ToUniversalTime();
        // Serialize the UTC value.
        sw.Write(utcDate.ToString("o", DateTimeFormatInfo.InvariantInfo));
        sw.Close();
    }
}
// The example displays the following output:
//       Date to serialize: Monday, August 18, 2014 8:16:35 AM
//       Current Culture:   en-US
//       Time Zone:         (UTC-08:00) Pacific Time (US & Canada)
```

브뤼셀, 코펜하겐, 마드리드 및 파리 표준 시간대의 시스템에 대한 데이터를 역직렬화하며 현재 문화는 fr-FR입니다. 복원된 날짜는 원래 날짜보다 9시간 늦습니다. 이는 UTC보다 8시간 뒤에서 UTC보다 1시간 앞당기는 표준 시간대 조정을 반영합니다. 원래 날짜와 복원된 날짜는 모두 동일한 시간을 나타냅니다.

C#

```
using System;
using System.Globalization;
using System.IO;

public class SerializeEx2
{
    public static void Main()
    {
        // Open the file and retrieve the date string.
        StreamReader sr = new StreamReader(@".\DateData.dat");
        String dateValue = sr.ReadToEnd();

        // Parse the date.
        DateTime parsedDate = DateTime.ParseExact(dateValue, "o",
                              DateTimeFormatInfo.InvariantInfo);
        // Convert it to local time.
        DateTime restoredDate = parsedDate.ToLocalTime();
        // Display information on the date and time.
        Console.WriteLine("Deserialized date: {0:F}", restoredDate);
        Console.WriteLine("Current Culture:   {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Time Zone:         {0}",
```

```
                          TimeZoneInfo.Local.DisplayName);
    }
}
// The example displays the following output:
//    Deserialized date: lundi 18 août 2014 17:16:35
//    Current Culture:   fr-FR
//    Time Zone:         (UTC+01:00) Brussels, Copenhagen, Madrid, Paris
```

NumberFormatInfo 클래스

아티클 • 2025. 03. 26.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

NumberFormatInfo 클래스에는 숫자 값의 서식을 지정하고 구문 분석할 때 사용되는 문화권별 정보가 포함됩니다. 이 정보에는 통화 기호, 10진수 기호, 그룹 구분 기호 및 양수 및 음수 기호가 포함됩니다.

NumberFormatInfo 개체 인스턴스화

현재 문화권, 고정 문화권, 특정 문화권 또는 중립 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 인스턴스화할 수 있습니다.

현재 문화권에 대한 NumberFormatInfo 개체 인스턴스화

다음과 같은 방법으로 현재 문화권에 대한 NumberFormatInfo 개체를 인스턴스화할 수 있습니다. 각 경우에서 반환된 NumberFormatInfo 개체는 읽기 전용입니다.

- CultureInfo.CurrentCulture 속성에서 현재 문화권을 나타내는 CultureInfo 개체를 검색하고 CultureInfo.NumberFormat 속성에서 CultureInfo 개체를 검색합니다.
- `static`(Visual Basic에서는`Shared`) CurrentInfo 속성에서 반환된 NumberFormatInfo 개체를 검색합니다.
- 현재 문화권을 나타내는 CultureInfo 개체를 사용하여 GetInstance 메서드를 호출합니다.

다음 예제에서는 이러한 세 가지 방법을 사용하여 현재 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 만듭니다. 또한 각 개체가 읽기 전용임을 보여 주는 IsReadOnly 속성의 값을 검색합니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx1
{
    public static void Main()
    {
        NumberFormatInfo current1 = CultureInfo.CurrentCulture.NumberFormat;
        Console.WriteLine(current1.IsReadOnly);
```

```
        NumberFormatInfo current2 = NumberFormatInfo.CurrentInfo;
        Console.WriteLine(current2.IsReadOnly);

        NumberFormatInfo current3 =
NumberFormatInfo.GetInstance(CultureInfo.CurrentCulture);
        Console.WriteLine(current3.IsReadOnly);
    }
}
// The example displays the following output:
//       True
//       True
//       True
```

다음과 같은 방법으로 현재 문화권의 규칙을 나타내는 쓰기 가능한 NumberFormatInfo 개체를 만들 수 있습니다.

- 이전 코드 예제에 설명된 방식으로 NumberFormatInfo 개체를 검색하고 반환된 NumberFormatInfo 개체에서 Clone 메서드를 호출합니다. 이렇게 하면 IsReadOnly 속성이 `false`것을 제외하고 원래 NumberFormatInfo 개체의 복사본이 만들어집니다.

- CultureInfo.CreateSpecificCulture 메서드를 호출하여 현재 문화권을 나타내는 CultureInfo 개체를 만든 다음 CultureInfo.NumberFormat 속성을 사용하여 NumberFormatInfo 개체를 검색합니다.

다음 예제에서는 이러한 두 가지 방법으로 NumberFormatInfo 개체를 인스턴스화하고 해당 IsReadOnly 속성 값을 표시하여 개체가 읽기 전용이 아님을 보여 줍니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx2
{
    public static void Main()
    {
        NumberFormatInfo current1 = NumberFormatInfo.CurrentInfo;
        current1 = (NumberFormatInfo)current1.Clone();
        Console.WriteLine(current1.IsReadOnly);

        CultureInfo culture2 =
CultureInfo.CreateSpecificCulture(CultureInfo.CurrentCulture.Name);
        NumberFormatInfo current2 = culture2.NumberFormat;
        Console.WriteLine(current2.IsReadOnly);
    }
}
// The example displays the following output:
```

```
//       False
//       False
```

Windows 운영 체제를 사용하면 제어판의 **지역 및 언어** 항목을 통해 숫자 서식 지정 및 구문 분석 작업에 사용되는 NumberFormatInfo 속성 값 중 일부를 재정의할 수 있습니다. 예를 들어 문화권이 영어(미국)인 사용자는 기본값인 $1.1 대신 통화 값을 1.1 USD로 표시하도록 선택할 수 있습니다. 앞에서 설명한 방식으로 검색된 NumberFormatInfo 개체는 모두 이러한 사용자 재정의를 반영합니다. 바람직하지 않은 경우 CultureInfo.CultureInfo(String, Boolean) 생성자를 호출하고 `useUserOverride` 인수에 대한 `false` 값을 제공하여 사용자 재정의를 반영하지 않고 읽기 전용이 아닌 읽기/쓰기인 NumberFormatInfo 개체를 만들 수 있습니다. 다음 예제에서는 현재 문화권이 영어(미국)이고 통화 기호가 기본값인 $에서 USD로 변경된 시스템에 대한 그림을 제공합니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx3
{
    public static void Main()
    {
        CultureInfo culture;
        NumberFormatInfo nfi;

        culture = CultureInfo.CurrentCulture;
        nfi = culture.NumberFormat;
        Console.WriteLine($"Culture Name:    {culture.Name}");
        Console.WriteLine($"User Overrides:  {culture.UseUserOverride}");
        Console.WriteLine($"Currency Symbol:
{culture.NumberFormat.CurrencySymbol}\n");

        culture = new CultureInfo(CultureInfo.CurrentCulture.Name, false);
        Console.WriteLine($"Culture Name:    {culture.Name}");
        Console.WriteLine($"User Overrides:  {culture.UseUserOverride}");
        Console.WriteLine($"Currency Symbol:
{culture.NumberFormat.CurrencySymbol}");
    }
}
// The example displays the following output:
//       Culture Name:    en-US
//       User Overrides:  True
//       Currency Symbol: USD
//
//       Culture Name:    en-US
//       User Overrides:  False
//       Currency Symbol: $
```

CultureInfo.UseUserOverride 속성이 `true`설정되면 사용자 설정에서 속성 CultureInfo.DateTimeFormat, CultureInfo.NumberFormat및 CultureInfo.TextInfo 검색됩니다. 사용자 설정이 CultureInfo 개체와 연결된 문화권과 호환되지 않는 경우(예: 선택한 달력이 OptionalCalendars 속성에 나열된 달력 중 하나가 아닌 경우) 메서드의 결과와 속성 값이 정의되지 않습니다.

고정 문화권에 대한 NumberFormatInfo 개체 인스턴스화

"불변 문화는 문화적 차이에 영향을 받지 않는 문화를 나타냅니다." 그것은 영어를 기반으로하지만 특정 영어권 국가 / 지역에 없습니다. 특정 문화권의 데이터는 동적일 수 있으며 새로운 문화권 규칙 또는 사용자 기본 설정을 반영하도록 변경할 수 있지만 고정 문화권의 데이터는 변경되지 않습니다. 고정 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체는 결과 문자열이 문화권에 따라 달라지지 않아야 하는 서식 지정 작업에 사용할 수 있습니다.

다음과 같은 방법으로 고정 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 인스턴스화할 수 있습니다.

- InvariantInfo 속성의 값을 검색하여. 반환된 NumberFormatInfo 개체는 읽기 전용입니다.
- CultureInfo.InvariantCulture 속성에서 반환되는 CultureInfo 개체에서 CultureInfo.NumberFormat 속성의 값을 검색합니다. 반환된 NumberFormatInfo 개체는 읽기 전용입니다.
- 매개 변수가 없는 NumberFormatInfo 클래스 생성자를 호출합니다. 반환된 NumberFormatInfo 개체는 읽기/쓰기가 가능합니다.

다음 예제에서는 이러한 각 메서드를 사용하여 고정 문화권을 나타내는 NumberFormatInfo 개체를 인스턴스화합니다. 그런 다음 개체가 읽기 전용인지 여부를 나타냅니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx4
{
    public static void Main()
    {
        NumberFormatInfo nfi;

        nfi = System.Globalization.NumberFormatInfo.InvariantInfo;
        Console.WriteLine(nfi.IsReadOnly);
```

```
        nfi = CultureInfo.InvariantCulture.NumberFormat;
        Console.WriteLine(nfi.IsReadOnly);

        nfi = new NumberFormatInfo();
        Console.WriteLine(nfi.IsReadOnly);
    }
}
// The example displays the following output:
//       True
//       True
//       False
```

특정 문화권에 대한 NumberFormatInfo 개체 인스턴스화

특정 문화권은 특정 국가/지역에서 사용되는 언어를 나타냅니다. 예를 들어 en-US 미국에서 사용되는 영어를 나타내는 특정 문화권이며, en-CA 캐나다에서 사용되는 영어를 나타내는 특정 문화권입니다. 다음과 같은 방법으로 특정 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 인스턴스화할 수 있습니다.

- CultureInfo.GetCultureInfo(String) 메서드를 호출하고 반환된 CultureInfo 개체의 NumberFormat 속성 값을 검색합니다. 반환된 NumberFormatInfo 개체는 읽기 전용입니다.
- 문화권의 NumberFormatInfo 개체를 검색하려면 해당 문화권을 나타내는 CultureInfo 개체를 정적 GetInstance 메서드에 전달합니다. 반환된 NumberFormatInfo 개체는 읽기/쓰기가 가능합니다.
- CultureInfo.CreateSpecificCulture 메서드를 호출하고 반환된 CultureInfo 개체의 NumberFormat 속성 값을 검색합니다. 반환된 NumberFormatInfo 개체는 읽기/쓰기가 가능합니다.
- CultureInfo.CultureInfo 클래스 생성자 중 하나를 호출하고 반환된 CultureInfo 개체의 NumberFormat 속성 값을 검색합니다. 반환된 NumberFormatInfo 개체는 읽기/쓰기가 가능합니다.

다음 예제에서는 이러한 네 가지 방법을 사용하여 인도네시아(인도네시아) 문화권의 서식 규칙을 반영하는 NumberFormatInfo 개체를 만듭니다. 또한 각 개체가 읽기 전용인지 여부를 나타냅니다.

C#

```
using System;
using System.Globalization;

public class InstantiateEx5
{
```

```
    public static void Main()
    {
        CultureInfo culture;
        NumberFormatInfo nfi;

        nfi = CultureInfo.GetCultureInfo("id-ID").NumberFormat;
        Console.WriteLine($"Read-only: {nfi.IsReadOnly}");

        culture = new CultureInfo("id-ID");
        nfi = NumberFormatInfo.GetInstance(culture);
        Console.WriteLine($"Read-only: {nfi.IsReadOnly}");

        culture = CultureInfo.CreateSpecificCulture("id-ID");
        nfi = culture.NumberFormat;
        Console.WriteLine($"Read-only: {nfi.IsReadOnly}");

        culture = new CultureInfo("id-ID");
        nfi = culture.NumberFormat;
        Console.WriteLine($"Read-only: {nfi.IsReadOnly}");
    }
}
// The example displays the following output:
//       Read-only: True
//       Read-only: False
//       Read-only: False
//       Read-only: False
```

중립 문화권에 대한 NumberFormatInfo 개체 인스턴스화

중립 문화권은 국가/지역과 독립적인 문화권 또는 언어를 나타냅니다. 일반적으로 하나 이상의 특정 문화권의 부모입니다. 예를 들어, fr은 프랑스어를 위한 중립 문화권이며 fr-FR 문화권의 상위 문화입니다. 특정 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 만드는 것과 같은 방식으로 중립 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 만듭니다.

그러나 특정 국가/지역과 독립적이므로 중립 문화권에는 문화권별 서식 지정 정보가 부족합니다. .NET은 NumberFormatInfo 개체를 제네릭 값으로 채우는 대신 중립 문화권의 자식인 특정 문화권의 서식 규칙을 반영하는 NumberFormatInfo 개체를 반환합니다. 예를 들어 중립 en 문화권의 NumberFormatInfo 개체는 en-US 문화권의 서식 규칙을 반영하고 fr 문화권의 NumberFormatInfo 개체는 fr-FR 문화권의 서식 규칙을 반영합니다.

다음과 같은 코드를 사용하여 각 중립 문화권이 나타내는 특정 문화권의 서식 규칙을 결정할 수 있습니다.

C#

```
using System;
using System.Collections;
```

```
using System.Collections.Generic;
using System.Globalization;
using System.Reflection;

public class InstantiateEx6
{
    public static void Main()
    {
        // Get all the neutral cultures
        List<String> names = new List<String>();
        Array.ForEach(CultureInfo.GetCultures(CultureTypes.NeutralCultures),
                      culture => names.Add(culture.Name));
        names.Sort();
        foreach (var name in names)
        {
            // Ignore the invariant culture.
            if (name == "") continue;

            ListSimilarChildCultures(name);
        }
    }

    private static void ListSimilarChildCultures(string name)
    {
        // Create the neutral NumberFormatInfo object.
        NumberFormatInfo nfi =
CultureInfo.GetCultureInfo(name).NumberFormat;
        // Retrieve all specific cultures of the neutral culture.
        CultureInfo[] cultures =
Array.FindAll(CultureInfo.GetCultures(CultureTypes.SpecificCultures),
                                  culture => culture.Name.StartsWith(name +
"-", StringComparison.OrdinalIgnoreCase));
        // Create an array of NumberFormatInfo properties
        PropertyInfo[] properties =
typeof(NumberFormatInfo).GetProperties(BindingFlags.Instance |
BindingFlags.Public);
        bool hasOneMatch = false;

        foreach (var ci in cultures)
        {
            bool match = true;
            // Get the NumberFormatInfo for a specific culture.
            NumberFormatInfo specificNfi = ci.NumberFormat;
            // Compare the property values of the two.
            foreach (var prop in properties)
            {
                // We're not interested in the value of IsReadOnly.
                if (prop.Name == "IsReadOnly") continue;

                // For arrays, iterate the individual elements to see if
they are the same.
                if (prop.PropertyType.IsArray)
                {
                    IList nList = (IList)prop.GetValue(nfi, null);
                    IList sList = (IList)prop.GetValue(specificNfi, null);
```

```
                    if (nList.Count != sList.Count)
                    {
                        match = false;
                        break;
                    }

                    for (int ctr = 0; ctr < nList.Count; ctr++)
                    {
                        if (!nList[ctr].Equals(sList[ctr]))
                        {
                            match = false;
                            break;
                        }
                    }
                }
                else if
(!prop.GetValue(specificNfi).Equals(prop.GetValue(nfi)))
                {
                    match = false;
                    break;
                }
            }
            if (match)
            {
                Console.WriteLine($"NumberFormatInfo object for '{name}'
matches '{ci.Name}'");
                hasOneMatch = true;
            }
        }
        if (!hasOneMatch)
            Console.WriteLine($"NumberFormatInfo object for '{name}' --> No
Match");

        Console.WriteLine();
    }
}
```

동적 데이터

NumberFormatInfo 클래스에서 제공하는 숫자 값의 서식을 지정하기 위한 문화권별 데이터는 CultureInfo 클래스에서 제공하는 문화권 데이터와 마찬가지로 동적입니다. 특정 CultureInfo 개체와 연결된 NumberFormatInfo 개체에 대한 값의 안정성을 가정해서는 안 됩니다. 고정 문화권 및 관련 NumberFormatInfo 개체에서 제공하는 데이터만 안정적입니다. 다른 데이터는 다음과 같은 이유로 애플리케이션 세션 간 또는 단일 세션 내에서 변경 될 수 있습니다.

- **시스템 업데이트**. 통화 기호 또는 통화 형식과 같은 문화권 기본 설정은 시간이 지남에 따라 변경됩니다. 이 경우 Windows 업데이트에는 특정 문화권에 대한 NumberFormatInfo 속성 값의 변경 내용이 포함됩니다.

- **대체 문화권.** CultureAndRegionInfoBuilder 클래스를 사용하여 기존 문화권의 데이터를 바꿀 수 있습니다.
- **속성 값에 대한 연속 변경 내용입니다.** 여러 문화권 관련 속성은 런타임에 변경될 수 있으며, 결과적으로 NumberFormatInfo 데이터에도 영향을 미칩니다. 예를 들어 현재 문화권은 프로그래밍 방식으로 또는 사용자 작업을 통해 변경할 수 있습니다. 이 경우 CurrentInfo 속성에서 반환된 NumberFormatInfo 개체가 현재 문화권과 연결된 개체로 변경됩니다.
- **사용자 기본 설정.** 애플리케이션 사용자는 제어판의 지역 및 언어 옵션을 통해 현재 시스템 문화권과 연결된 일부 값을 재정의할 수 있습니다. 예를 들어 사용자는 다른 통화 기호 또는 다른 소수 구분 기호를 선택할 수 있습니다. CultureInfo.UseUserOverride 속성이 `true`(기본값)로 설정된 경우 NumberFormatInfo 개체의 속성도 사용자 설정에서 검색됩니다.

개체를 만들 때 NumberFormatInfo 개체의 모든 사용자 재정의 가능한 속성이 초기화됩니다. 개체 만들기나 사용자 재정의 프로세스가 원자성이 없고 개체를 만드는 동안 관련 값이 변경되기 때문에 여전히 불일치가 발생할 수 있습니다. 그러나 이러한 불일치는 극히 드물어야 합니다.

사용자가 덮어쓴 내용이 현재 문화권과 같은 문화권을 나타내는 NumberFormatInfo 개체에 반영되는지 여부를 제어할 수 있습니다. 다음 표에서는 NumberFormatInfo 개체를 검색할 수 있는 방법을 나열하고 결과 개체가 사용자 재정의를 반영하는지 여부를 나타냅니다.

테이블 확장

CultureInfo 및 NumberFormatInfo 개체의 원본	사용자의 우선 적용 반영
`CultureInfo.CurrentCulture.NumberFormat` 속성	예
NumberFormatInfo.CurrentInfo 속성	예
CultureInfo.CreateSpecificCulture 메서드	예
CultureInfo.GetCultureInfo 메서드	아니오
CultureInfo(String) 생성자	예
CultureInfo.CultureInfo(String, Boolean) 생성자	`useUserOverride` 매개 변수의 값에 따라 달라집니다.

그렇지 않은 경우 강력한 이유가 없는 한 클라이언트 애플리케이션에서 NumberFormatInfo 개체를 사용하여 사용자 입력의 형식을 지정하고 구문 분석하거나 숫자 데이터를 표시할 때 사용자 재정의를 준수해야 합니다. 서버 애플리케이션 또는 무인

애플리케이션의 경우 사용자 재정의를 존중해서는 안 됩니다. 그러나 NumberFormatInfo 개체를 명시적으로 또는 암시적으로 사용하여 숫자 데이터를 문자열 형식으로 유지하는 경우 고정 문화권의 서식 규칙을 반영하는 NumberFormatInfo 개체를 사용하거나 문화권에 관계없이 사용하는 사용자 지정 숫자 서식 문자열을 지정해야 합니다.

IFormatProvider, NumberFormatInfo 및 숫자 서식 지정

NumberFormatInfo 개체는 모든 숫자 서식 지정 작업에서 암시적 또는 명시적으로 사용됩니다. 여기에는 다음 메서드에 대한 호출이 포함됩니다.

- Int32.ToString, Double.ToString 및 Convert.ToString(Int32)같은 모든 숫자 서식 지정 메서드.
- 주 복합 서식 지정 메서드인 String.Format.
- Console.WriteLine(String, Object[]) 및 StringBuilder.AppendFormat(String, Object[]) 같은 기타 복합 서식 지정 메서드

모든 숫자 서식 지정 작업은 IFormatProvider 구현을 사용합니다. IFormatProvider 인터페이스에는 단일 메서드 GetFormat(Type)포함됩니다. 서식 정보를 제공하는 데 필요한 형식을 나타내는 Type 개체에 전달되는 콜백 메서드입니다. 메서드는 해당 형식의 인스턴스를 제공할 수 없는 경우 `null`를 반환하거나, 형식의 인스턴스를 반환하는 역할을 맡고 있습니다. .NET은 숫자 서식 지정을 위한 두 가지 IFormatProvider 구현을 제공합니다.

- 특정 문화권(또는 특정 국가/지역의 특정 언어)을 나타내는 CultureInfo 클래스입니다. 숫자 서식 지정 작업에서 CultureInfo.GetFormat 메서드는 해당 CultureInfo.NumberFormat 속성과 연결된 NumberFormatInfo 개체를 반환합니다.
- 연결된 문화권의 서식 규칙에 대한 정보를 제공하는 NumberFormatInfo 클래스입니다. NumberFormatInfo.GetFormat 메서드는 자체 인스턴스를 반환합니다.

서식 지정 메서드에 IFormatProvider 구현이 명시적으로 제공되지 않으면 현재 문화권을 나타내는 CultureInfo.CurrentCulture 속성에서 반환된 CultureInfo 개체가 사용됩니다.

다음 예제에서는 사용자 지정 IFormatProvider 구현을 정의하여 서식 지정 작업에서 IFormatProvider 인터페이스와 NumberFormatInfo 클래스 간의 관계를 보여 줍니다. 해당 GetFormat 메서드는 서식 지정 작업에서 요청한 개체의 형식 이름을 표시합니다. 인터페이스가 NumberFormatInfo 개체를 요청하는 경우 이 메서드는 현재 문화권에 대한 NumberFormatInfo 개체를 제공합니다. 예제의 출력에서 알 수 있듯이 Decimal.ToString(IFormatProvider) 메서드는 서식 정보를 제공하기 위해 NumberFormatInfo 개체를 요청하는 반면 String.Format(IFormatProvider, String,

Object[]) 메서드는 ICustomFormatter 구현뿐만 아니라 NumberFormatInfo 및 DateTimeFormatInfo 개체를 요청합니다.

C#

```
using System;
using System.Globalization;

public class CurrentCultureFormatProvider : IFormatProvider
{
    public Object GetFormat(Type formatType)
    {
        Console.WriteLine($"Requesting an object of type
{formatType.Name}");
        if (formatType == typeof(NumberFormatInfo))
            return NumberFormatInfo.CurrentInfo;
        else if (formatType == typeof(DateTimeFormatInfo))
            return DateTimeFormatInfo.CurrentInfo;
        else
            return null;
    }
}

public class FormatProviderEx
{
    public static void Main()
    {
        Decimal amount = 1203.541m;
        string value = amount.ToString("C2", new
CurrentCultureFormatProvider());
        Console.WriteLine(value);
        Console.WriteLine();
        string composite = String.Format(new CurrentCultureFormatProvider(),
                                         "Date: {0}   Amount: {1}
Description: {2}",
                                         DateTime.Now, 1264.03m, "Service
Charge");
        Console.WriteLine(composite);
        Console.WriteLine();
    }
}
// The example displays output like the following:
//    Requesting an object of type NumberFormatInfo
//    $1,203.54
//
//    Requesting an object of type ICustomFormatter
//    Requesting an object of type DateTimeFormatInfo
//    Requesting an object of type NumberFormatInfo
//    Date: 11/15/2012 2:00:01 PM   Amount: 1264.03   Description: Service
Charge
```

IFormatProvider 구현이 숫자 서식 지정 메서드 호출에서 명시적으로 제공되지 않으면 메서드는 현재 문화권에 해당하는 NumberFormatInfo 개체를 반환하는 `CultureInfo.CurrentCulture.GetFormat` 메서드를 호출합니다.

문자열 서식 및 NumberFormatInfo 속성

모든 서식 지정 작업은 표준 또는 사용자 지정 숫자 형식 문자열을 사용하여 숫자에서 결과 문자열을 생성합니다. 경우에 따라 다음 예제와 같이 형식 문자열을 사용하여 결과 문자열을 생성하는 것은 명시적입니다. 이 코드는 Decimal.ToString(IFormatProvider) 메서드를 호출하여 en-US 문화권의 서식 지정 규칙을 사용하여 Decimal 값을 다양한 문자열 표현으로 변환합니다.

C#

```
using System;
using System.Globalization;

public class PropertiesEx1
{
    public static void Main()
    {
        string[] formatStrings = { "C2", "E1", "F", "G3", "N",
                                   "#,##0.000", "0,000,000,000.0##" };
        CultureInfo culture = CultureInfo.CreateSpecificCulture("en-US");
        Decimal[] values = { 1345.6538m, 1921651.16m };

        foreach (var value in values)
        {
            foreach (var formatString in formatStrings)
            {
                string resultString = value.ToString(formatString, culture);
                Console.WriteLine("{0,-18} -->  {1}", formatString,
resultString);
            }
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       C2                 -->  $1,345.65
//       E1                 -->  1.3E+003
//       F                  -->  1345.65
//       G3                 -->  1.35E+03
//       N                  -->  1,345.65
//       #,##0.000          -->  1,345.654
//       0,000,000,000.0##  -->  0,000,001,345.654
//
//       C2                 -->  $1,921,651.16
//       E1                 -->  1.9E+006
//       F                  -->  1921651.16
```

```
//       G3                  -->  1.92E+06
//       N                   -->  1,921,651.16
//       #,##0.000           -->  1,921,651.160
//       0,000,000,000.0##   -->  0,001,921,651.16
```

다른 경우에는 형식 문자열을 사용하는 것이 암시적입니다. 예를 들어 기본 또는 매개 변수가 없는 Decimal.ToString() 메서드를 호출하는 다음 메서드에서 Decimal 인스턴스의 값은 일반("G") 형식 지정자와 현재 문화권의 규칙을 사용하여 형식이 지정됩니다. 이 경우 en-US 문화권입니다.

C#

```
using System;

public class PropertiesEx2
{
    public static void Main()
    {
        Decimal[] values = { 1345.6538m, 1921651.16m };

        foreach (var value in values)
        {
            string resultString = value.ToString();
            Console.WriteLine(resultString);
            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//       1345.6538
//
//       1921651.16
```

각 표준 숫자 형식 문자열은 하나 이상의 NumberFormatInfo 속성을 사용하여 결과 문자열에 사용되는 패턴 또는 기호를 결정합니다. 마찬가지로 , "0" 및 "#"을 제외한 각 사용자 지정 숫자 형식 지정자는 NumberFormatInfo 속성으로 정의된 결과 문자열에 기호를 삽입합니다. 다음 표에서는 표준 및 사용자 지정 숫자 형식 지정자와 연결된 NumberFormatInfo 속성을 나열합니다. 특정 문화권에 대한 결과 문자열의 모양을 변경하려면 Modify NumberFormatInfo 속성 섹션을 참조하세요. 이러한 형식 지정자의 사용에 대한 자세한 내용은 표준 숫자 서식 문자열 및 사용자 지정 숫자 서식 문자열참조하세요.

테이블 확장

서식 지정자	연결된 속성
"C" 또는 "c"(통화 형식 지정자)	CurrencyDecimalDigits소수 자릿수의 기본 수를 정의합니다. CurrencyDecimalSeparator10진수 구분 기호를 정의합니다. CurrencyGroupSeparator그룹 또는 천 단위 구분 기호를 정의합니다. CurrencyGroupSizes, 정수 계열 그룹의 크기를 정의합니다. CurrencyNegativePattern음수 통화 값의 패턴을 정의합니다. CurrencyPositivePattern양수 통화 값의 패턴을 정의합니다. CurrencySymbol통화 기호를 정의합니다. NegativeSign음수 기호를 정의합니다.
"D" 또는 "d"(10진수 형식 지정자)	NegativeSign음수 기호를 정의합니다.
"E" 또는 "e" (지수 또는 과학 형식 지정자)	NegativeSign매니티사 및 지수에 음수 기호를 정의합니다. NumberDecimalSeparator10진수 구분 기호를 정의합니다. PositiveSign지수에서 양수 기호를 정의합니다.
"F" 또는 "f"(고정 소수점 형식 지정자)	NegativeSign음수 기호를 정의합니다. NumberDecimalDigits소수 자릿수의 기본 수를 정의합니다. NumberDecimalSeparator10진수 구분 기호를 정의합니다.
"G" 또는 "g"(일반 형식 지정자)	NegativeSign음수 기호를 정의합니다. NumberDecimalSeparator10진수 구분 기호를 정의합니다. PositiveSign결과 문자열의 양수 기호를 지수 형식으로 정의합니다.
"N" 또는 "n"(숫자 형식 지정자)	NegativeSign음수 기호를 정의합니다. NumberDecimalDigits소수 자릿수의 기본 수를 정의합니다. NumberDecimalSeparator10진수 구분 기호를 정의합니다. NumberGroupSeparator그룹 구분 기호(천)를 정의합니다. NumberGroupSizes는 그룹에서 정수 자릿수의 수를 정의합니다.

서식 지정자	연결된 속성
	NumberNegativePattern음수 값의 형식을 정의합니다.
"P" 또는 "p"(백분율 형식 지정자)	NegativeSign음수 기호를 정의합니다. PercentDecimalDigits소수 자릿수의 기본 수를 정의합니다. PercentDecimalSeparator10진수 구분 기호를 정의합니다. PercentGroupSeparator그룹 구분 기호를 정의합니다. PercentGroupSizes는 그룹 내 정수 자릿수의 개수를 정의합니다. PercentNegativePattern은 음수 값에 대한 백분율 기호와 음수 기호의 위치를 정의하는 데 사용됩니다. PercentPositivePattern양수 값에 대한 백분율 기호의 배치를 정의합니다. PercentSymbol백분율 기호를 정의합니다.
"R" 또는 "r"(왕복 형식 지정자)	NegativeSign음수 기호를 정의합니다. NumberDecimalSeparator10진수 구분 기호를 정의합니다. PositiveSign지수에서 양수 기호를 정의합니다.
"X" 또는 "x"(16진수 형식 지정자)	없음.
"." (소수점 사용자 지정 형식 지정자)	NumberDecimalSeparator10진수 구분 기호를 정의합니다.
","(그룹 구분 기호 사용자 지정 형식 지정자)	NumberGroupSeparator그룹(천 단위) 구분 기호를 정의합니다.
"%" (퍼센트 자리 표시자 사용자 정의 형식 지정자)	PercentSymbol백분율 기호를 정의합니다.
"‰" (퍼밀레 자리 표시자 사용자 지정 형식 지정자)	PerMilleSymbol밀리 단위 기호를 정의합니다.
"E"(지수 표기법 사용자 지정 형식 지정자)	NegativeSign은 매니티사와 지수에서 음수 기호를 정의하기 위한 것입니다. PositiveSign지수에서 양수 기호를 정의합니다.

NumberFormatInfo 클래스에는 특정 문화권에서 사용하는 기본 10자리를 지정하는 NativeDigits 속성이 포함되어 있습니다. 그러나 이 속성은 서식 지정 작업에 사용되지 않

습니다. 결과 문자열에는 기본 라틴 숫자 0(U+0030)에서 9(U+0039)만 사용됩니다. 또한 `NaN`, `PositiveInfinity` 및 `NegativeInfinity` Single 및 Double 값의 경우 결과 문자열은 각각 NaNSymbol, PositiveInfinitySymbol 및 NegativeInfinitySymbol 속성으로 정의된 기호로만 구성됩니다.

NumberFormatInfo 속성 수정

NumberFormatInfo 개체의 속성을 수정하여 숫자 서식 지정 작업에서 생성된 결과 문자열을 사용자 지정할 수 있습니다. 이렇게 하려면 다음을 수행합니다.

1. 서식 규칙을 수정하려는 NumberFormatInfo 개체의 읽기/쓰기 복사본을 만듭니다. 자세한 내용은 NumberFormatInfo 개체를 인스턴스화 섹션을 참조하세요.
2. 원하는 결과 문자열을 생성하는 데 사용되는 속성 또는 속성을 수정합니다. 서식 지정 메서드가 NumberFormatInfo 속성을 사용하여 결과 문자열을 정의하는 방법에 대한 자세한 내용은 형식 문자열 및 NumberFormatInfo 속성 섹션을 참조하세요.
3. 서식 지정 메서드 호출에서 사용자 지정 NumberFormatInfo 개체를 IFormatProvider 인수로 사용합니다.

> **참고**
>
> 애플리케이션이 시작될 때마다 문화권의 속성 값을 동적으로 수정하는 대신 CultureAndRegionInfoBuilder 클래스를 사용하여 사용자 지정 문화권(고유한 이름을 가지며 기존 문화권을 보완하는 문화권) 또는 대체 문화권(특정 문화권 대신 사용되는 문화권)을 정의할 수 있습니다.

다음 섹션에서는 몇 가지 예를 제공합니다.

통화 기호 및 패턴 수정

다음 예제에서는 en-US 문화권의 서식 규칙을 나타내는 NumberFormatInfo 개체를 수정합니다. ISO-4217 통화 기호를 CurrencySymbol 속성에 할당하고 통화 기호와 공백 및 숫자 값으로 구성된 통화 값의 패턴을 정의합니다.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
```

```
    {
        // Retrieve a writable NumberFormatInfo object.
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        NumberFormatInfo nfi = enUS.NumberFormat;

        // Use the ISO currency symbol instead of the native currency
symbol.
        nfi.CurrencySymbol = (new RegionInfo(enUS.Name)).ISOCurrencySymbol;
        // Change the positive currency pattern to <code><space><value>.
        nfi.CurrencyPositivePattern = 2;
        // Change the negative currency pattern to <code><space><sign>
<value>.
        nfi.CurrencyNegativePattern = 12;

        // Produce the result strings by calling ToString.
        Decimal[] values = { 1065.23m, 19.89m, -.03m, -175902.32m };
        foreach (var value in values)
            Console.WriteLine(value.ToString("C", enUS));

        Console.WriteLine();

        // Produce the result strings by calling a composite formatting
method.
        foreach (var value in values)
            Console.WriteLine(String.Format(enUS, "{0:C}", value));
    }
}
// The example displays the following output:
//       USD 1,065.23
//       USD 19.89
//       USD -0.03
//       USD -175,902.32
//
//       USD 1,065.23
//       USD 19.89
//       USD -0.03
//       USD -175,902.32
```

국가 식별 번호 서식 지정

많은 국가 식별 번호는 숫자로만 구성되므로 NumberFormatInfo 개체의 속성을 수정하여 쉽게 서식을 지정할 수 있습니다. 예를 들어 미국의 사회 보장 번호는 다음과 같이 정렬된 9자리 숫자로 구성됩니다. `xxx-xx-xxxx`. 다음 예제에서는 사회 보장 번호가 정수 값으로 저장되고 적절하게 서식을 지정한다고 가정합니다.

C#

```
using System;
using System.Globalization;

public class CustomizeSSNEx
```

```
{
    public static void Main()
    {
        // Instantiate a read-only NumberFormatInfo object.
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        NumberFormatInfo nfi = enUS.NumberFormat;

        // Modify the relevant properties.
        nfi.NumberGroupSeparator = "-";
        nfi.NumberGroupSizes = new int[] { 3, 2, 4 };
        nfi.NumberDecimalDigits = 0;

        int[] ids = { 111223333, 999776666 };

        // Produce the result string by calling ToString.
        foreach (var id in ids)
            Console.WriteLine(id.ToString("N", enUS));

        Console.WriteLine();

        // Produce the result string using composite formatting.
        foreach (var id in ids)
            Console.WriteLine(String.Format(enUS, "{0:N}", id));
    }
}
// The example displays the following output:
//       1112-23-333
//       9997-76-666
//
//       1112-23-333
//       9997-76-666
```

숫자형 문자열의 구문 분석

구문 분석에서는 숫자의 문자열 표현을 숫자로 변환합니다. .NET의 각 숫자 형식에는 `Parse` 및 `TryParse`두 개의 오버로드된 구문 분석 메서드가 포함됩니다. `Parse` 메서드는 문자열을 숫자로 변환하고 변환에 실패하면 예외를 throw합니다. `TryParse` 메서드는 문자열을 숫자로 변환하고, 숫자를 `out` 인수에 할당하고, 변환이 성공했는지 여부를 나타내는 Boolean 값을 반환합니다.

구문 분석 메서드는 NumberStyles 열거형 값을 암시적으로 또는 명시적으로 사용하여 구문 분석 작업이 성공하는 경우 문자열에 있을 수 있는 스타일 요소(예: 그룹 구분 기호, 소수 구분 기호 또는 통화 기호)를 결정합니다. 메서드 호출에 NumberStyles 값이 제공되지 않은 경우 기본값은 Float 및 AllowThousands 플래그를 포함하는 NumberStyles 값입니다. 이 값은 구문 분석된 문자열에 그룹 기호, 소수 구분 기호, 음수 기호 및 공백 문자를 포함하거나 지수 표기법으로 숫자의 문자열 표현이 될 수 있음을 지정합니다.

구문 분석 메서드는 구문 분석할 문자열에서 발생할 수 있는 특정 기호 및 패턴을 정의하는 NumberFormatInfo 개체를 암시적으로 또는 명시적으로 사용합니다. NumberFormatInfo 개체가 제공되지 않으면 기본값은 현재 문화권에 대한 NumberFormatInfo. 구문 분석에 대한 자세한 내용은 Int16.Parse(String), Int32.Parse(String, NumberStyles), Int64.Parse(String, IFormatProvider), Decimal.Parse(String, NumberStyles, IFormatProvider), Double.TryParse(String, Double)및 BigInteger.TryParse(String, NumberStyles, IFormatProvider, BigInteger)같은 개별 구문 분석 메서드를 참조하세요.

다음 예제에서는 구문 분석 문자열의 문화권 구분 특성을 보여 줍니다. en-US, fr-FR및 고정 문화권의 규칙을 사용하여 수천 개의 구분 기호를 포함하는 문자열을 구문 분석하려고 합니다. 쉼표를 그룹 구분 기호로 사용하고 마침표를 10진수 구분 기호로 사용하는 문자열은 fr-FR 문화에서 구문 분석에 실패하고, 공백을 그룹 구분 기호로, 쉼표를 10진수 구분 기호로 사용하는 문자열은 en-US 및 고정 문화에서 구문 분석에 실패합니다.

C#

```
using System;
using System.Globalization;

public class ParseEx1
{
    public static void Main()
    {
        String[] values = { "1,034,562.91", "9 532 978,07" };
        String[] cultureNames = { "en-US", "fr-FR", "" };

        foreach (var value in values)
        {
            foreach (var cultureName in cultureNames)
            {
                CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
                String name = culture.Name == "" ? "Invariant" :
culture.Name;
                try
                {
                    Decimal amount = Decimal.Parse(value, culture);
                    Console.WriteLine($"'{value}' --> {amount} ({name})");
                }
                catch (FormatException)
                {
                    Console.WriteLine($"'{value}': FormatException
({name})");
                }
            }
            Console.WriteLine();
        }
    }
}
```

```
// The example displays the following output:
//       '1,034,562.91' --> 1034562.91 (en-US)
//       '1,034,562.91': FormatException (fr-FR)
//       '1,034,562.91' --> 1034562.91 (Invariant)
//
//       '9 532 978,07': FormatException (en-US)
//       '9 532 978,07' --> 9532978.07 (fr-FR)
//       '9 532 978,07': FormatException (Invariant)
```

구문 분석은 일반적으로 다음 두 가지 컨텍스트에서 발생합니다.

- 사용자 입력을 숫자 값으로 변환하도록 설계된 작업입니다.
- 숫자 값을 왕복하도록 설계된 연산으로, 즉, 이전에 문자열로 serialize된 숫자 값을 역직렬화합니다.

다음 섹션에서는 이러한 두 작업에 대해 자세히 설명합니다.

사용자 문자열 해석

사용자가 입력한 숫자 문자열을 구문 분석할 때는 항상 사용자의 문화권 설정을 반영하는 NumberFormatInfo 개체를 인스턴스화해야 합니다. 사용자 사용자 지정을 반영하는 NumberFormatInfo 개체를 인스턴스화하는 방법에 대한 자세한 내용은 동적 데이터 섹션을 참조하세요.

다음 예제에서는 사용자 문화권 설정을 반영하는 구문 분석 작업과 그렇지 않은 구문 분석 작업의 차이점을 보여 줍니다. 이 경우 기본 시스템 문화권은 en-US입니다. 그러나 사용자는 제어판 **지역 및 언어**에서 ","를 소수 기호로, "."를 그룹 구분 기호로 정의했습니다. 일반적으로 이러한 기호는 기본 en-US 문화권에서 반전됩니다. 사용자가 사용자 설정을 반영하는 문자열을 입력하고 문자열이 사용자 설정(재정의)도 반영하는 NumberFormatInfo 개체에 의해 구문 분석되면 구문 분석 작업이 올바른 결과를 반환합니다. 그러나 표준 en-US 문화 설정을 반영하는 NumberFormatInfo 객체가 문자열을 구문 분석할 때 쉼표 기호를 그룹 구분 기호로 오인하여 잘못된 결과를 반환합니다.

C#

```
using System;
using System.Globalization;

public class ParseUserEx
{
    public static void Main()
    {
        CultureInfo stdCulture = CultureInfo.GetCultureInfo("en-US");
        CultureInfo custCulture = CultureInfo.CreateSpecificCulture("en-
US");
```

```
        String value = "310,16";
        try
        {
            Console.WriteLine($"{stdCulture.Name} culture reflects user
overrides: {stdCulture.UseUserOverride}");
            Decimal amount = Decimal.Parse(value, stdCulture);
            Console.WriteLine($"'{value}' -->
{amount.ToString(CultureInfo.InvariantCulture)}");
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{value}'");
        }
        Console.WriteLine();

        try
        {
            Console.WriteLine($"{custCulture.Name} culture reflects user
overrides: {custCulture.UseUserOverride}");
            Decimal amount = Decimal.Parse(value, custCulture);
            Console.WriteLine($"'{value}' -->
{amount.ToString(CultureInfo.InvariantCulture)}");
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{value}'");
        }
    }
}
// The example displays the following output:
//       en-US culture reflects user overrides: False
//       '310,16' --> 31016
//
//       en-US culture reflects user overrides: True
//       '310,16' --> 310.16
```

숫자 데이터 직렬화 및 역직렬화

숫자 데이터가 문자열 형식으로 직렬화되고 나중에 역직렬화되고 구문 분석되는 경우 고정 문화권의 규칙을 사용하여 문자열을 생성하고 구문 분석해야 합니다. 서식 지정 및 구문 분석 작업은 특정 문화권의 규칙을 반영해서는 안 됩니다. 문화권별 설정을 사용하는 경우 데이터의 이식성이 엄격하게 제한됩니다. 문화권별 설정이 serialize된 스레드의 설정과 동일한 스레드에서만 역직렬화할 수 있습니다. 경우에 따라 데이터가 직렬화된 동일한 시스템에서 데이터를 역직렬화할 수도 없습니다.

다음 예제에서는 이 원칙을 위반할 때 발생할 수 있는 작업을 보여 줍니다. 현재 스레드에서 en-US 문화권의 문화권별 설정을 사용하는 경우 배열의 부동 소수점 값이 문자열로 변환됩니다. 그런 다음 데이터는 pt-BR 문화권의 문화권별 설정을 사용하는 스레드에 의해 구문 분석됩니다. 이 경우 각 구문 분석 작업이 성공하지만 데이터가 왕복에 성공하지 못

하고 데이터 손상이 발생합니다. 다른 경우에는 구문 분석 작업이 실패하고 FormatException 예외가 throw될 수 있습니다.

C#

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Threading;

public class ParsePersistedEx
{
    public static void Main()
    {
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("en-
US");
        PersistData();

        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("pt-
BR");
        RestoreData();
    }

    private static void PersistData()
    {
        // Define an array of floating-point values.
        Double[] values = { 160325.972, 8631.16, 1.304e5, 98017554.385,
                            8.5938287084321676e94 };
        Console.WriteLine("Original values: ");
        foreach (var value in values)
            Console.WriteLine(value.ToString("R",
CultureInfo.InvariantCulture));

        // Serialize an array of doubles to a file
        StreamWriter sw = new StreamWriter(@".\NumericData.bin");
        for (int ctr = 0; ctr < values.Length; ctr++)
        {
            sw.Write(values[ctr].ToString("R"));
            if (ctr < values.Length - 1) sw.Write("|");
        }
        sw.Close();
        Console.WriteLine();
    }

    private static void RestoreData()
    {
        // Deserialize the data
        StreamReader sr = new StreamReader(@".\NumericData.bin");
        String data = sr.ReadToEnd();
        sr.Close();

        String[] stringValues = data.Split('|');
        List<Double> newValueList = new List<Double>();
```

```
        foreach (var stringValue in stringValues)
        {
            try
            {
                newValueList.Add(Double.Parse(stringValue));
            }
            catch (FormatException)
            {
                newValueList.Add(Double.NaN);
            }
        }

        Console.WriteLine("Restored values:");
        foreach (var newValue in newValueList)
            Console.WriteLine(newValue.ToString("R",
NumberFormatInfo.InvariantInfo));
    }
}
// The example displays the following output:
//       Original values:
//       160325.972
//       8631.16
//       130400
//       98017554.385
//       8.5938287084321671E+94
//
//       Restored values:
//       160325972
//       863116
//       130400
//       98017554385
//       8.5938287084321666E+110
```

PersianCalendar 클래스

아티클 • 2025. 04. 21.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

페르시아 달력은 페르시아어로 사용되는 대부분의 국가/지역에서 사용되지만 일부 지역에서는 월 이름이 다릅니다. 페르시아 달력은 이란과 아프가니스탄의 공식 달력이며 카자흐스탄과 타지키스탄과 같은 지역의 대체 달력 중 하나입니다.

> **참고**
>
> .NET에서 PersianCalendar 클래스 및 다른 일정 클래스를 사용하는 방법에 대한 자세한 내용은 **일정 작업을 참조하세요**.

페르시아 달력은 태양 히즈리 달력이며, 무하마드 (PBUH)가 메카에서 메디나로 이주 한 해 622 C.E.에 해당하는 히즈라의 해부터 시작됩니다.

페르시아 달력은 태양 연도를 기반으로 하며 약 365일 길이입니다. 1년은 사계절을 거치며, 해는 태양이 남반구에서 북반구까지 적도를 가로지르는 것처럼 보이자 지구의 중심에서 볼 수 있습니다. 새해는 북반구에서 봄의 첫 날 인 파바르딘의 달의 첫 날을 표시합니다. 예를 들어 2002년 3월 21일 C.E. 날짜는 1381년 안노 페르시코의 파바르딘 달 첫째 날에 해당합니다.

페르시아 달력의 처음 6개월은 각각 31일이며, 다음 5개월마다 30일이 있으며, 지난 달에는 평년 29일, 윤년 30일이 있습니다. 윤년은 33으로 나눌 때 1, 5, 9, 13, 17, 22, 26 또는 30의 나머지를 갖는 해입니다. 예를 들어 1370년은 33으로 나누면 나머지 17이 생성되기 때문에 윤년입니다. 33년 주기마다 약 8개의 윤년이 있습니다.

페르시아어 클래스 및 .NET 버전

.NET Framework 4.6부터 클래스는 PersianCalendar 관측 알고리즘 대신 Hijri 태양 천문학 알고리즘을 사용하여 날짜를 계산합니다. 이렇게 하면 PersianCalendar 페르시아 달력이 가장 널리 사용되고 있는 두 나라인 이란과 아프가니스탄에서 사용 중인 페르시아 달력과 일치하는 구현이 수행됩니다. .NET Framework 4.6이 설치된 경우 .NET Framework 4 이상에서 실행되는 모든 앱에 변경 내용이 적용됩니다.

변경된 알고리즘의 결과로:

- 두 알고리즘은 양력에서 날짜를 1800에서 2123 사이로 변환할 때 동일한 결과를 반환해야 합니다.
- 두 알고리즘은 양력에서 1800 이전 및 2123 이후 날짜를 변환할 때 서로 다른 결과를 반환할 수 있습니다.

- MinSupportedDateTime 속성 값은 그레고리오력의 0622년 3월 21일에서 그레고리오력의 0622년 3월 22일로 변경되었습니다.
- MaxSupportedDateTime 이 속성 값은 페르시아 달력에서 9378년 10월 10일에서 페르시아 달력의 9378년 10월 13일로 변경되었습니다.
- 메서드는 IsLeapYear 이전과 다른 결과를 반환할 수 있습니다.

페르시아어 클래스 사용

개체를 PersianCalendar 사용하여 페르시아 달력에서 날짜를 계산하거나 페르시아 날짜를 그레고리오 날짜로 변환할 수 있습니다. 페르시아 달력은 페르시아어(아프가니스탄) 및 중앙 쿠르드어(이란)와 같은 문화권의 기본 달력 입니다.

RegionInfo 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스와 CultureInfo 달리 클래스는 RegionInfo 사용자 기본 설정을 나타내지 않으며 사용자의 언어 또는 문화권에 의존하지 않습니다.

RegionInfo 개체와 연결된 이름

개체의 RegionInfo 이름은 국가/지역의 ISO 3166에 정의된 두 글자 코드 중 하나입니다. 경우는 중요하지 않습니다. , Name및 TwoLetterISORegionName 속성은 ThreeLetterISORegionName대문자로 적절한 코드를 반환합니다. 현재 이름 목록은 RegionInfoISO 3166: 국가 코드를 참조하세요.

RegionInfo 개체 인스턴스화

개체를 RegionInfo 인스턴스화하려면 생성자에게 미국의 경우 "US"와 같은 두 글자 영역 이름 또는 영어(미국)의 경우 "en-US"와 같은 특정 문화권의 이름을 전달 RegionInfo(String) 합니다. 그러나 개체가 완전히 언어 독립적이 아니므로 두 글자 영역 이름 대신 특정 문화권 이름을 RegionInfo 사용하는 것이 좋습니다. 문화권 이름에 따라 여러 RegionInfo 속성(예DisplayName: 및 NativeNameCurrencyNativeName)이 달라집니다.

다음 예제에서는 벨기에를 나타내는 세 개체의 RegionInfo 속성 값 차이를 보여 줍니다. 첫 번째는 지역 이름(`BE`)에서만 인스턴스화되고, 두 번째와 세 번째는 문화권 이름(프랑스어(`fr-BE` 벨기에) 및 `nl-BE` 네덜란드어(벨기에)에서 각각 인스턴스화됩니다. 이 예제에서는 리플렉션을 사용하여 각 RegionInfo 개체의 속성 값을 검색합니다.

C#

```
using System;
using System.Globalization;
using System.Reflection;

public class Example
{
    public static void Main()
    {
        // Instantiate three Belgian RegionInfo objects.
        RegionInfo BE = new RegionInfo("BE");
        RegionInfo frBE = new RegionInfo("fr-BE");
        RegionInfo nlBE = new RegionInfo("nl-BE");

        RegionInfo[] regions = { BE, frBE, nlBE };
```

```
        PropertyInfo[] props =
typeof(RegionInfo).GetProperties(BindingFlags.Instance | BindingFlags.Public);

        Console.WriteLine("{0,-30}{1,18}{2,18}{3,18}\n",
                          "RegionInfo Property", "BE", "fr-BE", "nl-BE");
        foreach (var prop in props)
        {
            Console.Write("{0,-30}", prop.Name);
            foreach (var region in regions)
                Console.Write("{0,18}", prop.GetValue(region, null));

            Console.WriteLine();
        }
    }
}
// The example displays the following output:
//    RegionInfo Property                          BE              fr-BE
nl-BE
//
//    Name                                         BE              fr-BE
nl-BE
//    EnglishName                             Belgium           Belgium
Belgium
//    DisplayName                             Belgium           Belgium
Belgium
//    NativeName                               België          Belgique
België
//    TwoLetterISORegionName                       BE                BE
BE
//    ThreeLetterISORegionName                    BEL               BEL
BEL
//    ThreeLetterWindowsRegionName                BEL               BEL
BEL
//    IsMetric                                   True              True
True
//    GeoId                                        21                21
21
//    CurrencyEnglishName                        Euro              Euro
Euro
//    CurrencyNativeName                         euro              euro
euro
//    CurrencySymbol                                €                 €
€
//    ISOCurrencySymbol                          EUR               EUR
EUR
```

다음과 같은 시나리오에서는 개체를 인스턴스화 RegionInfo 할 때 국가/지역 이름 대신 문화권 이름을 사용합니다.

- 언어 이름이 중요한 경우. 예를 들어 문화권 이름의 경우 `es-US` 애플리케이션에 "미국" 대신 "Estados Unidos"가 표시되도록 할 수 있습니다. 국가/지역 이름(`US`)만 사용하면 언어에 관계없이 "미국"이 생성되므로 문화권 이름을 대신 사용해야 합니다.

- 스크립트 차이점을 고려해야 하는 경우입니다. 예를 들어, `AZ` 국가/지역은 아제르바이잔 문화와 관련된 `az-Latn-AZ` 및 `az-Cyrl-AZ` 이름을 다루며, 라틴 문자와 키릴 자모는 이 국가/지역에서 매우 다를 수 있습니다.
- 세부 정보의 유지 관리가 중요한 경우. 멤버가 반환하는 RegionInfo 값은 문화권 이름 또는 지역 이름을 사용하여 개체를 RegionInfo 인스턴스화했는지 여부에 따라 다를 수 있습니다. 예를 들어 다음 표에서는 "미국" 지역, "en-US" 문화권 및 "es-US" 문화권을 사용하여 개체를 인스턴스화할 때 RegionInfo 반환 값의 차이를 보여 줍니다.

테이블 확장

회원	"미국"	"en-US"	"es-US"
CurrencyNativeName	`US Dollar`	`US Dollar`	`Dólar de EE.UU.`
Name	`US`	`en-US`	`es-US`
NativeName	`United States`	`United States`	`Estados Unidos`
ToString	`US`	`en-US`	`es-US`

SortKey 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

두 문자열의 문화권 구분 비교는 스크립트, 알파벳, 대/소문자 및 발음 가중치를 포함하여 여러 범주의 정렬 가중치가 있는 문자열의 각 문자에 따라 달라집니다. 정렬 키는 특정 문자열에 대한 이러한 가중치의 리포지토리 역할을 합니다.

CompareInfo.GetSortKey 메서드는 지정된 문자열에서 문화에 민감한 문자 매핑을 반영하는 SortKey 클래스의 인스턴스를 반환합니다. SortKey 개체의 값은 KeyData 속성에서 반환되는 키 데이터입니다. 이 키 데이터는 문자열, 문화권별 정렬 규칙 및 사용자가 지정한 비교 옵션을 인코딩하는 일련의 바이트로 구성됩니다. 정렬 키를 사용한 비교는 각 정렬 키의 해당 키 데이터에 대한 비트 비교로 구성됩니다. 예를 들어, GetSortKey(String, CompareOptions) 메서드를 CompareOptions.IgnoreCase 값과 함께 호출하여 정렬 키를 생성하면, 이 정렬 키를 사용하는 문자열 비교 작업은 대소문자를 구분하지 않습니다.

문자열에 대한 정렬 키를 만든 후 정적 SortKey.Compare 메서드를 호출하여 정렬 키를 비교합니다. 이 메서드는 간단한 바이트 바이트 비교를 수행하므로 메서드보다 String.CompareCompareInfo.Compare 훨씬 빠릅니다.

> **참고**
>
> Windows 운영 체제의 정렬 및 비교 작업에 사용되는 문자 가중치에 대한 정보가 포함된 텍스트 파일 집합인 **정렬** 가중치 **테이블, 기본 유니코드 데이터 정렬 요소 테이블**, Linux 및 macOS용 정렬 가중치 테이블을 다운로드할 수 있습니다.

성능 고려 사항

문자열 비교를 수행할 때 Compare와 CompareInfo.Compare 메서드는 동일한 결과를 생성하지만, 서로 다른 시나리오에 적합합니다.

상위 수준에서 메서드는 CompareInfo.Compare 각 문자열에 대한 정렬 키를 생성하고 비교를 수행한 다음 정렬 키를 삭제하고 비교 결과를 반환합니다. 그러나 메서드는 CompareInfo.Compare 실제로 비교를 수행하기 위해 전체 정렬 키를 생성하지 않습니다. 대신 메서드는 각 문자열에서 각 텍스트 요소(기본 문자, 서로게이트 쌍 또는 결합 문자 시퀀스)에 대한 키 데이터를 생성합니다. 그런 다음 메서드는 해당 텍스트 요소에 대한 키 데이터를 비교합니다. 비교의 최종 결과가 결정되는 즉시 작업이 종료됩니다. 정렬 키 정보는 계산되지만 개체는 만들어지지 않습니다 SortKey . 이 전략은 두 문자열을 한 번 비교하는 경우 성능 측면에서 경제적이지만 동일한 문자열을 여러 번 비교하면 비용이 많이 듭니다.

이 메서드는 비교를 수행하기 전에 각 문자열에 대해 Compare 개체를 생성해야 합니다. 이 전략은 개체를 생성하는 SortKey 데 투자된 시간과 메모리 때문에 첫 번째 비교의 성능 측면에서 비용이 많이 듭니다. 그러나 동일한 정렬 키를 여러 번 비교하면 경제적이 됩니다.

예를 들어 데이터베이스 테이블에서 문자열 기반 인덱스 열이 지정된 검색 문자열과 일치하는 행을 검색하는 애플리케이션을 작성한다고 가정합니다. 테이블에는 수천 개의 행이 포함되어 있으며 각 행의 인덱스와 검색 문자열을 비교하는 데 시간이 오래 걸릴 수 있습니다. 따라서 애플리케이션이 행과 해당 인덱스 열을 저장하는 경우 검색 성능 향상을 위해 인덱스에 대한 정렬 키도 생성하고 열에 저장합니다. 애플리케이션은 대상 행을 검색할 때 검색 문자열과 인덱스 문자열을 비교하는 대신 검색 문자열의 정렬 키를 인덱스 문자열의 정렬 키와 비교합니다.

보안 고려 사항

이 CompareInfo.GetSortKey(String, CompareOptions) 메서드는 지정된 문자열과 SortKey 값을 바탕으로 한 값을 가지며, 기본 CompareOptions 개체와 연결된 문화권이 있는 CompareInfo 개체를 반환합니다. 보안 결정이 문자열 비교 또는 대/소문자 변경에 따라 달라지는 경우, 운영 체제의 문화권 설정에 관계없이 작업의 동작이 일관되도록 불변 문화권의 CompareInfo.GetSortKey(String, CompareOptions) 메서드를 사용해야 합니다.

정렬 키를 가져오려면 다음 단계를 사용합니다.

1. CultureInfo.InvariantCulture 속성에서 불변 문화권을 검색합니다.
2. 속성 CompareInfo에서 CultureInfo.CompareInfo 고정 문화권에 대한 개체를 검색합니다.
3. CompareInfo.GetSortKey(String, CompareOptions) 메서드를 호출합니다.

개체 값 SortKey 으로 작업하는 것은 지정된 LCMAP_SORTKEY 값을 사용하여 Windows `LCMapString` 메서드를 호출하는 것과 같습니다. 그러나 개체의 SortKey 경우 영어 문자의 정렬 키는 한국어 문자의 정렬 키 앞에 섰습니다.

SortKey 개체는 직렬화할 수 있지만, 이것은 AppDomain 개체를 넘어 전달할 수 있도록 하기 위한 것입니다. 애플리케이션이 개체를 SortKey serialize하는 경우 새 버전의 .NET이 있는 경우 애플리케이션은 모든 정렬 키를 다시 생성해야 합니다.

정렬 키에 대한 자세한 내용은 유니코드 컨소시엄 웹 사이트에서 유니코드 기술 표준 #10, "유니코드 데이터 정렬 알고리즘"을 참조하세요.

SortVersion 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

.NET Framework의 정렬 및 문자열 비교

.NET Framework 4를 통해 각 버전의 .NET Framework에는 문자열 정규화에 대한 정렬 가중치 및 데이터가 포함되고 특정 버전의 유니코드를 기반으로 하는 테이블이 포함되었습니다. .NET Framework 4.5 이상 버전에서 이러한 테이블의 존재는 운영 체제에 따라 달라집니다.

- Windows 7 및 이전 버전에서는 테이블이 문자열을 비교하고 정렬하는 데 계속 사용됩니다.
- Windows 8에서 .NET Framework는 문자열 비교 및 순서 지정 작업을 운영 체제에 위임합니다.

따라서 다음 표와 같이 문자열 비교 결과는 .NET Framework 버전뿐만 아니라 운영 체제 버전에도 따라 달라질 수 있습니다. 지원되는 유니코드 버전 목록은 문자 비교 및 정렬에만 적용됩니다. 범주별 유니코드 문자 분류에는 적용되지 않습니다.

테이블 확장

.NET Framework 버전	운영 체제	유니코드 버전
4	모든 운영 체제	유니코드 5.0
4.5 이상 버전	Windows 7	유니코드 5.0
4.5 이상 버전	Windows 8 이상 버전	유니코드 6.0

Windows 8에서는 문자열 비교 및 순서 지정에 사용되는 유니코드 버전이 운영 체제 버전에 따라 달라지므로 특정 버전의 .NET Framework에서 실행되는 애플리케이션에서도 문자열 비교 결과가 다를 수 있습니다.

.NET Core의 정렬 및 문자열 비교

모든 버전의 .NET(Core)은 문자열 비교를 수행할 때 기본 운영 체제를 사용합니다. 따라서 문자열 비교의 결과 또는 문자열이 정렬되는 순서는 비교를 수행할 때 운영 체제에서 사용하는 유니코드 버전에 따라 달라집니다. Linux, macOS 및 Windows 10 이상 버전에서 유니코드용 International Components 라이브러리는 API 비교 및 정렬을 위한 구현을 제공합니다.

SortVersion 클래스 사용

클래스는 SortVersion 문자열 비교 및 순서 지정을 위해 .NET에서 사용하는 유니코드 버전에 대한 정보를 제공합니다. 이를 통해 개발자는 애플리케이션의 문자열을 비교하고 정렬하는 데 사용되는 유니코드 버전의 변경 내용을 감지하고 성공적으로 처리할 수 있는 애플리케이션을 작성할 수 있습니다.

다음 두 가지 방법으로 개체를 SortVersion 인스턴스화할 수 있습니다.

- 생성자를 호출 SortVersion 하여 버전 번호 및 정렬 ID에 따라 새 SortVersion 개체를 인스턴스화합니다. 이 생성자는 저장된 데이터에서 개체를 다시 만들 SortVersion 때 가장 유용합니다.
- CompareInfo.Version 속성의 값을 검색하여. 이 속성은 애플리케이션이 실행 중인 .NET 구현에서 사용하는 유니코드 버전에 대한 정보를 제공합니다.

클래스에는 SortVersion 유니코드 버전과 문자열 비교에 사용되는 특정 문화권을 나타내는 두 가지 속성 FullVersion 이 있습니다 SortId. 속성 FullVersion 은 문자열 비교에 사용되는 유니코드 버전을 반영하는 임의의 숫자 값이며 SortId , 이 속성은 문자열 비교에 규칙이 사용되는 문화권을 반영하는 임의 Guid 값입니다. 이 두 속성의 값은 SortVersion 메서드, Equals 연산자 또는 Equality 연산자를 사용하여 두 Inequality 개체를 비교할 때만 중요합니다.

일반적으로 인덱스 또는 리터럴 문자열 자체와 같이 문화권에 민감한 정렬된 문자열 데이터의 형식을 저장하거나 검색할 때 개체를 사용합니다 SortVersion . 이렇게 하려면 다음 단계가 필요합니다.

1. 정렬된 문자열 데이터가 저장되면, FullVersion와 SortId 속성 값도 함께 저장됩니다.
2. 순서가 지정된 문자열 데이터를 검색할 때 생성자를 호출 SortVersion 하여 문자열 순서 지정에 사용되는 개체를 다시 SortVersion 만들 수 있습니다.
3. 이 새로 인스턴스화된 SortVersion 개체는 문자열 데이터를 정렬하는 데 규칙을 사용하는 문화권을 반영하는 개체와 SortVersion 비교됩니다.
4. 두 SortVersion 개체가 같지 않으면 문자열 데이터의 순서를 다시 지정해야 합니다.

.NET 앱의 리소스

2025. 06. 17.

거의 모든 프로덕션 품질 앱은 리소스를 사용해야 합니다. 리소스는 앱과 함께 논리적으로 배포되는 실행 불가능한 데이터입니다. 리소스가 앱에 오류 메시지 또는 사용자 인터페이스의 일부로 표시될 수 있습니다. 리소스는 문자열, 이미지 및 지속형 개체를 비롯한 다양한 형식의 데이터를 포함할 수 있습니다. (지속형 개체를 리소스 파일에 쓰려면 개체를 직렬화할 수 있어야 합니다.) 리소스 파일에 데이터를 저장하면 전체 앱을 다시 컴파일하지 않고도 데이터를 변경할 수 있습니다. 또한 단일 위치에 데이터를 저장할 수 있으며 여러 위치에 저장된 하드 코딩된 데이터에 의존할 필요가 없습니다.

.NET은 리소스 만들기 및 지역화를 포괄적으로 지원합니다. 또한 .NET은 지역화된 리소스를 패키징하고 배포하기 위한 간단한 모델을 지원합니다.

리소스 만들기 및 지역화

지역화되지 않은 앱에서는 특히 소스 코드의 여러 위치에서 하드 코딩될 수 있는 문자열의 경우 리소스 파일을 앱 데이터의 리포지토리로 사용할 수 있습니다. 가장 일반적으로 리소스를 텍스트(.txt) 또는 XML(.resx) 파일로 만들고 Resgen.exe(리소스 파일 생성기) 를 사용하여 이진 .resources 파일로 컴파일합니다. 그런 다음, 언어 컴파일러에서 이러한 파일을 앱의 실행 파일에 포함할 수 있습니다. 리소스를 만드는 방법에 대한 자세한 내용은 리소스 파일 만들기를 참조하세요.

특정 문화권에 대한 앱의 리소스를 지역화할 수도 있습니다. 이렇게 하면 지역화된(번역된) 버전의 앱을 빌드할 수 있습니다. 지역화된 리소스를 사용하는 앱을 개발할 때 적절한 리소스를 사용할 수 없는 경우 리소스가 사용되는 중립 또는 대체 문화권 역할을 하는 문화권을 지정합니다. 일반적으로 중립 문화권의 리소스는 앱의 실행 파일에 저장됩니다. 개별 지역화된 문화권에 대한 나머지 리소스는 독립 실행형 위성 어셈블리에 저장됩니다. 자세한 내용은 위성 어셈블리 만들기를 참조하세요.

리소스 패키징 및 배포

위성 어셈블리에 지역화된 앱 리소스를 배포합니다. 위성 어셈블리에는 단일 문화권의 리소스가 포함됩니다. 앱 코드가 포함되어 있지 않습니다. 위성 어셈블리 배포 모델에서는 앱이 지원하는 각 문화권에 대해 하나의 기본 어셈블리(일반적으로 주 어셈블리)와 하나의 위성 어셈블리를 사용하여 앱을 만듭니다. 위성 어셈블리는 주 어셈블리의 일부가 아니므로 앱의 주 어셈블리를 대체하지 않고 특정 문화권에 해당하는 리소스를 쉽게 바꾸거나 업데이트할 수 있습니다.

앱의 기본 리소스 어셈블리를 구성할 리소스를 신중하게 결정합니다. 주 어셈블리의 일부이므로 주 어셈블리를 변경하려면 주 어셈블리를 바꿔야 합니다. 기본 리소스를 제공하지 않으면, 리소

스 대체 프로세스가 해당 리소스를 발견하려고 시도할 때 예외가 발생합니다. 잘 설계된 앱에서는 리소스를 사용할 때 예외가 발생해서는 안 됩니다.

자세한 내용은 리소스 패키징 및 배포 문서를 참조하세요.

리소스 검색

런타임에 앱은 속성에 지정된 문화권에 따라 스레드별로 적절한 지역화된 리소스를 CultureInfo.CurrentUICulture 로드합니다. 이 속성 값은 다음과 같이 파생됩니다.

- 지역화된 문화권을 나타내는 개체를 속성에 직접 할당합니다 CultureInfoThread.CurrentUICulture .
- 문화권이 명시적으로 할당되지 않은 경우 속성에서 CultureInfo.DefaultThreadCurrentUICulture 기본 스레드 UI 문화권을 검색합니다.
- 기본 스레드 UI 문화권이 명시적으로 할당되지 않은 경우 로컬 컴퓨터에서 현재 사용자의 문화권을 검색합니다. Windows에서 실행되는 .NET 구현은 Windows GetUserDefaultUILanguage 함수를 호출하여 이 작업을 수행합니다.

현재 UI 문화권이 설정되는 방법에 대한 자세한 내용은 CultureInfo 및 CultureInfo.CurrentUICulture 참조 페이지를 확인하세요.

그 후 System.Resources.ResourceManager 클래스를 사용하여 현재 UI 문화권 또는 특정 문화권에 대한 리소스를 검색할 수 있습니다. 클래스는 ResourceManager 리소스 검색에 가장 일반적으로 사용되지만 네 System.Resources 임스페이스에는 리소스를 검색하는 데 사용할 수 있는 추가 형식이 포함됩니다. 여기에는 다음이 포함됩니다.

- ResourceReader 어셈블리에 포함되거나 독립 실행형 이진 .resources 파일에 저장된 리소스를 열거할 수 있는 클래스입니다. 런타임에 사용할 수 있는 리소스의 정확한 이름을 모를 때 유용합니다.
- ResXResourceReader XML(.resx) 파일에서 리소스를 검색할 수 있는 클래스입니다.
- 대체 ResourceSet 규칙을 관찰하지 않고 특정 문화권의 리소스를 검색할 수 있는 클래스입니다. 리소스는 어셈블리 또는 독립 실행형 이진 .resources 파일에 저장할 수 있습니다. 다른 원본에서 리소스를 검색하기 위해 IResourceReader 클래스를 사용할 수 있도록 하는 ResourceSet 구현을 개발할 수도 있습니다.
- ResXResourceSet XML 리소스 파일의 모든 항목을 메모리로 검색할 수 있는 클래스입니다.

참고하십시오

- CultureInfo

- CultureInfo.CurrentUICulture
- 리소스 파일 만들기
- 리소스 패키징 및 배포
- 위성 어셈블리 만들기
- 리소스 검색
- .NET의 지역화

.NET 앱의 리소스 파일 만들기

문자열, 이미지 또는 개체 데이터와 같은 리소스를 리소스 파일에 포함하여 애플리케이션에서 쉽게 사용할 수 있게 설정할 수 있습니다. .NET Framework에서는 리소스 파일을 만드는 다섯 가지 방법을 제공합니다.

- 문자열 리소스가 포함된 텍스트 파일을 만듭니다. 리소스 파일 생성기(*resgen.exe*)를 사용하여 텍스트 파일을 이진 리소스(.resources) 파일로 변환할 수 있습니다. 그다음에 언어 컴파일러를 사용하여 이진 리소스 파일을 애플리케이션 실행 파일 또는 애플리케이션 라이브러리에 포함하거나, 어셈블리 링커(Al.exe)를 사용하여 위성 어셈블리에 포함할 수 있습니다. 자세한 내용은 텍스트 파일의 리소스 섹션을 참조하세요.

- 문자열, 이미지 또는 개체 데이터가 포함된 XML 리소스(.resx) 파일을 만듭니다. 리소스 파일 생성기(*resgen.exe*)를 사용하여 .resx 파일을 이진 리소스(.resources) 파일로 변환할 수 있습니다. 그다음에 언어 컴파일러를 사용하여 이진 리소스 파일을 애플리케이션 실행 파일 또는 애플리케이션 라이브러리에 포함하거나, 어셈블리 링커(Al.exe)를 사용하여 위성 어셈블리에 포함할 수 있습니다. 자세한 내용은 .resx 파일의 리소스 섹션을 참조하세요.

- System.Resources 네임스페이스의 형식을 사용하여 프로그래밍 방식으로 XML 리소스(.resx) 파일을 만듭니다. .resx 파일을 만들고, 해당 리소스를 열거하고, 특정 리소스를 이름으로 검색할 수 있습니다. 자세한 내용은 프로그래밍 방식으로 .resx 파일 작업을 참조하세요.

- 프로그래밍 방식으로 이진 리소스(.resources) 파일을 만듭니다. 그다음에 언어 컴파일러를 사용하여 파일을 애플리케이션 실행 파일 또는 애플리케이션 라이브러리에 포함하거나, 어셈블리 링커(Al.exe)를 사용하여 위성 어셈블리에 포함할 수 있습니다. 자세한 내용은 .resources 파일의 리소스 섹션을 참조하세요.

- Visual Studio 를 사용하여 리소스 파일을 만들고 프로젝트에 포함합니다. Visual Studio에서는 리소스를 추가, 삭제 및 수정할 수 있는 리소스 편집기를 제공합니다. 컴파일 시간에 리소스 파일은 자동으로 이진 .resources 파일로 변환되고 애플리케이션 어셈블리 또는 위성 어셈블리에 포함됩니다. 자세한 내용은 Visual Studio의 리소스 파일 섹션을 참조하세요.

텍스트 파일의 리소스

텍스트 파일(.txt 또는 .restext)을 사용하면 문자열 리소스만 저장할 수 있습니다. 문자열이 아닌 리소스의 경우 .resx 파일을 사용하거나 프로그래밍 방식으로 .resx 파일을 만듭니다. 문자열 리소스가 포함된 텍스트 파일의 형식은 다음과 같습니다.

text

```
# This is an optional comment.
name = value
```

```
; This is another optional comment.
name = value

; The following supports conditional compilation if X is defined.
#ifdef X
name1=value1
name2=value2
#endif

# The following supports conditional compilation if Y is undefined.
#if !Y
name1=value1
name2=value2
#endif
```

.txt 및 .restext 파일의 리소스 파일 형식은 같습니다. .restext 파일 확장명은 단순히 텍스트 파일을 텍스트 기반 리소스 파일로 즉시 식별할 수 있도록 합니다.

문자열 리소스는 *name/value* 쌍으로 나타납니다. 여기서 *name*은 리소스를 식별하는 문자열이고, *value*는 *name*을 ResourceManager.GetString과 같은 리소스 검색 메서드에 전달할 때 반환되는 리소스 문자열입니다. *name* 및 *value*는 등호(=)로 구분해야 합니다. 예시:

text

```
FileMenuName=File
EditMenuName=Edit
ViewMenuName=View
HelpMenuName=Help
```

> **⊗ 주의**
>
> 암호, 보안이 중요한 정보 또는 개인 데이터를 저장할 때는 리소스 파일을 사용하지 마세요.

빈 문자열(즉, String.Empty 값을 가진 리소스)은 텍스트 파일에 사용할 수 있습니다. 예시:

text

```
EmptyString=
```

.NET Framework 4.5부터, 그리고 모든 버전의 .NET Core에서, 텍스트 파일에서는 `#ifdef` *symbol...* `#endif` 및 `#if !` *symbol...* `#endif` 구문을 사용한 조건부 컴파일을 지원합니다. 그런 다음, *리소스 파일 생성기*(`/define`)와 함께 스위치를 하여 기호를 정의할 수 있습니다. 각 리소스에는 고유한 `#ifdef` *symbol...* `#endif` 또는 `#if !` *symbol...* `#endif` 구문이 필요합니다. `#ifdef` 문을 사용하고 *symbol*을 정의하면 연관된 리소스는 .resources 파일에 포함되고, 그렇지

않을 경우 포함되지 않습니다. `#if !` 문을 사용하고 *symbol*을 정의하지 않으면 연관된 리소스는 .resources 파일에 포함되고, 그렇지 않을 경우 포함되지 않습니다.

텍스트 파일에서 주석은 선택 사항이고 줄 시작 부분에서 세미콜론(;) 또는 파운드 기호(#) 뒤에 옵니다. 주석이 포함된 줄은 파일의 어느 곳에나 배치될 수 있습니다. 주석은 리소스 파일 생성기(*resgen.exe*)를 사용하여 만든 컴파일된 .resources 파일에 포함되지 않습니다.

텍스트 파일에 있는 빈 줄은 공백으로 간주하고 무시됩니다.

다음 예제에서는 `OKButton` 및 `CancelButton`이라는 두 개의 문자열 리소스를 정의합니다.

text

```
#Define resources for buttons in the user interface.
OKButton=OK
CancelButton=Cancel
```

텍스트 파일에 중복된 *name*이 포함된 경우 리소스 파일 생성기(*resgen.exe*)는 경고를 표시하고 두 번째 이름을 무시합니다.

*value*에는 줄 바꿈 문자가 포함될 수 없지만 `\n` 같은 C 언어 스타일의 이스케이프 문자를 사용하여 새 줄을 나타내고 `\t`를 사용하여 탭을 나타낼 수 있습니다. 이스케이프된 경우 백슬래시 문자를 포함할 수도 있습니다(예: "\\"). 또한 빈 문자열이 허용됩니다.

little-endian 또는 big-endian 바이트 순서의 UTF-8 인코딩이나 UTF-16 인코딩을 사용하여 텍스트 파일 형식으로 리소스를 저장합니다. 하지만 .txt 파일을 .resources 파일로 변환하는 리소스 파일 생성기(*resgen.exe*)는 기본적으로 파일을 UTF-8로 처리합니다. Resgen.exe가 UTF-16을 사용하여 인코딩된 파일을 인식하게 하려면 파일 시작 부분에 유니코드 바이트 순서 표시(U+FEFF)를 포함해야 합니다.

텍스트 형식의 리소스 파일을 .NET 어셈블리에 포함하려면 리소스 파일 생성기(*resgen.exe*)를 사용하여 파일을 이진 리소스(.resources) 파일로 변환해야 합니다. 그런 다음, 언어 컴파일러를 사용하여 .resources 파일을 .NET 어셈블리에 포함하거나, 어셈블리 링커(Al.exe)를 사용하여 위성 어셈블리에 포함할 수 있습니다.

다음 예제에서는 간단한 "Hello World" 콘솔 애플리케이션에 GreetingResources.txt라는 텍스트 형식의 리소스 파일을 사용합니다. 텍스트 파일은 사용자에게 이름을 입력하고 인사말을 표시하도록 지시하는 두 개의 문자열인 `prompt` 및 `greeting`을 정의합니다.

text

```
# GreetingResources.txt
# A resource file in text format for a "Hello World" application.
#
# Initial prompt to the user.
prompt=Enter your name:
```

```
# Format string to display the result.
greeting=Hello, {0}!
```

텍스트 파일은 다음 명령을 사용하여 .resources 파일로 변환됩니다.

콘솔

```
resgen GreetingResources.txt
```

다음 예제에서는 .resources 파일을 사용하여 사용자에게 메시지를 표시하는 콘솔 애플리케이션에 대한 소스 코드를 보여 줍니다.

C#

```
using System;
using System.Reflection;
using System.Resources;

public class Example
{
   public static void Main()
   {
      ResourceManager rm = new ResourceManager("GreetingResources",
                               typeof(Example).Assembly);
      Console.Write(rm.GetString("prompt"));
      string name = Console.ReadLine();
      Console.WriteLine(rm.GetString("greeting"), name);
   }
}
// The example displays output like the following:
//       Enter your name: Wilberforce
//       Hello, Wilberforce!
```

Visual Basic을 사용하고 있고 소스 코드 파일 이름이 Greeting.vb라면 다음 명령은 포함된 .resources 파일을 포함하는 실행 파일을 만듭니다.

콘솔

```
vbc greeting.vb -resource:GreetingResources.resources
```

C#을 사용하고 있고 소스 코드 파일 이름이 Greeting.cs라면 다음 명령은 포함된 .resources 파일을 포함하는 실행 파일을 만듭니다.

콘솔

```
csc greeting.cs -resource:GreetingResources.resources
```

.resx 파일의 리소스

문자열 리소스만 저장할 수 있는 텍스트 파일과 달리 XML 리소스(.resx) 파일은 문자열, 이진 데이터(예: 이미지, 아이콘, 오디오 클립) 및 프로그래밍 개체를 저장할 수 있습니다. .resx 파일에는 리소스 항목의 형식을 설명하고 데이터 구문 분석에 사용되는 XML에 대한 버전 관리 정보를 지정하는 표준 헤더가 포함됩니다. 리소스 파일 데이터는 XML 헤더를 따릅니다. 각 데이터 항목은 `data` 태그에 포함된 이름/값 쌍으로 구성됩니다. 해당 `name` 특성은 리소스 이름을 정의하고 중첩된 `value` 태그에는 리소스 값이 포함됩니다. 문자열 데이터의 경우 `value` 태그에 문자열이 포함됩니다.

예를 들어 다음 `data` 태그는 값이 "Enter your name:"인 `prompt`라는 문자열 리소스를 정의합니다.

XML

```xml
<data name="prompt" xml:space="preserve">
  <value>Enter your name:</value>
</data>
```

> **⊗ 주의**
>
> 암호, 보안이 중요한 정보 또는 개인 데이터를 저장할 때는 리소스 파일을 사용하지 마세요.

리소스 개체의 경우 **data** 태그에는 리소스의 데이터 형식을 나타내는 `type` 특성이 포함됩니다. 이진 데이터로 구성된 개체의 경우 `data` 태그에는 이진 데이터의 `mimetype` 형식을 나타내는 `base64` 특성도 포함됩니다.

> **ⓘ 참고 항목**
>
> 모든 .resx 파일에는 이진 serialization 포맷터를 사용하여 지정된 형식의 이진 데이터를 생성하고 구문 분석합니다. 따라서 개체에 대한 이진 serialization 형식이 호환되지 않는 방식으로 변경될 경우 .resx 파일이 유효하지 않을 수 있습니다.

다음 예제에서는 Int32 리소스 및 비트맵 이미지를 포함하는 .resx 파일의 일부가 포함됩니다.

XML

```xml
<data name="i1" type="System.Int32, mscorlib">
  <value>20</value>
</data>

<data name="flag" type="System.Drawing.Bitmap, System.Drawing,
    Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
```

```
    mimetype="application/x-microsoft.net.object.bytearray.base64">
  <value>
    AAEAAAD/////AQAAAAAAAAAMAgAAADtTeX...
  </value>
</data>
```

> **ⓘ Important**
>
> .resx 파일은 미리 정의된 형식의 잘 구성된(Well-Formed) XML로 구성되어야 하므로, 특히 .resx 파일에 문자열 이외의 리소스가 포함된 경우에는 .resx 파일을 수동으로 사용하지 않는 것이 좋습니다. 대신, Visual Studio 에서는 .resx 파일을 만들고 조작할 수 있는 투명한 인터페이스가 제공됩니다. 자세한 내용은 **Visual Studio의 리소스 파일** 섹션을 참조하세요. 프로그래밍 방식으로 .resx 파일을 만들고 조작할 수도 있습니다. 자세한 내용은 **프로그래밍 방식으로 .resx 파일 작업**을 참조하세요.

.resources 파일의 리소스

System.Resources.ResourceWriter 클래스를 사용하여 프로그래밍 방식으로 코드에서 직접 이진 리소스(.resources) 파일을 만들 수 있습니다. 리소스 파일 생성기(*resgen.exe*)를 사용하여 텍스트 파일 또는 .resx 파일에서 .resources 파일을 만들 수도 있습니다. .resources 파일에는 문자열 데이터 외에 이진 데이터(바이트 배열) 및 개체 데이터가 포함될 수 있습니다. 프로그래밍 방식으로 .resources 파일을 만들려면 다음 단계가 필요합니다.

1. 고유한 파일 이름을 사용하여 ResourceWriter 개체를 만듭니다. 파일 이름 또는 파일 스트림을 ResourceWriter 클래스 생성자로 지정하여 이 작업을 수행할 수 있습니다.

2. 파일에 추가할 각 명명된 리소스에 대해 ResourceWriter.AddResource 메서드의 오버로드 중 하나를 호출합니다. 리소스는 문자열, 개체 또는 이진 데이터 컬렉션(바이트 배열)일 수 있습니다.

3. ResourceWriter.Close 메서드를 호출하여 리소스를 파일에 쓰고 ResourceWriter 개체를 닫습니다.

> **⊗ 주의**
>
> 암호, 보안이 중요한 정보 또는 개인 데이터를 저장할 때는 리소스 파일을 사용하지 마세요.

다음 예제에서는 문자열 6개, 아이콘 한 개 및 애플리케이션 정의 개체 2개(`Automobile` 개체 2개)를 저장하는 CarResources.resources라는 .resources 파일을 프로그래밍 방식으로 만듭니다. 이 예제에서 정의되고 인스턴스화된 `Automobile` 클래스는 SerializableAttribute 특성으로 태그가 지정되어 이진 serialization 포맷터를 통해 유지될 수 있습니다.

C#

```
using System;
using System.Drawing;
using System.Resources;

[Serializable()] public class Automobile
{
   private string carMake;
   private string carModel;
   private int carYear;
   private int carDoors;
   private int carCylinders;

   public Automobile(string make, string model, int year) :
                     this(make, model, year, 0, 0)
   { }

   public Automobile(string make, string model, int year,
                     int doors, int cylinders)
   {
      this.carMake = make;
      this.carModel = model;
      this.carYear = year;
      this.carDoors = doors;
      this.carCylinders = cylinders;
   }

   public string Make {
      get { return this.carMake; }
   }

   public string Model {
      get { return this.carModel; }
   }

   public int Year {
      get { return this.carYear; }
   }

   public int Doors {
      get {
         return this.carDoors; }
   }

   public int Cylinders {
      get {
         return this.carCylinders; }
   }
}

public class Example
{
   public static void Main()
   {
```

```
      // Instantiate an Automobile object.
      Automobile car1 = new Automobile("Ford", "Model N", 1906, 0, 4);
      Automobile car2 = new Automobile("Ford", "Model T", 1909, 2, 4);
      // Define a resource file named CarResources.resx.
      using (ResourceWriter rw = new ResourceWriter(@".\CarResources.resources"))
      {
         rw.AddResource("Title", "Classic American Cars");
         rw.AddResource("HeaderString1", "Make");
         rw.AddResource("HeaderString2", "Model");
         rw.AddResource("HeaderString3", "Year");
         rw.AddResource("HeaderString4", "Doors");
         rw.AddResource("HeaderString5", "Cylinders");
         rw.AddResource("Information", SystemIcons.Information);
         rw.AddResource("EarlyAuto1", car1);
         rw.AddResource("EarlyAuto2", car2);
      }
   }
}
```

.resources 파일을 만든 후에는 언어 컴파일러의 `/resource` 스위치를 포함하여 런타임 실행 파일 또는 라이브러리에 포함하거나 어셈블리 링커(Al.exe)를 사용하여 위성 어셈블리에 포함할 수 있습니다.

Visual Studio의 리소스 파일

Visual Studio 프로젝트에 리소스 파일을 추가하면 Visual Studio에서는 프로젝트 디렉터리에 .resx 파일을 만듭니다. Visual Studio에서는 문자열, 이미지 및 이진 개체를 추가할 수 있는 리소스 편집기를 제공합니다. 편집기는 정적 데이터만 처리하도록 디자인되어 있으므로 프로그래밍 개체를 저장하는 데는 사용할 수 없습니다. 프로그래밍 방식으로 개체 데이터를 .resx 파일 또는 .resources 파일에 써야 합니다. 자세한 내용은 프로그래밍 방식으로 .resx 파일 작업 섹션과 .resources 파일의 리소스 섹션을 참조하세요.

지역화된 리소스를 추가할 경우 기본 리소스 파일과 같은 루트 파일 이름을 리소스에 지정해야 합니다. 파일 이름에서 문화권도 지정해야 합니다. 예를 들어 *Resources.resx* 리소스 파일을 추가할 경우 *Resources.en-US.resx* 및 *Resources.fr-FR.resx* 리소스 파일을 만들어 각각 영어(미국) 및 프랑스어(프랑스) 문화권에 대한 지역화된 리소스를 저장할 수도 있습니다. 애플리케이션의 기본 문화권도 지정해야 합니다. 이 문화권의 리소스는 특정 문화권에 대한 지역화된 리소스를 찾을 수 없는 경우 사용됩니다.

Visual Studio의 **솔루션 탐색기**에서 기본 문화권을 지정하려면 다음을 수행합니다.

- 프로젝트 속성을 열고 프로젝트를 마우스 오른쪽 단추로 클릭하고 **속성**을 선택합니다(또는 프로젝트를 선택할 때 `Alt` + `Enter`).
- **패키지** 탭을 선택합니다.
- **일반** 영역의 **어셈블리 중립 언어** 컨트롤에서 적절한 언어/문화권을 선택합니다.
- 변경 내용을 저장합니다.

컴파일 시간에 Visual Studio에서는 먼저 프로젝트의 .resx 파일을 이진 리소스(.resources) 파일로 변환하고 프로젝트 *obj* 디렉터리의 하위 디렉터리에 저장합니다. Visual Studio에서는 지역화된 리소스가 포함되지 않은 모든 리소스 파일을 프로젝트에서 생성된 주 어셈블리에 포함합니다. 리소스 파일에 지역화된 리소스가 포함된 경우 Visual Studio에서는 각 지역화된 문화권에 대한 개별 위성 어셈블리에 리소스 파일을 포함합니다. 그런 다음, 각 위성 어셈블리를 이름이 지역화된 문화권과 일치하는 디렉터리에 저장합니다. 예를 들어 지역화된 영어(미국) 리소스는 en-US 하위 디렉터리의 위성 어셈블리에 저장됩니다.

참고 항목

- System.Resources
- .NET 앱의 리소스
- 리소스 패키징 및 배포

Last updated on 2025. 10. 20.

프로그래밍 방식으로 .resx 파일 작업

> ⓘ **참고 항목**
>
> 이 문서는 .NET Framework에 적용됩니다. .NET 5 이상(.NET Core 포함)에 적용되는 자세한 내용은 **.resx 파일의 리소스**를 참조하세요.

XML 리소스(.resx) 파일은 이름/값 쌍의 데이터 뒤에 특정 스키마를 따라야 하는 헤더를 포함하여 잘 정의된 XML로 구성되어야 하므로 이러한 파일을 수동으로 만드는 것은 오류가 발생하기 쉽습니다. 또는 .NET 클래스 라이브러리의 형식 및 멤버를 사용하여 프로그래밍 방식으로 .resx 파일을 만들 수 있습니다. .NET 클래스 라이브러리를 사용하여 .resx 파일에 저장된 리소스를 검색할 수도 있습니다. 이 문서에서는 네임스페이스의 형식 및 멤버를 System.Resources 사용하여 .resx 파일을 사용하는 방법을 설명합니다.

이 문서에서는 리소스가 포함된 XML(.resx) 파일 작업에 대해 설명합니다. 어셈블리에 포함된 이진 리소스 파일 작업에 대한 자세한 정보는 ResourceManager을 참조하십시오.

> ⚠ **Warning**
>
> 프로그래밍 방식 이외의 .resx 파일로 작업하는 방법도 있습니다. Visual Studio 프로젝트에 리소스 파일을 추가할 때 Visual Studio는 .resx 파일을 만들고 유지 관리하기 위한 인터페이스를 제공하고 컴파일 시간에 .resx 파일을 .resources 파일로 자동으로 변환합니다. 텍스트 편집기를 사용하여 .resx 파일을 직접 조작할 수도 있습니다. 그러나 파일이 손상되지 않도록 하려면 파일에 저장된 이진 정보를 수정하지 않도록 주의해야 합니다.

.resx 파일 만들기

다음 단계에 따라 클래스를 사용하여 System.Resources.ResXResourceWriter 프로그래밍 방식으로 .resx 파일을 만들 수 있습니다.

1. 메서드를 ResXResourceWriter 호출 ResXResourceWriter(String) 하고 .resx 파일의 이름을 제공하여 개체를 인스턴스화합니다. 파일 이름에는 .resx 확장명을 포함해야 합니다. 블록에서 개체를 ResXResourceWriter 인스턴스화하는 경우 3단계에서 `using` 메서드를 ResXResourceWriter.Close 명시적으로 호출할 필요가 없습니다.

2. ResXResourceWriter.AddResource 파일에 추가하려는 각 리소스에 대한 메서드를 호출합니다. 이 메서드의 오버로드를 사용하여 문자열, 개체 및 이진(바이트 배열) 데이터를 추가합니다. 리소스가 개체인 경우 직렬화할 수 있어야 합니다.

3. 메서드를 ResXResourceWriter.Close 호출하여 리소스 파일을 생성하고 모든 리소스를 해제합니다. ResXResourceWriter 개체가 `using` 블록 내에서 생성된 경우, 리소스는 .resx 파일에 기록되고, ResXResourceWriter 개체가 사용하는 리소스는 `using` 블록의 끝에서 해제됩니다.

결과 .resx 파일에는 메서드에서 추가한 각 리소스에 대해 적절한 헤더와 `data` 태그가 있으며, ResXResourceWriter.AddResource 메서드에 의해 추가된 것입니다.

> ⚠ **Warning**
>
> 암호, 보안이 중요한 정보 또는 개인 데이터를 저장할 때는 리소스 파일을 사용하지 마세요.

다음 예제에서는 6개의 문자열, 아이콘 및 두 개의 애플리케이션 정의 개체(두 `Automobile` 개체)를 저장하는 CarResources.resx라는 .resx 파일을 만듭니다. `Automobile` 예제에서 정의되고 인스턴스화된 클래스는 SerializableAttribute 특성이 태그로 지정되어 있습니다.

C#

```
using System;
using System.Drawing;
using System.Resources;

[Serializable()] public class Automobile
{
   private string carMake;
   private string carModel;
   private int carYear;
   private int carDoors;
   private int carCylinders;

   public Automobile(string make, string model, int year) :
                     this(make, model, year, 0, 0)
   { }

   public Automobile(string make, string model, int year,
                     int doors, int cylinders)
   {
      this.carMake = make;
      this.carModel = model;
      this.carYear = year;
      this.carDoors = doors;
      this.carCylinders = cylinders;
   }

   public string Make {
      get { return this.carMake; }
   }

   public string Model {
```

```
      get {return this.carModel; }
   }

   public int Year {
      get { return this.carYear; }
   }

   public int Doors {
      get { return this.carDoors; }
   }

   public int Cylinders {
      get { return this.carCylinders; }
   }
}

public class Example
{
   public static void Main()
   {
      // Instantiate an Automobile object.
      Automobile car1 = new Automobile("Ford", "Model N", 1906, 0, 4);
      Automobile car2 = new Automobile("Ford", "Model T", 1909, 2, 4);
      // Define a resource file named CarResources.resx.
      using (ResXResourceWriter resx = new
ResXResourceWriter(@".\CarResources.resx"))
      {
         resx.AddResource("Title", "Classic American Cars");
         resx.AddResource("HeaderString1", "Make");
         resx.AddResource("HeaderString2", "Model");
         resx.AddResource("HeaderString3", "Year");
         resx.AddResource("HeaderString4", "Doors");
         resx.AddResource("HeaderString5", "Cylinders");
         resx.AddResource("Information", SystemIcons.Information);
         resx.AddResource("EarlyAuto1", car1);
         resx.AddResource("EarlyAuto2", car2);
      }
   }
}
```

> **팁**
>
> Visual Studio를 사용하여 .resx 파일을 만들 수도 있습니다. 컴파일 시 Visual Studio는 **리소스 파일 생성기(Resgen.exe)** 를 사용하여 .resx 파일을 이진 리소스(.resources) 파일로 변환하고 애플리케이션 어셈블리 또는 위성 어셈블리에도 포함합니다.

런타임 실행 파일에 .resx 파일을 포함하거나 위성 어셈블리로 컴파일할 수 없습니다. 리소스 파일 생성기(Resgen.exe)를 사용하여 .resx 파일을 이진 리소스(.resources) 파일로 변환해야 합니다. 그러면 결과 .resources 파일을 애플리케이션 어셈블리 또는 위성 어셈블리에 포함할 수 있습니다. 자세한 내용은 리소스 파일 만들기를 참조하세요.

리소스 열거

경우에 따라 .resx 파일에서 특정 리소스 대신 모든 리소스를 검색할 수 있습니다. 이렇게 하려면 .resx 파일의 System.Resources.ResXResourceReader 모든 리소스에 대한 열거자를 제공하는 클래스를 사용할 수 있습니다. 클래스 System.Resources.ResXResourceReader는 IDictionaryEnumerator을 구현하며, 이 구현은 루프의 각 반복에서 특정 리소스를 나타내는 DictionaryEntry 객체를 반환합니다. 해당 속성은 DictionaryEntry.Key 리소스의 키를 반환하고 해당 DictionaryEntry.Value 속성은 리소스의 값을 반환합니다.

다음 예제에서는 이전 예제에서 만든 CarResources.resx 파일에 대한 ResXResourceReader 개체를 생성하고 리소스 파일을 순회합니다. 리소스 파일에 정의된 두 `Automobile` 개체를 개체에 System.Collections.Generic.List<T> 추가하고 6개의 문자열 중 5개를 개체에 SortedList 추가합니다. 개체의 SortedList 값은 콘솔에 열 머리글을 표시하는 데 사용되는 매개 변수 배열로 변환됩니다. `Automobile` 속성 값도 콘솔에 표시됩니다.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Resources;

public class Example
{
   public static void Main()
   {
      string resxFile = @".\CarResources.resx";
      List<Automobile> autos = new List<Automobile>();
      SortedList headers = new SortedList();

      using (ResXResourceReader resxReader = new ResXResourceReader(resxFile))
      {
         foreach (DictionaryEntry entry in resxReader) {
            if (((string) entry.Key).StartsWith("EarlyAuto"))
               autos.Add((Automobile) entry.Value);
            else if (((string) entry.Key).StartsWith("Header"))
               headers.Add((string) entry.Key, (string) entry.Value);
         }
      }
      string[] headerColumns = new string[headers.Count];
      headers.GetValueList().CopyTo(headerColumns, 0);
      Console.WriteLine("{0,-8} {1,-10} {2,-4}   {3,-5}   {4,-9}\n",
                        headerColumns);
      foreach (var auto in autos)
         Console.WriteLine("{0,-8} {1,-10} {2,4}   {3,5}   {4,9}",
                           auto.Make, auto.Model, auto.Year,
                           auto.Doors, auto.Cylinders);
   }
}
// The example displays the following output:
```

```
//       Make     Model      Year   Doors   Cylinders
//
//       Ford     Model N    1906       0           4
//       Ford     Model T    1909       2           4
```

특정 리소스 검색

.resx 파일의 항목을 열거하는 것 외에도 클래스를 사용하여 이름으로 특정 리소스를 검색할 System.Resources.ResXResourceSet 수 있습니다. 메서드는 ResourceSet.GetString(String) 명명된 문자열 리소스의 값을 검색합니다. 메서드는 ResourceSet.GetObject(String) 명명된 개체 또는 이진 데이터의 값을 검색합니다. 그런 다음 C#으로 캐스팅하거나(Visual Basic에서) 적절한 형식의 개체로 변환해야 하는 개체를 반환합니다.

다음 예제에서는 해당 리소스 이름으로 양식의 캡션 문자열 및 아이콘을 검색합니다. 또한 이전 예제에서 사용된 애플리케이션 정의 `Automobile` 개체를 검색하여 컨트롤에 DataGridView 표시합니다.

C#

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Resources;
using System.Windows.Forms;

public class CarDisplayApp : Form
{
   private const string resxFile = @".\CarResources.resx";
   Automobile[] cars;

   public static void Main()
   {
      CarDisplayApp app = new CarDisplayApp();
      Application.Run(app);
   }

   public CarDisplayApp()
   {
      // Instantiate controls.
      PictureBox pictureBox = new PictureBox();
      pictureBox.Location = new Point(10, 10);
      this.Controls.Add(pictureBox);
      DataGridView grid = new DataGridView();
      grid.Location = new Point(10, 60);
      this.Controls.Add(grid);

      // Get resources from .resx file.
      using (ResXResourceSet resxSet = new ResXResourceSet(resxFile))
      {
         // Retrieve the string resource for the title.
```

```
         this.Text = resxSet.GetString("Title");
         // Retrieve the image.
         Icon image = (Icon) resxSet.GetObject("Information", true);
         if (image != null)
            pictureBox.Image = image.ToBitmap();

         // Retrieve Automobile objects.
         List<Automobile> carList = new List<Automobile>();
         string resName = "EarlyAuto";
         Automobile auto;
         int ctr = 1;
         do {
            auto = (Automobile) resxSet.GetObject(resName + ctr.ToString());
            ctr++;
            if (auto != null)
               carList.Add(auto);
         } while (auto != null);
         cars = carList.ToArray();
         grid.DataSource = cars;
      }
   }
}
```

.resx 파일을 이진 .resources 파일로 변환

.resx 파일을 포함된 이진 리소스(*.resources*) 파일로 변환하면 상당한 이점이 있습니다. .resx 파일은 애플리케이션 개발 중에 쉽게 읽고 유지 관리할 수 있지만 완성된 애플리케이션에는 거의 포함되지 않습니다. 애플리케이션과 함께 배포되는 경우 애플리케이션 실행 파일 및 함께 제공되는 라이브러리와는 별도로 별도의 파일로 존재합니다. 반면. *.resources* 파일은 애플리케이션 실행 파일 또는 함께 제공되는 어셈블리에 포함됩니다. 또한 지역화된 애플리케이션의 경우 런타임에 .resx 파일을 사용하는 경우 개발자가 리소스 대체를 처리해야 합니다. 반면에 포함된 .resources 파일이 포함된 위성 어셈블리 집합이 만들어진 경우 공용 언어 런타임은 리소스 대체 프로세스를 처리합니다.

.resx 파일을 *.resources* 파일로 변환하려면 다음과 같은 기본 구문이 있는 리소스 파일 생성기(*resgen.exe*)를 사용합니다.

콘솔

```
resgen.exe .resxFilename
```

그 결과 .resx 파일 및 .resources 파일 확장명과 루트 파일 이름이 같은 이진 리소스 파일이 생성됩니다. 그런 다음 이 파일을 컴파일 시간에 실행 파일 또는 라이브러리로 컴파일할 수 있습니다. Visual Basic 컴파일러를 사용하는 경우 다음 구문을 사용하여 애플리케이션의 실행 파일에 .resources 파일을 포함합니다.

```
```

콘솔

```
vbc filename .vb -resource: .resourcesFilename
```

C#을 사용하는 경우 구문은 다음과 같습니다.

콘솔

```
 csc filename .cs -resource: .resourcesFilename
```

.resources 파일은 다음과 같은 기본 구문이 있는 어셈블리 링커(*al.exe*)를 사용하여 위성 어셈블리에 포함할 수도 있습니다.

콘솔

```
al resourcesFilename -out: assemblyFilename
```

참고하십시오

- 리소스 파일 만들기
- 리소스 파일 생성기(*resgen.exe*))
- 어셈블리 링커(*al.exe*)

Last updated on 2025. 10. 20.

.NET 앱용 위성 어셈블리 만들기

아티클 • 2023. 04. 08.

리소스 파일은 지역화된 애플리케이션에서 중요한 역할을 합니다. 애플리케이션에서 문자열, 이미지 및 기타 데이터를 사용자의 언어 및 문화권에 표시하고 사용자의 언어 또는 문화권에 대한 리소스를 사용할 수 없는 경우 대체 데이터를 제공할 수 있습니다. .NET은 허브 및 스포크 모델을 사용하여 지역화된 리소스를 찾고 검색합니다. 허브는 지역화할 수 없는 실행 코드와 중립 또는 기본 문화권이라고 하는 단일 문화권의 리소스를 포함하는 주 어셈블리입니다. 기본 문화권은 애플리케이션의 대체 문화권입니다. 지역화된 리소스를 사용할 수 없는 경우에 사용됩니다. NeutralResourcesLanguageAttribute 특성을 사용하여 애플리케이션의 기본 문화권의 문화를 지정합니다. 각 스포크는 지역화된 단일 문화권의 리소스를 포함하지만 코드는 포함하지 않는 위성 어셈블리에 연결됩니다. 위성 어셈블리는 기본 어셈블리의 일부가 아니므로 애플리케이션의 기본 어셈블리를 바꾸지 않고 특정 문화권에 해당하는 리소스를 쉽게 업데이트하거나 바꿀 수 있습니다.

> ⓘ **참고**
>
> 애플리케이션의 기본 문화권의 리소스를 위성 어셈블리에 저장할 수도 있습니다. 그러려면 **NeutralResourcesLanguageAttribute** 특성에 **UltimateResourceFallbackLocation.Satellite**의 값을 할당합니다.

위성 어셈블리의 이름 및 위치

허브 및 스포크 모델을 사용하려면 리소스를 쉽게 찾아서 사용할 수 있도록 특정 위치에 두어야 합니다. 리소스를 예상대로 컴파일하고 이름을 지정하지 않거나 올바른 위치에 배치하지 않으면 공용 언어 런타임에서 리소스를 찾을 수 없으며 기본 문화권의 리소스를 대신 사용합니다. .NET 리소스 관리자는 형식으로 ResourceManager 표시되며 지역화된 리소스에 자동으로 액세스하는 데 사용됩니다. 리소스 관리자에는 다음이 필요합니다.

- 단일 위성 어셈블리는 특정 문화권에 대한 모든 리소스를 포함해야 합니다. 즉, 여러 *.txt* 또는 *.resx* 파일을 단일 이진 *.resources* 파일로 컴파일해야 합니다.
- 해당 문화권의 리소스를 저장하는 각 지역화된 문화권에 대한 애플리케이션 디렉터리에 있는 별도의 하위 디렉터리가 있어야 합니다. 하위 디렉터리 이름은 문화권 이름과 동일해야 합니다. 또는 위성 어셈블리를 전역 어셈블리 캐시에 저장할 수 있습니다. 이 경우 어셈블리의 강력한 이름의 문화권 정보 구성 요소는 해당 문화권을 나타내야 합니다. 자세한 내용은 전역 어셈블리 캐시에 위성 어셈블리 설치를 참조하세요.

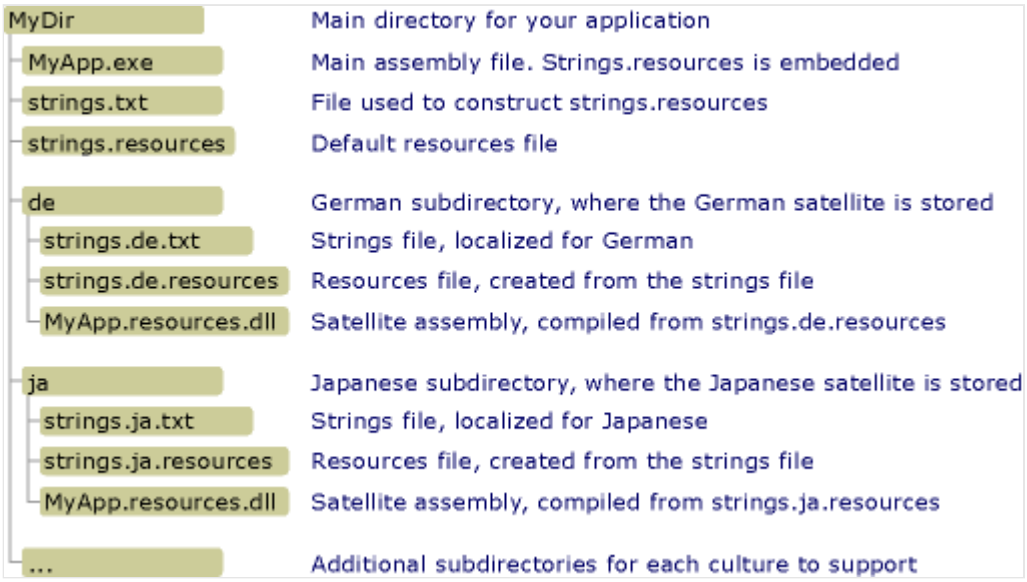
> ⓘ 참고
>
> 애플리케이션에 하위 문화권의 리소스가 들어 있는 경우 각 하위 문화권을 애플리케이션 디렉터리 아래 별도의 하위 디렉터리에 둡니다. 주 문화권의 디렉터리 아래 하위 디렉터리에 하위 문화권을 두면 안 됩니다.

- 위성 어셈블리는 애플리케이션과 동일한 이름을 가져야 하고 ".resources.dll"을 파일 이름 확장명으로 사용해야 합니다. 예를 들어 애플리케이션의 이름이 *Example.exe*인 경우, 각 위성 어셈블리의 이름은 *Example.resources.dll*이어야 합니다. 위성 어셈블리 이름은 리소스 파일의 문화권을 나타내지 않습니다. 그러나, 위성 어셈블리는 문화권을 지정하는 디렉터리에 나타납니다.

- 위성 어셈블리의 문화권에 대한 정보는 어셈블리의 메타데이터에 포함되어야 합니다. 문화권 이름을 위성 어셈블리의 메타데이터에 저장하려면 어셈블리 링커를 사용하여 리소스를 위성 어셈블리에 포함할 때 `/culture` 옵션을 지정합니다.

다음 그림에서는 전역 어셈블리 캐시에 설치하지 않는 애플리케이션에 대한 샘플 디렉터리 구조 및 위치 요구 사항을 보여 줍니다. *.txt* 및 *.resources 확장이 있는* 항목은 최종 애플리케이션과 함께 제공되지 않습니다. 이러한 중간 리소스 파일은 최종 위성 리소스 어셈블리를 만드는 데 사용됩니다. 이 예제에서는 *.resx* 파일을 *.txt* 파일로 대체할 수 있습니다. 자세한 내용은 리소스 패키징 및 배포를 참조하세요.

다음 이미지는 위성 어셈블리 디렉터리를 보여 줍니다.



위성 어셈블리 컴파일

리소스 파일 생성기(*resgen.exe*)를 사용하여 리소스를 포함하는 텍스트 파일 또는 XML(*.resx*) 파일을 이진 *.resources* 파일로 컴파일합니다. 그런 다음, 어셈블리 링커(*al.exe*)를 사용하여 *.resources* 파일을 위성 어셈블리로 컴파일합니다. *al.exe*는 지정한 *.resources* 파일

에서 어셈블리를 만듭니다. 위성 어셈블리는 리소스만 포함할 수 있습니다. 실행 코드를 포함할 수 없습니다.

다음 *al.exe* 명령은 독일 리소스 파일 *strings.de.resources*에서 `Example` 애플리케이션에 대한 위성 어셈블리를 만듭니다.

콘솔

```
al -target:lib -embed:strings.de.resources -culture:de -
out:Example.resources.dll
```

다음 *al.exe* 명령은 파일 *strings.de.resources*에서 `Example` 애플리케이션에 대한 위성 어셈블리도 만듭니다. **/template** 옵션을 사용하면 위성 어셈블리가 부모 어셈블리(*Example.dll*)로부터 문화권 관련 정보를 제외한 모든 어셈블리 메타데이터를 상속합니다.

콘솔

```
al -target:lib -embed:strings.de.resources -culture:de -
out:Example.resources.dll -template:Example.dll
```

다음 표에서는 이러한 명령에 사용된 *al.exe* 옵션을 더 자세히 설명합니다.

옵션	설명
`-target:lib`	위성 어셈블리가 라이브러리(.dll) 파일로 컴파일되도록 지정합니다. 위성 어셈블리는 실행 코드를 포함하지 않으며 애플리케이션의 기본 어셈블리가 아니므로 위성 어셈블리를 DLL로 저장해야 합니다.
`-embed:strings.de.resources`	*al.exe*가 어셈블리를 컴파일할 때 포함할 리소스 파일의 이름을 지정합니다. 위성 어셈블리에 여러 개의 .resources 파일을 포함할 수 있지만, 허브 및 스포크 모델을 따르는 경우, 각 문화권에 대해 하나의 위성 어셈블리를 컴파일해야 합니다. 그러나 문자열 및 개체에 대해 별도의 .resources 파일을 만들 수 있습니다.
`-culture:de`	컴파일한 리소스의 문화권을 지정합니다. 공용 언어 런타임은 지정된 문화권의 리소스를 검색할 때 이 정보를 사용합니다. 이 옵션을 생략하면 *al.exe* 여전히 리소스를 컴파일하지만 사용자가 요청할 때 런타임에서 찾을 수 없습니다.
`-out:Example.resources.dll`	출력 파일의 이름을 지정합니다. 이 이름은 명명 표준 *baseName*.resources.*extension*을 따라야 합니다. 여기서 *baseName*은 주 어셈블리의 이름이고 *extension*은 유효한 파일 이름 확장명(예: .dll)입니다. 런타임은 출력 파일 이름을 기반으로 위성 어셈블리의 문화권을 확인할 수 없습니다. **/culture** 옵션을 사용하여 지정해야 합니다.

옵션	설명
`-template:Example.dll`	위성 어셈블리가 culture 필드를 제외하고 모든 어셈블리 메타데이터를 상속할 어셈블리를 지정합니다. 이 옵션은 강력한 이름을 가진 어셈블리를 지정할 때에만 위성 어셈블리에 영향을 미칩니다.

*al.exe*에서 사용할 수 있는 전체 옵션 목록을 보려면 어셈블리 링커(*al.exe*)를 참조하세요.

> **참고**
>
> .NET Framework 대상으로 지정하더라도 .NET Core MSBuild 작업을 사용하여 위성 어셈블리를 컴파일하려는 경우가 있을 수 있습니다. 예를 들어 C# 컴파일러 **결정적** 옵션을 사용하여 여러 빌드의 어셈블리를 비교할 수 있습니다. 이 경우 *.csproj* 파일에서 **GenerateSatelliteAssembliesForCore** `true` 를 로 설정하여 **Al.exe(어셈블리 링커)** 대신 *csc.exe* 사용하여 위성 어셈블리를 생성합니다.
>
> XML
>
> ```
> <Project>
>     <PropertyGroup>
>
> <GenerateSatelliteAssembliesForCore>true</GenerateSatelliteAssembliesFo
> rCore>
>     </PropertyGroup>
> </Project>
> ```
>
> .NET Core MSBuild 작업은 기본적으로 *al.exe* 대신 *csc.exe* 사용하여 위성 어셈블리를 생성합니다. 자세한 내용은 **"Core" 위성 어셈블리 생성을 더 쉽게 옵트인할 수 있도록** 을 참조하세요.

위성 어셈블리: 예

다음은 지역화된 인사말이 들어 있는 메시지 상자를 표시하는 간단한 "Hello world" 예제입니다. 이 예제에서는 영어(미국), 프랑스어(프랑스) 및 러시아어(러시아) 문화권에 대한 리소스가 포함되고 해당 대체 문화권은 영어입니다. 예제를 만들려면 다음을 수행합니다.

1. 기본 문화권의 리소스를 포함하는 *Greeting.resx* 또는 *Greeting.txt*라는 리소스 파일을 만듭니다. 이 파일에 값이 "Hello world!"인 라는 `HelloString` 단일 문자열을 저장합니다.

2. 영어(en)가 애플리케이션의 기본 문화권임을 나타내려면 다음 System.Resources.NeutralResourcesLanguageAttribute 특성을 애플리케이션의

AssemblyInfo 파일 또는 애플리케이션의 주 어셈블리로 컴파일할 주 소스 코드 파일에 추가합니다.

```
C#

[assembly: NeutralResourcesLanguage("en")]
```

3. 다음과 같이 애플리케이션에 추가 문화권(`en-US`, `fr-FR`, `ru-RU`)에 대한 지원을 추가합니다.

   - `en-US` 또는 영어(미국) 문화권을 지원하려면 *Greeting.en-US.resx* 또는 *Greeting.en-US.txt*라는 리소스 파일을 만들어 값이 "Hi world!"인 `HelloString` 이라는 단일 문자열에 저장합니다.
   - `fr-FR` 또는 프랑스어(프랑스) 문화권을 지원하려면 *Greeting.fr-FR.resx* 또는 *Greeting.fr-FR.txt*라는 리소스 파일을 만들어 값이 "Salut tout le monde!"인 `HelloString`이라는 단일 문자열에 저장합니다.
   - `ru-RU` 또는 러시아어(러시아) 문화권을 지원하려면 *Greeting.ru-RU.resx* 또는 *Greeting.ru-RU.txt*라는 리소스 파일을 만들어 값이 "Всем привет!"인 `HelloString`이라는 단일 문자열에 저장합니다.

4. *resgen.exe*를 사용하여 각 텍스트 또는 XML 리소스 파일을 이진 *.resources* 파일로 컴파일합니다. 출력은 *.resx* 또는 *.txt* 파일과 동일한 루트 파일 이름을 가지고 있지만 *.resources* 확장명과는 다른 파일 세트입니다. Visual Studio를 사용하여 예제를 만들면 컴파일 프로세스는 자동으로 처리됩니다. Visual Studio를 사용하지 않는 경우, 다음 명령을 실행하여 *.resx* 파일을 *.resources* 파일로 컴파일합니다.

```
콘솔

resgen Greeting.resx
resgen Greeting.en-us.resx
resgen Greeting.fr-FR.resx
resgen Greeting.ru-RU.resx
```

   리소스가 XML 파일 대신 텍스트 파일에 있는 경우 *.resx* 확장명을 *.txt*로 바꿉니다.

5. 다음 소스 코드를 기본 문화권에 대한 리소스와 함께 애플리케이션의 주 어셈블리로 컴파일합니다.

> **ⓘ 중요**

Visual Studio 대신 명령줄을 사용하여 예제를 만들 경우 ResourceManager 클래스 생성자에 대한 호출을 `ResourceManager rm = new ResourceManager("Greeting", typeof(Example).Assembly);`와 같이 수정해야 합니다.

C#

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;
using System.Windows.Forms;

class Example
{
   static void Main()
   {
      // Create array of supported cultures
      string[] cultures = {"en-CA", "en-US", "fr-FR", "ru-RU"};
      Random rnd = new Random();
      int cultureNdx = rnd.Next(0, cultures.Length);
      CultureInfo originalCulture =
Thread.CurrentThread.CurrentCulture;

      try {
         CultureInfo newCulture = new
CultureInfo(cultures[cultureNdx]);
         Thread.CurrentThread.CurrentCulture = newCulture;
         Thread.CurrentThread.CurrentUICulture = newCulture;
         ResourceManager rm = new ResourceManager("Example.Greeting",

typeof(Example).Assembly);
         string greeting = String.Format("The current culture is
{0}.\n{1}",

Thread.CurrentThread.CurrentUICulture.Name,
                                         rm.GetString("HelloString"));

         MessageBox.Show(greeting);
      }
      catch (CultureNotFoundException e) {
         Console.WriteLine("Unable to instantiate culture {0}",
e.InvalidCultureName);
      }
      finally {
         Thread.CurrentThread.CurrentCulture = originalCulture;
         Thread.CurrentThread.CurrentUICulture = originalCulture;
      }
   }
}
```

애플리케이션 이름이 **Example** 이고 명령줄에서 컴파일하는 경우 C# 컴파일러에 대한 명령은 다음과 같습니다.

```
콘솔

csc Example.cs -res:Greeting.resources
```

해당 Visual Basic 컴파일러 명령은 다음과 같습니다.

```
콘솔

vbc Example.vb -res:Greeting.resources
```

6. 애플리케이션에서 지원되는 각 지역화된 문화권에 대한 주 애플리케이션 디렉터리에서 하위 디렉터리를 만듭니다. *en-US*, *fr-FR* 및 *ru-RU* 하위 디렉터리를 만들어야 합니다. Visual Studio는 컴파일 프로세스의 일부로 이러한 하위 디렉터리를 자동으로 만듭니다.

7. 각 문화권별 *.resources* 파일을 위성 어셈블리로 포함하고 해당 디렉터리에 저장합니다. 각 *.resources* 파일에 대해 이 작업을 수행하는 명령은 다음과 같습니다.

```
콘솔

al -target:lib -embed:Greeting.culture.resources -culture:culture -
out:culture\Example.resources.dll
```

여기서 *culture*는 리소스가 위성 어셈블리에 포함된 문화권의 이름입니다. Visual Studio는 이 프로세스를 자동으로 처리합니다.

그런 다음 예제를 실행할 수 있습니다. 지원되는 문화권 중 하나를 현재 문화권으로 임의로 만들고 지역화된 인사말을 표시합니다.

전역 어셈블리 캐시에 위성 어셈블리 설치

로컬 애플리케이션 하위 디렉터리에 어셈블리를 설치하는 대신, 전역 어셈블리 캐시에 설치할 수 있습니다. 이는 클래스 라이브러리 및 클래스 라이브러리 리소스 어셈블리가 여러 애플리케이션에서 사용되는 경우 특히 유용합니다.

전역 어셈블리 캐시에 어셈블리를 설치하려면 강력한 이름이 필요합니다. 강력한 이름의 어셈블리는 유효한 퍼블릭/프라이빗 키 쌍으로 서명됩니다. 바인딩 요청을 만족시키는 데 사용할 어셈블리를 결정하기 위해 런타임에서 사용하는 버전 정보가 포함됩니다. 강력한 이름과 버전 관리에 대한 자세한 내용은 어셈블리 버전 관리를 참조하세요. 강력한 이름에 대한 자세한 내용은 강력한 이름의 어셈블리를 참조하세요.

애플리케이션을 개발할 때 최종 퍼블릭/프라이빗 키 쌍에 액세스할 가능성은 거의 없습니다. 전역 어셈블리 캐시에 위성 어셈블리를 설치하고 예상대로 작동하는지 확인하려면 지연된 서명이라는 기술을 사용할 수 있습니다. 어셈블리 서명을 연기하면 빌드할 때 강력한 이름 시그니처에 사용할 파일 공간이 예약됩니다. 나중에 최종 퍼블릭/프라이빗 키 쌍을 사용할 수 있을 때까지 실제 서명이 연기됩니다. 지연된 서명에 대한 자세한 내용은 어셈블리 서명 연기를 참조하세요.

퍼블릭 키 얻기

어셈블리 서명을 연기하려면 공개 키에 대한 액세스 권한이 있어야 합니다. 최종 서명을 수행하는 회사 내 조직으로부터 실제 퍼블릭 키를 얻거나 강력한 이름 도구(*sn.exe*)를 사용하여 공개 키를 만들 수 있습니다.

다음 *Sn.exe* 명령으로 테스트 퍼블릭/프라이빗 키 쌍을 만듭니다. **–k** 옵션은 *Sn.exe*가 새 키 쌍을 만들고 *TestKeyPair.snk*라는 파일에 저장하도록 지정합니다.

콘솔

```
sn –k TestKeyPair.snk
```

테스트 키 쌍을 포함하는 파일에서 공개 키를 추출할 수 있습니다. 다음 명령은 *TestKeyPair.snk*에서 공개 키를 추출하고 *PublicKey.snk*에 저장합니다.

콘솔

```
sn –p TestKeyPair.snk PublicKey.snk
```

어셈블리 서명 연기

퍼블릭 키를 얻었거나 만든 후에는 어셈블리 링커(*al.exe*)를 사용하여 어셈블리를 컴파일하고 지연된 서명을 지정합니다.

다음 *al.exe* 명령은 *strings.ja.resources* 파일에서 StringLibrary 애플리케이션에 대한 강력한 이름의 위성 어셈블리를 만듭니다.

콘솔

```
al -target:lib -embed:strings.ja.resources -culture:ja -
out:StringLibrary.resources.dll -delay+ -keyfile:PublicKey.snk
```

-delay+ 옵션은 어셈블리 링커가 어셈블리 서명을 연기하도록 지정합니다. **-keyfile** 옵션은 어셈블리 서명을 연기하는 데 사용할 공개 키를 포함하는 키 파일의 이름을 지정합니

다.

어셈블리 다시 서명

애플리케이션을 배포하기 전에, 실제 키 쌍으로 지연 서명된 위성 어셈블리를 다시 서명해야 합니다. *Sn.exe*를 사용하여 이 작업을 수행할 수 있습니다.

다음 *Sn.exe* 명령은 *RealKeyPair.snk* 파일에 저장된 실제 키 쌍으로 *StringLibrary.resources.dll*에 서명합니다. **–R** 옵션은 이전에 서명했거나 지연 서명된 어셈블리에 다시 서명하도록 지정합니다.

콘솔

```
sn –R StringLibrary.resources.dll RealKeyPair.snk
```

전역 어셈블리 캐시에 위성 어셈블리 설치

런타임이 리소스 대체 프로세스에서 리소스를 검색할 때, 가장 먼저 전역 어셈블리 캐시에서 찾습니다. (자세한 내용은 패키지 및 리소스 배포의 "리소스 대체 프로세스" 섹션을 참조하세요.) 위성 어셈블리가 강력한 이름으로 서명되는 즉시 전역 어셈블리 캐시 도구(*gacutil.exe*)를 사용하여 전역 어셈블리 캐시에 설치할 수 있습니다.

다음 *Gacutil.exe* 명령은 *StringLibrary.resources.dll**을 글로벌 어셈블리 캐시에 설치합니다.

콘솔

```
gacutil -i:StringLibrary.resources.dll
```

/i 옵션은 *Gacutil.exe*가 지정된 어셈블리를 글로벌 어셈블리 캐시에 설치하도록 지정합니다. 위성 어셈블리가 캐시에 설치된 후, 포함된 리소스는 위성 어셈블리를 사용하도록 설계된 모든 애플리케이션에서 사용할 수 있게 됩니다.

글로벌 어셈블리 캐시의 리소스: 예제

다음 예제에서는 .NET 클래스 라이브러리의 메서드를 사용하여 리소스 파일에서 지역화된 인사말을 추출하고 반환합니다. 라이브러리 및 리소스를 전역 어셈블리 캐시에 등록합니다. 예제에는 영어(미국), 프랑스어(프랑스), 러시아어(러시아) 및 영미 문화권에 대한 리소스가 포함됩니다. 영어는 기본 문화권으로, 해당 리소스는 주 어셈블리에 저장됩니다. 이 예제에서는 처음에 라이브러리와 해당 위성 어셈블리를 공개 키로 지연 서명한 다음 퍼블릭/프라이빗 키 쌍으로 다시 서명합니다. 예제를 만들려면 다음을 수행합니다.

1. Visual Studio를 사용하지 않는 경우 다음 강력한 이름 도구(Sn.exe) 명령을 사용하여 *ResKey.snk*라는 퍼블릭/프라이빗 키 쌍을 만듭니다.

   콘솔

   ```
   sn -k ResKey.snk
   ```

   Visual Studio를 사용하는 경우 프로젝트 **속성** 대화 상자의 **서명** 탭을 사용하여 키 파일을 생성합니다.

2. 다음 강력한 이름 도구(Sn.exe) 명령을 사용하여 *PublicKey.snk*라는 공개 키 파일을 만듭니다.

   콘솔

   ```
   sn -p ResKey.snk PublicKey.snk
   ```

3. 기본 문화권의 리소스를 포함하는 *Strings.resx*라는 리소스 파일을 만듭니다. 해당 파일에 값이 "어떻게 합니까?" 라는 단일 `Greeting` 문자열을 저장합니다.

4. "en"가 애플리케이션의 기본 문화권임을 나타내려면 다음 System.Resources.NeutralResourcesLanguageAttribute 특성을 애플리케이션의 AssemblyInfo 파일 또는 애플리케이션의 주 어셈블리로 컴파일할 주 소스 코드 파일에 추가합니다.

   C#

   ```
   [assembly:NeutralResourcesLanguageAttribute("en")]
   ```

5. 추가 문화권(en-US, fr-FR 및 ru-RU 문화권)에 대한 지원을 애플리케이션에 다음과 같이 추가합니다.

   - "en-US" 또는 영어(미국) 문화권을 지원하려면 *Strings.en-US.resx* 또는 *Strings.en-US.txt*라는 리소스 파일을 만들어 값이 "Hello!"인 `Greeting`라는 단일 문자열에 저장합니다.

   - "fr-FR" 또는 프랑스어(프랑스) 문화권을 지원하려면 *Strings.fr-FR.resx* 또는 *Strings.fr-FR.txt*라는 리소스 파일을 만들어 값이 "Bon jour!"인 `Greeting`라는 단일 문자열에 저장합니다.

   - "ru-RU" 또는 러시아어(러시아) 문화권을 지원하려면 *Strings.ru-RU.resx* 또는 *Strings.ru-RU.txt*라는 리소스 파일을 만들어 값이 "Привет!"인 `Greeting`라는 단일 문자열에 저장합니다.

6. *resgen.exe*를 사용하여 각 텍스트 또는 XML 리소스 파일을 이진 .resources 파일로 컴파일합니다. 출력은 *.resx* 또는 *.txt* 파일과 동일한 루트 파일 이름을 가지고 있지만 *.resources* 확장명과는 다른 파일 세트입니다. Visual Studio를 사용하여 예제를 만들면 컴파일 프로세스는 자동으로 처리됩니다. Visual Studio를 사용하지 않는 경우 다음 명령을 실행하여 *.resx* 파일을 *.resources* 파일로 컴파일합니다.

```
콘솔

resgen filename
```

여기서 *filename*은 선택적 경로, 파일 이름 및 *.resx* 또는 텍스트 파일의 확장명입니다.

7. *StringLibrary.vb* 또는 *StringLibrary.cs*에 대한 다음 소스 코드를 기본 문화권의 리소스와 함께 *StringLibrary.dll*이라는 지연 서명된 라이브러리 어셈블리로 컴파일합니다.

> **ⓘ 중요**
>
> Visual Studio 대신 명령줄을 사용하여 예제를 만들 경우 **ResourceManager** 클래스 생성자에 대한 호출을 `ResourceManager rm = new ResourceManager("Strings", typeof(Example).Assembly);`로 수정해야 합니다.

```
C#

using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;

[assembly:NeutralResourcesLanguageAttribute("en")]

public class StringLibrary
{
   public string GetGreeting()
   {
      ResourceManager rm = new ResourceManager("Strings",

Assembly.GetAssembly(typeof(StringLibrary)));
      string greeting = rm.GetString("Greeting");
      return greeting;
   }
}
```

C# 컴파일러에 대한 명령은 다음과 같습니다.

콘솔

```
csc -t:library -resource:Strings.resources -delaysign+ -
keyfile:publickey.snk StringLibrary.cs
```

해당 Visual Basic 컴파일러 명령은 다음과 같습니다.

콘솔

```
vbc -t:library -resource:Strings.resources -delaysign+ -
keyfile:publickey.snk StringLibrary.vb
```

8. 애플리케이션에서 지원되는 각 지역화된 문화권에 대한 주 애플리케이션 디렉터리에서 하위 디렉터리를 만듭니다. *en-US*, *fr-FR* 및 *ru-RU* 하위 디렉터리를 만들어야 합니다. Visual Studio는 컴파일 프로세스의 일부로 이러한 하위 디렉터리를 자동으로 만듭니다. 모든 위성 어셈블리의 파일 이름은 동일하므로 하위 디렉터리가 공용/프라이빗 키 쌍으로 서명될 때까지 개별 문화권별 위성 어셈블리를 저장하는 데 사용됩니다.

9. 각 문화권별 *.resources* 파일을 지연 서명된 위성 어셈블리로 포함하고 해당 디렉터리에 저장합니다. 각 *.resources* 파일에 대해 이 작업을 수행하는 명령은 다음과 같습니다.

콘솔

```
al -target:lib -embed:Strings.culture.resources -culture:culture -
out:culture\StringLibrary.resources.dll -delay+ -keyfile:publickey.snk
```

여기서 *culture*는 문화권의 이름입니다. 이 예제에서 문화권 이름은 en-US, fr-FR 및 ru-RU입니다.

10. 다음과 같이 강력한 이름 도구(*sn.exe*)를 사용하여 *StringLibrary.dll*에 다시 서명합니다.

콘솔

```
sn –R StringLibrary.dll RealKeyPair.snk
```

11. 개별 위성 어셈블리에 다시 서명합니다. 이를 위해 각 위성 어셈블리에 대해 강력한 이름 도구(*sn.exe*)를 다음과 같이 사용합니다.

콘솔

```
sn –R StringLibrary.resources.dll RealKeyPair.snk
```

12. 다음 명령을 사용하여 *StringLibrary.dll* 및 글로벌 어셈블리 캐시에 있는 각 위성 어셈블리를 등록합니다.

```
콘솔

gacutil -i filename
```

여기서 *filename*은 등록할 파일의 이름입니다.

13. Visual Studio를 사용하는 경우 라는 `Example`새 **콘솔 애플리케이션** 프로젝트를 만들고 *StringLibrary.dll* 및 다음 소스 코드에 대한 참조를 추가하고 컴파일합니다.

```
C#

using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-GB", "en-US", "fr-FR", "ru-RU" };
      Random rnd = new Random();
      string cultureName = cultureNames[rnd.Next(0,
cultureNames.Length)];
      Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture(cultureName);
      Console.WriteLine("The current UI culture is {0}",
                        Thread.CurrentThread.CurrentUICulture.Name);
      StringLibrary strLib = new StringLibrary();
      string greeting = strLib.GetGreeting();
      Console.WriteLine(greeting);
   }
}
```

명령줄에서 컴파일하려면 C# 컴파일러에서 다음 명령을 사용합니다.

```
콘솔

csc Example.cs -r:StringLibrary.dll
```

Visual Basic 컴파일러의 명령줄은 다음과 같습니다.

```
콘솔

vbc Example.vb -r:StringLibrary.dll
```

14. *Example.exe*를 실행합니다.

참조

- 리소스 패키징 및 배포
- 어셈블리 서명 연기
- *al.exe*(어셈블리 링커)
- *sn.exe*(강력한 이름 도구)
- *gacutil.exe*(전역 어셈블리 캐시 도구)
- .NET의 리소스

.NET 앱에서 리소스 패키지 및 배포

2025. 06. 17.

애플리케이션은 클래스가 나타내는 .NET Framework Resource Manager를 ResourceManager 사용하여 지역화된 리소스를 검색합니다. Resource Manager는 허브 및 스포크 모델을 사용하여 리소스를 패키지하고 배포한다고 가정합니다. 허브는 중립 또는 기본 문화권이라는 단일 문화권에 대한 리소스 및 로컬화할 수 없는 실행 코드가 포함된 주 어셈블리입니다. 기본 문화권은 애플리케이션에 대한 대체 문화권입니다. 지역화된 리소스를 찾을 수 없는 경우 리소스가 사용되는 문화권입니다. 각 스포크는 단일 문화권에 대한 리소스를 포함하지만 코드를 포함하지 않는 위성 어셈블리에 연결합니다.

이 모델에는 다음과 같은 몇 가지 이점이 있습니다.

- 애플리케이션을 배포한 후 새 문화권에 대한 리소스를 증분 방식으로 추가할 수 있습니다. 문화권별 리소스의 후속 개발에는 상당한 시간이 필요할 수 있으므로 주 애플리케이션을 먼저 릴리스하고 나중에 문화권별 리소스를 제공할 수 있습니다.
- 애플리케이션을 다시 컴파일하지 않고도 애플리케이션의 위성 어셈블리를 업데이트하고 변경할 수 있습니다.
- 애플리케이션은 특정 문화권에 필요한 리소스를 포함하는 위성 어셈블리만 로드해야 합니다. 이렇게 하면 시스템 리소스의 사용을 크게 줄일 수 있습니다.

그러나 이 모델에는 다음과 같은 단점도 있습니다.

- 여러 리소스 집합을 관리해야 합니다.
- 여러 구성을 테스트해야 하므로 애플리케이션을 테스트하는 초기 비용이 증가합니다. 장기적으로는 여러 병렬 국제 버전을 테스트하고 유지하는 것보다 여러 위성으로 하나의 핵심 애플리케이션을 테스트하는 것이 더 쉽고 저렴합니다.

리소스 명명 규칙

애플리케이션의 리소스를 패키지할 때 공용 언어 런타임에서 예상하는 리소스 명명 규칙을 사용하여 이름을 지정해야 합니다. 런타임은 해당 문화권 이름으로 리소스를 식별합니다. 각 문화권에는 고유한 이름이 지정되며, 이는 일반적으로 언어와 연결된 소문자 문화권 이름과 필요한 경우 국가 또는 지역과 연결된 대문자 하위 문화권 이름 2자로 된 조합입니다. 하위 문화권 이름은 대시(-)로 구분된 문화권 이름을 따릅니다. 예를 들어 일본어로는 일본어에 대한 ja-JP, 미국에서 사용되는 영어 en-US, 독일에서 사용되는 독일어 de-DE, 오스트리아어로 de-AT 등이 있습니다. **Windows에서 지원하는 언어/지역 이름 목록에서 언어**태그 열을 참조하세요. 문화권 이름은 BCP 47 에 정의된 표준을 따릅니다.

> **참고**

중국어(간체)와 같이 `zh-Hans` 두 글자 문화권 이름에 대한 몇 가지 예외가 있습니다.

자세한 내용은 리소스 파일 만들기 및 위성 어셈블리 만들기를 참조하세요.

리소스 대체 프로세스

리소스 패키징 및 배포를 위한 허브 및 스포크 모델은 대체 프로세스를 사용하여 적절한 리소스를 찾습니다. 애플리케이션에서 사용할 수 없는 지역화된 리소스를 요청하는 경우 공용 언어 런타임은 문화권 계층에서 사용자의 애플리케이션 요청과 가장 밀접하게 일치하는 적절한 대체 리소스를 검색하고, 예외를 최후의 수단으로만 throw합니다. 계층의 각 수준에서 적절한 리소스가 발견되면 런타임에서 사용합니다. 리소스를 찾을 수 없는 경우 검색은 다음 수준에서 계속됩니다.

조회 성능을 향상시키려면 주 어셈블리에 특성을 적용 NeutralResourcesLanguageAttribute 하고 주 어셈블리에서 작동하는 중립 언어의 이름을 전달합니다.

.NET Framework 리소스 대체 프로세스

.NET Framework 리소스 대체 프로세스에는 다음 단계가 포함됩니다.

> **팁**
>
> relativeBindForResources< 구성 요소를 사용하여> 리소스 대체 프로세스 및 런타임이 리소스 어셈블리에 대해 프로브하는 프로세스를 최적화할 수 있습니다. 자세한 내용은 **리소스 대체 프로세스 최적화를 참조하세요**.

1. 런타임은 먼저 애플리케이션에 대해 요청된 문화권과 일치하는 어셈블리에 대한 전역 어셈블리 캐시 를 확인합니다.

   전역 어셈블리 캐시는 많은 애플리케이션에서 공유하는 리소스 어셈블리를 저장할 수 있습니다. 이렇게 하면 만드는 모든 애플리케이션의 디렉터리 구조에 특정 리소스 집합을 포함할 필요가 없습니다. 런타임이 어셈블리에 대한 참조를 찾으면 어셈블리에서 요청된 리소스를 검색합니다. 어셈블리에서 항목을 찾으면 요청된 리소스를 사용합니다. 항목을 찾지 못하면 검색을 계속합니다.

2. 다음으로 런타임은 요청된 문화권과 일치하는 하위 디렉터리에 대해 현재 실행 중인 어셈블리의 디렉터리를 확인합니다. 하위 디렉터리를 찾으면 해당 하위 디렉터리에서 요청된 문화권에 대한 유효한 위성 어셈블리를 검색합니다. 그런 다음 런타임은 위성 어셈블리에서 요청된 리소스를 검색합니다. 어셈블리에서 리소스를 찾으면 해당 리소스를 사용합니다. 리소스를 찾지 못하면 검색을 계속합니다.

3. 다음 런타임은 Windows Installer를 쿼리하여 요청 시 위성 어셈블리를 설치할지 여부를 확인합니다. 이 경우 설치를 처리하고, 어셈블리를 로드하고, 어셈블리 또는 요청된 리소스를 검색합니다. 어셈블리에서 리소스를 찾으면 해당 리소스를 사용합니다. 리소스를 찾지 못하면 검색을 계속합니다.

4. 런타임은 위성 어셈블리를 AppDomain.AssemblyResolve 찾을 수 없음을 나타내기 위해 이벤트를 발생시킵니다. 이벤트를 처리하도록 선택하는 경우 이벤트 처리기는 리소스가 조회에 사용되는 위성 어셈블리에 대한 참조를 반환할 수 있습니다. 그렇지 않으면 이벤트 처리기가 반환 `null` 되고 검색이 계속됩니다.

5. 다음으로 런타임은 전역 어셈블리 캐시를 다시 검색합니다. 이번에는 요청된 문화권의 부모 어셈블리를 검색합니다. 부모 어셈블리가 전역 어셈블리 캐시에 있는 경우 런타임은 어셈블리에서 요청된 리소스를 검색합니다.

   부모 문화권은 적절한 대체 문화권으로 정의됩니다. 리소스를 제공하는 것이 예외를 throw하는 것보다 낫기 때문에 부모를 백업 후보로 고려하세요. 이 프로세스를 통해 리소스를 다시 사용할 수도 있습니다. 자식 문화권이 요청된 리소스를 지역화할 필요가 없는 경우에만 부모 수준에서 특정 리소스를 포함해야 합니다. 예를 들어, `en` (중립 영어), `en-GB` (영국에서 사용되는 영어), `en-US` (미국에서 사용되는 영어) 위성 어셈블리를 제공하는 경우 `en` 위성에는 공통 용어가 포함되고, `en-GB` 및 `en-US` 위성은 다른 용어에 대해서만 재정의를 제공할 수 있습니다.

6. 다음으로 런타임은 현재 실행 중인 어셈블리의 디렉터리를 확인하여 부모 디렉터리가 포함되어 있는지 확인합니다. 부모 디렉터리가 있는 경우 런타임은 디렉터리에서 부모 문화권에 대한 유효한 위성 어셈블리를 검색합니다. 어셈블리를 찾으면 런타임은 어셈블리에서 요청된 리소스를 검색합니다. 리소스를 찾으면 해당 리소스를 사용합니다. 리소스를 찾지 못하면 검색을 계속합니다.

7. 다음 런타임은 Windows Installer를 쿼리하여 요청 시 부모 위성 어셈블리를 설치할지 여부를 확인합니다. 이 경우 설치를 처리하고, 어셈블리를 로드하고, 어셈블리 또는 요청된 리소스를 검색합니다. 어셈블리에서 리소스를 찾으면 해당 리소스를 사용합니다. 리소스를 찾지 못하면 검색을 계속합니다.

8. 런타임은 적절한 대체 리소스를 찾을 수 없음을 나타내기 위해 AppDomain.AssemblyResolve 이벤트를 발생합니다. 이벤트를 처리하도록 선택하는 경우 이벤트 처리기는 리소스가 조회에 사용되는 위성 어셈블리에 대한 참조를 반환할 수 있습니다. 그렇지 않으면 이벤트 처리기가 반환 `null` 되고 검색이 계속됩니다.

9. 다음 런타임은 이전 세 단계와 마찬가지로 여러 잠재적인 수준을 통해 부모 어셈블리를 검색합니다. 각 문화권에는 CultureInfo.Parent 속성에 의해 정의된 하나의 부모만 있습니다. 그러나 그 부모는 또 다른 부모를 가질 수도 있습니다. 문화권의 Parent 속성이 반환 CultureInfo.InvariantCulture되면 부모 문화권에 대한 검색이 중지됩니다. 리소스 대체의 경우 고정 문화권은 부모 문화권 또는 리소스를 가질 수 있는 문화권으로 간주되지 않습니다.

10. 원래 지정된 문화권과 모든 부모를 검색했으며 리소스를 아직 찾을 수 없는 경우 기본(대체) 문화권에 대한 리소스가 사용됩니다. 일반적으로 기본 문화권에 대한 리소스는 주 애플리케이션 어셈블리에 포함됩니다. 그러나 리소스의 최종 폴백 위치가 주 어셈블리가 아닌 위성 어셈블리임을 나타내기 위해 Satellite 특성의 Location 속성에 NeutralResourcesLanguageAttribute 값을 지정할 수 있습니다.

    > **참고**
    >
    > 기본 리소스는 주 어셈블리를 사용하여 컴파일할 수 있는 유일한 리소스입니다. 위성 어셈블리를 NeutralResourcesLanguageAttribute 특성을 사용하여 지정하지 않으면 궁극적인 대체(최종 부모)가 됩니다. 따라서 기본 어셈블리에 항상 기본 리소스 집합을 포함하는 것이 좋습니다. 이렇게 하면 예외가 발생하는 것을 방지할 수 있습니다. 기본 리소스를 포함하면 모든 리소스에 대한 대체를 제공하여 문화적으로 특정되지 않은 경우에도 항상 사용자에게 하나 이상의 리소스를 제공합니다.

11. 마지막으로 런타임에서 기본(대체) 문화권에 대한 리소스를 찾지 못하면 리소스를 찾을 수 없음을 나타내는 예외가 발생합니다 MissingManifestResourceExceptionMissingSatelliteAssemblyException.

예를 들어 애플리케이션이 스페인어(멕시코)(문화권)에 대해 지역화된 리소스를 `es-MX` 요청한다고 가정합니다. 런타임은 먼저 전역 어셈블리 캐시에서 일치하는 `es-MX`어셈블리를 검색하지만 찾지 못합니다. 그런 다음 런타임은 현재 실행 중인 어셈블리의 디렉터리에서 `es-MX` 디렉터리를 검색합니다. 실패하면 런타임은 전역 어셈블리 캐시에서 적절한 대체 문화권을 반영하는 부모 어셈블리(이 경우 `es` 스페인어)를 다시 검색합니다. 부모 어셈블리를 찾을 수 없는 경우, 런타임은 해당 리소스를 찾을 때까지 모든 잠재적 수준의 문화권별로 부모 어셈블리를 검색합니다 `es-MX`. 리소스를 찾을 수 없는 경우 런타임은 기본 문화권에 리소스를 사용합니다.

.NET Framework 리소스 대체 프로세스 최적화

다음 조건에서는 런타임이 위성 어셈블리에서 리소스를 검색하는 프로세스를 최적화할 수 있습니다.

- 위성 어셈블리는 코드 어셈블리와 동일한 위치에 배포됩니다. 코드 어셈블리가 전역 어셈블리 캐시에 설치되면 위성 어셈블리도 전역 어셈블리 캐시에 설치됩니다. 코드 어셈블리가 디렉터리에 설치된 경우 위성 어셈블리는 해당 디렉터리의 문화권별 폴더에 설치됩니다.
- 위성 어셈블리는 요청 시 설치되지 않습니다.
- 애플리케이션 코드는 이벤트를 처리 AppDomain.AssemblyResolve 하지 않습니다.

다음 예제와 같이 relativeBindForResources< 요소를 포함하고> 해당 `enabled` 특성을 `true` 애플리케이션 구성 파일에 설정하여 위성 어셈블리에 대한 프로브를 최적화합니다.

XML

```
<configuration>
   <runtime>
      <relativeBindForResources enabled="true" />
   </runtime>
</configuration>
```

위성 어셈블리에 최적화된 프로브는 옵트인 기능입니다. 즉, 런타임은 >리소스 대체 프로세스에 설명된 단계를 따릅니다.`enabled` `true` 이 경우 위성 어셈블리를 검색하는 프로세스는 다음과 같이 수정됩니다.

- 런타임은 부모 코드 어셈블리의 위치를 사용하여 위성 어셈블리를 검색합니다. 부모 어셈블리가 전역 어셈블리 캐시에 설치되어 있으면 런타임은 캐시에서 검색되지만 애플리케이션의 디렉터리에는 검색되지 않습니다. 부모 어셈블리가 애플리케이션 디렉터리에 설치된 경우 런타임은 전역 어셈블리 캐시가 아닌 애플리케이션 디렉터리에서 프로브합니다.
- 런타임은 Windows Installer에서 위성 어셈블리의 주문형 설치를 쿼리하지 않습니다.
- 특정 리소스 어셈블리에 대한 프로브가 실패하면 런타임에서 AppDomain.AssemblyResolve 이벤트를 발생시키지 않습니다.

.NET Core 리소스 대체 프로세스

.NET Core 리소스 대체 프로세스에는 다음 단계가 포함됩니다.

1. 런타임은 요청된 문화권에 대한 위성 어셈블리를 로드하려고 시도합니다.
    - 요청된 문화권과 일치하는 하위 디렉터리에 대해 현재 실행 중인 어셈블리의 디렉터리를 확인합니다. 하위 디렉터리를 찾으면 해당 하위 디렉터리에서 요청된 문화권에 대한 유효한 위성 어셈블리를 검색하고 로드합니다.

      > **참고**
      >
      > 대/소문자를 구분하는 파일 시스템(즉, Linux 및 macOS)이 있는 운영 체제에서 문화권 이름 하위 디렉터리 검색은 대/소문자를 구분합니다. 하위 디렉터리 이름은 CultureInfo.Name의 대소문자를 정확히 일치시켜야 합니다 (예: `es`, `es-MX`).

      > **참고**

프로그래머가 사용자 지정 어셈블리 로드 컨텍스트를 AssemblyLoadContext파생한 경우 상황은 복잡합니다. 실행 중인 어셈블리가 사용자 지정 컨텍스트에 로드된 경우 런타임은 위성 어셈블리를 사용자 지정 컨텍스트로 로드합니다. 세부 정보는 이 문서의 범위를 벗어났습니다. AssemblyLoadContext을(를) 참조하세요.

- 위성 어셈블리가 발견되지 않은 경우, AssemblyLoadContext가 위성 어셈블리를 찾을 수 없음을 나타내는 AssemblyLoadContext.Resolving 이벤트를 발생시킵니다. 이벤트를 처리하도록 선택하면 이벤트 처리기가 위성 어셈블리에 대한 참조를 로드하고 반환할 수 있습니다.

- 위성 어셈블리를 아직 찾을 수 없는 경우 AssemblyLoadContext는 AppDomain이 위성 어셈블리를 찾을 수 없음을 나타내는 이벤트를 트리거 AppDomain.AssemblyResolve 합니다. 이벤트를 처리하도록 선택하면 이벤트 처리기가 위성 어셈블리에 대한 참조를 로드하고 반환할 수 있습니다.

2. 위성 어셈블리가 발견되면 런타임은 요청된 리소스를 검색합니다. 어셈블리에서 리소스를 찾으면 해당 리소스를 사용합니다. 리소스를 찾지 못하면 검색을 계속합니다.

> **참고**
>
> 위성 어셈블리 내에서 리소스를 찾기 위해 런타임은 현재ResourceManager에 대해 요청된 CultureInfo.Name 리소스 파일을 검색합니다. 리소스 파일 내에서 요청된 리소스 이름을 검색합니다. 둘 중 하나를 찾을 수 없으면 리소스를 찾을 수 없는 것으로 처리됩니다.

3. 런타임은 다음으로 1단계와 2단계를 반복할 때마다 여러 잠재적인 수준을 통해 부모 문화권 어셈블리를 검색합니다.

   부모 문화권은 적절한 대체 문화권으로 정의됩니다. 리소스를 제공하는 것이 예외를 throw하는 것보다 낫기 때문에 부모를 백업 후보로 고려하세요. 이 프로세스를 통해 리소스를 다시 사용할 수도 있습니다. 자식 문화권이 요청된 리소스를 지역화할 필요가 없는 경우에만 부모 수준에서 특정 리소스를 포함해야 합니다. 예를 들어, `en` 위성 어셈블리(중립 영어), `en-GB` 위성 어셈블리(영국에서 사용되는 영어), `en-US` 위성 어셈블리(미국에서 사용되는 영어)를 제공하는 경우, `en` 위성에는 공통 용어가 포함되고, `en-GB` 및 `en-US` 위성은 차이가 있는 용어에 대해서만 재정의를 제공합니다.

   각 문화권에는 CultureInfo.Parent 속성에 의해 정의된 하나의 부모만 있습니다. 그러나 그 부모는 또 다른 부모를 가질 수도 있습니다. 문화의 Parent 속성이 CultureInfo.InvariantCulture를 반환할 때 부모 문화권의 검색이 중지됩니다. 자원 폴백의

경우 불변 문화권은 부모 문화권이나 리소스를 가질 수 있는 문화권으로 간주되지 않습니다.

4. 원래 지정된 문화권과 모든 부모를 검색했으며 리소스를 아직 찾을 수 없는 경우 기본(대체) 문화권에 대한 리소스가 사용됩니다. 일반적으로 기본 문화권에 대한 리소스는 주 애플리케이션 어셈블리에 포함됩니다. 그러나 속성 값을 지정하여 리소스의 SatelliteLocation 최종 대체 위치가 주 어셈블리가 아닌 위성 어셈블리임을 나타낼 수 있습니다.

   > **참고**
   >
   > 기본 리소스는 주 어셈블리를 사용하여 컴파일할 수 있는 유일한 리소스입니다. 위성 어셈블리를 NeutralResourcesLanguageAttribute 특성을 사용하여 지정하지 않으면 궁극적인 대체(최종 부모)가 됩니다. 따라서 기본 어셈블리에 항상 기본 리소스 집합을 포함하는 것이 좋습니다. 이렇게 하면 예외가 발생하는 것을 방지할 수 있습니다. 기본 리소스 파일을 포함하면 모든 리소스에 대한 대체를 제공하고 문화권에 특정하지 않더라도 항상 사용자에 대해 하나 이상의 리소스가 있는지 확인합니다.

5. 마지막으로 런타임이 기본(대체) 문화권에 대한 리소스 파일을 찾지 못하면 리소스를 찾을 수 없음을 알리는 MissingManifestResourceException 또는 MissingSatelliteAssemblyException 예외가 발생합니다. 리소스 파일을 찾았지만 요청된 리소스가 없는 경우 요청이 반환됩니다 `null`.

위성 어셈블리에 대한 최종 예비 옵션

필요에 따라 주 어셈블리에서 리소스를 제거하고 런타임이 특정 문화권에 해당하는 위성 어셈블리에서 최종 대체 리소스를 로드하도록 지정할 수 있습니다. 대체 프로세스를 제어하려면 생성자를 사용하고 NeutralResourcesLanguageAttribute(String, UltimateResourceFallbackLocation) Resource Manager가 주 어셈블리에서 또는 위성 어셈블리에서 대체 리소스를 추출해야 하는지 여부를 지정하는 매개 변수 값을 UltimateResourceFallbackLocation 제공합니다.

다음 .NET Framework 예제에서는 이 특성을 사용하여 NeutralResourcesLanguageAttribute 프랑스어(`fr`) 언어에 대한 위성 어셈블리에 애플리케이션의 대체 리소스를 저장합니다. 이 예제에는 이름이 지정된 `Greeting`단일 문자열 리소스를 정의하는 두 개의 텍스트 기반 리소스 파일이 있습니다. 첫 번째 resources.fr.txt프랑스어 리소스를 포함합니다.

```text
Greeting=Bon jour!
```

두 번째 리소스는ru.txt러시아어 리소스를 포함합니다.

text

```
Greeting=Добрый день
```

이 두 파일은 명령줄에서 리소스 파일 생성기(*resgen.exe*) 를 실행하여 .resources 파일로 컴파일 됩니다. 프랑스어 리소스의 경우 명령은 다음과 같습니다.

콘솔

```
resgen.exe resources.fr.txt
```

러시아어 리소스의 경우 명령은 다음과 같습니다.

콘솔

```
resgen.exe resources.ru.txt
```

.resources 파일은 다음과 같이 프랑스어 리소스에 대한 명령줄에서 어셈블리 링커(*al.exe*) 를 실행하여 동적 링크 라이브러리에 포함됩니다.

콘솔

```
al /t:lib /embed:resources.fr.resources /culture:fr /out:fr\Example1.resources.dll
```

러시아어 리소스는 다음과 같습니다.

콘솔

```
al /t:lib /embed:resources.ru.resources /culture:ru /out:ru\Example1.resources.dll
```

애플리케이션 소스 코드는 Example1.cs 또는 Example1.vb 파일에 상주합니다. 기본 애플리케이션 리소스가 fr 하위 디렉터리에 있음을 나타내는 특성이 포함 NeutralResourcesLanguageAttribute 됩니다. Resource Manager를 인스턴스화하고, 리소스 값을 `Greeting` 검색하고, 콘솔에 표시합니다.

C#

```
using System;
using System.Reflection;
using System.Resources;

[assembly:NeutralResourcesLanguage("fr",
UltimateResourceFallbackLocation.Satellite)]

public class Example
```

```
{
   public static void Main()
   {
      ResourceManager rm = new ResourceManager("resources",
                                               typeof(Example).Assembly);
      string greeting = rm.GetString("Greeting");
      Console.WriteLine(greeting);
   }
}
```

그런 다음 다음과 같이 명령줄에서 C# 소스 코드를 컴파일할 수 있습니다.

콘솔

```
csc Example1.cs
```

Visual Basic 컴파일러에 대한 명령은 매우 유사합니다.

콘솔

```
vbc Example1.vb
```

주 어셈블리에 포함된 리소스가 없으므로 스위치를 사용하여 `/resource` 컴파일할 필요가 없습니다.

언어가 러시아어 이외의 다른 시스템에서 예제를 실행하면 다음 출력이 표시됩니다.

출력

```
Bon jour!
```

제안된 패키징 대안

시간 또는 예산 제약 조건으로 인해 애플리케이션이 지원하는 모든 하위 문화권에 대한 리소스 집합을 만들지 못할 수 있습니다. 대신 모든 관련 하위 문화권에서 사용할 수 있는 부모 문화권에 대한 단일 위성 어셈블리를 만들 수 있습니다. 예를 들어 지역별 영어 리소스를 요청하는 사용자가 검색하는 단일 영어 위성 어셈블리(en)와 지역별 독일어 리소스를 요청하는 사용자를 위한 단일 독일어 위성 어셈블리(de)를 제공할 수 있습니다. 예를 들어 독일(de-DE), 오스트리아(de-AT), 스위스(de-CH)에서 사용되는 독일어에 대한 요청은 독일 위성 어셈블리(de)로 대체됩니다. 기본 리소스는 최종 대체이므로 대부분의 애플리케이션 사용자가 요청할 리소스이므로 이러한 리소스를 신중하게 선택합니다. 이 방법은 덜 문화적으로 구체적이지만 애플리케이션의 지역화 비용을 크게 줄일 수 있는 리소스를 배포합니다.

참고하십시오

- .NET 앱의 리소스
- 전역 어셈블리 캐시
- 리소스 파일 만들기
- 위성 어셈블리 만들기

.NET 앱에서 리소스 검색

2025. 06. 17.

.NET 앱에서 지역화된 리소스를 사용하는 경우 기본 또는 중립 문화권에 대한 리소스를 주 어셈블리로 패키지하고 앱이 지원하는 각 언어 또는 문화권에 대해 별도의 위성 어셈블리를 만드는 것이 이상적입니다. 그런 다음 다음 섹션에서 설명한 ResourceManager 대로 클래스를 사용하여 명명된 리소스에 액세스할 수 있습니다. 주 어셈블리 및 위성 어셈블리에 리소스를 포함하지 않도록 선택하는 경우 이 문서의 뒷부분에 있는 *.resources* 파일에서 리소스 검색 섹션에 설명된 대로 이진 .resources 파일에 직접 액세스할 수도 있습니다.

어셈블리에서 리소스 검색

클래스는 ResourceManager 런타임에 리소스에 대한 액세스를 제공합니다. 메서드를 ResourceManager.GetString 사용하여 문자열 리소스를 검색하고 또는 ResourceManager.GetObject 메서드를 ResourceManager.GetStream 사용하여 문자열이 아닌 리소스를 검색합니다. 각 메서드에는 두 개의 오버로드가 있습니다.

- 단일 매개 변수가 리소스의 이름을 포함하는 문자열인 오버로드입니다. 메서드는 현재 문화권에 대한 해당 리소스를 검색하려고 시도합니다. 자세한 내용은 GetString(String), GetObject(String), 및 GetStream(String) 메서드를 참조하세요.
- 두 매개 변수가 있는 오버로드: 리소스의 이름을 포함하는 문자열 및 CultureInfo 리소스를 검색할 문화권을 나타내는 개체입니다. 해당 문화권에 대한 리소스 집합을 찾을 수 없는 경우 리소스 관리자는 대체 규칙을 사용하여 적절한 리소스를 검색합니다. 자세한 내용은 GetString(String, CultureInfo), GetObject(String, CultureInfo), 및 GetStream(String, CultureInfo) 메서드를 참조하세요.

리소스 관리자는 리소스 대체 프로세스를 사용하여 앱이 문화권별 리소스를 검색하는 방법을 제어합니다. 자세한 내용은 패키지 및 리소스 배포의 "리소스 대체 프로세스" 섹션을 참조하세요. 개체를 인스턴스화하는 ResourceManager 방법에 대한 자세한 내용은 클래스 항목의 "ResourceManager 개체 인스턴스화" 섹션을 ResourceManager 참조하세요.

문자열 데이터 검색 예제

다음 예제에서는 메서드를 GetString(String) 호출하여 현재 UI 문화권의 문자열 리소스를 검색합니다. 여기에는 영어(미국) 문화권에 대한 중립 문자열 리소스와 프랑스(프랑스) 및 러시아어(러시아) 문화권에 대한 지역화된 리소스가 포함됩니다. 다음 영어(미국) 리소스는 Strings.txt파일에 있습니다.

```
text
```

```
TimeHeader=The current time is
```

프랑스어(프랑스) 리소스는 Strings.fr-FR.txt파일에 있습니다.

text

```
TimeHeader=L'heure actuelle est
```

러시아(러시아) 리소스는 Strings.ru-RU.txt파일에 있습니다.

text

```
TimeHeader=Текущее время —
```

이 예제의 소스 코드는 C# 버전의 코드에 대한 GetString.cs 파일에 있고 Visual Basic 버전의 GetString.vb 리소스를 사용할 수 있는 세 문화권과 스페인어(스페인) 문화권이라는 네 가지 문화권의 이름을 포함하는 문자열 배열을 정의합니다. 5번 반복 실행되는 루프는 이러한 문화권 중 하나를 임의로 선택하고, 그 결과를 Thread.CurrentCulture 및 CultureInfo.CurrentUICulture 속성에 할당합니다. 그런 다음 메서드를 GetString(String) 호출하여 현지화된 문자열을 검색합니다. 이 문자열은 하루 중 시간과 함께 표시됩니다.

C#

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguageAttribute("en-US")]

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "ru-RU", "es-ES" };
      Random rnd = new Random();
      ResourceManager rm = new ResourceManager("Strings",
                               typeof(Example).Assembly);

      for (int ctr = 0; ctr <= cultureNames.Length; ctr++) {
         string cultureName = cultureNames[rnd.Next(0, cultureNames.Length)];
         CultureInfo culture = CultureInfo.CreateSpecificCulture(cultureName);
         Thread.CurrentThread.CurrentCulture = culture;
         Thread.CurrentThread.CurrentUICulture = culture;

         Console.WriteLine($"Current culture: {culture.NativeName}");
         string timeString = rm.GetString("TimeHeader");
```

```
            Console.WriteLine($"{timeString} {DateTime.Now:T}\n");
        }
    }
}
// The example displays output like the following:
//    Current culture: English (United States)
//    The current time is 9:34:18 AM
//
//    Current culture: Español (España, alfabetización internacional)
//    The current time is 9:34:18
//
//    Current culture: русский (Россия)
//    Текущее время — 9:34:18
//
//    Current culture: français (France)
//    L'heure actuelle est 09:34:18
//
//    Current culture: русский (Россия)
//    Текущее время — 9:34:18
```

다음 일괄 처리(*.bat*) 파일은 예제를 컴파일하고 적절한 디렉터리에 위성 어셈블리를 생성합니다. C# 언어 및 컴파일러에 대한 명령이 제공됩니다. Visual Basic의 경우 `csc`를 `vbc`으로 변경하고, `GetString.cs`를 `GetString.vb`으로 변경합니다.

콘솔

```
resgen strings.txt
csc GetString.cs -resource:strings.resources

resgen strings.fr-FR.txt
md fr-FR
al -embed:strings.fr-FR.resources -culture:fr-FR -out:fr-
FR\GetString.resources.dll

resgen strings.ru-RU.txt
md ru-RU
al -embed:strings.ru-RU.resources -culture:ru-RU -out:ru-
RU\GetString.resources.dll
```

현재 UI 문화권이 스페인어(스페인)인 경우 스페인어 리소스를 사용할 수 없으며 영어가 예제의 기본 문화권이므로 예제에 영어 리소스가 표시됩니다.

개체 데이터 검색 예제

GetObject 및 GetStream 메서드를 사용하여 개체 데이터를 검색할 수 있습니다. 여기에는 기본 데이터 형식, 직렬화 가능한 개체 및 이진 형식(예: 이미지)으로 저장된 개체가 포함됩니다.

다음 예제에서는 이 메서드를 사용하여 GetStream(String) 앱의 시작 창을 여는 데 사용되는 비트맵을 검색합니다. CreateResources.cs(C#) 또는 CreateResources.vb(Visual Basic의 경우)라는 파일의 다음 소스 코드는 직렬화된 이미지를 포함하는 .resx 파일을 생성합니다. 이 경우 이미지가 SplashScreen.jpg; 파일에서 로드됩니다. 파일 이름을 수정하여 자신의 이미지를 대체할 수 있습니다.

C#

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Resources;

public class Example
{
   public static void Main()
   {
      Bitmap bmp = new Bitmap(@".\SplashScreen.jpg");
      MemoryStream imageStream = new MemoryStream();
      bmp.Save(imageStream, ImageFormat.Jpeg);

      ResXResourceWriter writer = new ResXResourceWriter("AppResources.resx");
      writer.AddResource("SplashScreen", imageStream);
      writer.Generate();
      writer.Close();
   }
}
```

다음 코드는 리소스를 검색하고 컨트롤에 PictureBox 이미지를 표시합니다.

C#

```
using System;
using System.Drawing;
using System.IO;
using System.Resources;
using System.Windows.Forms;

public class Example
{
   public static void Main()
   {
      ResourceManager rm = new ResourceManager("AppResources",
typeof(Example).Assembly);
      Bitmap screen = (Bitmap) Image.FromStream(rm.GetStream("SplashScreen"));

      Form frm = new Form();
      frm.Size = new Size(300, 300);

      PictureBox pic = new PictureBox();
```

```
      pic.Bounds = frm.RestoreBounds;
      pic.BorderStyle = BorderStyle.Fixed3D;
      pic.Image = screen;
      pic.SizeMode = PictureBoxSizeMode.StretchImage;

      frm.Controls.Add(pic);
      pic.Anchor = AnchorStyles.Top | AnchorStyles.Bottom |
                   AnchorStyles.Left | AnchorStyles.Right;

      frm.ShowDialog();
   }
}
```

다음 일괄 처리 파일을 사용하여 C# 예제를 빌드할 수 있습니다. Visual Basic의 경우 `csc` `vbc`변경하고 소스 코드 파일의 확장명은 `.cs` `.vb`변경합니다.

콘솔

```
csc CreateResources.cs
CreateResources

resgen AppResources.resx

csc GetStream.cs -resource:AppResources.resources
```

다음 예제에서는 메서드를 ResourceManager.GetObject(String) 사용하여 사용자 지정 개체를 역직렬화합니다. 이 예제에는 다음 구조체를 정의하는 UIElements.cs(Visual Basic용 UIElements.vb)라는 소스 코드 파일이 포함되어 `PersonTable`있습니다. 이 구조체는 테이블 열의 지역화된 이름을 표시하는 일반 테이블 표시 루틴에서 사용됩니다. 구조체는 `PersonTable` 특성에 의해 SerializableAttribute로 표시됩니다.

C#

```
using System;

[Serializable] public struct PersonTable
{
   public readonly int nColumns;
   public readonly string column1;
   public readonly string column2;
   public readonly string column3;
   public readonly int width1;
   public readonly int width2;
   public readonly int width3;

   public PersonTable(string column1, string column2, string column3,
                      int width1, int width2, int width3)
   {
      this.column1 = column1;
      this.column2 = column2;
```

```
      this.column3 = column3;
      this.width1 = width1;
      this.width2 = width2;
      this.width3 = width3;
      this.nColumns = typeof(PersonTable).GetFields().Length / 2;
   }
}
```

CreateResources.cs(Visual Basic용 CreateResources.vb)라는 파일의 다음 코드는 테이블 제목과 `PersonTable` 영어로 지역화된 앱에 대한 정보를 포함하는 개체를 저장하는 UIResources.resx라는 XML 리소스 파일을 만듭니다.

C#

```
using System;
using System.Resources;

public class CreateResource
{
   public static void Main()
   {
      PersonTable table = new PersonTable("Name", "Employee Number",
                                          "Age", 30, 18, 5);
      ResXResourceWriter rr = new ResXResourceWriter(@".\UIResources.resx");
      rr.AddResource("TableName", "Employees of Acme Corporation");
      rr.AddResource("Employees", table);
      rr.Generate();
      rr.Close();
   }
}
```

GetObject.vb(GetObject.cs)라는 소스 코드 파일의 다음 코드는 리소스를 검색하여 콘솔에 표시합니다.

C#

```
using System;
using System.Resources;

[assembly: NeutralResourcesLanguageAttribute("en")]

public class Example
{
   public static void Main()
   {
      string fmtString = String.Empty;
      ResourceManager rm = new ResourceManager("UIResources",
typeof(Example).Assembly);
      string title = rm.GetString("TableName");
      PersonTable tableInfo = (PersonTable) rm.GetObject("Employees");
```

```
      if (! String.IsNullOrEmpty(title)) {
         fmtString = "{0," + ((Console.WindowWidth + title.Length) / 2).ToString()
+ "}";
         Console.WriteLine(fmtString, title);
         Console.WriteLine();
      }

      for (int ctr = 1; ctr <= tableInfo.nColumns; ctr++) {
         string columnName = "column"  + ctr.ToString();
         string widthName = "width" + ctr.ToString();
         string value =
tableInfo.GetType().GetField(columnName).GetValue(tableInfo).ToString();
         int width = (int)
tableInfo.GetType().GetField(widthName).GetValue(tableInfo);
         fmtString = "{0,-" + width.ToString() + "}";
         Console.Write(fmtString, value);
      }
      Console.WriteLine();
   }
}
```

필요한 리소스 파일 및 어셈블리를 빌드하고 다음 일괄 처리 파일을 실행하여 앱을 실행할 수 있습니다. 구조 `/r`에 관한 정보를 참조할 수 있도록, UIElements.dll에 대한 참조를 Resgen.exe에 제공하는 옵션 `PersonTable`을 사용해야 합니다. C#을 사용하는 경우, 컴파일러 이름을 `vbc`에서 `csc`으로 바꾸고, 확장자를 `.vb`에서 `.cs`으로 바꾸십시오.

콘솔

```
vbc -t:library UIElements.vb
vbc CreateResources.vb -r:UIElements.dll
CreateResources

resgen UIResources.resx  -r:UIElements.dll
vbc GetObject.vb -r:UIElements.dll -resource:UIResources.resources

GetObject.exe
```

위성 어셈블리에 대한 버전 지원

기본적으로 개체는 ResourceManager 요청된 리소스를 검색할 때 주 어셈블리의 버전 번호와 일치하는 버전 번호가 있는 위성 어셈블리를 찾습니다. 앱을 배포한 후 주 어셈블리 또는 특정 리소스 위성 어셈블리를 업데이트할 수 있습니다. .NET Framework는 주 어셈블리 및 위성 어셈블리의 버전 관리 지원을 제공합니다.

이 특성은 SatelliteContractVersionAttribute 주 어셈블리에 대한 버전 관리 지원을 제공합니다. 앱의 주 어셈블리에서 이 특성을 지정하면 위성 어셈블리를 업데이트하지 않고 주 어셈블리를 업데이트하고 다시 배포할 수 있습니다. 주 어셈블리를 업데이트한 후 주 어셈블리의 버전 번호

를 증가하지만 위성 계약 버전 번호는 변경되지 않은 상태로 둡니다. 리소스 관리자는 요청된 리소스를 검색할 때 이 특성에 지정된 위성 어셈블리 버전을 로드합니다.

게시자 정책 어셈블리는 위성 어셈블리의 버전 관리 지원을 제공합니다. 주 어셈블리를 업데이트하지 않고 위성 어셈블리를 업데이트하고 다시 배포할 수 있습니다. 위성 어셈블리를 업데이트한 후 해당 버전 번호를 증가시키고 게시자 정책 어셈블리와 함께 제공합니다. 게시자 정책 어셈블리에서 새 위성 어셈블리가 이전 버전과 호환되도록 지정합니다. 리소스 관리자는 특성을 사용하여 SatelliteContractVersionAttribute 위성 어셈블리의 버전을 결정하지만 어셈블리 로더는 게시자 정책에 지정된 위성 어셈블리 버전에 바인딩됩니다. 게시자 정책 어셈블리에 대한 자세한 내용은 게시자 정책 파일 만들기를 참조하세요.

전체 어셈블리 버전 관리 지원을 사용하도록 설정하려면 전역 어셈블리 캐시 에 강력한 이름의 어셈블리를 배포하고 애플리케이션 디렉터리에 강력한 이름이 없는 어셈블리를 배포하는 것이 좋습니다. 애플리케이션 디렉터리에 강력한 이름의 어셈블리를 배포하려는 경우 어셈블리를 업데이트할 때 위성 어셈블리의 버전 번호를 증가시킬 수 없습니다. 대신 기존 코드를 업데이트된 코드로 바꾸고 동일한 버전 번호를 유지 관리하는 현재 위치 업데이트를 수행해야 합니다. 예를 들어 위성 어셈블리의 버전 1.0.0.0을 완전히 지정된 어셈블리 이름으로 업데이트하려면 "myApp.resources, Version=1.0.0.0, Culture=de, PublicKeyToken=b03f5f11d50a3a", 완전히 지정된 동일한 어셈블리 이름 "myApp.resources, Version=1.0.0.0, Culture=de, PublicKeyToken=b03f5f11d50a3a"로 컴파일된 업데이트된 myApp.resources.dll 덮어씁니다. 위성 어셈블리 파일에서 현재 위치 업데이트를 사용하면 앱에서 위성 어셈블리의 버전을 정확하게 확인하기가 어렵습니다.

어셈블리 버전 관리에 대한 자세한 내용은 어셈블리 버전 관리 및 런타임에서 어셈블리를 찾는 방법을 참조하세요.

.resources 파일에서 리소스 검색

위성 어셈블리에 리소스를 배포하지 않도록 선택한 경우에도 개체를 ResourceManager 사용하여 .resources 파일의 리소스에 직접 액세스할 수 있습니다. 이렇게 하려면 .resources 파일을 올바르게 배포해야 합니다. 그런 다음 이 메서드를 ResourceManager.CreateFileBasedResourceManager 사용하여 개체를 ResourceManager 인스턴스화하고 독립 실행형 .resources 파일이 포함된 디렉터리를 지정합니다.

.resources 파일 배포

애플리케이션 어셈블리 및 위성 어셈블리에 .resources 파일을 포함하는 경우 각 위성 어셈블리의 파일 이름은 같지만 위성 어셈블리의 문화권을 반영하는 하위 디렉터리에 배치됩니다. 반면. .resources 파일에서 리소스에 직접 액세스하는 경우 모든 .resources 파일을 단일 디렉터리(일반적으로 애플리케이션 디렉터리의 하위 디렉터리)에 배치할 수 있습니다. 앱의 기본 .resources 파일 이름은 문화권(예: strings.resources)을 표시하지 않고 루트 이름으로만 구성됩니다. 지역화된

각 문화권의 리소스는 파일에 저장되며, 이 파일의 이름은 루트 이름에 문화권을 덧붙여 구성됩니다(예: strings.ja.resources 또는 strings.de-DE.resources).

다음 그림에서는 디렉터리 구조에서 리소스 파일을 배치해야 하는 위치를 보여 줍니다. 또한 .resource 파일에 대한 명명 규칙도 제공합니다.

리소스 관리자 사용

리소스를 만들고 적절한 디렉터리에 배치한 후에는 메서드를 ResourceManager 호출하여 리소스를 사용할 개체를 CreateFileBasedResourceManager(String, String, Type) 만듭니다. 첫 번째 매개 변수는 앱의 기본 .resources 파일의 루트 이름을 지정합니다(이전 섹션의 예제에 대한 "문자열"). 두 번째 매개 변수는 리소스의 위치(이전 예제의 "리소스")를 지정합니다. 세 번째 매개 변수는 사용할 구현을 ResourceSet 지정합니다. 세 번째 매개 변수인 `null`경우 기본 런타임 ResourceSet 이 사용됩니다.

> **참고**
>
> 독립 실행형 .resources 파일을 사용하여 ASP.NET 앱을 배포하지 마세요. 이로 인해 잠금 문제가 발생하고 XCOPY 배포가 중단될 수 있습니다. 위성 어셈블리에 ASP.NET 리소스를 배포하는 것이 좋습니다. 자세한 내용은 **ASP.NET 웹 페이지 리소스 개요를 참조하세요**.

ResourceManager 개체를 인스턴스화한 후, 앞에서 설명한 대로 리소스를 검색하기 위해 GetString, GetObject, 및 GetStream 메서드를 사용합니다. 그러나 .resources 파일에서 직접 리소스를 검색하는 것은 어셈블리에서 포함된 리소스를 검색하는 것과 다릅니다. .resources 파일에서 리소스를 검색할 때, GetString(String), GetObject(String), GetStream(String) 메서드는 현재 문화권과 관계없이 항상 기본 문화권의 리소스를 검색합니다. 앱의 현재 문화권 또는 특정 문화권의 리소스를 검색하려면 , GetString(String, CultureInfo)또는 GetObject(String, CultureInfo) 메서드를 호출GetStream(String, CultureInfo)하고 리소스를 검색할 문화권을 지정해야 합니다. 현재 문화권의 리소스를 검색하려면 속성 값을 CultureInfo.CurrentCulture 인수로 `culture` 지정합니다. 리소스 관리자가 리소스 `culture`를 검색할 수 없는 경우 표준 리소스 대체 규칙을 사용하여 적절한 리소스를 검색합니다.

예제

다음 예제에서는 리소스 관리자가 .resources 파일에서 직접 리소스를 검색하는 방법을 보여 줍니다. 이 예제는 영어(미국), 프랑스어(프랑스) 및 러시아어(러시아) 문화권에 대한 세 개의 텍스트 기반 리소스 파일로 구성됩니다. 영어(미국)는 예제의 기본 문화권입니다. 해당 리소스는 *Strings.txt*다음 파일에 저장됩니다.

text

```
Greeting=Hello
Prompt=What is your name?
```

프랑스어(프랑스) 문화권에 대한 리소스는 Strings.fr-FR.txt이름이 지정된 다음 파일에 저장됩니다.

text

```
Greeting=Bon jour
Prompt=Comment vous appelez-vous?
```

러시아어(러시아) 문화권에 대한 리소스는 Strings.ru-RU.txt이름이 지정된 다음 파일에 저장됩니다.

text

```
Greeting=Здравствуйте
Prompt=Как вас зовут?
```

다음은 예제의 소스 코드입니다. 이 예제에서는 영어(미국), 영어(캐나다), 프랑스어(프랑스) 및 러시아어(러시아) 문화권에 대한 개체를 인스턴스화 CultureInfo 하고 각각 현재 문화권을 만듭니다. 그런 다음 ResourceManager.GetString(String, CultureInfo) 메서드는 CultureInfo.CurrentCulture 속성 값을 `culture` 인수로 제공하여 적절한 문화권별 리소스를 검색합니다.

C#

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguage("en-US")]

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "en-CA", "ru-RU", "fr-FR" };
```

```
      ResourceManager rm =
ResourceManager.CreateFileBasedResourceManager("Strings", "Resources", null);

      foreach (var cultureName in cultureNames) {
         Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(cultureName);
         string greeting = rm.GetString("Greeting", CultureInfo.CurrentCulture);
         Console.WriteLine($"\n{greeting}!");
         Console.Write(rm.GetString("Prompt", CultureInfo.CurrentCulture));
         string name = Console.ReadLine();
         if (! String.IsNullOrEmpty(name))
            Console.WriteLine("{0}, {1}!", greeting, name);
      }
      Console.WriteLine();
   }
}
// The example displays output like the following:
//       Hello!
//       What is your name? Dakota
//       Hello, Dakota!
//
//       Hello!
//       What is your name? Koani
//       Hello, Koani!
//
//       Здравствуйте!
//       Как вас зовут?Samuel
//       Здравствуйте, Samuel!
//
//       Bon jour!
//       Comment vous appelez-vous?Yiska
//       Bon jour, Yiska!
```

다음 일괄 처리 파일을 실행하여 예제의 C# 버전을 컴파일할 수 있습니다. Visual Basic을 사용하는 경우 `csc`을 `vbc`로 바꾸고, `.cs` 확장을 `.vb`로 바꾸세요.

콘솔

```
md Resources
resgen Strings.txt Resources\Strings.resources
resgen Strings.fr-FR.txt Resources\Strings.fr-FR.resources
resgen Strings.ru-RU.txt Resources\Strings.ru-RU.resources

csc Example.cs
```

참고하십시오

- ResourceManager
- .NET 앱의 리소스

- 리소스 패키징 및 배포
- 런타임에서 어셈블리를 찾는 방법

System.Resources.MissingManifestResourceException 클래스

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

.NET과 UWP 앱에서는 각각 다른 이유로 MissingManifestResourceException 예외가 발생합니다.

.NET 앱

.NET 앱에서는 특정 어셈블리에서 중립 문화권에 대한 리소스 집합을 로드할 수 없기 때문에 리소스 검색 시도가 실패할 때 MissingManifestResourceException이 throw됩니다. 특정 리소스를 검색하려고 할 때 예외가 throw되지만 리소스를 찾지 못한 것이 아니라 리소스 집합을 로드하지 못하여 발생합니다.

> **참고**
>
> 자세한 내용은 ResourceManager 클래스 항목의 "MissingManifestResourceException 예외 처리" 섹션을 참조하세요.

예외의 주요 원인은 다음과 같습니다.

- 리소스 집합은 정규화된 이름으로 식별되지 않습니다. 예를 들어 ResourceManager.ResourceManager(String, Assembly) 메서드 호출의 `baseName` 매개 변수가 네임스페이스가 없는 리소스 집합의 루트 이름을 지정하지만 리소스 집합이 어셈블리에 저장될 때 네임스페이스가 할당되는 경우 ResourceManager.GetString 메서드를 호출하면 이 예외가 throw됩니다.

  실행 파일에 기본 문화권의 리소스가 포함된 .resources 파일을 포함했으며, 앱이 MissingManifestResourceException오류를 발생시키는 경우, IL 디스어셈블러(Ildasm.exe) 같은 리플렉션 도구를 사용하여 리소스의 전체 이름을 확인할 수 있습니다. ILDasm에서 실행 파일의 **MANIFEST** 레이블을 두 번 클릭하여 **MANIFEST** 창을 엽니다. 리소스는 `.mresource` 항목으로 표시되며 외부 어셈블리 참조 및 사용자 지정 어셈블리 수준 특성 후에 나열됩니다. 명령줄 매개 변수로 이름이 전달되는 어셈블리에 포함된 리소스의 정규화된 이름을 나열하는 다음 간단한 유틸리티를 컴파일할 수도 있습니다.

  ```
  C#
  ```

```
using System;
using System.IO;
using System.Reflection;

public class Example0
{
   public static void Main()
   {
      if (Environment.GetCommandLineArgs().Length == 1) {
         Console.WriteLine("No filename.");
         return;
      }

      string filename = Environment.GetCommandLineArgs()[1].Trim();
      // Check whether the file exists.
      if (! File.Exists(filename)) {
         Console.WriteLine($"{filename} does not exist.");
         return;
      }

      // Try to load the assembly.
      Assembly assem = Assembly.LoadFrom(filename);
      Console.WriteLine($"File: {filename}");

      // Enumerate the resource files.
      string[] resNames = assem.GetManifestResourceNames();
      if (resNames.Length == 0)
         Console.WriteLine("   No resources found.");

      foreach (var resName in resNames)
         Console.WriteLine($"   Resource:
{resName.Replace(".resources", "")}");

      Console.WriteLine();
   }
}
```

- 리소스 집합은 해당 네임스페이스 및 루트 파일 이름만 사용하는 것이 아니라 해당 리소스 파일 이름(선택적 네임스페이스와 함께)과 해당 파일 확장명을 사용하여 식별합니다. 예를 들어, 중립 문화권의 리소스 집합이 `GlobalResources`으로 명명되고 ResourceManager.ResourceManager(String, Assembly) 생성자의 `baseName` 매개변수에 `GlobalResources`대신 `GlobalResources.resources` 값을 제공하는 경우, 이 예외가 발생합니다.

- 메서드 호출에서 식별된 문화권별 리소스 집합을 찾을 수 없으며 대체 리소스 집합을 로드할 수 없습니다. 예를 들어 영어(미국) 및 러시아(러시아어) 문화권에 대한 위성 어셈블리를 만들지만 중립 문화권에 대한 리소스 집합을 제공하지 못하는 경우 앱의 현재 문화권이 영어(영국)인 경우 이 예외가 throw됩니다.

MissingManifestResourceException는 값이 0x80131532인 HRESULT `COR_E_MISSINGMANIFESTRESOURCE`을 사용합니다.

MissingManifestResourceException 참조 같음을 지원하는 기본 Equals 구현을 사용합니다.

MissingManifestResourceException인스턴스의 초기 속성 값 목록은 MissingManifestResourceException 생성자를 참조하세요.

> **참고**
>
> 위성 어셈블리를 사용할 수 없는 경우 앱이 실패하지 않도록 주 어셈블리에 중립 리소스 집합을 포함하는 것이 좋습니다.

UWP(유니버설 Windows 플랫폼) 앱

UWP 앱은 중립 문화권을 포함한 여러 문화권에 대한 리소스를 단일 패키지 리소스 인덱스(.pri) 파일에 배포합니다. 따라서 UWP 앱에서 기본 문화권에 대한 리소스를 찾을 수 없는 경우, MissingManifestResourceException 오류가 다음 조건 중 하나에서 발생합니다.

- 앱에 .pri 파일이 없거나 .pri 파일을 열 수 없습니다.
- 앱의 .pri 파일에는 지정된 루트 이름에 대한 리소스 집합이 포함되어 있지 않습니다.

System.Resources.NeutralResourcesLanguageAttribute 클래스

아티클 • 2025. 03. 29.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

데스크톱 앱에서 NeutralResourcesLanguageAttribute 특성은 리소스 관리자에게 앱의 기본 문화권과 해당 리소스의 위치를 알려줍니다. 기본적으로 리소스는 주 앱 어셈블리에 포함되며 다음과 같이 특성을 사용할 수 있습니다. 이 문은 영어(미국)가 앱의 기본 문화권임을 지정합니다.

C#

```
[assembly: NeutralResourcesLanguage("en-US")]
```

NeutralResourcesLanguageAttribute 특성을 사용하여 ResourceManager 특성 문에 UltimateResourceFallbackLocation 열거형 값을 제공하여 기본 문화권의 리소스를 찾을 수 있는 위치를 나타낼 수도 있습니다. 이는 리소스가 위성 어셈블리에 상주함을 나타내기 위해 가장 일반적으로 수행됩니다. 예를 들어 다음 문은 영어(미국)가 앱의 기본 또는 중립 문화권이며 해당 리소스가 위성 어셈블리에 상주한다고 지정합니다. ResourceManager 개체는 en-US하위 디렉터리에서 찾습니다.

C#

```
[assembly: NeutralResourcesLanguage("en-US",
UltimateResourceFallbackLocation.Satellite)]
```

> **팁**
>
> 항상 NeutralResourcesLanguageAttribute 특성을 사용하여 앱의 기본 문화권을 정의하는 것이 좋습니다.

이 특성은 다음 두 가지 역할을 수행합니다.

- 기본 문화권의 리소스가 앱의 주 어셈블리에 포함되고 ResourceManager 기본 문화권과 동일한 문화권에 속하는 리소스를 검색해야 하는 경우 ResourceManager 위성 어셈블리를 검색하는 대신 주 어셈블리에 있는 리소스를 자동으로 사용합니다. 따라서 일반적인 어셈블리 검색이 무시되고 로드하는 첫 번째 리소스에 대한 조회 성능이 향상되며 작업 집합을 줄일 수 있습니다. ResourceManager 리소스 파일을 검색하는 데 사용하는 프로세스는 리소스 패키징 및 배포를 참조하세요.

- 기본 문화권의 리소스가 주 앱 어셈블리가 아닌 위성 어셈블리에 있는 경우 NeutralResourcesLanguageAttribute 특성은 런타임에서 리소스를 로드할 수 있는 문화권 및 디렉터리를 지정합니다.

Windows 8.x 스토어 앱

ResourceManager 클래스를 사용하여 리소스를 로드하고 검색하는 Windows 8.x 스토어 앱에서 NeutralResourcesLanguageAttribute 특성은 프로브 실패 시 리소스가 사용되는 중립 문화권을 정의합니다. 리소스의 위치를 지정하지 않습니다. 기본적으로 ResourceManager 앱의 PRI(패키지 리소스 인덱스) 파일을 사용하여 기본 문화권의 리소스를 찾습니다. NeutralResourcesLanguageAttribute 특성에 의해 정의된 중립 문화권은 이 효과를 시뮬레이션하기 위해 UI 언어 목록의 끝에 추가됩니다.

Windows 런타임Windows.ApplicationModel.Resources.ResourceLoader 클래스 또는 Windows.ApplicationModel.Resources.Core 네임스페이스의 형식을 사용하여 리소스를 로드하고 검색하면 NeutralResourcesLanguageAttribute 특성이 무시됩니다.

예시

다음 예제에서는 간단한 "Hello World" 앱을 사용하여 NeutralResourcesLanguageAttribute 특성을 사용하여 기본 문화권 또는 대체 문화권을 정의하는 방법을 보여 줍니다. 영어(en), 영어(미국) (en-US) 및 프랑스어(프랑스)(fr-FR) 문화권에 대한 별도의 리소스 파일을 생성해야 합니다. 다음은 영어 문화권의 ExampleResources.txt 텍스트 파일의 내용을 보여 줍니다.

```
# Resources for the default (en) culture.
Greeting=Hello
```

앱에서 리소스 파일을 사용하려면 다음과 같이 리소스 파일 생성기(Resgen.exe) 사용하여 파일을 텍스트(.txt) 형식에서 이진(.resources) 형식으로 변환해야 합니다.

```
resgen ExampleResources.txt
```

앱이 컴파일되면 이진 리소스 파일이 주 앱 어셈블리에 포함됩니다.

다음은 영어(미국) 문화권에 대한 리소스를 제공하는 ExampleResources.en-US.txt 텍스트 파일의 내용을 보여 줍니다.

```
# Resources for the en-US culture.
Greeting=Hi
```

텍스트 파일은 다음과 같이 명령줄에서 리소스 파일 생성기(ResGen.exe) 사용하여 이진 리소스 파일로 변환할 수 있습니다.

```
resgen ExampleResources.en-US.txt ExampleResources.en-US.resources
```

그런 다음 Assembly Linker(Al.exe) 사용하여 이진 리소스 파일을 어셈블리로 컴파일하고 다음 명령을 실행하여 앱 디렉터리의 en-US 하위 디렉터리에 배치해야 합니다.

```
al /t:lib /embed:ExampleResources.en-US.resources /culture:en-US /out:en-
us\Example.resources.dll
```

다음은 프랑스어(프랑스) 문화권에 대한 리소스를 제공하는 ExampleResources.fr-FR.txt 텍스트 파일의 내용을 보여 줍니다.

```
# Resources for the fr-FR culture.
Greeting=Bonjour
```

텍스트 파일은 다음과 같이 명령줄에서 ResGen.exe 사용하여 이진 리소스 파일로 변환할 수 있습니다.

```
resgen ExampleResources.fr-FR.txt ExampleResources.fr-FR.resources
```

그런 다음, 어셈블리 링커를 사용하여 이진 리소스 파일을 어셈블리로 컴파일하고 다음 명령을 실행하여 앱 디렉터리의 fr-FR 하위 디렉터리에 배치해야 합니다.

```
al /t:lib /embed:ExampleResources.fr-FR.resources /culture:fr-FR /out:fr-
FR\Example.resources.dll
```

다음 예제에서는 현재 문화권을 설정하고, 사용자 이름을 묻는 메시지를 표시하고, 지역화된 문자열을 표시하는 실행 코드를 제공합니다.

C#

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguageAttribute("en")]
public class Example
{
    public static void Main()
    {
        // Select the current culture randomly to test resource fallback.
        string[] cultures = { "de-DE", "en-us", "fr-FR" };
        Random rnd = new Random();
        int index = rnd.Next(0, cultures.Length);
        Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture(cultures[index]);
        Console.WriteLine($"The current culture is
{CultureInfo.CurrentUICulture.Name}");

        // Retrieve the resource.
        ResourceManager rm = new ResourceManager("ExampleResources",
                                                 typeof(Example).Assembly);
        string greeting = rm.GetString("Greeting");

        Console.Write("Enter your name: ");
        string name = Console.ReadLine();
        Console.WriteLine($"{greeting} {name}!");
    }
}
```

Visual Basic에서 다음 명령을 사용하여 컴파일할 수 있습니다.

```
vbc Example.vb /resource:ExampleResources.resources
```

또는 C#에서 다음 명령을 사용합니다.

```
csc Example.cs /resource:ExampleResources.resources
```

System.Resources.ResourceManager 클래스

아티클 • 2025. 03. 23.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> **ⓘ 중요**
>
> 신뢰할 수 없는 데이터를 사용하여 이 클래스에서 메서드를 호출하는 것은 보안 위험입니다. 신뢰할 수 있는 데이터로만 이 클래스의 메서드를 호출합니다. 자세한 내용은 **모든 입력 유효성 검사** 참조하세요.

ResourceManager 클래스는 어셈블리에 포함된 이진 .resources 파일 또는 독립 실행형 .resources 파일에서 리소스를 검색합니다. 앱이 지역화되고 지역화된 리소스가 위성 어셈블리에 배포된 경우 문화권별 리소스를 조회하고, 지역화된 리소스가 없을 때 리소스 대체를 제공하고, 리소스 직렬화를 지원합니다.

데스크톱 앱

데스크톱 앱의 경우 ResourceManager 클래스는 이진 리소스(.resources) 파일에서 리소스를 검색합니다. 일반적으로 언어 컴파일러 또는 어셈블리 링커(AL.exe) 이러한 리소스 파일을 어셈블리에 포함합니다. ResourceManager 개체를 사용하여 CreateFileBasedResourceManager 메서드를 호출하여 어셈블리에 포함되지 않은 .resources 파일에서 직접 리소스를 검색할 수도 있습니다.

> **⊗ 주의**
>
> ASP.NET 앱에서 독립 실행형 .resources 파일을 사용하면 리소스가 ReleaseAllResources 메서드에서 명시적으로 해제될 때까지 잠겨 있으므로 XCOPY 배포가 중단됩니다. ASP.NET 앱을 사용하여 리소스를 배포하려면 .resources 파일을 위성 어셈블리로 컴파일해야 합니다.

리소스 기반 앱에서 하나의 .resources 파일에는 문화권별 리소스를 찾을 수 없는 경우 리소스가 사용되는 기본 문화권의 리소스가 포함됩니다. 예를 들어 앱의 기본 문화권이 영어(en)인 경우 영어(미국) (en-US) 또는 프랑스어(프랑스)(fr-FR)와 같은 특정 문화권에 대해 지역화된 리소스를 찾을 수 없을 때마다 영어 리소스가 사용됩니다. 일반적으로 기본 문화권의 리소스는 주 앱 어셈블리에 포함되고 다른 지역화된 문화권에 대한 리소스는 위성 어셈블리에 포함됩니다. 위성 어셈블리에는 리소스만 포함됩니다. 주 어셈블리와 루

트 파일 이름이 동일하고 확장명이 .resources.dll. 어셈블리가 전역 어셈블리 캐시에 등록되지 않은 앱의 경우 위성 어셈블리는 어셈블리의 문화권에 해당하는 이름의 앱 하위 디렉터리에 저장됩니다.

리소스 만들기

리소스 기반 앱을 개발할 때 리소스 정보를 텍스트 파일(.txt 또는 .restext 확장명이 있는 파일) 또는 XML 파일(.resx 확장명이 있는 파일)에 저장합니다. 그런 다음 리소스 파일 생성기(Resgen.exe) 사용하여 텍스트 또는 XML 파일을 컴파일하여 이진 .resources 파일을 만듭니다. 그런 다음 C# 및 Visual Basic 컴파일러에 대한 `/resources` 같은 컴파일러 옵션을 사용하여 결과 .resources 파일을 실행 파일 또는 라이브러리에 포함하거나 어셈블리 링커(Al.exe)사용하여 위성 어셈블리에 포함할 수 있습니다. Visual Studio 프로젝트에 .resx 파일을 포함하는 경우 Visual Studio는 빌드 프로세스의 일부로 기본 및 지역화된 리소스의 컴파일 및 포함을 자동으로 처리합니다.

이상적으로는 앱이 지원하는 모든 언어 또는 적어도 각 언어의 의미 있는 하위 집합에 대한 리소스를 만들어야 합니다. 이진 .resources 파일 이름은 naming convention *basename*.*cultureName*.resources 형식을 따릅니다. 여기서 *basename* 은(는) 원하는 세부 수준에 따라 앱의 이름 또는 클래스의 이름이 됩니다. CultureInfo.Name 속성은 *cultureName*을 확인하는 데 사용됩니다. 앱의 기본 문화권에 대한 리소스는 .resources로 이름을 지정해야 합니다.

예를 들어 어셈블리에 기본 이름 MyResources가 있는 리소스 파일에 여러 리소스가 있다고 가정합니다. 이러한 리소스 파일에는 MyResources.ja-JP.resources는 일본(일본어) 문화권에 대해, MyResources.de.resources는 독일 문화권에 대해, MyResources.zh-CHS.resources는 간체 중국어 문화권에 대해, MyResources.fr-BE.resources는 프랑스(벨기에) 문화권에 대해 있어야 합니다. 기본 리소스 파일의 이름은 MyResources.resources여야 합니다. 문화권별 리소스 파일은 일반적으로 각 문화권에 대한 위성 어셈블리에 패키지됩니다. 기본 리소스 파일은 앱의 주 어셈블리에 포함되어야 합니다.

어셈블리 링커 리소스를 프라이빗으로 표시할 수 있지만 다른 어셈블리에서 액세스할 수 있도록 항상 공용으로 표시해야 합니다. 위성 어셈블리에 코드가 없기 때문에 프라이빗으로 표시된 리소스는 어떤 메커니즘을 통해든 앱에서 사용할 수 없습니다.

리소스 만들기, 패키징 및 배포에 대한 자세한 내용은 리소스 파일 만들기, 위성 어셈블리 만들기리소스 패키징 및 배포문서를 참조하세요.

ResourceManager 개체 인스턴스화

클래스 생성자 오버로드 중 하나를 호출하여 포함된 .resources 파일에서 리소스를 검색하는 ResourceManager 개체를 인스턴스화합니다. 이렇게 하면 ResourceManager 개체

를 특정 .resources 파일 및 위성 어셈블리의 모든 지역화된 .resources 파일과 긴밀하게 결합합니다.

가장 일반적으로 호출되는 두 생성자는 다음과 같습니다.

- ResourceManager(String, Assembly) 제공하는 두 가지 정보, 즉 .resources 파일의 기본 이름과 기본 .resources 파일이 있는 어셈블리를 기반으로 리소스를 조회합니다. 기본 이름에는 문화권 또는 확장명 없이 .resources 파일의 네임스페이스 및 루트 이름이 포함됩니다. 명령줄에서 컴파일된 .resources 파일은 일반적으로 네임스페이스 이름을 포함하지 않는 반면 Visual Studio 환경에서 만든 .resources 파일은 포함되지 않습니다. 예를 들어 리소스 파일 이름이 MyCompany.StringResources.resources이고 ResourceManager 생성자가 `Example.Main` 정적 메서드에서 호출되는 경우 다음 코드는 .resources 파일에서 리소스를 검색할 수 있는 ResourceManager 개체를 인스턴스화합니다.

  ```
  C#

  ResourceManager rm = new ResourceManager("MyCompany.StringResources",
                                           typeof(Example).Assembly);
  ```

- ResourceManager(Type) 형식 개체의 정보를 기반으로 위성 어셈블리에서 리소스를 조회합니다. 형식의 정규화된 이름은 파일 이름 확장명 없이 .resources 파일의 기본 이름에 해당합니다. Visual Studio 리소스 디자이너를 사용하여 만든 데스크톱 앱에서 Visual Studio는 정규화된 이름이 .resources 파일의 루트 이름과 동일한 래퍼 클래스를 만듭니다. 예를 들어 리소스 파일의 이름이 MyCompany.StringResources.resources이고 `MyCompany.StringResources` 래퍼 클래스가 있는 경우 다음 코드는 .resources 파일에서 리소스를 검색할 수 있는 ResourceManager 개체를 인스턴스화합니다.

  ```
  C#

  ResourceManager rm = new
  ResourceManager(typeof(MyCompany.StringResources));
  ```

적절한 리소스를 찾을 수 없는 경우 생성자 호출은 유효한 ResourceManager 개체를 만듭니다. 그러나 리소스를 가져오려는 시도에서 MissingManifestResourceException 예외가 발생합니다. 예외 처리에 대한 자세한 내용은 이 문서의 뒷부분에 나오는 Handle MissingManifestResourceException 및 MissingSatelliteAssemblyException 예외 섹션을 참조하세요.

다음 예제에서는 ResourceManager 개체를 인스턴스화하는 방법을 보여줍니다. ShowTime.exe실행 파일에 대한 소스 코드가 포함되어 있습니다. 파일 이름이 Strings.txt인 다음 텍스트 파일과 단일 문자열 리소스 `TimeHeader`이 포함되어 있습니다.

```
TimeHeader=The current time is
```

일괄 처리 파일을 사용하여 리소스 파일을 생성하고 실행 파일에 포함할 수 있습니다. 다음은 C# 컴파일러를 사용하여 실행 파일을 생성하는 일괄 처리 파일입니다.

```
resgen strings.txt
csc ShowTime.cs /resource:strings.resources
```

Visual Basic 컴파일러의 경우 다음 일괄 처리 파일을 사용할 수 있습니다.

```
resgen strings.txt
vbc ShowTime.vb /resource:strings.resources
```

C#

```
using System;
using System.Resources;

public class ShowTimeEx
{
    public static void Main()
    {
        ResourceManager rm = new ResourceManager("Strings",
                                 typeof(Example).Assembly);
        string timeString = rm.GetString("TimeHeader");
        Console.WriteLine($"{timeString} {DateTime.Now:T}");
    }
}
// The example displays output like the following:
//        The current time is 2:03:14 PM
```

ResourceManager 및 문화권별 리소스

지역화된 앱은 리소스를 배포해야 하며, 이는 문서 패키징 및 배포 리소스에서 설명한 대로입니다. 어셈블리가 제대로 구성된 경우 리소스 관리자는 현재 스레드의 Thread.CurrentUICulture 속성에 따라 검색할 리소스를 결정합니다. (이 속성은 현재 스레드의 UI 문화권도 반환합니다.) 예를 들어 앱이 주 어셈블리의 기본 영어 리소스와 두 위성 어셈블리의 프랑스어 및 러시아어 리소스로 컴파일되고 Thread.CurrentUICulture 속성이 fr-FR설정되면 리소스 관리자는 프랑스어 리소스를 검색합니다.

CurrentUICulture 속성을 명시적 또는 암시적으로 설정할 수 있습니다. 설정하는 방법은 ResourceManager 개체가 문화권에 따라 리소스를 검색하는 방법을 결정합니다.

- Thread.CurrentUICulture 속성을 특정 문화권으로 명시적으로 설정하는 경우 리소스 관리자는 사용자의 브라우저 또는 운영 체제 언어에 관계없이 항상 해당 문화권에 대한 리소스를 검색합니다. 기본 영어 리소스와 영어(미국), 프랑스어(프랑스) 및 러시아어(러시아)에 대한 리소스가 포함된 3개의 위성 어셈블리로 컴파일된 앱을 고려합니다. CurrentUICulture 속성이 fr-FR설정되면 ResourceManager 개체는 사용자의 운영 체제 언어가 프랑스어가 아니더라도 항상 프랑스어(프랑스) 리소스를 검색합니다. 속성을 명시적으로 설정하기 전에 이 동작이 원하는 동작인지 확인합니다.

  ASP.NET 앱에서는 서버의 설정이 들어오는 클라이언트 요청과 일치할 가능성이 낮기 때문에 Thread.CurrentUICulture 속성을 명시적으로 설정해야 합니다. ASP.NET 앱은 Thread.CurrentUICulture 속성을 사용자의 브라우저 수락 언어로 명시적으로 설정할 수 있습니다.

  Thread.CurrentUICulture 속성을 명시적으로 설정하면 해당 스레드의 현재 UI 문화권이 정의됩니다. 앱에 있는 다른 스레드의 현재 UI 문화권에는 영향을 주지 않습니다.

- 해당 문화권을 나타내는 CultureInfo 개체를 정적 CultureInfo.DefaultThreadCurrentUICulture 속성에 할당하여 앱 도메인에 있는 모든 스레드의 UI 문화권을 설정할 수 있습니다.

- 현재 UI 문화권을 명시적으로 설정하지 않고 현재 앱 도메인에 대한 기본 문화권을 정의하지 않는 경우 CultureInfo.CurrentUICulture 속성은 Windows `GetUserDefaultUILanguage` 함수에 의해 암시적으로 설정됩니다. 이 함수는 사용자가 기본 언어를 설정할 수 있도록 하는 MUI(다국어 사용자 인터페이스)에서 제공됩니다. 사용자가 UI 언어를 설정하지 않은 경우 기본적으로 운영 체제 리소스의 언어인 시스템 설치 언어로 설정됩니다.

다음은 현재 UI 문화권을 명시적으로 설정하는 간단한 "Hello world" 예제입니다. 영어(미국) 또는 en-US, 프랑스어(프랑스) 또는 fr-FR, 러시아어(러시아) 또는 ru-RU세 가지 문화권에 대한 리소스를 포함합니다. en-US 리소스는 Greetings.txt텍스트 파일에 포함됩니다.

```
HelloString=Hello world!
```

fr-FR 리소스는 Greetings.fr-FR.txt텍스트 파일에 포함됩니다.

```
HelloString=Salut tout le monde!
```

ru-RU 리소스는 Greetings.ru-RU.txt텍스트 파일에 포함됩니다.

```

HelloString=Всем привет!
```

다음은 예제의 소스 코드입니다(Visual Basic 버전의 경우 Example.vb 또는 C# 버전의 경우 Example.cs).

C#

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Create array of supported cultures
        string[] cultures = { "en-CA", "en-US", "fr-FR", "ru-RU" };
        Random rnd = new Random();
        int cultureNdx = rnd.Next(0, cultures.Length);
        CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;
        ResourceManager rm = new ResourceManager("Greetings",
typeof(Example).Assembly);
        try
        {
            CultureInfo newCulture = new CultureInfo(cultures[cultureNdx]);
            Thread.CurrentThread.CurrentCulture = newCulture;
            Thread.CurrentThread.CurrentUICulture = newCulture;
            string greeting = String.Format("The current culture is
{0}.\n{1}",

Thread.CurrentThread.CurrentUICulture.Name,
                                            rm.GetString("HelloString"));
            Console.WriteLine(greeting);
        }
        catch (CultureNotFoundException e)
        {
            Console.WriteLine($"Unable to instantiate culture
{e.InvalidCultureName}");
        }
        finally
        {
            Thread.CurrentThread.CurrentCulture = originalCulture;
            Thread.CurrentThread.CurrentUICulture = originalCulture;
```

```
        }
    }
}
// The example displays output like the following:
//       The current culture is ru-RU.
//       Всем привет!
```

이 예제를 컴파일하려면 다음 명령이 포함된 일괄 처리(.bat) 파일을 만들고 명령 프롬프트에서 실행합니다. C#을 사용하는 경우 `vbc` 대신 `csc` 지정하고 `Example.vb`대신 `Example.cs`.

```
resgen Greetings.txt
vbc Example.vb /resource:Greetings.resources

resgen Greetings.fr-FR.txt
Md fr-FR
al /embed:Greetings.fr-FR.resources /culture:fr-FR /out:fr-
FR\Example.resources.dll

resgen Greetings.ru-RU.txt
Md ru-RU
al /embed:Greetings.ru-RU.resources /culture:ru-RU /out:ru-
RU\Example.resources.dll
```

리소스 검색

GetObject(String) 및 GetString(String) 메서드를 호출하여 특정 리소스에 액세스합니다. GetStream(String) 메서드를 호출하여 문자열이 아닌 리소스를 바이트 배열로 검색할 수도 있습니다. 기본적으로 지역화된 리소스가 있는 앱에서 이러한 메서드는 호출을 수행한 스레드의 현재 UI 문화권에 의해 결정되는 문화권에 대한 리소스를 반환합니다. 스레드의 현재 UI 문화권이 정의되는 방법에 대한 자세한 내용은 ResourceManager 및 문화권별 리소스이전 섹션을 참조하세요. 리소스 관리자가 현재 스레드의 UI 문화권에 대한 리소스를 찾을 수 없는 경우 대체 프로세스를 사용하여 지정된 리소스를 검색합니다. 리소스 관리자가 지역화된 리소스를 찾을 수 없는 경우 기본 문화권의 리소스를 사용합니다. 리소스 대체 규칙에 대한 자세한 내용은 리소스 패키징 및 배포문서의 "리소스 대체 프로세스" 섹션을 참조하세요.

> **참고**
>
> ResourceManager 클래스 생성자에 지정된 .resources 파일을 찾을 수 없는 경우 리소스를 검색하려고 하면 MissingManifestResourceException 또는 MissingSatelliteAssemblyException 예외가 throw됩니다. 예외를 처리하는 방법에

대한 정보는 이 문서의 뒷부분에 나오는 Handle MissingManifestResourceException 및 MissingSatelliteAssemblyException 예외 섹션을 참조하세요.

다음 예제에서는 GetString 메서드를 사용하여 문화권별 리소스를 검색합니다. 영어(en), 프랑스어(프랑스) (fr-FR) 및 러시아어(러시아)(ru-RU) 문화권에 대한 .txt 파일에서 컴파일된 리소스로 구성됩니다. 이 예제에서는 현재 문화권과 현재 UI 문화권을 영어(미국), 프랑스어(프랑스), 러시아어(러시아) 및 스웨덴어(스웨덴)로 변경합니다. 그런 다음 GetString 메서드를 호출하여 지역화된 문자열을 검색합니다. 이 문자열은 현재 날짜 및 월과 함께 표시됩니다. 현재 UI 문화권이 스웨덴어(스웨덴)인 경우를 제외하고 출력에 적절한 지역화된 문자열이 표시됩니다. 스웨덴어 언어 리소스를 사용할 수 없으므로 앱은 대신 영어인 기본 문화권의 리소스를 사용합니다.

이 예제에서는 다음 표에 나열된 텍스트 기반 리소스 파일이 필요합니다. 각각에는 `DateStart`이라는 단일 문자열 리소스가 있습니다.

테이블 확장

문화	파일 이름	리소스 이름	리소스 값
en-US	DateStrings.txt	`DateStart`	현재는 다음과 입니다.
fr-FR	DateStrings.fr-FR.txt	`DateStart`	오늘은
ru-RU	DateStrings.ru-RU.txt	`DateStart`	Сегодня

다음은 예제의 소스 코드입니다(Visual Basic 버전에 대한 ShowDate.vb 또는 코드의 C# 버전에 대한 ShowDate.cs).

C#

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguage("en")]

public class ShowDateEx
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "fr-FR", "ru-RU", "sv-SE" };
        ResourceManager rm = new ResourceManager("DateStrings",
                                                 typeof(Example).Assembly);

        foreach (var cultureName in cultureNames)
```

```
        {
            CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
            Thread.CurrentThread.CurrentCulture = culture;
            Thread.CurrentThread.CurrentUICulture = culture;

            Console.WriteLine($"Current UI Culture:
{CultureInfo.CurrentUICulture.Name}");
            string dateString = rm.GetString("DateStart");
            Console.WriteLine($"{dateString} {DateTime.Now:M}.\n");
        }
    }
}
// The example displays output similar to the following:
//       Current UI Culture: en-US
//       Today is February 03.
//
//       Current UI Culture: fr-FR
//       Aujourd'hui, c'est le 3 février
//
//       Current UI Culture: ru-RU
//       Сегодня февраля 03.
//
//       Current UI Culture: sv-SE
//       Today is den 3 februari.
```

이 예제를 컴파일하려면 다음 명령이 포함된 일괄 처리 파일을 만들고 명령 프롬프트에서 실행합니다. C#을 사용하는 경우 `vbc` 대신 `csc` 지정하고 `showdate.vb`대신 `showdate.cs`.

```
resgen DateStrings.txt
vbc showdate.vb /resource:DateStrings.resources

md fr-FR
resgen DateStrings.fr-FR.txt
al /out:fr-FR\Showdate.resources.dll /culture:fr-FR /embed:DateStrings.fr-
FR.resources

md ru-RU
resgen DateStrings.ru-RU.txt
al /out:ru-RU\Showdate.resources.dll /culture:ru-RU /embed:DateStrings.ru-
RU.resources
```

현재 UI 문화권 이외의 특정 문화권의 리소스를 검색하는 방법에는 두 가지가 있습니다.

- GetString(String, CultureInfo), GetObject(String, CultureInfo)또는 GetStream(String, CultureInfo) 메서드를 호출하여 특정 문화권에 대한 리소스를 검색할 수 있습니다.

지역화된 리소스를 찾을 수 없는 경우 리소스 관리자는 리소스 대체 프로세스를 사용하여 적절한 리소스를 찾습니다.

- GetResourceSet 메서드를 호출하여 특정 문화권의 리소스를 나타내는 ResourceSet 개체를 가져올 수 있습니다. 메서드 호출에서 리소스 관리자가 지역화된 리소스를 찾을 수 없는 경우 부모 문화권에 대해 검색하는지 또는 단순히 기본 문화권의 리소스로 되돌아가는지 여부를 확인할 수 있습니다. 그런 다음 ResourceSet 메서드를 사용하여 이름으로 리소스(해당 문화권에 대해 지역화됨)에 액세스하거나 집합의 리소스를 열거할 수 있습니다.

MissingManifestResourceException 및 MissingSatelliteAssemblyException 예외 처리

특정 리소스를 검색하려고 하지만 리소스 관리자가 해당 리소스를 찾을 수 없고 기본 문화권이 정의되지 않았거나 기본 문화권의 리소스를 찾을 수 없는 경우 리소스 관리자는 주 어셈블리에서 리소스를 찾거나 위성 어셈블리에서 리소스를 찾을 것으로 예상되는 경우 MissingSatelliteAssemblyExceptionMissingManifestResourceException 예외를 throw 합니다. 예외는 ResourceManager 개체를 인스턴스화할 때가 아니라 GetString 또는 GetObject같은 리소스 검색 메서드를 호출할 때 throw됩니다.

예외는 일반적으로 다음 조건에서 던져집니다.

- 적절한 리소스 파일 또는 위성 어셈블리가 없습니다. 리소스 관리자가 앱의 기본 리소스가 주 앱 어셈블리에 포함될 것으로 예상하는 경우 해당 리소스가 없습니다. NeutralResourcesLanguageAttribute 특성이 앱의 기본 리소스가 위성 어셈블리에 있음을 나타내는 경우 해당 어셈블리를 찾을 수 없습니다. 앱을 컴파일할 때 리소스가 주 어셈블리에 포함되거나 필요한 위성 어셈블리가 생성되고 적절하게 이름이 지정되었는지 확인합니다. 해당 이름은 appName *.resources.dll*형식을 사용해야 하며, 리소스가 포함된 문화권의 이름을 따서 명명된 디렉터리에 있어야 합니다.

- 앱에 기본 또는 중립 문화권이 정의되지 않았습니다. 소스 코드 파일 또는 프로젝트 정보 파일(Visual Basic 앱의 경우 AssemblyInfo.vb 또는 C# 앱의 AssemblyInfo.cs) 파일에 NeutralResourcesLanguageAttribute 특성을 추가합니다.

- ResourceManager(String, Assembly) 생성자의 `baseName` 매개 변수는 .resources 파일의 이름을 지정하지 않습니다. 이름에는 리소스 파일의 정규화된 네임스페이스가 포함되어야 하지만 파일 이름 확장명은 포함하지 않아야 합니다. 일반적으로 Visual Studio에서 만든 리소스 파일에는 네임스페이스 이름이 포함되지만 명령 프롬프트에서 생성되고 컴파일된 리소스 파일은 포함되지 않습니다. 다음 유틸리티를 컴파일하고 실행하여 포함된 .resources 파일의 이름을 확인할 수 있습니다. 주 어셈블리 또는 위성 어셈블리의 이름을 명령줄 매개 변수로 허용하는 콘솔 앱입니다. 리소스 관리자가 리소스를 올바르게 식별할 수 있도록 `baseName` 매개 변수로 제공해야 하는 문자열을 표시합니다.

C#

```
using System;
using System.IO;
using System.Reflection;

public class Example0
{
   public static void Main()
   {
      if (Environment.GetCommandLineArgs().Length == 1) {
         Console.WriteLine("No filename.");
         return;
      }

      string filename = Environment.GetCommandLineArgs()[1].Trim();
      // Check whether the file exists.
      if (! File.Exists(filename)) {
         Console.WriteLine($"{filename} does not exist.");
         return;
      }

      // Try to load the assembly.
      Assembly assem = Assembly.LoadFrom(filename);
      Console.WriteLine($"File: {filename}");

      // Enumerate the resource files.
      string[] resNames = assem.GetManifestResourceNames();
      if (resNames.Length == 0)
         Console.WriteLine("   No resources found.");

      foreach (var resName in resNames)
         Console.WriteLine($"   Resource:
{resName.Replace(".resources", "")}");

      Console.WriteLine();
   }
}
```

애플리케이션의 현재 문화권을 명시적으로 변경하는 경우 리소스 관리자는 CultureInfo.CurrentCulture 속성이 아닌 CultureInfo.CurrentUICulture 속성의 값을 기반으로 리소스 집합을 검색한다는 점을 기억해야 합니다. 일반적으로 한 값을 변경하는 경우 다른 값도 변경해야 합니다.

리소스 버전 관리

앱의 기본 리소스를 포함하는 주 어셈블리는 앱의 위성 어셈블리와 별개이므로 위성 어셈블리를 다시 배포하지 않고도 새 버전의 주 어셈블리를 릴리스할 수 있습니다.

SatelliteContractVersionAttribute 특성을 사용하여 기존 위성 어셈블리를 사용하고 리소스 관리자에게 새 버전의 주 어셈블리로 다시 배포하지 않도록 지시합니다.

위성 어셈블리에 대한 버전 관리 지원에 대한 자세한 내용은 리소스 검색 문서를 참조하세요.

<satelliteassemblies> 구성 파일 노드

> **참고**
>
> 이 섹션은 .NET Framework 앱과 관련이 있습니다.

웹 사이트(HREF .exe 파일)에서 배포되고 실행되는 실행 파일의 경우 ResourceManager 개체는 웹을 통해 위성 어셈블리를 검색하여 앱의 성능을 저하할 수 있습니다. 성능 문제를 제거하기 위해 이 검색을 앱과 함께 배포한 위성 어셈블리로 제한할 수 있습니다. 이렇게 하려면 앱의 구성 파일에 `<satelliteassemblies>` 노드를 만들어 앱에 대한 특정 문화권 집합을 배포했으며 ResourceManager 개체가 해당 노드에 나열되지 않은 문화권을 검색하지 않도록 지정합니다.

> **참고**
>
> `<satelliteassemblies>` 노드를 만드는 기본 대안은 **ClickOnce 배포 매니페스트** 기능을 사용하는 것입니다.

앱의 구성 파일에서 다음과 유사한 섹션을 만듭니다.

XML

```
<?xml version ="1.0"?>
<configuration>
  <satelliteassemblies>
    <assembly name="MainAssemblyName, Version=versionNumber,
Culture=neutral, PublicKeyToken=null|yourPublicKeyToken">
      <culture>cultureName1</culture>
      <culture>cultureName2</culture>
      <culture>cultureName3</culture>
    </assembly>
  </satelliteassemblies>
</configuration>
```

다음과 같이 이 구성 정보를 편집합니다.

- 배포하는 각 주 어셈블리에 대해 하나 이상의 `<assembly>` 노드를 지정합니다. 여기서 각 노드는 정규화된 어셈블리 이름을 지정합니다. MainAssemblyName 대신 주 어셈블리의 이름을 지정하고 주 어셈블리에 해당하는 `Version`, `PublicKeyToken` 및 `Culture` 특성 값을 지정합니다.

  `Version` 특성의 경우 어셈블리의 버전 번호를 지정합니다. 예를 들어 어셈블리의 첫 번째 릴리스는 버전 번호 1.0.0.0일 수 있습니다.

  `PublicKeyToken` 특성의 경우 강력한 이름으로 어셈블리에 서명하지 않은 경우 `null` 키워드를 지정하거나 어셈블리에 서명한 경우 공개 키 토큰을 지정합니다.

  `Culture` 특성의 경우 주 어셈블리를 지정하고 ResourceManager 클래스가 `<culture>` 노드에 나열된 문화권에 대해서만 검색하도록 `neutral` 키워드를 지정합니다.

  정규화된 어셈블리 이름에 대한 자세한 내용은 문서 , 어셈블리 이름를 참조하세요. 강력한 이름의 어셈블리에 대한 자세한 내용은 강력한 이름의 어셈블리 만들기 및 사용문서를 참조하세요.

- "fr-FR"와 같은 특정 문화권 이름 또는 중립 문화권 이름(예: "fr")을 사용하여 하나 이상의 `<culture>` 노드를 지정합니다.

`<satelliteassemblies>` 노드 아래에 나열되지 않은 어셈블리에 리소스가 필요한 경우 ResourceManager 클래스는 표준 검색 규칙을 사용하여 문화권을 검색합니다.

Windows 8.x 앱

> ⓘ **중요**
>
> ResourceManager 클래스는 Windows 8.x 앱에서 지원되지만 사용하지 않는 것이 좋습니다. Windows 8.x 앱에서 사용할 수 있는 이식 가능한 클래스 라이브러리 프로젝트를 개발하는 경우에만 이 클래스를 사용합니다. Windows 8.x 앱에서 리소스를 검색하려면 대신 Windows.ApplicationModel.Resources.ResourceLoader 클래스를 사용합니다.

Windows 8.x 앱의 경우 ResourceManager 클래스는 PRI(패키지 리소스 인덱스) 파일에서 리소스를 검색합니다. 단일 PRI 파일(애플리케이션 패키지 PRI 파일)에는 기본 문화권과 지역화된 문화권 모두에 대한 리소스가 포함됩니다. MakePRI 유틸리티를 사용하여 XML 리소스(.resw) 형식인 하나 이상의 리소스 파일에서 PRI 파일을 만듭니다. Visual Studio 프로젝트에 포함된 리소스의 경우 Visual Studio는 PRI 파일을 자동으로 만들고 패키징하는

프로세스를 처리합니다. 그런 다음 .NET ResourceManager 클래스를 사용하여 앱 또는 라이브러리의 리소스에 액세스할 수 있습니다.

데스크톱 앱과 동일한 방식으로 Windows 8.x 앱에 대한 ResourceManager 개체를 인스턴스화할 수 있습니다.

그런 다음 검색할 리소스의 이름을 GetString(String) 메서드에 전달하여 특정 문화권의 리소스에 액세스할 수 있습니다. 기본적으로 이 메서드는 호출한 스레드의 현재 UI 문화권에 의해 결정되는 문화권에 대한 리소스를 반환합니다. 리소스의 이름과 리소스를 검색할 문화권을 나타내는 CultureInfo 개체를 GetString(String, CultureInfo) 메서드로 전달하여 특정 문화권에 대한 리소스를 검색할 수도 있습니다. 현재 UI 문화권 또는 지정된 문화권에 대한 리소스를 찾을 수 없는 경우 리소스 관리자는 UI 언어 대체 목록을 사용하여 적절한 리소스를 찾습니다.

예시

다음 예제에서는 명시적 문화권 및 암시적 현재 UI 문화권을 사용하여 주 어셈블리 및 위성 어셈블리에서 문자열 리소스를 가져오는 방법을 보여 줍니다. 자세한 내용은 위성 어셈블리 만들기 항목의 "전역 어셈블리 캐시에 설치되지 않은 위성 어셈블리의 디렉터리 위치" 섹션을 참조하세요.

이 예제를 실행하려면 다음을 수행합니다.

1. 앱 디렉터리에서 다음 리소스 문자열이 포함된 rmc.txt 파일을 만듭니다.

```
day=Friday
year=2006
holiday="Cinco de Mayo"
```

2. 리소스 파일 생성기 사용하여 다음과 같이 rmc.txt 입력 파일에서 rmc.resources 리소스 파일을 생성합니다.

```
resgen rmc.txt
```

3. 앱 디렉터리의 하위 디렉터리를 만들고 이름을 "es-MX"로 지정합니다. 다음 세 단계에서 만들 위성 어셈블리의 문화권 이름입니다.

4. es-MX 디렉터리에 다음 리소스 문자열이 포함된 rmc.es-MX.txt 파일을 만듭니다.

```
day=Viernes
year=2006
holiday="Cinco de Mayo"
```

5. rmc.es-MX.txt 입력 파일을 사용하여 리소스 파일 생성기로 rmc를 생성하고,es-MX.resources 리소스 파일을 다음과 같이 생성합니다.

```
resgen rmc.es-MX.txt
```

6. 이 예제의 파일 이름이 rmc.vb 또는 rmc.cs 가정합니다. 다음 소스 코드를 파일에 복사합니다. 그런 다음 컴파일하고 주 어셈블리 리소스 파일인 rmc.resources를 실행 어셈블리에 포함합니다. Visual Basic 컴파일러를 사용하는 경우 구문은 다음과 같습니다.

```
vbc rmc.vb /resource:rmc.resources
```

C# 컴파일러에 해당하는 구문은 다음과 같습니다.

```
csc /resource:rmc.resources rmc.cs
```

7. 어셈블리 링커 사용하여 위성 어셈블리를 만듭니다. 앱의 기본 이름이 rmc인 경우 위성 어셈블리 이름은 rmc.resources.dll합니다. 위성 어셈블리는 es-MX 디렉터리에 만들어야 합니다. es-MX 현재 디렉터리인 경우 다음 명령을 사용합니다.

```
al /embed:rmc.es-MX.resources /c:es-MX /out:rmc.resources.dll
```

8. rmc.exe 실행하여 포함된 리소스 문자열을 가져오고 표시합니다.

C#

```
using System;
using System.Globalization;
using System.Resources;

class Example2
{
```

```
    public static void Main()
    {
        string day;
        string year;
        string holiday;
        string celebrate = "{0} will occur on {1} in {2}.\n";

        // Create a resource manager.
        ResourceManager rm = new ResourceManager("rmc",
                                 typeof(Example).Assembly);

        Console.WriteLine("Obtain resources using the current UI
culture.");

        // Get the resource strings for the day, year, and holiday
        // using the current UI culture.
        day = rm.GetString("day");
        year = rm.GetString("year");
        holiday = rm.GetString("holiday");
        Console.WriteLine(celebrate, holiday, day, year);

        // Obtain the es-MX culture.
        CultureInfo ci = new CultureInfo("es-MX");

        Console.WriteLine("Obtain resources using the es-MX culture.");

        // Get the resource strings for the day, year, and holiday
        // using the specified culture.
        day = rm.GetString("day", ci);
        year = rm.GetString("year", ci);
        holiday = rm.GetString("holiday", ci);
        // ------------------------------------------------------------
---
        // Alternatively, comment the preceding 3 code statements and
        // uncomment the following 4 code statements:
        // ------------------------------------------------------------
----
        // Set the current UI culture to "es-MX" (Spanish-Mexico).
        //    Thread.CurrentThread.CurrentUICulture = ci;

        // Get the resource strings for the day, year, and holiday
        // using the current UI culture. Use those strings to
        // display a message.
        //    day  = rm.GetString("day");
        //    year = rm.GetString("year");
        //    holiday = rm.GetString("holiday");
        // ------------------------------------------------------------
---

        // Regardless of the alternative that you choose, display a
message
        // using the retrieved resource strings.
        Console.WriteLine(celebrate, holiday, day, year);
    }
}
```

```
/*
This example displays the following output:

   Obtain resources using the current UI culture.
   "5th of May" will occur on Friday in 2006.

   Obtain resources using the es-MX culture.
   "Cinco de Mayo" will occur on Viernes in 2006.
*/
```

System.Resources.ResourceManager 생성자

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

ResourceManager(Type) 생성자

이 섹션은 ResourceManager(Type) 생성자 오버로드와 관련이 있습니다.

데스크톱 앱

데스크톱 앱에서 리소스 관리자는 `resourceSource` 매개 변수를 사용하여 다음과 같이 특정 리소스 파일을 로드합니다.

- NeutralResourcesLanguageAttribute 특성을 사용하여 기본 문화권의 리소스가 위성 어셈블리에 상주함을 나타내지 않으면 리소스 관리자는 기본 문화권의 리소스 파일이 `resourceSource` 매개 변수에 지정된 형식과 동일한 어셈블리에 있다고 가정합니다.
- 리소스 관리자는 기본 리소스 파일이 `resourceSource` 매개 변수에 지정된 형식과 동일한 기본 이름을 가지고 있다고 가정합니다.
- 리소스 관리자는 기본 ResourceSet 클래스를 사용하여 리소스 파일을 조작합니다.

예를 들어 `MyCompany.MyProduct.MyType` 형식이 지정된 경우 리소스 관리자는 `MyType` 정의하는 어셈블리에서 *MyCompany.MyProduct.MyType.resources* 이름이 *.resources* 파일을 찾습니다.

Visual Studio에서 리소스 디자이너는 이름이 기본 문화권에 대한 *.resources* 파일의 기본 이름과 동일한 `internal` (C#) 또는 `Friend` (Visual Basic의 경우) 클래스를 정의하는 코드를 자동으로 생성합니다. 이렇게 하면 클래스가 컴파일러에 표시되는 한 리소스도 표시되어야 하므로 이름이 리소스 이름에 해당하는 형식 개체를 가져오면 ResourceManager 개체를 인스턴스화하고 특정 리소스 집합과 결합할 수 있습니다. 예를 들어 *.resources* 파일의 이름이 Resource1인 경우 다음 문은 ResourceManager 개체를 인스턴스화하여 Resource1이라는 *.resources* 파일을 관리합니다.

C#

```
ResourceManager rm = new ResourceManager(typeof(Resource1));
```

Visual Studio를 사용하지 않는 경우 네임스페이스와 이름이 기본 *.resources* 파일과 동일한 멤버가 없는 클래스를 만들 수 있습니다. 이 예제에서는 그림을 제공합니다.

Windows 8.x 앱

> **ⓘ 중요**
>
> ResourceManager 클래스는 Windows 8.x 앱에서 지원되지만 사용하지 않는 것이 좋습니다. Windows 8.x 앱에서 사용할 수 있는 이식 가능한 클래스 라이브러리 프로젝트를 개발하는 경우에만 이 클래스를 사용합니다. Windows 8.x 앱에서 리소스를 검색하려면 대신 Windows.ApplicationModel.Resources.ResourceLoader 클래스를 사용합니다.

Windows 8.x 앱에서 ResourceManager `resourceSource` 매개 변수를 사용하여 리소스 항목이 앱의 PRI(패키지 리소스 인덱스) 파일 내에 있을 수 있는 어셈블리, 기본 이름 및 네임스페이스를 유추합니다. 예를 들어 `MyAssembly` 정의된 `MyCompany.MyProduct.MyType` 형식이 지정된 경우 리소스 관리자는 `MyAssembly` 이라는 리소스 집합 식별자를 찾고 해당 리소스 집합 내에서 `MyCompany.MyProduct.MyType` 범위를 찾습니다. 리소스 관리자는 이 범위 내에서 기본 환경(현재 문화권, 현재 고대비 설정 등)에 따라 리소스 항목을 검색합니다.

예시

다음 예제에서는 ResourceManager(Type) 생성자를 사용하여 ResourceManager 개체를 인스턴스화합니다. 영어(en), 프랑스어(프랑스) (fr-FR) 및 러시아어(러시아)(ru-RU) 문화권에 대한 .txt 파일에서 컴파일된 리소스로 구성됩니다. 이 예제에서는 현재 문화권과 현재 UI 문화권을 영어(미국), 프랑스어(프랑스), 러시아어(러시아) 및 스웨덴어(스웨덴)로 변경합니다. 그런 다음 GetString(String) 메서드를 호출하여 현지화된 문자열을 검색합니다. 이 문자열은 하루 중 시간에 따라 인사말을 표시합니다.

이 예제에서는 다음 표에 나열된 대로 세 개의 텍스트 기반 리소스 파일이 필요합니다. 각 파일에는 `Morning`, `Afternoon` 및 `Evening` 문자열 리소스가 포함됩니다.

테이블 확장

문화	파일 이름	리소스 이름	리소스 값
en-US	GreetingResources.txt	`Morning`	안녕하세요
en-US	GreetingResources.txt	`Afternoon`	안녕하세요

문화	파일 이름	리소스 이름	리소스 값
en-US	GreetingResources.txt	`Evening`	안녕하세요
프랑스어(프랑스)	GreetingResources.fr-FR.txt	`Morning`	Bonjour
프랑스어(프랑스)	GreetingResources.fr-FR.txt	`Afternoon`	Bonjour
프랑스어(프랑스)	GreetingResources.fr-FR.txt	`Evening`	안녕하세요 좋은 저녁입니다
러시아어(ru-RU)	GreetingResources.ru-RU.txt	`Morning`	Доброе утро
러시아어(ru-RU)	GreetingResources.ru-RU.txt	`Afternoon`	Добрый день
러시아어(ru-RU)	GreetingResources.ru-RU.txt	`Evening`	Добрый вечер

다음 일괄 처리 파일을 사용하여 Visual Basic 예제를 컴파일하고 Greet.exe실행 파일을 만들 수 있습니다. C#으로 컴파일하려면 컴파일러 이름을 `vbc` `csc`, 파일 확장명을 `.vb` `.cs` 변경합니다.

```
resgen GreetingResources.txt
vbc Greet.vb /resource: GreetingResources.resources

md fr-FR
resgen GreetingResources.fr-FR.txt
al /out:fr-FR\Greet.resources.dll /culture:fr-FR /embed:
GreetingResources.fr-FR.resources

md ru-RU
resgen GreetingResources.ru-RU.txt
al /out:ru-RU\Greet.resources.dll /culture:ru-RU /embed:
GreetingResources.ru-RU.resources
```

다음은 예제의 소스 코드입니다(Visual Basic 버전에 대한 ShowDate.vb 또는 코드의 C# 버전에 대한 ShowDate.cs).

C#

```
using System;
using System.Resources;
using System.Globalization;
using System.Threading;

[assembly: NeutralResourcesLanguage("en")]

public class Example2
{
    public static void Main()
```

```
    {
        string[] cultureNames = [ "en-US", "fr-FR", "ru-RU", "sv-SE" ];
        DateTime noon = new DateTime(DateTime.Now.Year, DateTime.Now.Month,
                                     DateTime.Now.Day, 12, 0, 0);
        DateTime evening = new DateTime(DateTime.Now.Year,
DateTime.Now.Month,
                                        DateTime.Now.Day, 18, 0, 0);

        ResourceManager rm = new ResourceManager(typeof(GreetingResources));

        foreach (var cultureName in cultureNames)
        {
            Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture(cultureName);
            Console.WriteLine($"The current UI culture is
{CultureInfo.CurrentUICulture.Name}");
            if (DateTime.Now < noon)
                Console.WriteLine($"{rm.GetString("Morning")}!");
            else if (DateTime.Now < evening)
                Console.WriteLine($"{rm.GetString("Afternoon")}!");
            else
                Console.WriteLine($"{rm.GetString("Evening")}!");
            Console.WriteLine();
        }
    }

    internal class GreetingResources
    {
    }
}
// The example displays output like the following:
//       The current UI culture is en-US
//       Good afternoon!
//
//       The current UI culture is fr-FR
//       Bonjour!
//
//       The current UI culture is ru-RU
//       Добрый день!
//
//       The current UI culture is sv-SE
//       Good afternoon!
```

소스 코드는 `Example`라는 앱 클래스를 정의하는 것 외에도 이름이 `GreetingResources`리소스 파일의 기본 이름과 동일한 내부 클래스를 정의합니다. 이렇게 하면 ResourceManager(Type) 생성자를 호출하여 ResourceManager 개체를 성공적으로 인스턴스화할 수 있습니다.

현재 UI 문화권이 스웨덴어(스웨덴)인 경우를 제외하고 출력에 적절한 지역화된 문자열이 표시됩니다. 이 경우 영어 리소스를 사용합니다. 스웨덴어 언어 리소스를 사용할 수 없

으므로 앱은 NeutralResourcesLanguageAttribute 특성에 정의된 기본 문화권의 리소스를 대신 사용합니다.

ResourceManager(String, Assembly) 생성자

이 섹션은 ResourceManager(String, Assembly) 생성자 오버로드와 관련이 있습니다.

데스크톱 앱

데스크톱 앱에서 개별 문화권별 리소스 파일은 위성 어셈블리에 포함되어야 하며 기본 문화권의 리소스 파일은 주 어셈블리에 포함되어야 합니다. 위성 어셈블리는 해당 어셈블리의 매니페스트에 지정된 단일 문화권에 대한 리소스를 포함하는 것으로 간주되며 필요에 따라 로드됩니다.

> **참고**
>
> 어셈블리에서 리소스를 검색하는 대신 *.resources* 파일에서 직접 리소스를 검색하려면 ResourceManager 개체를 인스턴스화하는 대신 CreateFileBasedResourceManager 메서드를 호출해야 합니다.

`baseName` 식별된 리소스 파일을 `assembly`찾을 수 없는 경우 메서드는 ResourceManager 개체를 인스턴스화하지만 특정 리소스를 검색하려고 하면 예외가 발생하며 일반적으로 MissingManifestResourceException. 예외의 원인을 진단하는 방법에 대한 자세한 내용은 ResourceManager 클래스 항목의 "MissingManifestResourceException 예외 처리" 섹션을 참조하세요.

Windows 8.x 앱

> **중요**
>
> ResourceManager 클래스는 Windows 8.x 앱에서 지원되지만 사용하지 않는 것이 좋습니다. Windows 8.x 앱에서 사용할 수 있는 이식 가능한 클래스 라이브러리 프로젝트를 개발하는 경우에만 이 클래스를 사용합니다. Windows 8.x 앱에서 리소스를 검색하려면 대신 Windows.ApplicationModel.Resources.ResourceLoader 클래스를 사용합니다.

Windows 8.x 앱에서 리소스 관리자는 `assembly` 매개 변수의 간단한 이름을 사용하여 앱의 PRI(패키지 리소스 인덱스) 파일에서 일치하는 리소스 집합을 조회합니다. `baseName` 매개 변수는 리소스 집합 내에서 리소스 항목을 조회하는 데 사용됩니다. 예를 들어

PortableLibrary1.Resource1의 루트 이름입니다.de-DE.resources는 PortableLibrary1.Resource1입니다.

예시

다음 예제에서는 지역화되지 않은 간단한 "Hello World" 앱을 사용하여 ResourceManager(String, Assembly) 생성자를 보여 줍니다. *ExampleResources.txt*라는 텍스트 파일의 내용은 `Greeting=Hello`입니다. 앱이 컴파일되면 리소스가 주 앱 어셈블리에 포함됩니다.

텍스트 파일은 다음과 같이 명령 프롬프트에서 리소스 파일 생성기(ResGen.exe) 사용하여 이진 리소스 파일로 변환할 수 있습니다.

Windows 명령 프롬프트

```
resgen ExampleResources.txt
```

다음 예제에서는 ResourceManager 개체를 인스턴스화하고 사용자에게 이름을 입력하라는 메시지를 표시하고 인사말을 표시하는 실행 코드를 제공합니다.

C#

```
using System;
using System.Reflection;
using System.Resources;

public class Example1
{
    public static void Main()
    {
        // Retrieve the resource.
        ResourceManager rm = new ResourceManager("ExampleResources",
                                 typeof(Example).Assembly);
        string greeting = rm.GetString("Greeting");

        Console.Write("Enter your name: ");
        string name = Console.ReadLine();
        Console.WriteLine($"{greeting} {name}!");
    }
}
// The example produces output similar to the following:
//       Enter your name: John
//       Hello John!
```

C#에서 다음 명령을 사용하여 컴파일할 수 있습니다.

Windows 명령 프롬프트

```
csc Example.cs /resource:ExampleResources.resources
```

이 예제에서는 해당 어셈블리에 정의된 형식을 `typeof` 함수(C#) 또는 `GetType` 함수(Visual Basic)에 전달하고 해당 Type.Assembly 속성의 값을 검색하여 리소스 파일이 포함된 어셈블리에 대한 참조를 검색합니다.

System.Resources.ResourceManager.GetObject 메서드

아티클 • 2025. 04. 03.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 GetObject 메서드는 문자열이 아닌 리소스를 검색하는 데 사용됩니다. 여기에는 Int32 또는 Double과 같은 기본 데이터 형식에 속하는 값, 비트맵(예: System.Drawing.Bitmap 객체), 또는 사용자 지정 직렬화된 객체가 포함됩니다. 일반적으로 반환된 개체는 C#으로 캐스팅되거나(Visual Basic에서) 적절한 형식의 개체로 변환되어야 합니다.

이 속성은 IgnoreCase 리소스 이름과의 `name` 비교가 대/소문자를 구분하지 않는지 또는 대/소문자를 구분하는지 여부를 결정합니다(기본값).

> **참고**
>
> 이러한 메서드는 나열된 것보다 더 많은 예외를 throw할 수 있습니다. 이 문제가 발생할 수 있는 한 가지 이유는 이 메서드가 호출하는 메서드가 예외를 발생시키는 경우입니다. 예를 들어 위성 어셈블리를 배포하거나 설치하는 동안 오류가 발생하면 FileLoadException 예외가 throw되거나, 형식이 역직렬화될 때 사용자 정의 형식이 사용자 정의 예외를 throw하는 경우 SerializationException 예외가 throw될 수 있습니다.

GetObject(String) 메서드

반환된 리소스는 CultureInfo.CurrentUICulture 속성에 의해 정의된 현재 스레드의 UI 문화 설정에 맞게 지역화됩니다. 리소스가 해당 문화권에 대해 지역화되지 않은 경우 리소스 관리자는 대체 규칙을 사용하여 적절한 리소스를 로드합니다. ResourceManager 사용할 수 있는 지역화된 리소스 집합이 없으면 기본 문화권의 리소스가 사용됩니다. 기본 문화권에 대한 리소스 집합을 찾을 수 없는 경우 메서드는 예외를 MissingManifestResourceException throw하거나, 리소스 집합이 위성 어셈블리에 상주해야 하는 경우 예외를 MissingSatelliteAssemblyException throw합니다. 리소스 관리자가 적절한 리소스 집합을 로드할 수 있지만 `name` 리소스를 찾을 수 없는 경우 메서드는 `null` 반환합니다.

예시

다음 예제에서는 메서드를 GetObject(String) 사용하여 사용자 지정 개체를 역직렬화합니다. 이 예제에는 다음 구조체를 정의하는 UIElements.cs(Visual Basic을 사용하는 경우 UIElements.vb)라는 `PersonTable` 소스 코드 파일이 포함되어 있습니다. 이 구조체는 테이블 열의 지역화된 이름을 표시하는 일반 테이블 표시 루틴에서 사용됩니다. 구조체는 SerializableAttribute 특성에 의해 `PersonTable`로 표시됩니다.

C#

```
using System;

[Serializable] public struct PersonTable
{
   public readonly int nColumns;
   public readonly string column1;
   public readonly string column2;
   public readonly string column3;
   public readonly int width1;
   public readonly int width2;
   public readonly int width3;

   public PersonTable(string column1, string column2, string column3,
                      int width1, int width2, int width3)
   {
      this.column1 = column1;
      this.column2 = column2;
      this.column3 = column3;
      this.width1 = width1;
      this.width2 = width2;
      this.width3 = width3;
      this.nColumns = typeof(PersonTable).GetFields().Length / 2;
   }
}
```

CreateResources.cs(또는 Visual Basic용 CreateResources.vb)라는 파일의 다음 코드는 테이블 제목과 `PersonTable` 영어로 지역화된 앱에 대한 정보를 포함하는 개체를 저장하는 UIResources.resx라는 XML 리소스 파일을 만듭니다.

C#

```
using System;
using System.Resources;

public class CreateResource
{
   public static void Main()
   {
      PersonTable table = new PersonTable("Name", "Employee Number",
                                          "Age", 30, 18, 5);
      ResXResourceWriter rr = new ResXResourceWriter(@".\UIResources.resx");
      rr.AddResource("TableName", "Employees of Acme Corporation");
```

```
      rr.AddResource("Employees", table);
      rr.Generate();
      rr.Close();
   }
}
```

GetObject.cs(또는 GetObject.vb)라는 소스 코드 파일의 다음 코드는 리소스를 검색하여 콘솔에 표시합니다.

C#

```
using System;
using System.Resources;

[assembly: NeutralResourcesLanguageAttribute("en")]

public class Example3
{
   public static void Main()
   {
      string fmtString = String.Empty;
      ResourceManager rm = new ResourceManager("UIResources",
typeof(Example).Assembly);
      string title = rm.GetString("TableName");
      PersonTable tableInfo = (PersonTable) rm.GetObject("Employees");

      if (! String.IsNullOrEmpty(title)) {
         fmtString = "{0," + ((Console.WindowWidth + title.Length) /
2).ToString() + "}";
         Console.WriteLine(fmtString, title);
         Console.WriteLine();
      }

      for (int ctr = 1; ctr <= tableInfo.nColumns; ctr++) {
         string columnName = "column"  + ctr.ToString();
         string widthName = "width" + ctr.ToString();
         string value =
tableInfo.GetType().GetField(columnName).GetValue(tableInfo).ToString();
         int width = (int)
tableInfo.GetType().GetField(widthName).GetValue(tableInfo);
         fmtString = "{0,-" + width.ToString() + "}";
         Console.Write(fmtString, value);
      }
      Console.WriteLine();
   }
}
```

필요한 리소스 파일 및 어셈블리를 빌드하고 다음 일괄 처리 파일을 실행하여 앱을 실행할 수 있습니다. 구조 `PersonTable`에 관한 정보를 참조할 수 있도록, UIElements.dll에 대

한 참조를 Resgen.exe에 제공하는 옵션 `/r`을 사용해야 합니다. C#을 사용하는 경우, 컴파일러 이름을 `vbc`에서 `csc`으로 바꾸고, 확장자를 `.vb`에서 `.cs`으로 바꾸십시오.

```
vbc /t:library UIElements.vb
vbc CreateResources.vb /r:UIElements.dll
CreateResources

resgen UIResources.resx  /r:UIElements.dll
vbc GetObject.vb /r:UIElements.dll /resource:UIResources.resources

GetObject.exe
```

GetObject(String, CultureInfo) 메서드

반환된 리소스는 `culture`로 지정된 문화권에 맞게 지역화되며, `culture`가 `null`인 경우, CultureInfo.CurrentUICulture 속성에 지정된 문화권에 맞게 지역화됩니다. 리소스가 해당 문화권에 대해 지역화되지 않은 경우 리소스 관리자는 대체 규칙을 사용하여 적절한 리소스를 로드합니다. 사용할 수 있는 지역화된 리소스 집합이 없으면 리소스 관리자는 기본 문화권의 리소스를 다시 사용합니다. 기본 문화권에 대한 리소스 집합을 찾을 수 없는 경우 메서드는 예외를 MissingManifestResourceException throw하거나, 리소스 집합이 위성 어셈블리에 상주해야 하는 경우 예외를 MissingSatelliteAssemblyException throw합니다. 리소스 관리자가 적절한 리소스 집합을 로드할 수 있지만 `name`리소스를 찾을 수 없는 경우 메서드는 `null`반환합니다.

예시

다음 예제에서는 메서드를 GetObject(String, CultureInfo) 사용하여 사용자 지정 개체를 역직렬화합니다. 이 예제에는 다음 구조체를 정의하는 NumberInfo.cs(Visual Basic을 사용하는 경우 NumberInfo.vb)라는 `Numbers`소스 코드 파일이 포함되어 있습니다. 이 시스템은 비영어권 학생들에게 영어로 숫자 10까지 세는 법을 배우게 하는 간단한 교육 앱에서 사용됩니다. `Numbers` 클래스는 SerializableAttribute 특성으로 표시되어 있습니다.

C#

```
using System;

[Serializable] public class Numbers2
{
   public readonly string One;
   public readonly string Two;
   public readonly string Three;
   public readonly string Four;
```

```
   public readonly string Five;
   public readonly string Six;
   public readonly string Seven;
   public readonly string Eight;
   public readonly string Nine;
   public readonly string Ten;

   public Numbers2(string one, string two, string three, string four,
                   string five, string six, string seven, string eight,
                   string nine, string ten)
   {
      this.One = one;
      this.Two = two;
      this.Three = three;
      this.Four = four;
      this.Five = five;
      this.Six = six;
      this.Seven = seven;
      this.Eight = eight;
      this.Nine = nine;
      this.Ten = ten;
   }
}
```

CreateResources.cs(Visual Basic용 CreateResources.vb)라는 파일의 다음 소스 코드는 기본 영어 및 프랑스어, 포르투갈어 및 러시아어 언어에 대한 XML 리소스 파일을 만듭니다.

C#

```
using System;
using System.Resources;

public class CreateResource
{
   public static void Main()
   {
      Numbers en = new Numbers("one", "two", "three", "four", "five",
                               "six", "seven", "eight", "nine", "ten");
      CreateResourceFile(en, "en");
      Numbers fr = new Numbers("un", "deux", "trois", "quatre", "cinq",
                               "six", "sept", "huit", "neuf", "dix");
      CreateResourceFile(fr, "fr");
      Numbers pt = new Numbers("um", "dois", "três", "quatro", "cinco",
                               "seis", "sete", "oito", "nove", "dez");
      CreateResourceFile(pt, "pt");
      Numbers ru = new Numbers("один", "два", "три", "четыре", "пять",
                               "шесть", "семь", "восемь", "девять",
"десять");
      CreateResourceFile(ru, "ru");
   }

   public static void CreateResourceFile(Numbers n, string lang)
   {
```

```
      string filename = @".\NumberResources" +
                        (lang != "en" ? "." + lang : "" ) +
                        ".resx";
      ResXResourceWriter rr = new ResXResourceWriter(filename);
      rr.AddResource("Numbers", n);
      rr.Generate();
      rr.Close();
   }
}
```

리소스는 현재 UI 문화권을 프랑스어(프랑스), 포르투갈어(브라질) 또는 러시아어(러시아)로 설정하는 다음 앱에서 사용됩니다. 이 메서드는 GetObject(String) 지역화된 숫자가 포함된 개체를 가져오는 `Numbers` 메서드와 GetObject(String, CultureInfo) 영어 번호가 포함된 개체를 `Numbers` 가져오는 메서드를 호출합니다. 그런 다음 현재 UI 문화권과 영어를 사용하여 홀수 숫자를 표시합니다. 소스 코드 파일의 이름은 ShowNumbers.cs(ShowNumbers.vb)입니다.

C#

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly:NeutralResourcesLanguageAttribute("en-US")]

public class Example
{
   static string[] cultureNames = [ "fr-FR", "pt-BR", "ru-RU" ];

   public static void Main()
   {
      // Make any non-default culture the current culture.
      Random rnd = new Random();
      CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureNames[rnd.Next(0,
cultureNames.Length)]);
      Thread.CurrentThread.CurrentUICulture = culture;
      Console.WriteLine($"The current culture is
{CultureInfo.CurrentUICulture.Name}\n");
      CultureInfo enCulture = CultureInfo.CreateSpecificCulture("en-US");

      ResourceManager rm = new ResourceManager(typeof(NumberResources));
      Numbers numbers = (Numbers) rm.GetObject("Numbers");
      Numbers numbersEn = (Numbers) rm.GetObject("Numbers", enCulture);
      Console.WriteLine($"{numbers.One} --> {numbersEn.One}");
      Console.WriteLine($"{numbers.Three} --> {numbersEn.Three}");
      Console.WriteLine($"{numbers.Five} --> {numbersEn.Five}");
      Console.WriteLine($"{numbers.Seven} --> {numbersEn.Seven}");
      Console.WriteLine($"{numbers.Nine} --> {numbersEn.Nine}\n");
   }
```

```
}

internal class NumberResources
{
}
// The example displays output like the following:
//       The current culture is pt-BR
//
//       um --> one
//       três --> three
//       cinco --> five
//       sete --> seven
//       nove --> nine
```

다음 일괄 처리 파일을 사용하여 Visual Basic 버전의 예제를 빌드하고 실행할 수 있습니다. C#을 사용하는 경우 `vbc`을 `csc`으로 바꾸고, `.vb` 확장을 `.cs`으로 바꾸십시오.

```
vbc /t:library NumberInfo.vb

vbc CreateResources.vb /r:NumberInfo.dll
CreateResources

resgen NumberResources.resx /r:NumberInfo.dll

resgen NumberResources.fr.resx /r:Numberinfo.dll
Md fr
al /embed:NumberResources.fr.resources /culture:fr /t:lib
/out:fr\ShowNumbers.resources.dll

resgen NumberResources.pt.resx  /r:Numberinfo.dll
Md pt
al /embed:NumberResources.pt.resources /culture:pt /t:lib
/out:pt\ShowNumbers.resources.dll

resgen NumberResources.ru.resx /r:Numberinfo.dll
Md ru
al /embed:NumberResources.ru.resources /culture:ru /t:lib
/out:ru\ShowNumbers.resources.dll

vbc ShowNumbers.vb /r:NumberInfo.dll /resource:NumberResources.resources
ShowNumbers.exe
```

성능 고려 사항

동일한 `name` 매개 변수를 사용하여 메서드를 GetObject 여러 번 호출하는 경우 각 호출을 통해 동일한 개체에 대한 참조를 반환하는 메서드에 의존하지 마세요. 메서드가 캐시의

GetObject 기존 리소스 개체에 대한 참조를 반환하거나 리소스를 다시 로드하고 새 리소스 개체에 대한 참조를 반환할 수 있기 때문입니다.

System.Resources.ResourceManager.GetString 메서드

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

IgnoreCase 속성은 리소스 이름과 `name` 비교가 대/소문자를 구분하지 않는지(기본값) 또는 대/소문자를 구분하는지를 결정합니다.

> **참고**
>
> GetString 메서드는 나열된 것보다 더 많은 예외를 throw할 수 있습니다. 이 문제가 발생할 수 있는 한 가지 이유는 이 메서드가 호출하는 메서드가 예외를 발생시키는 경우입니다. 예를 들어 위성 어셈블리를 배포하거나 설치하는 동안 오류가 발생하면 FileLoadException 예외가 throw되거나, 형식이 역직렬화될 때 사용자 정의 형식이 사용자 정의 예외를 throw하는 경우 SerializationException 예외가 throw될 수 있습니다.

GetString(String) 메서드

데스크톱 앱

데스크톱 앱에서 반환되는 리소스는 CultureInfo.CurrentUICulture 속성에 정의된 대로 현재 스레드의 UI 문화권에 대해 지역화됩니다. 리소스가 해당 문화권에 대해 지역화되지 않은 경우 리소스 관리자는 패키징 및 리소스 배포 문서의 "리소스 대체 프로세스" 섹션에 설명된 단계에 따라 리소스를 검색합니다. 사용할 수 있는 지역화된 리소스 집합이 없으면 리소스 관리자는 기본 문화권의 리소스를 다시 사용합니다. 리소스 관리자가 기본 문화권의 리소스 집합을 로드할 수 없는 경우 메서드는 MissingManifestResourceException 예외를 throw하거나, 리소스 집합이 위성 어셈블리에 상주해야 하는 경우 MissingSatelliteAssemblyException 예외입니다. 리소스 관리자가 적절한 리소스 집합을 로드할 수 있지만 `name` 리소스를 찾을 수 없는 경우 메서드는 `null` 반환합니다.

Windows 8.x 앱

> **중요**

ResourceManager 클래스는 Windows 8.x 앱에서 지원되지만 사용하지 않는 것이 좋습니다. Windows 8.x 앱에서 사용할 수 있는 이식 가능한 클래스 라이브러리 프로젝트를 개발하는 경우에만 이 클래스를 사용합니다. Windows 8.x 앱에서 리소스를 검색하려면 대신 Windows.ApplicationModel.Resources.ResourceLoader 클래스를 사용합니다.

Windows 8.x 앱에서 GetString(String) 메서드는 호출자의 현재 UI 문화권 설정에 대해 지역화된 `name` 문자열 리소스의 값을 반환합니다. 문화권 목록은 운영 체제의 기본 UI 언어 목록에서 파생됩니다. 리소스 관리자가 `name`일치시킬 수 없으면 메서드는 `null`반환합니다.

예시

다음 예제에서는 GetString 메서드를 사용하여 문화권별 리소스를 검색합니다. 영어(en), 프랑스어(프랑스) (fr-FR) 및 러시아어(러시아)(ru-RU) 문화권에 대한 .txt 파일에서 컴파일된 리소스로 구성됩니다. 이 예제에서는 현재 문화권과 현재 UI 문화권을 영어(미국), 프랑스어(프랑스), 러시아어(러시아) 및 스웨덴어(스웨덴)로 변경합니다. 그런 다음 GetString 메서드를 호출하여 지역화된 문자열을 검색합니다. 이 문자열은 현재 날짜 및 월과 함께 표시됩니다. 현재 UI 문화권이 스웨덴어(스웨덴)인 경우를 제외하고 출력에 적절한 지역화된 문자열이 표시됩니다. 스웨덴어 언어 리소스를 사용할 수 없으므로 앱은 대신 영어인 기본 문화권의 리소스를 사용합니다. 이 예제에서는 다음 표에 나열된 텍스트 기반 리소스 파일이 필요합니다. 각각에는 `DateStart`이라는 단일 문자열 리소스가 있습니다.

테이블 확장

문화	파일 이름	리소스 이름	리소스 값
en-US	DateStrings.txt	`DateStart`	현재는 다음과 입니다.
프랑스어(프랑스)	DateStrings.fr-FR.txt	`DateStart`	오늘은
러시아어(ru-RU)	DateStrings.ru-RU.txt	`DateStart`	Сегодня

다음 일괄 처리 파일을 사용하여 C# 예제를 컴파일할 수 있습니다. Visual Basic의 경우 `csc` `vbc`변경하고 소스 코드 파일의 확장명은 `.cs` `.vb`변경합니다.

```
resgen DateStrings.txt
csc showdate.cs /resource:DateStrings.resources

md fr-FR
```

```
resgen DateStrings.fr-FR.txt
al /out:fr-FR\Showdate.resources.dll /culture:fr-FR /embed:DateStrings.fr-
FR.resources

md ru-RU
resgen DateStrings.ru-RU.txt
al /out:ru-RU\Showdate.resources.dll /culture:ru-RU /embed:DateStrings.ru-
RU.resources
```

다음은 예제의 소스 코드입니다(Visual Basic 버전에 대한 ShowDate.vb 또는 C# 버전의 ShowDate.cs).

C#

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = [ "en-US", "fr-FR", "ru-RU", "sv-SE" ];
      ResourceManager rm = new ResourceManager("DateStrings",
                                               typeof(Example).Assembly);

      foreach (var cultureName in cultureNames) {
         CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
         Thread.CurrentThread.CurrentCulture = culture;
         Thread.CurrentThread.CurrentUICulture = culture;

         Console.WriteLine($"Current UI Culture:
{CultureInfo.CurrentUICulture.Name}");
         string dateString = rm.GetString("DateStart");
         Console.WriteLine($"{dateString} {DateTime.Now:M}.\n");
      }
   }
}
// The example displays output similar to the following:
//       Current UI Culture: en-US
//       Today is February 03.
//
//       Current UI Culture: fr-FR
//       Aujourd'hui, c'est le 3 février
//
//       Current UI Culture: ru-RU
//       Сегодня февраля 03.
//
//       Current UI Culture: sv-SE
//       Today is den 3 februari.
```

GetString(String, CultureInfo) 메서드

데스크톱 앱

데스크톱 앱에서 `culture` `null`경우 GetString(String, CultureInfo) 메서드는 CultureInfo.CurrentUICulture 속성에서 가져온 현재 UI 문화권을 사용합니다.

반환되는 리소스는 `culture` 매개 변수로 지정된 문화권에 대해 지역화됩니다. 리소스가 `culture`대해 지역화되지 않은 경우 리소스 관리자는 패키징 및 리소스 배포 항목의 "리소스 대체 프로세스" 섹션에 설명된 단계에 따라 리소스를 검색합니다. 사용할 수 있는 리소스 집합을 찾을 수 없는 경우 리소스 관리자는 기본 문화권의 리소스로 돌아갑니다. 리소스 관리자가 기본 문화권의 리소스 집합을 로드할 수 없는 경우 메서드는 MissingManifestResourceException 예외를 throw하거나, 리소스 집합이 위성 어셈블리에 상주해야 하는 경우 MissingSatelliteAssemblyException 예외입니다. 리소스 관리자가 적절한 리소스 집합을 로드할 수 있지만 `name`리소스를 찾을 수 없는 경우 메서드는 `null`반환합니다.

Windows 8.x 앱

> **ⓘ 중요**
>
> ResourceManager 클래스는 Windows 8.x 앱에서 지원되지만 사용하지 않는 것이 좋습니다. Windows 8.x 앱에서 사용할 수 있는 이식 가능한 클래스 라이브러리 프로젝트를 개발하는 경우에만 이 클래스를 사용합니다. Windows 8.x 앱에서 리소스를 검색하려면 대신 Windows.ApplicationModel.Resources.ResourceLoader 클래스를 사용합니다.

Windows 8.x 앱에서 GetString(String, CultureInfo) 메서드는 `culture` 매개 변수로 지정된 문화권에 대해 지역화된 `name` 문자열 리소스의 값을 반환합니다. 리소스가 `culture` 문화권에 대해 지역화되지 않은 경우 조회는 전체 Windows 8 언어 대체 목록을 사용하고 기본 문화권을 살펴본 후 중지됩니다. 리소스 관리자가 `name`일치시킬 수 없으면 메서드는 `null`반환합니다.

예시

다음 예제에서는 GetString(String, CultureInfo) 메서드를 사용하여 문화권별 리소스를 검색합니다. 이 예제의 기본 문화권은 영어(en)이며 프랑스어(프랑스)(fr-FR) 및 러시아어(러시아)(ru-RU) 문화권에 대한 위성 어셈블리를 포함합니다. 이 예제에서는 GetString(String, CultureInfo)호출하기 전에 현재 문화권 및 현재 UI 문화권을 러시아어

(러시아)로 변경합니다. 그런 다음 GetString 메서드와 DateTime.ToString(String, IFormatProvider) 메서드를 호출하고 프랑스(프랑스) 및 스웨덴(스웨덴) 문화권을 나타내는 CultureInfo 개체를 각 메서드에 전달합니다. 출력에서는 GetString 메서드가 프랑스어 리소스를 검색할 수 있기 때문에 해당 월과 일 및 그 앞에 오는 문자열이 프랑스어로 표시됩니다. 그러나 스웨덴어(스웨덴) 문화권이 사용되는 경우 월과 날짜는 스웨덴어로 표시되지만 앞에 오는 문자열은 영어로 표시됩니다. 이는 리소스 관리자가 지역화된 스웨덴어 리소스를 찾을 수 없기 때문에 기본 영어 문화권에 대한 리소스를 대신 반환하기 때문입니다.

이 예제에서는 다음 표에 나열된 텍스트 기반 리소스 파일이 필요합니다. 각각에는 `DateStart`이라는 단일 문자열 리소스가 있습니다.

테이블 확장

문화	파일 이름	리소스 이름	리소스 값
en-US	DateStrings.txt	`DateStart`	현재는 다음과 입니다.
프랑스어(프랑스)	DateStrings.fr-FR.txt	`DateStart`	오늘은
러시아어(ru-RU)	DateStrings.ru-RU.txt	`DateStart`	Сегодня

다음 일괄 처리 파일을 사용하여 Visual Basic 예제를 컴파일할 수 있습니다. C#으로 컴파일하려면 `vbc` `csc`변경하고 소스 코드 파일의 확장을 `.vb` `.cs`변경합니다.

```
resgen DateStrings.txt
vbc showdate.vb /resource:DateStrings.resources

md fr-FR
resgen DateStrings.fr-FR.txt
al /out:fr-FR\Showdate.resources.dll /culture:fr-FR /embed:DateStrings.fr-FR.resources

md ru-RU
resgen DateStrings.ru-RU.txt
al /out:ru-RU\Showdate.resources.dll /culture:ru-RU /embed:DateStrings.ru-RU.resources
```

다음은 예제의 소스 코드입니다(Visual Basic 버전에 대한 ShowDate.vb 또는 C# 버전의 ShowDate.cs).

C#

```
using System;
using System.Globalization;
```

```
using System.Resources;
using System.Threading;

public class Example2
{
    public static void Main()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("ru-RU");
        Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture("ru-RU");

        string[] cultureNames = [ "fr-FR", "sv-SE" ];
        ResourceManager rm = new ResourceManager("DateStrings",
                                                 typeof(Example).Assembly);

        foreach (var cultureName in cultureNames)
        {
            CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
            string dateString = rm.GetString("DateStart", culture);
            Console.WriteLine($"{culture.DisplayName}: {dateString}
{DateTime.Now.ToString("M", culture)}.");
            Console.WriteLine();
        }
    }
}
// The example displays output similar to the following:
//       French (France): Aujourd'hui, c'est le 7 février.
//
//       Swedish (Sweden): Today is den 7 februari.
```

System.Resources.ResourceReader 클래스

아티클 • 2025. 03. 29.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> **ⓘ 중요**
>
> 신뢰할 수 없는 데이터를 사용하여 이 클래스에서 메서드를 호출하는 것은 보안 위험입니다. 신뢰할 수 있는 데이터로만 이 클래스의 메서드를 호출합니다. 자세한 내용은 **모든 입력 유효성 검사** 참조하세요.

ResourceReader 클래스는 IResourceReader 인터페이스의 표준 구현을 제공합니다. ResourceReader 인스턴스는 독립 실행형 .resources 파일 또는 어셈블리에 포함된 .resources 파일을 나타냅니다. .resources 파일에서 리소스를 열거하고 해당 이름/값 쌍을 검색하는 데 사용됩니다. 어셈블리에 포함된 .resources 파일에서 지정된 명명된 리소스를 검색하는 데 사용되는 ResourceManager 클래스와 다릅니다. ResourceManager 클래스는 이름이 미리 알려진 리소스를 검색하는 데 사용되는 반면, ResourceReader 클래스는 컴파일 시간에 개수 또는 정확한 이름을 알 수 없는 리소스를 검색하는 데 유용합니다. 예를 들어 애플리케이션은 리소스 파일을 사용하여 섹션의 섹션 및 항목으로 구성된 구성 정보를 저장할 수 있습니다. 여기서 섹션의 섹션 또는 항목 수는 미리 알 수 없습니다. 그런 다음 `Section1`, `Section1Item1`, `Section1Item2`등과 같이 리소스의 이름을 일반적으로 지정하고 ResourceReader 개체를 사용하여 검색할 수 있습니다.

> **ⓘ 중요**
>
> 이 형식은 IDisposable 인터페이스를 구현합니다. 형식 사용을 마쳤으면 직접 또는 간접적으로 삭제해야 합니다. 형식을 직접 삭제하려면 `try`/`catch` 블록에서 해당 Dispose 메서드를 호출합니다. 간접적으로 삭제하려면 `using`(C#) 또는 `Using`(Visual Basic)와 같은 언어 구문을 사용합니다. 자세한 내용은 IDisposable 인터페이스 설명서의 "IDisposable을 구현하는 개체 사용" 섹션을 참조하세요.

ResourceReader 개체 인스턴스화

.resources 파일은 Resgen.exe(리소스 파일 생성기)텍스트 파일 또는 XML .resx 파일에서 컴파일된 이진 파일입니다. ResourceReader 개체는 독립 실행형 .resources 파일 또는 어셈블리에 포함된 .resources 파일을 나타낼 수 있습니다.

독립 실행형 .resources 파일에서 읽는 ResourceReader 개체를 인스턴스화하려면 입력 스트림 또는 .resources 파일 이름이 포함된 문자열과 함께 ResourceReader 클래스 생성자를 사용합니다. 다음 예제에서는 두 가지 방법을 모두 보여 줍니다. 첫 번째는 파일 이름을 사용하여 `Resources1.resources` .resources 파일을 나타내는 ResourceReader 개체를 인스턴스화합니다. 두 번째는 파일에서 만든 스트림을 사용하여 `Resources2.resources` .resources 파일을 나타내는 ResourceReader 개체를 인스턴스화합니다.

C#

```
// Instantiate a standalone .resources file from its filename.
var rr1 = new System.Resources.ResourceReader("Resources1.resources");

// Instantiate a standalone .resources file from a stream.
var fs = new System.IO.FileStream(@".\Resources2.resources",
                                  System.IO.FileMode.Open);
var rr2 = new System.Resources.ResourceReader(fs);
```

포함된 .resources 파일을 나타내는 ResourceReader 개체를 만들려면 .resources 파일이 포함된 어셈블리에서 Assembly 개체를 인스턴스화합니다. 해당 Assembly.GetManifestResourceStream 메서드는 ResourceReader(Stream) 생성자에 전달할 수 있는 Stream 개체를 반환합니다. 다음 예제에서는 포함된 .resources 파일을 나타내는 ResourceReader 개체를 인스턴스화합니다.

C#

```
System.Reflection.Assembly assem =
            System.Reflection.Assembly.LoadFrom(@".\MyLibrary.dll");
System.IO.Stream fs =

assem.GetManifestResourceStream("MyCompany.LibraryResources.resources");
var rr = new System.Resources.ResourceReader(fs);
```

ResourceReader 개체의 리소스 열거

.resources 파일에서 리소스를 열거하려면 System.Collections.IDictionaryEnumerator 개체를 반환하는 GetEnumerator 메서드를 호출합니다. `IDictionaryEnumerator.MoveNext` 메서드를 호출하여 한 리소스에서 다음 리소스로 이동합니다. .resources 파일의 모든 리소스가 열거되면 메서드는 `false` 반환합니다.

> **참고**
>
> ResourceReader 클래스는 IEnumerable 인터페이스 및 IEnumerable.GetEnumerator 메서드를 구현하지만

ResourceReader.GetEnumerator 메서드는 IEnumerable.GetEnumerator 구현을 제공하지 않습니다. 대신 ResourceReader.GetEnumerator 메서드는 각 리소스의 이름/값 쌍에 대한 액세스를 제공하는 IDictionaryEnumerator 인터페이스 개체를 반환합니다.

다음 두 가지 방법으로 컬렉션의 개별 리소스를 검색할 수 있습니다.

- System.Collections.IDictionaryEnumerator 컬렉션의 각 리소스를 반복하고 System.Collections.IDictionaryEnumerator 속성을 사용하여 리소스 이름과 값을 검색할 수 있습니다. 모든 리소스가 동일한 형식이거나 각 리소스의 데이터 형식을 알고 있는 경우 이 기술을 사용하는 것이 좋습니다.

- System.Collections.IDictionaryEnumerator 컬렉션을 반복하고 GetResourceData 메서드를 호출하여 리소스의 데이터를 검색할 때 각 리소스의 이름을 검색할 수 있습니다. 데이터 형식을 알지 못하거나 이전 방법에서 예외가 발생할 경우 이 방식을 사용하는 것이 좋습니다.

IDictionaryEnumerator 속성을 사용하여 리소스 검색

.resources 파일에서 리소스를 열거하는 첫 번째 방법은 각 리소스의 이름/값 쌍을 직접 검색하는 것입니다. `IDictionaryEnumerator.MoveNext` 메서드를 호출하여 컬렉션의 각 리소스로 이동한 후 IDictionaryEnumerator.Key 속성에서 리소스 이름과 IDictionaryEnumerator.Value 속성의 리소스 데이터를 검색할 수 있습니다.

다음 예제에서는 IDictionaryEnumerator.Key 및 IDictionaryEnumerator.Value 속성을 사용하여 .resources 파일에서 각 리소스의 이름과 값을 검색하는 방법을 보여 줍니다. 예제를 실행하려면 문자열 리소스를 정의하기 위해 ApplicationResources.txt이라는 다음 텍스트 파일을 만듭니다.

```
Title="Contact Information"
Label1="First Name:"
Label2="Middle Name:"
Label3="Last Name:"
Label4="SSN:"
Label5="Street Address:"
Label6="City:"
Label7="State:"
Label8="Zip Code:"
Label9="Home Phone:"
Label10="Business Phone:"
Label11="Mobile Phone:"
Label12="Other Phone:"
Label13="Fax:"
```

```
Label14="Email Address:"
Label15="Alternate Email Address:"
```

그런 다음, 다음 명령을 사용하여 텍스트 리소스 파일을 ApplicationResources.resources 라는 이진 파일로 변환할 수 있습니다.

```
resgen ApplicationResources.txt
```

다음 예제에서는 ResourceReader 클래스를 사용하여 독립 실행형 이진 .resources 파일의 각 리소스를 열거하고 해당 키 이름과 해당 값을 표시합니다.

C#

```
using System;
using System.Collections;
using System.Resources;

public class Example1
{
   public static void Run()
   {
      Console.WriteLine("Resources in ApplicationResources.resources:");
      ResourceReader res = new
ResourceReader(@".\ApplicationResources.resources");
      IDictionaryEnumerator dict = res.GetEnumerator();
      while (dict.MoveNext())
         Console.WriteLine($"   {dict.Key}: '{dict.Value}' (Type
{dict.Value.GetType().Name})");
      res.Close();
   }
}
// The example displays the following output:
//       Resources in ApplicationResources.resources:
//          Label3: '"Last Name:"' (Type String)
//          Label2: '"Middle Name:"' (Type String)
//          Label1: '"First Name:"' (Type String)
//          Label7: '"State:"' (Type String)
//          Label6: '"City:"' (Type String)
//          Label5: '"Street Address:"' (Type String)
//          Label4: '"SSN:"' (Type String)
//          Label9: '"Home Phone:"' (Type String)
//          Label8: '"Zip Code:"' (Type String)
//          Title: '"Contact Information"' (Type String)
//          Label12: '"Other Phone:"' (Type String)
//          Label13: '"Fax:"' (Type String)
//          Label10: '"Business Phone:"' (Type String)
//          Label11: '"Mobile Phone:"' (Type String)
//          Label14: '"Email Address:"' (Type String)
//          Label15: '"Alternate Email Address:"' (Type String)
```

IDictionaryEnumerator.Value 속성에서 리소스 데이터를 검색하려고 하면 다음과 같은 예외가 발생할 수 있습니다.

- 데이터가 예상 형식이 아닌 경우 FormatException.
- 데이터가 속한 형식을 포함하는 어셈블리를 찾을 수 없는 경우 FileNotFoundException.
- 데이터가 속한 형식을 찾을 수 없을 경우 TypeLoadException입니다.

일반적으로 .resources 파일이 수동으로 수정되었거나, 형식이 정의된 어셈블리가 애플리케이션에 포함되지 않았거나 실수로 삭제되었거나, 어셈블리가 형식보다 이전 버전인 경우 이러한 예외가 throw됩니다. 이러한 예외 중 하나가 throw되면 다음 섹션과 같이 각 리소스를 열거하고 GetResourceData 메서드를 호출하여 리소스를 검색할 수 있습니다. 이 방법은 IDictionaryEnumerator.Value 속성이 반환하려고 시도한 데이터 형식에 대한 몇 가지 정보를 제공합니다.

GetResourceData를 사용하여 이름으로 리소스 검색

.resources 파일에서 리소스를 열거하는 두 번째 방법은 `IDictionaryEnumerator.MoveNext` 메서드를 호출하여 파일의 리소스를 탐색하는 작업도 포함합니다. 각 리소스에 대해 IDictionaryEnumerator.Key 속성에서 리소스의 이름을 검색한 다음 GetResourceData(String, String, Byte[]) 메서드에 전달하여 리소스의 데이터를 검색합니다. 이 값은 `resourceData` 인수에서 바이트 배열로 반환됩니다.

이 방법은 리소스 값을 형성하는 실제 바이트를 반환하기 때문에 IDictionaryEnumerator.Key 및 IDictionaryEnumerator.Value 속성에서 리소스 이름과 값을 검색하는 것보다 더 어색합니다. 그러나 리소스를 검색하려고 하면 예외가 throw되는 경우 GetResourceData 메서드는 리소스의 데이터 형식에 대한 정보를 제공하여 예외의 원본을 식별하는 데 도움이 될 수 있습니다. 리소스의 데이터 형식을 나타내는 문자열에 대한 자세한 내용은 GetResourceData참조하세요.

다음 예제에서는 이 방법을 사용하여 리소스를 검색하고 throw되는 예외를 처리하는 방법을 보여 줍니다. 프로그래밍적으로 4개의 문자열, 1개의 부울, 1개의 정수, 1개의 비트맵이 포함된 .resources 이진 파일을 만듭니다. 예제를 실행하려면 다음을 수행합니다.

1. ContactResources.resources라는 .resources 파일을 만드는 다음 소스 코드를 컴파일하고 실행합니다.

```
C#

using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Resources;
using System.Runtime.Versioning;
```

```
public class Example5
{
    [SupportedOSPlatform("windows")]
    public static void Run()
    {
        // Bitmap as stream.
        MemoryStream bitmapStream = new MemoryStream();
        Bitmap bmp = new Bitmap(@".\ContactsIcon.jpg");
        bmp.Save(bitmapStream, ImageFormat.Jpeg);

        // Define resources to be written.
        using (ResourceWriter rw = new
ResourceWriter(@".\ContactResources.resources"))
        {
            rw.AddResource("Title", "Contact List");
            rw.AddResource("NColumns", 5);
            rw.AddResource("Icon", bitmapStream);
            rw.AddResource("Header1", "Name");
            rw.AddResource("Header2", "City");
            rw.AddResource("Header3", "State");
            rw.AddResource("ClientVersion", true);
            rw.Generate();
        }
    }
}
```

소스 코드 파일의 이름은 CreateResources.cs. 다음 명령을 사용하여 C#에서 컴파일 할 수 있습니다.

```
csc CreateResources.cs /r:library.dll
```

2. 다음 코드를 컴파일하고 실행하여 ContactResources.resources 파일의 리소스를 열거합니다.

C#

```
using System;
using System.Collections;
using System.Drawing;
using System.IO;
using System.Resources;
using System.Runtime.Versioning;

public class Example6
{
    [SupportedOSPlatform("windows")]
    public static void Run()
    {
```

```
        ResourceReader rdr = new
ResourceReader(@".\ContactResources.resources");
        IDictionaryEnumerator dict = rdr.GetEnumerator();
        while (dict.MoveNext())
        {
            Console.WriteLine($"Resource Name: {dict.Key}");
            try
            {
                Console.WriteLine($"   Value: {dict.Value}");
            }
            catch (FileNotFoundException)
            {
                Console.WriteLine("   Exception: A file cannot be
found.");
                DisplayResourceInfo(rdr, (string)dict.Key, false);
            }
            catch (FormatException)
            {
                Console.WriteLine("   Exception: Corrupted data.");
                DisplayResourceInfo(rdr, (string)dict.Key, true);
            }
            catch (TypeLoadException)
            {
                Console.WriteLine("   Exception: Cannot load the data
type.");
                DisplayResourceInfo(rdr, (string)dict.Key, false);
            }
        }
    }

    [SupportedOSPlatform("windows")]
    private static void DisplayResourceInfo(ResourceReader rr,
                                    string key, bool loaded)
    {
        string dataType = null;
        byte[] data = null;
        rr.GetResourceData(key, out dataType, out data);

        // Display the data type.
        Console.WriteLine($"   Data Type: {dataType}");
        // Display the bytes that form the available data.
        Console.Write("   Data: ");
        int lines = 0;
        foreach (var dataItem in data)
        {
            lines++;
            Console.Write("{0:X2} ", dataItem);
            if (lines % 25 == 0)
                Console.Write("\n         ");
        }
        Console.WriteLine();
        // Try to recreate current state of data.
        // Do: Bitmap, DateTimeTZI
        switch (dataType)
        {
```

```
            // Handle internally serialized string data
(ResourceTypeCode members).
            case "ResourceTypeCode.String":
                BinaryReader reader = new BinaryReader(new
MemoryStream(data));
                string binData = reader.ReadString();
                Console.WriteLine($"   Recreated Value: {binData}");
                break;
            case "ResourceTypeCode.Int32":
                Console.WriteLine($"   Recreated Value:
{BitConverter.ToInt32(data, 0)}");
                break;
            case "ResourceTypeCode.Boolean":
                Console.WriteLine($"   Recreated Value:
{BitConverter.ToBoolean(data, 0)}");
                break;
            // .jpeg image stored as a stream.
            case "ResourceTypeCode.Stream":
                const int OFFSET = 4;
                int size = BitConverter.ToInt32(data, 0);
                Bitmap value1 = new Bitmap(new MemoryStream(data,
OFFSET, size));
                Console.WriteLine($"   Recreated Value: {value1}");
                break;
            default:
                break;
        }
        Console.WriteLine();
    }
}
```

소스 코드를 수정한 후(예: `try` 블록 끝에 FormatException 의도적으로 throw) 예제를 실행하여 GetResourceData 호출을 통해 일부 리소스 정보를 검색하거나 다시 만드는 방법을 확인할 수 있습니다.

System.Resources.SatelliteContractVersionAttribute 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

데스크톱 앱에서 SatelliteContractVersionAttribute 속성은 주 어셈블리와 모든 위성 어셈블리 간에 계약을 설정합니다. 이 특성을 주 어셈블리에 적용하고 이 버전의 주 어셈블리에서 작동하는 위성 어셈블리의 버전 번호를 전달합니다. 리소스 관리자(ResourceManager 개체)가 리소스를 조회하면 주 어셈블리에 이 특성으로 지정된 위성 버전을 명시적으로 로드합니다.

주 어셈블리를 업데이트하면 어셈블리 버전 번호가 증가합니다. 그러나 기존 어셈블리가 앱과 호환되는 경우 위성 어셈블리의 새 복사본을 제공하지 않을 수 있습니다. 이 경우 주 어셈블리의 버전 번호를 증가하지만 위성 계약 버전 번호는 동일하게 유지합니다. 리소스 관리자는 기존 위성 어셈블리를 사용합니다.

주 어셈블리가 아닌 위성 어셈블리를 수정하려면 위성의 버전 번호를 증분합니다. 이 경우, 새 위성 어셈블리가 이전 위성 어셈블리와의 이전 버전 호환성을 가지고 있음을 설명하는 게시자 정책 어셈블리를 위성 어셈블리와 함께 발송하십시오. 리소스 관리자는 특성에 SatelliteContractVersionAttribute 따라 주 어셈블리에 기록된 이전 계약 번호를 계속 사용합니다. 그러나 로더는 정책 어셈블리에 지정된 위성 어셈블리 버전에 바인딩됩니다.

공유 구성 요소의 공급업체는 게시자 정책 어셈블리를 사용하여 릴리스된 어셈블리의 특정 버전에 대한 호환성 문을 만듭니다. 게시자 정책 어셈블리는 이름이 형식 `policy.<major>.<minor>.<ComponentAssemblyName>`인 강력한 이름의 어셈블리이며 GAC(전역 어셈블리 캐시)에 등록됩니다. 게시자 정책은 < 도구를 사용하여 XML 구성 파일(bindingRedirect> 요소 참조)에서 생성됩니다. 어셈블리 링커는 `/link` 옵션을 사용하여 XML 구성 파일을 매니페스트 어셈블리에 연결한 후 전역 어셈블리 캐시에 저장하는 데 사용됩니다. 공급업체에서 버그 수정이 포함된 유지 관리 릴리스(서비스 팩)를 제공하는 경우 게시자 정책 어셈블리를 사용할 수 있습니다.

Windows 8.x 스토어 앱

PRI(패키지 리소스 인덱스) 파일에 버전 관리 의미 체계가 없으므로 Windows 8.x 스토어 앱에서는 이 특성이 무시됩니다. 또한 Windows 8.x 스토어 패키징 모델을 사용하려면 위성 어셈블리 또는 PRI 파일을 다시 배포할 가능성 없이 모든 리소스를 동일한 패키지로 배송해야 합니다.

.NET의 작업자 서비스

2025. 05. 29.

다음과 같은 장기 실행 서비스를 만드는 데는 여러 가지 이유가 있습니다.

- CPU 집약적 데이터 처리.
- 백그라운드에서 작업 항목을 대기열에 추가합니다.
- 일정에 따라 시간 기반 작업을 수행합니다.

백그라운드 서비스 처리에는 일반적으로 UI(사용자 인터페이스)가 포함되지 않지만 UI를 기반으로 빌드할 수 있습니다. .NET Framework를 사용하는 초기에 Windows 개발자는 이러한 용도로 Windows 서비스를 만들 수 있습니다. 이제 .NET을 사용하여 BackgroundService의 구현체인 IHostedService을 사용할 수 있으며, 직접 구현할 수도 있습니다.

.NET을 사용하면 더 이상 Windows로 제한되지 않습니다. 플랫폼 간 백그라운드 서비스를 개발할 수 있습니다. 호스팅된 서비스는 로깅, 구성 및 DI(종속성 주입)를 준비합니다. 라이브러리의 확장 제품군의 일부이므로 일반 호스트사용하는 모든 .NET 워크로드의 기본입니다.

> ⓘ **중요**
>
> .NET SDK를 설치하면 `Microsoft.NET.Sdk.Worker` 및 작업자 템플릿도 설치됩니다. 즉, .NET SDK를 설치한 후 **dotnet new worker** 명령을 사용하여 새 작업자를 만들 수 있습니다. Visual Studio를 사용하는 경우 선택적 ASP.NET 및 웹 개발 워크로드가 설치될 때까지 템플릿이 숨겨집니다.

용어

많은 용어가 동의어로 잘못 사용됩니다. 이 섹션에서는 이 문서의 의도를 보다 명확하게 하기 위해 이러한 용어 중 일부를 정의합니다.

- **백그라운드 서비스**: BackgroundService 형식입니다.
- **호스티드 서비스**: IHostedService구현 또는 IHostedService 자체.
- **장기 실행 서비스**: 지속적으로 실행되는 모든 서비스.
- **Windows 서비스**: *Windows 서비스* 인프라는 원래 .NET Framework 중심이지만 이제는 .NET을 통해 액세스할 수 있습니다.
- **작업자 서비스**: *작업자 서비스* 템플릿입니다.

작업자 서비스 템플릿

작업자 서비스 템플릿은 .NET CLI 및 Visual Studio에서 사용할 수 있습니다. 자세한 내용은 .NET CLI, dotnet new worker - 템플릿참조하세요. 템플릿은 `Program` 및 `Worker` 클래스로 구성됩니다.

C#

```
using App.WorkerService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();

IHost host = builder.Build();
host.Run();
```

이전 `Program` 클래스:

- HostApplicationBuilder만듭니다.
- AddHostedService 호출하여 `Worker` 호스트된 서비스로 등록합니다.
- 빌더에서 IHost을(를) 생성합니다.
- 앱을 실행하는 `Run` 인스턴스에서 `host` 호출합니다.

템플릿 기본값

작업자 템플릿은 기본적으로 GC(서버 가비지 수집)를 사용하도록 설정하지 않습니다. 그 필요성을 결정하는 데 중요한 역할을 하는 여러 요소가 있기 때문에. 장기 실행 서비스가 필요한 모든 시나리오는 이 기본값의 성능 영향을 고려해야 합니다. 서버 GC를 사용하도록 설정하려면 프로젝트 파일에 `ServerGarbageCollection` 노드를 추가합니다.

XML

```
<PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

절충 및 고려 사항

테이블 확장

활성화됨	비활성화
효율적인 메모리 관리: 메모리 누수 방지 및 리소스 사용 최적화를 위해 사용되지 않는 메모리를 자동으로 회수합니다.	실시간 성능 향상: 대기 시간에 민감한 애플리케이션에서 가비지 수집으로 인한 잠재적 일시 중지 또는 중단을 방지합니다.
장기 안정성: 오랜 기간 동안 메모리를 관리하여 장기 실행 서비스에서 안정적인 성능을 유지하는 데	리소스 효율성: 리소스가 제한된 환경에서 CPU 및 메모리 리소스를 절약할 수 있습니다.

활성화됨	비활성화
도움이 됩니다.	
유지 관리 감소: 수동 메모리 관리의 필요성을 최소화하여 유지 관리를 간소화합니다.	수동 메모리 제어: 특수 애플리케이션에 대한 메모리에 대한 세분화된 제어를 제공합니다.
예측 가능한 동작: 일관되고 예측 가능한 애플리케이션 동작에 기여합니다.	단기 프로세스에 적합: 단기 또는 임시 프로세스에 대한 가비지 수집 오버헤드를 최소화합니다.

성능 고려 사항에 대한 자세한 내용은 Server GC참조하세요. 서버 GC를 구성하는 방법에 대한 자세한 내용은 Server GC 구성 예제를 참조하세요.

작업자 클래스

`Worker` 템플릿은 간단한 구현을 제공합니다.

C#

```
namespace App.WorkerService;

public sealed class Worker(ILogger<Worker> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            logger.LogInformation("Worker running at: {time}",
DateTimeOffset.Now);
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

앞의 `Worker` 클래스는 BackgroundService구현하는 IHostedService하위 클래스입니다. BackgroundService는 `abstract class`이며 하위 클래스가 BackgroundService.ExecuteAsync(CancellationToken)를 구현해야 합니다. 템플릿 구현에서 `ExecuteAsync` 초당 한 번 반복되어 프로세스가 취소하라는 신호를 보낼 때까지 현재 날짜와 시간을 기록합니다.

프로젝트 파일

작업자 템플릿은 다음 프로젝트 파일 `Sdk`사용합니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

자세한 내용은 .NET 프로젝트 SDK 참조하세요.

NuGet 패키지

작업자 템플릿을 기반으로 하는 앱은 `Microsoft.NET.Sdk.Worker` SDK를 사용하며 Microsoft.Extensions.Hosting 패키지에 대한 명시적 패키지 참조가 있습니다.

컨테이너 및 클라우드 적응성

대부분의 최신 .NET 워크로드에서는 컨테이너가 실행 가능한 옵션입니다. Visual Studio의 작업자 템플릿에서 장기 실행 서비스를 만들 때 **Docker 지원**에 대해 옵트인할 수 있습니다. 이렇게 하면 .NET 앱을 컨테이너화하는 *Dockerfile* 만들어집니다. *Dockerfile* 이미지를 빌드하기 위한 지침 집합입니다. .NET 앱의 경우 *Dockerfile* 일반적으로 솔루션 파일 옆에 있는 디렉터리의 루트에 있습니다.

Dockerfile

```
# See https://aka.ms/containerfastmode to understand how Visual Studio uses this
# Dockerfile to build your images for faster debugging.

FROM
mcr.microsoft.com/dotnet/runtime:8.0@sha256:e6b552fd7a0302e4db30661b16537f7efcdc0b
67790a47dbf67a5e798582d3a5 AS base
WORKDIR /app

FROM
mcr.microsoft.com/dotnet/sdk:8.0@sha256:35792ea4ad1db051981f62b313f1be3b46b1f45cad
baa3c288cd0d3056eefb83 AS build
WORKDIR /src
COPY ["background-service/App.WorkerService.csproj", "background-service/"]
RUN dotnet restore "background-service/App.WorkerService.csproj"
COPY . .
WORKDIR "/src/background-service"
RUN dotnet build "App.WorkerService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "App.WorkerService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "App.WorkerService.dll"]
```

이전 *Dockerfile* 단계는 다음과 같습니다.

- 기본 이미지를 `mcr.microsoft.com/dotnet/runtime:8.0`에서 `base`이라는 별칭으로 설정합니다.
- 작업 디렉터리를 /app 변경합니다.
- `build` 이미지에서 `mcr.microsoft.com/dotnet/sdk:8.0` 별칭을 설정합니다.
- 작업 디렉터리를 /src 변경합니다.
- 콘텐츠를 복사하고 .NET 앱을 게시합니다.
  - 앱은 dotnet publish 명령을 사용하여 게시됩니다.
- `mcr.microsoft.com/dotnet/runtime:8.0` .NET SDK 이미지를 재구성하고 있습니다 (`base` 별칭).
- */publish*에서 게시된 빌드 출력을 복사합니다.
- dotnet App.BackgroundService.dll에 위임하는 진입점을 정의하기.

> **팁**
>
> `mcr.microsoft.com` MCR은 "Microsoft Container Registry"를 의미하며 공식 Docker 허브에서 Microsoft의 신디케이트 컨테이너 카탈로그입니다. **Microsoft 신디케이트 컨테이너 카탈로그** 문서에는 추가 세부 정보가 포함되어 있습니다.

Docker를 .NET 작업자 서비스에 대한 배포 전략으로 대상으로 지정하는 경우 프로젝트 파일에 몇 가지 고려 사항이 있습니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>App.WorkerService</RootNamespace>
    <DockerDefaultTargetOS>Linux</DockerDefaultTargetOS>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.4" />
    <PackageReference
Include="Microsoft.VisualStudio.Azure.Containers.Tools.Targets" Version="1.21.0"
/>
  </ItemGroup>
</Project>
```

이전 프로젝트 파일에서 `<DockerDefaultTargetOS>` 요소는 `Linux` 대상으로 지정합니다. Windows 컨테이너를 대상으로 지정하려면 대신 `Windows` 사용합니다.

`Microsoft.VisualStudio.Azure.Containers.Tools.Targets` 템플릿에서 선택하면 **NuGet 패키지** 패키지 참조로 자동으로 추가됩니다.

.NET을 사용하는 Docker에 대한 자세한 내용은 자습서: .NET 앱컨테이너화를 참조하세요. Azure에 배포하는 방법에 대한 자세한 내용은 자습서: Azure작업자 서비스 배포를 참조하세요.

> **중요**
>
> 작업자 템플릿을 사용하여 *사용자 비밀* 활용하려면 `Microsoft.Extensions.Configuration.UserSecrets` NuGet 패키지를 명시적으로 참조해야 합니다.

호스트된 서비스 확장성

IHostedService 인터페이스는 두 가지 메서드를 정의합니다.

- IHostedService.StartAsync(CancellationToken)
- IHostedService.StopAsync(CancellationToken)

이러한 두 메서드는 *수명 주기* 메서드로 사용되며 호스트 시작 및 중지 이벤트 중에 각각 호출됩니다.

> **참고**
>
> StartAsync 또는 StopAsync 메서드를 재정의하는 경우 `await` 클래스 메서드를 호출하고 `base` 서비스가 제대로 시작 및/또는 종료되는지 확인해야 합니다.

> **중요**
>
> 인터페이스는 AddHostedService<THostedService>(IServiceCollection) 확장 메서드에서 제네릭 형식 매개 변수 제약 조건으로 사용되므로 구현만 허용됩니다. 제공된 BackgroundService 서브클래스와 함께 사용하거나 완전히 구현할 수 있습니다.

신호 완성

대부분의 일반적인 시나리오에서는 호스트된 서비스의 완료를 명시적으로 알릴 필요가 없습니다. 호스트가 서비스를 시작하면 호스트가 중지될 때까지 실행되도록 설계되었습니다. 그러나 일부 시나리오에서는 서비스가 완료될 때 전체 호스트 애플리케이션의 완료를 신호로 표시해야 할 수 있습니다. 완료를 알리려면 다음 `Worker` 클래스를 고려합니다.

C#

```
namespace App.SignalCompletionService;

public sealed class Worker(
    IHostApplicationLifetime hostApplicationLifetime,
    ILogger<Worker> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        // TODO: implement single execution logic here.
        logger.LogInformation(
            "Worker running at: {Time}", DateTimeOffset.Now);

        await Task.Delay(1_000, stoppingToken);

        // When completed, the entire app host will stop.
        hostApplicationLifetime.StopApplication();
    }
}
```

앞의 코드에서 BackgroundService.ExecuteAsync(CancellationToken) 메서드는 반복되지 않으며 완료되면 IHostApplicationLifetime.StopApplication()호출합니다.

> ⓘ **중요**
>
> 그러면 호스트가 중지되어야 한다는 신호가 표시되고, `StopApplication` 대한 이 호출이 없으면 호스트가 무기한으로 실행됩니다. 수명이 짧은 호스티드 서비스를 실행하려는 경우(시나리오를 한 번 실행) 작업자 템플릿을 사용하려는 경우 호스트에 중지 신호를 요청 `StopApplication` 해야 합니다.

자세한 내용은 다음을 참조하세요.

- .NET 제네릭 호스트: IHostApplicationLifetime
- .NET 제네릭 호스트: 호스트 종료
- .NET 제네릭 호스트: 호스팅 종료 프로세스

대체 방법

종속성 주입, 로깅 및 구성이 필요한 수명이 짧은 앱의 경우 작업자 템플릿 대신 .NET 제네릭 호스트 를 사용합니다. 이렇게 하면 클래스 없이 이러한 기능을 사용할 수 있습니다 `Worker` . 제네릭 호스트를 사용하는 단기 앱의 간단한 예제는 다음과 같이 프로젝트 파일을 정의할 수 있습니다.

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>ShortLived.App</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.5" />
  </ItemGroup>
</Project>
```

클래스는 `Program` 다음과 같이 표시될 수 있습니다.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var builder = Host.CreateApplicationBuilder(args);
builder.Services.AddSingleton<JobRunner>();

using var host = builder.Build();

try
{
    var runner = host.Services.GetRequiredService<JobRunner>();

    await runner.RunAsync();

    return 0; // success
}
catch (Exception ex)
{
    var logger = host.Services.GetRequiredService<ILogger<Program>>();

    logger.LogError(ex, "Unhandled exception occurred during job execution.");

    return 1; // failure
}
```

앞의 코드는 작업을 실행할 논리를 포함하는 사용자 지정 클래스인 서비스를 만듭니다 `JobRunner` . 메서드가 `RunAsync` 에서 `JobRunner` 에 호출되고 성공적으로 완료되면 앱이 `0` 를 반환합니다. 처리되지 않은 예외가 발생하면 오류를 기록하고 반환 `1` 합니다.

이 간단한 시나리오에서 클래스는 `JobRunner` 다음과 같이 표시할 수 있습니다.

C#

```
using Microsoft.Extensions.Logging;

internal sealed class JobRunner(ILogger<JobRunner> logger)
{
    public async Task RunAsync()
    {
        logger.LogInformation("Starting job...");

        // Simulate work
        await Task.Delay(1000);

        // Simulate failure
        // throw new InvalidOperationException("Something went wrong!");

        logger.LogInformation("Job completed successfully.");
    }
}
```

메서드에 `RunAsync` 실제 논리를 추가해야 하지만, 이 예제는 `Worker` 클래스가 없이도, 그리고 호스트 완료를 명시적으로 알리지 않고도 수명이 짧은 앱에 제네릭 호스트를 사용하는 방법을 보여 줍니다.

또한 참고하십시오

- BackgroundService 하위 클래스 자습서:
  - .NET에서 큐 서비스 생성하기
  - BackgroundService 내에서 .NET 범위가 지정된 서비스 사용
  - `BackgroundService` 사용하여 Windows 서비스 만들기
- 사용자 지정 IHostedService 구현:
  - `IHostedService` 인터페이스 구현

큐 서비스 만들기

큐 서비스는 이전 작업 항목이 완료될 때 작업 항목을 큐에 대기하고 순차적으로 작업할 수 있는 장기 실행 서비스의 좋은 예입니다. 작업자 서비스 템플릿을 기반으로 하여 BackgroundService에 새로운 기능을 추가합니다.

이 튜토리얼에서는 다음을 배우게 됩니다:

- ✔ 큐 서비스를 만듭니다.
- ✔ 작업 큐에 작업을 위임합니다.
- ✔ IHostApplicationLifetime 이벤트에 대한 콘솔 키 수신기를 등록합니다.

> **팁**
>
> ".NET의 작업자" 예제 소스 코드는 모두 **샘플 브라우저** 에서 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET 작업자**를 참조하세요.

필수 조건

- .NET 8.0 SDK 버전 이상
- .NET IDE(통합 개발 환경)
  - Visual Studio 자유롭게 사용하세요.

새 프로젝트 만들기

Visual Studio를 사용하여 새 작업자 서비스 프로젝트를 만들려면 **파일** > **새** > **Project...** 선택합니다. **새 프로젝트 만들기** 대화 상자에서 "작업자 서비스"를 검색하고 작업자 서비스 템플릿을 선택합니다. .NET CLI를 사용하려는 경우 작업 디렉터리에서 즐겨 찾는 터미널을 엽니다. `dotnet new` 명령을 실행하고 `<Project.Name>`을(를) 원하는 프로젝트 이름으로 바꿉니다.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

.NET CLI 새 작업자 서비스 프로젝트 명령에 대한 자세한 내용은 dotnet new Worker를 참조하세요.

> **팁**

Visual Studio Code를 사용하는 경우 통합 터미널에서 .NET CLI 명령을 실행할 수 있습니다. 자세한 내용은 Visual Studio Code: 통합 터미널을 참조하세요.

대기열 서비스 만들기

`System.Web.Hosting` 네임스페이스의 QueueBackgroundWorkItem(Func<CancellationToken,Task>) 기능에 익숙할 수 있습니다.

> **팁**
>
> 네임스페이 `System.Web` 스의 기능은 의도적으로 .NET으로 이식되지 않았으며 .NET Framework 전용으로 유지됩니다. 자세한 내용은 ASP.NET을 ASP.NET Core로 점진적으로 마이그레이션 시작하기를 참조하세요.

.NET에서 `QueueBackgroundWorkItem`에서 영감을 얻은 서비스를 모델링하려는 경우, 먼저 프로젝트에 `IBackgroundTaskQueue` 인터페이스를 추가합니다.

C#

```
namespace App.QueueService;

public interface IBackgroundTaskQueue
{
    ValueTask QueueBackgroundWorkItemAsync(
        Func<CancellationToken, ValueTask> workItem);

    ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken);
}
```

큐 기능을 노출하는 메서드와 이전에 큐에 대기 중인 작업 항목을 큐에서 제거하는 두 가지 방법이 있습니다. *작업 항목*은 `Func<CancellationToken, ValueTask>`입니다. 다음으로, 프로젝트에 기본 구현을 추가합니다.

C#

```
using System.Threading.Channels;

namespace App.QueueService;

public sealed class DefaultBackgroundTaskQueue : IBackgroundTaskQueue
{
    private readonly Channel<Func<CancellationToken, ValueTask>> _queue;

    public DefaultBackgroundTaskQueue(int capacity)
```

```
    {
        BoundedChannelOptions options = new(capacity)
        {
            FullMode = BoundedChannelFullMode.Wait
        };
        _queue = Channel.CreateBounded<Func<CancellationToken, ValueTask>>(options);
    }

    public async ValueTask QueueBackgroundWorkItemAsync(
        Func<CancellationToken, ValueTask> workItem)
    {
        ArgumentNullException.ThrowIfNull(workItem);

        await _queue.Writer.WriteAsync(workItem);
    }

    public async ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        Func<CancellationToken, ValueTask>? workItem =
            await _queue.Reader.ReadAsync(cancellationToken);

        return workItem;
    }
}
```

이전 구현은 큐로 Channel<T> 사용합니다. BoundedChannelOptions(Int32) 명시적 용량으로 호출됩니다. 용량은 예상된 애플리케이션 로드 및 큐에 액세스하는 동시 스레드 수에 따라 설정되어야 합니다. BoundedChannelFullMode.Wait는 ChannelWriter<T>.WriteAsync 호출 시 공간이 확보될 때까지 완료되지 않는 작업을 반환합니다. 너무 많은 게시자/호출이 누적되는 경우 백프레셔로 이어집니다.

작업자 클래스 다시 쓰기

다음 `QueueHostedService` 예제에서

- `ProcessTaskQueueAsync` 메서드는 Task의 `ExecuteAsync`을 반환합니다.
- `ProcessTaskQueueAsync`에서 큐의 백그라운드 작업이 큐에서 제거되고 실행됩니다.
- 작업 항목은 `StopAsync`에서 서비스가 중지되기 전에 대기 상태입니다.

기존 `Worker` 클래스를 다음 C# 코드로 바꾸고 파일 이름을 *QueueHostedService.cs*.

C#

```
namespace App.QueueService;

public sealed class QueuedHostedService(
        IBackgroundTaskQueue taskQueue,
        ILogger<QueuedHostedService> logger) : BackgroundService
```

```
{
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation("""
            {Name} is running.
            Tap W to add a work item to the
            background queue.
            """,
            nameof(QueuedHostedService));

        return ProcessTaskQueueAsync(stoppingToken);
    }

    private async Task ProcessTaskQueueAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                Func<CancellationToken, ValueTask>? workItem =
                    await taskQueue.DequeueAsync(stoppingToken);

                await workItem(stoppingToken);
            }
            catch (OperationCanceledException)
            {
                // Prevent throwing if stoppingToken was signaled
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Error occurred executing task work item.");
            }
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            $"{nameof(QueuedHostedService)} is stopping.");

        await base.StopAsync(stoppingToken);
    }
}
```

입력 장치에서 `MonitorLoop` 키가 선택될 때마다 `w` 서비스가 호스팅된 서비스에 대해 큐에 넣는 작업을 처리합니다.

- `IBackgroundTaskQueue`가 `MonitorLoop` 서비스에 삽입됩니다.
- `IBackgroundTaskQueue.QueueBackgroundWorkItemAsync`이 호출되어 작업 항목을 큐에 넣습니다.
- 작업 항목은 장기 실행 백그라운드 작업을 시뮬레이션합니다.
  - 3개의 5초 지연이 실행됩니다 Delay.

- 작업이 취소되면 `try-catch` 문이 OperationCanceledException를 가로챕니다.

C#

```
namespace App.QueueService;

public sealed class MonitorLoop(
    IBackgroundTaskQueue taskQueue,
    ILogger<MonitorLoop> logger,
    IHostApplicationLifetime applicationLifetime)
{
    private readonly CancellationToken _cancellationToken =
applicationLifetime.ApplicationStopping;

    public void StartMonitorLoop()
    {
        logger.LogInformation($"{nameof(MonitorAsync)} loop is starting.");

        // Run a console user input loop in a background thread
        Task.Run(async () => await MonitorAsync());
    }

    private async ValueTask MonitorAsync()
    {
        while (!_cancellationToken.IsCancellationRequested)
        {
            var keyStroke = Console.ReadKey();
            if (keyStroke.Key == ConsoleKey.W)
            {
                // Enqueue a background work item
                await taskQueue.QueueBackgroundWorkItemAsync(BuildWorkItemAsync);
            }
        }
    }

    private async ValueTask BuildWorkItemAsync(CancellationToken token)
    {
        // Simulate three 5-second tasks to complete
        // for each enqueued work item

        int delayLoop = 0;
        var guid = Guid.NewGuid();

        logger.LogInformation("Queued work item {Guid} is starting.", guid);

        while (!token.IsCancellationRequested && delayLoop < 3)
        {
            try
            {
                await Task.Delay(TimeSpan.FromSeconds(5), token);
            }
            catch (OperationCanceledException)
            {
                // Prevent throwing if the Delay is cancelled
```

```
            }

            ++ delayLoop;

            logger.LogInformation("Queued work item {Guid} is running.
{DelayLoop}/3", guid, delayLoop);
        }

        if (delayLoop is 3)
        {
            logger.LogInformation("Queued Background Task {Guid} is complete.",
guid);
        }
        else
        {
            logger.LogInformation("Queued Background Task {Guid} was cancelled.",
guid);
        }
    }
}
```

기존 `Program` 콘텐츠를 다음 C# 코드로 바꿉니다.

C#

```
using App.QueueService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddSingleton<MonitorLoop>();
builder.Services.AddHostedService<QueuedHostedService>();
builder.Services.AddSingleton<IBackgroundTaskQueue>(_ =>
{
    if (!int.TryParse(builder.Configuration["QueueCapacity"], out var
queueCapacity))
    {
        queueCapacity = 100;
    }

    return new DefaultBackgroundTaskQueue(queueCapacity);
});

IHost host = builder.Build();

MonitorLoop monitorLoop = host.Services.GetRequiredService<MonitorLoop>()!;
monitorLoop.StartMonitorLoop();

host.Run();
```

서비스는 (*Program.cs*)에 등록됩니다. 호스팅된 서비스는 `AddHostedService` 확장 메서드를 사용하여 등록됩니다. `MonitorLoop` 는 *Program.cs* 최상위 문에서 시작됩니다.

C#

```
MonitorLoop monitorLoop = host.Services.GetRequiredService<MonitorLoop>()!;
monitorLoop.StartMonitorLoop();
```

서비스 등록에 대한 자세한 내용은 .NET 종속성 주입을 참조하세요.

서비스 기능 확인

Visual Studio에서 애플리케이션을 실행하려면 `F5` 를 선택하거나 **디버그>시작 디버깅** 메뉴 옵션을 선택합니다. .NET CLI를 사용하는 경우 작업 디렉터리에서 명령을 실행 `dotnet run` 합니다.

.NET CLI

```
dotnet run
```

.NET CLI 실행 명령에 대한 자세한 내용은 dotnet run을 참조하세요.

메시지가 표시되면 예제 출력에 나와 있는 것처럼 에뮬레이트된 작업 항목을 큐에 넣으려면 `W` (또는 `W`)을 한 번 이상 입력하세요.

Output

```
info: App.QueueService.MonitorLoop[0]
      MonitorAsync loop is starting.
info: App.QueueService.QueuedHostedService[0]
      QueuedHostedService is running.

      Tap W to add a work item to the background queue.

info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\queue-service
winfo: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is starting.
info: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 1/3
info: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 2/3
info: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 3/3
info: App.QueueService.MonitorLoop[0]
      Queued Background Task 8453f845-ea4a-4bcb-b26e-c76c0d89303e is complete.
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
```

```
info: App.QueueService.QueuedHostedService[0]
      QueuedHostedService is stopping.
```

Visual Studio 내에서 애플리케이션을 실행하는 경우 **디버그**>**디버깅 중지...를** 선택합니다. 또는 콘솔 창에서 `Ctrl` + `C` 를 선택하여 취소 신호를 표시합니다.

참고하십시오

- .NET에서의 작업자 서비스
- BackgroundService 내에서 범위가 지정된 서비스 사용
- 를 사용하여 Windows 서비스 만들기 BackgroundService
- IHostedService 인터페이스 구현
- 웹-큐-워커 아키텍처 스타일

Last updated on 2026. 01. 24.

`BackgroundService` 내에서 범위가 지정된 서비스 사용

IHostedService 구현을 AddHostedService 확장 메서드를 사용하여 등록하면 서비스가 싱글톤으로 등록됩니다. 범위가 지정된 서비스에 의존하려는 시나리오가 있을 수 있습니다. 자세한 내용은 서비스 수명을 참조하세요.

이 튜토리얼에서는 다음을 배우게 됩니다:

- ✓ 단일 항목 BackgroundService에서 범위가 지정된 종속성을 올바르게 해결합니다.
- ✓ 범위가 지정된 서비스에 작업을 위임합니다.
- ✓ `override`의 BackgroundService.StopAsync(CancellationToken) 구현

> **팁**
>
> 모든 '.NET 작업자' 예제 소스 코드는 **샘플 브라우저에서** 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET 작업자**를 참조하세요.

필수 조건

- .NET 8.0 SDK 버전 이상
- .NET IDE(통합 개발 환경)(예: Visual Studio)

새 프로젝트 만들기

Visual Studio를 사용하여 새 작업자 서비스 프로젝트를 만들려면 **파일** > **새** > **Project...**선택합니다. **새 프로젝트 만들기** 대화 상자에서 "작업자 서비스"를 검색하고 작업자 서비스 템플릿을 선택합니다. .NET CLI를 사용하려는 경우 작업 디렉터리에서 즐겨 찾는 터미널을 엽니다. `dotnet new` 명령을 실행하고 `<Project.Name>` 원하는 프로젝트 이름으로 바꿉다.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

.NET CLI 새 작업자 서비스 프로젝트 명령에 대한 자세한 내용은 dotnet 새 작업자참조하세요.

> **팁**

Visual Studio Code를 사용하는 경우 통합 터미널에서 .NET CLI 명령을 실행할 수 있습니다. 자세한 내용은 Visual Studio Code: 통합 터미널 참조하세요.

범위가 지정된 서비스 만들기

범위가 지정된 서비스를 `BackgroundService` 내에서 사용하려면, IServiceScopeFactory.CreateScope() API를 통해 범위를 만드십시오. 기본적으로 호스팅되는 서비스에 대한 범위는 생성되지 않습니다. 범위가 지정된 백그라운드 서비스에는 백그라운드 작업의 논리가 포함됩니다.

C#

```
namespace App.ScopedService;

public interface IScopedProcessingService
{
    Task DoWorkAsync(CancellationToken stoppingToken);
}
```

앞의 인터페이스는 단일 `DoWorkAsync` 메서드를 정의합니다. *DefaultScopedProcessingService.cs* 라는 새 클래스에서 구현을 만듭니다.

C#

```
namespace App.ScopedService;

public sealed class DefaultScopedProcessingService(
    ILogger<DefaultScopedProcessingService> logger) : IScopedProcessingService
{
    private readonly string _instanceId = Guid.NewGuid().ToString();

    public Task DoWorkAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{ServiceName} doing work, instance ID: {Id}",
            nameof(DefaultScopedProcessingService),
            _instanceId);

        return Task.CompletedTask;
    }
}
```

- 기본 ILogger 생성자를 사용하여 서비스에 삽입됩니다.
- 메서드는 `DoWorkAsync` 를 반환하고, `Task` 를 받아들여 CancellationToken를 포함합니다.
  - 메서드는 인스턴스 식별자를 `_instanceId` 기록합니다. 클래스가 인스턴스화될 때마다 할당됩니다.

작업자 클래스 다시 쓰기

기존 `Worker` 클래스를 다음 C# 코드로 바꾸고 파일 이름을 *ScopedBackgroundService.cs*.

C#

```
namespace App.ScopedService;

public sealed class ScopedBackgroundService(
    IServiceScopeFactory serviceScopeFactory,
    ILogger<ScopedBackgroundService> logger) : BackgroundService
{
    private const string ClassName = nameof(ScopedBackgroundService);

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Name} is running.", ClassName);

        while (!stoppingToken.IsCancellationRequested)
        {
            using IServiceScope scope = serviceScopeFactory.CreateScope();

            IScopedProcessingService scopedProcessingService =
                scope.ServiceProvider.GetRequiredService<IScopedProcessingService>
();

            await scopedProcessingService.DoWorkAsync(stoppingToken);

            await Task.Delay(10_000, stoppingToken);
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Name} is stopping.", ClassName);

        await base.StopAsync(stoppingToken);
    }
}
```

위의 코드에서 `stoppingToken`가 취소되지 않는 동안, `IServiceScopeFactory`는 범위를 만드는 데 사용됩니다. `IServiceScope`에서는 `IScopedProcessingService`이 해결됩니다. `DoWorkAsync` 메서드가 대기되고 `stoppingToken` 메서드에 전달됩니다. 마지막으로 실행이 10초 동안 지연되고 루프가 계속됩니다. 메서드가 `DoWorkAsync` 호출될 때마다 새 인스턴스가 `DefaultScopedProcessingService` 만들어지고 인스턴스 식별자가 기록됩니다.

템플릿 *Program.cs* 파일 내용을 다음 C# 코드로 바꿉니다.

C#

```
using App.ScopedService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<ScopedBackgroundService>();
builder.Services.AddScoped<IScopedProcessingService, DefaultScopedProcessingService>
();

IHost host = builder.Build();
host.Run();
```

서비스는 (*Program.cs*)에 등록됩니다. 호스트된 서비스는 확장 메서드에 AddHostedService 등록됩니다.

서비스 등록에 대한 자세한 내용은 .NET 종속성 주입을 참조하세요.

서비스 기능 확인

Visual Studio에서 애플리케이션을 실행하려면 `F5` 를 선택하거나 **디버그>시작 디버깅** 메뉴 옵션을 선택합니다. .NET CLI를 사용하는 경우 작업 디렉터리에서 명령을 실행 `dotnet run` 합니다.

.NET CLI

```
dotnet run
```

.NET CLI 실행 명령에 대한 자세한 내용은 dotnet run을 참조하세요.

애플리케이션을 잠시 실행하여 여러 호출을 생성하고, 그 결과 `DoWorkAsync`에 새 인스턴스 식별자를 로깅합니다. 다음 로그와 유사한 출력이 표시됩니다.

Output

```
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is running.
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService doing work, instance ID: 8986a86f-b444-4139-
b9ea-587daae4a6dd
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\scoped-service
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService doing work, instance ID: 07a4a760-8e5a-4c0a-
9e73-fcb2f93157d3
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService doing work, instance ID: c847f432-acca-47ee-
```

```
8720-1030859ce354
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is stopping.
```

Visual Studio 내에서 애플리케이션을 실행하는 경우 **디버그**>**디버깅 중지...를** 선택합니다. 또는 콘솔 창에서 `Ctrl` + `C` 를 선택하여 취소 신호를 표시합니다.

참고하십시오

- .NET에서의 작업자 서비스
- 큐 서비스 만들기
- 를 사용하여 Windows 서비스 만들기 BackgroundService
- IHostedService 인터페이스 구현

Last updated on 2026. 01. 24.

`BackgroundService` 사용하여 Windows 서비스 만들기

.NET Framework 개발자는 Windows 서비스 앱에 익숙할 것입니다. .NET Core 및 .NET 5 이상 이전에는 .NET Framework에 의존했던 개발자가 백그라운드 작업을 수행하거나 장기 실행 프로세스를 실행하는 Windows 서비스를 만들 수 있습니다. 이 기능은 계속 사용할 수 있으며 Windows 서비스로 실행되는 작업자 서비스를 만들 수 있습니다.

이 자습서에서는 다음 방법을 알아봅니다.

- ✔ .NET 작업자 앱을 단일 파일 실행 파일로 게시합니다.
- ✔ Windows 서비스를 만듭니다.
- ✔ `BackgroundService` 앱을 Windows 서비스로 만듭니다.
- ✔ Windows 서비스를 시작하고 중지합니다.
- ✔ 이벤트 로그를 봅니다.
- ✔ Windows 서비스를 삭제합니다.

> **팁**
>
> 모든 '.NET 작업자' 예제 소스 코드는 **샘플 브라우저에서** 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET 작업자**를 참조하세요.

> **중요**
>
> .NET SDK를 설치하면 `Microsoft.NET.Sdk.Worker` 및 작업자 템플릿도 설치됩니다. 즉, .NET SDK를 설치한 후 dotnet new worker 명령을 사용하여 새 작업자를 만들 수 있습니다. Visual Studio를 사용하는 경우 선택적 ASP.NET 및 웹 개발 워크로드가 설치될 때까지 템플릿이 숨겨집니다.

필수 구성 요소

- .NET 8.0 SDK 버전 이상
- 윈도우 운영 체제
- .NET IDE(통합 개발 환경)
  - Visual Studio 자유롭게 사용하세요.

새 프로젝트 만들기

Visual Studio를 사용하여 새 작업자 서비스 프로젝트를 만들려면 **파일**>**새**>**Project...**선택합니다. **새 프로젝트 만들기** 대화 상자에서 "작업자 서비스"를 검색하고 작업자 서비스 템플릿을 선택합니다. .NET CLI를 사용하려는 경우 작업 디렉터리에서 즐겨 찾는 터미널을 엽니다. `dotnet new` 명령을 실행하고 `<Project.Name>` 원하는 프로젝트 이름으로 바꿉다.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

.NET CLI 새 작업자 서비스 프로젝트 명령에 대한 자세한 내용은 dotnet 새 작업자참조하세요.

> **팁**
>
> Visual Studio Code를 사용하는 경우 통합 터미널에서 .NET CLI 명령을 실행할 수 있습니다. 자세한 내용은 **Visual Studio Code: 통합 터미널** 참조하세요.

NuGet 패키지 설치

.NET IHostedService 구현에서 네이티브 Windows 서비스와 상호 작용하려면 Microsoft.Extensions.Hosting.WindowsServices NuGet 패키지 설치해야 합니다.

Visual Studio에서 설치하려면 **NuGet 패키지 관리** 대화 상자를 사용합니다. "Microsoft.Extensions.Hosting.WindowsServices"를 검색하고 설치합니다. .NET CLI를 사용하려는 경우 다음 명령을 실행합니다. (.NET 9 이하의 SDK 버전을 사용하는 경우 대신 양식을 사용 `dotnet add package` 하세요.)

.NET CLI

```
dotnet package add Microsoft.Extensions.Hosting.WindowsServices
```

자세한 내용은 dotnet 패키지 추가를 참조하세요.

패키지를 성공적으로 추가한 후 프로젝트 파일에는 이제 다음 패키지 참조가 포함되어야 합니다.

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.10" />
  <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="9.0.10" />
</ItemGroup>
```

프로젝트 파일 업데이트

이 작업자 프로젝트는 C#의nullable 참조 형식을 사용합니다. 전체 프로젝트에 대해 사용하도록 설정하려면 프로젝트 파일을 적절하게 업데이트합니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>App.WindowsService</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.10" />
    <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="9.0.10" />
  </ItemGroup>
</Project>
```

위의 프로젝트 파일 변경 내용은 `<Nullable>enable<Nullable>` 노드를 추가합니다. 자세한 내용은 nullable 컨텍스트 설정을 참조하세요.

서비스 만들기

*JokeService.cs*프로젝트에 새 클래스를 추가하고 해당 내용을 다음 C# 코드로 바꿉니다.

C#

```csharp
namespace App.WindowsService;

public sealed class JokeService
{
    public string GetJoke()
    {
        Joke joke = _jokes.ElementAt(
            Random.Shared.Next(_jokes.Count));

        return $"{joke.Setup}{Environment.NewLine}{joke.Punchline}";
    }

    // Programming jokes borrowed from:
    // https://github.com/eklavyadev/karljoke/blob/main/source/jokes.json
    private readonly HashSet<Joke> _jokes = new()
    {
        new Joke("What's the best thing about a Boolean?", "Even if you're wrong,
```

```
you're only off by a bit."),
        new Joke("What's the object-oriented way to become wealthy?",
"Inheritance"),
        new Joke("Why did the programmer quit their job?", "Because they didn't get
arrays."),
        new Joke("Why do programmers always mix up Halloween and Christmas?",
"Because Oct 31 == Dec 25"),
        new Joke("How many programmers does it take to change a lightbulb?", "None
that's a hardware problem"),
        new Joke("If you put a million monkeys at a million keyboards, one of them
will eventually write a Java program", "the rest of them will write Perl"),
        new Joke("['hip', 'hip']", "(hip hip array)"),
        new Joke("To understand what recursion is...", "You must first understand
what recursion is"),
        new Joke("There are 10 types of people in this world...", "Those who
understand binary and those who don't"),
        new Joke("Which song would an exception sing?", "Can't catch me - Avicii"),
        new Joke("Why do Java programmers wear glasses?", "Because they don't C#"),
        new Joke("How do you check if a webpage is HTML5?", "Try it out on Internet
Explorer"),
        new Joke("A user interface is like a joke.", "If you have to explain it then
it is not that good."),
        new Joke("I was gonna tell you a joke about UDP...", "...but you might not
get it."),
        new Joke("The punchline often arrives before the set-up.", "Do you know the
problem with UDP jokes?"),
        new Joke("Why do C# and Java developers keep breaking their keyboards?",
"Because they use a strongly typed language."),
        new Joke("Knock-knock.", "A race condition. Who is there?"),
        new Joke("What's the best part about TCP jokes?", "I get to keep telling
them until you get them."),
        new Joke("A programmer puts two glasses on their bedside table before going
to sleep.", "A full one, in case they gets thirsty, and an empty one, in case they
don’t."),
        new Joke("There are 10 kinds of people in this world.", "Those who
understand binary, those who don't, and those who weren't expecting a base 3
joke."),
        new Joke("What did the router say to the doctor?", "It hurts when IP."),
        new Joke("An IPv6 packet is walking out of the house.", "He goes nowhere."),
        new Joke("3 SQL statements walk into a NoSQL bar. Soon, they walk out",
"They couldn't find a table.")
    };
}

readonly record struct Joke(string Setup, string Punchline);
```

위의 농담 서비스 소스 코드는 단일 기능인 `GetJoke` 메서드를 노출합니다. 임의 프로그래밍 농담을 나타내는 `string` 반환 메서드입니다. 클래스 스코프 `_jokes` 필드는 농담 목록을 저장하는 데 사용됩니다. 임의 농담이 목록에서 선택되어 반환됩니다.

`Worker` 클래스 다시 쓰기

템플릿의 기존 `Worker` 다음 C# 코드로 바꾸고 파일 이름을 *WindowsBackgroundService.cs*바꿉니다.

C#

```
namespace App.WindowsService;

public sealed class WindowsBackgroundService(
    JokeService jokeService,
    ILogger<WindowsBackgroundService> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        try
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                string joke = jokeService.GetJoke();
                logger.LogWarning("{Joke}", joke);

                await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
            }
        }
        catch (OperationCanceledException)
        {
            // When the stopping token is canceled, for example, a call made from
services.msc,
            // we shouldn't exit with a non-zero exit code. In other words, this is
expected...
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "{Message}", ex.Message);

            // Terminates this process and returns an exit code to the operating
system.
            // This is required to avoid the 'BackgroundServiceExceptionBehavior',
which
            // performs one of two scenarios:
            // 1. When set to "Ignore": will do nothing at all, errors cause zombie
services.
            // 2. When set to "StopHost": will cleanly stop the host, and log
errors.
            //
            // In order for the Windows Service Management system to leverage
configured
            // recovery options, we need to terminate the process with a non-zero
exit code.
            Environment.Exit(1);
        }
    }
}
```

앞의 코드에서 `JokeService` `ILogger`함께 삽입됩니다. 둘 다 클래스에서 필드로 사용할 수 있습니다. `ExecuteAsync` 방법에서 농담 서비스는 농담을 요청하고 로거에게 씁니다. 이 경우 로거는 Windows 이벤트 로그 Microsoft.Extensions.Logging.EventLog.EventLogLoggerProvider의해 구현됩니다. 로그는 기록되며 **이벤트 뷰어**에서 확인할 수 있습니다.

> ⓘ **참고**
>
> 기본적으로 *이벤트 로그* 심각도는 Warning. 이를 구성할 수 있지만 데모용으로 `WindowsBackgroundService` LogWarning 확장 메서드를 사용하여 로그합니다. `EventLog` 수준을 구체적으로 대상으로 지정하려면 **appsettings에 항목을 추가합니다. {Environment}.json**또는 EventLogSettings.Filter 값을 제공합니다.
>
> JSON
>
> ```
> {
>   "Logging": {
>     "LogLevel": {
>       "Default": "Warning"
>     },
>     "EventLog": {
>       "SourceName": "The Joke Service",
>       "LogName": "Application",
>       "LogLevel": {
>         "Microsoft": "Information",
>         "Microsoft.Hosting.Lifetime": "Information"
>       }
>     }
>   }
> }
> ```
>
> 로그 수준을 구성하는 방법에 대한 자세한 내용은 .NET의 로깅 공급자: Windows EventLog 구성을 참조하세요.

`Program` 클래스 다시 쓰기

템플릿 *Program.cs* 파일 내용을 다음 C# 코드로 바꿉니다.

C#

```
using App.WindowsService;
using Microsoft.Extensions.Logging.Configuration;
using Microsoft.Extensions.Logging.EventLog;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
```

```
builder.Services.AddWindowsService(options =>
{
    options.ServiceName = ".NET Joke Service";
});

LoggerProviderOptions.RegisterProviderOptions<
    EventLogSettings, EventLogLoggerProvider>(builder.Services);

builder.Services.AddSingleton<JokeService>();
builder.Services.AddHostedService<WindowsBackgroundService>();

IHost host = builder.Build();
host.Run();
```

`AddWindowsService` 확장 메서드는 Windows 서비스로 작동하도록 앱을 구성합니다. 서비스 이름이 `".NET Joke Service"`로 설정되었습니다. 호스트된 서비스는 종속성 주입을 위해 등록됩니다.

서비스 등록에 대한 자세한 내용은 .NET 종속성 주입을 참조하세요.

앱 게시

.NET Worker Service 앱을 Windows 서비스로 만들려면 앱을 단일 파일 실행 파일로 게시하는 것이 좋습니다. 파일 시스템 주위에 있는 종속 파일이 없으므로 자체 포함 실행 파일이 있는 것은 오류가 적습니다. 그러나 Windows 서비스 제어 관리자가 대상으로 지정할 수 있는 **.exe* 파일을 만드는 한 완벽하게 허용되는 다른 게시 형식을 선택할 수 있습니다.

> ⓘ **중요**
>
> 다른 게시 방법은 **.exe*대신 **.dll* 빌드하고 Windows Service Control Manager를 사용하여 게시된 앱을 설치할 때 .NET CLI에 위임하고 DLL을 전달하는 것입니다. 자세한 내용은 .NET CLI: dotnet 명령 참조하세요.
>
> PowerShell
>
> ```
> sc.exe create ".NET Joke Service" binpath= "C:\Path\To\dotnet.exe
> C:\Path\To\App.WindowsService.dll"
> ```

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
```

```
    <RootNamespace>App.WindowsService</RootNamespace>
    <OutputType>exe</OutputType>
    <PublishSingleFile Condition="'$(Configuration)' ==
'Release'">true</PublishSingleFile>
    <RuntimeIdentifier>win-x64</RuntimeIdentifier>
    <PlatformTarget>x64</PlatformTarget>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.10" />
    <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="9.0.10" />
  </ItemGroup>
</Project>
```

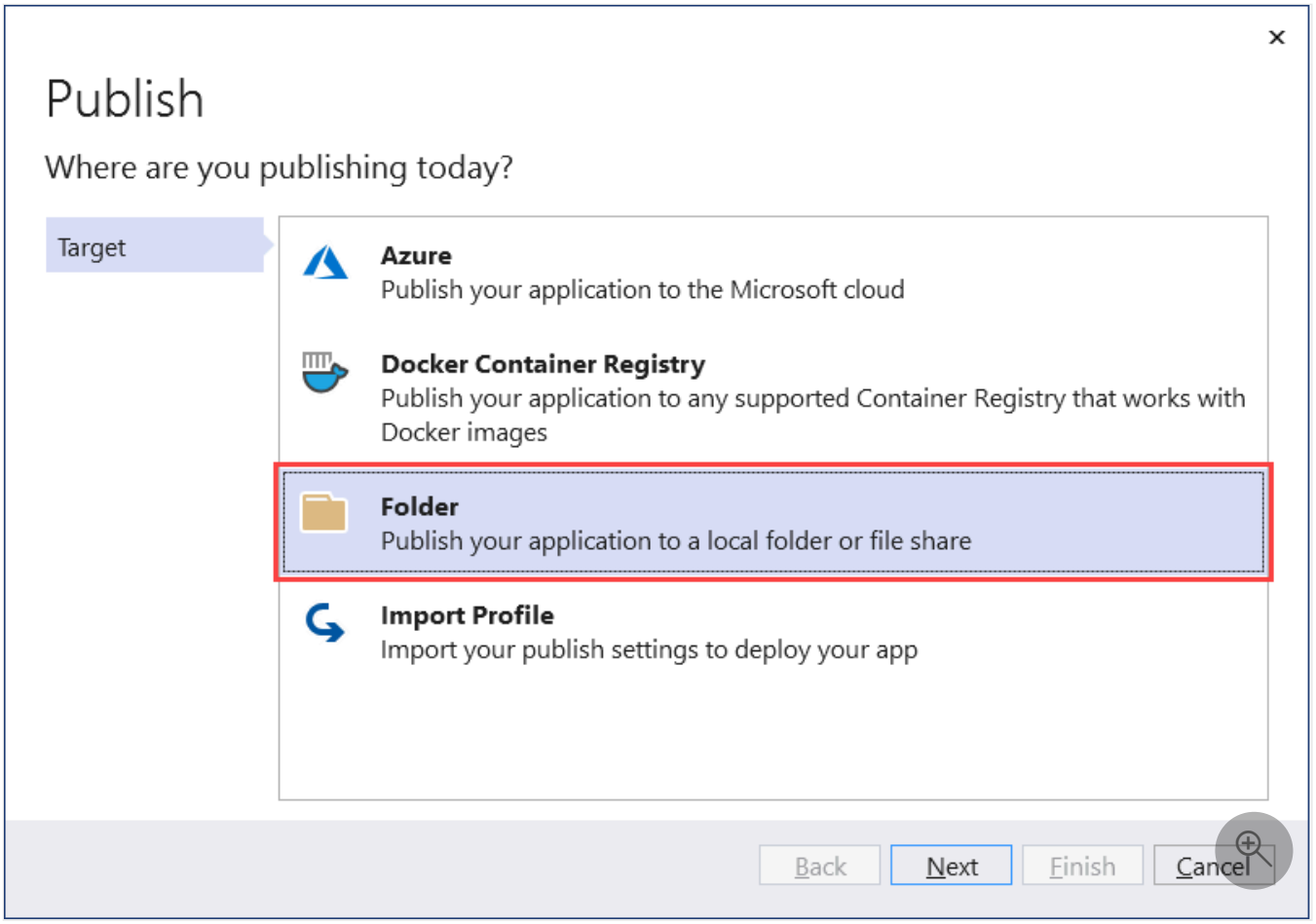
프로젝트 파일의 앞에 강조 표시된 줄은 다음 동작을 정의합니다.

- `<OutputType>exe</OutputType>`: 콘솔 애플리케이션을 만듭니다.
- `<PublishSingleFile Condition="'$(Configuration)' == 'Release'">true</PublishSingleFile>`: 단일 파일 게시를 사용하도록 설정합니다.
- `<RuntimeIdentifier>win-x64</RuntimeIdentifier>`: 의 `win-x64`를 지정합니다.
- `<PlatformTarget>x64</PlatformTarget>`: 64비트 대상 플랫폼 CPU를 지정합니다.

Visual Studio에서 앱을 게시하려면 유지되는 게시 프로필을 만들 수 있습니다. 게시 프로필은 XML 기반이며 *.pubxml* 파일 확장자를 가집니다. Visual Studio는 이 프로필을 사용하여 암시적으로 앱을 게시하는 반면, .NET CLI를 사용하는 경우 사용할 게시 프로필을 명시적으로 지정해야 합니다.

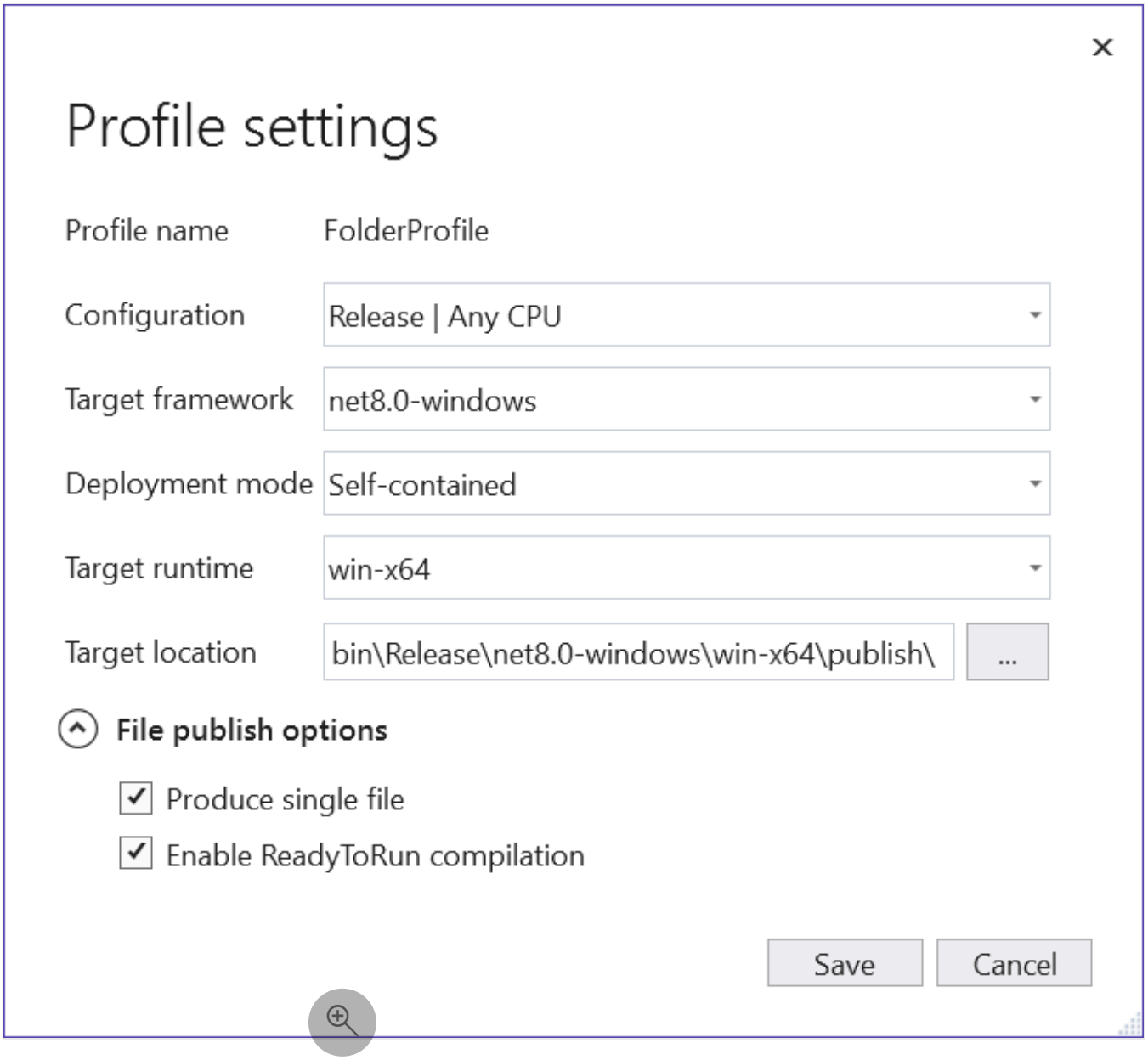
솔루션 탐색기에서 프로젝트를 마우스 오른쪽 버튼으로 클릭한 다음 **게시**를 선택합니다. 그런 다음, **게시 프로필 추가** 을 선택하여 프로필을 만듭니다. **게시** 대화 상자에서 **폴더** 를 **대상**로 선택합니다.

Visual Studio 게시 대화 상자



기본 **위치**를 그대로 두고, **마침**을 선택합니다. 프로필이 만들어지면, **모든 설정을 표시**선택한 후, **프로필 설정을**확인하세요.

Visual Studio 프로필 설정The Visual Studio Profile settings



The Visual Studio Profile settings

다음 설정이 지정되어 있는지 확인합니다.

- **배포 모드**: 자체 포함
- **단일 파일**생성: 체크됨
- **ReadyToRun 컴파일을 사용하도록 설정**: 선택됨
- **사용되지 않는 어셈블리 자르기(미리 보기)**: 선택 취소됨

마지막으로 **게시**선택합니다. 앱이 컴파일되고 결과 .exe 파일이 */publish* 출력 디렉터리에 게시됩니다.

또는 .NET CLI를 사용하여 앱을 게시할 수 있습니다.

.NET CLI

```
dotnet publish --output "C:\custom\publish\directory"
```

자세한 내용은 dotnet publish참조하세요.

> **ⓘ 중요**
>
> .NET 6에서는 `<PublishSingleFile>true</PublishSingleFile>` 설정으로 앱을 디버그하려고 시도하면, 앱을 디버그할 수 없습니다. 자세한 내용은 'PublishSingleFile' .NET 6 앱 디버깅할 때 CoreCLR에 연결할 수 없음을 참조하세요.

Windows 서비스 만들기

PowerShell 사용에 익숙하지 않고 서비스에 대한 설치 관리자를 만들려면 Windows 서비스 설치 관리자 만들기참조하세요. 그렇지 않은 경우 Windows 서비스를 만들려면 네이티브 Windows 서비스 제어 관리자(*sc.exe*) create 명령을 사용합니다. 관리자 권한으로 PowerShell을 실행합니다.

PowerShell

```
sc.exe create ".NET Joke Service" binpath= "C:\Path\To\App.WindowsService.exe"
```

> **팁**
>
> 호스트 구성콘텐츠 루트를 변경해야 하는 경우 `binpath` 지정할 때 명령줄 인수로 전달할 수 있습니다.
>
> PowerShell
>
> ```
> sc.exe create "Svc Name" binpath= "C:\Path\To\App.exe --contentRoot
> C:\Other\Path"
> ```

출력 메시지가 표시됩니다.

PowerShell

```
[SC] CreateService SUCCESS
```

자세한 내용은 sc.exe만들기를 참조하세요.

Windows 서비스 구성

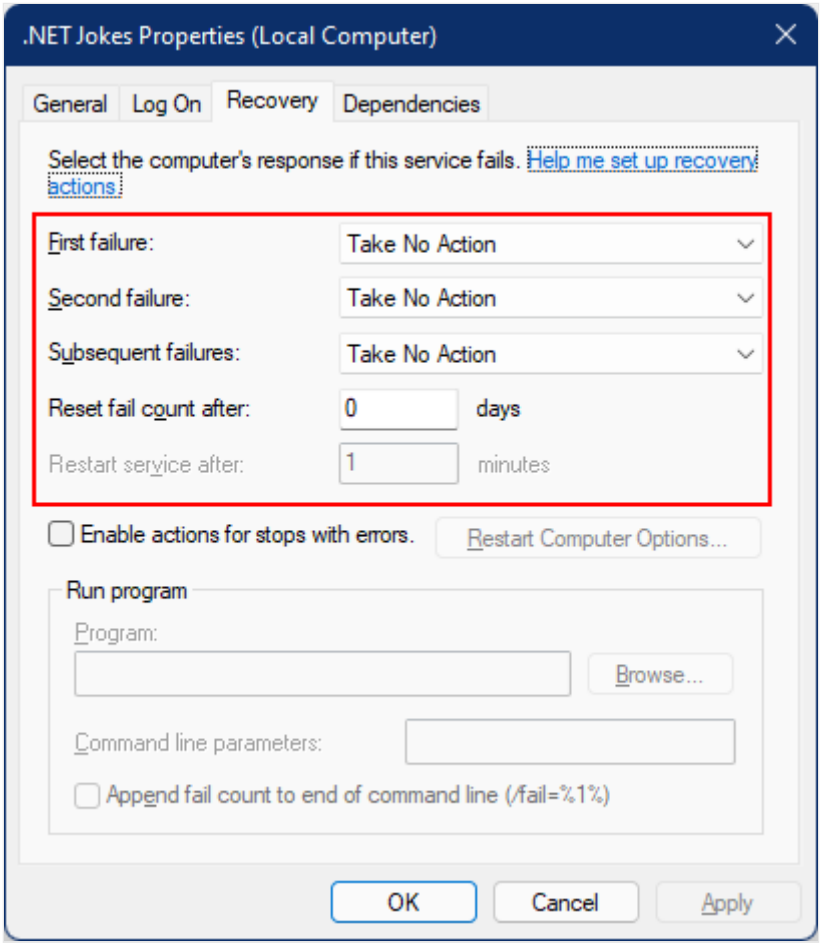
서비스를 만든 후 필요에 따라 구성할 수 있습니다. 서비스 기본값이 괜찮으면 서비스 기능 확인 섹션으로 건너뜁니다.

Windows 서비스는 복구 구성 옵션을 제공합니다. `sc.exe qfailure "<Service Name>"`(`<Service Name>` 서비스의 이름) 명령을 사용하여 현재 구성을 쿼리하여 현재 복구 구성 값을 읽을 수 있습니다.

PowerShell

```
sc qfailure ".NET Joke Service"
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: .NET Joke Service
        RESET_PERIOD (in seconds)    : 0
        REBOOT_MESSAGE               :
        COMMAND_LINE                 :
```

이 명령은 아직 구성되지 않았기 때문에 기본값인 복구 구성을 출력합니다.



복구를 구성하려면 서비스의 이름인 `sc.exe failure "<Service Name>"` 위치에 `<Service Name>`를 사용합니다.

PowerShell

```
sc.exe failure ".NET Joke Service" reset= 0 actions=
restart/60000/restart/60000/run/1000
[SC] ChangeServiceConfig2 SUCCESS
```

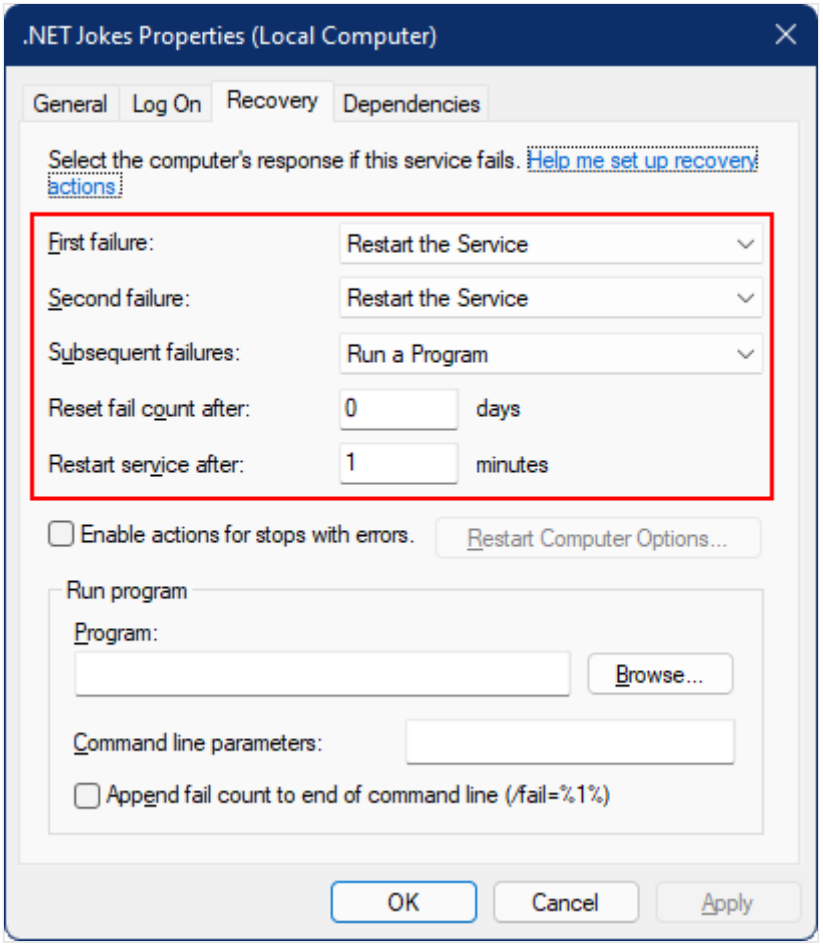
> **팁**
>
> 복구 옵션을 구성하려면 터미널 세션을 관리자 권한으로 실행해야 합니다.

성공적으로 구성되면 `sc.exe qfailure "<Service Name>"` 명령을 사용하여 값을 다시 한 번 쿼리할 수 있습니다.

PowerShell

```
sc qfailure ".NET Joke Service"
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: .NET Joke Service
        RESET_PERIOD (in seconds)    : 0
        REBOOT_MESSAGE               :
        COMMAND_LINE                 :
        FAILURE_ACTIONS              : RESTART -- Delay = 60000 milliseconds.
                                       RESTART -- Delay = 60000 milliseconds.
                                       RUN PROCESS -- Delay = 1000 milliseconds.
```

구성된 다시 시작 값이 표시됩니다.



서비스 복구 옵션 및 .NET `BackgroundService` 인스턴스

.NET 6에서는 새 호스팅 예외 처리 동작이 .NET에 추가되었습니다. BackgroundServiceExceptionBehavior 열거형은 `Microsoft.Extensions.Hosting` 네임스페이스에 추가되었으며 예외가 throw된 경우 서비스의 동작을 지정하는 데 사용됩니다. 다음 표에서는 사용 가능한 옵션을 나열합니다.

테이블 확장

선택	묘사
Ignore	`BackgroundService`에서 던져진 예외를 무시합니다.
StopHost	`IHost` 처리되지 않은 예외가 throw되면 중지됩니다.

.NET 6 이전의 기본 동작은 `Ignore`때문에 *좀비 프로세스*(아무 작업도 수행하지 않은 실행 프로세스)가 발생합니다. .NET 6에서는 기본 동작이 `StopHost`이며, 이는 예외가 throw될 때 호스트가 중지되는 결과를 초래합니다. 그러나 완전히 중지됩니다. 즉, Windows 서비스 관리 시스템이 서비스를 다시 시작하지 않습니다. 서비스를 다시 시작하도록 올바르게 허용하려면 0이 아닌 종료 코드로 Environment.Exit 호출할 수 있습니다. 강조 표시된 다음 `catch` 블록을 고려합니다.

C#

```
namespace App.WindowsService;

public sealed class WindowsBackgroundService(
    JokeService jokeService,
    ILogger<WindowsBackgroundService> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        try
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                string joke = jokeService.GetJoke();
                logger.LogWarning("{Joke}", joke);

                await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
            }
        }
        catch (OperationCanceledException)
        {
            // When the stopping token is canceled, for example, a call made from
services.msc,
            // we shouldn't exit with a non-zero exit code. In other words, this is
expected...
        }
```

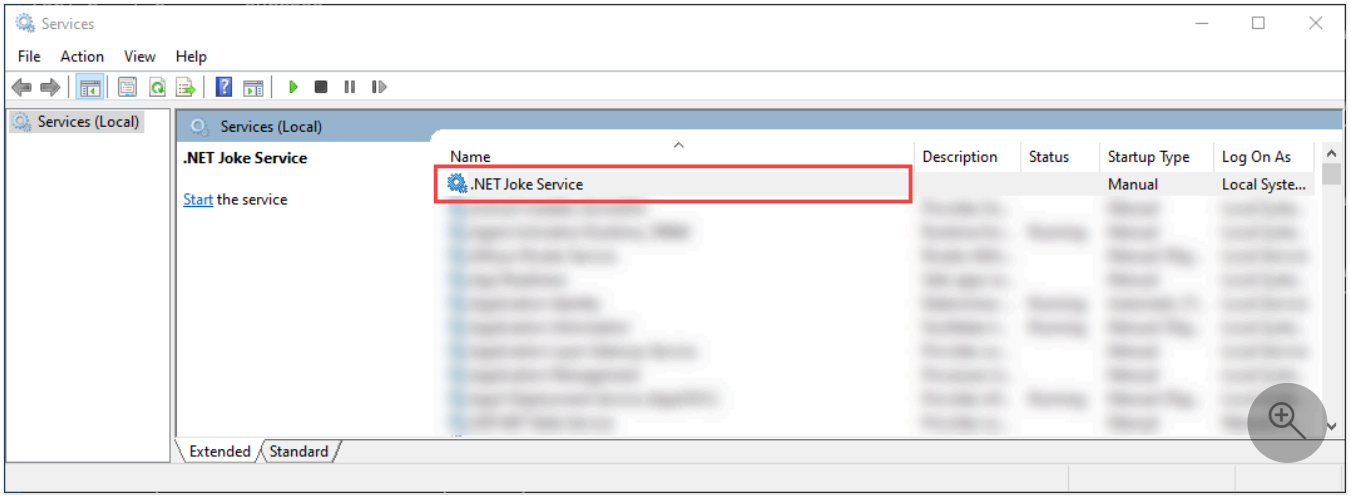
```
        catch (Exception ex)
        {
            logger.LogError(ex, "{Message}", ex.Message);

            // Terminates this process and returns an exit code to the operating
system.
            // This is required to avoid the 'BackgroundServiceExceptionBehavior',
which
            // performs one of two scenarios:
            // 1. When set to "Ignore": will do nothing at all, errors cause zombie
services.
            // 2. When set to "StopHost": will cleanly stop the host, and log
errors.
            //
            // In order for the Windows Service Management system to leverage
configured
            // recovery options, we need to terminate the process with a non-zero
exit code.
            Environment.Exit(1);
        }
    }
}
```

서비스 기능 확인

Windows 서비스로 만든 앱을 보려면 **Services**엽니다. Windows 키(또는 Ctrl + `Esc`)를 선택하고 "서비스"에서 검색합니다. **Services** 앱에서 해당 이름으로 서비스를 찾을 수 있어야 합니다.

> ⓘ **중요**
>
> 기본적으로 일반(비관리자) 사용자는 Windows 서비스를 관리할 수 없습니다. 이 앱이 예상대로 작동하는지 확인하려면 관리자 계정을 사용해야 합니다.

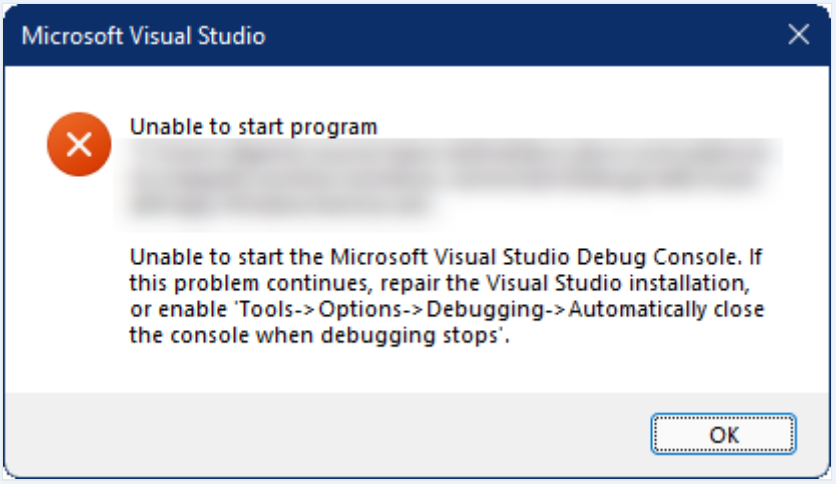
서비스 사용자 인터페이스를



서비스가 예상대로 작동하는지 확인하려면 다음을 수행해야 합니다.

- 서비스 시작
- 로그 보기
- 서비스 중지

> ⓘ **중요**
>
> 애플리케이션을 디버그하려면 Windows Services 프로세스 내에서 현재 실행 중인 실행 파일을 디버그하려고 시도하지 *말아야 합니다*.



Windows 서비스 시작

Windows 서비스를 시작하려면 `sc.exe start` 명령을 사용합니다.

PowerShell

```
sc.exe start ".NET Joke Service"
```

다음과 유사한 출력이 표시됩니다.

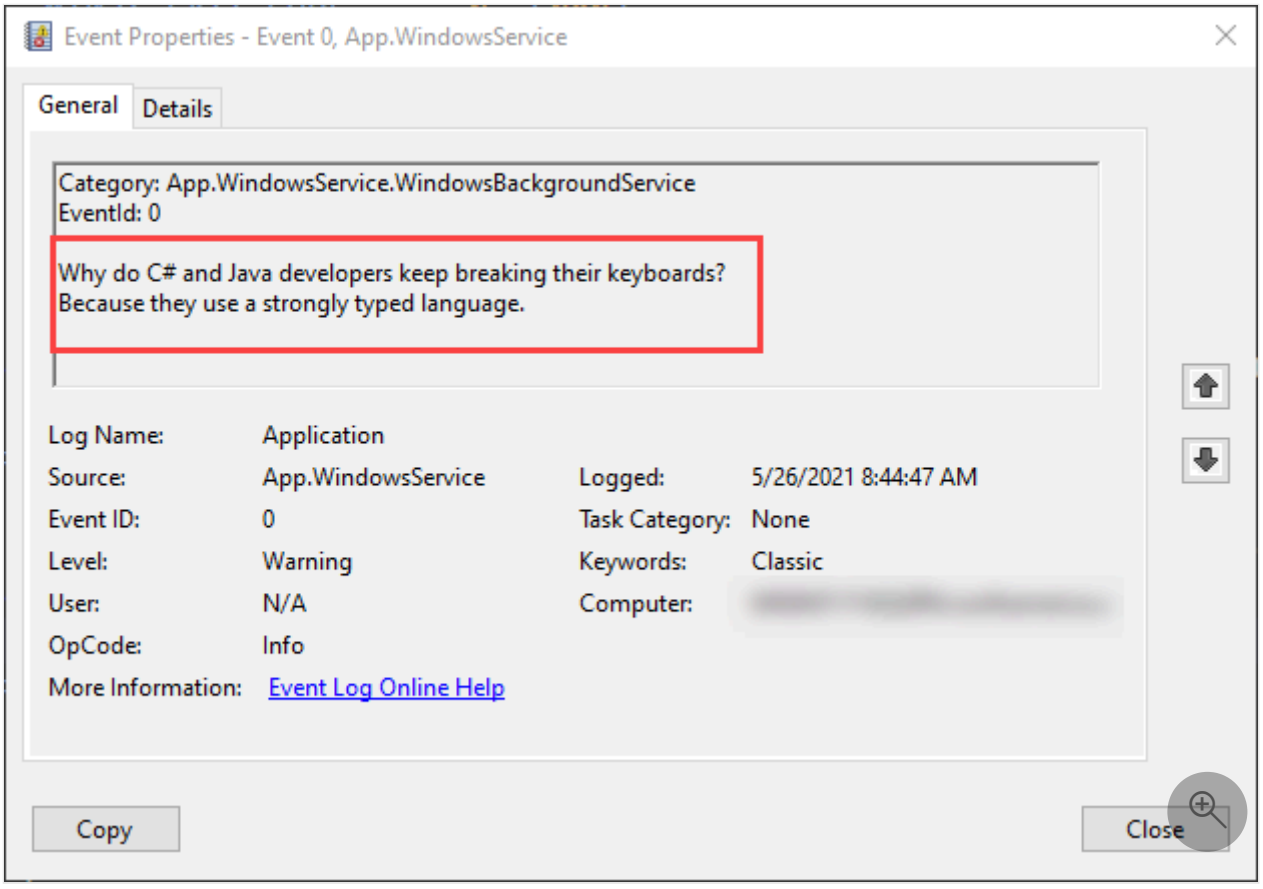
PowerShell

```
SERVICE_NAME: .NET Joke Service
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 2  START_PENDING
                            (NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE  : 0  (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x7d0
    PID                : 37636
    FLAGS
```

서비스 **상태**가 `START_PENDING`에서 **실행 중인**로 전환됩니다.

로그 보기

로그를 보려면 **이벤트 뷰어**엡니다. Windows 키(또는 Ctrl + `Esc`)를 선택하고 `"Event Viewer"`검색합니다. **이벤트 뷰어(로컬)**>**Windows 로그**>**애플리케이션** 노드를 선택합니다. 앱 네임스페이스와 일치하는 **원본**을 포함한 **경고** 수준의 항목을 확인해야 합니다. 항목을 두 번 클릭하거나 마우스 오른쪽 단추를 클릭하고 **이벤트 속성** 선택하여 세부 정보를 봅니다.



기록된 세부 정보가 포함된 이벤트 속성 대화 상자

이벤트 로그로그가 표시되면 서비스를 중지해야 합니다. 그것은 분당 한 번 임의의 농담을 기록하도록 설계되었습니다. 이는 *의도적인 동작이지만* 프로덕션 서비스에는 실용적이지 않습니다.

Windows 서비스 중지

Windows 서비스를 중지하려면 `sc.exe stop` 명령을 사용합니다.

PowerShell

```
sc.exe stop ".NET Joke Service"
```

다음과 유사한 출력이 표시됩니다.

PowerShell

```
SERVICE_NAME: .NET Joke Service
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 3  STOP_PENDING
                            (STOPPABLE, NOT_PAUSABLE, ACCEPTS_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE  : 0  (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x0
```

서비스 **상태** `STOP_PENDING` 에서 **중지됨** 로 전환됩니다.

Windows 서비스 삭제

Windows 서비스를 삭제하려면 네이티브 Windows 서비스 제어 관리자(sc.exe) 삭제 명령을 사용합니다. 관리자 권한으로 PowerShell을 실행합니다.

> ⓘ **중요**
>
> 서비스가 **중지됨** 상태가 아니면 즉시 삭제되지 않습니다. delete 명령을 실행하기 전에 서비스가 중지되었는지 확인합니다.

PowerShell

```
sc.exe delete ".NET Joke Service"
```

출력 메시지가 표시됩니다.

PowerShell

```
[SC] DeleteService SUCCESS
```

자세한 내용은 sc.exe 삭제참조하세요.

참고

- Windows 서비스 설치 관리자 만들기
- .NET에서의 작업자 서비스
- 큐 서비스 만들기
- BackgroundService 내에서 범위가 지정된 서비스 사용
- IHostedService 인터페이스 구현

다음

Windows 서비스 설치 관리자 만들기

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

Windows 서비스 설치 프로그램 만들기

.NET Windows 서비스를 만들 때(.NET Framework Windows 서비스로 오인하지 않음) 서비스에 대한 설치 관리자를 만들 수 있습니다. 설치 관리자가 없으면 사용자는 서비스를 설치하고 구성하는 방법을 알아야 합니다. 설치 관리자는 앱의 실행 파일을 번들로 묶고 사용자 지정 가능한 설치 사용자 환경을 노출합니다. 이 자습서는 Windows 서비스 만들기 자습서의 연속입니다. .NET Windows 서비스에 대한 설치 관리자를 만드는 방법을 보여 줍니다.

이 자습서에서는 다음 방법을 알아봅니다.

- ✓ Visual Studio 설치 관리자 프로젝트 확장을 설치합니다.
- ✓ 설치 프로젝트를 만듭니다.
- ✓ 설치를 지원하도록 기존 .NET Worker 프로젝트를 업데이트합니다.
- ✓ Windows 서비스 제어 관리자를 사용하여 설치 및 제거를 자동화합니다.

필수 조건

- Windows 서비스 만들기 자습서를 완료했거나 샘플 리포지토리를 복제할 준비를 해야 합니다.
- .NET 8.0 SDK 버전 이상
- 윈도우 운영 체제
- .NET IDE(통합 개발 환경)
  - Visual Studio 자유롭게 사용하세요.
- 기존 .NET Windows 서비스

도구 종속성 설치

Wix 도구 집합

먼저 Wix 도구 집합을 설치합니다. Wix 도구 집합은 XML 소스 코드에서 Windows 설치 패키지를 빌드하는 도구 집합입니다.

.NET CLI

```
dotnet tool install --global wix
```

다음으로 VS2022용 HeatWave 확장을 설치합니다. 설치한 후 Visual Studio를 다시 시작하면 사용할 수 있는 새 프로젝트 템플릿이 표시됩니다.

기존 프로젝트 가져오기

이 자습서는 BackgroundService 자습서를 사용하여 Windows 서비스 만들기 의 일부로 만든 앱을 기반으로 합니다. 샘플 리포지토리를 복제하거나 이전 자습서에서 빌드한 앱을 사용할 수 있습니다.

> **팁**
>
> 모든 '.NET 작업자' 예제 소스 코드는 **샘플 브라우저에서** 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET 작업자**를 참조하세요.

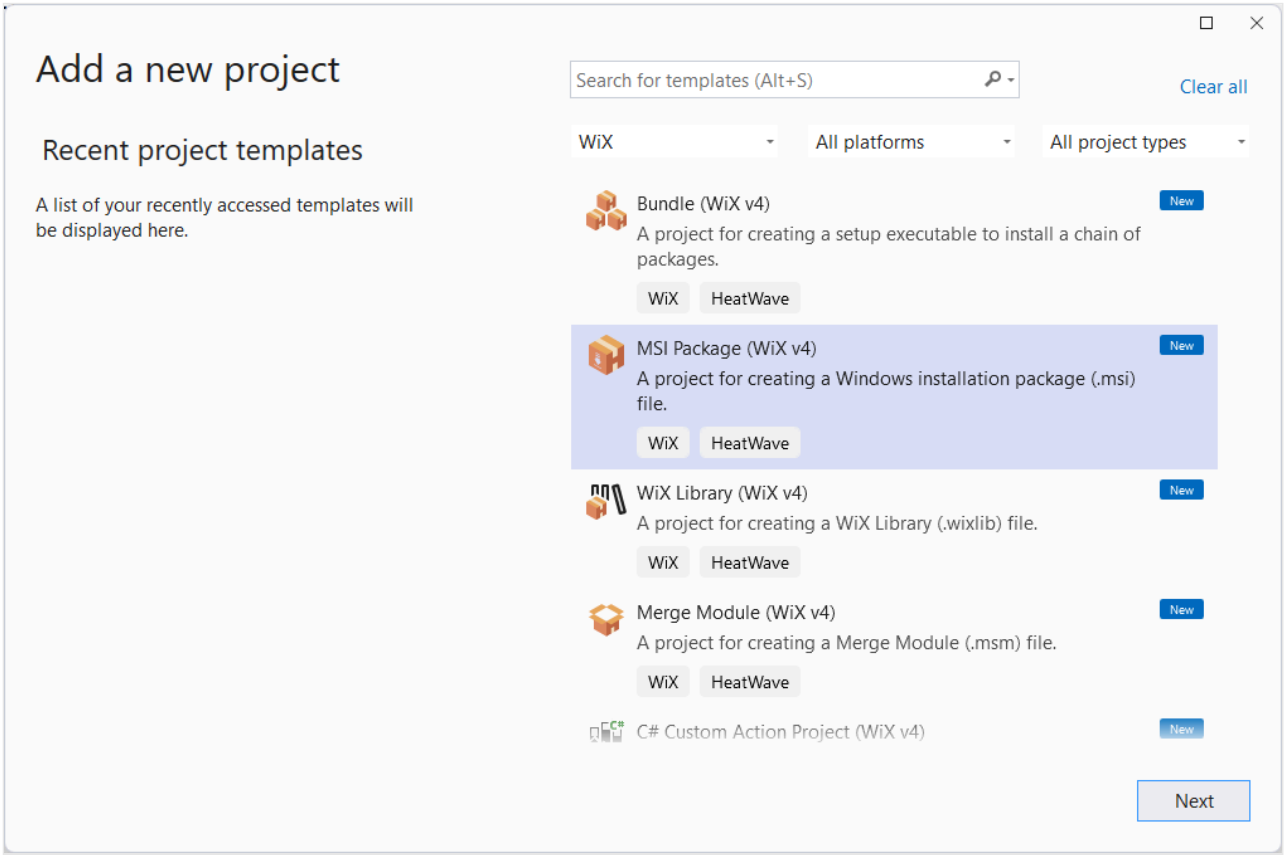
Wix 도구 집합

Visual Studio에서 솔루션을 열고 `F5` 를 선택하여 앱이 예상대로 빌드되고 실행되도록 합니다. `Ctrl` + `C` 를 눌러 앱을 중지합니다.

새 설치 프로젝트 추가

Wix 도구 집합

새 Wix 설치 프로젝트를 추가하려면 **솔루션 탐색기**에서 솔루션을 마우스 오른쪽 단추로 클릭하고 **새 프로젝트 추가**>를 선택합니다.



사용 가능한 템플릿에서 **MSI 패키지(Wix v4)** 를 선택한 다음, **다음**을 선택합니다. 원하는 **이름** 및 **위치를** 입력한 다음 **만들기**를 선택합니다.

설치 프로그램 프로젝트 구성

Wix 도구 집합

설치 프로젝트를 구성하려면 먼저 프로젝트에 대한 참조를 `App.WindowsService` 추가해야 합니다. **솔루션 탐색기**에서 설치 프로젝트를 마우스 오른쪽 단추로 클릭한 다음 **프로젝트 참조 추가>**를 선택합니다.

템플릿에는 예제 구성 요소 및 지역화 파일이 포함됩니다. *Package.wxs* 파일만 남겨 두는 파일을 삭제합니다. 이제 프로젝트에 다음과 유사한 요소가 포함되어 `ProjectReference` 야 합니다.

XML

```xml
<Project Sdk="WixToolset.Sdk/4.0.0">
  <ItemGroup>
    <ProjectReference Include="..\App.WindowsService.csproj" />
  </ItemGroup>
</Project>
```

프로젝트 참조가 추가된 후 *Package.wxs* 파일을 구성합니다. 편집기에서 파일을 열고 내용을 다음으로 바꿉니다.

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>

<!-- Define the variables in "$(var.*) expressions" -->
<?define Name = ".NET Joke Service" ?>
<?define Manufacturer = "Microsoft" ?>
<?define Version = "1.0.0.0" ?>
<?define UpgradeCode = "9ED3FF33-8718-444E-B44B-69A2344B7E98" ?>

<Wix xmlns="http://wixtoolset.org/schemas/v4/wxs">
    <Package Name="$(Name)"
             Manufacturer="$(Manufacturer)"
             Version="$(Version)"
             UpgradeCode="$(var.UpgradeCode)"
             Compressed="true">

        <!-- Allow upgrades and prevent downgrades -->
        <MajorUpgrade DowngradeErrorMessage="A later version of [ProductName] is
already installed. Setup will now exit." />
```

```
        <!-- Define the directory structure -->
        <StandardDirectory Id="ProgramFiles64Folder">

            <!-- Create a folder inside program files -->
            <Directory Id="ROOTDIRECTORY" Name="$(var.Manufacturer)">

                <!-- Create a folder within the parent folder given the name -->
                <Directory Id="INSTALLFOLDER" Name="$(Name)" />
            </Directory>
        </StandardDirectory>

        <!-- The files inside this DirectoryRef are linked to
             the App.WindowsService directory via INSTALLFOLDER -->
        <DirectoryRef Id="INSTALLFOLDER">

            <!-- Create a single component which is the App.WindowsService.exe
file -->
            <Component Id="ServiceExecutable" Bitness="always64">

                <!-- Copies the App.WindowsService.exe file using the
                     project reference preprocessor variables -->
                <File Id="App.WindowsService.exe"

Source="$(var.App.WindowsService.TargetDir)publish\App.WindowsService.exe"
                      KeyPath="true" />

                <!-- Remove all files from the INSTALLFOLDER on uninstall -->
                <RemoveFile Id="ALLFILES" Name="*.*" On="both" />

                <!-- Tell WiX to install the Service -->
                <ServiceInstall Id="ServiceInstaller"
                                Type="ownProcess"
                                Name="App.WindowsService"
                                DisplayName="$(Name)"
                                Description="A joke service that periodically
logs nerdy humor."
                                Start="auto"
                                ErrorControl="normal" />

                <!-- Tell WiX to start the Service -->
                <ServiceControl Id="StartService"
                                Start="install"
                                Stop="both"
                                Remove="uninstall"
                                Name="App.WindowsService"
                                Wait="true" />
            </Component>
        </DirectoryRef>

        <!-- Tell WiX to install the files -->
        <Feature Id="Service" Title="App.WindowsService Setup" Level="1">
            <ComponentRef Id="ServiceExecutable" />
        </Feature>
```

```
    </Package>
</Wix>
```

프로젝트를 빌드할 때 출력은 서비스를 설치하고 제거하는 데 사용할 수 있는 MSI 파일입니다.

테스트 설치

Wix 도구 집합

설치 관리자를 테스트하려면 *App.WindowsService* 프로젝트를 게시합니다. **솔루션 탐색기**에서 프로젝트를 마우스 오른쪽 단추로 클릭한 다음 **게시**를 선택합니다. 이전 자습서에서 만든 프로필과 함께 게시되면 실행 파일은 게시 디렉터리에 있습니다. 다음으로 설치 프로젝트를 **빌드** 하고 설치 관리자를 실행합니다.

관리자 권한으로 설치를 실행해야 합니다. 이렇게 하려면 MSI 파일을 마우스 오른쪽 단추로 클릭한 다음 **관리자 권한으로 실행을** 선택합니다.

서비스가 설치되면 **서비스를** 열어 실행 중인 서비스를 확인할 수 있습니다. 서비스를 제거하려면 **Windows 프로그램 추가 또는 제거** 기능을 사용하여 설치 관리자를 호출합니다.

참고하십시오

- .NET에서의 작업자 서비스
- 큐 서비스 만들기
- BackgroundService 내에서 범위가 지정된 서비스 사용
- IHostedService 인터페이스 구현

Last updated on 2025. 10. 20.

`IHostedService` 인터페이스 구현

제공된 것 이상으로 한정된 BackgroundService제어가 필요한 경우 사용자 고유 IHostedService 의 컨트롤을 구현할 수 있습니다. 인터페이스는 IHostedService .NET의 모든 장기 실행 서비스의 기초입니다. 사용자 정의 구현은 AddHostedService<THostedService>(IServiceCollection) 확장 메서드에 등록됩니다.

이 튜토리얼에서는 다음을 배우게 됩니다:

- ✓ IHostedService 및 IAsyncDisposable 인터페이스를 구현합니다.
- ✓ 타이머 기반 서비스를 만듭니다.
- ✓ 종속성 주입 및 로깅을 사용하여 사용자 지정 구현을 등록합니다.

> **팁**
>
> ".NET의 작업자" 예제 소스 코드는 모두 **샘플 브라우저** 에서 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET 작업자**를 참조하세요.

필수 조건

- .NET 8.0 SDK 버전 이상
- .NET IDE(통합 개발 환경)
  - Visual Studio 자유롭게 사용하세요.

새 프로젝트 만들기

Visual Studio를 사용하여 새 작업자 서비스 프로젝트를 만들려면 **파일** > **새** > **Project...**선택합니다. **새 프로젝트 만들기** 대화 상자에서 "작업자 서비스"를 검색하고 작업자 서비스 템플릿을 선택합니다. .NET CLI를 사용하려는 경우 작업 디렉터리에서 즐겨 찾는 터미널을 엽니다. `dotnet new` 명령을 실행하고 `<Project.Name>`을(를) 원하는 프로젝트 이름으로 바꿉니다.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

.NET CLI 새 작업자 서비스 프로젝트 명령에 대한 자세한 내용은 dotnet new Worker를 참조하세요.

> **팁**

Visual Studio Code를 사용하는 경우 통합 터미널에서 .NET CLI 명령을 실행할 수 있습니다. 자세한 내용은 Visual Studio Code: 통합 터미널을 참조하세요.

타이머 서비스 만들기

타이머 기반 백그라운드 서비스는 클래스를 System.Threading.Timer 사용합니다. 타이머가 `DoWork` 메서드를 트리거합니다. 서비스 컨테이너가 IHostLifetime.StopAsync(CancellationToken)에서 삭제될 때 IAsyncDisposable.DisposeAsync()에서 타이머가 비활성화되고 삭제됩니다.

템플릿의 `Worker` 내용을 다음 C# 코드로 바꾸고 파일 이름을 *TimerService.cs*.

C#

```
namespace App.TimerHostedService;

public sealed class TimerService(ILogger<TimerService> logger) : IHostedService,
IAsyncDisposable
{
    private readonly Task _completedTask = Task.CompletedTask;
    private int _executionCount = 0;
    private Timer? _timer;

    public Task StartAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation("{Service} is running.", nameof(TimerHostedService));
        _timer = new Timer(DoWork, null, TimeSpan.Zero, TimeSpan.FromSeconds(5));

        return _completedTask;
    }

    private void DoWork(object? state)
    {
        int count = Interlocked.Increment(ref _executionCount);

        logger.LogInformation(
            "{Service} is working, execution count: {Count:#,0}",
            nameof(TimerHostedService),
            count);
    }

    public Task StopAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Service} is stopping.", nameof(TimerHostedService));

        _timer?.Change(Timeout.Infinite, 0);

        return _completedTask;
    }
```

```
    public async ValueTask DisposeAsync()
    {
        if (_timer is IAsyncDisposable timer)
        {
            await timer.DisposeAsync();
        }

        _timer = null;
    }
}
```

> **ⓘ Important**
>
> 의 `Worker` 하위 클래스 BackgroundService였습니다. 이제 `TimerService`은 IHostedService 및 IAsyncDisposable 인터페이스를 모두 구현합니다.

`TimerService`는 `sealed`이며, 자신의 `_timer` 인스턴스에서 `DisposeAsync` 호출을 계단식으로 배열합니다. "연계 삭제 패턴"에 대한 자세한 내용은 DisposeAsync 메서드 구현을 참조하세요.

StartAsync 호출되면 타이머가 인스턴스화되어 타이머가 시작됩니다.

> **팁**
>
> Timer는 이전에 실행된 `DoWork`를 마칠 때까지 기다리지 않으므로 제시된 방법이 모든 시나리오에 적합한 것은 아닐 수 있습니다. Interlocked.Increment 는 여러 스레드가 동시에 업데이트 `_executionCount` 되지 않도록 하는 원자성 작업으로 실행 카운터를 증가시키는 데 사용됩니다.

기존 `Program` 콘텐츠를 다음 C# 코드로 바꿉니다.

C#

```
using App.TimerHostedService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<TimerService>();

IHost host = builder.Build();
host.Run();
```

서비스는 *Program.cs*에서 `AddHostedService` 확장 메서드를 사용하여 등록됩니다. 이 메서드는 둘 다 IHostedService 인터페이스를 구현하므로 BackgroundService 서브클래스를 등록할 때 사용하는 것과 동일한 확장 메서드입니다.

서비스 등록에 대한 자세한 내용은 .NET 종속성 주입을 참조하세요.

서비스 기능 확인

Visual Studio에서 애플리케이션을 실행하려면 `F5` 를 선택하거나 **디버그>시작 디버깅** 메뉴 옵션을 선택합니다. .NET CLI를 사용하는 경우 작업 디렉터리에서 명령을 실행 `dotnet run` 합니다.

.NET CLI

```
dotnet run
```

.NET CLI 실행 명령에 대한 자세한 내용은 dotnet run을 참조하세요.

애플리케이션을 잠시 동안 실행하여 여러 실행 횟수를 증가시키도록 합니다. 다음과 유사한 출력이 표시됩니다.

Output

```
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is running.
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\timer-service
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 1
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 2
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 3
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 4
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is stopping.
```

Visual Studio 내에서 애플리케이션을 실행하는 경우 **디버그>디버깅 중지...를** 선택합니다. 또는 콘솔 창에서 `Ctrl` + `C` 를 선택하여 취소 신호를 표시합니다.

참고하십시오

고려해야 할 몇 가지 관련 자습서가 있습니다.

- .NET에서의 작업자 서비스
- 큐 서비스 만들기
- BackgroundService 내에서 범위가 지정된 서비스 사용

- 를 사용하여 Windows 서비스 만들기 BackgroundService

Last updated on 2026. 01. 24.

Azure에 Worker Service 배포

2025. 10. 14.

이 문서에서는 Azure에 .NET 작업자 서비스를 배포하는 방법을 알아봅니다. 작업자가 Azure Container Registry (ACR)에서 Azure Container Instance (ACI)로 실행되면 클라우드에서 마이크로서비스 역할을 할 수 있습니다. 장기 실행 서비스에 대한 많은 사용 사례가 있으며 이러한 이유로 작업자 서비스가 존재합니다.

이 튜토리얼에서는 다음을 배우게 됩니다:

- ✓ 작업자 서비스를 만듭니다.
- ✓ 컨테이너 레지스트리 리소스를 만듭니다.
- ✓ 이미지를 컨테이너 레지스트리에 푸시합니다.
- ✓ 컨테이너 인스턴스로 배포합니다.
- ✓ 작업자 서비스 기능을 확인합니다.

> **팁**
>
> ".NET의 작업자" 예제 소스 코드는 모두 **샘플 브라우저** 에서 다운로드할 수 있습니다. 자세한 내용은 **코드 샘플 찾아보기: .NET의 작업자를** 참조하세요.

필수 조건

- .NET 5.0 SDK 이상
- Docker Desktop(Windows 또는 Mac).
- 활성 구독이 있는 Azure 계정. 무료로 계정을 만듭니다.
- 선택한 개발자 환경에 따라:
  - Visual Studio 또는 Visual Studio Code.
  - .NET CLI
  - Azure CLI

새 프로젝트 만들기

Visual Studio를 사용하여 새 작업자 서비스 프로젝트를 만들려면**새**> 프로젝트 **파일**>...을 선택합니다. **새 프로젝트 만들기** 대화 상자에서 "작업자 서비스"를 검색하고 작업자 서비스 템플릿을 선택합니다. 원하는 프로젝트 이름을 입력하고, 적절한 위치를 선택한 **다음, 다음**을 선택합니다. **추가 정보** 페이지에서 **대상 프레임워크**의 경우 Docker 사용 옵션을 선택하여 `.NET 5.0`**Docker** 지원을 사용하도록 설정합니다. 원하는 **Docker OS**를 선택합니다.

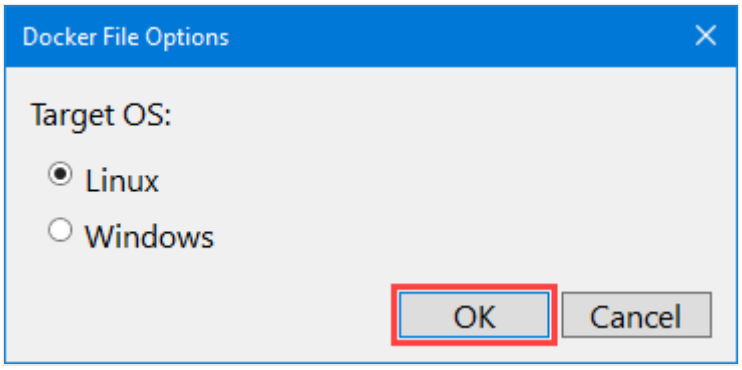
애플리케이션을 빌드하여 종속 패키지를 복원하고 오류 없이 컴파일합니다.

Visual Studio에서 애플리케이션을 빌드하려면 F6을 선택하거나**빌드 솔루션 빌드**> 메뉴 옵션을 선택합니다.

Docker 지원 추가

새 작업자 프로젝트를 만들 때 **Docker 사용** 확인란을 올바르게 선택한 경우 Docker 이미지 빌드 단계로 건너뜁니다.

이 옵션을 선택하지 않은 경우에도 지금 추가할 수 있습니다. Visual Studio에서 *솔루션 탐색기*에서 **프로젝트 노드**를 마우스 오른쪽 단추로 클릭하고**Docker 지원**>를 선택합니다. **대상 OS**를 선택하라는 메시지가 표시됩니다. 기본 OS 선택 항목으로 **확인을** 선택합니다.



Docker 지원에는 *Dockerfile*이 필요합니다. 이 파일은 .NET 작업자 서비스를 Docker 이미지로 빌드하기 위한 포괄적인 지침 집합입니다. *Dockerfile*은 파일 확장명 *없는* 파일입니다. 다음 코드는 *Dockerfile* 예제이며 프로젝트 파일의 루트 디렉터리에 있어야 합니다.

Dockerfile

```
FROM
mcr.microsoft.com/dotnet/runtime:8.0@sha256:e6b552fd7a0302e4db30661b16537f7efcdc0b
67790a47dbf67a5e798582d3a5 AS base
WORKDIR /app

# Creates a non-root user with an explicit UID and adds permission to access the
/app folder
# For more info, please refer to https://aka.ms/vscode-docker-dotnet-configure-
containers
RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser
/app
USER appuser

FROM
mcr.microsoft.com/dotnet/sdk:8.0@sha256:35792ea4ad1db051981f62b313f1be3b46b1f45cad
baa3c288cd0d3056eefb83 AS build
WORKDIR /src
COPY ["App.CloudService.csproj", "./"]
RUN dotnet restore "App.CloudService.csproj"
COPY . .
WORKDIR "/src/."
```

```
RUN dotnet build "App.CloudService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "App.CloudService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "App.CloudService.dll"]
```

Docker 이미지 빌드

Docker 이미지를 빌드하려면 Docker 엔진이 실행 중이어야 합니다.

> **ⓘ 중요**
>
> Docker Desktop 및 Visual Studio를 사용하는 경우 볼륨 공유와 관련된 오류를 방지하려면 *볼륨 공유*를 사용하도록 설정해야 합니다.
>
> 1. Docker Desktop의 **설정** 화면에서 **공유 드라이브를** 선택합니다.
> 2. 프로젝트 파일이 포함된 드라이브를 선택합니다.
>
> 자세한 내용은 Docker를 사용하여 Visual Studio 개발 문제 해결을 참조하세요.

*솔루션 탐색기*에서 **Dockerfile**을 마우스 오른쪽 단추로 클릭하고 **Docker 이미지 빌드를** 선택합니다. 명령어 진행 상태를 표시하는 **출력** 창이 `docker build`에 나타납니다.

명령이 `docker build` 실행되면 *Dockerfile* 의 각 줄을 명령 단계로 처리합니다. 이 명령은 이미지를 빌드하고 이미지를 가리키는 **appcloudservice** 라는 로컬 리포지토리를 만듭니다.

> **팁**
>
> 생성된 *Dockerfile* 은 개발 환경 간에 다릅니다. 예를 들어 Visual Studio에서 Docker 지원을 추가하는 경우 Dockerfile 단계가 다르기 때문에 Visual Studio Code에서 *Docker 이미지를 빌드*하려고 하면 문제가 발생할 수 있습니다. 단일 *개발 환경을* 선택하고 이 자습서 전체에서 사용하는 것이 가장 좋습니다.

컨테이너 레지스트리 만들기

ACR(Azure Container Registry) 리소스를 사용하면 프라이빗 레지스트리에서 컨테이너 이미지 및 아티팩트를 빌드, 저장 및 관리할 수 있습니다. 컨테이너 레지스트리를 만들려면 Azure Portal에서 새 리소스를 만들어야 합니다.

1. **구독** 및 해당 리소스 그룹을 선택하거나 새 **리소스 그룹을** 만듭니다.
2. **레지스트리 이름을 입력합니다**.
3. **위치를** 선택합니다.
4. 적절한 **SKU**(예: **기본**)를 선택합니다.
5. **Review + create**를 선택합니다.
6. **유효성 검사가 통과된** 후 **만들기**를 선택합니다.

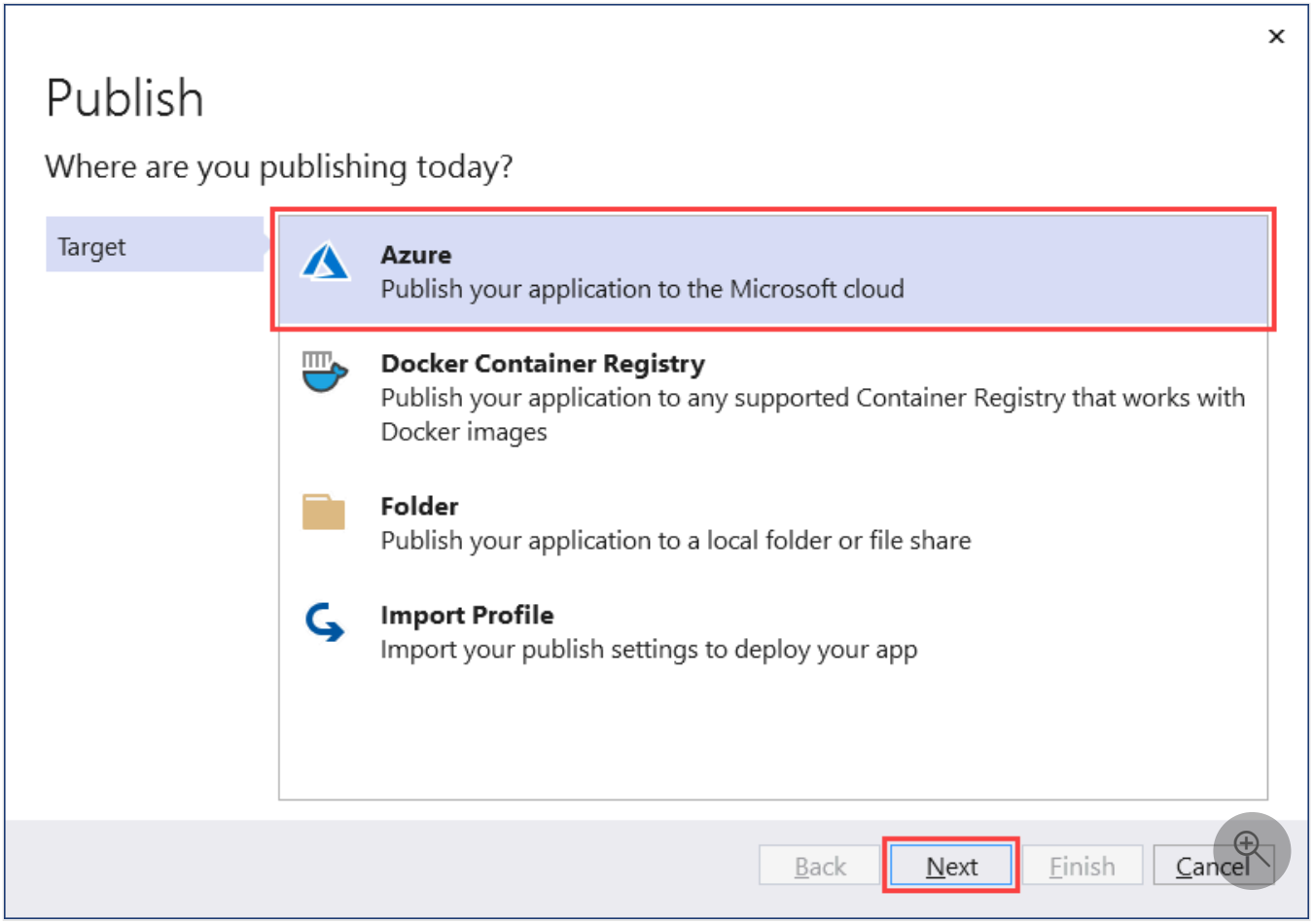
> ⓘ **중요**
>
> 컨테이너 인스턴스를 만들 때 이 컨테이너 레지스트리를 사용하려면 **관리자 사용자를** 사용하도록 설정해야 합니다. **액세스 키를** 선택하고 **관리자 사용자를** 사용하도록 설정합니다.

자세한 내용은 빠른 시작: Azure 컨테이너 레지스트리 만들기를 참조하세요.

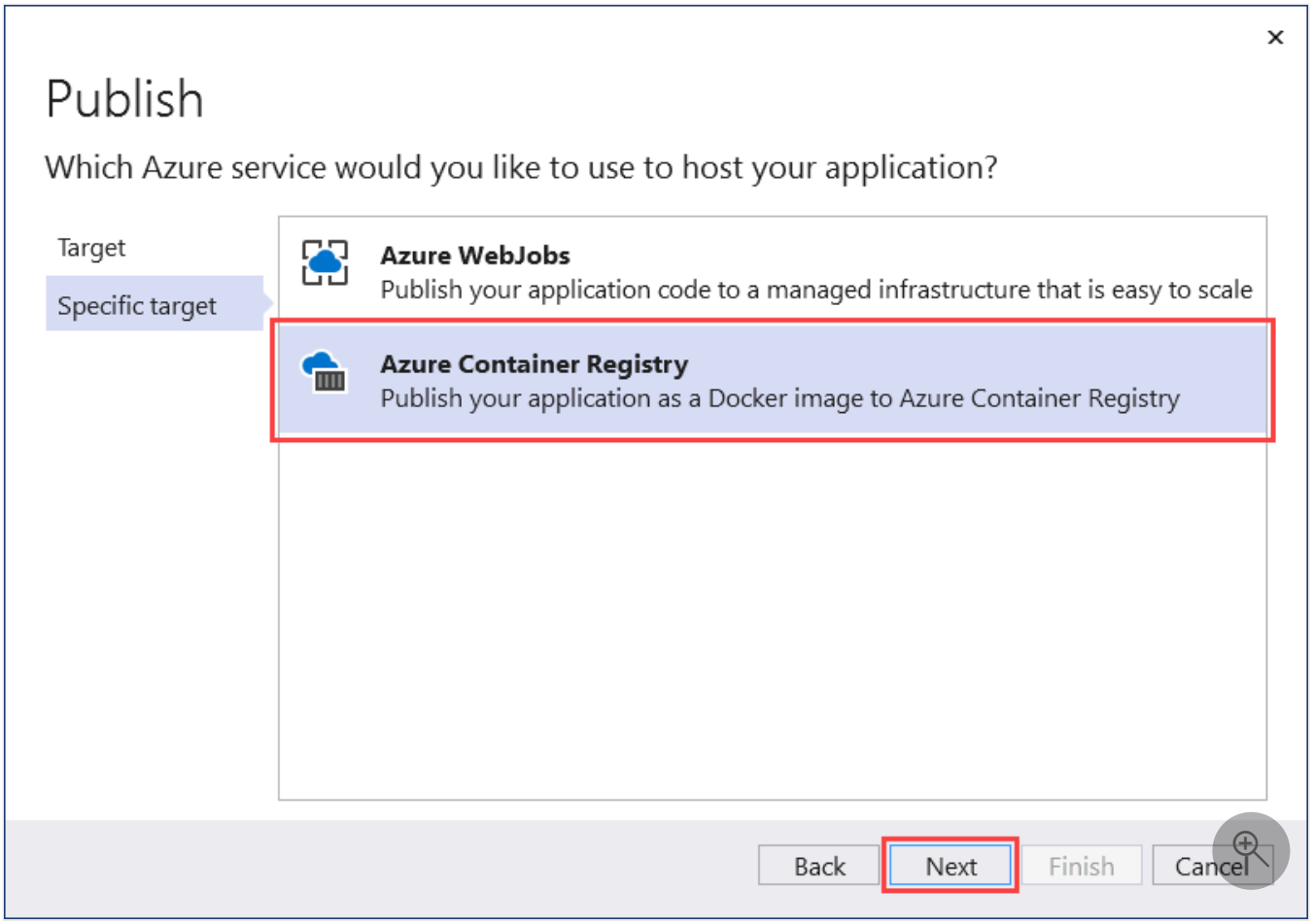
컨테이너 레지스트리에 이미지 푸시

.NET Docker 이미지가 빌드되고 컨테이너 레지스트리 리소스가 만들어짐에 따라 이제 이미지를 컨테이너 레지스트리에 푸시할 수 있습니다.

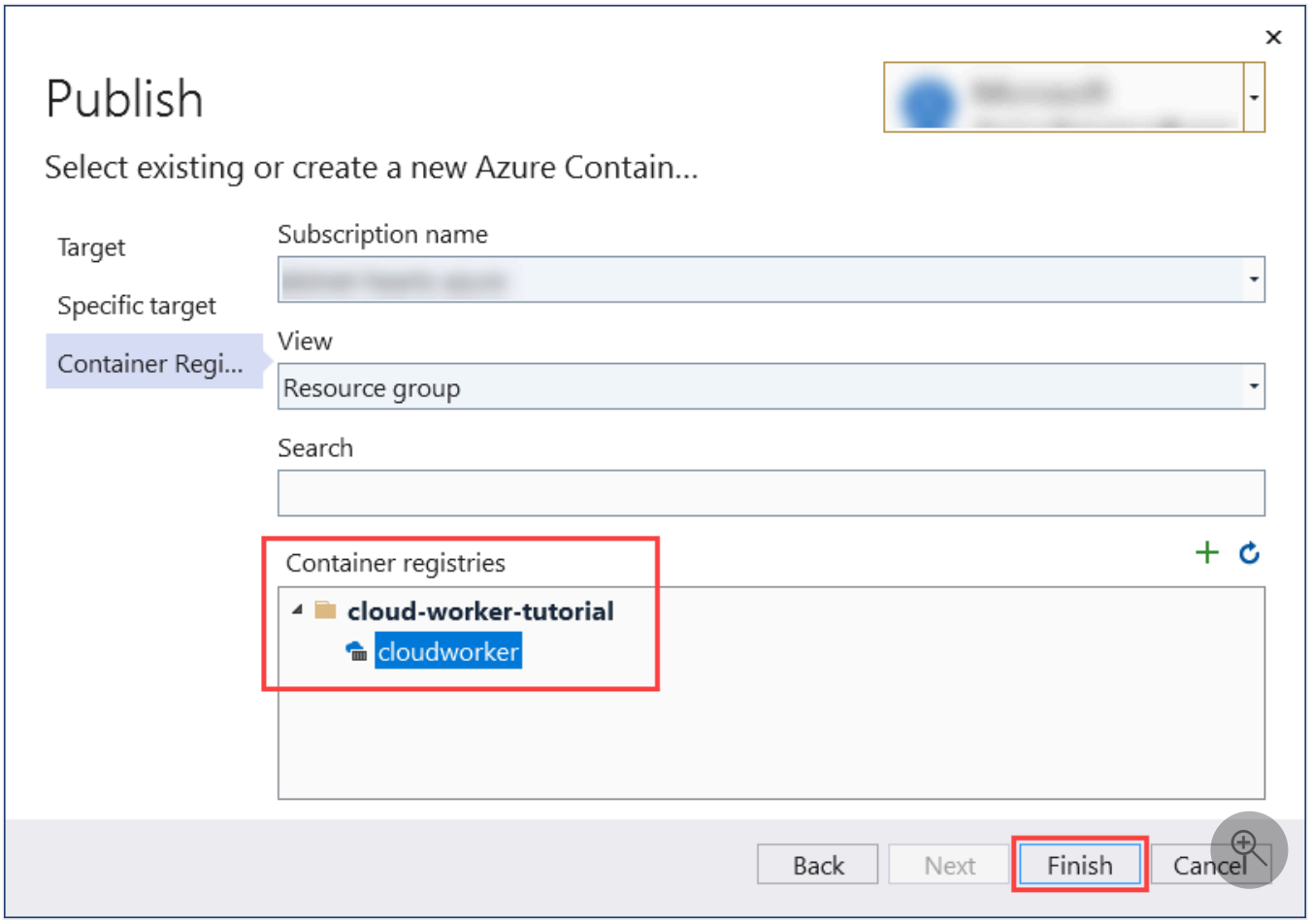
솔루션 탐색기에서 프로젝트를 마우스 오른쪽 단추로 클릭하고 **게시**를 선택합니다. **게시** 대화 상자가 표시됩니다. **대상**에 대해 **Azure**를 선택한 다음, **다음**을 선택합니다.



특정 대상에 대해 **Azure Container Registry**를 선택한 다음, **다음**을 선택합니다.



다음으로 **Container Registry**의 경우 ACR 리소스를 만드는 데 사용한 **구독 이름을** 선택합니다. **컨테이너 레지스트리** 선택 영역에서 만든 컨테이너 레지스트리를 선택한 다음 **마침**을 선택합니다.



그러면 컨테이너 레지스트리에 이미지를 게시하는 데 사용할 수 있는 게시 프로필이 만들어집니다. **게시** 단추를 선택하여 이미지를 컨테이너 레지스트리로 푸시하고 **출력 창에서** 게시 진행률을 보고합니다. 성공적으로 완료되면 "성공적으로 게시됨" 메시지가 표시됩니다.

이미지가 컨테이너 레지스트리에 성공적으로 푸시되었는지 확인하려면 Azure Portal로 이동합니다. 컨테이너 레지스트리 리소스를 열고 **서비스에서리포지토리**를 선택합니다. 이미지가 표시됩니다.

컨테이너 인스턴스로 배포

컨테이너 인스턴스를 만들려면 Azure Portal에서도 새 리소스를 만들어야 합니다.

1. 이전 섹션에서 동일한 **구독** 및 해당 **리소스 그룹을** 선택합니다.
2. **컨테이너 이름을** `appcloudservice-container` 입력합니다.
3. 이전 **위치** 선택 영역에 해당하는 **지역을** 선택합니다.
4. **이미지 원본**의 경우 **Azure Container Registry를** 선택합니다.
5. 이전 단계에서 제공된 이름으로 **레지스트리** 를 선택합니다.
6. **이미지** 및 **이미지 태그**를 선택합니다.
7. **Review + create**를 선택합니다.
8. **유효성 검사가 통과되었다고 가정하고만들기**를 선택합니다.

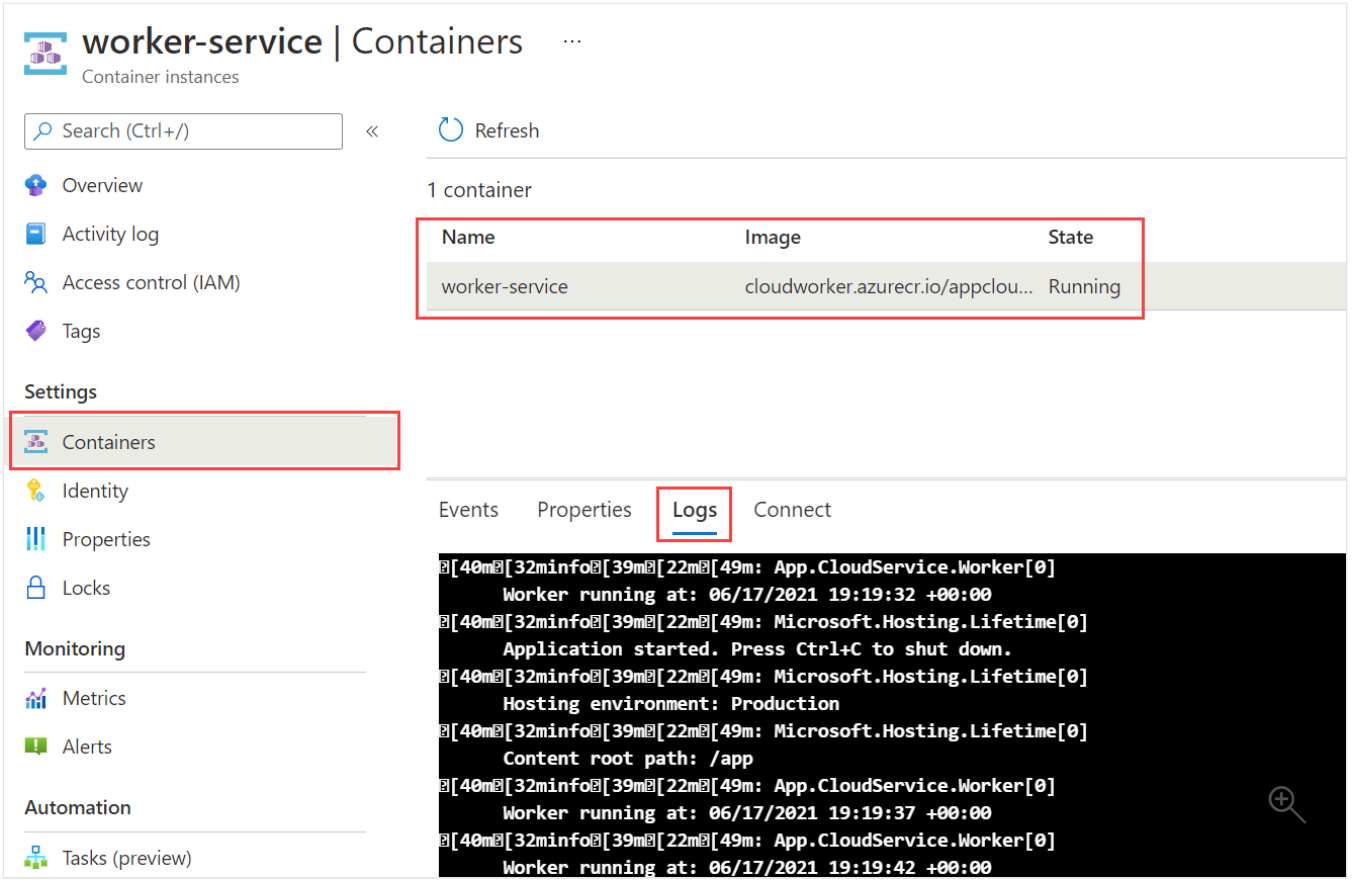
리소스가 만들어지면 리소스로 **이동** 단추를 선택하면 잠시 시간이 걸릴 수 있습니다.

자세한 내용은 빠른 시작: Azure 컨테이너 인스턴스 만들기를 참조하세요.

서비스 기능 확인

컨테이너 인스턴스를 만든 직후 실행이 시작됩니다.

작업자 서비스가 제대로 작동하는지 확인하려면 컨테이너 인스턴스 리소스에서 Azure Portal로 이동하여 **컨테이너** 옵션을 선택합니다.



컨테이너와 현재 **상태가** 표시됩니다. 이 경우 **실행** 중입니다. **로그를** 선택하여 .NET 작업자 서비스 출력을 확인합니다.

참고하십시오

- .NET의 Worker Services
- 범위가 지정된 서비스 사용 BackgroundService
- 를 사용하여 Windows 서비스 만들기 BackgroundService
- IHostedService 인터페이스 구현
- 자습서: .NET Core 앱 컨테이너화

.NET에서 캐싱

이 문서에서는 다양한 캐싱 메커니즘에 대해 알아봅니다. 캐싱은 중간 계층에 데이터를 저장하여 후속 데이터 검색을 더 빠르게 만드는 작업입니다. 개념적으로 캐싱은 성능 최적화 전략 및 디자인 고려 사항입니다. 캐싱은 자주 변경되지 않거나 검색 비용이 많이 드는 데이터를 더 쉽게 사용할 수 있게 함으로써 앱 성능을 크게 향상시킬 수 있습니다. 이 문서에서는 세 가지 캐싱 방법을 소개하고 각각에 대한 샘플 소스 코드를 제공합니다.

- Microsoft.Extensions.Caching.Memory: 단일 서버 시나리오에 대한 메모리 내 캐싱
- Microsoft.Extensions.Caching.Hybrid: 메모리 내 및 분산 캐싱을 추가 기능과 결합하는 하이브리드 캐싱
- Microsoft.Extensions.Caching.Distributed: 다중 서버 시나리오에 대한 분산 캐싱

> **Important**
>
> .NET 내에는 두 개의 `MemoryCache` 클래스가 `System.Runtime.Caching` 네임스페이스와 `Microsoft.Extensions.Caching` 네임스페이스에 각각 하나씩 있습니다.
>
> - System.Runtime.Caching.MemoryCache
> - Microsoft.Extensions.Caching.Memory.MemoryCache
>
> 이 문서에서는 캐싱에 중점을 두지만 NuGet 패키지는 System.Runtime.Caching 포함하지 않습니다. 모든 `MemoryCache` 참조는 `Microsoft.Extensions.Caching` 네임스페이스 내에 있습니다.

모든 `Microsoft.Extensions.*` 패키지는 DI(종속성 주입)를 지원합니다. IMemoryCache, HybridCache및 IDistributedCache 인터페이스를 서비스로 사용할 수 있습니다.

메모리 내 캐싱

이 섹션에서는 Microsoft.Extensions.Caching.Memory 패키지에 대해 알아봅니다. 현재 구현된 IMemoryCache는 ConcurrentDictionary<TKey,TValue> 주위에 래퍼를 구현한 것으로, 기능이 풍부한 API를 제공합니다. 캐시 내의 항목은 ICacheEntry으로 표시되며 `object`에 해당하는 어떤 항목도 될 수 있습니다. 메모리 내 캐시 솔루션은 캐시된 데이터가 앱 프로세스에서 메모리를 임대하는 단일 서버에서 실행되는 앱에 적합합니다.

> **팁**
>
> 다중 서버 캐싱 시나리오의 경우 **메모리 내 캐싱 대신 분산 캐싱** 방법을 고려합니다.

메모리 내 캐싱 API

캐시의 소비자는 슬라이딩 및 절대 만료를 모두 제어할 수 있습니다.

- ICacheEntry.AbsoluteExpiration
- ICacheEntry.AbsoluteExpirationRelativeToNow
- ICacheEntry.SlidingExpiration

만료를 설정하면 만료 시간 할당 내에 액세스하지 않으면 캐시의 항목이 *제거* 됩니다. 소비자는 MemoryCacheEntryOptions를 통해 캐시 항목을 제어할 수 있는 추가 옵션이 있습니다. 각각의 ICacheEntry는 MemoryCacheEntryOptions에 대한 만료 회피 기능을 제공하고, 우선 순위 설정은 IChangeToken과 CacheItemPriority의 제어를 포함합니다. 관련 확장 메서드는 다음과 같습니다.

- MemoryCacheEntryExtensions.AddExpirationToken
- MemoryCacheEntryExtensions.RegisterPostEvictionCallback
- MemoryCacheEntryExtensions.SetSize
- MemoryCacheEntryExtensions.SetPriority

메모리 내 캐시 예제

기본 IMemoryCache 구현을 사용하려면 확장 메서드를 AddMemoryCache 호출하여 필요한 모든 서비스를 DI에 등록합니다. 다음 코드 샘플에서 제네릭 호스트는 DI 기능을 노출하는 데 사용됩니다.

C#

```
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddMemoryCache();
using IHost host = builder.Build();
```

.NET 워크로드에 따라 `IMemoryCache`에 접근하는 방법이 다를 수 있으며, 그 예로 생성자 주입이 있습니다. 이 샘플에서는 `IServiceProvider`에서 `host` 인스턴스를 사용하고 제네릭 GetRequiredService<T>(IServiceProvider) 확장 메서드를 호출합니다.

C#

```
IMemoryCache cache =
    host.Services.GetRequiredService<IMemoryCache>();
```

메모리 내 캐싱 서비스를 등록하고 DI를 통해 확인하면 캐싱을 시작할 준비가 된 것입니다. 이 샘플은 영어 알파벳의 'A'부터 'Z'까지의 각 문자를 순차적으로 반복합니다. 이 형식은 `record AlphabetLetter` 문자에 대한 참조를 유지하고 메시지를 생성합니다.

C#

```
file record AlphabetLetter(char Letter)
{
    internal string Message =>
        $"The '{Letter}' character is the {Letter - 64} letter in the English
alphabet.";
}
```

> **팁**
>
> `file` 파일 내에서 정의되고 액세스되기 때문에 `AlphabetLetter` 형식에 액세스 한정자가 사용됩니다. 자세한 내용은 **파일(C# 참조)을 참조**하세요. 전체 소스 코드를 보려면 Program.cs 섹션을 참조하세요.

샘플에는 알파벳 문자를 반복하는 도우미 함수가 포함되어 있습니다.

C#

```
static async ValueTask IterateAlphabetAsync(
    Func<char, Task> asyncFunc)
{
    for (char letter = 'A'; letter <= 'Z'; ++letter)
    {
        await asyncFunc(letter);
    }

    Console.WriteLine();
}
```

위의 C# 코드에서:

- 각 `Func<char, Task> asyncFunc` 반복에서 대기하여 현재 `letter`를 전달합니다.
- 모든 문자가 처리되면 콘솔에 빈 줄이 기록됩니다.

캐시에 항목을 추가하려면 다음 중 하나 또는 `Create` API를 `Set`호출합니다.

C#

```
var addLettersToCacheTask = IterateAlphabetAsync(letter =>
{
    MemoryCacheEntryOptions options = new()
```

```
    {
        AbsoluteExpirationRelativeToNow =
            TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
    };

    _ = options.RegisterPostEvictionCallback(OnPostEviction);

    AlphabetLetter alphabetLetter =
        cache.Set(
            letter, new AlphabetLetter(letter), options);

    Console.WriteLine($"{alphabetLetter.Letter} was cached.");

    return Task.Delay(
        TimeSpan.FromMilliseconds(MillisecondsDelayAfterAdd));
});
await addLettersToCacheTask;
```

위의 C# 코드에서:

- 변수 `addLettersToCacheTask`가 `IterateAlphabetAsync`에게 위임되고 대기됩니다.
- `Func<char, Task> asyncFunc` 람다로 주장된다.
- 현재 시점을 기준으로 `MemoryCacheEntryOptions`가 절대 만료로 인스턴스화됩니다.
- 퇴거 후 콜백이 등록됩니다.
- `AlphabetLetter` 개체가 인스턴스화된 후, Set에 `letter` 및 `options`와 함께 전달됩니다.
- 문자는 캐시되는 것으로 콘솔에 기록됩니다.
- 마지막으로 하나가 Task.Delay 반환됩니다.

알파벳의 각 문자에 대해 캐시 항목은 만료 및 제거 후 콜백으로 작성됩니다.

제거 후 콜백은 콘솔에 제거된 값의 세부 정보를 기록합니다.

C#

```
static void OnPostEviction(
    object key, object? letter, EvictionReason reason, object? state)
{
    if (letter is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{alphabetLetter.Letter} was evicted for {reason}.");
    }
};
```

이제 캐시가 채워졌으므로 `IterateAlphabetAsync`에 대한 다른 호출이 대기됩니다. 그러나 이번에는 IMemoryCache.TryGetValue를 호출합니다.

C#

```
var readLettersFromCacheTask = IterateAlphabetAsync(letter =>
{
    if (cache.TryGetValue(letter, out object? value) &&
        value is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{letter} is still in cache. {alphabetLetter.Message}");
    }

    return Task.CompletedTask;
});
await readLettersFromCacheTask;
```

`cache`가 `letter` 키를 포함하고 있으며, 만약 `value`가 `AlphabetLetter`의 인스턴스일 경우, 콘솔에 기록됩니다. `letter` 키가 캐시에 없으면 제거되고 제거 후 콜백이 호출되었습니다.

추가 확장 메서드

`IMemoryCache`와 `GetOrCreateAsync`를 비롯한 많은 편리한 확장 메서드가 비동기로 함께 제공됩니다.

- CacheExtensions.Get
- CacheExtensions.GetOrCreate
- CacheExtensions.GetOrCreateAsync
- CacheExtensions.Set
- CacheExtensions.TryGetValue

전부 합치세요

전체 샘플 앱 소스 코드는 최상위 프로그램이며 두 개의 NuGet 패키지가 필요합니다.

- Microsoft.Extensions.Caching.Memory
- Microsoft.Extensions.Hosting

C#

```
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddMemoryCache();
using IHost host = builder.Build();

IMemoryCache cache =
    host.Services.GetRequiredService<IMemoryCache>();

const int MillisecondsDelayAfterAdd = 50;
```

```
const int MillisecondsAbsoluteExpiration = 750;

static void OnPostEviction(
    object key, object? letter, EvictionReason reason, object? state)
{
    if (letter is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{alphabetLetter.Letter} was evicted for {reason}.");
    }
};

static async ValueTask IterateAlphabetAsync(
    Func<char, Task> asyncFunc)
{
    for (char letter = 'A'; letter <= 'Z'; ++letter)
    {
        await asyncFunc(letter);
    }

    Console.WriteLine();
}

var addLettersToCacheTask = IterateAlphabetAsync(letter =>
{
    MemoryCacheEntryOptions options = new()
    {
        AbsoluteExpirationRelativeToNow =
            TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
    };

    _ = options.RegisterPostEvictionCallback(OnPostEviction);

    AlphabetLetter alphabetLetter =
        cache.Set(
            letter, new AlphabetLetter(letter), options);

    Console.WriteLine($"{alphabetLetter.Letter} was cached.");

    return Task.Delay(
        TimeSpan.FromMilliseconds(MillisecondsDelayAfterAdd));
});
await addLettersToCacheTask;

var readLettersFromCacheTask = IterateAlphabetAsync(letter =>
{
    if (cache.TryGetValue(letter, out object? value) &&
        value is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{letter} is still in cache. {alphabetLetter.Message}");
    }

    return Task.CompletedTask;
});
await readLettersFromCacheTask;
```

```
await host.RunAsync();

file record AlphabetLetter(char Letter)
{
    internal string Message =>
        $"The '{Letter}' character is the {Letter - 64} letter in the English
alphabet.";
}
```

`MillisecondsDelayAfterAdd` 값과 `MillisecondsAbsoluteExpiration` 값을 조정하여 캐시된 항목의 만료 및 제거와 관련된 동작 변화를 관찰할 수 있습니다. 다음은 이 코드를 실행하는 샘플 출력입니다. (.NET 이벤트의 비결정적 특성으로 인해 출력이 다를 수 있습니다.)

콘솔

```
A was cached.
B was cached.
C was cached.
D was cached.
E was cached.
F was cached.
G was cached.
H was cached.
I was cached.
J was cached.
K was cached.
L was cached.
M was cached.
N was cached.
O was cached.
P was cached.
Q was cached.
R was cached.
S was cached.
T was cached.
U was cached.
V was cached.
W was cached.
X was cached.
Y was cached.
Z was cached.

A was evicted for Expired.
C was evicted for Expired.
B was evicted for Expired.
E was evicted for Expired.
D was evicted for Expired.
F was evicted for Expired.
H was evicted for Expired.
K was evicted for Expired.
L was evicted for Expired.
J was evicted for Expired.
G was evicted for Expired.
```

```
M was evicted for Expired.
N was evicted for Expired.
I was evicted for Expired.
P was evicted for Expired.
R was evicted for Expired.
O was evicted for Expired.
Q was evicted for Expired.
S is still in cache. The 'S' character is the 19 letter in the English alphabet.
T is still in cache. The 'T' character is the 20 letter in the English alphabet.
U is still in cache. The 'U' character is the 21 letter in the English alphabet.
V is still in cache. The 'V' character is the 22 letter in the English alphabet.
W is still in cache. The 'W' character is the 23 letter in the English alphabet.
X is still in cache. The 'X' character is the 24 letter in the English alphabet.
Y is still in cache. The 'Y' character is the 25 letter in the English alphabet.
Z is still in cache. The 'Z' character is the 26 letter in the English alphabet.
```

절대 만료(MemoryCacheEntryOptions.AbsoluteExpirationRelativeToNow)가 설정되었으므로 캐시된 모든 항목은 결국 제거됩니다.

작업 서비스 캐싱

데이터를 캐싱하기 위한 일반적인 전략 중 하나는 캐시를 소비하는 데이터 서비스와 독립적으로 업데이트하는 것입니다. *작업자 서비스* 템플릿은 다른 애플리케이션 코드에서 독립적으로(또는 백그라운드에서) 실행되므로 BackgroundService 좋은 예입니다. 애플리케이션이 구현 IHostedService을 호스트하는 실행을 시작하면 해당 구현(이 경우 `BackgroundService` "작업자")이 동일한 프로세스에서 실행되기 시작합니다. 이러한 호스티드 서비스는 확장 메서드를 통해 AddHostedService<THostedService>(IServiceCollection) DI에 싱글톤으로 등록됩니다. 다른 서비스는 모든 서비스 수명 동안 DI에 등록할 수 있습니다.

> ⓘ **Important**
>
> 서비스 수명 주기를 파악하는 것이 중요합니다. 모든 메모리 내 캐싱 서비스를 등록하기 위해 호출 AddMemoryCache 하면 서비스가 싱글톤으로 등록됩니다.

사진 서비스 시나리오

HTTP를 통해 액세스할 수 있는 타사 API를 사용하는 사진 서비스를 개발하고 있다고 상상해 보십시오. 이 사진 데이터는 자주 변경되지 않지만 많은 데이터가 있습니다. 각 사진은 간단한 `record`다음으로 표시됩니다.

C#

```
namespace CachingExamples.Memory;
```

```
public readonly record struct Photo(
    int AlbumId,
    int Id,
    string Title,
    string Url,
    string ThumbnailUrl);
```

다음 예제에서는 여러 서비스가 DI에 등록되는 것을 볼 수 있습니다. 각 서비스에는 단일 책임이 있습니다.

C#

```
using CachingExamples.Memory;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMemoryCache();
builder.Services.AddHttpClient<CacheWorker>();
builder.Services.AddHostedService<CacheWorker>();
builder.Services.AddScoped<PhotoService>();
builder.Services.AddSingleton(typeof(CacheSignal<>));

using IHost host = builder.Build();

await host.StartAsync();
```

위의 C# 코드에서:

- 제네릭 호스트는 기본값으로 만들어집니다.
- 메모리 내 캐싱 서비스는 AddMemoryCache에 등록됩니다.
- `HttpClient` 인스턴스가 `CacheWorker`를 사용하여 AddHttpClient<TClient>(IServiceCollection) 클래스에 등록됩니다.
- `CacheWorker` 클래스가 AddHostedService<THostedService>(IServiceCollection)에 등록되어 있습니다.
- `PhotoService` 클래스가 AddScoped<TService>(IServiceCollection)에 등록되어 있습니다.
- `CacheSignal<T>` 클래스가 AddSingleton에 등록되어 있습니다.
- 빌더 `host`에서 인스턴스화되어 비동기적으로 시작됩니다.

지정된 `PhotoService` 조건(또는 `filter`)에 맞는 사진을 가져오는 책임이 있습니다.

C#

```
using Microsoft.Extensions.Caching.Memory;

namespace CachingExamples.Memory;

public sealed class PhotoService(
        IMemoryCache cache,
```

```
        CacheSignal<Photo> cacheSignal,
        ILogger<PhotoService> logger)
{
    public async IAsyncEnumerable<Photo> GetPhotosAsync(Func<Photo, bool>? filter =
default)
    {
        try
        {
            await cacheSignal.WaitAsync();

            Photo[] photos =
                (await cache.GetOrCreateAsync(
                    "Photos", _ =>
                    {
                        logger.LogWarning("This should never happen!");

                        return Task.FromResult(Array.Empty<Photo>());
                    }))!;

            // If no filter is provided, use a pass-thru.
            filter ??= _ => true;

            foreach (Photo photo in photos)
            {
                if (!default(Photo).Equals(photo) && filter(photo))
                {
                    yield return photo;
                }
            }
        }
        finally
        {
            cacheSignal.Release();
        }
    }
}
```

위의 C# 코드에서:

- 생성자에는 `IMemoryCache`, `CacheSignal<Photo>`, 및 `ILogger`.
- 메서드: `GetPhotosAsync`
  - `Func<Photo, bool> filter` 매개 변수를 정의하고 `IAsyncEnumerable<Photo>`을 반환합니다.
  - `_cacheSignal.WaitAsync()`를 호출하고 해제될 때까지 기다립니다. 이렇게 하면 캐시에 액세스하기 전에 캐시가 채워지게 합니다.
  - 캐시에 있는 모든 사진을 비동기적으로 가져오는 호출 `_cache.GetOrCreateAsync()`입니다.
  - 인수는 `factory` 경고를 기록하고 빈 사진 배열을 반환합니다. 이렇게 하면 안 됩니다.
  - 캐시에 있는 각 사진은 `yield return`을 사용하여 반복되고, 필터링되며, 구체화됩니다.
  - 마지막으로 캐시 신호가 다시 설정됩니다.

이 서비스의 소비자는 자유롭게 메서드를 호출 `GetPhotosAsync` 하고 그에 따라 사진을 처리할 수 있습니다. 필요한 `HttpClient` 것은 없습니까? 캐시에 사진이 포함되어 있습니다.

비동기 신호는 제네릭 형식 제한 싱글톤 내의 캡슐화된 SemaphoreSlim 인스턴스를 기반으로 합니다. `CacheSignal<T>`은/는 `SemaphoreSlim` 인스턴스에 의존합니다.

C#

```
namespace CachingExamples.Memory;

public sealed class CacheSignal<T>
{
    private readonly SemaphoreSlim _semaphore = new(1, 1);

    /// <summary>
    /// Exposes a <see cref="Task"/> that represents the asynchronous wait
operation.
    /// When signaled (consumer calls <see cref="Release"/>), the
    /// <see cref="Task.Status"/> is set as <see
cref="TaskStatus.RanToCompletion"/>.
    /// </summary>
    public Task WaitAsync() => _semaphore.WaitAsync();

    /// <summary>
    /// Exposes the ability to signal the release of the <see cref="WaitAsync"/>'s
operation.
    /// Callers who were waiting, will be able to continue.
    /// </summary>
    public void Release() => _semaphore.Release();
}
```

위의 C# 코드에서 데코레이터 패턴은 인스턴스 `SemaphoreSlim`를 래핑하는 데 사용됩니다. `CacheSignal<T>`이 싱글톤으로 등록되어 있으므로, 모든 제네릭 형식의 서비스 수명 전반에 걸쳐 사용할 수 있습니다. 이 경우에는 `Photo`. 캐시 시딩을 알리는 역할을 담당합니다.

의 `CacheWorker` 하위 클래스입니다.BackgroundService

C#

```
using System.Net.Http.Json;
using Microsoft.Extensions.Caching.Memory;

namespace CachingExamples.Memory;

public sealed class CacheWorker(
    ILogger<CacheWorker> logger,
    HttpClient httpClient,
    CacheSignal<Photo> cacheSignal,
    IMemoryCache cache) : BackgroundService
{
```

```
    private readonly TimeSpan _updateInterval = TimeSpan.FromHours(3);

    private bool _isCacheInitialized = false;

    private const string Url = "https://jsonplaceholder.typicode.com/photos";

    public override async Task StartAsync(CancellationToken cancellationToken)
    {
        await cacheSignal.WaitAsync();
        await base.StartAsync(cancellationToken);
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            logger.LogInformation("Updating cache.");

            try
            {
                Photo[]? photos =
                    await httpClient.GetFromJsonAsync<Photo[]>(
                        Url, stoppingToken);

                if (photos is { Length: > 0 })
                {
                    cache.Set("Photos", photos);
                    logger.LogInformation(
                        "Cache updated with {Count:#,#} photos.", photos.Length);
                }
                else
                {
                    logger.LogWarning(
                        "Unable to fetch photos to update cache.");
                }
            }
            finally
            {
                if (!_isCacheInitialized)
                {
                    cacheSignal.Release();
                    _isCacheInitialized = true;
                }
            }

            try
            {
                logger.LogInformation(
                    "Will attempt to update the cache in {Hours} hours from now.",
                    _updateInterval.Hours);

                await Task.Delay(_updateInterval, stoppingToken);
            }
            catch (OperationCanceledException)
            {
```

```
                logger.LogWarning("Cancellation acknowledged: shutting down.");
                break;
            }
        }
    }
}
```

위의 C# 코드에서:

- 생성자에는 `ILogger`, `HttpClient`, 및 `IMemoryCache`.
- `_updateInterval` 3시간 동안 정의됩니다.
- 메서드: `ExecuteAsync`
  - 앱이 실행되는 동안 반복합니다.
  - HTTP 요청을 수행하고 `"https://jsonplaceholder.typicode.com/photos"`응답을 개체 배열 `Photo` 로 매핑합니다.
  - 사진 배열은 `IMemoryCache` 내에서 `"Photos"` 키 아래에 배치됩니다.
  - `_cacheSignal.Release()`가 호출되면 신호를 기다리고 있던 모든 소비자가 해제됩니다.
  - 업데이트 간격이 Task.Delay 지정된 경우 호출이 대기됩니다.
  - 3시간 동안 지연된 후 캐시가 다시 업데이트됩니다.

동일한 프로세스의 소비자는 `IMemoryCache`에게 사진을 요청할 수 있지만, `CacheWorker`는 캐시를 업데이트할 책임이 있습니다.

하이브리드 캐싱

라이브러리는 HybridCache 기존 캐싱 API와 일반적인 문제를 해결하는 동시에 메모리 내 및 분산 캐싱의 이점을 결합합니다. .NET 9 `HybridCache` 에서 도입된 이 API는 캐싱 구현을 간소화하고 스탬피드 보호 및 구성 가능한 serialization과 같은 기본 제공 기능을 포함하는 통합 API를 제공합니다.

주요 기능

`HybridCache`은 `IMemoryCache`와 `IDistributedCache`를 따로 사용하는 것에 비해 몇 가지 이점을 제공합니다.

- **2단계 캐싱**: 메모리 내(L1) 및 분산(L2) 캐시 계층을 모두 자동으로 관리합니다. 데이터는 먼저 메모리 내 캐시에서 빠른 속도로 검색된 다음, 필요한 경우 분산 캐시에서, 마지막으로 원본에서 검색됩니다.
- **스탬피드 보호**: 여러 동시 요청이 동일한 비용이 드는 작업을 실행하지 못하도록 방지합니다. 하나의 요청만 데이터를 가져오는 반면 다른 요청은 결과를 기다립니다.
- **구성 가능한 serialization**: JSON(기본값), protobuf 및 XML을 비롯한 여러 serialization 형식을 지원합니다.

- **태그 기반 무효화**: 효율적인 일괄 처리 무효화를 위해 태그를 사용하여 관련 캐시 항목을 그룹화합니다.
- **간소화된 API**: 메서드는 `GetOrCreateAsync` 캐시 누락, serialization 및 스토리지를 자동으로 처리합니다.

HybridCache를 사용하는 경우

다음 경우에 사용하는 것이 `HybridCache` 좋습니다.

- 다중 서버 환경에서 로컬(메모리 내) 및 분산 캐싱이 모두 필요합니다.
- 캐시 스탬피드 시나리오로부터 보호받고 싶습니다.
- 수동으로 `IMemoryCache` 및 `IDistributedCache`를 조정하는 것보다 간소화된 API를 선호합니다.
- 관련 항목에 대한 태그 기반 캐시 무효화가 필요합니다.

> **팁**
>
> 간단한 캐싱 요구 사항이 있는 단일 서버 애플리케이션의 경우 **메모리 내 캐싱** 으로 충분할 수 있습니다. 스탬피드 보호 또는 태그 기반 무효화가 필요 없는 다중 서버 애플리케이션의 경우 **분산 캐싱을** 고려합니다.

HybridCache 설정

`HybridCache`를 사용하려면 Microsoft.Extensions.Caching.Hybrid NuGet 패키지를 설치하세요.

.NET CLI

```
dotnet add package Microsoft.Extensions.Caching.Hybrid
```

DI에 서비스를 등록하려면 `HybridCache`를 호출하여 AddHybridCache를 사용하세요.

C#

```
var builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHybridCache();
```

위의 코드는 기본 옵션으로 등록됩니다 `HybridCache` . 전역 옵션을 구성할 수도 있습니다.

C#

```
var builderWithOptions = Host.CreateApplicationBuilder(args);
builderWithOptions.Services.AddHybridCache(options =>
```

```
{
    options.MaximumPayloadBytes = 1024 * 1024; // 1 MB
    options.MaximumKeyLength = 1024;
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromMinutes(5),
        LocalCacheExpiration = TimeSpan.FromMinutes(2)
    };
});
```

기본 사용법

ko-KR: `HybridCache`와 상호 작용하는 기본 방법은 GetOrCreateAsync입니다. 이 메서드는 지정된 키를 가진 항목에 대한 캐시를 확인하고, 찾을 수 없는 경우 팩터리 메서드를 호출하여 데이터를 검색합니다.

C#

```
async Task<WeatherData> GetWeatherDataAsync(HybridCache cache, string city)
{
    return await cache.GetOrCreateAsync(
        $"weather:{city}",
        async cancellationToken =>
        {
            // Simulate fetching from an external API
            await Task.Delay(100, cancellationToken);
            return new WeatherData(city, 72, "Sunny");
        }
    );
}
```

위의 C# 코드에서:

- 이 메서드는 `GetOrCreateAsync` 고유 키와 팩터리 메서드를 사용합니다.
- 데이터가 캐시에 없으면 팩터리 메서드를 호출하여 검색합니다.
- 데이터는 메모리 내 및 분산 캐시 모두에 자동으로 저장됩니다.
- 하나의 동시 요청만 팩터리 메서드를 실행합니다. 다른 사용자는 결과를 기다립니다.

입력 옵션

다음을 사용하여 HybridCacheEntryOptions특정 캐시 항목에 대한 전역 기본값을 재정의할 수 있습니다.

C#

```
async Task<WeatherData> GetWeatherWithOptionsAsync(HybridCache cache, string city)
{
```

```
    var entryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromMinutes(10),
        LocalCacheExpiration = TimeSpan.FromMinutes(5)
    };

    return await cache.GetOrCreateAsync(
        $"weather:{city}",
        async cancellationToken => new WeatherData(city, 72, "Sunny"),
        entryOptions
    );
}
```

항목 옵션을 사용하면 다음을 구성할 수 있습니다.

- HybridCacheEntryOptions.Expiration: 분산 캐시에 항목을 캐시해야 하는 기간입니다.
- HybridCacheEntryOptions.LocalCacheExpiration: 로컬 메모리에 항목을 캐시해야 하는 기간입니다.
- HybridCacheEntryOptions.Flags: 캐시 동작을 제어하기 위한 추가 플래그입니다.

태그 기반 무효화

태그를 사용하면 관련 캐시 항목을 그룹화하고 함께 무효화할 수 있습니다. 이는 관련 데이터를 한 단위로 새로 고쳐야 하는 시나리오에 유용합니다.

C#

```
async Task<CustomerData> GetCustomerAsync(HybridCache cache, int customerId)
{
    var tags = new[] { "customer", $"customer:{customerId}" };

    return await cache.GetOrCreateAsync(
        $"customer:{customerId}",
        async cancellationToken => new CustomerData(customerId, "John Doe",
"john@example.com"),
        new HybridCacheEntryOptions { Expiration = TimeSpan.FromMinutes(30) },
        tags
    );
}
```

특정 태그를 사용하여 모든 항목을 무효화하려면 다음을 수행합니다.

C#

```
async Task InvalidateCustomerCacheAsync(HybridCache cache, int customerId)
{
    await cache.RemoveByTagAsync($"customer:{customerId}");
}
```

여러 태그를 한 번에 무효화할 수도 있습니다.

C#

```
async Task InvalidateAllCustomersAsync(HybridCache cache)
{
    await cache.RemoveByTagAsync(new[] { "customer", "orders" });
}
```

> ⓘ **참고 항목**
>
> 태그 기반 무효화는 논리적 작업입니다. 캐시에서 값을 적극적으로 제거하지는 않지만 태그가 지정된 항목이 캐시 누락으로 처리되도록 합니다. 결국 항목은 구성된 수명에 따라 만료됩니다.

캐시 항목 제거

키별로 특정 캐시 항목을 제거하려면 다음 메서드를 RemoveAsync 사용합니다.

C#

```
async Task RemoveWeatherDataAsync(HybridCache cache, string city)
{
    await cache.RemoveAsync($"weather:{city}");
}
```

캐시된 모든 항목을 무효화하려면 예약된 와일드카드 태그 `"*"`를 사용합니다.

C#

```
async Task InvalidateAllCacheAsync(HybridCache cache)
{
    await cache.RemoveByTagAsync("*");
}
```

직렬화

분산 캐싱 시나리오의 `HybridCache` 경우 serialization이 필요합니다. 기본적으로 `string`와 `byte[]`를 내부적으로 처리하고, 다른 형식에 대해서는 `System.Text.Json`를 사용합니다. 특정 형식에 대해 사용자 지정 직렬 변환기를 구성하거나 범용 직렬 변환기를 사용할 수 있습니다.

C#

```
// Custom serialization example
// Note: This requires implementing a custom IHybridCacheSerializer<T>
var builderWithSerializer = Host.CreateApplicationBuilder(args);
builderWithSerializer.Services.AddHybridCache(options =>
{
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromMinutes(10),
        LocalCacheExpiration = TimeSpan.FromMinutes(5)
    };
});
// To add a custom serializer, uncomment and provide your implementation:
// .AddSerializer<WeatherData, CustomWeatherDataSerializer>();
```

분산 캐시 구성

`HybridCache` 는 L2(분산) 캐시에 대해 구성된 IDistributedCache 구현을 사용합니다. 구성되지 않은 `IDistributedCache`에도 `HybridCache`은 여전히 메모리 내 캐싱 및 스탬피드 방지를 제공합니다. Redis를 분산 캐시로 추가하려면 다음을 수행합니다.

C#

```
// Distributed cache with Redis
var builderWithRedis = Host.CreateApplicationBuilder(args);
builderWithRedis.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost:6379";
});
builderWithRedis.Services.AddHybridCache(options =>
{
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromMinutes(30),
        LocalCacheExpiration = TimeSpan.FromMinutes(5)
    };
});
```

분산 캐시 구현에 대한 자세한 내용은 분산 캐싱을 참조하세요.

분산 캐싱

일부 시나리오에서는 분산 캐시가 필요합니다. 예를 들어 여러 앱 서버가 있는 경우입니다. 분산 캐시는 메모리 내 캐싱 방법보다 더 높은 스케일 아웃을 지원합니다. 분산 캐시를 사용하면 캐시 메모리가 외부 프로세스로 오프로드되지만 추가 네트워크 I/O가 필요하고 약간의 대기 시간이 발생합니다(명목상인 경우에도).

분산 캐싱 추상화는 NuGet 패키지의 Microsoft.Extensions.Caching.Memory 일부이며 확장 메서드도 `AddDistributedMemoryCache` 있습니다.

> ⊗ **주의**
>
> AddDistributedMemoryCache 는 개발 또는 테스트 시나리오에서만 사용해야 하며 실행 가능한 프로덕션 구현이 **아닙니다** .

다음 패키지에서 사용할 수 있는 `IDistributedCache`의 어떤 구현이든 고려해 보세요.

- Microsoft.Extensions.Caching.SqlServer
- Microsoft.Extensions.Caching.StackExchangeRedis
- NCache.Microsoft.Extensions.Caching.OpenSource

분산 캐싱 API

분산 캐싱 API는 메모리 내 캐싱 API보다 약간 더 기본적입니다. 키-값 쌍은 좀 더 기본적입니다. 메모리 내 캐싱 키는 `object`을 기반으로 하며, 반면에 분산 키는 `string`입니다. 메모리 내 캐싱을 사용하면 값이 강력한 형식의 제네릭일 수 있지만 분산 캐싱의 값은 다음과 같이 `byte[]`유지됩니다. 즉, 다양한 구현이 강력한 형식의 제네릭 값을 노출하지는 않지만 구현 세부 정보입니다.

값 만들기

분산 캐시에 값을 만들려면 집합 API 중 하나를 호출합니다.

- IDistributedCache.SetAsync
- IDistributedCache.Set

`AlphabetLetter` 메모리 내 캐시 예제의 레코드를 사용하여 객체를 JSON으로 직렬화한 다음 `string`를 `byte[]`로 인코딩할 수 있습니다.

C#

```
DistributedCacheEntryOptions options = new()
{
    AbsoluteExpirationRelativeToNow =
        TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
};

AlphabetLetter alphabetLetter = new(letter);
string json = JsonSerializer.Serialize(alphabetLetter);
byte[] bytes = Encoding.UTF8.GetBytes(json);

await cache.SetAsync(letter.ToString(), bytes, options);
```

메모리 내 캐싱과 마찬가지로 캐시 항목에는 캐시의 존재를 미세 조정하는 데 도움이 되는 옵션이 있을 수 있습니다. 이 경우 DistributedCacheEntryOptions.

확장 메서드 만들기

값을 만들기 위한 몇 가지 편의 기반 확장 메서드가 있습니다. 개체의 표현을 `string`로 인코딩하지 않도록 하는 데 이러한 메서드가 `byte[]`의 도움이 됩니다.

- DistributedCacheExtensions.SetStringAsync
- DistributedCacheExtensions.SetString

값 읽기

분산 캐시에서 값을 읽으려면 API 중 `Get` 하나를 호출합니다.

- IDistributedCache.GetAsync
- IDistributedCache.Get

C#

```
AlphabetLetter? alphabetLetter = null;
byte[]? bytes = await cache.GetAsync(letter.ToString());
if (bytes is { Length: > 0 })
{
    string json = Encoding.UTF8.GetString(bytes);
    alphabetLetter = JsonSerializer.Deserialize<AlphabetLetter>(json);
}
```

캐시 항목을 캐시에서 읽은 후에는 `string`로부터 UTF8로 인코딩된 `byte[]` 표현을 가져올 수 있습니다.

확장 메서드 읽기

값을 읽기 위한 몇 가지 편의 기반 확장 메서드가 있습니다. 이러한 메서드는 `byte[]`가 `string` 객체 표현으로 디코딩되지 않도록 하는 데 도움이 됩니다.

- DistributedCacheExtensions.GetStringAsync
- DistributedCacheExtensions.GetString

값 업데이트

단일 API 호출을 사용하여 분산 캐시의 값을 업데이트할 수 있는 방법은 없습니다. 대신 값은 새로 고침 API 중 하나를 사용하여 슬라이딩 만료를 재설정할 수 있습니다.

- IDistributedCache.RefreshAsync
- IDistributedCache.Refresh

실제 값을 업데이트해야 하는 경우 값을 삭제한 다음 다시 추가해야 합니다.

값 삭제

분산 캐시의 값을 삭제하려면 API 중 `Remove` 하나를 호출합니다.

- IDistributedCache.RemoveAsync
- IDistributedCache.Remove

> **팁**
>
> 이러한 API의 동기 버전이 있지만 분산 캐시의 구현이 네트워크 I/O에 의존한다는 사실을 고려합니다. 이러한 이유로 일반적으로 비동기 API를 사용하는 것이 좋습니다.

참고하십시오

- .NET에서 종속성 주입
- .NET 제네릭 호스트
- .NET에서의 작업자 서비스
- .NET 개발자용 Azure
- ASP.NET Core 메모리 내 캐시
- ASP.NET Core 분산 캐싱
- ASP.NET Core의 HybridCache 라이브러리

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 13.

System.Threading.Channels 라이브러리

System.Threading.Channels 네임스페이스는 생산자와 소비자 간에 데이터를 비동기적으로 전달하기 위한 일련의 동기화 데이터 구조를 제공합니다. 라이브러리는 .NET, .NET Standard 및 .NET Framework를 대상으로 하며 모든 .NET 구현에서 작동합니다.

이 라이브러리는 System.Threading.Channels NuGet 패키지에서 사용할 수 있습니다. 그러나 .NET Core 3.0 이상을 사용하는 경우 패키지는 공유 프레임워크의 일부로 포함됩니다.

생산자/소비자 개념적 프로그래밍 모델

채널은 생산자/소비자 개념적 프로그래밍 모델을 구현한 것입니다. 이 프로그래밍 모델에서 생산자는 비동기적으로 데이터를 생성하고 소비자는 해당 데이터를 비동기적으로 소비합니다. 즉, 이 모델은 FIFO(선입선출) 큐를 통해 한 당사자에서 다른 당사자로 데이터를 전달합니다. 채널을 다른 일반적인 제네릭 컬렉션 형식(예: .)으로 `List<T>`간주합니다. 가장 큰 차이점은 이 컬렉션이 동기화를 관리하고 팩터리 만들기 옵션을 통해 다양한 사용량 모델을 제공한다는 것입니다. 이러한 옵션은 다음과 같은 채널의 동작을 제어합니다.

- 저장할 수 있는 요소 수와 해당 제한에 도달하면 어떻게 되나요?
- 여러 생산자 또는 여러 소비자가 동시에 채널에 액세스하는지 여부입니다.

기본 사용법

다음 예제에서는 생산자가 항목을 작성하고 소비자가 항목을 읽는 채널의 기본 사용을 보여 줍니다.

C#

```
static async Task BasicUsageAsync()
{
    Channel<int> channel = Channel.CreateUnbounded<int>();

    Task producer = ProduceAsync(channel.Writer);
    Task consumer = ConsumeAsync(channel.Reader);

    await Task.WhenAll(producer, consumer);

    static async Task ProduceAsync(ChannelWriter<int> writer)
    {
        for (int i = 0; i < 5; i++)
        {
            await writer.WriteAsync(i);
```

```
        }

        writer.Complete();
    }

    static async Task ConsumeAsync(ChannelReader<int> reader)
    {
        await foreach (int item in reader.ReadAllAsync())
        {
            Console.WriteLine($"Received: {item}");
        }
    }
}
```

경계 전략

`Channel<T>`가 만들어지는 방식에 따라 읽기 권한자와 기록자가 다르게 동작합니다.

최대 용량을 지정하는 채널을 만들려면 Channel.CreateBounded를 호출합니다. 여러 읽기 권한자와 기록자가 동시에 사용하는 채널을 만들려면 Channel.CreateUnbounded를 호출합니다. 각 경계 전략은 각각 BoundedChannelOptions 또는 UnboundedChannelOptions 등 다양한 작성자 정의 옵션을 노출합니다.

> ⓘ **참고 항목**
>
> 경계 전략에 관계없이 채널이 닫힌 후 사용되면 항상 ChannelClosedException를 던집니다.

무제한 채널

제한되지 않은 채널을 만들려면 Channel.CreateUnbounded 오버로드 중 하나를 호출합니다.

C#

```
var channel = Channel.CreateUnbounded<T>();
```

제한되지 않은 채널을 만들면 기본적으로 무제한의 읽기 권한자와 작성자가 동시에 채널을 사용할 수 있습니다. 또는 `UnboundedChannelOptions` 인스턴스를 제공하여 무제한 채널을 만들 때 네이티브가 아닌 동작을 지정할 수 있습니다. 채널의 용량에는 제한이 없으며 모든 쓰기는 동기식으로 수행됩니다. 더 많은 예를 보려면 무제한 만들기 패턴을 참조하세요.

제한된 채널

제한된 채널을 만들려면 Channel.CreateBounded 오버로드 중 하나를 호출합니다.

C#

```
var channel = Channel.CreateBounded<T>(7);
```

앞의 코드는 최대 `7`개 항목 용량을 갖는 채널을 만듭니다. 제한된 채널을 만들면 채널이 최대 용량으로 제한됩니다. 경계에 도달하면 기본 동작은 공간을 사용할 수 있을 때까지 채널이 생산자를 비동기적으로 차단하는 것입니다. 채널을 만들 때 옵션을 지정하여 이 동작을 구성할 수 있습니다. 제한된 채널은 0보다 큰 용량 값으로 만들어질 수 있습니다. 다른 예는 제한된 만들기 패턴을 참조하세요.

전체 모드 동작

제한된 채널을 사용할 때 구성된 경계에 도달했을 때 채널이 준수하는 동작을 지정할 수 있습니다. 다음 표에는 각 BoundedChannelFullMode 값에 대한 전체 모드 동작이 나열되어 있습니다.

테이블 확장

값	동작
BoundedChannelFullMode.Wait	기본값입니다. `WriteAsync`에 대한 호출은 쓰기 작업을 완료하기 위해 사용 가능한 공간이 생길 때까지 기다립니다. `TryWrite`를 호출하면 즉시 `false`가 반환됩니다.
BoundedChannelFullMode.DropNewest	항목을 쓸 공간을 확보하기 위해 채널의 가장 새로운 항목을 제거하고 무시합니다.
BoundedChannelFullMode.DropOldest	항목을 쓸 공간을 확보하기 위해 채널의 가장 오래된 항목을 제거하고 무시합니다.
BoundedChannelFullMode.DropWrite	쓰고 있는 항목을 삭제합니다.

> **Important**
>
> Channel<TWrite,TRead>.Writer가 Channel<TWrite,TRead>.Reader가 소비할 수 있는 속도보다 더 빠르게 생성될 때마다 채널 작성자는 역압을 경험하게 됩니다.

생산자 API

생산자 기능은 Channel<TWrite,TRead>.Writer에 노출됩니다. 생산자 API 및 예상 동작은 다음 표에 자세히 설명되어 있습니다.

테이블 확장

응용 프로그램 인터페이스 (API)	예상되는 동작
ChannelWriter<T>.Complete	채널을 완료된 것으로 표시합니다. 즉, 더 이상 항목이 기록되지 않음을 의미합니다.
ChannelWriter<T>.TryComplete	채널을 완료된 것으로 표시하려고 시도합니다. 즉, 더 이상 데이터가 기록되지 않음을 의미합니다.
ChannelWriter<T>.TryWrite	지정된 항목을 채널에 기록하려고 시도합니다. 제한되지 않은 채널과 함께 사용하면 채널 작성자가 `true` 또는 ChannelWriter<T>.Complete로 완료 신호를 보내지 않는 한 항상 ChannelWriter<T>.TryComplete를 반환합니다.
ChannelWriter<T>.WaitToWriteAsync	항목을 쓸 수 있는 공간이 생기면 완료되는 ValueTask<TResult>를 반환합니다.
ChannelWriter<T>.WriteAsync	채널에 항목을 비동기적으로 씁니다.

소비자 API

소비자 기능은 Channel<TWrite,TRead>.Reader에 노출됩니다. 소비자 API 및 예상 동작은 다음 표에 자세히 설명되어 있습니다.

테이블 확장

응용 프로그램 인터페이스 (API)	예상되는 동작
ChannelReader<T>.ReadAllAsync	채널의 모든 데이터를 읽을 수 있도록 IAsyncEnumerable<T>을 만듭니다.
ChannelReader<T>.ReadAsync	채널에서 항목을 비동기적으로 읽습니다.
ChannelReader<T>.TryPeek	채널의 항목을 엿보려고 시도합니다.
ChannelReader<T>.TryRead	채널에서 항목을 읽으려고 합니다.
ChannelReader<T>.WaitToReadAsync	데이터를 읽을 수 있게 되면 완료되는 ValueTask<TResult>를 반환합니다.

일반적인 사용 패턴

채널에 대한 몇 가지 사용 패턴이 있습니다.

- 만들기 패턴
- 생산자 패턴
- 소비자 패턴

API는 최대한 단순하고 일관되며 유연하게 설계되었습니다. 모든 비동기 메서드는 작업이 동기적으로 완료되고 잠재적으로 심지어 비동기적으로 완료되는 경우 할당을 피할 수 있는 경량 비동기 작업을 나타내는 `ValueTask`(또는 `ValueTask<bool>`)를 반환합니다. 또한 API는 채널 작성자가 의도한 사용량에 대한 약속을 한다는 측면에서 구성 가능하도록 설계되었습니다. 특정 매개 변수를 사용하여 채널을 만들면 이러한 프라미스를 알고 내부 구현이 보다 효율적으로 작동할 수 있습니다.

만들기 패턴

GPS(Global Position System)용 생산자/소비자 솔루션을 만들고 있다고 가정해 보겠습니다. 시간이 지남에 따라 디바이스의 좌표를 추적하려고 합니다. 샘플 좌표 개체는 다음과 같습니다.

C#

```
/// <summary>
/// A representation of a device's coordinates,
/// which includes latitude and longitude.
/// </summary>
/// <param name="DeviceId">A unique device identifier.</param>
/// <param name="Latitude">The latitude of the device.</param>
/// <param name="Longitude">The longitude of the device.</param>
public readonly record struct Coordinates(
    Guid DeviceId,
    double Latitude,
    double Longitude);
```

제한 없는 만들기 패턴

일반적인 사용 패턴 중 하나는 *기본 바인딩되지 않은* 채널을 만드는 것입니다.

C#

```
var channel = Channel.CreateUnbounded<Coordinates>();
```

그러나 여러 생산자 및 소비자와 함께 바인딩되지 않은 채널을 만들고 싶다고 상상해 보십시오. 채널 옵션에서 `SingleWriter = false` 및 `SingleReader = false`를 설정하십시오:

C#

```
var channel = Channel.CreateUnbounded<Coordinates>(
    new UnboundedChannelOptions
    {
        SingleWriter = false,
        SingleReader = false,
```

```
        AllowSynchronousContinuations = true
    });
```

이 경우 모든 쓰기가 동기적으로 이루어지며, `WriteAsync`도 포함됩니다. 이 동작은 바인딩되지 않은 채널에 항상 즉시 쓰기를 위한 사용 가능한 공간이 있기 때문에 발생합니다. 그러나 `AllowSynchronousContinuations`을 `true`로 설정하면, 쓰기가 판독기와 관련된 연속 작업을 실행할 수 있게 됩니다. 이 설정은 작업의 동시성에 영향을 주지 않습니다.

제한된 만들기 패턴

제한된 채널을 사용하면 적절한 소비를 보장하기 위해 채널의 구성 가능성을 소비자에게 알려야 합니다. 즉, 소비자는 구성된 경계에 도달했을 때 채널이 나타내는 동작을 알아야 합니다. 다음 예제에서는 몇 가지 일반적인 바인딩된 만들기 패턴을 보여 줍니다.

제한된 채널을 만드는 가장 간단한 방법은 용량을 지정하는 것입니다. 다음 코드는 최대 용량 `1`으로 제한된 채널을 만듭니다.

C#

```
var channel = Channel.CreateBounded<Coordinates>(1);
```

다른 옵션을 사용할 수 있습니다. 일부 옵션은 바인딩되지 않은 채널과 동일하지만 다른 옵션은 제한된 채널과 관련이 있습니다. 다음 코드에서 채널은 1,000개의 항목으로 제한되는 제한된 채널로 생성되며, 단일 작성기는 있지만 많은 판독기가 있습니다. 전체 모드 동작은 `DropWrite`로 정의됩니다. 즉, 채널이 가득 차면 기록 중인 항목을 삭제한다는 의미입니다.

C#

```
var channel = Channel.CreateBounded<Coordinates>(
    new BoundedChannelOptions(1_000)
    {
        SingleWriter = true,
        SingleReader = false,
        AllowSynchronousContinuations = false,
        FullMode = BoundedChannelFullMode.DropWrite
    });
```

제한된 채널을 사용할 때 삭제되는 항목을 관찰하려면 `itemDropped` 콜백을 등록합니다.

C#

```
var channel = Channel.CreateBounded(
    new BoundedChannelOptions(10)
    {
```

```
        AllowSynchronousContinuations = true,
        FullMode = BoundedChannelFullMode.DropOldest
    },
    static void (Coordinates dropped) =>
        Console.WriteLine($"Coordinates dropped: {dropped}"));
```

채널이 가득 차고 새 항목이 추가될 때마다 `itemDropped` 콜백이 호출됩니다. 이 예에서 제공된 콜백은 항목을 콘솔에 기록하지만 원하는 다른 작업을 자유롭게 수행할 수 있습니다.

생산자 패턴

이 시나리오의 생산자가 채널에 새 좌표를 쓰고 있다고 상상해 보세요. 생산자는 TryWrite를 호출하여 이 작업을 수행할 수 있습니다.

C#

```
static void ProduceWithWhileAndTryWrite(
    ChannelWriter<Coordinates> writer, Coordinates coordinates)
{
    while (coordinates is { Latitude: < 90, Longitude: < 180 })
    {
        var tempCoordinates = coordinates with
        {
            Latitude = coordinates.Latitude + .5,
            Longitude = coordinates.Longitude + 1
        };

        if (writer.TryWrite(item: tempCoordinates))
        {
            coordinates = tempCoordinates;
        }
    }
}
```

이전 생산자 코드:

- 초기 `Channel<Coordinates>.Writer`와 함께 `ChannelWriter<Coordinates>`(`Coordinates`)을 인수로 허용합니다.
- `while`를 사용하여 좌표 이동을 시도하는 조건부 `TryWrite` 루프를 정의합니다.

대체 생산자는 `WriteAsync` 메서드를 사용할 수 있습니다.

C#

```
static async ValueTask ProduceWithWhileWriteAsync(
    ChannelWriter<Coordinates> writer, Coordinates coordinates)
{
    while (coordinates is { Latitude: < 90, Longitude: < 180 })
```

```
    {
        await writer.WriteAsync(
            item: coordinates = coordinates with
            {
                Latitude = coordinates.Latitude + .5,
                Longitude = coordinates.Longitude + 1
            });
    }

    writer.Complete();
}
```

다시 말하지만, `Channel<Coordinates>.Writer`는 `while` 루프 내에서 사용됩니다. 하지만 이번에는 WriteAsync 메서드가 호출됩니다. 메서드는 좌표를 쓴 후에만 계속됩니다. `while` 루프가 종료되면 Complete가 호출되어 더 이상 채널에 데이터가 기록되지 않는다는 신호를 보냅니다.

또 다른 생산자 패턴은 WaitToWriteAsync 메서드를 사용하는 것입니다. 다음 코드를 고려합니다.

C#

```
static async ValueTask ProduceWithWaitToWriteAsync(
    ChannelWriter<Coordinates> writer, Coordinates coordinates)
{
    while (coordinates is { Latitude: < 90, Longitude: < 180 } &&
        await writer.WaitToWriteAsync())
    {
        var tempCoordinates = coordinates with
        {
            Latitude = coordinates.Latitude + .5,
            Longitude = coordinates.Longitude + 1
        };

        if (writer.TryWrite(item: tempCoordinates))
        {
            coordinates = tempCoordinates;
        }

        await Task.Delay(TimeSpan.FromMilliseconds(10));
    }

    writer.Complete();
}
```

조건부 `while`의 일부로 `WaitToWriteAsync` 호출의 결과는 루프를 계속할지 여부를 결정하는 데 사용됩니다.

소비자 패턴

몇 가지 일반적인 채널 소비자 패턴이 있습니다. 채널이 끝나지 않는 경우, 즉 데이터를 무기한 생성하는 경우 소비자는 `while (true)` 루프를 사용하고 데이터가 사용 가능해지면 읽을 수 있습니다.

C#

```
static async ValueTask ConsumeWithWhileAsync(
    ChannelReader<Coordinates> reader)
{
    while (true)
    {
        // May throw ChannelClosedException if
        // the parent channel's writer signals complete.
        Coordinates coordinates = await reader.ReadAsync();
        Console.WriteLine(coordinates);
    }
}
```

> ⓘ **참고 항목**
>
> 이 코드는 채널이 닫힌 경우 예외를 throw합니다.

대체 소비자는 다음 코드와 같이 중첩된 while 루프를 사용하여 이러한 문제를 피할 수 있습니다.

C#

```
static async ValueTask ConsumeWithNestedWhileAsync(
    ChannelReader<Coordinates> reader)
{
    while (await reader.WaitToReadAsync())
    {
        while (reader.TryRead(out Coordinates coordinates))
        {
            Console.WriteLine(coordinates);
        }
    }
}
```

이전 코드에서 소비자는 데이터 읽기를 기다립니다. 데이터를 사용할 수 있게 되면 소비자는 데이터를 읽으려고 시도합니다. 이러한 루프는 채널 생성자가 더 이상 읽을 데이터가 없다는 신호를 보낼 때까지 계속 평가합니다. 즉, 생산자가 생산하는 항목의 수가 한정되어 있고 완료 신호를 보내는 경우 소비자는 `await foreach` 의미 체계를 사용하여 항목을 반복할 수 있습니다.

C#

```
static async ValueTask ConsumeWithAwaitForeachAsync(
    ChannelReader<Coordinates> reader)
{
    await foreach (Coordinates coordinates in reader.ReadAllAsync())
    {
        Console.WriteLine(coordinates);
    }
}
```

앞의 코드는 ReadAllAsync 메서드를 사용하여 채널에서 모든 좌표를 읽습니다.

여러 생산자 및 소비자

채널은 여러 동시 생산자와 소비자를 지원합니다. 이를 가능하게 하려면 채널 옵션에 `SingleWriter = false` 및 `SingleReader = false`을 사용하여 채널을 만드세요. 그런 다음 여러 생산자 태스크에 대해 쓰기를 확장하고 여러 소비자 태스크에 대해 읽기를 통합합니다.

C#

```
static async Task UseMultipleProducersAndConsumersAsync()
{
    Channel<Coordinates> channel = Channel.CreateUnbounded<Coordinates>(
        new UnboundedChannelOptions
        {
            SingleWriter = false,
            SingleReader = false
        });

    // Start three concurrent producer tasks.
    Task[] producerTasks = Enumerable.Range(0, 3)
        .Select(id => ProduceAsync(id, channel))
        .ToArray();

    // Start two concurrent consumer tasks.
    Task[] consumerTasks = Enumerable.Range(0, 2)
        .Select(_ => ConsumeAsync(channel))
        .ToArray();

    // Wait for all producers to finish, then mark the channel as complete.
    await Task.WhenAll(producerTasks);
    channel.Writer.Complete();

    // Wait for all consumers to finish.
    await Task.WhenAll(consumerTasks);

    static async Task ProduceAsync(int id, Channel<Coordinates> channel)
    {
        Coordinates coordinates = new(
            DeviceId: Guid.NewGuid(),
            Latitude: -90 + (id * 30),
```

```
            Longitude: -180 + (id * 60));

        while (coordinates is { Latitude: < 90, Longitude: < 180 })
        {
            coordinates = coordinates with
            {
                Latitude = coordinates.Latitude + 0.5,
                Longitude = coordinates.Longitude + 1
            };

            await channel.Writer.WriteAsync(coordinates);
        }
    }

    static async Task ConsumeAsync(Channel<Coordinates> channel)
    {
        await foreach (Coordinates coordinates in channel.Reader.ReadAllAsync())
        {
            Console.WriteLine(coordinates);
        }
    }
}
```

앞의 코드는 다음과 같습니다.

- 여러 동시 작성기와 판독기를 명시적으로 지원하는 바인딩되지 않은 채널을 만듭니다.
- 각각 고유한 디바이스 식별자를 사용하여 일련의 좌표를 작성하는 세 가지 동시 생산자 작업을 시작합니다.
- `ReadAllAsync`을 사용하여 동일한 채널에서 각각 읽는 두 개의 동시 소비자 작업을 시작합니다.
- 모든 생산자가 완료되기를 기다린 다음, 데이터가 더 이상 채널에 기록되지 않음을 알리기 위해 Complete를 호출합니다.
- 모든 소비자가 채널에서 남은 데이터를 처리하기를 기다립니다.

> **팁**
>
> 여러 생산자를 사용하면 `channel.Writer.Complete()` 생산자가 쓰기를 마친 후에만 호출 합니다. 그러면 더 이상 데이터가 기록되지 않음을 알리고 `ReadAllAsync()` 나머지 항목을 모두 사용한 후 완료할 수 있습니다.

참고 항목

- .NET 쇼: .NET에서 채널을 활용한 작업
- .NET 블로그: System.Threading.Channels 소개

- 관리되는 스레드 처리 기본 사항

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 04. 03.

System.MidpointRounding 열거형

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

MidpointRounding 열거형을 적절한 Math.Round, MathF.Round, Decimal.Round 오버로드와 함께 사용하여 반올림 프로세스를 보다 세밀하게 제어할 수 있습니다.

전체 반올림 전략 두 개(*가장 가까운 반올림* 및 *방향 반올림*)가 있으며 각 열거형 필드는 이러한 전략 중 하나에 정확히 참여합니다.

가장 가까운 수로 반올림

필드:

- AwayFromZero
- ToEven

가장 가까운 반올림 작업은 암시적 또는 지정된 정밀도를 가진 원래 숫자를 취하고, 그 정밀도보다 하나 높은 자리의 숫자를 검사하여 원래 수와 같은 정밀도를 가진 가장 가까운 숫자를 반환합니다. 양수의 경우 다음 숫자가 0에서 4까지인 경우 가장 가까운 숫자는 음수 무한대입니다. 다음 숫자가 6에서 9까지인 경우 가장 가까운 숫자는 양수 무한대입니다. 음수의 경우 다음 숫자가 0에서 4까지인 경우 가장 가까운 숫자는 양수 무한대입니다. 다음 숫자가 6에서 9까지인 경우 가장 가까운 숫자는 음수 무한대입니다.

다음 숫자가 0에서 4 또는 6에서 9 `MidpointRounding.AwayFromZero` `MidpointRounding.ToEven` 까지이면 반올림 작업의 결과에 영향을 미치지 않습니다. 그러나 다음 숫자가 5이고, 두 개의 가능한 결과 사이의 중간점이고, 나머지 모든 숫자가 0이거나 나머지 숫자가 없는 경우 가장 가까운 숫자는 모호합니다. 이 경우, `MidpointRounding`의 가장 가까운 반올림 모드는 반올림 연산이 0에서 떨어진 가장 가까운 숫자를 반환할지, 아니면 가장 가까운 짝수를 반환할지를 지정할 수 있게 합니다.

다음 표는 반올림 모드를 사용하여 일부 음수와 양수를 가장 가까운 값으로 반올림한 결과를 보여줍니다. 숫자를 반올림하는 데 사용되는 정밀도는 0입니다. 즉, 소수점 뒤의 숫자가 반올림 작업에 영향을 줍니다. 예를 들어 숫자 -2.5의 경우 소수점 뒤의 숫자는 5입니다. 해당 숫자가 중간점이므로 `MidpointRounding` 값을 사용하여 반올림 결과를 결정할 수 있습니다. `AwayFromZero`이 지정된 경우, -3이 반환됩니다. 이는 자릿수가 0인 상태에서 0에 가장 가까운 숫자이기 때문입니다. 지정된 경우 `ToEven`는 0과 같은 정밀도를 가진 가장 가까운 짝수인 -2로 반환됩니다.

테이블 확장

원래 번호	AwayFromZero	ToEven
3.5	4	4
2.8	3	3
2.5	3	2
2.1	2	2
-2.1	-2	-2
-2.5	-3	-2
-2.8	-3	-3
-3.5	-4	-4

지시된 반올림

필드:

- ToNegativeInfinity
- ToPositiveInfinity
- ToZero

방향 반올림 작업은 암시적 또는 지정된 정밀도로 원래 숫자를 사용하고 원래 숫자와 동일한 정밀도로 특정 방향으로 다음으로 가장 가까운 숫자를 반환합니다. 미리 정의된 숫자를 향해 반올림 방향을 제어하는 모드입니다.

다음 표에서는 지시된 반올림 방법과 함께 일부 음수 및 양수를 반올림한 결과를 보여 줍니다. 숫자를 반올림할 때 사용되는 정밀도는 0입니다. 즉, 소수점 앞의 숫자가 반올림의 영향을 받습니다.

테이블 확장

원래 번호	ToNegativeInfinity	ToPositiveInfinity	ToZero
3.5	3	4	3
2.8	2	3	2
2.5	2	3	2
2.1	2	3	2
-2.1	-3	-2	-2

원래 번호	ToNegativeInfinity	ToPositiveInfinity	ToZero
-2.5	-3	-2	-2
-2.8	-3	-2	-2
-3.5	-4	-3	-3

Microsoft.Win32.Registry 클래스

> **참고 항목**
>
> 이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

이 클래스는 Registry Windows를 실행하는 컴퓨터의 레지스트리에 있는 표준 루트 키 집합을 제공합니다. 레지스트리는 애플리케이션, 사용자 및 기본 시스템 설정에 대 한 정보에 대 한 스토리지 기능을 합니다. 애플리케이션은 레지스트리를 사용하여 애플리케이션을 닫은 후 보존해야 하는 정보를 저장하고 애플리케이션을 다시 로드할 때 동일한 정보에 액세스할 수 있습니다. 예를 들어 색 기본 설정, 화면 위치 또는 창 크기를 저장할 수 있습니다. 레지스트리의 다른 위치에 정보를 저장하여 각 사용자에 대해 이 데이터를 제어할 수 있습니다.

클래스에서 RegistryKey 노출하는 기본 또는 루트 `Registry` 인스턴스는 레지스트리의 하위 키 및 값에 대한 기본 저장 메커니즘을 구체화합니다. 레지스트리가 이들의 존재에 의존하기 때문에 모든 키는 읽기 전용입니다. 노출되는 `Registry` 키는 다음과 같습니다.

테이블 확장

암호키	설명
CurrentUser	사용자 기본 설정에 대한 정보를 저장합니다.
LocalMachine	로컬 컴퓨터에 대한 구성 정보를 저장합니다.
ClassesRoot	형식(및 클래스) 및 해당 속성에 대한 정보를 저장합니다.
Users	기본 사용자 구성에 대한 정보를 저장합니다.
PerformanceData	소프트웨어 구성 요소에 대한 성능 정보를 저장합니다.
CurrentConfig	사용자별이 아닌 하드웨어 정보를 저장합니다.
DynData	동적 데이터를 저장합니다.

레지스트리에서 정보를 저장/검색하려는 루트 키를 식별한 후에는 클래스를 사용하여 RegistryKey 하위 키를 추가 또는 제거하고 지정된 키의 값을 조작할 수 있습니다.

플러그 앤 플레이 인터페이스를 사용하여 하드웨어 장치는 자동으로 정보를 레지스트리에 저장할 수 있습니다. 소프트웨어 디바이스 드라이버 설치에 대 한 표준 Api를 작성 하 여 레지스트리에서 정보를 배치할 수 있습니다.

값을 가져오고 설정하기 위한 정적 메서드

클래스에는 Registry 레지스트리 키에서 값을 설정하고 검색하는 메서드도 포함되어 `static` GetValueSetValue 있습니다. 이러한 메서드는 사용할 때마다 레지스트리 키를 열고 닫습니다. 많은 수의 값에 액세스할 때, RegistryKey 클래스의 유사한 메서드에 비해 성능이 떨어집니다.

클래스는 RegistryKey 다음을 수행할 수 있는 메서드도 제공합니다.

- 레지스트리 키에 대한 Windows 액세스 제어 보안을 설정합니다.
- 값을 검색하기 전에 값의 데이터 형식을 테스트합니다.
- 키를 삭제합니다.

Last updated on 2026. 02. 12.

System.Uri 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

URI(Uniform Resource Identifier)는 인트라넷 또는 인터넷에서 애플리케이션에서 사용할 수 있는 리소스의 압축 표현입니다. 클래스는 Uri 구문 분석, 비교 및 결합을 포함하여 URI를 처리하기 위한 속성과 메서드를 정의합니다. 클래스 속성은 Uri 읽기 전용입니다. 수정 가능한 개체를 만들려면 클래스를 UriBuilder 사용합니다.

상대 URI(예: "/new/index.htm")는 절대 URI가 되도록 기본 URI와 관련하여 확장해야 합니다. MakeRelativeUri 필요한 경우 절대 URI를 상대 URI로 변환하기 위해 메서드가 제공됩니다.

문자열이 Uri 스키마 식별자를 포함하는 올바른 형식의 URI인 경우 생성자는 URI 문자열을 이스케이프하지 않습니다.

이 속성은 Uri 이스케이프된 인코딩에서 정식 데이터 표현을 반환하며, 유니코드 값이 127보다 큰 모든 문자는 해당 16진수로 바뀝니다. URI를 정식 형식 Uri 으로 배치하기 위해 생성자는 다음 단계를 수행합니다.

- URI 구성표를 소문자로 변환합니다.
- 호스트 이름을 소문자로 변환합니다.
- 호스트 이름이 IPv6 주소인 경우 정식 IPv6 주소가 사용됩니다. ScopeId 및 기타 선택적 IPv6 데이터가 제거됩니다.
- 기본 포트 번호와 빈 포트 번호를 제거합니다.
- file:// 체계가 없는 암시적 파일 경로(예: "C:\my\file")를 file:// 체계를 사용하여 명시적 파일 경로로 변환합니다.
- 예약된 용도가 없는 이스케이프된 문자(백분율로 인코딩된 옥텟이라고도 함)는 디코딩됩니다(언이스케이프라고도 함). 이러한 예약되지 않은 문자에는 대문자와 소문자(%41-%5A 및 %61-%7A), 10진수(%30-%39), 하이픈(%2D), 마침표(%2E), 밑줄(%5F), 타일드(%7E)가 포함됩니다.
- 계층 URI의 경로를 표준화하여 `/./` 및 `/../`과 같은 시퀀스를 이스케이프 여부에 관계없이 압축합니다. 이러한 시퀀스가 압축되지 않는 몇 가지 구성표가 있습니다.
- 계층적 URI의 경우 호스트가 슬래시(/)로 종료되지 않으면 URI가 추가됩니다.
- 기본적으로 URI의 예약 문자는 RFC 2396에 따라 이스케이프됩니다. 이 동작은 RFC 3986 및 RFC 3987에 따라 URI의 예약 문자가 이스케이프되는 경우 국가별 리소스 식별자 또는 국제 도메인 이름 구문 분석이 사용하도록 설정된 경우 변경됩니다.

일부 구성표에 대한 생성자의 정식화의 일부로 점 세그먼트(`/./` 및 `/../`)가 압축됩니다(즉, 제거됨). 세그먼트를 압축하는 Uri 구성표에는 http, https, tcp, net.pipe 및 net.tcp가 포함됩니다. 다른 구성표의 경우 이러한 시퀀스는 압축되지 않습니다. 다음 코드 조각은 압축이 실제로 어떻

게 보이는지 보여줍니다. 이스케이프된 시퀀스는 필요한 경우 이스케이프 해제된 후 압축됩니다.

C#

```
var uri = new Uri("http://myUrl/../.."); // http scheme, unescaped
OR
var uri = new Uri("http://myUrl/%2E%2E/%2E%2E"); // http scheme, escaped
OR
var uri = new Uri("ftp://myUrl/../.."); // ftp scheme, unescaped
OR
var uri = new Uri("ftp://myUrl/%2E%2E/%2E%2E"); // ftp scheme, escaped

Console.WriteLine($"AbsoluteUri: {uri.AbsoluteUri}");
Console.WriteLine($"PathAndQuery: {uri.PathAndQuery}");
```

이 코드가 실행되면 다음 텍스트와 유사한 출력을 반환합니다.

출력

```
AbsoluteUri: http://myurl/
PathAndQuery: /
```

메서드를 Uri 사용하여 이스케이프 인코딩된 URI 참조에서 읽기 가능한 URI 참조로 클래스의 ToString 내용을 변환할 수 있습니다. 참고로 메서드의 ToString 출력에서 일부 특수 문자가 여전히 이스케이프될 수 있습니다. 이는 반환 ToString된 값에서 URI의 명확한 재구성을 지원하기 위한 것입니다.

일부 URI에는 조각 식별자 또는 쿼리 또는 둘 다 포함됩니다. 조각 식별자는 숫자 기호를 포함하지 않고 숫자 기호(#)를 따르는 텍스트입니다. 조각 텍스트가 속성에 Fragment 저장됩니다. 쿼리 정보는 URI의 물음표(?)를 따르는 텍스트입니다. 쿼리 텍스트는 속성에 Query 저장됩니다.

> ⓘ **참고 항목**
>
> URI 클래스는 IPv4 프로토콜에 대한 쿼드 표기법과 IPv6 프로토콜의 콜론-16진수 모두에서 IP 주소 사용을 지원합니다. http://[::1]에서와 같이 IPv6 주소를 대괄호로 묶는 것을 잊지 마세요.

국가별 리소스 식별자 지원

웹 주소는 일반적으로 매우 제한된 문자 집합으로 구성된 URI를 사용하여 표현됩니다.

- 영어 알파벳의 대문자 및 소문자 ASCII 문자입니다.
- 0에서 9까지의 숫자입니다.

- 소수의 다른 ASCII 기호입니다.

URI 사양은 IETF(인터넷 엔지니어링 태스크 포스)에서 게시한 RFC 2396, RFC 2732, RFC 3986 및 RFC 3987에 설명되어 있습니다.

영어 이외의 언어를 사용하여 리소스를 식별하고 비 ASCII 문자(유니코드/ISO 10646 문자 집합의 문자)를 허용하는 식별자를 URI(International Resource Identifier)라고 합니다. IRIs의 사양은 IETF에서 게시한 RFC 3987에 문서화되어 있습니다. IRIs를 사용하면 URL에 유니코드 문자가 포함될 수 있습니다.

.NET Framework 4.5 이상 버전에서는 IRI가 항상 사용하도록 설정되어 있으며 구성 옵션을 사용하여 변경할 수 없습니다. *machine.config* 또는 *app.config* 파일에서 구성 옵션을 설정하여 IDN(Internationalized Domain Name) 구문 분석을 도메인 이름에 적용할지 여부를 지정할 수 있습니다. 다음은 그 예입니다.

XML

```
<configuration>
  <uri>
    <idn enabled="All" />
  </uri>
</configuration>
```

IDN을 사용하도록 설정하면 도메인 이름의 모든 유니코드 레이블이 해당하는 Punycode 레이블로 변환됩니다. Punycode 이름은 ASCII 문자만 포함하며 항상 xn-- 접두사로 시작합니다. 대부분의 DNS 서버는 ASCII 문자만 지원하므로 인터넷에서 기존 DNS 서버를 지원하기 때문입니다(RFC 3940 참조).

IDN을 사용하도록 설정하면 속성 값에 영향을 줍니다 Uri.DnsSafeHost . IDN을 사용하도록 설정하면 Equals, OriginalString, GetComponents, 및 IsWellFormedOriginalString 메서드의 동작도 변경할 수 있습니다.

사용되는 DNS 서버에 따라 IDN에 대한 세 가지 가능한 값이 있습니다.

- IDN 활성화됨 = 모두

  유니코드 도메인 이름을 해당하는 Punycode(IDN 이름)로 변환합니다.

- idn 활성화됨 = 내부 네트워크 제외 모두

  로컬 인트라넷에 없는 모든 유니코드 도메인 이름을 변환하여 IDN 이름(Punycode 등가물)을 사용합니다. 이 경우 로컬 인트라넷에서 국가별 이름을 처리하려면 인트라넷에 사용되는 DNS 서버가 유니코드 이름 확인을 지원해야 합니다.

- idn 활성화 상태 = 없음

Punycode를 사용하도록 변환된 유니코드 도메인 이름은 없습니다. 기본값입니다.

정규화 및 문자 검사는 RFC 3986 및 RFC 3987의 최신 IRI 규칙에 따라 수행됩니다.

Uri 클래스에서 IRI 및 IDN 처리는 System.Configuration.IriParsingElement, System.Configuration.IdnElement, System.Configuration.UriSection 구성 설정 클래스들을 사용하여 제어할 수 있습니다. 이 설정은 System.Configuration.IriParsingElement 클래스에서 Uri IRI 처리를 사용하거나 사용하지 않도록 설정합니다. 이 설정은 System.Configuration.IdnElement 클래스에서 Uri IDN 처리를 사용하거나 사용하지 않도록 설정합니다.

첫 번째 System.Configuration.IriParsingElement 클래스가 생성될 때, System.Configuration.IdnElement 및 System.Uri에 대한 구성 설정이 한 번 읽힙니다. 해당 시간 이후의 구성 설정 변경 내용은 무시됩니다.

System.GenericUriParser 또한 IRI 및 IDN을 지원하는 사용자 지정 가능한 파서 만들기를 허용하도록 클래스가 확장되었습니다. System.GenericUriParser 개체의 동작은 System.GenericUriParserOptions 열거형에서 사용할 수 있는 값들을 비트 조합으로 하여 System.GenericUriParser 생성자에 전달함으로써 지정됩니다. 이 형식은 GenericUriParserOptions.IriParsing 파서가 RFC 3987 for International Resource Identifiers(IRI)에 지정된 구문 분석 규칙을 지원했음을 나타냅니다.

이 형식은 GenericUriParserOptions.Idn 파서가 호스트 이름의 IDN(Internationalized Domain Name) 구문 분석을 지원한다는 것을 나타냅니다. .NET 5 이상 버전(.NET Core 포함) 및 .NET Framework 4.5 이상에서는 항상 IDN이 사용됩니다. 이전 버전에서 구성 옵션은 IDN이 사용되는지 여부를 결정합니다.

암시적 파일 경로 지원

Uri 를 사용하여 로컬 파일 시스템 경로를 나타낼 수도 있습니다. 이러한 경로는 file:// 스키마로 시작하는 URI에서 *명시적으로* 표시될 수 있으며 file:// 체계가 없는 URI에서 *암시적으로* 나타낼 수 있습니다. 구체적인 예로, 다음 두 URI는 모두 유효하며 동일한 파일 경로를 나타냅니다.

C#

```
Uri uri1 = new Uri("C:/test/path/file.txt") // Implicit file path.
Uri uri2 = new Uri("file:///C:/test/path/file.txt") // Explicit file path.
```

이러한 암시적 파일 경로는 URI 사양을 준수하지 않으므로 가능하면 피해야 합니다. Unix 기반 시스템에서 .NET Core를 사용하는 경우 절대 암시적 파일 경로가 상대 경로와 *구별할* 수 없으므로 암시적 파일 경로가 특히 문제가 될 수 있습니다. 이러한 모호성이 있는 Uri 경우 기본적으로 경로를 절대 URI로 해석합니다.

보안 고려 사항

보안상의 이유로 애플리케이션은 신뢰할 수 없는 원본에서 Uri 인스턴스를 수락할 때 주의해야 하며, `dontEscape`가 `true`로 설정된 상태에서 생성자와 함께 사용해야 합니다. 메서드를 호출 IsWellFormedOriginalString 하여 URI 문자열의 유효성을 확인할 수 있습니다.

신뢰할 수 없는 사용자 입력을 처리할 때 속성을 신뢰하기 전에 새로 만든 `Uri` 인스턴스에 대한 가정을 확인합니다. 이 작업은 다음과 같은 방법으로 수행할 수 있습니다.

C#

```
string userInput = ...;

Uri baseUri = new Uri("https://myWebsite/files/");

if (!Uri.TryCreate(baseUri, userInput, out Uri newUri))
{
    // Fail: invalid input.
}

if (!baseUri.IsBaseOf(newUri))
{
    // Fail: the Uri base has been modified - the created Uri is not rooted in the
original directory.
}
```

이 유효성 검사는 단순히 `baseUri`을 변경해서 UNC 경로를 처리할 때와 같은 경우에 사용할 수 있습니다.

C#

```
Uri baseUri = new Uri(@"\\host\share\some\directory\name\");
```

디자인 및 보안 고려 사항에 UriUriBuilder대한 자세한 내용은 다음 위협 모델 문서를 검토하세요.

- System.Uri 위협 모델
- System.UriBuilder 위협 모델

성능 고려 사항

URI가 포함된 *Web.config* 파일을 사용하여 애플리케이션을 초기화하는 경우 스키마 식별자가 비표준인 경우 URI를 처리하는 데 추가 시간이 필요합니다. 이러한 경우 시작 시간이 아닌 URI가 필요할 때 애플리케이션의 영향을 받는 부분을 초기화합니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 18.

System.Type 클래스

아티클 • 2025. 04. 05.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스는 Type 기능의 System.Reflection 루트이며 메타데이터에 액세스하는 기본 방법입니다. 멤버를 Type 사용하여 형식 선언, 형식의 멤버(예: 클래스의 생성자, 메서드, 필드, 속성 및 이벤트)와 클래스가 배포되는 모듈 및 어셈블리에 대한 정보를 가져옵니다.

코드가 리플렉션을 사용하여 액세스 수준에 관계없이 형식 및 해당 멤버에 대한 정보를 가져오는 데 필요한 권한은 없습니다. 코드가 리플렉션을 사용하여 공용 멤버에 액세스하거나 액세스 수준이 일반 컴파일 중에 표시되도록 하는 다른 멤버에 대한 사용 권한이 필요하지 않습니다. 그러나 코드가 리플렉션을 사용하여 일반적으로 액세스할 수 없는 멤버(예: 프라이빗 또는 내부 메서드 또는 클래스가 상속하지 않는 형식의 보호된 필드)에 액세스하려면 코드에 있어야 ReflectionPermission합니다. 리플렉션에 대한 보안 고려 사항을 참조하세요.

`Type` 는 여러 구현을 허용하는 추상 기본 클래스입니다. 시스템은 항상 파생 클래스 `RuntimeType`를 제공합니다. 리플렉션에서 런타임이라는 단어로 시작하는 모든 클래스는 시스템의 개체당 한 번만 만들어지고 비교 작업을 지원합니다.

> ⓘ **참고**
>
> 다중 스레딩 시나리오에서는 **Type** 개체를 잠가 `static` 데이터에 대한 액세스를 동기화하지 마세요. 제어할 수 없는 다른 코드도 클래스 형식을 잠글 수 있습니다. 이로 인해 교착 상태가 발생할 수 있습니다. 대신 프라이빗 `static` 개체를 잠가 정적 데이터에 대한 액세스를 동기화합니다.

> ⓘ **참고**
>
> 파생 클래스는 호출 코드의 기본 클래스의 보호된 멤버에 액세스할 수 있습니다. 또한 호출 코드 어셈블리의 어셈블리 멤버에 대한 액세스가 허용됩니다. 일반적으로 초기 바인딩된 코드에서 액세스가 허용되는 경우, 후기 바인딩된 코드에서도 액세스가 허용됩니다.

> ⓘ **참고**
>
> 다른 인터페이스를 확장하는 인터페이스는 확장된 인터페이스에 정의된 메서드를 상속하지 않습니다.

Type 개체는 어떤 형식을 나타내나요?

이 클래스는 스레드로부터 안전합니다. 여러 스레드는 이 형식의 인스턴스에서 동시에 읽을 수 있습니다. 클래스의 인스턴스는 Type 다음 형식을 나타낼 수 있습니다.

- 수업
- 값 형식
- 배열
- 인터페이스
- 열거
- 대표자
- 생성된 제네릭 형식 및 제네릭 형식 정의
- 생성된 제네릭 형식, 제네릭 형식 정의 및 제네릭 메서드 정의의 형식 인수 및 형식 매개 변수

Type 개체 검색

특정 형식과 연결된 개체는 Type 다음과 같은 방법으로 가져올 수 있습니다.

- 인스턴스 메서드는 인스턴스 Object.GetType 의 형식을 나타내는 개체를 반환 Type 합니다. 모든 관리되는 형식이 Object파생되므로 모든 형식의 GetType 인스턴스에서 메서드를 호출할 수 있습니다.

  다음 예제에서는 메서드를 Object.GetType 호출하여 개체 배열에 있는 각 개체의 런타임 형식을 확인합니다.

  C#

  ```
  object[] values = { "word", true, 120, 136.34, 'a' };
  foreach (var value in values)
      Console.WriteLine($"{value} - type {value.GetType().Name}");

  // The example displays the following output:
  //       word - type String
  //       True - type Boolean
  //       120 - type Int32
  //       136.34 - type Double
  //       a - type Char
  ```

- 정적 Type.GetType 메서드는 정규화된 이름으로 지정된 형식을 나타내는 개체를 반환 Type 합니다.

- Module.GetTypes, Module.GetType, 및 Module.FindTypes 메서드는 모듈에 정의된 형식을 나타내는 `Type` 개체를 반환합니다. 첫 번째 메서드는 모듈에 정의된 모든 public 및 private 형식에 대한 개체 배열 Type 을 가져오는 데 사용할 수 있습니다. (c0 /> 인스턴스는

Assembly.GetModule 또는 Assembly.GetModules 메서드를 통해, 혹은 Type.Module 속성을 통해 얻을 수 있습니다.)

- 개체 System.Reflection.Assembly 에는 어셈블리에 정의된 클래스(예 Assembly.GetType: , 및)를 검색하는 여러 메서드가 Assembly.GetTypes포함되어 있습니다 Assembly.GetExportedTypes.
- 메서드는 FindInterfaces 형식에서 지원하는 인터페이스 형식의 필터링된 목록을 반환합니다.
- GetElementType 메서드는 요소를 나타내는 `Type` 개체를 반환합니다.
- GetInterfaces 및 GetInterface 메서드는 형식이 지원하는 인터페이스 형식을 나타내는 Type 개체를 반환합니다.
- GetTypeArray 메서드는 임의의 개체 집합으로 지정된 형식을 나타내는 Type 개체 배열을 반환합니다. 개체는 형식 Object의 배열로 지정됩니다.
- GetTypeFromProgID 및 GetTypeFromCLSID 메서드는 COM 상호 운용성을 위해 제공됩니다. Type 객체를 반환하여 `ProgID` 또는 `CLSID`에 지정된 형식을 나타냅니다.
- 상호 운용성을 위해 GetTypeFromHandle 메서드가 제공됩니다. 클래스 핸들에 `Type` 지정된 형식을 나타내는 개체를 반환합니다.
- C# `typeof` 연산자, C++ `typeid` 연산자 및 Visual Basic `GetType` 연산자는 형식에 `Type` 대한 개체를 가져옵니다.
- MakeGenericType 메서드는 생성된 제네릭 형식을 나타내는 Type 개체를 반환합니다. 이 형식은 ContainsGenericParameters 속성이 `true`을(를) 반환할 경우 열린 생성 형식이며, 그렇지 않으면 닫힌 생성 형식입니다. 제네릭 형식은 닫힌 경우에만 인스턴스화할 수 있습니다.
- MakeArrayType, MakePointerType, 및 MakeByRefType 메서드는 각각 지정된 형식의 배열, 지정된 형식에 대한 포인터, 참조 매개 변수의 형식을 나타내는 Type 개체를 반환합니다 (`ref` C#, F#의 'byref', `ByRef` Visual Basic).

형식 개체의 동등성 비교

Type 형식을 나타내는 개체는 고유합니다. 즉, 동일한 형식을 나타내는 경우에만 두 Type 개의 개체 참조가 동일한 개체를 참조합니다. 이렇게 하면 참조 같음을 사용하여 개체를 Type 비교할 수 있습니다. 다음 예제에서는 여러 정수 값을 나타내는 개체를 비교하여 Type 동일한 형식인지 여부를 확인합니다.

```
C#
```

```
long number1 = 1635429;
int number2 = 16203;
double number3 = 1639.41;
long number4 = 193685412;

// Get the type of number1.
Type t = number1.GetType();

// Compare types of all objects with number1.
Console.WriteLine($"Type of number1 and number2 are equal:
{Object.ReferenceEquals(t, number2.GetType())}");
Console.WriteLine($"Type of number1 and number3 are equal:
{Object.ReferenceEquals(t, number3.GetType())}");
Console.WriteLine($"Type of number1 and number4 are equal:
{Object.ReferenceEquals(t, number4.GetType())}");

// The example displays the following output:
//       Type of number1 and number2 are equal: False
//       Type of number1 and number3 are equal: False
//       Type of number1 and number4 are equal: True
```

System.Type.GetProperty 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

다음 지침은 모든 오버로드에 적용됩니다.

- 속성은 공용 접근자가 하나 이상 있으면 리플렉션에서 공용으로 간주됩니다. 그렇지 않으면 속성이 프라이빗으로 간주되며, 속성을 얻으려면 BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static를 사용해야 합니다 (Visual Basic에서는 `Or`를 사용하여 값을 결합합니다).
- 현재 Type 가 생성된 제네릭 형식을 나타내는 경우 이 메서드는 적절한 형식 인수로 대체된 형식 매개 변수를 반환 PropertyInfo 합니다.
- 현재 Type 제네릭 형식 또는 제네릭 메서드 정의에서 형식 매개 변수를 나타내는 경우 이 메서드는 클래스 제약 조건의 속성을 검색합니다.

GetProperty(String) 메서드

검색에서 `name`은 대/소문자를 구분합니다. 검색에는 공용 정적 및 공용 인스턴스 속성이 포함됩니다.

발생하는 상황에는 AmbiguousMatchException 다음이 포함됩니다.

- 형식에는 이름이 같지만 매개 변수 수가 다른 두 개의 인덱싱된 속성이 포함됩니다. 모호성을 해결하려면 매개 변수 형식을 지정하는 메서드의 GetProperty 오버로드를 사용합니다.
- 파생 형식은 `new` 한정자를 사용하여 동일한 이름의 상속된 속성을 가리는 속성을 선언하며, 이는 Visual Basic에서 `Shadows` 한정자를 사용합니다. 모호성을 해결하려면 메서드 오버로드를 GetProperty(String, BindingFlags) 사용하고 플래그를 BindingFlags.DeclaredOnly 추가하여 상속되지 않은 멤버로 검색을 제한합니다.

GetProperty(String, BindingFlags) 메서드

다음 BindingFlags 필터 플래그를 사용하여 검색에 포함할 속성을 정의할 수 있습니다.

- 반환을 얻으려면 `BindingFlags.Instance` 또는 `BindingFlags.Static` 중 하나를 지정해야 합니다.
- 검색에 공용 속성을 포함하도록 지정 `BindingFlags.Public` 합니다.
- 검색에 public이 아닌 속성(즉, 프라이빗, 내부 및 보호된 속성)을 포함하도록 지정 `BindingFlags.NonPublic` 합니다.

- 위계에서 `BindingFlags.FlattenHierarchy` 및 `public` 정적 멤버를 포함하도록 `protected`을(를) 지정하세요. 상속된 클래스에서는 `private` 정적 멤버가 포함되지 않습니다.

다음 BindingFlags 한정자 플래그를 사용하여 검색 작동 방식을 변경할 수 있습니다.

- `BindingFlags.IgnoreCase`의 대소문자를 무시합니다.
- `BindingFlags.DeclaredOnly` 단순히 상속된 속성을 검색하지 않고 Type에 선언된 속성만 검색합니다.

발생하는 상황에는 AmbiguousMatchException 다음이 포함됩니다.

- 형식에는 이름이 같지만 매개 변수 수가 다른 두 개의 인덱싱된 속성이 포함됩니다. 모호성을 해결하려면 매개 변수 형식을 지정하는 메서드의 GetProperty 오버로드를 사용합니다.
- 파생된 형식은 `new` 한정자(`Shadows` Visual Basic)를 사용하여 동일한 이름의 상속된 속성을 숨기는 속성을 선언합니다. 모호성을 해결하려면 상속되지 않은 멤버로 검색을 제한하도록 포함합니다 BindingFlags.DeclaredOnly .

GetProperty(System.String, System.Reflection.BindingFlags, System.Reflection.Binder, System.Type, System.Type[], System.Reflection.ParameterModifier[]) 메서드

기본 바인더는 ParameterModifier를 처리하지 않지만 (`modifiers` 매개 변수) 추상 System.Reflection.Binder 클래스를 사용하여 `modifiers`를 처리하는 사용자 지정 바인더를 작성할 수 있습니다. `ParameterModifier` 는 COM interop을 통해 호출할 때만 사용되며 참조로 전달되는 매개 변수만 처리됩니다.

다음 표에서는 형식을 반영할 때 메서드에서 반환되는 기본 클래스의 `Get` 멤버를 보여 줍니다.

테이블 확장

멤버 형식	정적인	비정적
생성자	아니오	아니오
분야	아니오	예. 필드는 항상 이름과 서명을 기준으로 숨겨집니다.

멤버 형식	정적인	비정적
이벤트	해당 없음	일반적인 형식 시스템 규칙은 상속이 속성을 구현하는 메서드의 상속과 동일하다는 것입니다. 반영은 속성을 이름 및 서명에 따라 숨김 처리합니다. [2]
메서드	아니오	예. 메서드(가상 및 가상이 아닌 메서드)는 이름으로 숨기기 또는 이름과 서명으로 숨기기가 가능합니다.
중첩 타입	아니오	아니오
재산	해당 없음	일반적인 형식 시스템 규칙은 상속이 속성을 구현하는 메서드의 상속과 동일하다는 것입니다. 반영은 속성을 이름 및 서명에 따라 숨김 처리합니다. [2]

노트:

- 이름별 및 서명에 의한 숨기기는 사용자 지정 한정자, 반환 형식, 매개 변수 형식, 센티넬 및 비관리 호출 규칙을 포함하여 서명의 모든 부분을 고려합니다. 이진 비교입니다.
- 리플렉션의 경우 속성과 이벤트는 이름 및 서명별로 숨겨집니다. 기본 클래스에 get 및 set 접근자가 모두 있는 속성이 있지만 파생 클래스에 get 접근자만 있는 경우 파생 클래스 속성은 기본 클래스 속성을 숨기며 기본 클래스의 setter에 액세스할 수 없습니다.
- 사용자 지정 특성은 공용 형식 시스템의 일부가 아닙니다.

다음 BindingFlags 필터 플래그를 사용하여 검색에 포함할 속성을 정의할 수 있습니다.

- 반환을 얻으려면 `BindingFlags.Instance` 또는 `BindingFlags.Static` 중 하나를 지정해야 합니다.
- 검색에 공용 속성을 포함하도록 지정 `BindingFlags.Public` 합니다.
- 검색에 public이 아닌 속성(즉, 프라이빗, 내부 및 보호된 속성)을 포함하도록 지정 `BindingFlags.NonPublic` 합니다.
- 위계에서 `BindingFlags.FlattenHierarchy` 및 `public` 정적 멤버를 포함하도록 `protected`을(를) 지정하세요. 상속된 클래스에서는 `private` 정적 멤버가 포함되지 않습니다.

다음 BindingFlags 한정자 플래그를 사용하여 검색 작동 방식을 변경할 수 있습니다.

- `BindingFlags.IgnoreCase`의 대소문자를 무시합니다.
- `BindingFlags.DeclaredOnly` 단순히 상속된 속성을 검색하지 않고 Type에 선언된 속성만 검색합니다.

인덱서 및 기본 속성

Visual Basic, C# 및 C++에는 인덱싱된 속성에 액세스하기 위한 구문이 간소화되었으며 인덱싱된 속성 하나가 해당 형식의 기본값이 되도록 허용합니다. 예를 들어 변수 `myList`가 ArrayList를

참조하는 경우, Visual Basic 에서 `myList[3]` (`myList(3)` 사용)의 구문은 인덱스가 3인 요소를 검색합니다. 속성을 오버로드할 수 있습니다.

C#에서 이 기능을 인덱서라고 하며 이름으로 참조할 수 없습니다. 기본적으로 C# 인덱서는 메타데이터에 이름이 `Item`인 인덱싱된 속성으로 나타납니다. 그러나 클래스 라이브러리 개발자는 특성을 사용하여 IndexerNameAttribute 메타데이터에서 인덱서의 이름을 변경할 수 있습니다. 예를 들어, String 클래스에는 Chars[]이라는 이름의 인덱서가 있습니다. C# 이외의 언어를 사용하여 만든 인덱싱된 속성에는 다른 `Item`이름도 있을 수 있습니다.

형식에 기본 속성이 있는지 확인하려면 GetCustomAttributes(Type, Boolean) 메서드를 사용하여 DefaultMemberAttribute 특성을 테스트합니다. 형식 DefaultMemberAttribute이 있는 경우 속성은 MemberName 기본 속성의 이름을 반환합니다.

System.Type.GetType 메서드

아티클 • 2025. 04. 09.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

GetType(String, Func<AssemblyName,Assembly>, Func<Assembly,String,Boolean,Type>, Boolean, Boolean) 메서드 오버로드 및 관련 오버로드(GetType(String, Func<AssemblyName,Assembly>, Func<Assembly,String,Boolean,Type>)및GetType(String, Func<AssemblyName,Assembly>, Func<Assembly,String,Boolean,Type>, Boolean))를 사용하여 메서드의 GetType 기본 구현을 보다 유연한 구현으로 바꿉니다. 형식 이름과 형식을 포함하는 어셈블리의 이름을 확인하는 고유한 메서드를 제공하여 다음을 수행할 수 있습니다.

- 어셈블리의 특정 버전에서 형식을 로드하도록 선택합니다.
- 어셈블리 이름을 포함하지 않는 형식 이름을 찾을 다른 위치를 제공합니다.
- 부분 어셈블리 이름을 사용하여 어셈블리를 로드합니다.
- CLR(공용 언어 런타임)에서 생성되지 않은 하위 클래스 System.Type 를 반환합니다.

예를 들어 버전 허용 직렬화에서 이 메서드를 사용하면 부분 이름을 사용하여 "최적" 어셈블리를 검색할 수 있습니다. 메서드의 GetType 다른 오버로드에는 버전 번호를 포함하는 어셈블리 정규화된 형식 이름이 필요합니다.

형식 시스템의 대체 구현은 CLR에서 생성되지 않은 서브클래스를 System.Type 반환해야 할 수 있습니다. 메서드의 GetType 다른 오버로드에서 반환되는 모든 형식은 런타임 형식입니다.

사용 현황 정보

이 메서드 오버로드 및 관련된 오버로드들은 `typeName`를 구문 분석하여 형식 이름과 어셈블리 이름으로 변환한 후, 해당 이름을 확인합니다. 어셈블리의 컨텍스트에서 형식 이름을 확인해야 하므로 형식 이름을 확인하기 전에 어셈블리 이름 확인이 수행됩니다.

> **참고**
>
> 어셈블리 정규화된 형식 이름의 개념에 익숙하지 않은 경우 "AssemblyQualifiedName" 속성을 참조하십시오.

`typeName`이 어셈블리 정규화된 이름이 아닐 경우, 어셈블리 확인이 건너뛰어집니다. 정규화되지 않은 형식 이름은 mscorlib.dll/System.Private.CoreLib.dll 또는 현재 실행 중인 어셈블리의 컨텍스트에서 확인하거나 필요에 따라 매개 변수에 `typeResolver` 어셈블리를 제공할 수 있습니다. 다양한 종류의 이름 확인에 대한 어셈블리 이름을 포함하거나 생략하면 혼합 이름 확인 섹션에 테이블로 표시됩니다.

일반 사용 현황 정보:

- 메서드를 `assemblyResolver` `typeResolver` 알 수 없거나 신뢰할 수 없는 호출자에게 전달하지 마세요. 사용자가 제공하거나 익숙한 메서드만 사용합니다.

  > **⊗ 주의**
  >
  > 알 수 없거나 신뢰할 수 없는 호출자의 메서드를 사용하면 악성 코드에 대한 권한 상승이 발생할 수 있습니다.

- 및/또는 `typeResolver` 매개 변수를 `assemblyResolver` 생략하면 매개 변수 값 `throwOnError` 이 기본 해상도를 수행하는 메서드에 전달됩니다.

- `throwOnError`가 `true`인 경우, `typeResolver`가 `null`을 반환하면 TypeLoadException을 throw하고, `assemblyResolver`가 `null`을 반환하면 FileNotFoundException을 throw합니다.

- 이 메서드는 `assemblyResolver` 및 `typeResolver`에서 발생한 예외를 catch하지 않습니다. 해결 프로그램 메서드에 의해 throw되는 모든 예외에 대한 책임이 있습니다.

어셈블리 해결

메서드는 `typeName`에 포함된 문자열 어셈블리 이름을 구문 분석하여 생성된 AssemblyName 개체를 `assemblyResolver`로 받습니다. `typeName`에 어셈블리 이름이 없는 경우, `assemblyResolver`은 호출되지 않고 `null`가 `typeResolver`에 전달됩니다.

`assemblyResolver` 제공되지 않으면 표준 어셈블리 검색을 사용하여 어셈블리를 찾습니다. 만약 `assemblyResolver`가 제공된다면, GetType 메서드는 표준 탐색을 수행하지 않습니다. 이 경우 당신의 `assemblyResolver`가 전달한 모든 어셈블리를 처리할 수 있도록 해야 합니다.

어셈블리를 확인할 수 없는 경우 메서드는 `null`를 반환해야 합니다. `assemblyResolver`가 `null`를 반환하면 `typeResolver`는 호출되지 않고 추가 처리가 발생하지 않습니다. 또한, `throwOnError`가 `true`이면 FileNotFoundException가 발생합니다.

AssemblyName가 `assemblyResolver`에 전달될 때 부분 이름일 경우, 그 부분들 중 하나 이상은 `null`입니다. 예를 들어 버전이 없는 경우 속성은 Version .입니다 `null`. Version 속성, CultureInfo 속성 및 메서드가 GetPublicKeyToken 모두 반환`null`되면 어셈블리의 단순 이름만 제공되었습니다. 메서드는 `assemblyResolver` 어셈블리 이름의 모든 부분을 사용하거나 무시할 수 있습니다.

다양한 어셈블리 확인 옵션의 효과는 단순 및 어셈블리 정규화된 형식 이름에 대한 혼합 이름 확인 섹션에 테이블로 표시됩니다.

유형 해결

`typeName` 어셈블리 이름을 `typeResolver` 지정하지 않으면 항상 호출됩니다. 어셈블리 이름을 `typeName`에서 지정하면, 어셈블리 이름이 성공적으로 확인된 경우에만 `typeResolver`이 호출됩니다. 표준 어셈블리 검색이 반환 `null` `typeResolver` 되는 경우 `assemblyResolver` 호출되지 않습니다.

메서드는 `typeResolver` 세 가지 인수를 받습니다.

- 검색할 어셈블리이거나 `null` 어셈블리 이름이 없는 경우 `typeName`
- 형식의 단순 이름입니다. 중첩된 형식의 경우 가장 바깥쪽에 포함된 형식입니다. 제네릭 형식의 경우 제네릭 형식의 단순 이름입니다.
- 형식 이름의 대/소문자를 무시해야 하는 `true` 경우의 부울 값입니다.

구현은 이러한 인수를 사용하는 방법을 결정합니다. 형식을 `typeResolver` 해결할 수 없는 경우 메서드는 `null`을 반환해야 합니다. `typeResolver`가 `null`을 반환하고 `throwOnError`가 `true`일 경우, 이 GetType 오버로드는 TypeLoadException를 발생시킵니다.

다양한 형식 확인 옵션의 효과는 단순 및 어셈블리 정규화된 형식 이름에 대한 혼합 이름 확인 섹션의 테이블로 표시됩니다.

중첩 형식 해결

중첩 형식인 경우 `typeName` 가장 바깥쪽에 포함된 형식의 이름만 에 전달됩니다 `typeResolver`. `typeResolver` 이 형식을 반환하면 가장 안쪽에 GetNestedType 중첩된 형식이 확인될 때까지 메서드가 재귀적으로 호출됩니다.

제네릭 형식 해결

제 GetType 네릭 형식을 확인하기 위해 재귀적으로 호출됩니다. 먼저 제네릭 형식 자체를 확인하고 해당 형식 인수를 확인합니다. 형식 인수가 제네릭인 경우, 형식 인수를 해결하기 위해 GetType가 재귀적으로 호출됩니다.

사용자가 제공하는 조합 `assemblyResolver` `typeResolver` 은 이 재귀의 모든 수준을 해결할 수 있어야 합니다. 예를 들어, `assemblyResolver`를 제공하여 `MyAssembly`의 로딩을 제어한다고 가정해 보십시오. 제네릭 형식 `Dictionary<string, MyType>`(Visual Basic에서 `Dictionary(Of String, MyType)`)을 해결하려고 가정해 보겠습니다. 다음 제네릭 형식 이름을 전달할 수 있습니다.

```
"System.Collections.Generic.Dictionary`2[System.String,[MyNamespace.MyType,
MyAssembly]]"
```

`MyType` 유일한 어셈블리 정규화된 형식 인수입니다. 클래스 Dictionary<TKey,TValue>와 String의 이름은 어셈블리로 한정되지 않았습니다. `typeResolver`는 어셈블리 또는 `null`을 처리할 수 있어야 합니다. 왜냐하면 Dictionary<TKey,TValue> 및 String용 `null`를 수신할 것이기 때문입니다. 정규화되지 않은 형식 이름이 모두 mscorlib.dll/System.Private.CoreLib.dll에 있으므로 문자열을 사용하는 GetType 메서드의 오버로드를 호출하여 이 경우를 처리할 수 있습니다.

C#

```
Type t = Type.GetType(test,
                      (aName) => aName.Name == "MyAssembly" ?
                          Assembly.LoadFrom(@".\MyPath\v5.0\MyAssembly.dll") :
null,
                      (assem, name, ignore) => assem == null ?
                          Type.GetType(name, false, ignore) :
                              assem.GetType(name, false, ignore)
                     );
```

`assemblyResolver` 이러한 형식 이름은 어셈블리로 한정되지 않으므로 사전 형식 및 문자열 형식에 대해 메서드가 호출되지 않습니다.

이제 `System.String` 대신에, 첫 번째 제네릭 인수 형식이 `YourType`이며, 이것이 `YourAssembly`로부터 온다고 가정해 봅시다.

```
"System.Collections.Generic.Dictionary`2[[YourNamespace.YourType, YourAssembly,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null], [MyNamespace.MyType,
MyAssembly]]"
```

이 어셈블리가 mscorlib.dll/System.Private.CoreLib.dll도 아니고 현재 실행 중인 어셈블리도 아니기 때문에, 어셈블리 정규화된 이름 없이는 `YourType`를 확인할 수 없습니다. `assemblyResolve` 재귀적으로 호출되므로 이 경우를 처리할 수 있어야 합니다. 이제 `MyAssembly` 이외의 다른 어셈블리에 대해 `null`을 반환하는 대신 제공된 AssemblyName 개체를 사용하여 어셈블리 로드를 수행합니다.

C#

```
Type t2 = Type.GetType(test,
                       (aName) => aName.Name == "MyAssembly" ?
                           Assembly.LoadFrom(@".\MyPath\v5.0\MyAssembly.dll") :
                           Assembly.Load(aName),
                       (assem, name, ignore) => assem == null ?
                           Type.GetType(name, false, ignore) :
                               assem.GetType(name, false, ignore), true
                      );
```

특수 문자를 사용하여 형식 이름 확인

특정 문자는 어셈블리로 한정된 이름에 특별한 의미를 갖습니다. 단순 형식 이름에 이러한 문자가 포함된 경우 단순 이름이 어셈블리 정규화된 이름의 일부일 때 문자 구문 분석 오류가 발생합니다. 구문 분석 오류를 방지하려면 GetType 메서드에 어셈블리 정규화된 이름을 전달하기 전에 특수 문자를 백슬래시로 이스케이프해야 합니다. 예를 들어 형식의 이름이 지정 `Strange]Type`되면 다음과 `Strange\]Type`같이 대괄호 앞에 이스케이프 문자를 추가해야 합니다.

> **참고**
>
> 이러한 특수 문자를 가진 이름은 Visual Basic 또는 C#에서 만들 수 없지만 CIL(공용 중간 언어)을 사용하거나 동적 어셈블리를 내보내서 만들 수 있습니다.

다음 표에는 형식 이름의 특수 문자가 표시됩니다.

테이블 확장

캐릭터	의미
`,` (쉼표)	어셈블리 한정 이름의 구분 기호입니다.
`[]` (대괄호)	접미사 쌍으로 배열 형식을 나타냅니다. 구분 기호 쌍으로 제네릭 인수 목록과 어셈블리 정규화된 이름을 묶습니다.
`&`(앰퍼샌드)	접미사로 형식이 참조 형식임을 나타냅니다.
`*` (별표)	접미사로 형식이 포인터 형식임을 나타냅니다.
`+` (더하기)	중첩된 형식의 구분 기호입니다.
`\` (백슬래시)	이스케이프 문자입니다.

올바르게 이스케이프된 문자열 반환과 같은 AssemblyQualifiedName 속성입니다. 메서드에 올바르게 이스케이프된 문자열을 GetType 전달해야 합니다. 이 메서드는 GetType 이스케이프된 이름을 기본 형식 확인 메서드와 `typeResolver` 올바르게 전달합니다. 이름을 이스케이프되지 않은 이름과 `typeResolver`비교해야 하는 경우 이스케이프 문자를 제거해야 합니다.

혼합 이름 확인

다음 표는 `typeName`에서 형식 이름과 어셈블리 이름의 모든 조합에 대해 `assemblyResolver`와 `typeResolver` 및 기본 이름 확인 간의 상호 작용을 요약합니다.

테이블 확장

형식 이름의 내용	어셈블리 해결자 메서드	형식 해결자 메서드	결과
유형, 조립	null	null	메서드 오버로드를 호출하는 Type.GetType(String, Boolean, Boolean) 것과 같습니다.
유형, 어셈블리	제공	null	`assemblyResolver` 는 어셈블리를 반환하거나 어셈블리를 확인할 수 없는 경우 `null` 반환합니다. 어셈블리가 해석되면, Assembly.GetType(String, Boolean, Boolean) 메서드 오버로드를 사용하여 어셈블리에서 형식을 로드하고, 그렇지 않으면 형식을 해석하려고 시도하지 않습니다.
유형, 조립	null	제공	어셈블리 이름을 AssemblyName 개체로 변환하고 메서드 오버로드를 Assembly.Load(AssemblyName) 호출하여 어셈블리를 가져오는 것과 같습니다. 어셈블리가 확인되면 `typeResolver`에 전달됩니다. 그렇지 않으면 `typeResolver` 이 호출되지 않고 형식을 더 이상 확인하려고 시도하지 않습니다.
유형, 어셈블리	제공	제공	`assemblyResolver` 는 어셈블리를 반환하거나, 어셈블리를 확인할 수 없는 경우 `null` 를 반환합니다. 어셈블리가 확인되면 `typeResolver`로 전달되고, 그렇지 않으면 `typeResolver`가 호출되지 않으며 형식을 더 이상 확인하려고 시도하지 않습니다.
유형	null, 제공됨	null	Type.GetType(String, Boolean, Boolean) 메서드 오버로드를 호출하는 것과 동일합니다. 어셈블리 이름이 제공되지 않으므로 mscorlib.dll/System.Private.CoreLib.dll 현재 실행 중인 어셈블리만 검색됩니다. 제공된 경우 `assemblyResolver` 무시됩니다.
유형	null, 제공됨	제공	`typeResolver` 가 호출되고 `null` 어셈블리에 전달됩니다. `typeResolver` 는 특정 목적을 위해 로드된 어셈블리를 포함하여, 모든 어셈블리에서 형식을 제공할 수 있습니다. 제공된 경우 `assemblyResolver` 무시됩니다.
모임	null, 제공됨	null, 제공됨	어셈블리 이름이 어셈블리 정규화된 형식 이름인 것처럼 구문 분석되기 때문에 FileLoadException이(가) throw됩니다. 이로 인해 어셈블리 이름이 유효하지 않습니다.

System.Type.MakeGenericType 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 MakeGenericType 메서드를 사용하면 특정 형식을 제네릭 형식 정의의 형식 매개 변수에 할당하는 코드를 작성하여 생성된 특정 형식을 Type 나타내는 개체를 만들 수 있습니다. 이 Type 개체를 사용하여 생성된 형식의 런타임 인스턴스를 만들 수 있습니다.

생성된 MakeGenericType 형식은 열 수 있습니다. 즉, 해당 형식 인수 중 일부는 제네릭 메서드 또는 형식을 묶는 형식 매개 변수일 수 있습니다. 동적 어셈블리를 내보낸 경우 이러한 개방형 생성 형식을 사용할 수 있습니다. 예를 들어, 다음 코드에서 클래스 `Base`와 `Derived`을 고려하십시오.

C#

```
public class Base<T, U> { }
public class Derived<V> : Base<int, V> { }
```

동적 어셈블리에서 생성 `Derived` 하려면 기본 형식을 생성해야 합니다. 이렇게 하려면, MakeGenericType 클래스를 나타내는 Type 객체에서 `Base` 메서드를 호출할 때, Int32의 제네릭 형식 인수 `V`와 형식 매개변수 `Derived`를 사용하십시오. 형식과 제네릭 형식 매개 변수는 모두 개체로 Type 표현되므로 두 매개 변수를 모두 포함하는 배열을 메서드에 MakeGenericType 전달할 수 있습니다.

> ⓘ **참고**
>
> 생성된 `Base<int, V>` 형식은 코드를 내보낼 때 유용하지만 제네릭 형식 정의가 아니므로 이 형식에서 메서드를 호출 MakeGenericType 할 수 없습니다. 인스턴스화할 수 있는 닫힌 생성된 형식을 만들려면 먼저 메서드를 호출 GetGenericTypeDefinition 하여 Type 제네릭 형식 정의를 나타내는 개체를 만든 다음 원하는 형식 인수를 사용하여 호출 MakeGenericType 합니다.

Type에 의해 반환된 개체는 결과로 생성된 형식의 MakeGenericType 메서드를 호출하여 얻은 Type와 동일하며, 또는 동일한 형식 인수를 사용하여 동일한 제네릭 형식 정의로부터 생성된 다른 생성된 형식의 GetType 메서드를 호출하여도 동일합니다.

> ⓘ **참고**
>
> 제네릭 형식의 배열 자체는 제네릭 형식이 아닙니다. (Visual Basic에서)MakeGenericType와 같은 `C<T>[]` 배열 형식을 호출 `Dim ac() As C(Of T)` 할 수 없습니다. 닫힌 제네릭 형식을

`C<T>[]` 생성하려면 제네릭 형식 정의를 GetElementType가져오려면 호출 `C<T>` 하고, 제네릭 형식 정의를 호출 MakeGenericType 하여 생성된 형식을 만들고, 마지막으로 생성된 형식의 메서드를 호출 MakeArrayType 하여 배열 형식을 만듭니다. 포인터 형식 및 `ref` 형식도 마찬가지입니다(`ByRef` Visual Basic의 경우).

제네릭 리플렉션에 사용되는 용어에 대한 고정 조건 목록은 속성 비고를 IsGenericType 참조하세요.

중첩 형식

제네릭 형식이 C#, C++또는 Visual Basic을 사용하여 정의된 경우 중첩된 형식은 모두 제네릭입니다. 세 언어 모두 중첩 형식의 형식 매개 변수 목록에 바깥쪽 형식의 형식 매개 변수를 포함하므로 중첩된 형식에 고유한 형식 매개 변수가 없는 경우에도 마찬가지입니다. 다음 클래스를 고려합니다.

C#

```
public class Outermost<T>
{
    public class Inner<U>
    {
        public class Innermost1<V> {}
        public class Innermost2 {}
    }
}
```

중첩 클래스 `Inner` 의 형식 매개 변수 목록에는 두 개의 형식 매개 변수가 있으며`T`, `U` 그 중 첫 번째는 바깥쪽 클래스의 형식 매개 변수입니다. 마찬가지로, 중첩 클래스 `Innermost1`의 형식 매개변수 목록에는 세 개의 형식 매개변수 `T`, `U`, `V`가 있으며, 이 중 `T`와 `U`는 바깥쪽 클래스에서 유래합니다. 중첩된 클래스 `Innermost2` 에는 두 개의 형식 매개 변수가 있으며`T`, `U` 이 매개 변수는 바깥쪽 클래스에서 가져옵니다.

포함된 형식의 매개 변수 목록에 둘 이상의 형식 매개 변수가 있는 경우, 모든 형식 매개 변수가 순서에 따라 중첩된 형식의 매개 변수 목록에 포함됩니다.

중첩된 형식에 대한 제네릭 형식 정의에서 제네릭 형식을 생성하려면 가장 바깥쪽 제네릭 형식부터 시작하여 모든 바깥쪽 형식의 형식 인수 배열을 연결하고 자체 형식 매개 변수가 있는 경우 중첩된 형식 자체의 형식 인수 배열로 끝나는 배열을 사용하여 메서드를 호출 MakeGenericType 합니다. 인스턴스 `Innermost1`를 만들려면 T, U 및 V에 할당할 세 가지 형식을 포함하는 배열을 사용하여 메서드를 호출 MakeGenericType 합니다. 인스턴스 `Innermost2`를 만들려면 T 및 U에 할당할 두 가지 형식이 포함된 배열을 사용하여 메서드를 호출 MakeGenericType 합니다.

언어는 바깥쪽 형식의 형식 매개 변수를 사용하여 중첩된 형식의 필드를 정의할 수 있도록 이러한 방식으로 바깥쪽 형식의 형식 매개 변수를 전파합니다. 그렇지 않으면 형식 매개 변수가 중첩된 형식의 본문 내에서 범위에 포함되지 않습니다. 동적 어셈블리에서 코드를 내보내거나 IL 어셈블러(Ilasm.exe)를 사용하여 바깥쪽 형식의 형식 매개 변수를 전파하지 않고 중첩된 형식을 정의할 수 있습니다. CIL 어셈블러에 대해 다음 코드를 고려합니다.

msil

```
.class public Outer<T> {
    .class nested public Inner<U> {
        .class nested public Innermost {
        }
    }
}
```

이 예제에서는 형식 매개 변수가 범위에 없기 때문에 형식 `T` 또는 `U` 클래스 `Innermost`의 필드를 정의할 수 없습니다. 다음 어셈블러 코드는 C++, Visual Basic 및 C#에 정의된 경우처럼 동작하는 중첩 클래스를 정의합니다.

msil

```
.class public Outer<T> {
    .class nested public Inner<T, U> {
        .class nested public Innermost<T, U, V> {
        }
    }
}
```

Ildasm.exe(IL 디스어셈블러)를 사용하여 상위 수준 언어로 정의된 중첩 클래스를 검사하고 이 명명 체계를 관찰할 수 있습니다.

System.Reflection.Context.CustomReflectionContext 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

CustomReflectionContext 에서는 전체 리플렉션 모델을 다시 구현하지 않고 리플렉션 개체에서 사용자 지정 특성을 추가하거나 제거하거나 해당 개체에 더미 속성을 추가할 수 있는 방법을 제공합니다. 기본값 CustomReflectionContext 은 단순히 리플렉션 개체를 변경하지 않고 래핑하지만 관련 메서드를 서브클래싱 및 재정의하여 반사된 매개 변수 또는 멤버에 적용되는 특성을 추가, 제거 또는 변경하거나, 반영된 형식에 새 속성을 추가할 수 있습니다.

예를 들어 코드가 팩터리 메서드에 특정 특성을 적용하는 규칙을 따르지만 이제 특성이 없는 타사 코드를 사용해야 하는 경우를 가정해 보겠습니다. 특성이 있어야 하는 개체를 식별하고 코드에서 볼 때 해당 특성을 포함하는 개체를 제공하는 규칙을 지정할 수 CustomReflectionContext 있습니다.

효과적으로 사용 CustomReflectionContext 하려면 반사된 개체를 사용하는 코드는 반사된 모든 개체가 런타임 리플렉션 컨텍스트와 연결되어 있다고 가정하는 대신 리플렉션 컨텍스트를 지정하는 개념을 지원해야 합니다. .NET Framework의 많은 리플렉션 메서드는 이 목적을 위한 매개 변수를 ReflectionContext 제공합니다.

반영된 매개 변수 또는 멤버에 적용되는 특성을 수정하려면 GetCustomAttributes(ParameterInfo, IEnumerable<Object>) 메서드나 GetCustomAttributes(MemberInfo, IEnumerable<Object>) 메서드를 재정의하십시오. 이러한 메서드는 현재 리플렉션 컨텍스트에서 반사된 개체와 특성 목록을 가져와서 사용자 지정 리플렉션 컨텍스트 아래에 있어야 하는 특성 목록을 반환합니다.

> ⚠ **Warning**
>
> CustomReflectionContext메서드는 제공된 MemberInfo 또는 ParameterInfo 인스턴스에서 GetCustomAttributes 메서드를 호출하여 반사된 객체 또는 메서드의 특성 목록에 직접 액세스해서는 안 되며, 대신 GetCustomAttributes 메서드 오버로드에 매개변수로 전달되는 `declaredAttributes` 목록을 사용해야 합니다.

반영된 형식에 속성을 추가하려면 AddProperties 메서드를 재정의합니다. 이 메서드는 반사된 형식을 지정하는 매개 변수를 허용하고 추가 속성 목록을 반환합니다. 속성 개체를 반환하려면 CreateProperty 메서드를 사용해야 합니다. 속성 접근자 역할을 하는 속성을 만들 때 대리자를

지정할 수 있으며 접근자 중 하나를 생략하여 읽기 전용 또는 쓰기 전용 속성을 만들 수 있습니다. 이러한 더미 속성에는 메타데이터 또는 CIL(공용 중간 언어) 백업이 없습니다.

> ⚠ **Warning**
>
> - 반사 컨텍스트를 사용할 때는 개체가 복수의 컨텍스트에서 동일한 반사 개체를 나타낼 수 있으므로 반사된 개체 간의 동일성에 주의를 기울여야 합니다. 이 메서드를 MapType 사용하여 특정 리플렉션 컨텍스트의 리플렉션된 개체 버전을 가져올 수 있습니다.
> - CustomReflectionContext 객체는 GetCustomAttributes 메서드를 통해 얻은 특성과 같은 특정 리플렉션 객체에서 반환된 특성을 수정합니다. 메서드에서 반환 GetCustomAttributesData 된 사용자 지정 특성 데이터는 변경되지 않으며 사용자 지정 리플렉션 컨텍스트를 사용하는 경우 이러한 두 목록이 일치하지 않습니다.

예시

다음 예제에서는 이름이 "To"로 시작하는 지정된 형식의 모든 멤버에 사용자 지정 특성을 추가하도록 서브클래스 CustomReflectionContext 하는 방법을 보여 줍니다. 이 코드를 실행하려면 빈 콘솔 프로젝트에 붙여넣고 NuGet 패키지에 대한 참조를 `System.Reflection.Context` 추가합니다.

C#

```
//A blank example attribute.
class MyAttribute : Attribute
{
}

//Reflection context with custom rules.
class MyCustomReflectionContext : CustomReflectionContext
{
    //Called whenever the reflection context checks for custom attributes.
    protected override IEnumerable<object> GetCustomAttributes(MemberInfo member,
IEnumerable<object> declaredAttributes)
    {
        //Add example attribute to "To*" members.
        if (member.Name.StartsWith("To"))
        {
            yield return new MyAttribute();
        }
        //Keep existing attributes as well.
        foreach (var attr in declaredAttributes) yield return attr;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        MyCustomReflectionContext mc = new MyCustomReflectionContext();
        Type t = typeof(String);

        //A representation of the type in the default reflection context.
        TypeInfo ti = t.GetTypeInfo();

        //A representation of the type in the customized reflection context.
        TypeInfo myTI = mc.MapType(ti);

        //Display all the members of the type and their attributes.
        foreach (MemberInfo m in myTI.DeclaredMembers)
        {
            Console.WriteLine(m.Name + ":");
            foreach (Attribute cd in m.GetCustomAttributes())
            {
                Console.WriteLine(cd.GetType());
            }
        }

        Console.WriteLine();

        //The "ToString" member as represented in the default reflection context.
        MemberInfo mi1 = ti.GetDeclaredMethods("ToString").FirstOrDefault();

        //All the attributes of "ToString" in the default reflection context.
        Console.WriteLine("'ToString' Attributes in Default Reflection Context:");
        foreach (Attribute cd in mi1.GetCustomAttributes())
        {
            Console.WriteLine(cd.GetType());
        }

        Console.WriteLine();

        //The same member in the custom reflection context.
        mi1 = myTI.GetDeclaredMethods("ToString").FirstOrDefault();

        //All its attributes, for comparison.  MyAttribute is now included.
        Console.WriteLine("'ToString' Attributes in Custom Reflection Context:");
        foreach (Attribute cd in mi1.GetCustomAttributes())
        {
            Console.WriteLine(cd.GetType());
        }

        Console.ReadLine();
    }
}
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

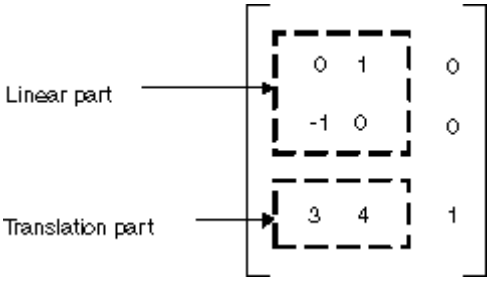
Last updated on 2026. 02. 13.

System.Drawing.Drawing2D.Matrix 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스는 Matrix 기하학적 변환을 나타내는 3-by-3 아핀 행렬을 캡슐화합니다.

GDI+에서는 Matrix 개체에 아핀 변환을 저장할 수 있습니다. 3단 변환을 나타내는 행렬의 세 번째 열은 항상 (0, 0, 1)이므로, Matrix 개체를 생성할 때는 처음 두 열의 6개 숫자만 지정합니다. 이 문 `Matrix myMatrix = new Matrix(0, 1, -1, 0, 3, 4)` 은 다음 그림에 표시된 행렬을 생성합니다.



> ⓘ **참고 항목**
>
> .NET 6 이상 버전에서는 이 형식을 포함하는 System.Drawing.Common 패키지 가 Windows 운영 체제에서만 지원됩니다. 플랫폼 간 앱에서 이 형식을 사용하면 컴파일 시간 경고 및 런타임 예외가 발생합니다. 자세한 내용은 Windows에서만 지원되는 System.Drawing.Common을 참조 하세요.

복합 변환

복합 변환은 변환의 시퀀스이며, 그 뒤에 다른 변환이 잇습니다. 다음 목록에서 행렬 및 변환을 고려합니다.

테이블 확장

매트릭스	변형
행렬 A	90도 회전
행렬 B	x 방향에서 2 배율로 크기 조정
행렬 C	y 방향으로 3단위 이동

행렬 [2 1 1]로 표현되는 점(2, 1)으로 시작하고 A, B, C를 곱하면 점(2, 1)은 나열된 순서대로 세 가지 변환을 거칩니다.

`[2 1 1]ABC = [-2 5 1]`

복합 변환의 세 부분을 별도의 세 행렬에 저장하는 대신, A, B 및 C를 함께 곱하여 전체 복합 변환을 저장하는 단일 3×3 행렬을 가져올 수 있습니다. 를 가정해 보겠습니다 `ABC = D`. 그런 다음 D를 곱한 점이 A, B, C를 곱한 점과 동일한 결과를 제공합니다.

`[2 1 1]D = [-2 5 1]`

다음 그림에서는 행렬 A, B, C 및 D를 보여 줍니다.

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -2 & 0 & 0 \\ 0 & 3 & 1 \end{bmatrix}$$

A B C = D

복합 변환의 행렬이 개별 변환 행렬을 곱하여 형성될 수 있다는 사실은 모든 아핀 변환 시퀀스를 하나의 Matrix 개체에 저장할 수 있음을 의미합니다.

> **⊗ 주의**
>
> 복합 변환의 순서가 중요합니다. 일반적으로 회전한 다음 크기를 조정하고 변환하는 것과 크기를 조정한 다음 회전하고 변환하는 것은 다릅니다. 마찬가지로 행렬 곱셈 순서도 중요합니다. 일반적으로 ABC는 BAC와 동일하지 않습니다.

Matrix 클래스는 복합 변환을 생성하기 위한 여러 메서드인 Multiply, Rotate, RotateAt, Scale, Shear, 그리고 Translate를 제공합니다. 다음 예제에서는 먼저 30도 회전한 다음, y 방향으로 배율 2로 스케일링한 다음, x 방향으로 5 단위를 이동하는 복합 변환의 행렬을 만듭니다.

C#

```
Matrix myMatrix = new Matrix();
myMatrix.Rotate(30);
myMatrix.Scale(1, 2, MatrixOrder.Append);
myMatrix.Translate(5, 0, MatrixOrder.Append);
```

Last updated on 2026. 02. 12.

System.Runtime.CompilerServices.InternalsVisibleToAttribute 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

특성은 InternalsVisibleToAttribute 현재 어셈블리 내에서만 일반적으로 표시되는 형식이 지정된 어셈블리에 표시되도록 지정합니다.

일반적으로 C#의 범위 또는 internal Visual Basic의 범위가 있는 형식 및 멤버Friend는 정의된 어셈블리에서만 표시됩니다. protected internal 범위를 갖는 형식 및 멤버는 자신의 어셈블리에서만 보이거나 해당 형식을 포함하는 클래스에서 파생된 형식에만 보입니다 (Protected Friend 범위는 Visual Basic의 경우). 범위가 있는 private protected 형식 및 멤버(Private Protected Visual Basic의 범위)는 포함하는 클래스 또는 현재 어셈블리 내의 포함하는 클래스에서 파생된 형식에 표시됩니다.

InternalsVisibleToAttribute 특성은 해당 형식과 멤버가 friend 어셈블리라고 불리는 지정된 어셈블리의 형식에도 노출되도록 합니다. 이는 `internal` (`Friend` Visual Basic의 경우), `protected internal` (`Protected Friend` Visual Basic의 경우), 및 `private protected` (`Private Protected` Visual Basic의 경우) 멤버에만 적용되지만 `private` 멤버에는 적용되지 않습니다.

> **참고**
>
> `private protected` (`Private Protected` Visual Basic에서) 멤버의 InternalsVisibleToAttribute 특성은 멤버를 포함하는 *클래스*에서 파생된 형식으로만 접근성을 확장합니다.

특성은 어셈블리 수준에서 적용됩니다. 즉, 소스 코드 파일의 시작 부분에 포함되거나 Visual Studio 프로젝트의 AssemblyInfo 파일에 포함될 수 있습니다. 특성을 사용하여 현재 어셈블리의 내부 형식 및 멤버에 액세스할 수 있는 단일 friend 어셈블리를 지정할 수 있습니다. 여러 friend 어셈블리를 두 가지 방법으로 정의할 수 있습니다. 다음 예제와 같이 개별 어셈블리 수준 특성으로 표시할 수 있습니다.

C#

```
[assembly:InternalsVisibleTo("Friend1a")]
[assembly:InternalsVisibleTo("Friend1b")]
```

다음 예제와 같이 별도의 InternalsVisibleToAttribute 태그를 사용하지만 단일 `assembly` 키워드로 표시할 수도 있습니다.

C#

```
[assembly:InternalsVisibleTo("Friend2a"),
          InternalsVisibleTo("Friend2b")]
```

friend 어셈블리는 생성자로 식별됩니다 InternalsVisibleToAttribute . 현재 어셈블리와 friend 어셈블리는 모두 서명되지 않아야 합니다. 그렇지 않으면 두 어셈블리가 모두 강력한 이름으로 서명되어야 합니다.

두 어셈블리가 모두 서명 `assemblyName` 되지 않은 경우 인수는 디렉터리 경로 또는 파일 이름 확장명 없이 지정된 friend 어셈블리의 이름으로 구성됩니다.

두 어셈블리가 모두 강력한 이름으로 서명된 경우 생성자에 대한 InternalsVisibleToAttribute 인수는 디렉터리 경로 또는 파일 이름 확장명 없이 어셈블리의 이름과 공개 키 토큰이 아닌 전체 공개 키로 구성되어야 합니다. 강력한 이름의 어셈블리의 전체 공개 키를 얻으려면 이 문서의 뒷부 분에 있는 전체 공개 키 가져오기 섹션을 참조하세요. 강력한 이름의 어셈블리를 사용하는 InternalsVisibleToAttribute 방법에 대한 자세한 내용은 생성자 InternalsVisibleToAttribute를 참조하세요.

인수에 , CultureInfo또는 Version 필드에 대한 ProcessorArchitecture값을 포함하지 마세요. Visual Basic, C#및 C++ 컴파일러는 이를 컴파일러 오류로 처리합니다. IL 어셈블러(ILAsm.exe)와 같이 이를 오류로 처리하지 않는 컴파일러를 사용하고 어셈블리에 강력한 이름이 지정된 경우, 지정된 friend 어셈블리가 MethodAccessException 특성을 포함하는 어셈블리에 처음으로 액세스할 때 예외가 발생합니다InternalsVisibleToAttribute.

이 특성을 사용하는 방법에 대한 자세한 내용은 Friend 어셈블리 및 C++ friend 어셈블리를 참조하세요.

전체 공개 키 가져오기

강력한 이름 도구(Sn.exe)를 사용하여 강력한 이름의 키(.snk) 파일에서 전체 공개 키를 검색할 수 있습니다. 이렇게 하려면 다음 단계를 수행합니다.

1. 강력한 이름의 키 파일에서 공개 키를 별도의 파일로 추출합니다.

   `Sn -p <snk_file> <outfile>`

2. 콘솔에 전체 공개 키를 표시합니다.

   `Sn -tp <outfile>`

3. 전체 공개 키 값을 복사하여 소스 코드에 붙여넣습니다.

C를 사용하여 friend 어셈블리 컴파일#

C# 컴파일러를 사용하여 friend 어셈블리를 컴파일하는 경우 **/out** 컴파일러 옵션을 사용하여 출력 파일(.exe 또는 .dll)의 이름을 명시적으로 지정해야 합니다. 컴파일러가 외부 참조에 바인딩할 때 빌드 중인 어셈블리의 이름을 아직 생성하지 않았기 때문에 이 작업이 필요합니다. **/out** 컴파일러 옵션은 Visual Basic 컴파일러에 대한 선택 사항이며 F# 컴파일러를 사용하여 friend 어셈블리를 컴파일할 때는 해당 **-out** 또는 **-o** 컴파일러 옵션을 사용하면 안 됩니다.

C++를 사용하여 friend 어셈블리 컴파일

C++에서 InternalsVisibleToAttribute 특성으로 활성화된 내부 멤버를 friend 어셈블리에 액세스할 수 있도록 하려면, C++ 지시문에서 `as_friend` 특성을 사용해야 합니다. 자세한 내용은 Friend 어셈블리(C++)를 참조하세요.

System.Runtime.CompilerServices.RuntimeHelpers.GetHashCode 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

RuntimeHelpers.GetHashCode 메서드는 개체의 형식이 Object.GetHashCode 메서드를 재정의한 경우에도 항상 Object.GetHashCode 메서드를 가상이 아닌 방법으로 호출합니다. 따라서 RuntimeHelpers.GetHashCode 사용은 `GetHashCode` 메서드를 사용하여 개체에서 Object.GetHashCode을 직접 호출하는 것과 다를 수 있습니다.

> ⚠ **경고**
>
> 메서드는 RuntimeHelpers.GetHashCode 동일한 개체 참조에 대해 동일한 해시 코드를 반환하지만 이 해시 코드는 개체 참조를 고유하게 식별하지 않으므로 이 메서드를 사용하여 개체 ID를 테스트하면 안 됩니다. 개체 ID를 테스트하려면(즉, 두 개체가 메모리에서 동일한 개체를 참조하는지 테스트하려면) 메서드를 호출합니다 Object.ReferenceEquals . 문자열이 인턴되어 있기 때문에 두 문자열이 동일한 객체 참조를 나타내는지 테스트하는 데 GetHashCode을 사용해서는 안 됩니다. 문자열 인턴링을 테스트하려면 메서드를 호출합니다 String.IsInterned .

Object.GetHashCode 메서드와 RuntimeHelpers.GetHashCode 메서드는 다음과 같이 다릅니다.

- Object.GetHashCode 는 개체의 같음 정의를 기반으로 하는 해시 코드를 반환합니다. 예를 들어 내용이 동일한 두 문자열은 동일한 값을 Object.GetHashCode반환합니다.
- RuntimeHelpers.GetHashCode 는 개체 ID를 나타내는 해시 코드를 반환합니다. 즉, 내용이 동일하고 인턴된 문자열(문자열 인턴링 섹션 참조) 또는 메모리의 단일 문자열을 나타내는 문자열을 나타내는 두 문자열 변수는 동일한 해시 코드를 반환합니다.

> ⓘ **중요**
>
> GetHashCode 항상 동일한 개체 참조에 대해 동일한 해시 코드를 반환합니다. 그러나 반대의 경우는 true가 아닙니다. 동일한 해시 코드는 동일한 개체 참조를 나타내지 않습니다. 특정 해시 코드 값은 특정 개체 참조에 고유하지 않습니다. 다른 개체 참조는 동일한 해시 코드를 생성할 수 있습니다.

이 메서드는 컴파일러에서 사용됩니다.

문자열 인터닝

CLR(공용 언어 런타임)은 내부 문자열 풀을 유지하고 풀에 리터럴을 저장합니다. 두 문자열(예: `str1` 및 `str2`)이 동일한 문자열 리터럴에서 형성되는 경우 CLR은 메모리를 절약하기 위해 관리되는 힙의 동일한 위치를 설정하고 `str1` `str2` 가리킵니다. 이러한 두 문자열 개체를 호출 RuntimeHelpers.GetHashCode 하면 이전 섹션의 두 번째 글머리 기호 항목과 달리 동일한 해시 코드가 생성됩니다.

CLR은 풀에 리터럴만 추가합니다. 컴파일러가 문자열 연결을 단일 문자열 리터럴로 해결하지 않는 한 연결과 같은 문자열 작업의 결과는 풀에 추가되지 않습니다. 따라서, `str2`가 연결 작업의 결과로 만들어지고 `str2`가 `str1`와 동일한 경우, 이 두 문자열 개체에 대해 RuntimeHelpers.GetHashCode를 사용하면 동일한 해시 코드가 생성되지 않습니다.

연결된 문자열을 풀에 명시적으로 추가하려면 메서드를 String.Intern 사용합니다.

이 메서드를 사용하여 문자열에 String.IsInterned 인턴 참조가 있는지 확인할 수도 있습니다.

예시

다음 예제에서는 Object.GetHashCode 메서드와 RuntimeHelpers.GetHashCode 메서드의 차이점을 보여줍니다. 예제의 출력은 다음을 보여 줍니다.

- 문자열이 완전히 다르기 때문에 메서드에 `ShowHashCodes` 전달된 첫 번째 문자열 집합에 대한 해시 코드 집합은 모두 다릅니다.
- Object.GetHashCode 는 문자열이 같기 때문에 메서드에 `ShowHashCodes` 전달된 두 번째 문자열 집합에 대해 동일한 해시 코드를 생성합니다. 메서드RuntimeHelpers.GetHashCode는 그렇지 않습니다. 첫 번째 문자열은 문자열 리터럴을 사용하여 정의되므로 인턴됩니다. 두 번째 문자열의 값은 동일하지만 메서드 호출 String.Format 에 의해 반환되기 때문에 인턴되지 않습니다.
- 세 번째 문자열의 경우 두 문자열에 대해 Object.GetHashCode 생성된 해시 코드는 으로 생성된 RuntimeHelpers.GetHashCode해시 코드와 동일합니다. 컴파일러가 두 문자열에 할당된 값을 단일 문자열 리터럴로 처리했기 때문에 문자열 변수가 동일한 인턴 문자열을 참조하기 때문입니다.

C#

```
using System;
using System.Runtime.CompilerServices;

public class Example
{
   public static void Main()
   {
      Console.WriteLine("{0,-18} {1,6} {2,18:N0}    {3,6} {4,18:N0}\n",
                        "", "Var 1", "Hash Code", "Var 2", "Hash Code");
```

```
      // Get hash codes of two different strings.
      String sc1 = "String #1";
      String sc2 = "String #2";
      ShowHashCodes("sc1", sc1, "sc2", sc2);

      // Get hash codes of two identical non-interned strings.
      String s1 = "This string";
      String s2 = String.Format("{0} {1}", "This", "string");
      ShowHashCodes("s1", s1, "s2", s2);

      // Get hash codes of two (evidently concatenated) strings.
      String si1 = "This is a string!";
      String si2 = "This " + "is " + "a " + "string!";
      ShowHashCodes("si1", si1, "si2", si2);
   }

   private static void ShowHashCodes(String var1, Object value1,
                                     String var2, Object value2)
   {
      Console.WriteLine("{0,-18} {1,6} {2,18:X8}    {3,6} {4,18:X8}",
                        "Obj.GetHashCode", var1, value1.GetHashCode(),
                        var2, value2.GetHashCode());

      Console.WriteLine("{0,-18} {1,6} {2,18:X8}    {3,6} {4,18:X8}\n",
                        "RTH.GetHashCode", var1,
RuntimeHelpers.GetHashCode(value1),
                        var2, RuntimeHelpers.GetHashCode(value2));
   }
}
// The example displays output similar to the following:
//                        Var 1          Hash Code     Var 2          Hash Code
//
//    Obj.GetHashCode       sc1           94EABD27       sc2           94EABD24
//    RTH.GetHashCode       sc1           02BF8098       sc2           00BB8560
//
//    Obj.GetHashCode        s1           29C5A397        s2           29C5A397
//    RTH.GetHashCode        s1           0297B065        s2           03553390
//
//    Obj.GetHashCode       si1           941BCEA5       si2           941BCEA5
//    RTH.GetHashCode       si1           01FED012       si2           01FED012
```

System.Runtime.Versioning.ComponentGuaranteesAttribute 클래스

아티클 • 2025. 04. 21.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

ComponentGuaranteesAttribute 구성 요소 및 클래스 라이브러리 개발자가 라이브러리 소비자가 여러 버전에서 기대할 수 있는 호환성 수준을 나타내는 데 사용됩니다. 라이브러리 또는 구성 요소의 이후 버전이 기존 클라이언트를 중단하지 않을 것이라는 보장 수준을 나타냅니다. 그런 다음 클라이언트는 여러 버전에서 안정성을 보장하기 위해 자체 인터페이스를 설계할 때 ComponentGuaranteesAttribute를 도구로 사용할 수 있습니다.

> **참고**
>
> CLR(공용 언어 런타임)은 이 특성을 어떤 방식으로도 사용하지 않습니다. 해당 값은 구성 요소 작성자의 의도를 공식적으로 문서화하는 데 있습니다. 컴파일 시간 도구는 이러한 선언을 사용하여 선언된 보증을 위반하는 컴파일 시간 오류를 검색할 수도 있습니다.

호환성 수준

ComponentGuaranteesAttribute는 ComponentGuaranteesOptions 열거형의 멤버로 나타내는 다음과 같은 호환성 수준을 지원합니다.

- 버전 대 버전 호환성(ComponentGuaranteesOptions.None)이 없습니다. 클라이언트는 이후 버전이 기존 클라이언트를 중단시킬 것으로 예상할 수 있습니다. 자세한 내용은 이 문서의 뒷부분에 있는 호환성 없음 섹션을 참조하세요.

- 병렬 버전 대 버전 호환성(ComponentGuaranteesOptions.SideBySide). 동일한 애플리케이션 도메인에 둘 이상의 어셈블리 버전이 로드될 때 구성 요소가 작동하도록 테스트되었습니다. 일반적으로 이후 버전은 호환성을 손상할 수 있습니다. 그러나 호환성이 손상되는 변경이 수행되면 이전 버전은 수정되지 않지만 새 버전과 함께 존재합니다. 병렬 실행은 호환성이 손상되는 변경이 수행될 때 기존 클라이언트가 작동하도록 하는 데 필요한 방법입니다. 자세한 내용은 이 문서의 뒷부분에 있는 병렬 호환성 섹션을 참조하세요.

- 안정적인 버전 대 버전 호환성(ComponentGuaranteesOptions.Stable). 이후 버전은 클라이언트를 중단해서는 안 되며 병렬 실행이 필요하지 않습니다. 그러나 클라이언트가 실수로 손상된 경우 병렬 실행을 사용하여 문제를 해결할 수 있습니다. 자세한 내용은 안정적인 호환성 섹션을 참조하세요.

- Exchange 버전 간 호환성(ComponentGuaranteesOptions.Exchange). 이후 버전이 클라이언트를 중단하지 않도록 주의를 기울입니다. 클라이언트는 서로 독립적으로 배포된 다른

어셈블리와의 통신에 사용되는 인터페이스의 서명에서 이러한 형식만 사용해야 합니다. 이러한 형식의 한 버전만 지정된 애플리케이션 도메인에 있어야 합니다. 즉, 클라이언트가 중단될 경우 병렬 실행으로 호환성 문제를 해결할 수 없습니다. 자세한 내용은 Exchange 형식 호환성 섹션을 참조하세요.

다음 섹션에서는 각 보장 수준에 대해 자세히 설명합니다.

호환성 없음

구성 요소를 표시 ComponentGuaranteesOptions.None 하면 공급자가 호환성을 보장하지 않음을 나타냅니다. 클라이언트는 노출된 인터페이스에 대한 종속성을 사용하지 않아야 합니다. 이 수준의 호환성은 실험적이거나 공개적으로 노출되지만 항상 동시에 업데이트되는 구성 요소에만 적용되는 형식에 유용합니다. None 은 외부 구성 요소에서 이 구성 요소를 사용하면 안 됨을 명시적으로 나타냅니다.

병렬 호환성

구성 요소를 ComponentGuaranteesOptions.SideBySide으로 표시하는 것은 동일한 애플리케이션 도메인에 둘 이상의 어셈블리 버전이 로드될 때 해당 구성 요소가 작동함을 테스트하여 확인했음을 나타냅니다. 버전 번호가 더 큰 어셈블리에 대해 호환성을 깨뜨리는 변경은 허용됩니다. 이전 버전의 어셈블리에 바인딩된 구성 요소는 이전 버전에 계속 바인딩되어야 하며 다른 구성 요소는 새 버전에 바인딩할 수 있습니다. SideBySide로 선언된 구성 요소를 이전 버전을 파괴적으로 수정하여 업데이트할 수도 있습니다.

안정적인 호환성

형식을 버전 간에 안정적으로 ComponentGuaranteesOptions.Stable 유지해야 하며 형식을 표시합니다. 그러나 안정적인 형식의 병렬 버전이 동일한 애플리케이션 도메인에 존재할 수도 있습니다.

안정적인 형식은 높은 이진 호환성 표시줄을 유지합니다. 이 때문에 공급자는 안정적인 형식에 대한 호환성이 손상되는 변경을 방지해야 합니다. 다음과 같은 종류의 변경이 허용됩니다.

- 직렬화 형식을 깨뜨리지 않는 한 타입에 프라이빗 인스턴스 필드를 추가하거나 필드를 제거합니다.
- 직렬화할 수 없는 형식을 직렬화할 수 있는 형식으로 변경합니다. 그러나 직렬화할 수 있는 형식은 직렬화할 수 없는 형식으로 변경할 수 없습니다.
- 메서드에서 새로 정의된 파생 예외를 던집니다.
- 메서드의 성능 향상
- 변경 내용이 대부분의 클라이언트에 부정적인 영향을 주지 않는 한 반환 값의 범위를 변경합니다.

- 비즈니스 정당성이 높고 부정적인 영향을 받는 클라이언트 수가 낮은 경우 심각한 버그를 수정합니다.

안정적인 구성 요소의 새 버전은 기존 클라이언트를 중단하지 않을 것으로 예상되므로 일반적으로 애플리케이션 도메인에는 안정적인 구성 요소의 버전이 하나만 필요합니다. 그러나 안정적인 형식은 모든 구성 요소가 동의하는 잘 알려진 교환 형식으로 사용되지 않으므로 이는 요구 사항이 아닙니다. 따라서 안정적인 구성 요소의 새 버전이 실수로 일부 구성 요소를 중단하고 다른 구성 요소에 새 버전이 필요한 경우 이전 구성 요소와 새 구성 요소를 모두 로드하여 문제를 해결할 수 있습니다.

Stable 는 .보다 None더 강력한 버전 호환성 보장을 제공합니다. 다중 버전 구성 요소에 대한 일반적인 기본값입니다.

Stable 는 구성 요소가 호환성을 손상시키지 않지만 지정된 애플리케이션 도메인에 둘 이상의 버전이 로드될 때 작동하도록 테스트됨을 나타내는 와 SideBySide결합할 수 있습니다.

형식 또는 메서드가 Stable로 표시된 후에는 Exchange으로 업그레이드할 수 있습니다. 그러나 None으로 다운그레이드할 수는 없습니다.

Exchange 형식 호환성

형식을 ComponentGuaranteesOptions.Exchange로 표시하면 Stable로 표시하는 것보다 강력한 버전 호환성을 보장하여 모든 형식 중에서 가장 안정적인 것에 적용해야 합니다. 이러한 형식은 모든 구성 요소 경계(모든 버전의 CLR 또는 구성 요소 또는 애플리케이션의 모든 버전)와 공간(프로세스 간, 한 프로세스의 교차 CLR, 단일 CLR의 애플리케이션 간 도메인) 간에 독립적으로 빌드된 구성 요소 간의 교환에 사용됩니다. 교환 형식에 대한 호환성이 손상되는 변경이 발생하는 경우 여러 버전의 형식을 로드하여 문제를 해결할 수 없습니다.

교환 형식은 문제가 매우 심각하거나(예: 심각한 보안 문제) 중단 가능성이 매우 낮은 경우에만 변경해야 합니다(즉, 코드가 종속성을 가질 수 없는 임의의 방식으로 동작이 이미 중단된 경우). 다음과 같은 종류의 교환 유형을 변경할 수 있습니다.

- 새 인터페이스 정의의 상속을 추가합니다.
- 새로 상속된 인터페이스 정의의 메서드를 구현하는 새 프라이빗 메서드를 추가합니다.
- 새 정적 필드를 추가합니다.
- 새 정적 메서드를 추가합니다.
- 가상이 아닌 새 인스턴스 메서드를 추가합니다.

다음은 호환성이 손상되는 변경으로 간주되며 기본 형식에는 허용되지 않습니다.

- 직렬화 형식 변경 버전 호환 직렬화가 필요합니다.

- 프라이빗 인스턴스 필드 추가 또는 제거 이렇게 하면 형식의 serialization 형식이 변경되고 리플렉션을 사용하는 클라이언트 코드가 손상될 위험이 있습니다.
- 형식의 직렬화 가능 여부 변경 직렬화할 수 없는 형식은 직렬화할 수 없으며 그 반대의 경우도 마찬가지입니다.
- 메서드에서 다양한 예외를 던집니다.
- 멤버 정의가 이 가능성을 높이고 클라이언트에서 알 수 없는 값을 처리하는 방법을 명확하게 나타내지 않는 한 메서드의 반환 값 범위를 변경합니다.
- 대부분의 버그를 수정합니다. 형식의 소비자는 기존 동작을 사용합니다.

구성 요소, 형식 또는 멤버가 Exchange 보장으로 표시된 후에는 Stable나 None로 변경할 수 없습니다.

일반적으로 교환 형식은 공용 인터페이스에서 일반적으로 사용되는 기본 형식(예: Int32String .NET) 및 인터페이스(예: IList<T>, IEnumerable<T> 및 IComparable<T>)입니다.

Exchange 형식은 호환성으로 Exchange 표시된 다른 형식만 공개적으로 노출할 수 있습니다. 또한 교환 유형은 변경되기 쉬운 Windows API의 동작에 따라 달라질 수 없습니다.

구성 요소 보장

다음 표에서는 구성 요소의 특성 및 사용이 호환성 보장에 미치는 영향을 나타냅니다.

테이블 확장

구성 요소 특성	교환	안정	나란히	없음
독립적으로 버전이 있는 구성 요소 간의 인터페이스에서 사용할 수 있습니다.	Y	N	N	N
독립적으로 버전이 있는 어셈블리에서 (비공개로) 사용할 수 있습니다.	Y	Y	Y	N
단일 애플리케이션 도메인에 여러 버전을 사용할 수 있습니다.	N	Y	Y	Y
중대한 변경을 할 수 있습니다.	N	N	Y	Y
어셈블리의 특정 여러 버전을 함께 로드할 수 있도록 테스트되었습니다.	N	N	Y	N
호환성이 손상되는 변경을 수행할 수 있습니다.	N	N	N	Y
안전하게 영향을 주지 않는 상태로 서비스 변경을 적용할 수 있습니다.	Y	Y	Y	Y

특성 적용

어셈블리, 형식 또는 형식 멤버에 적용 ComponentGuaranteesAttribute 할 수 있습니다. 애플리케이션은 계층적입니다. 즉, 기본적으로 어셈블리 수준에서 특성의 속성에 의해 Guarantees 정의된 보장은 어셈블리의 모든 형식과 해당 형식의 모든 멤버에 대한 보장을 정의합니다. 마찬가지로, 보장이 형식에 적용되는 경우 기본적으로 형식의 각 멤버에도 적용됩니다.

이 상속된 보장은 개별 형식 및 형식 멤버에 적용하여 재정의 ComponentGuaranteesAttribute 할 수 있습니다. 그러나 기본값을 재정의하면 보증만 약화할 수 있습니다. 그들은 그것을 강화할 수 없습니다. 예를 들어 어셈블리가 보장으로 None 표시된 경우 해당 형식과 멤버는 호환성을 보장하지 않으며 어셈블리의 형식 또는 멤버에 적용되는 다른 보장은 무시됩니다.

보증 테스트

Guarantees 속성은 ComponentGuaranteesOptions 열거형의 FlagsAttribute 특성으로 표시된 멤버를 반환합니다. 즉, 잠재적으로 알 수 없는 플래그를 마스킹하여 관심 있는 플래그를 테스트해야 합니다. 예를 들어, 다음 예제는 형식이 Stable로 표시되었는지 여부를 테스트합니다.

C#

```
// Test whether guarantee is Stable.
if ((guarantee & ComponentGuaranteesOptions.Stable) ==
ComponentGuaranteesOptions.Stable)
   Console.WriteLine($"{typ.Name} is marked as {guarantee}.");
```

다음 예제에서는 형식이 로 StableExchange표시되는지 여부를 테스트합니다.

C#

```
// Test whether guarantee is Stable or Exchange.
if ((guarantee & (ComponentGuaranteesOptions.Stable |
ComponentGuaranteesOptions.Exchange)) > 0)
   Console.WriteLine($"{typ.Name} is marked as Stable or Exchange.");
```

다음 예제에서는 형식이 None (즉, Stable도 Exchange도 아닌)으로 표시되는지 여부를 테스트합니다.

C#

```
// Test whether there is no guarantee (neither Stable nor Exchange).
if ((guarantee & (ComponentGuaranteesOptions.Stable |
ComponentGuaranteesOptions.Exchange)) == 0)
   Console.WriteLine($"{typ.Name} has no compatibility guarantee.");
```

System.Runtime.Loader.AssemblyLoadContext 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

AssemblyLoadContext 부하 컨텍스트를 나타냅니다. 개념적으로 로드 컨텍스트는 어셈블리 집합을 로드, 확인 및 언로드하기 위한 범위를 만듭니다.

AssemblyLoadContext 주로 어셈블리 로드 격리를 제공하기 위해 존재합니다. 단일 프로세스 내에서 동일한 어셈블리의 여러 버전을 로드할 수 있습니다. .NET Framework의 여러 AppDomain 인스턴스에서 제공하는 격리 메커니즘을 대체합니다.

> ⓘ **참고**
>
> - AssemblyLoadContext 에서는 보안 기능을 제공하지 않습니다. 모든 코드에는 프로세스의 모든 권한이 있습니다.
> - .NET Core 2.0 - 2.2에서만 AssemblyLoadContext 추상 클래스입니다. 이러한 버전에서 구체적인 클래스를 만들려면 메서드를 구현합니다 AssemblyLoadContext.Load(AssemblyName) .

런타임의 사용량

런타임은 두 개의 어셈블리 로드 컨텍스트를 구현합니다.

- AssemblyLoadContext.Default 는 애플리케이션 주 어셈블리 및 정적 종속성에 사용되는 런타임의 기본 컨텍스트를 나타냅니다.
- Assembly.LoadFile(String) 메서드는 가장 기본적인 AssemblyLoadContext을 인스턴스화하여 로드하는 어셈블리를 격리합니다. 단순한 격리 체계로, 각 어셈블리를 종속성 확인 없이 독립적으로 AssemblyLoadContext에서 로드합니다.

애플리케이션 사용량

애플리케이션은 고급 시나리오에 대한 맞춤 솔루션을 만들기 위해 자체 AssemblyLoadContext 을(를) 적용할 수 있습니다. 사용자 지정은 종속성 확인 메커니즘을 정의하는 데 중점을 둡니다.

AssemblyLoadContext는 관리되는 어셈블리 해결을 구현하기 위한 두 가지 확장 지점을 제공합니다.

1. 이 AssemblyLoadContext.Load(AssemblyName) 메서드는 AssemblyLoadContext가 어셈블리를 확인하고 로드하며 반환할 수 있는 첫 번째 기회를 제공합니다. 메서드가 AssemblyLoadContext.Load(AssemblyName)를 `null`로 반환하면 로더는 어셈블리를 AssemblyLoadContext.Default에 로드하려고 시도합니다.
2. 어셈블리를 AssemblyLoadContext.Default에서 해결할 수 없는 경우, 원래 AssemblyLoadContext이 어셈블리를 해결할 수 있는 두 번째 기회가 준비됩니다. 런타임이 Resolving 이벤트를 발생시킵니다.

또한 가상 메서드를 AssemblyLoadContext.LoadUnmanagedDll(String) 사용하면 관리되지 않는 기본 어셈블리 해상도를 사용자 지정할 수 있습니다. 기본 구현은 `null`를 반환하여 런타임 검색이 기본 검색 정책을 사용하도록 합니다. 기본 검색 정책은 대부분의 시나리오에 충분합니다.

기술 과제

- 단일 프로세스에서 여러 버전의 런타임을 로드할 수 없습니다.

  > **⊗ 주의**
  >
  > 여러 복사본 또는 다른 버전의 프레임워크 어셈블리를 로드하면 예기치 않고 진단하기 어려운 동작이 발생할 수 있습니다.

  > **팁**
  >
  > 원격 또는 프로세스 간 통신과 함께 프로세스 경계를 사용하여 이 격리 문제를 해결합니다.

- 어셈블리 로드 타이밍으로 인해 테스트 및 디버깅이 어려울 수 있습니다. 어셈블리는 일반적으로 종속성이 즉시 확인되지 않고 로드됩니다. 종속성은 필요에 따라 로드됩니다.
  - 코드가 종속 어셈블리로 분기되는 경우
  - 코드가 리소스를 로드하는 경우
  - 코드가 어셈블리를 명시적으로 로드하는 경우

- AssemblyLoadContext.Load(AssemblyName)를 구현하면 서로 다른 버전이 존재할 수 있도록 격리해야 할지도 모르는 새로운 종속성이 추가될 수 있습니다. 가장 자연스러운 구현은 이러한 종속성을 기본 컨텍스트에 배치합니다. 신중한 디자인은 새 종속성을 격리할 수 있습니다.

- 동일한 어셈블리가 여러 컨텍스트에 여러 번 로드됩니다.
  - 이로 인해 "'Sample.Plugin' 형식의 개체를 'Sample.Plugin' 형식으로 캐스팅할 수 없습니다."와 같은 혼동스러운 오류 메시지가 생성될 수 있습니다.

- 격리 경계를 넘어 데이터를 정리하고 구성하는 작업은 간단하지 않습니다. 일반적인 솔루션은 기본 로드 컨텍스트로만 로드되는 어셈블리에 정의된 인터페이스를 사용하는 것입니다.

System.Diagnostics.ProcessStartInfo.UseShellExecute 속성

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 클래스는 ProcessStartInfo 프로세스를 시작할 때 사용되는 값 집합을 지정합니다.

UseShellExecute 속성을 설정하면 `false` 입력, 출력 및 오류 스트림을 리디렉션할 수 있습니다.

이 컨텍스트의 "shell"(`UseShellExecute`)이라는 단어는 명령 셸(예: 또는`bash`)이 아닌 그래픽 셸(Windows `sh` 셸과 유사)을 참조하며 사용자가 그래픽 애플리케이션을 시작하거나 문서를 열 수 있도록 합니다.

> ⓘ **참고 항목**
>
> UseShellExecute은 `false` 속성이 UserName이나 빈 문자열이 아닌 경우에 `null` 이어야 하며, 그렇지 않으면 InvalidOperationException 메서드를 호출하면 Process.Start(ProcessStartInfo) 오류가 발생합니다.

운영 체제 셸을 사용하여 프로세스를 시작하는 경우 모든 문서(기본 열기 작업이 있는 실행 파일과 연결된 등록된 파일 형식)를 시작하고 개체를 사용하여 Process 인쇄와 같은 파일에 대한 작업을 수행할 수 있습니다. UseShellExecute이 `false` 상태일 때, Process 객체를 사용하여 실행 파일만 시작할 수 있습니다.

> ⓘ **참고 항목**
>
> UseShellExecute은 `true` 속성을 ErrorDialog로 설정한 경우 `true` 이어야 합니다.

WorkingDirectory

WorkingDirectory 속성은 UseShellExecute 속성의 값에 따라 다르게 동작합니다. UseShellExecute이 `true`일 때, WorkingDirectory 속성은 실행 파일의 위치를 지정합니다. 빈 문자열인 경우 WorkingDirectory 현재 디렉터리에 실행 파일이 포함되어 있다고 가정합니다.

이 UseShellExecute`false` 경우 WorkingDirectory 속성은 실행 파일을 찾는 데 사용되지 않습니다. 대신 시작되고 새 프로세스의 컨텍스트 내에서만 의미가 있는 프로세스에서만 사용됩니다. UseShellExecute가 `false`일 때, FileName 속성은 실행 파일의 완전한 경로 또는 시스템이 `PATH`

환경 변수에 지정된 폴더 내에서 찾으려고 시도하는 간단한 실행 파일 이름이 될 수 있습니다. 검색 경로의 해석은 운영 체제에 따라 달라집니다. 자세한 내용은 명령 프롬프트에서 HELP PATH 또는 man sh를 입력하십시오.

Last updated on 2026. 02. 12.

System.Environment.GetEnvironmentVariable 메서드

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

메서드는 GetEnvironmentVariable 현재 프로세스에서 환경 변수의 값을 검색합니다.

환경 변수 이름은 Unix와 유사한 시스템에서 대/소문자를 구분하지만 Windows에서는 대/소문자를 구분하지 않습니다.

> **참고**
>
> 네이티브 라이브러리에서 수행한 In-Process 환경 수정은 관리되는 호출자가 볼 수 없습니다. 반대로, 관리되는 호출자가 수정한 내용은 네이티브 라이브러리에서 볼 수 없습니다.

GetEnvironmentVariable(String) 메서드

메서드는 GetEnvironmentVariable(String) 현재 프로세스의 환경 블록에서만 환경 변수를 검색합니다. GetEnvironmentVariable(String, EnvironmentVariableTarget) 메서드를 `target` 값 EnvironmentVariableTarget.Process로 호출하는 것과 같습니다.

값과 함께 모든 환경 변수를 검색하려면 메서드를 호출합니다 GetEnvironmentVariables .

Windows 시스템에서

Windows 시스템에서 현재 프로세스의 환경 블록에는 다음이 포함됩니다.

- 해당 변수를 만든 부모 프로세스에서 제공하는 모든 환경 변수입니다. 예를 들어 콘솔 창에서 시작된 .NET 애플리케이션은 콘솔 창의 모든 환경 변수를 상속합니다.

  부모 프로세스가 없는 경우 컴퓨터별 및 사용자별 환경 변수가 대신 사용됩니다. 예를 들어 새 콘솔 창에는 컴퓨터당 및 사용자별 환경 변수가 모두 시작되었을 때 정의됩니다.

- 프로세스가 실행되는 동안 SetEnvironmentVariable(String, String) 메서드 또는 SetEnvironmentVariable(String, String, EnvironmentVariableTarget) 메서드를 `target` 값 EnvironmentVariableTarget.Process으로 호출하여 프로세스 블록에 추가된 모든 변수. 이러한 환경 변수는 .NET 애플리케이션이 종료될 때까지 유지됩니다.

프로세스가 시작된 후 환경 변수가 만들어지면, SetEnvironmentVariable(String, String) 메서드 또는 SetEnvironmentVariable(String, String, EnvironmentVariableTarget) 메서드를 `target` 값

.EnvironmentVariableTarget.Process로 호출하여 만들어진 변수만 이 메서드를 사용하여 검색할 수 있습니다.

Unix와 유사한 시스템에서

Unix와 유사한 시스템에서 현재 프로세스의 환경 블록에는 다음과 같은 환경 변수가 포함됩니다.

- 해당 변수를 만든 부모 프로세스에서 제공하는 모든 환경 변수입니다. 셸에서 시작된 .NET 애플리케이션의 경우 셸에 정의된 모든 환경 변수가 포함됩니다.
- 프로세스가 실행되는 동안 SetEnvironmentVariable(String, String) 메서드 또는 SetEnvironmentVariable(String, String, EnvironmentVariableTarget) 메서드를 `target` 값 EnvironmentVariableTarget.Process으로 호출하여 프로세스 블록에 추가된 모든 변수. 이러한 환경 변수는 .NET 애플리케이션이 종료될 때까지 유지됩니다.

Unix와 유사한 시스템의 .NET은 컴퓨터별 또는 사용자별 환경 변수를 지원하지 않습니다.

GetEnvironmentVariable(String, EnvironmentVariableTarget) 메서드

값과 함께 모든 환경 변수를 검색하려면 메서드를 호출합니다 GetEnvironmentVariables .

Windows 시스템에서

Windows에서 `target` 매개 변수는 환경 변수가 현재 프로세스에서 검색되는지 아니면 현재 사용자 또는 로컬 컴퓨터에 대한 Windows 운영 체제 레지스트리 키에서 검색되는지 여부를 지정합니다. 모든 사용자별 및 컴퓨터별 환경 변수는 .NET 프로세스를 만든 부모 프로세스에서 사용할 수 있는 다른 환경 변수와 마찬가지로 현재 프로세스의 환경 블록에 자동으로 복사됩니다. 현재 프로세스의 환경 블록에 SetEnvironmentVariable(String, String) 메서드나 SetEnvironmentVariable(String, String, EnvironmentVariableTarget) 메서드를 `target` 값 EnvironmentVariableTarget.Process으로 호출하여 추가된 환경 변수는 프로세스 실행 중에만 지속됩니다.

Unix와 유사한 시스템에서

Unix와 유사한 시스템에서는 메서드가 오직 `GetEnvironmentVariable(String, EnvironmentVariableTarget)` `target` 값만 지원합니다. `target`값이 EnvironmentVariableTarget.Machine 또는 EnvironmentVariableTarget.User인 호출은 지원되지 않으며 `null`을 반환합니다.

프로세스별 환경 변수는 다음과 같습니다.

- 부모 프로세스에서 상속된 항목에는 `dotnet.exe`를 소환하거나 .NET 애플리케이션을 시작하는 데 사용되는 셸 등이 있습니다.
- SetEnvironmentVariable(String, String) 값이 SetEnvironmentVariable(String, String, EnvironmentVariableTarget)일 때, `target` 메서드 또는 EnvironmentVariableTarget.Process 메서드를 호출하여 정의된 것입니다. 이러한 환경 변수는 프로세스 또는 .NET 애플리케이션이 종료될 때까지 `dotnet` 만 유지됩니다.

CLR(공용 언어 런타임) 개요

.NET은 코드를 실행하는 공용 언어 런타임이라는 런타임 환경을 제공하고 개발 프로세스를 더 쉽게 만드는 서비스를 제공합니다.

컴파일러 및 도구는 공용 언어 런타임의 기능을 노출하고 관리되는 실행 환경의 이점을 제공하는 코드를 작성할 수 있도록 합니다. 런타임을 대상으로 하는 언어 컴파일러를 사용하여 개발하는 코드를 관리 코드라고 합니다. 관리 코드는 언어 간 통합, 언어 간 예외 처리, 향상된 보안, 버전 관리 및 배포 지원, 구성 요소 상호 작용을 위한 간소화된 모델, 디버깅 및 프로파일링 서비스와 같은 기능의 이점을 누릴 수 있습니다.

> ⓘ **참고**
>
> 컴파일러 및 도구는 형식 시스템, 메타데이터 형식 및 런타임 환경(가상 실행 시스템)이 모두 공용 표준인 ECMA 공용 언어 인프라 사양에 의해 정의되기 때문에 공용 언어 런타임에서 사용할 수 있는 출력을 생성할 수 있습니다. 자세한 내용은 **ECMA C# 및 공용 언어 인프라 사양을 참조하세요**.

런타임이 관리 코드에 서비스를 제공할 수 있도록 하려면 언어 컴파일러가 코드의 형식, 멤버 및 참조를 설명하는 메타데이터를 내보내야 합니다. 메타데이터는 코드와 함께 저장됩니다. 로드 가능한 모든 공용 언어 런타임 PE(이식 가능한 실행 파일) 파일에는 메타데이터가 포함됩니다. 런타임은 메타데이터를 사용하여 클래스를 찾아서 로드하고, 메모리에 인스턴스를 배치하고, 메서드 호출을 확인하고, 네이티브 코드를 생성하고, 보안을 적용하고, 런타임 컨텍스트 경계를 설정합니다.

런타임은 개체 레이아웃을 자동으로 처리하고 개체에 대한 참조를 관리하여 더 이상 사용되지 않을 때 해제합니다. 이러한 방식으로 수명이 관리되는 개체를 관리되는 데이터라고 합니다. 가비지 수집은 메모리 누수 및 기타 일반적인 프로그래밍 오류를 제거합니다. 코드가 관리되는 경우 .NET 애플리케이션에서 관리되는 데이터, 관리되지 않는 데이터 또는 둘 다를 사용할 수 있습니다. 언어 컴파일러는 기본 형식과 같은 고유한 형식을 제공하므로 데이터가 관리되고 있는지 여부를 항상 알거나 알 필요가 없을 수 있습니다.

공용 언어 런타임을 사용하면 개체가 언어 간에 상호 작용하는 구성 요소 및 애플리케이션을 쉽게 디자인할 수 있습니다. 다른 언어로 작성된 개체는 서로 통신할 수 있으며 해당 동작을 긴밀하게 통합할 수 있습니다. 예를 들어 클래스를 정의한 다음 다른 언어를 사용하여 원래 클래스에서 클래스를 파생하거나 원래 클래스에서 메서드를 호출할 수 있습니다. 클래스의 인스턴스를 다른 언어로 작성된 클래스의 메서드에 전달할 수도 있습니다. 이 언어 간 통합은 런타임을 대상으로 하는 언어 컴파일러 및 도구가 런타임에 정의된 공용 형식 시스템을 사용하기 때문에 가능합니다. 새 형식을 정의하고 형식에 대한 생성, 사용, 유지 및 바인딩에 대한 런타임의 규칙을 따릅니다.

메타데이터의 일부로 모든 관리되는 구성 요소는 빌드된 구성 요소 및 리소스에 대한 정보를 전달합니다. 런타임은 이 정보를 사용하여 구성 요소 또는 애플리케이션에 필요한 모든 항목의 지정된 버전이 있는지 확인하므로 일부 충족되지 않은 종속성으로 인해 코드가 중단될 가능성이 줄어듭니다. 등록 정보 및 상태 데이터는 더 이상 레지스트리에 저장되지 않으므로 설정 및 유지 관리가 어려울 수 있습니다. 대신 정의한 형식 및 해당 종속성에 대한 정보는 코드와 함께 메타데이터로 저장됩니다. 이렇게 하면 구성 요소 복제 및 제거 작업이 덜 복잡합니다.

언어 컴파일러 및 도구는 개발자에게 유용하고 직관적인 방식으로 런타임의 기능을 노출합니다. 런타임의 일부 기능은 다른 환경보다 한 환경에서 더 두드러질 수 있습니다. 런타임을 경험하는 방법은 사용하는 언어 컴파일러 또는 도구에 따라 달라집니다. 예를 들어 Visual Basic 개발자인 경우 공용 언어 런타임을 사용하면 Visual Basic 언어에 이전보다 더 많은 개체 지향 기능이 있음을 알 수 있습니다. 런타임은 다음과 같은 이점을 제공합니다.

- 성능 향상
- 다른 언어로 개발된 구성 요소를 쉽게 사용할 수 있는 기능입니다.
- 클래스 라이브러리에서 제공하는 확장 가능한 형식입니다.
- 개체 지향 프로그래밍에 대한 상속, 인터페이스 및 오버로드와 같은 언어 기능입니다.
- 다중 스레드 및 확장 가능한 애플리케이션을 만들 수 있는 명시적 무료 스레딩을 지원합니다.
- 구조적 예외 처리를 지원합니다.
- 사용자 지정 특성에 대한 지원.
- 가비지 수집.
- 형식 안전성 및 보안을 강화하기 위해 함수 포인터 대신 대리자를 사용합니다. 대리자에 대한 자세한 내용은 Common Type System을 참조하세요.

CLR 버전

.NET Core 및 .NET 5+ 릴리스에는 단일 제품 버전이 있습니다. 즉, 별도의 CLR 버전은 없습니다. .NET Core 버전 목록은 .NET Core 다운로드를 참조하세요.

그러나 .NET Framework 버전 번호가 포함된 CLR의 버전 번호와 반드시 일치하는 것은 아닙니다. .NET Framework 버전 및 해당 CLR 버전 목록은 .NET Framework 버전 및 종속성을 참조하세요.

관련 문서

테이블 확장

제목	설명
관리되는 실행 프로세스	공용 언어 런타임을 활용하는 데 필요한 단계를 설명합니다.
자동 메모리 관리	가비지 수집기가 메모리를 할당하고 해제하는 방법을 설명합니다.
.NET Framework 개요	공용 형식 시스템, 언어 간 상호 운용성, 관리되는 실행, 애플리케이션 도메인 및 어셈블리와 같은 주요 .NET Framework 개념에 대해 설명합니다.
공용 형식 시스템	언어 간 통합을 지원하기 위해 런타임에서 형식을 선언, 사용 및 관리하는 방법을 설명합니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 21.

관리형 실행 프로세스

2025. 06. 17.

관리되는 실행 프로세스에는 이 항목의 뒷부분에서 자세히 설명하는 다음 단계가 포함됩니다.

1. 컴파일러 선택 공용 언어 런타임에서 제공하는 이점을 얻으려면 런타임을 대상으로 하는 하나 이상의 언어 컴파일러를 사용해야 합니다.
2. 코드를 중간 언어로 컴파일합니다. 컴파일은 소스 코드를 CIL(공용 중간 언어)로 변환하고 필요한 메타데이터를 생성합니다.
3. CIL을 네이티브 코드로 컴파일합니다. 실행 시 JIT(Just-In-Time) 컴파일러는 CIL을 네이티브 코드로 변환합니다. 이 컴파일 중에 코드는 CIL 및 메타데이터를 검사하는 확인 프로세스를 통과하여 코드가 형식이 안전한지 확인할 수 있는지 확인해야 합니다.
4. 코드를 실행합니다. 공용 언어 런타임은 실행이 수행될 수 있는 인프라와 실행 중에 사용할 수 있는 서비스를 제공합니다.

컴파일러 선택

CLR(공용 언어 런타임)에서 제공하는 이점을 얻으려면 런타임을 대상으로 하는 하나 이상의 언어 컴파일러(예: Visual Basic, C#, Visual C++, F#) 또는 Eiffel, Perl 또는 COBOL 컴파일러와 같은 여러 타사 컴파일러 중 하나를 사용해야 합니다.

다국어 실행 환경이므로 런타임은 다양한 데이터 형식 및 언어 기능을 지원합니다. 사용하는 언어 컴파일러에 따라 사용 가능한 런타임 기능이 결정되며 이러한 기능을 사용하여 코드를 디자인합니다. 런타임이 아닌 컴파일러는 코드에서 사용해야 하는 구문을 설정합니다. 다른 언어로 작성된 구성 요소에서 구성 요소를 완전히 사용할 수 있어야 하는 경우 구성 요소의 내보낸 형식은 CLS(공용 언어 사양)에 포함된 언어 기능만 노출해야 합니다. 특성을 사용하여 CLSCompliantAttribute 코드가 CLS 규격인지 확인할 수 있습니다. 자세한 내용은 언어 독립성 및 언어 독립적 구성 요소를 참조하세요.

CIL로 컴파일

관리 코드로 컴파일할 때 컴파일러는 소스 코드를 네이티브 코드로 효율적으로 변환할 수 있는 CPU 독립적 명령 집합인 CIL(공용 중간 언어)로 변환합니다. CIL에는 개체에 대한 메서드 로드, 저장, 초기화 및 호출에 대한 지침과 산술 및 논리 작업, 제어 흐름, 직접 메모리 액세스, 예외 처리 및 기타 작업에 대한 지침이 포함되어 있습니다. 코드를 실행하려면 먼저 일반적으로 JIT(Just-In-Time) 컴파일러에서 CIL을 CPU별 코드로 변환해야 합니다. 공용 언어 런타임은 지원하는 각 컴퓨터 아키텍처에 대해 하나 이상의 JIT 컴파일러를 제공하므로 지원되는 모든 아키텍처에서 동일한 CIL 집합을 JIT 컴파일하고 실행할 수 있습니다.

컴파일러가 CIL을 생성하면 메타데이터도 생성됩니다. 메타데이터는 각 형식의 정의, 각 형식 멤버의 서명, 코드가 참조하는 멤버 및 런타임이 실행 시 사용하는 기타 데이터를 포함하여 코드의 형식을 설명합니다. CIL 및 메타데이터는 이전에 실행 콘텐츠에 사용되었던 Microsoft의 PE(이식 가능한 실행 파일) 및 COFF(공용 개체 파일 형식) 형식을 기반으로 하고 이를 확장한 PE 파일에 포함됩니다. CIL 또는 네이티브 코드와 메타데이터를 수용하는 이 파일 형식을 사용하면 운영 체제에서 공용 언어 런타임 이미지를 인식할 수 있습니다. 파일에 메타데이터가 CIL과 함께 있으면 코드가 자신을 설명할 수 있으므로 형식 라이브러리 또는 IDL(인터페이스 정의 언어)이 필요하지 않습니다. 런타임은 실행 중에 필요에 따라 파일에서 메타데이터를 찾아 추출합니다.

CIL을 네이티브 코드로 컴파일

CIL(공용 중간 언어)을 실행하려면 먼저 공용 언어 런타임에 대해 대상 컴퓨터 아키텍처의 네이티브 코드로 컴파일해야 합니다. .NET은 이 변환을 수행하는 두 가지 방법을 제공합니다.

- .NET JIT(Just-In-Time) 컴파일러입니다.
- Ngen.exe(네이티브 이미지 생성기).

JIT 컴파일러에 의한 컴파일

JIT 컴파일은 어셈블리의 콘텐츠가 로드되고 실행될 때 애플리케이션 런타임에 요청 시 CIL을 네이티브 코드로 변환합니다. 공용 언어 런타임은 지원되는 각 CPU 아키텍처에 대해 JIT 컴파일러를 제공하므로 개발자는 JIT 컴파일 및 다른 컴퓨터 아키텍처를 사용하는 다른 컴퓨터에서 실행할 수 있는 CIL 어셈블리 집합을 빌드할 수 있습니다. 그러나 관리 코드가 플랫폼별 네이티브 API 또는 플랫폼별 클래스 라이브러리를 호출하는 경우 해당 운영 체제에서만 실행됩니다.

JIT 컴파일은 실행 중에 일부 코드가 호출되지 않을 가능성을 고려합니다. PE 파일의 모든 CIL을 네이티브 코드로 변환하는 데 시간과 메모리를 사용하는 대신 실행 중에 필요에 따라 CIL을 변환하고 결과 네이티브 코드를 메모리에 저장하여 해당 프로세스의 컨텍스트에서 후속 호출에 액세스할 수 있도록 합니다. 로더는 형식이 로드되고 초기화될 때 형식의 각 메서드에 스텁을 만들고 연결합니다. 메서드가 처음으로 호출되면 스텁은 JIT 컴파일러에 제어를 전달합니다. 이 컴파일러는 해당 메서드의 CIL을 네이티브 코드로 변환하고 생성된 네이티브 코드를 직접 가리키도록 스텁을 수정합니다. 따라서 JIT 컴파일 메서드에 대한 후속 호출은 네이티브 코드로 직접 이동합니다.

NGen.exe 사용하여 설치 시간 코드 생성

JIT 컴파일러는 해당 어셈블리에 정의된 개별 메서드가 호출되면 어셈블리의 CIL을 네이티브 코드로 변환하므로 런타임에 성능에 부정적인 영향을 줍니다. 대부분의 경우 성능 저하가 허용됩니다. 더 중요한 것은 JIT 컴파일러에서 생성된 코드가 컴파일을 트리거한 프로세스에 바인딩된다는 것입니다. 여러 프로세스에서 공유할 수 없습니다. 생성된 코드를 애플리케이션의 여러 호출 또는 어셈블리 집합을 공유하는 여러 프로세스에서 공유할 수 있도록 공용 언어 런타임은 미

리 컴파일 모드를 지원합니다. 이 미리 컴파일 모드는 Ngen.exe(네이티브 이미지 생성기) 를 사용하여 JIT 컴파일러와 마찬가지로 CIL 어셈블리를 네이티브 코드로 변환합니다. 그러나 Ngen.exe 작업은 다음과 같은 세 가지 방법으로 JIT 컴파일러의 작업과 다릅니다.

- 애플리케이션이 실행되는 동안 대신 애플리케이션을 실행하기 전에 CIL에서 네이티브 코드로 변환을 수행합니다.
- 한 번에 하나의 메서드 대신 전체 어셈블리를 한 번에 컴파일합니다.
- 생성된 코드를 네이티브 이미지 캐시에 디스크의 파일로 유지합니다.

코드 확인

네이티브 코드에 대한 컴파일의 일환으로 CIL 코드는 관리자가 코드를 통해 확인을 우회할 수 있는 보안 정책을 설정하지 않은 한 확인 프로세스를 통과해야 합니다. 확인은 CIL 및 메타데이터를 검사하여 코드가 형식이 안전한지 여부를 확인합니다. 즉, 액세스 권한이 부여된 메모리 위치에만 액세스합니다. 형식 안전은 개체를 서로 격리하는 데 도움이 되며 실수로 인한 손상이나 악의적인 손상으로부터 개체를 보호하는 데 도움이 됩니다. 또한 코드에 대한 보안 제한을 안정적으로 적용할 수 있음을 보장합니다.

런타임은 다음 명령문이 안전한 형식의 코드에 대해 true라는 사실에 의존합니다.

- 형식에 대한 참조는 참조되는 형식과 엄격하게 호환됩니다.
- 개체에서 적절하게 정의된 작업만 호출됩니다.
- 정체성은 그들이 주장하는 대로입니다.

확인 프로세스 중에 코드가 메모리 위치에 액세스하고 올바르게 정의된 형식을 통해서만 메서드를 호출할 수 있는지 확인하기 위해 CIL 코드가 검사됩니다. 예를 들어 코드는 메모리 위치를 오버런할 수 있는 방식으로 개체의 필드에 액세스하도록 허용할 수 없습니다. 또한 잘못된 CIL이 형식 안전 규칙을 위반할 수 있으므로 확인은 코드를 검사하여 CIL이 올바르게 생성되었는지 여부를 확인합니다. 확인 프로세스는 잘 정의된 형식 안전 코드 집합을 전달하고 형식이 안전한 코드만 전달합니다. 그러나 일부 형식 안전 코드는 확인 프로세스의 일부 제한 사항으로 인해 확인을 통과하지 못할 수 있으며 일부 언어는 의도적으로 검증 가능한 형식 안전 코드를 생성하지 않습니다. 보안 정책에서 타입-안전한 코드를 요구하지만 코드가 확인을 통과하지 못하면 코드가 실행될 때 예외가 발생합니다.

코드 실행

공용 언어 런타임은 관리형 실행을 수행할 수 있는 인프라와 실행 중에 사용할 수 있는 서비스를 제공합니다. 메서드를 실행하려면 먼저 프로세서별 코드로 컴파일해야 합니다. CIL이 생성된 각 메서드는 처음 호출된 후 실행될 때 JIT 컴파일됩니다. 다음에 메서드를 실행할 때 기존 JIT 컴파일 네이티브 코드가 실행됩니다. 실행이 완료될 때까지 JIT 컴파일 및 코드 실행 프로세스가 반복됩니다.

실행하는 동안 관리 코드는 가비지 수집, 보안, 비관리 코드와의 상호 운용성, 언어 간 디버깅 지원, 향상된 배포 및 버전 관리 지원과 같은 서비스를 받습니다.

Microsoft Windows Vista에서 운영 체제 로더는 COFF 헤더에서 비트를 검사하여 관리되는 모듈을 확인합니다. 설정되는 비트는 관리되는 모듈을 표시합니다. 로더가 관리되는 모듈을 감지하면 mscoree.dll가 로드되고, 관리되는 모듈 이미지가 로드되거나 언로드될 때 `_CorValidateImage`와 `_CorImageUnloading`가 로더에 알립니다. `_CorValidateImage` 다음 작업을 수행합니다.

1. 코드가 유효한 관리 코드인지 확인합니다.
2. 이미지의 진입점을 런타임의 진입점으로 변경합니다.

64비트 Windows `_CorValidateImage` 에서는 PE32에서 PE32+ 형식으로 변환하여 메모리에 있는 이미지를 수정합니다.

참고하십시오

- 개요
- 언어 독립성 및 언어 독립적 구성 요소
- 메타데이터 및 Self-Describing 구성 요소
- Ilasm.exe(IL 어셈블러)
- 보안
- 비관리 코드와의 상호 운용
- 배포
- .NET의 어셈블리
- 애플리케이션 도메인

.NET의 어셈블리

어셈블리는 배포, 버전 제어, 재사용, 활성화 범위 지정 및 보안 권한의 기본 단위입니다. NET 기반 애플리케이션. 어셈블리는 함께 작동하고 기능의 논리적 단위를 형성하기 위해 빌드된 형식 및 리소스의 컬렉션입니다. 어셈블리는 실행 파일(*.exe*) 또는 동적 링크 라이브러리(*.dll*) 파일 형식을 사용하며 .NET 애플리케이션의 구성 요소입니다. 형식 구현을 인식하는 데 필요한 정보를 공용 언어 런타임에 제공합니다.

.NET 및 .NET Framework에서는 하나 이상의 소스 코드 파일에서 어셈블리를 빌드할 수 있습니다. .NET Framework에서 어셈블리에는 하나 이상의 모듈이 포함될 수 있습니다. 이렇게 하면 여러 개발자가 단일 어셈블리를 만들기 위해 결합된 별도의 소스 코드 파일 또는 모듈에서 작업할 수 있도록 대규모 프로젝트를 계획할 수 있습니다. 모듈에 대한 자세한 내용은 방법: 다중 파일 어셈블리 빌드를 참조하세요.

어셈블리에는 다음과 같은 속성이 있습니다.

- 어셈블리는 *.exe* 또는 *.dll* 파일로 구현됩니다.
- .NET Framework를 대상으로 하는 라이브러리의 경우 GAC(전역 어셈블리 캐시)에 배치하여 애플리케이션 간에 어셈블리를 공유할 수 있습니다. GAC에 포함하려면 먼저 어셈블리에 강력한 이름을 지정해야 합니다. 자세한 내용은 강력한 이름의 어셈블리를 참조하세요.
- 어셈블리는 필요한 경우에만 메모리에 로드됩니다. 사용되지 않으면 로드되지 않습니다. 따라서 어셈블리는 대규모 프로젝트에서 리소스를 관리하는 효율적인 방법이 될 수 있습니다.
- 리플렉션을 사용하여 어셈블리에 대한 정보를 프로그래밍 방식으로 가져올 수 있습니다. 자세한 내용은 리플렉션(C#) 또는 리플렉션(Visual Basic)을 참조하세요.
- .NET 및 .NET Framework에서 MetadataLoadContext 클래스를 사용하여 단순히 검사 목적으로 어셈블리를 로드할 수 있습니다. MetadataLoadContext 는 메서드를 Assembly.ReflectionOnlyLoad 대체합니다.

공용 언어 런타임에서의 어셈블리

어셈블리는 형식 구현을 인식하는 데 필요한 정보를 공용 언어 런타임에 제공합니다. 런타임에서 형식은 어셈블리의 컨텍스트 외부에서는 존재하지 않습니다.

어셈블리는 다음 정보를 정의합니다.

- 공용 언어 런타임이 실행하는 **코드**입니다. 각 어셈블리에는 진입점`DllMain`이 `WinMain` `Main`하나만 있을 수 있습니다.

- **보안 경계**입니다. 어셈블리는 사용 권한이 요청되고 부여되는 단위입니다. 어셈블리의 보안 경계에 대한 자세한 내용은 어셈블리 보안 고려 사항을 참조하세요.
- **형식 경계**입니다. 모든 형식의 ID에는 해당 ID가 있는 어셈블리의 이름이 포함됩니다. 한 어셈블리의 범위에서 로드되는 형식 `MyType` 은 다른 어셈블리의 범위에 로드되는 형식과 `MyType` 다릅니다.
- **참조 범위 경계**: 어셈블리 매니페스트에는 형식을 확인하고 리소스 요청을 충족하는 데 사용되는 메타데이터가 있습니다. 매니페스트는 어셈블리 외부에 노출할 형식 및 리소스를 지정하고 종속되는 다른 어셈블리를 열거합니다. PE(이식 가능한 실행 파일) 파일의 CIL(공용 중간 언어) 코드는 연결된 어셈블리 매니페스트가 없는 한 실행되지 않습니다.
- **버전 경계**입니다. 어셈블리는 공용 언어 런타임에서 가장 작은 버전 관리 가능한 단위입니다. 동일한 어셈블리의 모든 형식 및 리소스는 단위로 버전이 지정됩니다. 어셈블리 매니페스트는 종속 어셈블리에 대해 지정하는 버전 종속성을 설명합니다. 버전 관리에 대한 자세한 내용은 어셈블리 버전 관리를 참조하세요.
- **배포 단위**: 애플리케이션이 시작될 때 애플리케이션이 처음에 호출하는 어셈블리만 있어야 합니다. 지역화 리소스 또는 유틸리티 클래스를 포함하는 어셈블리와 같은 다른 어셈블리는 요청 시 검색할 수 있습니다. 이 프로세스를 사용하면 처음 다운로드할 때 앱이 간단하고 얇아질 수 있습니다. 어셈블리 배포에 대한 자세한 내용은 애플리케이션 배포를 참조하세요.
- **병렬 실행 단위**: 여러 버전의 어셈블리를 실행하는 방법에 대한 자세한 내용은 어셈블리 및 병렬 실행을 참조하세요.

어셈블리 만들기

어셈블리는 정적 또는 동적일 수 있습니다. 정적 어셈블리는 PE(이식 가능한 실행 파일) 파일의 디스크에 저장됩니다. 정적 어셈블리에는 인터페이스, 클래스 및 비트맵, JPEG 파일 및 기타 리소스 파일과 같은 리소스가 포함될 수 있습니다. 메모리에서 직접 실행되고 실행 전에 디스크에 저장되지 않는 동적 어셈블리를 만들 수도 있습니다. 동적 어셈블리가 실행된 후 디스크에 저장할 수 있습니다.

어셈블리를 만드는 방법에는 여러 가지가 있습니다. *.dll* 만들거나 파일을 *.exe* 수 있는 Visual Studio와 같은 개발 도구를 사용할 수 있습니다. Windows SDK의 도구를 사용하여 다른 개발 환경의 모듈을 사용하여 어셈블리를 만들 수 있습니다. 같은 공용 언어 런타임 API System.Reflection.Emit를 사용하여 동적 어셈블리를 만들 수도 있습니다.

Visual Studio에서 어셈블리를 빌드하거나, .NET Core 명령줄 인터페이스 도구를 사용하여 빌드하거나, 명령줄 컴파일러를 사용하여 .NET Framework 어셈블리를 빌드하여 어셈블리를 컴파일합니다. .NET CLI를 사용하여 어셈블리를 빌드하는 방법에 대한 자세한 내용은 .NET CLI 개요를 참조하세요.

> ⓘ **참고**
>
> Visual Studio에서 어셈블리를 빌드하려면 **빌드** 메뉴에서 **빌드**를 선택합니다.

어셈블리 매니페스트

모든 어셈블리에는 *어셈블리 매니페스트 파일이 있습니다*. 목차와 마찬가지로 어셈블리 매니페스트에는 다음이 포함됩니다.

- 어셈블리의 ID(이름 및 버전)입니다.
- 어셈블리를 구성하는 다른 모든 파일을 설명하는 파일 테이블입니다. 여기에는 *.exe* 또는 *.dll* 파일이 의존하는 다른 어셈블리, 비트맵 파일 또는 추가 정보 파일이 포함됩니다.
- .dll파일 또는 기타 *파일과 같은* 모든 외부 종속성의 목록인 *어셈블리 참조 목록*입니다. 어셈블리 참조에는 전역 개체와 프라이빗 개체 모두에 대한 참조가 포함됩니다. 전역 개체는 다른 모든 애플리케이션에서 사용할 수 있습니다. .NET Core에서 전역 개체는 특정 .NET Core 런타임과 결합됩니다. .NET Framework에서 전역 개체는 GAC(전역 어셈블리 캐시)에 상주합니다. *System.IO.dll* GAC의 어셈블리 예제입니다. 프라이빗 개체는 앱이 설치된 디렉터리 이하의 디렉터리 수준에 있어야 합니다.

어셈블리에는 콘텐츠, 버전 관리 및 종속성에 대한 정보가 포함됩니다. 따라서 이를 사용하는 애플리케이션은 Windows 시스템의 레지스트리와 같은 외부 원본을 사용하여 제대로 작동할 필요가 없습니다. 어셈블리는 *.dll* 충돌을 줄이고 애플리케이션을 보다 안정적이고 쉽게 배포할 수 있도록 합니다. 대부분의 경우 . NET 기반 애플리케이션은 단순히 해당 파일을 대상 컴퓨터에 복사합니다. 자세한 내용은 어셈블리 매니페스트를 참조하세요.

어셈블리에 대한 참조 추가

애플리케이션에서 어셈블리를 사용하려면 해당 어셈블리에 대한 참조를 추가해야 합니다. 어셈블리를 참조할 때 해당 네임스페이스의 액세스 가능한 모든 형식, 속성, 메서드 및 기타 멤버를 애플리케이션에서 소스 파일의 일부인 것처럼 사용할 수 있습니다.

> ⓘ **참고**
>
> .NET 클래스 라이브러리의 대부분의 어셈블리는 자동으로 참조됩니다. 시스템 어셈블리가 자동으로 참조되지 않는 경우 다음 방법 중 하나로 참조를 추가합니다.
>
> - .NET 및 .NET Core의 경우 어셈블리가 포함된 NuGet 패키지에 대한 참조를 추가합니다. Visual Studio에서 NuGet 패키지 관리자를 **<사용하거나 어셈블리에 대한**

PackageReference> 요소를 *.csproj* 또는 *.vbproj* 프로젝트에 추가합니다.
- .NET Framework의 경우 Visual Studio의 **참조 추가** 대화 상자 또는 `-reference` 또는 Visual Basic 컴파일러에 대한 명령줄 옵션을 사용하여 어셈블리에 대한 참조를 추가합니다.

C#에서는 단일 애플리케이션에서 동일한 어셈블리의 두 버전을 사용할 수 있습니다. 자세한 내용은 extern 별칭을 참조하세요.

관련 콘텐츠

테이블 확장

제목	설명
어셈블리 내용	어셈블리를 구성하는 요소입니다.
어셈블리 매니페스트	어셈블리 매니페스트의 데이터 및 어셈블리에 저장되는 방법
글로벌 어셈블리 캐시	GAC에서 어셈블리를 저장하고 사용하는 방법입니다.
강력한 이름의 어셈블리	강력한 이름의 어셈블리의 특징
어셈블리 보안 고려 사항	어셈블리에서 보안이 작동하는 방식.
어셈블리 버전 관리	.NET Framework 버전 관리 정책의 개요입니다.
어셈블리 배치	어셈블리를 찾아야 할 장소.
어셈블리 및 병렬 실행	여러 버전의 런타임 또는 어셈블리를 동시에 사용합니다.
동적 메서드 및 어셈블리 내보내기	동적 어셈블리를 만드는 방법입니다.
런타임에서 어셈블리를 찾는 방법	.NET Framework가 런타임에 어셈블리 참조를 확인하는 방법입니다.

참고 문헌

System.Reflection.Assembly

참고하십시오

- .NET 어셈블리 파일 형식
- 프렌드 어셈블리
- 참조 어셈블리

- 방법: 어셈블리 로드 및 언로드
- 방법: .NET Core에서 어셈블리 언로드 가능성 사용 및 디버그
- 방법: 파일이 어셈블리인지 확인
- 방법: MetadataLoadContext를 사용하여 어셈블리 콘텐츠 검사

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 21.

어셈블리 콘텐츠

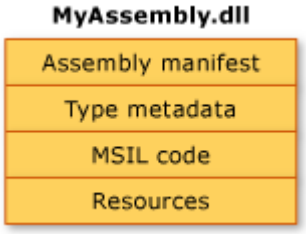
2025. 06. 17.

일반적으로 정적 어셈블리는 다음 네 가지 요소로 구성됩니다.

- 어셈블리 매니페스트는 어셈블리 메타데이터를 포함합니다.
- 메타데이터를 입력합니다.
- 형식을 구현하는 CIL(공용 중간 언어) 코드입니다. 컴파일러에 의해 하나 이상의 소스 코드 파일에서 생성됩니다.
- 리소스 집합입니다.

어셈블리 매니페스트만 필요하지만 어셈블리에 의미 있는 기능을 제공하려면 형식이나 리소스가 필요합니다.

다음 그림에서는 이러한 요소를 단일 물리적 파일로 그룹화하여 보여 줍니다.



소스 코드를 디자인할 때 애플리케이션의 기능을 하나 이상의 파일로 분할하는 방법을 명시적으로 결정합니다. .NET 코드를 디자인할 때 기능을 하나 이상의 어셈블리로 분할하는 방법에 대해 비슷한 결정을 내릴 수 있습니다.

참고하십시오

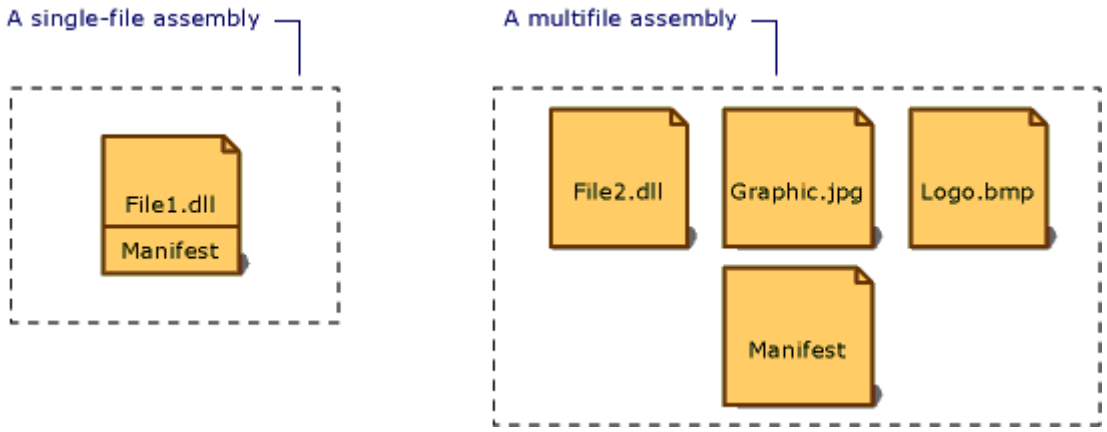
- .NET의 어셈블리
- 어셈블리 매니페스트
- 어셈블리 보안 고려 사항

어셈블리 매니페스트

2025. 06. 17.

정적 또는 동적 어셈블리의 모든 어셈블리에는 어셈블리의 요소가 서로 어떻게 관련되는지 설명하는 데이터 컬렉션이 포함되어 있습니다. 어셈블리 매니페스트에는 이 어셈블리 메타데이터가 포함됩니다. 어셈블리 매니페스트에는 어셈블리의 버전 요구 사항 및 보안 ID를 지정하는 데 필요한 모든 메타데이터와 어셈블리의 범위를 정의하고 리소스 및 클래스에 대한 참조를 확인하는 데 필요한 모든 메타데이터가 포함됩니다. 어셈블리 매니페스트는 CIL(공용 중간 언어) 코드가 있는 PE 파일(*.exe* 또는 *.dll*) 또는 어셈블리 매니페스트 정보만 포함하는 독립 실행형 PE 파일에 저장할 수 있습니다.

다음 그림에서는 매니페스트를 저장할 수 있는 다양한 방법을 보여 줍니다.



연결된 파일이 하나 있는 어셈블리의 경우 매니페스트가 PE 파일에 통합되어 단일 파일 어셈블리를 형성합니다. 독립 실행형 매니페스트 파일 또는 어셈블리의 PE 파일 중 하나에 통합된 매니페스트를 사용하여 다중 파일 어셈블리를 만들 수 있습니다.

각 어셈블리의 매니페스트는 다음 함수를 수행합니다.

- 어셈블리를 구성하는 파일을 열거합니다.
- 어셈블리의 형식 및 리소스에 대한 참조가 해당 선언 및 구현을 포함하는 파일에 매핑되는 방식을 제어합니다.
- 어셈블리가 종속되는 다른 어셈블리를 열거합니다.
- 어셈블리의 소비자와 어셈블리의 구현 세부 정보 간에 간접 참조 수준을 제공합니다.
- 어셈블리가 자체 설명형이 되도록 만듭니다.

어셈블리 매니페스트 내용

다음 표에서는 어셈블리 매니페스트에 포함된 정보를 보여줍니다. 어셈블리 이름, 버전 번호, 문화권 및 강력한 이름 정보의 처음 네 가지 항목은 어셈블리의 ID를 구성합니다.

테이블 확장

정보	설명
어셈블리 이름	어셈블리의 이름을 지정하는 텍스트 문자열입니다.
버전 번호	주 버전 및 부 버전 번호, 수정 버전 및 빌드 번호입니다. 공용 언어 런타임은 이러한 숫자를 사용하여 버전 정책을 적용합니다.
문화	어셈블리가 지원하는 문화권 또는 언어에 대한 정보입니다. 이 정보는 어셈블리를 문화권 또는 언어별 정보를 포함하는 위성 어셈블리로 지정하는 데만 사용해야 합니다. 문화권 정보가 있는 어셈블리는 자동으로 위성 어셈블리로 간주됩니다.
강력한 이름 정보	어셈블리에 강력한 이름이 지정된 경우 게시자의 공개 키입니다.
어셈블리의 모든 파일 목록	어셈블리에 포함된 각 파일의 해시 및 파일 이름입니다. 어셈블리를 구성하는 모든 파일은 어셈블리 매니페스트를 포함하는 파일과 동일한 디렉터리에 있어야 합니다.
형식 참조 정보	런타임에서 형식 참조를 선언 및 구현이 포함된 파일에 매핑하는 데 사용하는 정보입니다. 이것은 어셈블리에서 내보내진 형식에 사용됩니다.
참조된 어셈블리에 대한 정보	어셈블리에서 정적으로 참조하는 다른 어셈블리 목록입니다. 각 참조에는 어셈블리 이름이 강력한 경우 종속 어셈블리의 이름, 어셈블리 메타데이터(버전, 문화권, 운영 체제 등) 및 공개 키가 포함됩니다.

코드에서 어셈블리 특성을 사용하여 어셈블리 매니페스트에서 일부 정보를 추가하거나 변경할 수 있습니다. 상표, 저작권, 제품, 회사 및 정보 버전을 비롯한 버전 정보 및 정보 특성을 변경할 수 있습니다. 어셈블리 특성의 전체 목록은 어 셈블리 특성 설정을 참조하세요.

참고하십시오

- 어셈블리 내용
- 어셈블리 버전 관리
- 위성 어셈블리 만들기
- 강력한 이름의 어셈블리

어셈블리 보안 고려 사항

2025. 06. 17.

어셈블리를 빌드할 때 어셈블리를 실행하는 데 필요한 사용 권한 집합을 지정할 수 있습니다. 증거를 기반으로 어셈블리에 특정 권한이 부여되는지 여부가 결정됩니다.

증거가 사용되는 방법에는 두 가지가 있습니다.

- 입력 증거는 로더가 수집한 증거와 병합되어 정책 해결에 사용되는 최종 증거 집합을 만듭니다. 이 의미 체계를 사용하는 메서드에는 **Assembly.Load**, **Assembly.LoadFrom** 및 **Activator.CreateInstance가 포함됩니다**.
- 입력 증거는 정책 해결에 사용되는 최종 증거 집합으로 변환되지 않고 사용됩니다. 이 의미 체계를 사용하는 메서드에는 **Assembly.Load(byte[])** 및 **AppDomain.DefineDynamicAssembly()**가 포함됩니다.

  어셈블리가 실행되는 컴퓨터의 보안 정책 집합에서 선택적 권한을 부여할 수 있습니다. 코드가 모든 잠재적 보안 예외를 처리하도록 하려면 다음 중 하나를 수행할 수 있습니다.

- 코드에 있어야 하는 모든 권한에 대한 사용 권한 요청을 삽입하고 권한이 부여되지 않은 경우 발생하는 로드 시간 오류를 미리 처리합니다.
- 사용 권한 요청을 사용하여 코드에 필요할 수 있는 권한을 가져오지 말고 권한이 부여되지 않은 경우 보안 예외를 처리하도록 준비해야 합니다.

> ⓘ **참고**
>
> 보안은 복잡한 영역이며 선택할 수 있는 많은 옵션이 있습니다. 자세한 내용은 **주요 보안 개념을 참조하세요**.

로드 시 어셈블리의 증거는 보안 정책에 대한 입력으로 사용됩니다. 보안 정책은 엔터프라이즈 및 컴퓨터의 관리자뿐만 아니라 사용자 정책 설정에 의해 설정되며 실행 시 모든 관리 코드에 부여되는 사용 권한 집합을 결정합니다. 어셈블리 게시자(서명 도구에서 생성된 서명이 있는 경우), 어셈블리가 다운로드된 웹 사이트 및 영역(Internet Explorer 개념) 또는 어셈블리의 강력한 이름에 대해 보안 정책을 설정할 수 있습니다. 예를 들어 컴퓨터 관리자는 웹 사이트에서 다운로드하고 지정된 소프트웨어 회사에서 서명한 모든 코드가 컴퓨터의 데이터베이스에 액세스할 수 있도록 허용하는 보안 정책을 설정할 수 있지만 컴퓨터의 디스크에 쓸 수 있는 액세스 권한을 부여하지는 않습니다.

강력한 이름의 어셈블리 및 서명 도구

> ⚠ **경고**
>
> 보안을 위해 강력한 이름을 사용하지 마세요. 고유한 ID만 제공합니다.

강력한 이름을 사용하거나 SignTool.exe(서명 도구)를 사용하여 서로 다르지만 보완적인 두 가지 방법으로 어셈블리에 서명할 수 있습니다. 강력한 이름으로 어셈블리에 서명하면 어셈블리 매니페스트가 포함된 파일에 공개 키 암호화가 추가됩니다. 강력한 이름 서명은 이름 고유성을 확인하고, 이름 스푸핑을 방지하며, 참조가 확인될 때 호출자에게 ID를 제공하는 데 도움이 됩니다.

강력한 이름과 연결된 신뢰 수준이 없으므로 SignTool.exe(서명 도구) 가 중요합니다. 두 서명 도구를 사용하려면 게시자가 해당 ID를 타사 기관에 증명하고 인증서를 획득해야 합니다. 그러면 이 인증서가 파일에 포함되며 관리자가 코드의 신뢰성을 신뢰할지 여부를 결정하는 데 사용할 수 있습니다.

강력한 이름과SignTool.exe(서명 도구)를 사용하여 만든 디지털 서명을 어셈블리에 모두 지정하거나 단독으로 사용할 수 있습니다. 두 서명 도구는 한 번에 하나의 파일만 서명할 수 있습니다. 다중 파일 어셈블리의 경우 어셈블리 매니페스트가 포함된 파일에 서명합니다. 강력한 이름은 어셈블리 매니페스트를 포함하는 파일에 저장되지만 SignTool.exe(서명 도구) 를 사용하여 만든 서명은 어셈블리 매니페스트를 포함하는 PE(이식 가능한 실행 파일) 파일의 예약된 슬롯에 저장됩니다. SignTool.exe(서명 도구) 생성된 서명을 사용하는 트러스트 계층 구조가 이미 있거나 정책에서 키 부분만 사용하고 신뢰 체인을 확인하지 않는 경우 SignTool.exe(서명 도구)를 사용하여 어셈블리 서명(강력한 이름 포함 또는 제외)을 사용할 수 있습니다.

> ⓘ **참고**
>
> 어셈블리에서 강력한 이름과 서명 도구 서명을 모두 사용하는 경우 먼저 강력한 이름을 할당해야 합니다.

공용 언어 런타임은 해시 확인도 수행합니다. 어셈블리 매니페스트에는 매니페스트가 빌드될 때 존재했던 각 파일의 해시를 포함하여 어셈블리를 구성하는 모든 파일 목록이 포함됩니다. 각 파일이 로드될 때 해당 콘텐츠는 해시되고 매니페스트에 저장된 해시 값과 비교됩니다. 두 해시가 일치하지 않으면 어셈블리가 로드되지 않습니다.

SignTool.exe(서명 도구)를 사용하는 강력한 명명 및 서명은 디지털 서명 및 인증서를 통한 무결성을 보장합니다. 언급된 모든 기술, 즉 해시 확인, 강력한 이름 지정 및 SignTool.exe(서명 도구)를 사용한 서명은 어셈블리가 어떤 방식으로도 변경되지 않았는지 확인하기 위해 함께 작동합니다.

참고하십시오

- 강력한 이름의 어셈블리

- .NET의 어셈블리
- SignTool.exe(서명 도구)

어셈블리 버전 관리

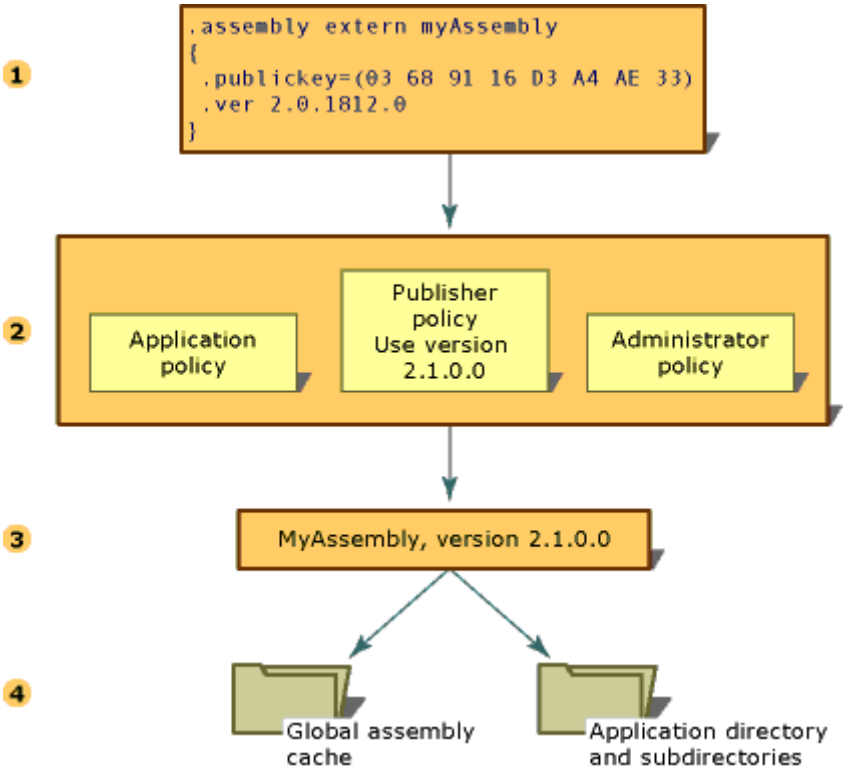
2025. 06. 17.

공용 언어 런타임을 사용하는 어셈블리의 모든 버전 관리가 어셈블리 수준에서 수행됩니다. 어셈블리의 특정 버전과 종속 어셈블리 버전은 어셈블리의 매니페스트에 기록됩니다. 런타임에 대한 기본 버전 정책은 구성 파일(애플리케이션 구성 파일, 게시자 정책 파일 및 컴퓨터의 관리자 구성 파일)에서 명시적 버전 정책으로 재정의되지 않는 한 애플리케이션이 빌드 및 테스트된 버전에서만 실행되는 것입니다.

런타임은 어셈블리 바인딩 요청을 해결하기 위해 여러 단계를 수행합니다.

1. 바인딩할 어셈블리의 버전을 확인하려면 원래 어셈블리 참조를 확인합니다.
2. 버전 정책을 적용하기 위해 적용 가능한 모든 구성 파일을 확인합니다.
3. 원래 어셈블리 참조 및 구성 파일에 지정된 리디렉션에서 올바른 어셈블리를 결정하고 호출 어셈블리에 바인딩해야 하는 버전을 결정합니다.
4. 전역 어셈블리 캐시, 구성 파일에 지정된 코드베이스를 확인한 다음 런타임에서 어셈블리를 찾는 방법에 설명된 검색 규칙을 사용하여 애플리케이션의 디렉터리 및 하위 디렉터리를 확인합니다.

다음 그림에서는 다음 단계를 보여 줍니다.



애플리케이션 구성에 대한 자세한 내용은 앱 구성을 참조하세요. 바인딩 정책에 대한 자세한 내용은 런타임에서 어셈블리를 찾는 방법을 참조하세요.

버전 정보

각 어셈블리에는 버전 정보를 표현하는 두 가지 방법이 있습니다.

- 어셈블리 이름 및 문화권 정보와 함께 어셈블리의 버전 번호는 어셈블리 ID의 일부입니다. 이 숫자는 런타임에서 버전 정책을 적용하는 데 사용되며 런타임 시 형식 확인 프로세스에서 핵심적인 역할을 합니다.

- 정보 제공용으로만 포함된 추가 버전 정보를 나타내는 문자열인 정보 버전입니다.

어셈블리 버전 번호

각 어셈블리에는 ID의 일부로 버전 번호가 있습니다. 따라서 버전 번호와 다른 두 어셈블리는 런타임에서 완전히 다른 어셈블리로 간주됩니다. 이 버전 번호는 물리적으로 다음 형식의 네 부분으로 구성된 문자열로 표시됩니다.

<*주 버전*>입니다.<*부 버전*>입니다.<*빌드 번호*>입니다.<*개정판*>

예를 들어 버전 1.5.1254.0은 1을 주 버전으로, 5를 부 버전으로, 1254를 빌드 번호로, 0을 수정 번호로 나타냅니다.

버전 번호는 어셈블리 이름 및 공개 키를 비롯한 다른 ID 정보뿐만 아니라 애플리케이션과 연결된 다른 어셈블리의 관계 및 ID에 대한 정보와 함께 어셈블리 매니페스트에 저장됩니다.

어셈블리가 빌드되면 개발 도구는 어셈블리 매니페스트에서 참조되는 각 어셈블리에 대한 종속성 정보를 기록합니다. 런타임은 관리자, 애플리케이션 또는 게시자가 설정한 구성 정보와 함께 이러한 버전 번호를 사용하여 참조된 어셈블리의 적절한 버전을 로드합니다.

런타임은 버전 관리 목적으로 일반 어셈블리와 강력한 이름의 어셈블리를 구분합니다. 버전 검사는 강력한 이름의 어셈블리에서만 발생합니다.

버전 바인딩 정책을 지정하는 방법에 대한 자세한 내용은 앱 구성을 참조하세요. 런타임에서 버전 정보를 사용하여 특정 어셈블리를 찾는 방법에 대한 자세한 내용은 런타임에서 어셈블리를 찾는 방법을 참조하세요.

어셈블리 정보 버전

정보 버전은 정보 제공 목적으로만 어셈블리에 추가 버전 정보를 연결하는 문자열입니다. 이 정보는 런타임에 사용되지 않습니다. 텍스트 기반 정보 버전은 제품의 마케팅 문헌, 패키징 또는 제품 이름에 해당하며 런타임에는 사용되지 않습니다. 예를 들어 정보 버전은 "공용 언어 런타임 버전 1.0" 또는 "NET Control SP 2"일 수 있습니다. Microsoft Windows의 파일 속성 대화 상자 버전 탭에서 이 정보는 "제품 버전" 항목에 표시됩니다.

> **참고**
>
> 텍스트를 지정할 수 있지만 문자열이 어셈블리 버전 번호에서 사용하는 형식이 아니거나 해당 형식이지만 와일드카드가 포함된 경우 컴파일에 경고 메시지가 나타납니다. 이 경고는 무해합니다.

정보 버전은 사용자 지정 특성을 System.Reflection.AssemblyInformationalVersionAttribute사용하여 표시됩니다. 정보 버전 특성에 대한 자세한 내용은 어셈블리 특성 설정을 참조하세요.

참고하십시오

- 런타임에서 어셈블리를 찾는 방법
- 앱 구성
- 어셈블리 특성 설정
- .NET의 어셈블리

System.Version 클래스

아티클 • 2025. 04. 01.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

Version 클래스는 어셈블리, 운영 체제 또는 공용 언어 런타임의 버전 번호를 나타냅니다. 버전 번호는 주, 부, 빌드 및 수정의 2~4개 구성 요소로 구성됩니다. 주 구성 요소와 부 구성 요소가 필요합니다. 빌드 및 수정 버전 구성 요소는 선택 사항이지만 수정 구성 요소가 정의된 경우 빌드 구성 요소가 필요합니다. 정의된 모든 구성 요소는 0보다 크거나 같은 정수여야 합니다. 버전 번호의 형식은 다음과 같습니다(선택적 구성 요소는 대괄호로 표시됨).

주요.부[.*빌드*[.*수정*]]

구성 요소는 다음과 같이 규칙에 의해 사용됩니다.

- *주*: 이름이 같지만 주 버전이 다른 어셈블리는 서로 교환할 수 없습니다. 버전 번호가 높을수록 이전 버전과의 호환성을 가정할 수 없는 제품의 주요 재작성이 표시될 수 있습니다.
- *부*: 두 어셈블리의 이름과 주 버전 번호가 동일하지만 부 버전 번호가 다른 경우, 이는 이전 버전과의 호환성을 염두에 둔 중요 업데이트를 나타냅니다. 부 버전 번호가 높을수록 제품의 포인트 릴리스 또는 완전히 이전 버전과 호환되는 새 버전의 제품을 나타낼 수 있습니다.
- *빌드*: 빌드 번호의 차이는 동일한 원본의 다시 컴파일을 나타냅니다. 프로세서, 플랫폼 또는 컴파일러가 변경될 때 다른 빌드 번호가 사용될 수 있습니다.
- *수정 버전*: 이름, 주 버전 및 부 버전 번호가 같지만 수정 버전이 다른 어셈블리는 완전히 교환 가능하도록 의도되었습니다. 이전에 릴리스된 어셈블리의 보안 허점을 수정하는 빌드에서 더 높은 수정 번호를 사용할 수 있습니다.

빌드 또는 수정 번호에 의해서만 다른 어셈블리의 후속 버전은 이전 버전의 핫픽스 업데이트로 간주됩니다.

> ⓘ **중요**
>
> 명시적으로 할당되지 않은 Version 속성 값은 정의되지 않았습니다(-1).

MajorRevision 및 MinorRevision 속성을 사용하면 애플리케이션의 임시 버전을 식별할 수 있습니다. 예를 들어 영구 솔루션을 해제할 때까지 문제를 수정합니다. 또한 Windows NT 운영 체제는 MajorRevision 속성을 사용하여 서비스 팩 번호를 인코딩합니다.

어셈블리에 버전 정보 할당

일반적으로 Version 클래스는 어셈블리에 버전 번호를 할당하는 데 사용되지 않습니다. 대신 AssemblyVersionAttribute 클래스는 이 문서의 예제에 설명된 대로 어셈블리의 버전을 정의하는 데 사용됩니다.

버전 정보 검색

Version 개체는 일부 시스템 또는 애플리케이션 구성 요소(예: 운영 체제), 공용 언어 런타임, 현재 애플리케이션의 실행 파일 또는 특정 어셈블리에 대한 버전 정보를 저장하는 데 가장 자주 사용됩니다. 다음 예제에서는 가장 일반적인 몇 가지 시나리오를 보여 줍니다.

- 운영 체제 버전을 검색합니다. 다음 예제에서는 OperatingSystem.Version 속성을 사용 하 여 운영 체제의 버전 번호를 검색 합니다.

  C#

  ```
  // Get the operating system version.
  OperatingSystem os = Environment.OSVersion;
  Version ver = os.Version;
  Console.WriteLine($"Operating System: {os.VersionString}
  ({ver.ToString()})");
  ```

- 공용 언어 런타임의 버전을 검색합니다. 다음 예제에서는 Environment.Version 속성을 사용하여 공용 언어 런타임에 대한 버전 정보를 검색합니다.

  C#

  ```
  // Get the common language runtime version.
  Version ver = Environment.Version;
  Console.WriteLine($"CLR Version {ver.ToString()}");
  ```

- 현재 애플리케이션의 어셈블리 버전 검색 다음 예제에서는 Assembly.GetEntryAssembly 메서드를 사용하여 애플리케이션 실행 파일을 나타내는 Assembly 개체에 대한 참조를 가져온 다음 해당 어셈블리 버전 번호를 검색합니다.

  C#

  ```
  using System;
  using System.Reflection;

  public class Example4
  {
     public static void Main()
  ```

```
   {
      // Get the version of the executing assembly (that is, this
assembly).
      Assembly assem = Assembly.GetEntryAssembly();
      AssemblyName assemName = assem.GetName();
      Version ver = assemName.Version;
      Console.WriteLine("Application {0}, Version {1}", assemName.Name,
ver.ToString());
   }
}
```

- 현재 어셈블리의 버전 가져오기 다음 예제에서는 Type.Assembly 속성을 사용 하 여 애플리케이션 진입점을 포함 하는 어셈블리를 나타내는 Assembly 개체에 대 한 참조를 가져온 다음 해당 버전 정보를 검색 합니다.

C#

```
using System;
using System.Reflection;

public class Example3
{
   public static void Main()
   {
      // Get the version of the current assembly.
      Assembly assem = typeof(Example).Assembly;
      AssemblyName assemName = assem.GetName();
      Version ver = assemName.Version;
      Console.WriteLine("{0}, Version {1}", assemName.Name,
ver.ToString());
   }
}
```

- 특정 어셈블리의 버전을 검색합니다. 다음 예제에서는 Assembly.ReflectionOnlyLoadFrom 메서드를 사용하여 특정 파일 이름이 있는 Assembly 개체에 대한 참조를 가져온 다음 해당 버전 정보를 검색합니다. 파일 이름 또는 강력한 이름으로 Assembly 개체를 인스턴스화하는 다른 여러 메서드도 있습니다.

C#

```
using System;
using System.Reflection;

public class Example5
{
   public static void Main()
   {
      // Get the version of a specific assembly.
      string filename = @".\StringLibrary.dll";
```

```
      Assembly assem = Assembly.ReflectionOnlyLoadFrom(filename);
      AssemblyName assemName = assem.GetName();
      Version ver = assemName.Version;
      Console.WriteLine("{0}, Version {1}", assemName.Name,
ver.ToString());
   }
}
```

- ClickOnce 애플리케이션의 게시 버전을 검색합니다. 다음 예제에서는 ApplicationDeployment.CurrentVersion 속성을 사용하여 애플리케이션의 게시 버전을 표시합니다. 성공적으로 실행하려면 예제의 애플리케이션 ID를 설정해야 합니다. Visual Studio 게시 마법사에서 자동으로 처리됩니다.

```
C#

using System;
using System.Deployment.Application;

public class Example
{
   public static void Main()
   {
      Version ver =
ApplicationDeployment.CurrentDeployment.CurrentVersion;
      Console.WriteLine($"ClickOnce Publish Version: {ver}");
   }
}
```

> ⓘ **중요**
>
> ClickOnce 배포용 애플리케이션의 게시 버전은 어셈블리 버전과 완전히 독립적입니다.

버전 개체 비교

CompareTo 메서드를 사용하여 한 Version 개체가 두 번째 Version 개체보다 이전인지, 같은 시간인지 또는 이후인지 여부를 확인할 수 있습니다. 다음 예제에서는 버전 2.1이 버전 2.0보다 늦음임을 나타냅니다.

```
C#

Version v1 = new Version(2, 0);
Version v2 = new Version("2.1");
Console.Write("Version {0} is ", v1);
switch(v1.CompareTo(v2))
{
```

```
      case 0:
         Console.Write("the same as");
         break;
      case 1:
         Console.Write("later than");
         break;
      case -1:
         Console.Write("earlier than");
         break;
   }
   Console.WriteLine($" Version {v2}.");
   // The example displays the following output:
   //       Version 2.0 is earlier than Version 2.1.
```

두 버전이 같아야 하려면 첫 번째 Version 개체의 주, 부, 빌드 및 수정 번호는 두 번째 Version 개체의 주 버전과 동일해야 합니다. Version 개체의 빌드 또는 수정 번호가 정의되지 않은 경우 해당 Version 개체는 빌드 또는 수정 번호가 0인 Version 개체보다 이전으로 간주됩니다. 다음 예제에서는 정의되지 않은 버전 구성 요소가 있는 세 개의 Version 개체를 비교하여 이를 보여 줍니다.

C#

```
using System;

enum VersionTime {Earlier = -1, Same = 0, Later = 1 };

public class Example2
{
   public static void Main()
   {
      Version v1 = new Version(1, 1);
      Version v1a = new Version("1.1.0");
      ShowRelationship(v1, v1a);

      Version v1b = new Version(1, 1, 0, 0);
      ShowRelationship(v1b, v1a);
   }

   private static void ShowRelationship(Version v1, Version v2)
   {
      Console.WriteLine($"Relationship of {v1} to {v2}: {(VersionTime)
v1.CompareTo(v2)}");
   }
}
// The example displays the following output:
//       Relationship of 1.1 to 1.1.0: Earlier
//       Relationship of 1.1.0.0 to 1.1.0: Later
```

어셈블리 및 병렬 실행

2025. 06. 17.

병렬 실행은 동일한 컴퓨터에 여러 버전의 애플리케이션 또는 구성 요소를 저장하고 실행하는 기능입니다. 즉, 동일한 컴퓨터에서 런타임 버전을 사용하는 여러 버전의 런타임과 여러 버전의 애플리케이션 및 구성 요소를 동시에 사용할 수 있습니다. 병렬 실행을 사용하면 애플리케이션이 바인딩하는 구성 요소의 버전을 더 많이 제어하고 애플리케이션에서 사용하는 런타임 버전을 더 많이 제어할 수 있습니다.

동일한 어셈블리의 서로 다른 버전의 병렬 스토리지 및 실행에 대한 지원은 강력한 명명의 필수적인 부분이며 런타임의 인프라에 기본 제공됩니다. 강력한 이름의 어셈블리 버전 번호는 ID의 일부이므로 런타임은 동일한 어셈블리의 여러 버전을 전역 어셈블리 캐시에 저장하고 런타임에 해당 어셈블리를 로드할 수 있습니다.

런타임은 병렬 애플리케이션을 만드는 기능을 제공하지만 병렬 실행은 자동으로 수행되지 않습니다. 병렬 실행을 위한 애플리케이션을 만드는 방법에 대한 자세한 내용은 병렬 실행을 위한 구성 요소를 만들기 위한 지침을 참조하세요.

참고하십시오

- 런타임에서 어셈블리를 찾는 방법
- .NET의 어셈블리

.NET 어셈블리 파일 형식

.NET은 .NET 프로그램을 완전히 설명하고 포함하는 데 사용되는 이진 파일 형식 어 *셈블리*를 정의합니다. 어셈블리는 프로그램 자체뿐만 아니라 모든 종속 라이브러리에도 사용됩니다. .NET 프로그램은 적절한 .NET 구현 이외의 다른 필수 아티팩트를 사용하지 않고 하나 이상의 어셈블리로 실행할 수 있습니다. 운영 체제 API를 비롯한 네이티브 종속성은 별도의 문제이며 .NET 어셈블리 형식 내에 포함되지 않지만 이 형식(예: WinRT)으로 설명되기도 합니다.

> 각 CLI 구성 요소는 해당 구성 요소와 관련된 선언, 구현 및 참조에 대한 메타데이터를 전달합니다. 따라서 구성 요소별 메타데이터를 구성 요소 메타데이터라고 하며, 결과 구성 요소는 ECMA 335 I.9.1, 구성 요소 및 어셈블리에서 자체 설명이라고 합니다.

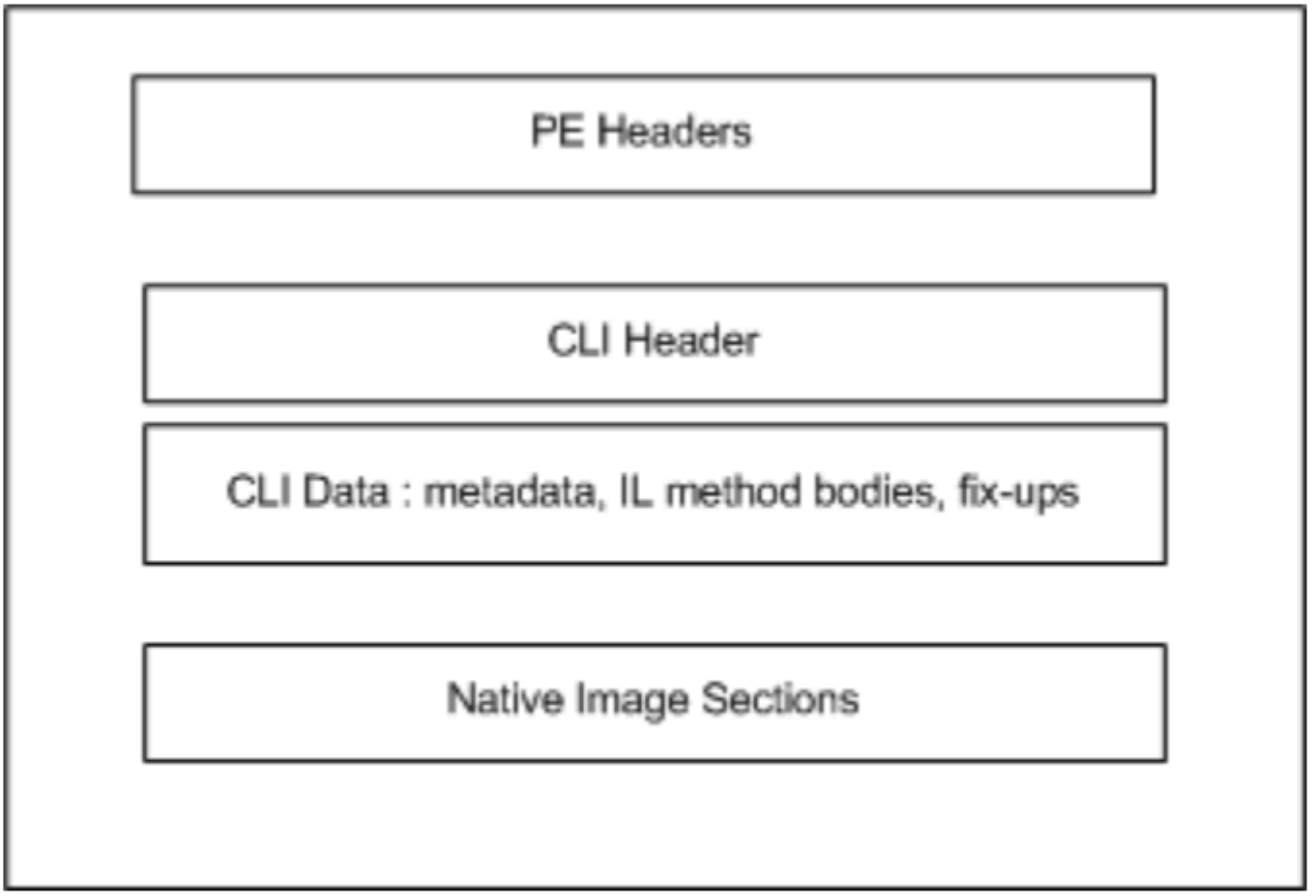
이 형식은 ECMA 335로 완전히 지정되고 표준화됩니다. 모든 .NET 컴파일러 및 런타임은 이 형식을 사용합니다. 문서화되고 자주 업데이트되지 않는 이진 형식의 존재는 상호 운용성에 대한 주요 이점(틀림없이 요구 사항)이었습니다. 이 형식은 제네릭 및 프로세서 아키텍처를 수용하기 위해 2005년(.NET Framework 2.0)에서 실질적인 방식으로 마지막으로 업데이트되었습니다.

형식은 CPU 및 OS에 구애받지 않습니다. 많은 칩과 CPU를 대상으로 하는 .NET 구현의 일부로 사용되었습니다. 형식 자체에는 Windows 유산이 있지만 모든 운영 체제에서 구현할 수 있습니다. OS 상호 운용성을 위한 가장 중요한 선택은 대부분의 값이 little-endian 형식으로 저장된다는 것입니다. 컴퓨터 포인터 크기에 대한 특정 선호도가 없습니다(예: 32비트, 64비트).

.NET 어셈블리 형식은 지정된 프로그램 또는 라이브러리의 구조에 대해서도 매우 설명적입니다. 어셈블리의 내부 구성 요소, 특히 정의된 어셈블리 참조 및 형식과 해당 내부 구조에 대해 설명합니다. 도구 또는 API는 이 정보를 읽고 처리하여 표시하거나 프로그래밍 방식으로 결정할 수 있습니다.

포맷

.NET 이진 형식은 Windows PE 파일 형식을 기반으로 합니다. 실제로 .NET 클래스 라이브러리는 Windows PE를 준수하며, 언뜻 보기에 Windows DLL(동적 링크 라이브러리) 또는 EXE(애플리케이션 실행 파일)로 표시됩니다. 이는 Windows에서 매우 유용한 특성으로, 네이티브 실행 파일 이진 파일로 가장하고 동일한 처리(예: OS 로드, PE 도구)를 얻을 수 있습니다.



ECMA 335 II.25.1의 어셈블리 헤더, 런타임 파일 형식의 구조입니다.

어셈블리 처리

어셈블리를 처리하는 도구 또는 API를 작성할 수 있습니다. 어셈블리 정보를 사용하면 런타임에 프로그래밍 방식으로 의사 결정을 내리고, 어셈블리를 다시 작성하고, 편집기에서 API IntelliSense를 제공하고, 설명서를 생성할 수 있습니다. System.Reflection, System.Reflection.MetadataLoadContext및 Mono.Cecil 은 이러한 용도로 자주 사용되는 도구의 좋은 예입니다.

> ⊗ **주의**
>
> System.Reflection.Metadata 라이브러리이며 PEReader 신뢰할 수 없는 입력을 처리하도록 설계되지 않았습니다. 형식이 잘못되었거나 악의적인 PE 파일은 범위를 벗어난 메모리 액세스, 충돌 또는 중단을 포함하여 예기치 않은 동작을 일으킬 수 있습니다. 신뢰할 수 있는 어셈블리에서만 이러한 API를 사용합니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 19.

.NET에서 어셈블리 언로드 가능성을 사용하고 디버그하는 방법

.NET(Core)에는 어셈블리 집합을 로드하고 나중에 언로드하는 기능이 도입되었습니다. .NET Framework에서 사용자 지정 앱 도메인은 이 용도로 사용되었지만 .NET(Core)은 단일 기본 앱 도메인만 지원합니다.

언로드 기능은 AssemblyLoadContext을 통해 지원됩니다. 어셈블리 집합을 수집 가능한 `AssemblyLoadContext`으로 로드하고, 메서드를 실행하거나 리플렉션을 사용하여 검사하며, 마지막으로 `AssemblyLoadContext`을 언로드합니다. 로드된 어셈블리를 `AssemblyLoadContext`에서 언로드합니다.

AppDomains를 사용하는 `AssemblyLoadContext` 언로드와 사용 간에는 한 가지 중요한 차이점이 있습니다. AppDomains를 사용하면 언로드가 강제로 적용됩니다. 언로드 시 대상 AppDomain에서 실행되는 모든 스레드가 중단되고, 대상 AppDomain에서 만든 관리되는 COM 개체가 제거됩니다. `AssemblyLoadContext`에서는 언로드가 "협력적"입니다. 메서드를 호출하면 AssemblyLoadContext.Unload 언로드가 시작됩니다. 다음 후에 언로드가 완료됩니다.

- 스레드에는 호출 스택에 로드된 어셈블리의 `AssemblyLoadContext` 메서드가 없습니다.
- 어셈블리에 로드된 `AssemblyLoadContext`의 모든 형식, 그 형식의 인스턴스 및 어셈블리 자체는 다음에서 참조되지 않습니다.
  - `AssemblyLoadContext` 외부의 참조, 단 약한 참조(WeakReference 또는 WeakReference<T>)는 제외합니다.
  - 내부 및 외부에서 강력한 GC(가비지 수집기) 핸들(GCHandleType.Normal 또는 `AssemblyLoadContext`)을 처리합니다.

collectible AssemblyLoadContext를 사용

이 섹션에는 .NET(Core) 애플리케이션을 수집 가능한 `AssemblyLoadContext`애플리케이션으로 로드하고 진입점을 실행한 다음 언로드하는 간단한 방법을 보여 주는 자세한 단계별 자습서가 포함되어 있습니다. 에서 전체 샘플을 https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading 찾을 수 있습니다.

수집 가능한 AssemblyLoadContext 만들기

AssemblyLoadContext 클래스를 상속받아 AssemblyLoadContext.Load 메서드를 재정의하십시오. 이 메서드는 해당 `AssemblyLoadContext`에 로드된 어셈블리의 종속성인 모든 어셈블리에 대한 참조를 처리합니다.

다음 코드는 가장 간단한 사용자 지정 `AssemblyLoadContext`의 예입니다.

C#

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {
    }

    protected override Assembly? Load(AssemblyName name)
    {
        return null;
    }
}
```

보시다시피 `Load` 메서드가 `null`을 반환합니다. 즉, 모든 종속성 어셈블리가 기본 컨텍스트로 로드되고 새 컨텍스트에 명시적으로 로드된 어셈블리만 포함됩니다.

종속성의 `AssemblyLoadContext` 일부 또는 전부를 로드하려면 `Load` 메서드에서 `AssemblyDependencyResolver`을 사용할 수 있습니다. `AssemblyDependencyResolver` 어셈블리 이름을 절대 어셈블리 파일 경로로 변환합니다. 확인자는 컨텍스트에 로드된 주 어셈블리의 디렉터리에 있는 *.deps.json* 파일 및 어셈블리 파일을 사용합니다.

C#

```
using System.Reflection;
using System.Runtime.Loader;

namespace complex
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) :
base(isCollectible: true)
        {
            _resolver = new AssemblyDependencyResolver(mainAssemblyToLoadPath);
        }

        protected override Assembly? Load(AssemblyName name)
        {
            string? assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
```

```
    }
}
```

사용자 지정 수집 가능한 AssemblyLoadContext 사용

이 섹션에서는 더 간단한 버전의 `TestAssemblyLoadContext` 사용 중이라고 가정합니다.

다음과 같이 사용자 지정 `AssemblyLoadContext` 인스턴스를 만들고 어셈블리를 로드할 수 있습니다.

C#

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

로드된 어셈블리에서 참조하는 각 어셈블리에 대해 어셈블리 `TestAssemblyLoadContext.Load` 를 가져올 위치를 결정할 수 있도록 `TestAssemblyLoadContext` 메서드가 호출됩니다. 이 경우, 런타임이 기본적으로 어셈블리를 로드하는 데 사용하는 위치에서 기본 컨텍스트로 로드되어야 한다는 것을 나타내기 위해 `null`를 반환합니다.

이제 어셈블리가 로드되었으므로 해당 어셈블리에서 메서드를 실행할 수 있습니다. 메서드를 실행합니다.`Main`

C#

```
var args = new object[1] {new string[] {"Hello"}};
_ = a.EntryPoint?.Invoke(null, args);
```

`Main` 메서드가 반환되면 사용자 지정 `AssemblyLoadContext`에서 `Unload` 메서드를 호출하거나 가지고 있는 `AssemblyLoadContext`에 대한 참조를 제거하여 언로드를 시작할 수 있습니다.

C#

```
alc.Unload();
```

테스트 어셈블리를 언로드하기에 충분합니다. 다음으로, `TestAssemblyLoadContext`, `Assembly`, 및 `MethodInfo`를 스택 슬롯 참조(실제 로컬 또는 JIT에서 도입된 로컬)를 통해 활성 상태로 유지할 수 없도록 `Assembly.EntryPoint`(인라인할 수 없는 별도의 메서드)에 배치하십시오. `TestAssemblyLoadContext`을(를) 활성 상태로 유지하고 언로드를 방지할 수 있습니다.

또한, 나중에 언로드 완료를 감지하기 위해 사용할 수 있도록 `AssemblyLoadContext`에 대한 약한 참조를 반환합니다.

C#

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void ExecuteAndUnload(string assemblyPath, out WeakReference alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    _ = a.EntryPoint?.Invoke(null, args);

    alc.Unload();
}
```

이제 이 함수를 실행하여 어셈블리를 로드, 실행 및 언로드할 수 있습니다.

C#

```
WeakReference testAlcWeakRef;
ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

그러나 언로드가 즉시 완료되지는 않습니다. 앞에서 설명한 것처럼 가비지 수집기를 사용하여 테스트 어셈블리에서 모든 개체를 수집합니다. 대부분의 경우 언로드 완료를 기다릴 필요가 없습니다. 그러나 언로드가 완료되었음을 아는 것이 유용한 경우가 있습니다. 예를 들어 디스크에서 사용자 지정 `AssemblyLoadContext` 에 로드된 어셈블리 파일을 삭제할 수 있습니다. 이러한 경우 다음 코드 조각을 사용할 수 있습니다. 가비지 수집을 시작하고 사용자 지정 `AssemblyLoadContext`에 대한 약한 참조를 설정하여 `null`가 될 때까지 루프에서 대기하면서 보류 중인 종료자를 처리합니다. 이 작업은 대상 개체가 수집되었음을 나타냅니다. 대부분의 경우 루프를 한 번만 통과해야 합니다. 그러나 `AssemblyLoadContext`에서 실행되는 코드가 생성한 개체가 종료자를 가진 경우처럼 더 복잡한 사례에서는 더 많은 패스가 필요할 수 있습니다.

C#

```
for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

제한점

수집 가능한 어셈블리에 로드된 어셈블리는 수집 가능한 `AssemblyLoadContext`어셈블리에 대한 일반적인 제한을 준수해야 합니다. 다음 제한 사항이 추가로 적용됩니다.

- C++/CLI로 작성된 어셈블리는 지원되지 않습니다.
- ReadyToRun 생성 코드는 무시됩니다.

언로드 이벤트

경우에 따라 언로드가 시작될 때 사용자 지정 `AssemblyLoadContext` 에 로드된 코드가 일부 정리를 수행해야 할 수 있습니다. 예를 들어 스레드를 중지하거나 강한 GC 핸들을 처리해야 할 수 있습니다. 이러한 `Unloading` 경우 이벤트를 사용할 수 있습니다. 이 이벤트에 필요한 정리를 수행하는 처리기를 후크할 수 있습니다.

언로드 불가능 문제 해결

언로드의 협조적 특성으로 인해 물건을 수집 가능한 `AssemblyLoadContext` 상태로 유지하고 언로드를 방지할 수 있는 참조를 잊기 쉽습니다. 참조를 보유할 수 있는 엔터티는 다음과 같습니다 (일부는 명백하지 않을 수 있습니다).

- 스택 슬롯 또는 프로세서 레지스터(사용자 코드에서 명시적으로 만들거나 JIT(Just-In-Time) 컴파일러에서 암시적으로 만든 메서드 로컬), 정적 변수, 강력한(고정된) GC 핸들에 있는 일반 참조는, 수집 가능한 `AssemblyLoadContext` 외부에서 유지되며, 전이적으로 가리킵니다.
  - 어셈블리가 수집 가능한 모듈에 로드되었습니다 `AssemblyLoadContext`.
  - 이러한 어셈블리의 유형입니다.
  - 이렇게 어셈블리로부터 나온 형식의 한 인스턴스.
- 수집 가능한 `AssemblyLoadContext`어셈블리에 로드된 어셈블리에서 코드를 실행하는 스레드입니다.
- 수집할 수 있는 `AssemblyLoadContext` 내부에서 생성된 사용자 지정 `AssemblyLoadContext` 형식의 수집할 수 없는 인스턴스입니다.
- RegisteredWaitHandle 인스턴스 보류 중, 콜백이 사용자 지정 `AssemblyLoadContext`의 메서드로 설정됨.
- 로드 가능한`AssemblyLoadContext`에 로드된 형식의 어셈블리, 형식 또는 인스턴스를 참조하는 사용자 지정 `AssemblyLoadContext` 하위 클래스의 필드입니다. 언로드가 진행되는 동안 런타임은 강력한 GC 핸들을 `AssemblyLoadContext`에 보유하여 언로드를 조정합니다. 즉, GC는 직접 참조를 삭제한 후에도 해당 필드 참조를 `AssemblyLoadContext`수집하지 않습니다. 언로드를 완료할 수 있도록 이러한 필드를 지웁다.

> **팁**
>
> 스택 슬롯 또는 프로세서 레지스터에 저장되고 언로드 `AssemblyLoadContext` 를 방지할 수 있는 개체 참조는 다음과 같은 경우에 발생할 수 있습니다.

- 사용자가 만든 지역 변수가 없더라도 함수 호출 결과가 다른 함수에 직접 전달되는 경우
- JIT 컴파일러가 메서드의 특정 지점에서 사용할 수 있는 개체에 대한 참조를 유지하는 경우

언로드 문제 디버깅

언로드와 관련된 디버깅 문제는 지루할 수 있습니다. 오브젝트가 살아 있는 상태를 유지하도록 하는 것이 무엇인지 알 수 없는 상황에 빠질 수 있지만, 그러나 언로드에는 실패합니다. 이를 도와주는 가장 좋은 도구는 SOS 플러그 인을 사용하는 WinDbg(또는 Unix의 LLDB)입니다. 특정 `AssemblyLoadContext`에 속한 `LoaderAllocator`가 계속 유지되는 원인을 찾아야 합니다. SOS 플러그 인을 사용하면 GC 힙 개체, 해당 계층 구조 및 루트를 볼 수 있습니다.

SOS 플러그 인을 디버거에 로드하려면 디버거 명령줄에 다음 명령 중 하나를 입력합니다.

WinDbg에서(아직 로드되지 않은 경우):

콘솔

```
.loadby sos coreclr
```

LLDB에서:

콘솔

```
plugin load /path/to/libsosplugin.so
```

이제 언로드에 문제가 있는 예제 프로그램을 디버그합니다. 소스 코드는 예제 소스 코드 섹션에서 사용할 수 있습니다. WinDbg에서 실행하면 프로그램이 언로드 성공 여부를 확인한 직후 디버거로 중단됩니다. 그런 다음 범인을 찾고 시작할 수 있습니다.

> **팁**
>
> Unix에서 LLDB를 사용하여 디버그하는 경우, 다음 예제의 SOS 명령 앞에 `!`가 없습니다.

콘솔

```
!dumpheap -type LoaderAllocator
```

이 명령은 GC 힙에 있는 형식 이름을 가진 `LoaderAllocator` 모든 개체를 덤프합니다. 예제는 다음과 같습니다.

콘솔

```
         Address               MT     Size
000002b78000ce40 00007ffadc93a288       48
000002b78000ceb0 00007ffadc93a218       24

Statistics:
              MT    Count    TotalSize Class Name
00007ffadc93a218        1           24 System.Reflection.LoaderAllocatorScout
00007ffadc93a288        1           48 System.Reflection.LoaderAllocator
Total 2 objects
```

"'통계:' 부분에서 관심 있는 개체인 `System.Reflection.LoaderAllocator`에 속하는 `MT` (`MethodTable`)을 확인합니다." 그런 다음, 시작 부분의 목록에서 해당 항목과 `MT` 일치하는 항목을 찾고 개체 자체의 주소를 가져옵니다. 이 경우 "000002b78000ce40"입니다.

이제 개체의 `LoaderAllocator` 주소를 알게 되었으므로 다른 명령을 사용하여 해당 GC 루트를 찾을 수 있습니다.

콘솔

```
!gcroot 0x000002b78000ce40
```

이 명령은 `LoaderAllocator` 인스턴스로 이어지는 개체 참조 체인을 덤프합니다. 목록은 루트로 시작합니다. 루트는 `LoaderAllocator`의 생명력을 유지하는 엔터티로, 문제의 핵심입니다. 루트는 스택 슬롯, 프로세서 레지스터, GC 핸들 또는 정적 변수일 수 있습니다.

다음은 명령 출력의 예입니다.`gcroot`

콘솔

```
Thread 4ac:
    000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
        rbp-20: 000000cf9499dd90
            ->  000002b78000d328 System.Reflection.RuntimeMethodInfo
            ->  000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
            ->  000002b78000d1d0 System.RuntimeType
            ->  000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
    000002b7f8a81198 (strong handle)
    -> 000002b78000d948 test.Test
    -> 000002b78000ce40 System.Reflection.LoaderAllocator
```

```
    000002b7f8a815f8 (pinned handle)
    -> 000002b790001038 System.Object[]
    -> 000002b78000d390 example.TestInfo
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.
```

다음 단계는 루트가 있는 위치를 파악하여 해결할 수 있도록 하는 것입니다. 가장 쉬운 경우는 루트가 스택 슬롯 또는 프로세서 레지스터인 경우입니다. 이 경우 `gcroot` 프레임에 루트가 포함된 함수의 이름과 해당 함수를 실행하는 스레드가 표시됩니다. 어려운 경우는 루트가 정적 변수 또는 GC 핸들인 경우입니다.

이전 예제에서 첫 번째 루트는 주소 `System.Reflection.RuntimeMethodInfo` 에 있는 함수 `example.Program.Main(System.String[])` 프레임에 저장된 형식 `rbp-20` 의 로컬입니다(`rbp`프로세서 레지스터 `rbp` 이고 -20 해당 레지스터의 16진수 오프셋임).

두 번째 루트는 클래스 인스턴스 `GCHandle` 에 대한 참조를 보유하는 일반(강력한) `test.Test` 입니다.

세 번째 루트는 고정된 `GCHandle`입니다. 이것은 실제로 정적 변수이지만 불행히도 알 수있는 방법은 없습니다. 참조 형식에 대한 정적은 내부 런타임 구조의 관리되는 개체 배열에 저장됩니다.

언로드를 방지할 수 있는 또 다른 경우는 스레드의 스택에 `AssemblyLoadContext`에 로드된 어셈블리의 메서드 프레임이 있을 때입니다. 모든 스레드의 관리되는 호출 스택을 덤프하여 확인할 수 있습니다.

콘솔

```
~*e !clrstack
```

명령은 "`!clrstack` 명령을 모든 스레드에 적용"을 의미합니다. 다음은 예제에 대한 해당 명령의 출력입니다. 아쉽게도 Unix의 LLDB에는 모든 스레드에 명령을 적용할 수 있는 방법이 없으므로 스레드를 수동으로 전환하고 명령을 반복 `clrstack` 해야 합니다. 디버거가 "관리 스택을 탐색할 수 없음"이라고 표시된 모든 스레드를 무시합니다.

콘솔

```
OS Thread Id: 0x6ba8 (0)
        Child SP               IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[])
```

```
[E:\unloadability\example\Program.cs @ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
        Child SP               IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame: 0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
        Child SP               IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
        Child SP               IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc()
[E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame: 0000001fc727f7f0]
```

볼 수 있듯이 마지막 스레드에는 `test.Program.ThreadProc()`. 이 함수는 로드된 어셈블리의 `AssemblyLoadContext`로부터 온 것이므로 `AssemblyLoadContext`을(를) 활성 상태로 유지합니다.

예제 소스 코드

언로드 가능성 문제가 포함된 다음 코드는 이전 디버깅 예제에서 사용됩니다.

주 테스트 프로그램

C#

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;

namespace example
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly? Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo? mi)
        {
            _entryPoint = mi;
        }

        MethodInfo? _entryPoint;
    }

    class Program
    {
        static TestInfo? entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference
testAlcWeakRef, out MethodInfo? testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a method
```

```
            // for an assembly loaded into the TestAssemblyLoadContext in a static
variable.
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            var oResult = a.EntryPoint?.Invoke(null, args);
            alc.Unload();
            return (oResult is int result) ? result : -1;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a method for
an assembly loaded into the TestAssemblyLoadContext in a local variable
            MethodInfo? testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out
testAlcWeakRef, out testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();

            Console.WriteLine($"Test completed, result={result}, entryPoint:
{testEntryPoint} unload success: {!testAlcWeakRef.IsAlive}");
        }
    }
}
```

TestAssemblyLoadContext에 로드된 프로그램

다음 코드는 주 테스트 프로그램의 메서드에 전달된 `ExecuteAndUnload` 나타냅니다.

C#

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

namespace test
{
    class Test
    {
    }

    class Program
    {
        public static void ThreadProc()
```

```
        {
            // Issue preventing unloading #4 - a thread running method inside of
the TestAssemblyLoadContext at the unload time
            Thread.Sleep(Timeout.Infinite);
        }

        static GCHandle handle;
        static int Main(string[] args)
        {
            // Issue preventing unloading #3 - normal GC handle
            handle = GCHandle.Alloc(new Test());
            Thread t = new Thread(new ThreadStart(ThreadProc));
            t.IsBackground = true;
            t.Start();
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");

            return 1;
        }
    }
}
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 19.

참조 어셈블리

2025. 06. 17.

참조 어셈블리 는 라이브러리의 공용 API 화면을 나타내는 데 필요한 최소 양의 메타데이터만 포함하는 특수 형식의 어셈블리입니다. 빌드 도구에서 어셈블리를 참조할 때 중요한 모든 멤버에 대한 선언을 포함하지만, 해당 API 계약에 영향을 미치지 않는 프라이빗 멤버의 선언과 모든 멤버 구현은 제외됩니다. 반면, 일반 어셈블리를 *구현 어셈블리라고합니다*.

참조 어셈블리는 실행을 위해 로드할 수 없지만 구현 어셈블리와 동일한 방식으로 컴파일러 입력으로 전달될 수 있습니다. 참조 어셈블리는 일반적으로 특정 플랫폼 또는 라이브러리의 SDK(소프트웨어 개발 키트)와 함께 배포됩니다.

참조 어셈블리를 사용하면 개발자가 해당 버전에 대한 전체 구현 어셈블리 없이 특정 라이브러리 버전을 대상으로 하는 프로그램을 빌드할 수 있습니다. 컴퓨터에 일부 라이브러리의 최신 버전만 있지만 해당 라이브러리의 이전 버전을 대상으로 하는 프로그램을 빌드하려고 합니다. 구현 어셈블리에 대해 직접 컴파일하는 경우 이전 버전에서 사용할 수 없는 API 멤버를 실수로 사용할 수 있습니다. 대상 컴퓨터에서 프로그램을 테스트할 때만 이 실수를 발견할 수 있습니다. 이전 버전의 참조 어셈블리에 대해 컴파일하는 경우 컴파일 시간 오류가 즉시 발생합니다.

참조 어셈블리는 구체적인 구현 어셈블리에 해당하지 않는 API 집합인 계약을 나타낼 수도 있습니다. *계약 어셈블리*라고 하는 이러한 참조 어셈블리를 사용하여 동일한 API 집합을 지원하는 여러 플랫폼을 대상으로 지정할 수 있습니다. 예를 들어, .NET Standard는 서로 다른 .NET 플랫폼 간에 공유되는 공통 API 집합을 나타내는 계약 어셈블리 *netstandard.dll*를 제공합니다. 이러한 API의 구현은 .NET Framework의mscorlib.dll또는 .NET Core *의System.Private.CoreLib.dll* 같은 여러 플랫폼의 다양한 어셈블리에 포함되어 있습니다. .NET Standard를 대상으로 하는 라이브러리는 .NET Standard를 지원하는 모든 플랫폼에서 실행할 수 있습니다.

참조 어셈블리 사용

프로젝트에서 특정 API를 사용하려면 해당 어셈블리에 참조를 추가해야 합니다. 구현 어셈블리 또는 참조 어셈블리에 대한 참조를 추가할 수 있습니다. 참조 어셈블리를 사용할 수 있을 때마다 사용하는 것이 좋습니다. 이렇게 하면 API 디자이너에서 사용하기 위해 대상 버전에서 지원되는 API 멤버만 사용하게 됩니다. 참조 어셈블리를 사용하면 구현 세부 정보에 종속되지 않습니다.

.NET Framework 라이브러리에 대한 참조 어셈블리는 대상 지정 팩과 함께 배포됩니다. 독립 실행형 설치 관리자를 다운로드하거나 Visual Studio 설치 관리자에서 구성 요소를 선택하여 가져올 수 있습니다. 자세한 내용은 개발자용 .NET Framework 설치를 참조하세요. .NET Core 및 .NET Standard의 경우 참조 어셈블리는 필요에 따라(NuGet을 통해) 자동으로 다운로드되고 참조됩니다. .NET Core 3.0 이상의 경우 핵심 프레임워크에 대한 참조 어셈블리는 Microsoft.NETCore.App.Ref 패키지에 있습니다(Microsoft.NETCore.App 패키지는 3.0 이전 버전에 대신 사용됨).

참조 추가 대화 상자를 사용하여 Visual Studio에서 .NET Framework 어셈블리에 대한 참조를 추가하면 목록에서 어셈블리를 선택하고 Visual Studio는 프로젝트에서 선택한 대상 프레임워크 버전에 해당하는 참조 어셈블리를 자동으로 찾습니다. 참조 프로젝트 항목을 사용하여 MSBuild 프로젝트에 직접 참조를 추가하는 경우도 마찬가지입니다. 전체 파일 경로가 아닌 어셈블리 이름만 지정하면 됩니다. 컴파일러 옵션(`-reference` 및 Visual Basic)을 사용 하거나 Roslyn API의 메서드를 사용하여 Compilation.AddReferences 명령줄에서 이러한 어셈블리에 대한 참조를 추가하는 경우 올바른 대상 플랫폼 버전에 대한 참조 어셈블리 파일을 수동으로 지정해야 합니다. .NET Framework 참조 어셈블리 파일은 *%ProgramFiles(x86)%\Reference Assemblies\Microsoft\Framework\.NETFramework* 디렉터리에 위치해 있습니다. .NET Core의 경우, 프로젝트 속성을 로 설정하여 게시 작업이 대상 플랫폼의 참조 어셈블리를 출력 디렉터리의 `PreserveCompilationContext` 하위 디렉터리에 복사하도록 강제할 수 있습니다. 그런 다음 이러한 참조 어셈블리 파일을 컴파일러에 전달할 수 있습니다. `DependencyContext`을(를) 사용하여 Microsoft.Extensions.DependencyModel 패키지 내에서 경로를 찾는 데 도움이 될 수 있습니다.

구현이 없으므로 실행을 위해 참조 어셈블리를 로드할 수 없습니다. 이렇게 하려고 하면 System.BadImageFormatException가 발생합니다. 참조 어셈블리의 내용을 검사하려면 .NET Framework의 리플렉션 전용 컨텍스트에 Assembly.ReflectionOnlyLoad 메서드를 사용하여 로드하거나, .NET 및 .NET Framework에 MetadataLoadContext 를 사용할 수 있습니다.

참조 어셈블리 생성

라이브러리 소비자가 다양한 버전의 라이브러리에 대해 프로그램을 빌드해야 하는 경우 라이브러리에 대한 참조 어셈블리를 생성하는 것이 유용할 수 있습니다. 이러한 모든 버전에 대해 구현 어셈블리를 배포하는 것은 크기가 크기 때문에 실용적이지 않을 수 있습니다. 참조 어셈블리의 크기는 더 작으며 라이브러리 SDK의 일부로 배포하면 다운로드 크기가 줄어들고 디스크 공간이 절약됩니다.

또한 IDE 및 빌드 도구는 참조 어셈블리를 활용하여 여러 클래스 라이브러리로 구성된 대규모 솔루션의 경우 빌드 시간을 줄일 수 있습니다. 일반적으로 증분 빌드 시나리오에서 프로젝트는 의존하는 어셈블리를 포함하여 입력 파일이 변경될 때 다시 빌드됩니다. 프로그래머가 멤버의 구현을 변경할 때마다 구현 어셈블리가 변경됩니다. 참조 어셈블리는 공용 API가 영향을 받는 경우에만 변경됩니다. 따라서 참조 어셈블리를 구현 어셈블리 대신 입력 파일로 사용하면 경우에 따라 종속 프로젝트의 빌드를 건너뛸 수 있습니다.

참조 어셈블리를 생성할 수 있습니다.

- MSBuild 프로젝트에서 프로젝트 속성을 사용합니다ProduceReferenceAssembly.
- 명령줄에서 프로그램을 컴파일할 때 (`-refonly`Visual Basic) 또는 / (C# `-refout` /) 컴파일러 옵션을 지정합니다 .

- Roslyn API를 사용할 때, EmitOptions.EmitMetadataOnly를 `true`로 설정하고 EmitOptions.IncludePrivateMembers를 `false`로 설정하여 메서드에 전달된 Compilation.Emit 개체를 사용합니다.

NuGet 패키지를 사용하여 참조 어셈블리를 배포하려면 구현 어셈블리에 사용되는 *lib* 하위 디렉터리가 아닌 패키지 디렉터리 아래의 *ref* 하위 디렉터리에 포함해야 합니다.

참조 어셈블리 구조

참조 어셈블리는 관련 개념인 *메타데이터 전용 어셈블리*의 확장입니다. 메타데이터 전용 어셈블리에는 메서드 본문이 단일 `throw null` 본문으로 대체되지만 익명 형식을 제외한 모든 멤버가 포함됩니다. 본문이 없는 것과 달리 본문을 사용하는 `throw null` 이유는 **PEVerify** 를 실행하고 전달할 수 있기 때문입니다(따라서 메타데이터의 완전성 유효성 검사).

참조 어셈블리는 메타데이터 전용 어셈블리에서 메타데이터(프라이빗 멤버)를 추가로 제거합니다.

- 참조 어셈블리에는 API 화면에 필요한 내용에 대한 참조만 있습니다. 실제 어셈블리에는 특정 구현과 관련된 추가 참조가 있을 수 있습니다. 예를 들어, `class C { private void M() { dynamic d = 1; ... } }`에 대한 참조 어셈블리는 `dynamic`에 필요한 형식을 참조하지 않습니다.
- 프라이빗 함수 멤버(메서드, 속성 및 이벤트)는 제거가 컴파일에 눈에 띄게 영향을 주지 않는 경우 제거됩니다. InternalsVisibleTo 특성이 없으면 내부 함수 멤버도 제거됩니다.

참조 어셈블리의 메타데이터는 다음 정보를 계속 유지합니다.

- 프라이빗 및 중첩 형식을 비롯한 모든 형식입니다.
- 모든 특성, 심지어 내부 특성까지.
- 모든 가상 메서드.
- 명시적 인터페이스 구현.
- 해당 접근자가 가상이기 때문에 명시적으로 구현된 속성 및 이벤트입니다.
- 구조체의 모든 필드입니다.

참조 어셈블리에는 어셈블리 수준 ReferenceAssembly 특성이 포함됩니다. 이 특성은 원본에서 지정할 수 있습니다. 그러면 컴파일러가 합성할 필요가 없습니다. 이 특성 때문에 런타임은 실행을 위해 참조 어셈블리 로드를 거부하지만 리플렉션 전용 모드에서 로드할 수 있습니다.

정확한 참조 어셈블리 구조 세부 정보는 컴파일러 버전에 따라 달라집니다. 최신 버전은 공용 API 화면에 영향을 주지 않는 것으로 판단되는 경우 더 많은 메타데이터를 제외하도록 선택할 수 있습니다.

> ⓘ **참고**

이 섹션의 정보는 C# 버전 7.1 또는 Visual Basic 버전 15.3부터 Roslyn 컴파일러에서 생성된 어셈블리를 참조하는 데만 적용됩니다. .NET Framework 및 .NET Core 라이브러리에 대한 참조 어셈블리의 구조는 참조 어셈블리를 생성하는 자체 메커니즘을 사용하므로 일부 세부 정보가 다를 수 있습니다. 예를 들어 그들은 `throw null` 본문 대신 메서드 본문이 완전히 비어 있을 수 있습니다. 그러나 일반적인 원칙은 여전히 적용됩니다. 사용 가능한 메서드 구현이 없으며 공용 API 관점에서 관찰 가능한 영향을 미치는 멤버에 대해서만 메타데이터를 포함합니다.

참고하십시오

- .NET의 어셈블리
- Framework 대상 지정 개요
- 방법: 참조 관리자를 사용하여 참조 추가 또는 제거

어셈블리 로드 해결

2025. 06. 17.

.NET은 어셈블리 로드를 보다 세게 제어해야 하는 애플리케이션에 대한 이벤트를 제공합니다 AppDomain.AssemblyResolve . 이 이벤트를 처리하여 애플리케이션은 일반 검색 경로 외부에서 로드 컨텍스트로 어셈블리를 로드하고, 로드할 여러 어셈블리 버전을 선택하고, 동적 어셈블리를 내보내고, 반환하는 등의 작업을 수행할 수 있습니다. 이 항목에서는 이벤트 처리 AssemblyResolve 에 대한 지침을 제공합니다.

> ⓘ **참고**
>
> 리플렉션 전용 컨텍스트에서 어셈블리 로드를 확인하려면 대신 이벤트를 사용합니다 AppDomain.ReflectionOnlyAssemblyResolve .

AssemblyResolve 이벤트의 작동 방식

이벤트에 대한 AssemblyResolve 처리기를 등록하면 런타임이 이름으로 어셈블리에 바인딩하지 못할 때마다 처리기가 호출됩니다. 예를 들어 사용자 코드에서 다음 메서드를 호출하면 AssemblyResolve 이벤트가 발생할 수 있습니다.

- AppDomain.Load 메서드 오버로드 또는 Assembly.Load 메서드 오버로드로, 첫 번째 인수가 로드할 어셈블리의 표시 이름을 나타내는 문자열입니다(즉, 속성에서 반환된 Assembly.FullName 문자열).
- AppDomain.Load 메서드 오버로드 또는 Assembly.Load 메서드 오버로드로, 첫 번째 인수가 로드할 어셈블리를 식별하는 개체인 AssemblyName입니다.
- Assembly.LoadWithPartialName 메서드 오버로드입니다.
- AppDomain.CreateInstance 또는 AppDomain.CreateInstanceAndUnwrap 메서드 오버로드는 다른 애플리케이션 도메인 중 하나에서 개체를 인스턴스화합니다.

이벤트 처리기가 수행하는 기능

이벤트에 대한 AssemblyResolve 처리기는 로드할 어셈블리의 표시 이름을 ResolveEventArgs.Name 속성에서 받습니다. 처리기가 어셈블리 이름을 인식하지 못하면 `null` (C#), `Nothing` (Visual Basic) 또는 `nullptr` (Visual C++)을 반환합니다.

처리기가 어셈블리 이름을 인식하는 경우 요청을 충족하는 어셈블리를 로드하고 반환할 수 있습니다. 다음 목록에서는 몇 가지 샘플 시나리오를 설명합니다.

- 처리기가 어셈블리 버전의 위치를 알고 있는 경우 또는 Assembly.LoadFrom 메서드를 사용하여 Assembly.LoadFile 어셈블리를 로드할 수 있으며, 성공적으로 로드된 어셈블리를 반환할 수 있습니다.
- 처리기가 바이트 배열로 저장된 어셈블리 데이터베이스에 액세스할 수 있는 경우 바이트 배열을 사용하는 메서드 오버로드 중 Assembly.Load 하나를 사용하여 바이트 배열을 로드할 수 있습니다.
- 처리기는 동적 어셈블리를 생성하고 반환할 수 있습니다.

> ⓘ **참고**
>
> 처리기는 어셈블리를 load-from 컨텍스트, 로드 컨텍스트, 또는 컨텍스트 없이 로드해야 합니다. 처리기가 Assembly.ReflectionOnlyLoad 방법 또는 Assembly.ReflectionOnlyLoadFrom 메서드를 사용하여 어셈블리를 리플렉션 전용 컨텍스트에 로드하는 경우, AssemblyResolve 이벤트를 발생시킨 로드 시도가 실패합니다.

적절한 어셈블리를 반환하는 것은 이벤트 처리기의 책임입니다. 처리기는 요청된 어셈블리의 표시 이름을 구문 분석할 수 있도록 ResolveEventArgs.Name 속성 값을 AssemblyName(String) 생성자에 전달할 수 있습니다. .NET Framework 4부터 처리기는 속성을 사용하여 ResolveEventArgs.RequestingAssembly 현재 요청이 다른 어셈블리의 종속성인지 여부를 확인할 수 있습니다. 이 정보는 종속성을 충족하는 어셈블리를 식별하는 데 도움이 될 수 있습니다.

이벤트 처리기는 요청된 버전과 다른 버전의 어셈블리를 반환할 수 있습니다.

대부분의 경우 처리기가 반환하는 어셈블리는 처리기가 로드하는 컨텍스트에 관계없이 로드 컨텍스트에 표시됩니다. 예를 들어 처리기가 메서드를 Assembly.LoadFrom 사용하여 어셈블리를 로드 원본 컨텍스트로 로드하는 경우 어셈블리는 처리기가 반환할 때 로드 컨텍스트에 표시됩니다. 그러나 다음 경우 어셈블리는 처리기가 반환할 때 컨텍스트 없이 표시됩니다.

- 처리기는 컨텍스트 없이 어셈블리를 로드합니다.
- 속성이 ResolveEventArgs.RequestingAssembly null이 아닙니다.
- 요청 어셈블리(즉, 속성에서 ResolveEventArgs.RequestingAssembly 반환되는 어셈블리)가 컨텍스트 없이 로드되었습니다.

컨텍스트에 대한 자세한 내용은 메서드 오버로드를 Assembly.LoadFrom(String) 참조하세요.

동일한 어셈블리의 여러 버전을 동일한 애플리케이션 도메인에 로드할 수 있습니다. 형식 할당 문제가 발생할 수 있으므로 이 방법은 권장되지 않습니다. 어셈블리 로드에 대한 모범 사례를 참조하세요.

이벤트 처리기가 수행하지 않아야 하는 사항

이벤트 처리를 AssemblyResolve 위한 기본 규칙은 인식할 수 없는 어셈블리를 반환하려고 시도해서는 안 된다는 것입니다. 처리기를 작성할 때 이벤트를 발생시킬 수 있는 어셈블리를 알아야 합니다. 처리기는 다른 어셈블리에 대해 null을 반환해야 합니다.

> ⓘ **중요**
>
> .NET Framework 4에서 위성 어셈블리에 대한 AssemblyResolve 이벤트가 시작됩니다. 이 변경 내용은 처리기가 모든 어셈블리 로드 요청을 해결하려고 하는 경우 이전 버전의 .NET Framework용으로 작성된 이벤트 처리기에 영향을 줍니다. 인식할 수 없는 어셈블리를 무시하는 이벤트 처리기는 이 변경의 영향을 받지 않습니다. 반환 `null` 되며 일반적인 대체 메커니즘이 뒤따릅니다.

어셈블리를 로드할 때, 이벤트 처리기가 AppDomain.Load 또는 Assembly.Load 메서드 오버로드 중 하나를 사용하지 않도록 해야 합니다. 이러한 사용은 AssemblyResolve 이벤트가 재귀적으로 발생하게 되어 스택 오버플로로 이어질 수 있기 때문입니다. (이 항목의 앞부분에서 제공된 목록을 참조하세요.) 이는 모든 이벤트 처리기가 반환될 때까지 예외가 throw되지 않으므로 부하 요청에 대한 예외 처리를 제공하는 경우에도 발생합니다. 따라서 다음 코드를 찾을 수 없는 경우 `MyAssembly` 스택 오버플로가 발생합니다.

C#

```
using System;
using System.Reflection;

class BadExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;

        try
        {
            object obj = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral,
publicKeyToken=null",
                "MyType");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    static Assembly MyHandler(object source, ResolveEventArgs e)
```

```
    {
        Console.WriteLine("Resolving {0}", e.Name);
        // DO NOT DO THIS: This causes a StackOverflowException
        return Assembly.Load(e.Name);
    }
}

/* This example produces output similar to the following:

Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
...
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null

Process is terminated due to StackOverflowException.
 */
```

AssemblyResolve를 처리하는 올바른 방법

이벤트 처리기에서 어셈블리를 AssemblyResolve 통해 확인하는 경우, 처리기가 StackOverflowException 메서드 호출을 사용하면 결국 Assembly.Load 또는 AppDomain.Load 예외가 발생할 것입니다. 대신 LoadFile 또는 LoadFrom 메서드를 사용하세요. 이 메서드들은 `AssemblyResolve` 이벤트를 발생시키지 않습니다.

`MyAssembly.dll`가 실행 중인 어셈블리 근처에 위치해 있으며, 어셈블리에 대한 경로가 주어질 경우 `Assembly.LoadFile`을 사용하여 이를 해결할 수 있다고 가정해 보십시오.

C#

```
using System;
using System.IO;
using System.Reflection;

class CorrectExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;

        try
        {
            object obj = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral,
publicKeyToken=null",
                "MyType");
        }
        catch (Exception ex)
        {
```

```
            Console.WriteLine(ex.Message);
        }
    }

    static Assembly MyHandler(object source, ResolveEventArgs e)
    {
        Console.WriteLine("Resolving {0}", e.Name);

        var path = Path.GetFullPath("../../MyAssembly.dll");
        return Assembly.LoadFile(path);
     }
}
```

참고하십시오

- 어셈블리 로드 모범 사례
- 애플리케이션 도메인 사용

어셈블리 만들기

2025. 06. 17.

Visual Studio와 같은 IDE 또는 Windows SDK에서 제공하는 컴파일러 및 도구를 사용하여 단일 파일 또는 다중 파일 어셈블리를 만들 수 있습니다. 가장 간단한 어셈블리는 간단한 이름을 가지며 단일 애플리케이션 도메인에 로드되는 단일 파일입니다. 이 어셈블리는 애플리케이션 디렉터리 외부의 다른 어셈블리에서 참조할 수 없으며 버전 검사를 거치지 않습니다. 어셈블리로 구성된 애플리케이션을 제거하려면 어셈블리가 있는 디렉터리를 삭제하기만 하면 됩니다. 많은 개발자에게 이러한 기능이 있는 어셈블리는 애플리케이션을 배포하는 데 필요한 모든 것입니다.

여러 코드 모듈 및 리소스 파일에서 다중 파일 어셈블리를 만들 수 있습니다. 여러 애플리케이션에서 공유할 수 있는 어셈블리를 만들 수도 있습니다. 공유 어셈블리에는 강력한 이름이 있어야 하며 전역 어셈블리 캐시에 배포할 수 있습니다.

다음 요인에 따라 코드 모듈 및 리소스를 어셈블리로 그룹화할 때 몇 가지 옵션이 있습니다.

- 버전 관리

  동일한 버전 정보가 있어야 하는 모듈을 그룹화합니다.

- 배치

  배포 모델을 지원하는 코드 모듈 및 리소스를 그룹화합니다.

- 재사용

  모듈을 논리적으로 함께 사용할 수 있는 경우 모듈을 그룹화합니다. 예를 들어 프로그램 유지 관리에 자주 사용되지 않는 형식 및 클래스로 구성된 어셈블리를 동일한 어셈블리에 배치할 수 있습니다. 또한 여러 애플리케이션과 공유하려는 형식은 어셈블리로 그룹화되어야 하며 어셈블리는 강력한 이름으로 서명되어야 합니다.

- 안전

  동일한 보안 권한이 필요한 형식을 포함하는 모듈을 그룹화합니다.

- 범위 지정

  표시 유형을 동일한 어셈블리로 제한해야 하는 형식을 포함하는 모듈을 그룹화합니다.

관리되지 않는 COM 애플리케이션에서 공용 언어 런타임 어셈블리를 사용할 수 있도록 하는 경우 특별한 고려 사항이 있습니다. 관리되지 않는 코드 작업에 대한 자세한 내용은 COM에 .NET Framework 구성 요소 노출을 참조하세요.

참고하십시오

- 어셈블리 버전 관리
- 방법: 단일 파일 어셈블리 빌드
- 방법: 다중 파일 어셈블리 빌드
- 런타임에서 어셈블리를 찾는 방법
- 다중 파일 어셈블리

어셈블리 이름

2025. 06. 17.

어셈블리의 이름은 메타데이터에 저장되며 어셈블리의 범위와 애플리케이션의 사용에 큰 영향을 미칩니다. 강력한 이름의 어셈블리에는 어셈블리의 이름, 문화권, 공개 키, 버전 번호 및 필요에 따라 프로세서 아키텍처를 포함하는 정규화된 이름이 있습니다. FullName 로드된 어셈블리의 표시 이름이라고도 하는 정규화된 이름을 가져오려면 이 속성을 사용합니다.

런타임은 이름 정보를 사용하여 어셈블리를 찾고 이름이 같은 다른 어셈블리와 구분합니다. 예를 들어 `myTypes`이라는 강력한 이름의 어셈블리는 다음과 같은 전체 정규화된 이름을 가질 수 있습니다.

```
myTypes, Version=1.0.1234.0, Culture=en-US, PublicKeyToken=b77a5c561934e089c,
ProcessorArchitecture=msil
```

이 예제에서 어셈블리의 정규화된 이름은 `myTypes` 공개 키 토큰을 가진 강력한 이름을 가지고 있고, 미국 영어의 문화 값을 가지고 있으며, 버전 번호가 1.0.1234.0임을 나타냅니다. 프로세서 아키텍처는 `msil`운영 체제 및 프로세서에 따라 JIT(Just-In-Time)가 32비트 코드 또는 64비트 코드로 컴파일됨을 의미합니다.

> **팁**
>
> 이 `ProcessorArchitecture` 정보를 통해 프로세서별 버전의 어셈블리가 허용됩니다. 프로세서 아키텍처(예: 32비트 및 64비트 프로세서별 버전)에서만 ID가 다른 어셈블리 버전을 만들 수 있습니다. 강력한 이름에는 프로세서 아키텍처가 필요하지 않습니다. 자세한 내용은 AssemblyName.ProcessorArchitecture를 참조하세요.

어셈블리의 형식을 요청하는 코드는 정규화된 어셈블리 이름을 사용해야 합니다. 이를 정규화된 바인딩이라고 합니다. 어셈블리 이름만 지정하는 부분 바인딩은 .NET Framework에서 어셈블리를 참조할 때 허용되지 않습니다.

.NET Framework를 구성하는 어셈블리에 대한 모든 어셈블리 참조에는 어셈블리의 정규화된 이름도 포함되어야 합니다. 예를 들어 버전 1.0에 대한 System.Data .NET Framework 어셈블리에 대한 참조에는 다음이 포함됩니다.

```
System.data, version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

버전은 .NET Framework 버전 1.0과 함께 제공된 모든 .NET Framework 어셈블리의 버전 번호에 해당합니다. .NET Framework 어셈블리의 경우 문화권 값은 항상 중립적이며 공개 키는 위의 예제와 동일합니다.

예를 들어 추적 수신기를 설정하기 위해 구성 파일에 어셈블리 참조를 추가하려면 시스템 .NET Framework 어셈블리의 정규화된 이름을 포함합니다.

XML

```
<add name="myListener" type="System.Diagnostics.TextWriterTraceListener, System,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
initializeData="c:\myListener.log" />
```

참고

런타임은 어셈블리를 바인딩할 때 어셈블리 이름의 대/소문자를 구분하지 않지만, 어셈블리 이름에 사용된 대/소문자는 그대로 유지합니다. Windows SDK의 여러 도구는 어셈블리 이름을 대/소문자를 구분하는 것으로 처리합니다. 최상의 결과를 위해 대/소문자를 구분하는 것처럼 어셈블리 이름을 관리합니다.

애플리케이션 구성 요소 이름 지정

런타임은 어셈블리의 ID를 결정할 때 파일 이름을 고려하지 않습니다. 어셈블리 이름, 버전, 문화권 및 강력한 이름으로 구성된 어셈블리 ID는 런타임에 명확해야 합니다.

예를 들어 *myAssembly.dll* 어셈블리를 참조하는 *myAssembly.exe*호출된 어셈블리가 있는 경우 *myAssembly.exe*실행하면 바인딩이 올바르게 수행됩니다. 그러나 다른 애플리케이션이 *myAssembly.exe*를 메서드AppDomain.ExecuteAssembly를 사용하여 실행하는 경우, `myAssembly` 이 와의 바인딩을 요청할 때 런타임은 `myAssembly`이 이미 로드된 것을 확인합니다. 이 경우 *myAssembly.dll* 로드되지 않습니다. *myAssembly.exe* 요청된 형식 TypeLoadException 을 포함하지 않으므로 발생합니다.

이 문제를 방지하려면 애플리케이션을 구성하는 어셈블리에 동일한 어셈블리 이름이 없거나 이름이 같은 어셈블리를 다른 디렉터리에 배치해야 합니다.

참고

.NET Framework에서 강력한 이름의 어셈블리를 전역 어셈블리 캐시에 배치하는 경우 어셈블리의 파일 이름은 *.exe* 또는 *.dll*같은 파일 이름 확장명을 포함하지 않고 어셈블리 이름과 일치해야 합니다. 예를 들어 어셈블리의 파일 이름이 *myAssembly.dll*경우 어셈블리 이름은

여야 `myAssembly`합니다. 루트 애플리케이션 디렉터리에만 배포된 프라이빗 어셈블리에는 파일 이름과 다른 어셈블리 이름이 있을 수 있습니다.

참고하십시오

- 방법: 어셈블리의 정규화된 이름을 확인하는 방법
- 어셈블리 만들기
- 강력한 이름의 어셈블리
- 글로벌 어셈블리 캐시
- 런타임에서 어셈블리를 찾는 방법

방법: 어셈블리의 정규화된 이름 찾기

아티클 • 2025. 04. 30.

전역 어셈블리 캐시에서 .NET Framework 어셈블리의 정규화된 이름을 검색하려면 전역 어셈블리 캐시 도구(Gacutil.exe)를 사용합니다. 방법: 전역 어셈블리 캐시의 내용을 보는 방법

.NET Core 어셈블리 및 전역 어셈블리 캐시에 없는 .NET Framework 어셈블리의 경우 다음과 같은 여러 가지 방법으로 정규화된 어셈블리 이름을 가져올 수 있습니다.

- 코드를 사용하여 콘솔 또는 변수에 정보를 출력하거나 Ildasm.exe(IL 디스어셈블러)를 사용하여 정규화된 이름을 포함하는 어셈블리의 메타데이터를 검사할 수 있습니다.
- 어셈블리가 애플리케이션에서 이미 로드된 경우, Assembly.FullName 속성의 값을 검색하여 완전한 정규화된 이름을 가져올 수 있습니다. 해당 어셈블리에 정의된 Type의 Assembly 속성을 사용하여 Assembly 개체에 대한 참조를 검색할 수 있습니다. 이 예제에서는 그림을 제공합니다.
- 어셈블리의 파일 시스템 경로를 알고 있는 경우 (C#) 또는 `Shared` (Visual Basic) AssemblyName.GetAssemblyName 메서드를 호출 `static` 하여 정규화된 어셈블리 이름을 가져올 수 있습니다. 다음은 간단한 예제입니다.

  C#

  ```
  using System;
  using System.Reflection;

  public class Example
  {
     public static void Main()
     {

  Console.WriteLine(AssemblyName.GetAssemblyName(@".\UtilityLibrary.dll"));
     }
  }
  // The example displays output like the following:
  //   UtilityLibrary, Version=1.1.0.0, Culture=neutral, PublicKeyToken=null
  ```

- Ildasm.exe(IL 디스어셈블러)를 사용하여 정규화된 이름을 포함하는 어셈블리의 메타데이터를 검사할 수 있습니다.

버전, 문화권 및 어셈블리 이름과 같은 어셈블리 특성을 설정하는 방법에 대한 자세한 내용은 어셈블리 특성 설정을 참조하세요. 어셈블리에 강력한 이름을 지정하는 방법에 대한 자세한 내용은 강력한 이름의 어셈블리 만들기 및 사용을 참조하세요.

예시

다음 예제에서는 콘솔에 지정 된 클래스를 포함 하는 어셈블리의 정규화된 이름을 표시 하는 방법을 보여줍니다. Type.Assembly 속성은 해당 어셈블리에 정의된 형식으로부터 어셈블리에 대한 참조를 취득하는 데 사용됩니다.

C#

```
using System;
using System.Reflection;

class asmname
{
    public static void Main()
    {
        Type t = typeof(System.Data.DataSet);
        string s = t.Assembly.FullName.ToString();
        Console.WriteLine("The fully qualified assembly name " +
            "containing the specified class is {0}.", s);
    }
}
```

참고하십시오

- 어셈블리 이름
- 어셈블리 만들기
- 강력한 이름의 어셈블리 만들기 및 사용
- 전역 어셈블리 캐시
- 런타임에서 어셈블리를 찾는 방법

어셈블리 위치

2025. 06. 17.

어셈블리의 위치는 공용 언어 런타임이 참조될 때 해당 어셈블리를 찾을 수 있는지 여부를 결정하고 어셈블리를 다른 어셈블리와 공유할 수 있는지 여부를 결정할 수도 있습니다. 다음 위치에 어셈블리를 배포할 수 있습니다.

- 애플리케이션의 디렉터리 또는 하위 디렉터리입니다.

  어셈블리를 배포하기 위한 가장 일반적인 위치입니다. 애플리케이션 루트 디렉터리의 하위 디렉터리 언어 또는 문화권을 기반으로 할 수 있습니다. 어셈블리에 문화권 특성에 정보가 있는 경우 해당 문화권 이름이 있는 애플리케이션 디렉터리 아래의 하위 디렉터리에 있어야 합니다.

- 전역 어셈블리 캐시

  공용 언어 런타임이 설치될 때마다 설치되는 컴퓨터 전체 코드 캐시입니다. 대부분의 경우 어셈블리를 여러 애플리케이션과 공유하려는 경우 전역 어셈블리 캐시에 배포해야 합니다.

- HTTP 서버에서

  HTTP 서버에 배포된 어셈블리에는 강력한 이름이 있어야 합니다. 애플리케이션 구성 파일의 코드베이스 섹션에서 어셈블리를 가리킵니다.

참고하십시오

- 어셈블리 만들기
- 글로벌 어셈블리 캐시
- 런타임에서 어셈블리를 찾는 방법

코드에서 어셈블리 특성 설정

2025. 06. 17.

어셈블리 특성은 어셈블리에 대한 정보를 제공하는 값입니다. 일반적으로 *AssemblyInfo.cs* 파일에 설정됩니다. 특성은 다음 정보 집합으로 나뉩니다.

- 어셈블리 ID 속성
- 정보 특성
- 어셈블리 매니페스트 특성
- 강력한 이름 특성

이 기사는 코드에서 어셈블리 속성을 추가하는 방법에 중점을 둡니다. 코드가 아닌 프로젝트에 어셈블리 특성을 추가하는 방법에 대한 자세한 내용은 프로젝트 파일에서 어셈블리 특성 설정을 참조하세요.

어셈블리 ID 속성

강력한 이름(해당하는 경우)과 함께 세 가지 특성이 어셈블리의 ID(이름, 버전 및 문화권)를 결정합니다. 이러한 특성은 어셈블리의 전체 이름을 형성하며 코드에서 어셈블리를 참조할 때 필요합니다. 특성을 사용하여 어셈블리의 버전 및 문화권을 설정할 수 있습니다. 컴파일러 또는 어셈블리 링커(Al.exe) 는 어셈블리 매니페스트가 포함된 파일에 따라 어셈블리를 만들 때 이름 값을 설정합니다.

다음 표에서는 버전 및 문화권 특성을 설명합니다.

테이블 확장

어셈블리 ID 특성	설명
AssemblyCultureAttribute	어셈블리가 지원하는 문화권을 나타내는 열거형 필드입니다. 어셈블리는 문화권 독립성도 지정할 수 있으며 기본 문화권에 대한 리소스가 포함되어 있음을 나타냅니다. **메모:** 런타임은 문화권 특성이 null로 설정되지 않은 모든 어셈블리를 위성 어셈블리로 처리합니다. 이러한 어셈블리에는 위성 어셈블리 바인딩 규칙이 적용됩니다. 자세한 내용은 런타임에서 어셈블리를 찾는 방법을 참조하세요.
AssemblyFlagsAttribute	어셈블리를 나란히 실행할 수 있는지 여부와 같은 어셈블리 특성을 설정하는 값입니다.
AssemblyVersionAttribute	*주* 형식의 숫자 값입니다. *minor*. *빌드합니다*. *수정 버전*(예: 2.4.0.0). 공용 언어 런타임은 이 값을 사용하여 강력한 이름의 어셈블리에서 바인딩 작업을 수행합니다. **메모:**AssemblyInformationalVersionAttribute 특성이 어셈블리에 적용되지 않으면 AssemblyVersionAttribute 특성에 지정된 버전 번호가

어셈블리 ID 특성	설명
	Application.ProductVersion, Application.UserAppDataPath, Application.UserAppDataRegistry 속성에서 사용됩니다.

다음 코드 예제에서는 어셈블리에 버전 및 문화권 특성을 적용하는 방법을 보여 있습니다.

C#

```
// Set version number for the assembly.
[assembly:AssemblyVersionAttribute("4.3.2.1")]
// Set culture as German.
[assembly:AssemblyCultureAttribute("de")]
```

정보 특성

정보 특성을 사용하여 어셈블리에 대한 추가 회사 또는 제품 정보를 제공할 수 있습니다. 다음 표에서는 어셈블리에 적용할 수 있는 정보 특성을 설명합니다.

테이블 확장

정보 특성	설명
AssemblyCompanyAttribute	회사 이름을 지정하는 문자열 값입니다.
AssemblyCopyrightAttribute	저작권 정보를 지정하는 문자열 값입니다.
AssemblyFileVersionAttribute	Win32 파일 버전 번호를 지정하는 문자열 값입니다. 이는 일반적으로 어셈블리 버전으로 기본 설정됩니다.
AssemblyInformationalVersionAttribute	공용 언어 런타임에서 사용되지 않는 버전 정보(예: 전체 제품 버전 번호)를 지정하는 문자열 값입니다. **메모:** 이 특성이 어셈블리에 적용되는 경우 런타임에 속성을 사용하여 지정하는 문자열을 Application.ProductVersion 가져올 수 있습니다. 문자열은 Application.UserAppDataPath 및 Application.UserAppDataRegistry 속성에서 제공하는 경로 및 레지스트리 키에서도 사용됩니다.
AssemblyProductAttribute	제품 정보를 지정하는 문자열 값입니다.
AssemblyTrademarkAttribute	상표 정보를 지정하는 문자열 값입니다.

이러한 특성은 어셈블리의 Windows 속성 페이지에 표시되거나 **/win32res** 컴파일러 옵션을 사용하여 재정의하여 Win32 리소스 파일을 지정할 수 있습니다.

어셈블리 매니페스트 특성

어셈블리 매니페스트 특성을 사용하여 제목, 설명, 기본 별칭 및 구성을 비롯한 정보를 어셈블리 매니페스트에 제공할 수 있습니다. 다음 표에서는 어셈블리 매니페스트 특성을 설명합니다.

테이블 확장

어셈블리 매니페스트 특성	설명
AssemblyConfigurationAttribute	Retail 또는 Debug와 같은 어셈블리의 구성을 나타내는 문자열 값입니다. 런타임은 이 값을 사용하지 않습니다.
AssemblyDefaultAliasAttribute	어셈블리를 참조하여 사용할 기본 별칭을 지정하는 문자열 값입니다. 이 값은 어셈블리 자체의 이름이 친숙하지 않은 경우(예: GUID 값) 이름을 제공합니다. 이 값은 전체 어셈블리 이름의 짧은 형식으로 사용할 수도 있습니다.
AssemblyDescriptionAttribute	어셈블리의 특성과 용도를 요약하는 간단한 설명을 지정하는 문자열 값입니다.
AssemblyTitleAttribute	어셈블리의 친숙한 이름을 지정하는 문자열 값입니다. 예를 들어 *comdlg* 라는 어셈블리에 Microsoft Common Dialog Control이라는 제목이 있을 수 있습니다.

강력한 이름 특성

강력한 이름 특성을 사용하여 어셈블리의 강력한 이름을 설정할 수 있습니다. 다음 표에서는 강력한 이름 특성을 설명합니다.

테이블 확장

강력한 이름 특성	설명
AssemblyDelaySignAttribute	지연 서명이 사용되고 있음을 나타내는 부울 값입니다.
AssemblyKeyFileAttribute	공개 키(지연 서명을 사용하는 경우) 또는 이 특성의 생성자에 매개 변수로 전달된 퍼블릭 키와 프라이빗 키를 모두 포함하는 파일의 이름을 나타내는 문자열 값입니다. 파일 이름은 원본 파일 경로가 아닌 출력 파일 경로(*.exe* 또는 *.dll*)를 기준으로 합니다.
AssemblyKeyNameAttribute	이 특성의 생성자에 매개 변수로 전달된 키 쌍을 포함하는 키 컨테이너를 나타냅니다.

다음 코드 예제에서는 지연 서명을 사용하여 *myKey.snk*라는 공개 키 파일을 사용하여 강력한 이름의 어셈블리를 만들 때 적용할 특성을 보여줍니다.

```
C#
```

```
[assembly:AssemblyKeyFileAttribute("myKey.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```

참고하십시오

- 어셈블리 만들기
- 어셈블리 속성 MSBuild 속성

프로젝트 파일에서 어셈블리 특성 설정

MSBuild 속성을 사용하여 패키지 관련 프로젝트 속성을 생성된 코드 파일의 어셈블리 특성으로 변환할 수 있습니다. 또한 MSBuild 항목을 사용하여 생성된 파일에 임의의 어셈블리 특성을 추가할 수 있습니다.

어셈블리 특성으로 패키지 속성 사용

MSBuild 속성은GenerateAssemblyInfo 프로젝트의 특성 생성을 제어 `AssemblyInfo` 합니다. 값이 `GenerateAssemblyInfo` 기본값이 `true` 면 패키지 관련 프로젝트 속성 이 어셈블리 특성으로 변환됩니다. 다음 표에서는 특성을 생성하는 프로젝트 속성을 나열합니다. 또한 특성별로 해당 생성을 사용하지 않도록 설정하는 데 사용할 수 있는 속성도 나열합니다. 예를 들면 다음과 같습니다.

XML

```xml
<PropertyGroup>

<GenerateNeutralResourcesLanguageAttribute>false</GenerateNeutralResourcesLanguageAttribute>
</PropertyGroup>
```

테이블 확장

MSBuild 속성	어셈블리 특성	특성 생성을 사용하지 않도록 설정하는 속성
`Company`	AssemblyCompanyAttribute	`GenerateAssemblyCompanyAttribute`
`Configuration`	AssemblyConfigurationAttribute	`GenerateAssemblyConfigurationAttribute`
`Copyright`	AssemblyCopyrightAttribute	`GenerateAssemblyCopyrightAttribute`
`Description`	AssemblyDescriptionAttribute	`GenerateAssemblyDescriptionAttribute`
`FileVersion`	AssemblyFileVersionAttribute	`GenerateAssemblyFileVersionAttribute`
`InformationalVersion`	AssemblyInformationalVersionAttribute	`GenerateAssemblyInformationalVersionAttribute`
`Product`	AssemblyProductAttribute	`GenerateAssemblyProductAttribute`
`AssemblyTitle`	AssemblyTitleAttribute	`GenerateAssemblyTitleAttribute`
`AssemblyVersion`	AssemblyVersionAttribute	`GenerateAssemblyVersionAttribute`
`NeutralLanguage`	NeutralResourcesLanguageAttribute	`GenerateNeutralResourcesLanguageAttribute`
`SourceRevisionId`	AssemblyInformationalVersionAttribute	N/A

이러한 설정에 대한 참고 사항:

- `AssemblyVersion` 및 `FileVersion` 기본값은 접미사가 없는 값 `$(Version)` 입니다. 예를 들어 이 경우 `$(Version)` `1.2.3-beta.4`값은 다음과 같습니다 `1.2.3`.
- `InformationalVersion`의 기본값은 `$(Version)`의 값으로 설정됩니다.
- .NET 8 SDK 또는 그 이후 버전으로 빌드할 때 `SourceRevisionId`는 항상 `InformationalVersion` 에 추가됩니다. .NET 8 SDK에는 git 리포지토리 빌드에 대한 커밋 해시로 자동으로 설정하는 `SourceRevisionId` 포함되어 있습니다. 리포지토리가 아닌 빌드의 경우 비어 있습니다 `SourceRevisionId` . `IncludeSourceRevisionInInformationalVersion`을 `false`로 설정하여 이 동작을 비활성화할 수 있습니다.
- `Copyright` 및 `Description` 속성은 NuGet 메타데이터에도 사용됩니다.
- `Configuration`, 이는 기본적으로 `Debug`으로 설정되며, 모든 MSBuild 대상과 공유됩니다. `--configuration` 명령의 `dotnet` 옵션을 통해 설정할 수 있습니다(예: dotnet pack).
- 일부 속성은 NuGet 패키지를 만들 때 사용됩니다. 자세한 내용은 패키지 속성을 참조하세요.

임의의 특성 설정

생성된 파일에도 고유한 어셈블리 특성을 추가할 수 있습니다. 이렇게 하려면 SDK에 만들 특성의 유형을 알려주는 MSBuild 항목을 정의 `<AssemblyAttribute>` 합니다. 이러한 항목에는 해당 특성에 필요한 생성자 매개 변수도 포함되어야 합니다. 예를 들어, System.Reflection.AssemblyMetadataAttribute 특성에는 두 개의 문자열을 사용하는 생성자가 있습니다.

- 임의 값을 설명하는 이름입니다.
- 저장할 값입니다.

MSBuild에 어셈블리가 만들어진 날짜가 포함된 속성이 있는 경우 `Date` 다음 MSBuild 코드를 사용하여 해당 날짜를 어셈블리 특성에 포함하는 데 사용할 `AssemblyMetadataAttribute` 수 있습니다.

XML

```
<ItemGroup>
  <!-- Include must be the fully qualified .NET type name of the Attribute to create. -
->
  <AssemblyAttribute Include="System.Reflection.AssemblyMetadataAttribute">
    <!-- _Parameter1, _Parameter2, etc. correspond to the
        matching parameter of a constructor of that .NET attribute type -->
    <_Parameter1>BuildDate</_Parameter1>
    <_Parameter2>$(Date)</_Parameter2>
  </AssemblyAttribute>
</ItemGroup>
```

이 항목은 .NET SDK에 다음 C#(또는 이와 동등한 F# 또는 Visual Basic)을 어셈블리 수준 특성으로 내보내도록 지시합니다.

C#

```
[assembly: System.Reflection.AssemblyMetadataAttribute("BuildDate", "01/19/2024")]
```

(실제 날짜 문자열은 빌드 시 제공한 문자열입니다.)

특성에 매개 변수 형식이 `System.String`가 아닌 경우, MSBuild `WriteCodeFragment` 작업에서 지원하는 특정 XML 요소 패턴을 사용하여 매개 변수를 지정할 수 있습니다. WriteCodeFragment 작업 - 어셈블리 수준 특성 생성을 참조하세요.

.NET Framework에서 마이그레이션

.NET Framework 프로젝트를 .NET 6 이상으로 마이그레이션하는 경우 중복 어셈블리 정보 파일과 관련된 오류가 발생할 수 있습니다. .NET Framework 프로젝트 템플릿은 어셈블리 정보 특성이 설정된 코드 파일을 만들기 때문입니다. 파일은 일반적으로 *.\Properties\AssemblyInfo.cs* 또는 *.\Properties\AssemblyInfo.vb* 있습니다. 그러나 SDK 스타일 프로젝트는 프로젝트 설정에 따라 이 파일을 *생성* 합니다.

코드를 .NET 6 이상으로 포팅하는 경우 다음 중 하나를 수행합니다.

- 프로젝트 파일에서 `GenerateAssemblyInfo`를 `false`로 설정하여 어셈블리 정보 특성이 포함된 임시 코드 파일의 생성을 비활성화합니다. 이렇게 하면 *AssemblyInfo* 파일을 유지할 수 있습니다.
- *AssemblyInfo* 파일의 설정을 프로젝트 파일로 마이그레이션한 다음 *AssemblyInfo* 파일을 삭제합니다.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

강력한 이름의 어셈블리

어셈블리의 강력한 이름 지정은 어셈블리에 대한 고유 ID를 만들고 어셈블리 충돌을 방지할 수 있습니다.

강력한 이름을 가진 어셈블리는 무엇으로 구성되나요?

강력한 명명된 어셈블리는 어셈블리와 함께 배포된 공개 키와 어셈블리 자체에 해당하는 프라이빗 키를 사용하여 생성됩니다. 어셈블리에는 어셈블리를 구성하는 모든 파일의 이름과 해시가 포함된 어셈블리 매니페스트가 포함됩니다. "강력한 이름을 가진 어셈블리는 동일해야 합니다."

Visual Studio 또는 명령줄 도구를 사용하여 어셈블리에 강력한 이름을 지정할 수 있습니다. 자세한 내용은 방법: 강력한 이름 또는Sn.exe 사용하여 어셈블리 서명 (강력한 이름 도구)을 참조하세요.

강력한 이름의 어셈블리를 만들면 어셈블리의 단순 텍스트 이름, 버전 번호, 선택적 문화권 정보, 디지털 서명 및 서명에 사용되는 프라이빗 키에 해당하는 공개 키가 포함됩니다.

> ⚠ **Warning**
>
> 보안을 위해 강력한 이름을 사용하지 마세요. 고유한 ID만 제공합니다.

어셈블리의 이름을 강력한 이름으로 지정하는 이유는 무엇인가요?

.NET Framework의 경우 강력한 이름의 어셈블리는 다음 시나리오에서 유용합니다.

- 귀하의 어셈블리가 강력한 이름의 어셈블리에서 참조될 수 있도록 설정하거나, 다른 강력한 이름의 어셈블리가 귀하의 어셈블리에 액세스할 수 있도록 `friend` 권한을 부여하려고 합니다.
- 앱은 동일한 어셈블리의 다른 버전에 액세스해야 합니다. 즉, 충돌 없이 동일한 앱 도메인에서 나란히 로드하려면 다른 버전의 어셈블리가 필요합니다. 예를 들어 간단한 이름이 동일한 어셈블리에 API의 다른 확장이 있는 경우 강력한 명명은 어셈블리의 각 버전에 대해 고유한 ID를 제공합니다.
- 어셈블리를 사용하는 앱의 성능에 부정적인 영향을 미치지 않으므로 어셈블리가 도메인 중립이 되도록 합니다. 전역 어셈블리 캐시에 도메인 중립 어셈블리를 설치해야 하므로 강력한 이름을 지정해야 합니다.

- 게시자 정책을 적용하여 앱에 대한 서비스를 중앙 집중화하려고 합니다. 즉, 어셈블리를 전역 어셈블리 캐시에 설치해야 합니다.

.NET Core 및 .NET 5 이상의 경우 강력한 이름의 어셈블리는 중요한 이점을 제공하지 않습니다. 런타임은 강력한 이름 서명의 유효성을 검사하지 않으며 어셈블리 바인딩에 강력한 이름을 사용하지도 않습니다.

오픈 소스 개발자이고 .NET Framework와의 호환성을 높이기 위해 강력한 이름의 어셈블리의 ID 이점을 원하는 경우 어셈블리와 연결된 프라이빗 키를 소스 제어 시스템에 체크 인하는 것이 좋습니다.

참고하십시오

- 글로벌 어셈블리 캐시
- 어떻게 어셈블리에 강력한 이름을 서명하는지
- Sn.exe(강력한 이름 도구)
- 강력한 이름의 어셈블리 만들기 및 사용

Last updated on 2025. 10. 20.

강력한 이름의 어셈블리 만들기 및 사용

강력한 이름은 어셈블리의 ID(단순 텍스트 이름, 버전 번호 및 문화권 정보(제공된 경우)와 공개 키 및 디지털 서명으로 구성됩니다. 해당 프라이빗 키를 사용하여 어셈블리 파일에서 생성됩니다. 어셈블리 파일에는 어셈블리를 구성하는 모든 파일의 이름과 해시가 포함된 어셈블리 매니페스트가 포함되어 있습니다.

> ⚠ **Warning**
>
> 보안을 위해 강력한 이름을 사용하지 마세요. 고유한 ID만 제공합니다.

강력한 이름의 어셈블리는 다른 강력한 이름의 어셈블리의 형식만 사용할 수 있습니다. 그렇지 않으면 강력한 이름의 어셈블리의 무결성이 손상될 것입니다.

> ⓘ **참고 항목**
>
> .NET Core는 강력한 이름의 어셈블리를 지원하고 .NET Core 라이브러리의 모든 어셈블리는 서명되지만 대부분의 타사 어셈블리에는 강력한 이름이 필요하지 않습니다. 자세한 내용은 GitHub에서 **강력한 이름 서명을** 참조하세요.

강력한 이름 시나리오

다음 시나리오에서는 강력한 이름으로 어셈블리에 서명하고 나중에 해당 이름으로 참조하는 프로세스를 간략하게 설명합니다.

1. 어셈블리 A는 다음 방법 중 하나를 사용하여 강력한 이름으로 만들어집니다.
   - Visual Studio와 같은 강력한 이름 만들기를 지원하는 개발 환경 사용
   - 강력한 이름 도구(Sn.exe)를 사용하여 암호화 키 쌍을 만들고 명령줄 컴파일러 또는 어셈블리 링커(Al.exe)를 사용하여 해당 키 쌍을 어셈블리에 할당합니다. Windows SDK는 Sn.exe 및 Al.exe모두 제공합니다.
2. 개발 환경 또는 도구는 개발자의 프라이빗 키를 사용하여 어셈블리 매니페스트가 포함된 파일의 해시에 서명합니다. 이 디지털 서명은 어셈블리 A의 매니페스트가 포함된 PE(이식 가능한 실행 파일) 파일에 저장됩니다.
3. 어셈블리 B는 어셈블리 A의 소비자입니다. 어셈블리 B 매니페스트의 참조 섹션에는 어셈블리 A의 공개 키를 나타내는 토큰이 포함되어 있습니다. 토큰은 전체 공개 키의 일부이며, 공간을 절약하기 위해 키 자체가 아니라 사용됩니다.

4. 공용 언어 런타임은 어셈블리가 전역 어셈블리 캐시에 배치되는 경우 강력한 이름 서명을 확인합니다. 런타임에 강력한 이름으로 바인딩하는 경우 공용 언어 런타임은 어셈블리 B의 매니페스트에 저장된 키를 어셈블리 A의 강력한 이름을 생성하는 데 사용되는 키와 비교합니다. .NET 보안 검사가 통과되고 바인딩이 성공하면 어셈블리 B는 어셈블리 A의 비트가 변조되지 않았으며 이러한 비트가 실제로 어셈블리 A의 개발자로부터 온 것을 보장합니다.

> **참고 항목**
>
> 이 시나리오는 신뢰 문제를 해결하지 않습니다. 어셈블리는 강력한 이름 외에도 전체 Microsoft Authenticode 서명을 전달할 수 있습니다. Authenticode 서명에는 트러스트를 설정하는 인증서가 포함됩니다. 강력한 이름은 코드를 이런 식으로 서명할 필요가 없다는 점에 유의해야 합니다. 강력한 이름은 고유한 ID만 제공합니다.

신뢰할 수 있는 어셈블리의 서명 확인 무시

.NET Framework 3.5 서비스 팩 1부터는 어셈블리가 `MyComputer` 영역에 대한 기본 애플리케이션 도메인과 같은 전체 신뢰 애플리케이션 도메인에 로드될 때 강력한 이름 서명의 유효성이 검사되지 않습니다. 이를 강력한 이름 바이패스 기능이라고 합니다. 완전 신뢰 환경에서는 서명에 관계없이 서명된 완전 신뢰 어셈블리에 대한 StrongNameIdentityPermission 요구가 항상 성공합니다. 강력한 이름 바이패스 기능은 이 상황에서 완전 신뢰 어셈블리의 강력한 이름 서명 확인의 불필요한 오버헤드를 방지하여 어셈블리가 더 빠르게 로드되도록 합니다.

바이패스 기능은 강력한 이름으로 서명되고 다음과 같은 특성을 가진 모든 어셈블리에 적용됩니다.

- 증명 정보 없이도 StrongName를 완전히 신뢰할 수 있습니다(예: `MyComputer` 영역 증명 정보를 갖고 있습니다).
- 완전히 신뢰할 수 있는 에 로드됩니다 AppDomain.
- 해당 ApplicationBase속성 아래 AppDomain 의 위치에서 로드됩니다.
- 지연 서명되지 않았습니다.

이 기능은 개별 애플리케이션 또는 컴퓨터에서 사용하지 않도록 설정할 수 있습니다. 방법: 강력한 이름 바이패스 기능을 사용하지 않도록 설정합니다.

관련 항목

테이블 확장

제목	Description
방법: 퍼블릭-프라이빗 키 쌍 만들기	어셈블리에 서명하기 위한 암호화 키 쌍을 만드는 방법을 설명합니다.
어떻게 어셈블리에 강력한 이름을 서명하는지	강력한 이름의 어셈블리를 만드는 방법을 설명합니다.
향상된 강력한 이름 지정	.NET Framework 4.5의 강력한 이름에 대한 향상된 기능을 설명합니다.
방법: 강력한 이름의 어셈블리 참조	컴파일 시간이나 런타임에 강력한 이름이 있는 어셈블리에서 형식 또는 리소스를 참조하는 방법을 설명합니다.
방법: 강력한 이름 바이패스 기능 사용 안 함	강력한 이름 서명의 유효성 검사를 무시하는 기능을 사용하지 않도록 설정하는 방법을 설명합니다. 이 기능은 전체 또는 특정 애플리케이션에 대해 사용하지 않도록 설정할 수 있습니다.
어셈블리 만들기	단일 파일 및 다중 파일 어셈블리에 대한 개요를 제공합니다.
Visual Studio에서 어셈블리 서명을 지연하는 방법	어셈블리를 만든 후 강력한 이름으로 어셈블리에 서명하는 방법을 설명합니다.
Sn.exe(강력한 이름 도구)	강력한 이름을 가진 어셈블리를 만드는 데 도움이 되는 .NET Framework에 포함된 도구에 대해 설명합니다. 이 도구는 키 관리, 서명 생성 및 서명 확인을 위한 옵션을 제공합니다.
Al.exe(어셈블리 링커)	모듈 또는 리소스 파일에서 어셈블리 매니페스트가 있는 파일을 생성하는 .NET Framework에 포함된 도구에 대해 설명합니다.

Last updated on 2025. 12. 05.

방법: 퍼블릭-프라이빗 키 쌍 만들기

강력한 이름으로 어셈블리에 서명하려면 퍼블릭/프라이빗 키 쌍이 있어야 합니다. 이 퍼블릭 및 프라이빗 암호화 키 쌍은 컴파일 중에 강력한 이름의 어셈블리를 만드는 데 사용됩니다. 강력한 이름 도구(Sn.exe)를 사용하여 키 쌍을 만들 수 있습니다. 키 쌍 파일에는 일반적으로 *.snk* 확장명이 있습니다.

> **참고 항목**
>
> .NET(.NET Core 및 .NET 5 이상)에서는 강력한 이름에 런타임 유효성 검사가 없습니다. 강력한 이름 서명은 주로 .NET Framework 상호 운용성 시나리오를 사용하는 .NET Framework 및 .NET Standard 2.0과 관련이 있습니다. .NET Framework를 대상으로 하지 않는 경우 일반적으로 조직이나 소비자가 요구하지 않는 한 어셈블리의 이름을 강력하게 지정할 필요가 없습니다.

> **참고 항목**
>
> Visual Studio C# 및 Visual Basic 프로젝트 속성 페이지에는 **사인** 탭이 포함되어 있으므로 *Sn.exe* 사용하지 않고 기존 키 파일을 선택하거나 새 키 파일을 생성할 수 있습니다. Visual C++에서 **속성 페이지** 창의 **구성 속성** 섹션에 있는 **링커** 섹션의 **고급** 속성 페이지에서 기존 키 파일의 위치를 지정할 수 있습니다. AssemblyKeyFileAttribute 특성을 사용하여 키 파일 쌍을 식별하는 것은 Visual Studio 2005부터 사용되지 않습니다.

키 쌍 만들기

> **참고 항목**
>
> `Sn.exe` 운영 체제의 .NET SDK에 포함되지 않습니다. Windows에서만 사용할 수 있으며, Visual Studio 또는 Windows SDK를 설치하여 얻을 수 있습니다.

키 쌍을 만들려면 명령 프롬프트에서 다음 명령을 입력합니다.

sn –k<*파일 이름*>

이 명령에서 *파일 이름은* 키 쌍을 포함하는 출력 파일의 이름입니다.

다음 예제에서는 *sgKey.snk*라는 키 쌍을 만듭니다.

Windows 명령 프롬프트

```
sn -k sgKey.snk
```

어셈블리 서명을 지연시키고 전체 키 쌍(테스트 시나리오에서는 불가능)을 제어하는 경우 다음 명령을 사용하여 키 쌍을 생성한 다음 공개 키를 별도의 파일로 추출할 수 있습니다. 먼저 키 쌍을 만듭니다.

Windows 명령 프롬프트

```
sn -k keypair.snk
```

다음으로 키 쌍에서 공개 키를 추출하고 별도의 파일에 복사합니다.

Windows 명령 프롬프트

```
sn -p keypair.snk public.snk
```

키 쌍을 만든 후에는 강력한 이름 서명 도구에서 찾을 수 있는 파일을 배치해야 합니다.

강력한 이름으로 어셈블리에 서명할 때 어셈블리 링커(Al.exe) 는 현재 디렉터리와 출력 디렉터리를 기준으로 키 파일을 찾습니다. 명령줄 컴파일러를 사용하는 경우 코드 모듈이 포함된 현재 디렉터리에 키를 복사하기만 하면 됩니다.

프로젝트 속성에 **Signing** 탭이 없는 이전 버전의 Visual Studio 사용하는 경우 권장되는 키 파일 위치는 다음과 같이 지정된 파일 특성이 있는 프로젝트 디렉터리입니다.

C#

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

참고하십시오

- 강력한 이름의 어셈블리 만들기 및 사용

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 06.

방법: 강력한 이름으로 어셈블리에 서명

> ⓘ **참고 항목**
>
> .NET Core가 강력한 이름의 어셈블리를 지원하고 .NET Core 라이브러리의 모든 어셈블리는 서명되어 있지만 대부분의 타사 어셈블리에는 강력한 이름이 필요하지 않습니다. 자세한 내용은 GitHub의 **강력한 이름 서명**을 참조하세요.

강력한 이름으로 어셈블리에 서명하는 여러 가지 방법이 있습니다.

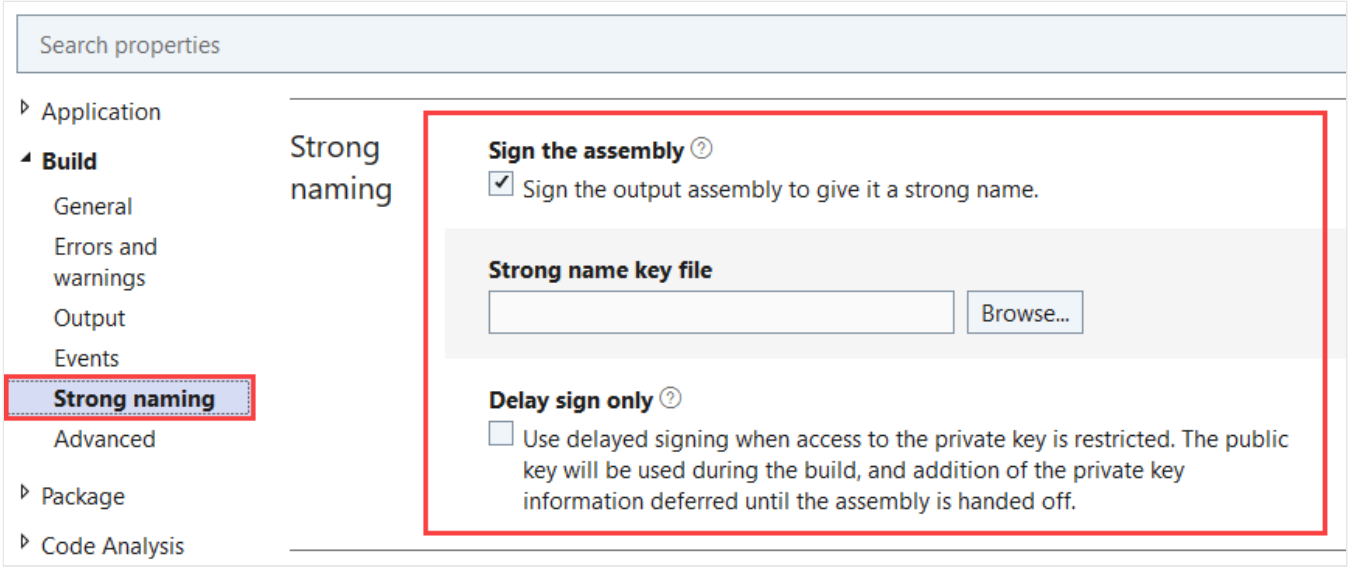
- Visual Studio의 프로젝트에서 프로젝트 디자이너에 **빌드**>**강력한 명명** 페이지를 사용합니다. 이는 가장 쉽고 편리하게 강력한 이름으로 어셈블리에 서명하는 방법입니다.
- 어셈블리 링커(Al.exe)를 사용하여 .NET Framework 코드 모듈(*.netmodule* 파일)을 키 파일과 연결하는 방법.
- 어셈블리 특성을 사용하여 강력한 이름 정보를 코드에 삽입하는 방법. 사용할 키 파일의 위치에 따라 AssemblyKeyFileAttribute 또는 AssemblyKeyNameAttribute 특성을 사용할 수 있습니다.
- 컴파일러 옵션을 사용하는 방법.

강력한 이름으로 어셈블리를 서명하려면 암호화 키 쌍이 있어야 합니다. 키 쌍을 만드는 방법에 대한 자세한 내용은 방법: 프라이빗-퍼블릭 키 쌍 만들기를 참조하세요.

Visual Studio를 사용하여 강력한 이름으로 어셈블리를 만들고 서명

1. **솔루션 탐색기**에서 프로젝트의 바로 가기 메뉴를 열고 **속성**을 선택합니다.
2. **빌드** 탭에 **강력한 명명** 노드가 있습니다.
3. **어셈블리 서명** 확인란을 선택하여 옵션을 확장합니다.
4. **찾아보기** 단추를 선택하여 **강력한 이름 키 파일** 경로를 선택합니다.

> ⓘ **참고 항목**
>
> **어셈블리를 지연 서명**하기 위해 공개 키 파일을 선택합니다.



어셈블리 링커를 사용하여 강력한 이름으로 어셈블리를 만들고 서명

Visual Studio 개발자 명령 프롬프트나 Visual Studio 개발자 PowerShell을 열고 다음 명령을 입력합니다.

al /out:<*assemblyName*> <*moduleName*> **/keyfile:**<*keyfileName*>

여기서

- *assemblyName*은 어셈블리 링커가 내보낼 강력하게 서명된 어셈블리(*.dll* 또는 *.exe* 파일)의 이름입니다.
- *moduleName*은 하나 이상의 형식을 포함하는 .NET Framework 코드 모듈(*.netmodule* 파일)의 이름입니다. C# 또는 Visual Basic에서 `/target:module` 스위치로 코드를 컴파일하여 *.netmodule* 파일을 만들 수 있습니다.
- *keyfileName*은 키 쌍을 포함하는 컨테이너 또는 파일의 이름입니다. 어셈블리 링커는 현재 디렉터리를 기준으로 상대 경로를 해석합니다.

다음 예제에서는 키 쌍 파일 *sgKey.snk*를 사용하여 강력한 이름으로 *MyAssembly.dll* 어셈블리에 서명합니다.

콘솔

```
al /out:MyAssembly.dll MyModule.netmodule /keyfile:sgKey.snk
```

이 도구에 대한 자세한 내용은 어셈블리 링커를 참조하십시오.

특성을 사용하여 강력한 이름으로 어셈블리 서명

1. System.Reflection.AssemblyKeyFileAttribute 또는 AssemblyKeyNameAttribute 특성을 소스 코드 모듈에 추가하고, 강력한 이름으로 어셈블리를 서명할 때 사용할 키 쌍이 포함된 컨테이너 또는 파일의 이름을 지정합니다.

2. 소스 코드 파일을 정상적으로 컴파일합니다.

> **ⓘ 참고 항목**
>
> C# 및 Visual Basic 컴파일러에서는 소스 코드에 AssemblyKeyFileAttribute 또는 AssemblyKeyNameAttribute 특성이 나올 때 컴파일러 경고(각각 CS1699 및 BC41008)를 발생시킵니다. 이런 경고는 무시할 수 있습니다.

다음 예제에서는 어셈블리가 컴파일된 디렉터리에 있는 *keyfile.snk*라는 키 파일과 함께 AssemblyKeyFileAttribute 특성을 사용합니다.

C#

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

또한, 소스 파일을 컴파일할 때 어셈블리 서명을 연기할 수 있습니다. 자세한 내용은 어셈블리 지연 서명을 참조하세요.

컴파일러를 사용하여 강력한 이름으로 어셈블리 서명

C# 및 Visual Basic 컴파일러에서 `/keyfile` 또는 `/delaysign` 컴파일러 옵션을 사용하거나 C++에서 `/KEYFILE` 또는 `/DELAYSIGN` 링커 옵션을 사용하여 소스 코드 파일을 컴파일합니다. 옵션 이름 다음에 콜론과 키 파일의 이름을 추가합니다. 명령줄 컴파일러를 사용할 때, 소스 코드 파일이 포함된 디렉터리에 키 파일을 복사할 수 있습니다.

지연 서명에 대한 자세한 내용은 어셈블리 지연 서명을 참조하세요.

다음 예제에서는 C# 컴파일러를 사용하고 키 파일 *sgKey.snk*를 사용하여 강력한 이름으로 *UtilityLibrary.dll* 어셈블리에 서명합니다.

Windows 명령 프롬프트

```
csc /t:library UtilityLibrary.cs /keyfile:sgKey.snk
```

참고 항목

- 강력한 이름의 어셈블리 만들기 및 사용
- 방법: 퍼블릭/프라이빗 키 쌍 만들기
- AI.exe(어셈블리 링커)
- 어셈블리 지연 서명
- 강력한 이름 API가 PlatformNotSupportedException을 throw함
- 어셈블리 및 매니페스트 서명 관리
- 서명 페이지, 프로젝트 디자이너

Last updated on 2025. 10. 20.

향상된 강력한 이름 지정

2025. 06. 17.

강력한 이름 서명은 어셈블리를 식별하기 위한 .NET Framework의 ID 메커니즘입니다. 일반적으로 발신자(서명자)에서 받는 사람(검증 도구)으로 전달되는 데이터의 무결성을 확인하는 데 사용되는 공개 키 디지털 서명입니다. 이 서명은 어셈블리에 대한 고유 ID로 사용되며 어셈블리에 대한 참조가 모호하지 않도록 합니다. 어셈블리는 빌드 프로세스의 일부로 서명된 다음 로드될 때 확인됩니다.

강력한 이름 서명은 악의적인 당사자가 어셈블리를 변조한 다음 원래 서명자의 키로 어셈블리에 다시 서명하는 것을 방지하는 데 도움이 됩니다. 그러나 강력한 이름 키는 게시자에 대한 신뢰할 수 있는 정보를 포함하지 않으며 인증서 계층 구조를 포함하지도 않습니다. 강력한 이름 서명은 어셈블리에 서명한 사람의 신뢰성을 보장하거나 해당 사용자가 키의 합법적인 소유자인지 여부를 나타내지 않습니다. 키 소유자가 어셈블리에 서명했음을 나타냅니다. 따라서 강력한 이름 서명을 타사 코드를 신뢰하기 위한 보안 유효성 검사기로 사용하지 않는 것이 좋습니다. Microsoft Authenticode는 코드를 인증하는 권장 방법입니다.

기존 강력한 이름의 제한 사항

.NET Framework 4.5 이전 버전에서 사용되는 강력한 명명 기술에는 다음과 같은 단점이 있습니다.

- 키는 지속적으로 공격을 받고 있으며, 향상된 기술과 하드웨어를 통해 공개 키에서 프라이빗 키를 더 쉽게 유추할 수 있습니다. 공격을 방지하려면 더 큰 키가 필요합니다. .NET Framework 4.5 이전의 .NET Framework 버전은 모든 크기 키(기본 크기는 1024비트)로 서명하는 기능을 제공하지만 새 키로 어셈블리에 서명하면 어셈블리의 이전 ID를 참조하는 모든 이진 파일이 중단됩니다. 따라서 호환성을 유지하려는 경우 서명 키의 크기를 업그레이드하기가 매우 어렵습니다.
- 강력한 이름 서명은 SHA-1 알고리즘만 지원합니다. SHA-1은 최근 보안 해시 애플리케이션에 적합하지 않은 것으로 밝혀졌습니다. 따라서 더 강력한 알고리즘(SHA-256 이상)이 필요합니다. SHA-1은 FIPS 규격의 지위를 잃을 수 있으며, 이는 FIPS 규격 소프트웨어 및 알고리즘만 사용하도록 선택하는 사람들에게 문제를 발생시킬 수 있습니다.

향상된 강력한 이름의 장점

향상된 강력한 이름의 주요 장점은 기존 강력한 이름과의 호환성 및 한 ID가 다른 ID와 동일하다고 주장하는 기능입니다.

- 기존 서명된 어셈블리가 있는 개발자는 이전 ID를 참조하는 어셈블리와의 호환성을 유지하면서 해당 ID를 SHA-2 알고리즘으로 마이그레이션할 수 있습니다.

- 새 어셈블리를 만들고 기존 강력한 이름 서명에 관심이 없는 개발자는 보다 안전한 SHA-2 알고리즘을 사용하고 항상 사용하던 것처럼 어셈블리에 서명할 수 있습니다.

향상된 강력한 이름 사용

강력한 이름 키는 서명 키와 ID 키로 구성됩니다. 어셈블리는 서명 키로 서명되고 ID 키로 식별됩니다. .NET Framework 4.5 이전에는 이 두 키가 동일했습니다. .NET Framework 4.5부터 ID 키는 이전 .NET Framework 버전과 동일하게 유지되지만 서명 키는 더 강력한 해시 알고리즘으로 향상됩니다. 또한 서명 키는 ID 키로 서명되어 대리 서명을 생성합니다.

이 AssemblySignatureKeyAttribute 특성을 사용하면 어셈블리 메타데이터가 어셈블리 ID에 기존 공개 키를 사용할 수 있으므로 이전 어셈블리 참조가 계속 작동할 수 있습니다. 이 특성은 AssemblySignatureKeyAttribute 카운터 서명을 사용하여 새 서명 키의 소유자도 이전 ID 키의 소유자인지 확인합니다.

키 마이그레이션 없이 SHA-2로 서명

명령 프롬프트에서 다음 명령을 실행하여 강력한 이름 서명을 마이그레이션하지 않고 어셈블리에 서명합니다.

1. 필요한 경우 새 ID 키를 생성합니다.

```
콘솔
```

```
sn -k IdentityKey.snk
```

2. ID 공개 키를 추출하고 이 키로 서명할 때 SHA-2 알고리즘을 사용해야 한다고 지정합니다.

```
콘솔
```

```
sn -p IdentityKey.snk IdentityPubKey.snk sha256
```

3. ID 공개 키 파일을 사용하여 어셈블리를 지연 서명합니다.

```
콘솔
```

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

4. 전체 ID 키 쌍을 사용하여 어셈블리에 다시 서명합니다.

```
콘솔
```

```
sn -Ra MyAssembly.exe IdentityKey.snk
```

키 마이그레이션을 사용하여 SHA-2로 서명

명령 프롬프트에서 다음 명령을 실행하여 마이그레이션된 강력한 이름 서명으로 어셈블리에 서명합니다.

1. ID 및 서명 키 쌍을 생성합니다(필요한 경우).

   콘솔

   ```
   sn -k IdentityKey.snk
   sn -k SignatureKey.snk
   ```

2. 서명 공개 키를 추출하고 이 키로 서명할 때 SHA-2 알고리즘을 사용해야 한다고 지정합니다.

   콘솔

   ```
   sn -p SignatureKey.snk SignaturePubKey.snk sha256
   ```

3. 카운터 서명을 생성하는 해시 알고리즘을 결정하는 ID 공개 키를 추출합니다.

   콘솔

   ```
   sn -p IdentityKey.snk IdentityPubKey.snk
   ```

4. 특성에 대한 매개 변수를 AssemblySignatureKeyAttribute 생성하고 어셈블리에 특성을 연결합니다.

   콘솔

   ```
   sn -a IdentityPubKey.snk IdentityKey.snk SignaturePubKey.snk
   ```

   그러면 다음과 유사한 출력이 생성됩니다.

   출력

   ```
   Information for key migration attribute.
   (System.Reflection.AssemblySignatureKeyAttribute):
   publicKey=
   002400000c800000940000000602000000240000525341310004000001000100005a3a81ac0a51
   9
   ```

```
d96244a9c589fc147c7d403e40ccf184fc290bdd06c7339389a76b738e255a2bce1d56c3e7e93
6
e4fc87d45adc82ca94c716b50a65d39d373eea033919a613e4341c66863cb2dc622bcb541762b
4
3893434d219d1c43f07e9c83fada2aed400b9f6e44ff05e3ecde6c2827830b8f43f7ac8e3270a
3
4d153cdd

counterSignature=
e3cf7c211678c4d1a7b8fb20276c894ab74c29f0b5a34de4d61e63d4a997222f78cdcbfe4c91e
b
e1ddf9f3505a32edcb2a76f34df0450c4f61e376b70fa3cdeb7374b1b8e2078b121e2ee6e8c6a
8
ed661cc35621b4af53ac29c9e41738f199a81240e8fd478c887d1a30729d34e954a97cddce66e
3
ae5fec2c682e57b7442738
```

그런 다음 이 출력을 AssemblySignatureKeyAttribute로 변환할 수 있습니다.

C#

```
[assembly:System.Reflection.AssemblySignatureKeyAttribute(
"002400000c800000940000000602000000240000525341310004000001000100005a3a81ac0a5
19d96244a9c589fc147c7d403e40ccf184fc290bdd06c7339389a76b738e255a2bce1d56c3e7e
936e4fc87d45adc82ca94c716b50a65d39d373eea033919a613e4341c66863cb2dc622bcb5417
62b43893434d219d1c43f07e9c83fada2aed400b9f6e44ff05e3ecde6c2827830b8f43f7ac8e3
270a34d153cdd",
"e3cf7c211678c4d1a7b8fb20276c894ab74c29f0b5a34de4d61e63d4a997222f78cdcbfe4c91
ebe1ddf9f3505a32edcb2a76f34df0450c4f61e376b70fa3cdeb7374b1b8e2078b121e2ee6e8c
6a8ed661cc35621b4af53ac29c9e41738f199a81240e8fd478c887d1a30729d34e954a97cddce
66e3ae5fec2c682e57b7442738"
)]
```

5. ID 공개 키를 사용하여 어셈블리를 지연 서명합니다.

콘솔

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

6. 서명 키 쌍을 사용하여 어셈블리에 완전히 서명합니다.

콘솔

```
sn -Ra MyAssembly.exe SignatureKey.snk
```

참고하십시오

- 강력한 이름의 어셈블리 만들기 및 사용

방법: 강력한 이름의 어셈블리 참조

강력한 이름의 어셈블리에서 형식이나 리소스를 참조하는 프로세스는 일반적으로 투명합니다. 컴파일 시간(초기 바인딩) 또는 런타임에 참조를 만들 수 있습니다.

컴파일 시간 참조는 컴파일할 어셈블리가 다른 어셈블리를 명시적으로 참조한다고 컴파일러에 표시할 때 발생합니다. 컴파일 시간 참조를 사용하는 경우 컴파일러가 대상으로 지정된 강력한 이름의 어셈블리의 공개 키를 자동으로 가져오고 컴파일되는 어셈블리의 어셈블리 참조에 배치합니다.

> ⓘ **참고 항목**
>
> 강력한 이름의 어셈블리는 다른 강력한 이름의 어셈블리에서 형식만 사용할 수 있습니다. 그러지 않으면 강력한 이름의 어셈블리 보안이 손상됩니다.

강력한 이름의 어셈블리에 대한 컴파일 시간 참조 만들기

명령 프롬프트에서 다음 명령을 입력합니다.

< 컴파일러 명령> **/참조:** *<어셈블리 이름>*

이 명령에서 *compiler command*는 사용되는 언어의 컴파일러 명령이고, *assembly name*은 참조되는 어셈블리의 강력한 이름입니다. 라이브러리 어셈블리를 만들기 위해 **/t:library** 옵션과 같은 다른 컴파일러 옵션을 사용할 수도 있습니다.

다음 예제에서는 *myLibAssembly.dll*이라는 강력한 이름의 어셈블리를 참조하는 *myAssembly.dll*이라는 어셈블리를 *myAssembly.cs*라는 코드 모듈에서 만듭니다.

Windows 명령 프롬프트

```
csc /t:library myAssembly.cs /reference:myLibAssembly.dll
```

강력한 이름의 어셈블리에 대한 런타임 참조 만들기

예를 들어 Assembly.Load 또는 Assembly.GetType 메서드를 사용하여 강력한 이름의 어셈블리에 대한 런타임 참조를 만들 때는, 참조하려는 강력한 이름의 어셈블리의 표시 이름을 반드시 사용해야 합니다. 표시 이름의 구문은 다음과 같습니다.

< 어셈블리 이름>,*<버전 번호>*,*<문화>*,*<공개 키 토큰>*

예시:

콘솔

```
myDll, Version=1.1.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33
```

이 예제에서 `PublicKeyToken`은 16진수 형식의 공개 키 토큰입니다. 문화권 값이 없는 경우 `Culture=neutral`을 사용합니다.

다음 코드 예제에서는 Assembly.Load 메서드와 함께 이 정보를 사용하는 방법을 보여 줍니다.

C#

```
Assembly myDll =
    Assembly.Load("myDll, Version=1.0.0.1, Culture=neutral,
PublicKeyToken=9b35aa32c18d4fb1");
```

다음 강력한 이름(Sn.exe) 명령을 사용하여 특정 어셈블리에 대한 공개 키와 공개 키 토큰의 16진수 형식을 인쇄할 수 있습니다.

sn -Tp <*어셈블리*>

공개 키 파일이 있는 경우 다음 명령을 대신 사용할 수 있습니다(명령줄 옵션의 대/소문자 차이 확인).

sn -tp <*공개 키 파일*>

참고 항목

- 강력한 이름의 어셈블리 만들기 및 사용

Last updated on 2025. 12. 05.

방법: 강력한 이름 바이패스 기능 사용 안 함

.NET Framework의 버전 3.5 서비스 팩 1(SP1)부터는 어셈블리가 AppDomain 영역의 기본 AppDomain 개체와 같은 완전 신뢰 개체에 로드될 때 강력한 이름 서명이 `MyComputer` 검증되지 않습니다. 이를 강력한 이름 바이패스 기능이라고 합니다. 완전 신뢰 환경에서는 서명에 관계없이 서명된 완전 신뢰 어셈블리에 대한 StrongNameIdentityPermission 요구가 항상 성공합니다. 유일한 제한 사항은 해당 영역이 완전히 신뢰할 수 있기 때문에 어셈블리를 완전히 신뢰할 수 있어야 한다는 것입니다. 강력한 이름은 이러한 조건에서 결정 요소가 아니므로 유효성을 검사할 이유가 없습니다. 강력한 이름 서명의 유효성 검사를 무시하면 성능이 크게 향상됩니다.

바이패스 기능은 지연 서명되지 않은 모든 완전 신뢰 어셈블리에 적용되며, 이러한 어셈블리는 해당 속성에 지정된 디렉터리에서 모든 완전 신뢰 AppDomain 로 로드됩니다 ApplicationBase.

레지스트리 키 값을 설정하여 컴퓨터의 모든 애플리케이션에 대한 바이패스 기능을 재정의할 수 있습니다. 애플리케이션 구성 파일을 사용하여 단일 애플리케이션에 대한 설정을 재정의할 수 있습니다. 레지스트리 키에서 사용하지 않도록 설정한 경우 단일 애플리케이션에 대한 바이패스 기능을 복원할 수 없습니다.

바이패스 기능을 재정의하면 강력한 이름은 정확성에 대해서만 유효성이 검사되고, StrongNameIdentityPermission에 대해 확인되지 않습니다. 특정 강력한 이름을 확인하려면 해당 검사를 별도로 수행해야 합니다.

> **Important**
>
> 강력한 이름 유효성 검사를 강제 적용하는 기능은 다음 절차에 설명된 대로 레지스트리 키에 따라 달라집니다. 애플리케이션이 해당 레지스트리 키에 액세스할 수 있는 ACL(액세스 제어 목록) 권한이 없는 계정으로 실행 중인 경우 이 설정은 효과가 없습니다. 모든 어셈블리에 대해 읽을 수 있도록 이 키에 대해 ACL 권한이 구성되어 있는지 확인해야 합니다.

모든 애플리케이션에 대해 강력한 이름 바이패스 기능을 사용하지 않도록 설정

- 32비트 컴퓨터의 시스템 레지스트리에서 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework 키 아래에 이름이 `AllowStrongNameBypass` 0인 DWORD 항목을 만듭니다.
- 64비트 컴퓨터의 시스템 레지스트리에서 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework 및 HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework 키 아래에 이름이 0인 `AllowStrongNameBypass` DWORD 항목을 만듭니다.

단일 애플리케이션에 대해 강력한 이름 무시 기능 사용 안 함

1. 애플리케이션 구성 파일을 열거나 만듭니다.

   이 파일에 대한 자세한 내용은 앱 구성의 Application Configuration Files 섹션을 참조하세요.

2. 다음 항목을 추가합니다.

```xml
<configuration>
  <runtime>
    <bypassTrustedAppStrongNames enabled="false" />
  </runtime>
</configuration>
```

구성 파일 설정을 제거하거나 특성을 .로 설정하여 애플리케이션에 대한 바이패스 기능을 복원할 수 있습니다 `true`.

> **참고 항목**
>
> 컴퓨터에서 바이패스 기능을 활성화한 경우에만 애플리케이션에 대한 강력한 이름 검사를 켜거나 끌 수 있습니다. 컴퓨터에 대한 바이패스 기능이 해제된 경우 모든 애플리케이션에 대해 강력한 이름의 유효성이 검사되며 단일 애플리케이션에 대한 유효성 검사를 무시할 수 없습니다.

참고하십시오

- Sn.exe (Strong Name Tool, 강력한 이름 도구)
- <bypassTrustedAppStrongNames> 요소
- 강력한 이름의 어셈블리 만들기 및 사용

Last updated on 2025. 12. 02.

어셈블리 지연 서명

2025. 06. 17.

조직은 개발자가 매일 액세스할 수 없는 밀접하게 보호된 키 쌍을 가질 수 있습니다. 공개 키는 종종 사용할 수 있지만 프라이빗 키에 대한 액세스는 소수의 개인으로만 제한됩니다. 강력한 이름을 가진 어셈블리를 개발할 때 강력한 이름의 대상 어셈블리를 참조하는 각 어셈블리에는 대상 어셈블리에 강력한 이름을 지정하는 데 사용되는 공개 키의 토큰이 포함됩니다. 이렇게 하려면 개발 프로세스 중에 공개 키를 사용할 수 있어야 합니다.

빌드 시 지연 또는 부분 서명을 사용하여 강력한 이름 서명에 대한 PE(이식 가능한 실행 파일) 파일의 공간을 예약할 수 있지만, 일반적으로 어셈블리를 배송하기 직전에 일부 이후 단계까지 실제 서명을 연기할 수 있습니다.

어셈블리를 지연 서명하려면:

1. 최종 서명을 수행할 조직에서 키 쌍의 공개 키 부분을 가져옵니다. 일반적으로 이 키는 Windows SDK에서 제공하는 *강력한 이름 도구(Sn.exe)*를 사용하여 만들 수 있는 .snk 파일 형식입니다.

2. 어셈블리의 소스 코드에 System.Reflection의 두 사용자 지정 속성을 주석으로 삽입합니다.

   - AssemblyKeyFileAttribute- 공개 키가 포함된 파일의 이름을 해당 생성자에 매개 변수로 전달합니다.

   - AssemblyDelaySignAttribute- **true** 를 해당 생성자에 매개 변수로 전달하여 지연 서명이 사용되고 있음을 나타냅니다.

   다음은 그 예입니다.

   C#

   ```
   [assembly:AssemblyKeyFileAttribute("myKey.snk")]
   [assembly:AssemblyDelaySignAttribute(true)]
   ```

3. 컴파일러는 공개 키를 어셈블리 매니페스트에 삽입하고 PE 파일에서 전체 강력한 이름 서명을 위해 공간을 예약합니다. 이 어셈블리를 참조하는 다른 어셈블리가 자체 어셈블리 참조에 저장할 키를 가져올 수 있도록 어셈블리가 빌드되는 동안 실제 공개 키를 저장해야 합니다.

4. 어셈블리에 유효한 강력한 이름 서명이 없으므로 해당 서명의 확인을 해제해야 합니다. 강력한 이름 도구와 함께 **-Vr** 옵션을 사용하여 이 작업을 수행할 수 있습니다.

   다음 예제에서는 *myAssembly.dll*라는 어셈블리에 대 한 확인을 해제 합니다.

콘솔

```
sn -Vr myAssembly.dll
```

ARM(Advanced RISC Machine) 마이크로프로세서와 같이 강력한 이름 도구를 실행할 수 없는 플랫폼에서 확인을 해제하려면 **-Vk** 옵션을 사용하여 레지스트리 파일을 만듭니다. 확인을 해제하려는 컴퓨터의 레지스트리로 레지스트리 파일을 가져옵니다. 다음 예제에서는 에 대한 `myAssembly.dll` 레지스트리 파일을 만듭니다.

콘솔

```
sn -Vk myRegFile.reg myAssembly.dll
```

-Vr 또는 **-Vk** 옵션을 사용하면 선택적으로 테스트 키 서명에 *.snk* 파일을 포함할 수 있습니다.

> **⚠ 경고**
>
> 보안을 위해 강력한 이름을 사용하지 마세요. 고유한 ID만 제공합니다.

> **ⓘ 참고**
>
> 64비트 컴퓨터에서 Visual Studio를 사용하여 개발하는 동안 서명 지연을 사용하고 **모든 CPU**에 대한 어셈블리를 컴파일하는 경우 **-Vr** 옵션을 두 번 적용해야 할 수 있습니다. (Visual Studio에서 **모든 CPU** 는 **플랫폼 대상** 빌드 속성의 값입니다. 명령줄에서 컴파일하는 경우 기본값입니다.) 명령줄 또는 파일 탐색기에서 애플리케이션을 실행하려면 64비트 버전의 Sn.exe(강력한 이름 도구) 를 사용하여 어셈블리에 **-Vr** 옵션을 적용합니다. 디자인 타임에 어셈블리를 Visual Studio에 로드하려면(예: 어셈블리에 애플리케이션의 다른 어셈블리에서 사용하는 구성 요소가 포함된 경우) 강력한 이름 도구의 32비트 버전을 사용합니다. 이는 JIT(Just-In-Time) 컴파일러가 명령줄에서 어셈블리를 실행할 때 어셈블리를 64비트 네이티브 코드로 컴파일하고, 어셈블리가 디자인 타임 환경에 로드될 때 32비트 네이티브 코드로 컴파일하기 때문입니다.

5. 나중에 일반적으로 배송 직전에 강력한 이름 도구와 함께 **-R** 옵션을 사용하여 실제 강력한 이름 서명에 대한 어셈블리를 조직의 서명 기관에 제출합니다.

   다음 예제에서는 *sgKey.snk* 키 쌍을 사용하여 강력한 이름으로 *myAssembly.dll* 이라는 어셈블리에 서명합니다.

   콘솔

```
sn -R myAssembly.dll sgKey.snk
```

참고하십시오

- 어셈블리 만들기
- 방법: 퍼블릭-프라이빗 키 쌍 만들기
- Sn.exe(강력한 이름 도구)

방법: 어셈블리 내용 보기

2025. 06. 22.

Ildasm.exe(IL 디스어셈블러)를 사용하여 파일에서 CIL(공용 중간 언어) 정보를 볼 수 있습니다. 검사 중인 파일이 어셈블리인 경우 이 정보에는 어셈블리의 특성과 다른 모듈 및 어셈블리에 대한 참조가 포함될 수 있습니다. 이 정보는 파일이 어셈블리인지 어셈블리의 일부인지, 파일에 다른 모듈 또는 어셈블리에 대한 참조가 있는지 여부를 결정하는 데 유용할 수 있습니다.

*Ildasm.exe*사용하여 어셈블리의 내용을 표시하려면 명령 프롬프트에 **ildasm <어셈블리 이름을** > 입력합니다. 예를 들어 다음 명령은 *Hello.exe* 어셈블리를 디스어셈블합니다.

Windows 명령 프롬프트

```
ildasm Hello.exe
```

어셈블리 매니페스트 정보를 보려면 IL 디스어셈블러 창에서 **매니페스트** 아이콘을 두 번 클릭합니다.

예시

다음 예제는 기본 "Hello World" 프로그램으로 시작합니다. 프로그램을 컴파일한 후 *Ildasm.exe* 사용하여 *Hello.exe* 어셈블리를 디스어셈블하고 어셈블리 매니페스트를 봅니다.

C#

```
using System;

class MainApp
{
    public static void Main()
    {
        Console.WriteLine("Hello World using C#!");
    }
}
```

Hello.exe 어셈블리에서 명령 *ildasm.exe* 실행하고 IL 디스어셈블러 창에서 **매니페스트** 아이콘을 두 번 클릭하면 다음과 같은 출력이 생성됩니다.

출력

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
```

```
  .ver 4:0:0:0
}
.assembly Hello
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(i
nt32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() =
( 01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Hello.exe
// MVID: {7C2770DB-1594-438D-BAE5-98764C39CCCA}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00600000
```

다음 표에서는 예제에 사용된 *Hello.exe* 어셈블리의 어셈블리 매니페스트에 있는 각 지시문을 설명합니다.

테이블 확장

지시	설명
.assembly extern <**어셈블리 이름**>	현재 모듈에서 참조하는 항목을 포함하는 다른 어셈블리를 지정합니다(이 예제 `mscorlib`에서는).
.publickeytoken <**토큰**>	참조된 어셈블리의 실제 키 토큰을 지정합니다.
.ver <**버전 번호**>	참조된 어셈블리의 버전 번호를 지정합니다.
.assembly <**어셈블리 이름**>	어셈블리 이름을 지정합니다.
.hash **알고리즘** <int32 **값**>	사용되는 해시 알고리즘을 지정합니다.
.ver <**버전 번호**>	어셈블리의 버전 번호를 지정합니다.
.module <**파일 이름**>	어셈블리를 구성하는 모듈의 이름을 지정합니다. 이 예제에서 어셈블리는 하나의 파일로만 구성됩니다.
.subsystem <**값**>	프로그램에 필요한 애플리케이션 환경을 지정합니다. 이 예제에서 값 3은 이 실행 파일이 콘솔에서 실행되었음을 나타냅니다.

지시	설명
.코플래그	현재 메타데이터의 예약된 필드입니다.

어셈블리 매니페스트는 어셈블리의 내용에 따라 다양한 지시문을 포함할 수 있습니다. 어셈블리 매니페스트의 지시문에 대한 광범위한 목록은 Ecma 설명서, 특히 "파티션 II: 메타데이터 정의 및 의미 체계" 및 "파티션 III: CIL 명령 집합"을 참조하세요.

- ECMA C# 및 공용 언어 인프라 표준
- 표준 ECMA-335 - CLI(공용 언어 인프라)

참고하십시오

- 애플리케이션 도메인 및 어셈블리
- Ildasm.exe(IL 디스어셈블러)

공용 언어 런타임의 타입 전달

2025. 06. 17.

형식 전달을 사용하면 원래 어셈블리를 사용하는 애플리케이션을 다시 컴파일하지 않고도 형식을 다른 어셈블리로 이동할 수 있습니다.

예를 들어, 애플리케이션이 `Example` 클래스를 *Utility.dll* 어셈블리에서 사용한다고 가정합니다. *Utility.dll* 개발자는 어셈블리를 리팩터링하기로 결정할 수 있으며, 그 과정에서 클래스를 `Example` 다른 어셈블리로 이동할 수 있습니다. 이전 버전의 *Utility.dll* 새 버전의 *Utility.dll* 및 해당 도우미 어셈블리로 대체되는 경우 새 버전의 `Example`클래스를 찾을 `Example` 수 없으므로 클래스를 사용하는 애플리케이션이 실패합니다.

Utility.dll 개발자는 `Example` 특성을 사용하고 TypeForwardedToAttribute 클래스에 대한 요청을 전달하여 이를 방지할 수 있습니다. 특성이 새 버전의 *Utility.dll*적용된 경우 클래스에 대한 `Example` 요청은 이제 클래스가 포함된 어셈블리로 전달됩니다. 기존 애플리케이션은 다시 컴파일하지 않고 정상적으로 계속 작동합니다.

형식 전달

형식을 전달하는 네 가지 단계가 있습니다.

1. 원래 어셈블리에서 대상 어셈블리로 형식의 소스 코드를 이동합니다.
2. 이전 형식이 있던 어셈블리에서 이동된 형식에 대해 TypeForwardedToAttribute를 추가합니다. 다음 코드는 이동된 형식 `Example` 의 특성을 보여줍니다.

   ```
   C#
   [assembly:TypeForwardedToAttribute(typeof(Example))]
   ```

3. 이제 형식이 포함된 어셈블리를 컴파일합니다.
4. 현재 형식이 포함된 어셈블리에 대한 참조를 사용하여 형식이 있던 어셈블리를 다시 컴파일합니다. 예를 들어 명령줄에서 C# 파일을 컴파일하는 경우 **참조** (C# 컴파일러 옵션) 옵션을 사용하여 형식이 포함된 어셈블리를 지정합니다. C++에서 소스 파일의 #using 지시문을 사용하여 형식이 포함된 어셈블리를 지정합니다.

C# 형식 전달 예제

위의 인위적인 예제 설명을 계속 진행하면서, 당신이 *Utility.dll*을 개발 중이고 클래스가 있다고 `Example` 상상해 보세요. *Utility.csproj*는 기본 클래스 라이브러리입니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsing>true</ImplicitUsing>
  </PropertyGroup>

</Project>
```

`Example` 클래스는 몇 가지 속성과 메서드 재정의를 제공합니다:Object.ToString

C#

```csharp
using System;

namespace Common.Objects;

public class Example
{
    public string Message { get; init; } = "Hi friends!";

    public Guid Id { get; init; } = Guid.NewGuid();

    public DateOnly Date { get; init; } = DateOnly.FromDateTime(DateTime.Today);

    public sealed override string ToString() =>
        $"[{Id} - {Date}]: {Message}";
}
```

이제 소비 프로젝트가 있고 *소비자* 어셈블리에 표현되었다고 상상해 보십시오. 이 사용 프로젝트는 *유틸리티* 어셈블리를 참조합니다. 예를 들어 개체를 `Example` 인스턴스화하고 *Program.cs* 파일의 콘솔에 씁니다.

C#

```csharp
using System;
using Common.Objects;

Example example = new();

Console.WriteLine(example);
```

사용 중인 앱이 실행되면 개체의 `Example` 상태가 출력됩니다. 이 시점에서 *Consuming.csproj* 가 *Utility.csproj*를 참조하므로 형식 전달이 없습니다. 그러나 *유틸리티* 어셈블리의 개발자들은 리팩

터링의 일환으로 `Example` 개체를 제거하기로 결정했습니다. 이 형식은 새로 만든 *Common.csproj*로 이동됩니다.

유틸리티 어셈블리에서 이 형식을 제거하면 개발자는 호환성이 손상되는 변경을 도입하고 있습니다. 사용 중인 모든 프로젝트는 최신 *유틸리티* 어셈블리로 업데이트할 때 중단됩니다.

사용 중인 프로젝트에서 *Common* 어셈블리에 새 참조를 추가하도록 요구하는 대신 형식을 전달할 수 있습니다. 유틸리티 *어셈블리*에서 이 형식이 제거되었으므로, *Utility.csproj*에서 *Common.csproj*를 참조해야 합니다.

XML

```xml
<ItemGroup>
  <ProjectReference Include="..\Common\Common.csproj" />
</ItemGroup>
```

이전 *C#* 프로젝트는 이제 새로 만든 *Common* 어셈블리를 참조합니다. 이는 `PackageReference`이거나 `ProjectReference`일 수 있습니다. *유틸리티* 어셈블리는 형식 전달 정보를 제공해야 합니다. 규칙 유형에 따라 정방향 선언은 일반적으로 이름이 지정된 `TypeForwarders`단일 파일에 캡슐화됩니다. *유틸리티* 어셈블리에서 다음 *TypeForwarders.cs C#* 파일을 고려합니다.

C#

```csharp
using System.Runtime.CompilerServices;
using Common.Objects;

[assembly:TypeForwardedTo(typeof(Example))]
```

유틸리티 어셈블리는 *Common* 어셈블리를 참조하고 형식을 `Example` 전달합니다. 형식 전달 선언을 사용하여 *유틸리티* 어셈블리를 컴파일하고 *Utility.dll*을 *소비* 폴더에 넣으면, 소비하는 애플리케이션을 별도로 컴파일하지 않아도 작동합니다.

참고하십시오

- TypeForwardedToAttribute
- 형식 전달(C++/CLI)
- #using 지시문

friend 어셈블리

2025. 06. 22.

*friend 어셈블리*는 다른 어셈블리의 내부(C#) 또는 Friend(Visual Basic) 형식 및 멤버에 액세스할 수 있는 어셈블리입니다. *AssemblyB*를 friend 어셈블리로 식별하기 위해 *AssemblyA에 어셈블리* 특성을 추가하는 경우 AssemblyB에서 액세스하기 위해 *더 이상 AssemblyA*의 형식 및 멤버를 공용으로 표시할 필요가 *없습니다*. 이는 다음 시나리오에서 특히 편리합니다.

- 단위 테스트 중에 테스트 코드가 별도의 어셈블리에서 실행될 때, C#에서 `internal`로 표시되거나 Visual Basic에서 `Friend`로 표시된 테스트 중인 어셈블리의 멤버에 대한 액세스가 필요한 경우
- 클래스 라이브러리를 개발하는 경우 라이브러리에 대한 추가는 별도의 어셈블리에 포함되지만 C# 또는 `internal` Visual Basic으로 표시된 `Friend` 기존 어셈블리의 멤버에 대한 액세스가 필요합니다.

비고

InternalsVisibleToAttribute 특성을 사용하여 특정 어셈블리에 대해 하나 이상의 friend 어셈블리를 식별할 수 있습니다. 다음 예제에서는 InternalsVisibleToAttribute의 특성을 사용하고 *AssemblyB* 어셈블리를 프렌드 어셈블리로 지정합니다. 이렇게 하면 *어셈블리B*에 C#에서 또는 Visual Basic에서 `internal`으로 표시된 `Friend`의 모든 타입 및 멤버에 대한 액세스 권한을 부여합니다.

> ⓘ **참고**
>
> *AssemblyA*와 같은 다른 어셈블리의 내부 형식 또는 내부 멤버에 액세스하는 *AssemblyB*와 같은 어셈블리를 컴파일하는 경우 *-out* 컴파일러 옵션을 사용하여 출력 파일(*.exe* 또는 **.dll**)의 이름을 명시적으로 지정해야 합니다. 컴파일러가 외부 참조에 바인딩할 때 빌드 중인 어셈블리의 이름을 아직 생성하지 않았기 때문에 이 작업이 필요합니다. 자세한 내용은 OutputAssembly (C#) 또는 -out(Visual Basic)을 참조하세요.

C#

```
using System.Runtime.CompilerServices;
using System;

[assembly: InternalsVisibleTo("AssemblyB")]

// The class is internal by default.
class FriendClass
{
```

```
    public void Test()
    {
        Console.WriteLine("Sample Class");
    }
}

// Public class that has an internal method.
public class ClassWithFriendMethod
{
    internal void Test()
    {
        Console.WriteLine("Sample Method");
    }

}
```

명시적으로 친구로 지정한 어셈블리만 `internal` (C#) 또는 `Friend` (Visual Basic) 형식 및 멤버에 액세스할 수 있습니다. 예를 들어 *AssemblyB*가 *어셈블리 A*의 친구이고 *어셈블리 C*가 *AssemblyB*를 참조하는 경우 *어셈블리 C는 어셈블리* `internal` 에서 (C#) 또는 `Friend` (Visual Basic) 형식에 액세스할 수 없습니다.

컴파일러는 특성에 전달된 InternalsVisibleToAttribute friend 어셈블리 이름의 몇 가지 기본 유효성 검사를 수행합니다. *어셈블리 A*가 *AssemblyB*를 friend 어셈블리로 선언하는 경우 유효성 검사 규칙은 다음과 같습니다.

- *어셈블리 A*가 강력한 이름이면 *AssemblyB*도 강력한 이름을 지정해야 합니다. 특성에 전달되는 friend 어셈블리 이름은 어셈블리 이름과 *AssemblyB*에 서명하는 데 사용되는 강력한 이름 키의 공개 키로 구성되어야 합니다.

  특성 InternalsVisibleToAttribute에 전달되는 친구 어셈블리 이름은 *AssemblyB*의 강력한 이름이 될 수 없습니다. 어셈블리 버전, 문화권, 아키텍처 또는 공개 키 토큰을 포함하지 마세요.

- *어셈블리 A*가 강력한 이름이 아닌 경우, friend 어셈블리 이름은 어셈블리 이름 하나로만 구성되어야 합니다. 자세한 내용은 방법: 서명되지 않은 friend 어셈블리 만들기를 참조하세요.

- *AssemblyB* 이름이 강력한 경우 프로젝트 설정 또는 명령줄 컴파일러 옵션을 사용하여 `/keyfile` 에 대한 강력한 이름 키를 지정해야 합니다. 자세한 내용은 방법: 서명된 friend 어셈블리 만들기를 참조하세요.

클래스는 StrongNameIdentityPermission 다음과 같은 차이점과 함께 형식을 공유하는 기능도 제공합니다.

- StrongNameIdentityPermission 는 개별 형식에 적용되며 friend 어셈블리는 전체 어셈블리에 적용됩니다.

- AssemblyB와 공유하려는 *어셈블리 A*에 수백 개의 형식이 있는 경우 모든 형식에 추가 StrongNameIdentityPermission 해야 합니다. friend 어셈블리를 사용하는 경우 친구 관계를 한 번만 선언하면 됩니다.

- 사용하는 StrongNameIdentityPermission경우 공유하려는 형식을 public으로 선언해야 합니다. friend 어셈블리를 사용하는 경우 공유 형식이 (C#) 또는 `internal` (Visual Basic)으로 `Friend` 선언됩니다.

모듈 파일(`internal` 확장자를 사용하는 파일)에서 어셈블리 `Friend` (C#) 또는 (Visual Basic) 형식 및 메서드에 액세스하는 방법에 대한 자세한 내용은 **ModuleAssemblyName**(C#) 또는 -moduleassemblyname(Visual Basic)을 참조하세요.

참고하십시오

- InternalsVisibleToAttribute
- StrongNameIdentityPermission
- 방법: 서명되지 않은 friend 어셈블리 만들기
- 방법: 서명된 friend 어셈블리 만들기
- .NET의 어셈블리
- C# 프로그래밍 가이드
- 프로그래밍 개념(Visual Basic)

방법: 서명되지 않은 friend 어셈블리 만들기

아티클 • 2024. 03. 12.

이 예제에서는 서명되지 않은 어셈블리와 함께 friend 어셈블리를 사용하는 방법을 보여 줍니다.

어셈블리 및 friend 어셈블리 만들기

1. 명령 프롬프트가 엽니다.
2. 다음 코드를 포함하는 *friend_unsigned_A*라는 C# 또는 Visual Basic 파일을 만듭니다. 이 코드는 InternalsVisibleToAttribute 특성을 사용하여 *friend_unsigned_B*를 friend 어셈블리로 선언합니다.

   C#

   ```
   // friend_unsigned_A.cs
   // Compile with:
   // csc /target:library friend_unsigned_A.cs
   using System.Runtime.CompilerServices;
   using System;

   [assembly: InternalsVisibleTo("friend_unsigned_B")]

   // Type is internal by default.
   class Class1
   {
       public void Test()
       {
           Console.WriteLine("Class1.Test");
       }
   }

   // Public type with internal member.
   public class Class2
   {
       internal void Test()
       {
           Console.WriteLine("Class2.Test");
       }
   }
   ```

3. 다음 명령을 사용하여 *friend_unsigned_A*를 컴파일하고 서명합니다.

   C#

```
csc /target:library friend_unsigned_A.cs
```

4. 다음 코드를 포함하는 *friend_unsigned_B*라는 C# 또는 Visual Basic 파일을 만듭니다. *friend_unsigned_A*는 *friend_unsigned_B*를 friend 어셈블리로 지정하기 때문에 *friend_unsigned_B*는 *friend_unsigned_A*의 `internal`(C#) 또는 `Friend`(Visual Basic) 형식과 멤버에 액세스할 수 있습니다.

C#

```
// friend_unsigned_B.cs
// Compile with:
// csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe
friend_unsigned_B.cs
public class Program
{
    static void Main()
    {
        // Access an internal type.
        Class1 inst1 = new Class1();
        inst1.Test();

        Class2 inst2 = new Class2();
        // Access an internal member of a public type.
        inst2.Test();

        System.Console.ReadLine();
    }
}
```

5. 다음 명령을 사용하여 *friend_unsigned_B*를 컴파일합니다.

C#

```
csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe
friend_unsigned_B.cs
```

컴파일러에서 생성된 어셈블리 이름은 InternalsVisibleToAttribute 특성에 전달된 friend 어셈블리 이름과 일치해야 합니다. `-out` 컴파일러 옵션을 사용하여 출력 어셈블리(*.exe* 또는 *.dll*)의 이름을 명시적으로 지정해야 합니다. 자세한 내용은 **OutputAssembly** (C# 컴파일러 옵션) 또는 -out(Visual Basic)을 참조하세요.

6. *friend_unsigned_B.exe* 파일을 실행합니다.

이 프로그램은 Class1.Test 및 **Class2.Test라는 두 개의 문자열을 출력합니다**.

.NET 보안

InternalsVisibleToAttribute 특성과 StrongNameIdentityPermission 클래스 간에는 유사점이 있습니다. 기본 차이점은 특정 코드 섹션을 실행하기 위해 보안 권한을 요구할 수 있는 StrongNameIdentityPermission 반면 InternalsVisibleToAttribute 특성은 또는 `Friend` (Visual Basic) 형식 및 멤버의 `internal` 표시 여부를 제어한다는 것입니다.

참고 항목

- InternalsVisibleToAttribute
- .NET 어셈블리
- friend 어셈블리
- 방법: 서명된 friend 어셈블리 만들기
- C# 프로그래밍 가이드
- 프로그래밍 개념(Visual Basic)

방법: 서명된 friend 어셈블리 만들기

아티클 • 2023. 04. 08.

이 예제에서는 강력한 이름을 가진 어셈블리와 함께 friend 어셈블리를 사용하는 방법을 보여 줍니다. 두 어셈블리에 모두 강력한 이름을 지정해야 합니다. 이 예제의 두 어셈블리는 모두 동일한 키를 사용하지만 두 어셈블리에 서로 다른 키를 사용할 수 있습니다.

서명된 어셈블리 및 friend 어셈블리 만들기

1. 명령 프롬프트를 엽니다.
2. 강력한 이름 도구와 함께 다음 명령 시퀀스를 사용하여 키 파일을 생성하고 해당 공개 키를 표시합니다. 자세한 내용은 Sn.exe(강력한 이름 도구)를 참조하세요.
   a. 이 예제를 위한 강력한 이름 키를 생성하고 *FriendAssemblies.snk* 파일에 저장합니다.

   ```
   sn -k FriendAssemblies.snk
   ```

   b. *FriendAssemblies.snk*에서 공개 키를 추출하고 *FriendAssemblies.publickey*에 넣습니다.

   ```
   sn -p FriendAssemblies.snk FriendAssemblies.publickey
   ```

   c. *FriendAssemblies.publickey* 파일에 저장된 공개 키를 표시합니다.

   ```
   sn -tp FriendAssemblies.publickey
   ```

3. 다음 코드를 포함하는 *friend_signed_A*라는 C# 또는 Visual Basic 파일을 만듭니다. 이 코드는 InternalsVisibleToAttribute 특성을 사용하여 *friend_signed_B*를 friend 어셈블리로 선언합니다.

   강력한 이름 도구는 실행할 때마다 새 공개 키를 생성합니다. 따라서 다음 예제와 같이 다음 코드의 공개 키를 방금 생성한 공개 키로 대체해야 합니다.

   C#

   ```
   // friend_signed_A.cs
   // Compile with:
   // csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
   using System.Runtime.CompilerServices;

   [assembly: InternalsVisibleTo("friend_signed_B,
   PublicKey=0024000004800000940000000602000000240000525341310004000001000
   100e3aedce99b7e10823920206f8e46cd5558b4ec7345bd1a5b201ffe71660625dcb8f9
   ```

```
a08687d881c8f65a0dcf042f81475d2e88f3e3e273c8311ee40f952db306c02fbfc5d8b
c6ee1e924e6ec8fe8c01932e0648a0d3e5695134af3bb7fab370d3012d083fa6b83179d
d3d031053f72fc1f7da8459140b0af5afc4d2804deccb6")]
class Class1
{
    public void Test()
    {
        System.Console.WriteLine("Class1.Test");
        System.Console.ReadLine();
    }
}
```

4. 다음 명령을 사용하여 *friend_signed_A*를 컴파일하고 서명합니다.

```
C#

csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
```

5. 다음 코드를 포함하는 *friend_signed_B*라는 C# 또는 Visual Basic 파일을 만듭니다. *friend_signed_A*는 *friend_signed_B*를 friend 어셈블리로 지정하기 때문에 *friend_signed_B*의 코드는 *friend_signed_A*의 `internal`(C#) 또는 `Friend`(Visual Basic) 형식과 멤버에 액세스할 수 있습니다. 파일에는 다음 코드가 포함되어 있습니다.

```
C#

// friend_signed_B.cs
// Compile with:
// csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll
/out:friend_signed_B.exe friend_signed_B.cs
public class Program
{
    static void Main()
    {
        Class1 inst = new Class1();
        inst.Test();
    }
}
```

6. 다음 명령을 사용하여 *friend_signed_B*를 컴파일하고 서명합니다.

```
C#

csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll
/out:friend_signed_B.exe friend_signed_B.cs
```

컴파일러에서 생성된 어셈블리 이름은 InternalsVisibleToAttribute 특성에 전달된 friend 어셈블리 이름과 일치해야 합니다. `-out` 컴파일러 옵션을 사용하여 출력 어

셈블리(*.exe* 또는 *.dll*)의 이름을 명시적으로 지정해야 합니다. 자세한 내용은 **OutputAssembly** (C# 컴파일러 옵션) 또는 -out(Visual Basic)을 참조하세요.

7. *friend_signed_B.exe* 파일을 실행합니다.

   프로그램이 **Class1.Test** 문자열을 출력합니다.

.NET 보안

InternalsVisibleToAttribute 특성과 StrongNameIdentityPermission 클래스 간에는 유사점이 있습니다. 주요 차이점은 StrongNameIdentityPermission은 코드의 특정 섹션을 실행하는 보안 권한을 요구할 수 있는 반면, InternalsVisibleToAttribute 특성은 `internal`(C#) 또는 `Friend`(Visual Basic) 형식 및 멤버의 표시 유형을 제어한다는 것입니다.

참조

- InternalsVisibleToAttribute
- .NET 어셈블리
- friend 어셈블리
- 방법: 서명되지 않은 friend 어셈블리 만들기
- **KeyFile** (C#)
- -keyfile(Visual Basic)
- Sn.exe(강력한 이름 도구)
- 강력한 이름의 어셈블리 만들기 및 사용
- C# 프로그래밍 가이드
- 프로그래밍 개념(Visual Basic)

방법: 파일이 어셈블리인지 확인

파일은 관리되는 경우에만 어셈블리이며 해당 메타데이터에 어셈블리 항목을 포함합니다. 어셈블리 및 메타데이터에 대한 자세한 내용은 어셈블리 매니페스트를 참조하세요.

파일이 어셈블리인지 수동으로 확인하는 방법

1. Ildasm.exe(IL 디스어셈블러) 도구를 시작합니다.
2. 테스트할 파일을 로드합니다.
3. **ILDASM**에서 파일이 PE(이식 가능한 실행 파일) 파일이 아니라고 보고하는 경우 어셈블리가 아닙니다. 자세한 내용은 방법 : 어셈블리 콘텐츠 보기 항목을 참조하세요.

파일이 어셈블리인지 프로그래밍 방식으로 확인하는 방법

AssemblyName 클래스 사용

1. 메서드를 AssemblyName.GetAssemblyName 호출하여 테스트할 파일의 전체 파일 경로와 이름을 전달합니다.
2. 예외가 발생하면 파일이 어셈블리가 아닙니다.

이 예제에서는 DLL을 테스트하여 어셈블리인지 확인합니다.

C#

```
using System;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;

static class ExampleAssemblyName
{
    public static void CheckAssembly()
    {
        try
        {
            string path = Path.Combine(
                RuntimeEnvironment.GetRuntimeDirectory(),
                "System.Net.dll");

            AssemblyName testAssembly = AssemblyName.GetAssemblyName(path);
            Console.WriteLine("Yes, the file is an assembly.");
```

```
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("The file cannot be found.");
        }
        catch (BadImageFormatException)
        {
            Console.WriteLine("The file is not an assembly.");
        }
        catch (FileLoadException)
        {
            Console.WriteLine("The assembly has already been loaded.");
        }
    }

    /* Output:
    Yes, the file is an assembly.
    */
}
```

메서드는 GetAssemblyName 테스트 파일을 로드한 다음 정보를 읽은 후 해제합니다.

PEReader 클래스 사용

> **⊗ 주의**
>
> PEReader 및 System.Reflection.Metadata 라이브러리는 신뢰할 수 없는 입력을 처리하도록 설계되지 않았습니다. 형식이 잘못되었거나 악의적인 PE 파일은 범위를 벗어난 메모리 액세스, 충돌 또는 중단을 포함하여 예기치 않은 동작을 일으킬 수 있습니다. 신뢰할 수 있는 어셈블리에서만 이러한 API를 사용합니다.

1. .NET Standard 또는 .NET Framework를 대상으로 하는 경우 System.Reflection.Metadata NuGet 패키지를 설치합니다. (.NET Core 또는 .NET 5+를 대상으로 하는 경우 이 라이브러리가 공유 프레임워크에 포함되어 있으므로 이 단계가 필요하지 않습니다.)

2. System.IO.FileStream 테스트 중인 파일에서 데이터를 읽을 인스턴스를 만듭니다.

3. 인스턴스를 System.Reflection.PortableExecutable.PEReader 만들어 파일 스트림을 생성자에 전달합니다.

4. 속성 값을 확인합니다 HasMetadata . 값이 `false` 면 파일이 어셈블리가 아닙니다.

5. PE 판독기 GetMetadataReader 인스턴스에서 메서드를 호출하여 메타데이터 판독기를 만듭니다.

6. 속성 값을 확인합니다 IsAssembly . 값이 `true` 면 파일이 어셈블리입니다.

있기 GetAssemblyName 메서드와 달리, PEReader 클래스는 네이티브 이식 가능한 실행 파일(PE)에 대해 예외를 throw하지 않습니다. 이렇게 하면 이러한 파일을 확인해야 할 때 예외로 인한 추가 성능 비용을 방지할 수 있습니다. 파일이 없거나 PE 파일이 아닌 경우에도 예외를 처리해야 합니다.

이 예제에서는 클래스를 사용하여 PEReader 파일이 어셈블리인지 확인하는 방법을 보여줍니다.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection.Metadata;
using System.Reflection.PortableExecutable;
using System.Runtime.InteropServices;

static class ExamplePeReader
{
    static bool IsAssembly(string path)
    {
        using var fs = new FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);

        // Try to read CLI metadata from the PE file.
        using var peReader = new PEReader(fs);

        if (!peReader.HasMetadata)
        {
            return false; // File does not have CLI metadata.
        }

        // Check that file has an assembly manifest.
        MetadataReader reader = peReader.GetMetadataReader();
        return reader.IsAssembly;
    }

    public static void CheckAssembly()
    {
        string path = Path.Combine(
                RuntimeEnvironment.GetRuntimeDirectory(),
                "System.Net.dll");

        try
        {
            if (IsAssembly(path))
            {
                Console.WriteLine("Yes, the file is an assembly.");
            }
            else
            {
                Console.WriteLine("The file is not an assembly.");
            }
        }
```

```
        catch (BadImageFormatException)
        {
            Console.WriteLine("The file is not an executable.");
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("The file cannot be found.");
        }
    }

    /* Output:
    Yes, the file is an assembly.
    */
}
```

참고하십시오

- AssemblyName
- C# 프로그래밍 가이드
- 프로그래밍 개념(Visual Basic)
- .NET의 어셈블리

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 19.

방법: 어셈블리 로드 및 언로드

아티클 • 2024. 03. 12.

프로그램에서 참조하는 어셈블리는 공용 언어 런타임에 의해 자동으로 로드되지만 현재 애플리케이션 도메인에 특정 어셈블리를 동적으로 로드할 수도 있습니다. 자세한 내용은 방법: 애플리케이션에 어셈블리 로드기본 참조하세요.

.NET Framework에서는 해당 어셈블리가 포함된 애플리케이션 도메인을 모두 언로드하지 않으면 개별 어셈블리를 언로드할 수 없습니다. 어셈블리가 범위를 벗어난 경우에도 실제 어셈블리 파일은 해당 파일을 포함하는 애플리케이션 도메인이 모두 언로드될 때까지 로드된 상태로 유지됩니다. .NET Core에서는 System.Runtime.Loader.AssemblyLoadContext 클래스가 어셈블리 언로드를 처리합니다. 자세한 내용은 .NET Core에서 어셈블리 언로드 기능을 사용하고 디버그하는 방법을 참조하세요.

어셈블리 로드 및 언로드

어셈블리를 애플리케이션 도메인에 로드하려면 AppDomain 및 Assembly 클래스에 포함된 여러 로드 메서드 중 하나를 사용하세요. 자세한 내용은 방법: 애플리케이션에 어셈블리 로드기본 참조하세요. .NET Core는 단일 애플리케이션 도메인만 지원합니다.

.NET Framework에서 어셈블리를 언로드하려면 해당 어셈블리를 포함하는 모든 애플리케이션 도메인을 언로드해야 합니다. 애플리케이션 도메인을 언로드하려면 AppDomain.Unload 메서드를 사용하세요. 자세한 내용은 방법: 애플리케이션 언로드를 참조하세요기본.

.NET Framework 애플리케이션에서 일부 어셈블리만 언로드하고 다른 어셈블리는 언로드하지 않으려는 경우 새 애플리케이션 도메인을 만들어 이 도메인 내에서 코드를 실행한 다음 해당 애플리케이션 도메인을 언로드하는 것이 좋습니다. 자세한 내용은 방법: 애플리케이션 언로드를 참조하세요기본.

참고 항목

- C# 프로그래밍 가이드
- 프로그래밍 개념(Visual Basic)
- .NET 어셈블리
- 방법: 어셈블리를 애플리케이션에 로드합니다기본

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 참여자 가이드를 참조하세요.

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

설명서 문제 열기

제품 사용자 의견 제공

연습: Visual Studio에 관리되는 어셈블리의 형식 포함

아티클 • 2023. 04. 08.

강력한 이름의 관리되는 어셈블리에서 형식 정보를 포함하는 경우 애플리케이션에서 유형을 느슨하게 연결하여 버전 독립성을 확보할 수 있습니다. 즉, 새로운 버전마다 다시 컴파일하지 않고도 관리되는 라이브러리의 모든 버전에서 형식을 사용하도록 프로그램을 작성할 수 있습니다.

형식 포함은 Microsoft Office의 자동화 개체를 사용하는 애플리케이션과 같은 COM interop에서 자주 사용됩니다. 형식 정보를 포함하면 서로 다른 컴퓨터의 서로 다른 Microsoft Office 버전에서 동일한 빌드의 프로그램을 작동할 수 있습니다. 그러나 완전 관리형 솔루션에서도 형식 포함을 사용할 수 있습니다.

포함할 수 있는 공용 인터페이스를 지정한 후 이러한 인터페이스를 구현하는 런타임 클래스를 만듭니다. 클라이언트 프로그램은 공용 인터페이스가 포함된 어셈블리를 참조하고 참조의 `Embed Interop Types` 속성을 `True`로 설정하여 디자인 타임에 해당 인터페이스의 형식 정보를 포함할 수 있습니다. 그러면 클라이언트 프로그램은 해당 인터페이스로 형식이 지정된 런타임 개체의 인스턴스를 로드할 수 있습니다. 이는 명령줄 컴파일러를 사용하고 **EmbedInteropTypes** 컴파일러 옵션을 사용하여 어셈블리를 참조하는 것과 같습니다.

강력한 이름의 런타임 어셈블리의 새 버전을 만들면 클라이언트 프로그램을 다시 컴파일할 필요가 없습니다. 클라이언트 프로그램은 공용 인터페이스의 포함된 형식 정보를 통해 사용 가능한 런타임 어셈블리 버전을 계속 사용합니다.

이 연습에서는 다음과 같은 작업을 수행합니다.

1. 포함할 수 있는 형식 정보가 있는 공개 인터페이스와 함께 강력한 이름의 어셈블리를 만듭니다.
2. 공용 인터페이스를 구현하는 강력한 이름의 런타임 어셈블리를 만듭니다.
3. 공용 인터페이스에서 형식 정보를 포함하고 런타임 어셈블리에서 클래스의 인스턴스를 만드는 클라이언트 프로그램을 만듭니다.
4. 런타임 어셈블리를 수정하고 다시 빌드합니다.
5. 클라이언트 프로그램을 실행하면 다시 컴파일하지 않고도 클라이언트 프로그램이 런타임 어셈블리의 새 버전을 사용하는지 확인할 수 있습니다.

> **참고**
>
> 일부 Visual Studio 사용자 인터페이스 요소의 경우 다음 지침에 설명된 것과 다른 이름 또는 위치가 시스템에 표시될 수 있습니다. 이러한 요소는 사용하는 Visual Studio

버전 및 설정에 따라 결정됩니다. 자세한 내용은 **IDE 개인 설정**을 참조하세요.

조건 및 제한 사항

다음 조건에서는 어셈블리의 형식 정보를 포함할 수 있습니다.

- 어셈블리는 하나 이상의 공용 인터페이스를 제공합니다.
- 포함된 인터페이스는 고유한 GUID가 있는 `ComImport` 특성 및 `Guid` 특성으로 주석이 추가됩니다.
- 어셈블리는 `ImportedFromTypeLib` 특성이나 `PrimaryInteropAssembly` 특성, 그리고 어셈블리 수준의 `Guid` 특성으로 주석이 추가됩니다. Visual C# 및 Visual Basic 프로젝트 템플릿에는 기본적으로 어셈블리 수준의 `Guid` 특성이 포함됩니다.

형식 포함의 기본 기능은 COM interop 어셈블리를 지원하는 것이므로 완전 관리형 솔루션에 형식 정보를 포함할 때 다음과 같은 제한 사항이 적용됩니다.

- COM interop 관련 특성만 포함됩니다. 다른 특성은 무시됩니다.
- 형식이 제네릭 매개 변수를 사용하고 제네릭 매개 변수의 형식이 포함된 형식인 경우 해당 형식을 어셈블리 경계를 넘어 사용할 수 없습니다. 어셈블리 경계를 넘어가는 예로는 다른 어셈블리에서 메서드를 호출하는 경우 또는 다른 어셈블리에 정의된 형식에서 형식이 파생되는 경우를 들 수 있습니다.
- 상수는 포함되지 않습니다.
- System.Collections.Generic.Dictionary<TKey,TValue> 클래스는 포함된 형식을 키로 지원하지 않습니다. 포함된 형식을 키로서 지원하도록 고유한 사전 형식을 구현할 수 있습니다.

인터페이스 만들기

첫 번째 단계는 형식 동등 인터페이스 프로젝트를 만드는 것입니다.

1. Visual Studio에서 **파일**>**새로 만들기**>**프로젝트**를 선택합니다.

2. **새 프로젝트 만들기** 대화 상자에서 **템플릿 검색** 상자에 *class library*를 입력합니다. 목록에서 C# 또는 Visual Basic **클래스 라이브러리(.NET Framework)** 템플릿을 선택한 후 **다음**을 선택합니다.

3. **새 프로젝트 구성** 대화 상자에서 **프로젝트 이름**에 *TypeEquivalenceInterface*를 입력한 다음 **만들기**를 선택합니다. 새 프로젝트가 만들어집니다.

4. **솔루션 탐색기**에서 *Class1.cs* 또는 *Class1.vb* 파일을 마우스 오른쪽 단추로 클릭하고 **이름 바꾸기**를 선택하여 파일 이름을 *Class1*에서 *ISampleInterface*로 바꿉니다. 클래

스의 이름도 `ISampleInterface`로 바꿀지 묻는 메시지가 표시되면 **예**라고 답합니다. 이 클래스는 클래스의 공용 인터페이스를 나타냅니다.

5. **솔루션 탐색기**에서 TypeEquivalenceInterface 프로젝트를 마우스 오른쪽 단추로 클릭한 다음 **속성**을 선택합니다.

6. **속성** 화면의 왼쪽 창에서 **빌드**를 선택하고 **출력 경로**를 *C:\TypeEquivalenceSample* 같은 컴퓨터의 위치로 설정합니다. 이 연습 전체에서 동일한 위치를 사용합니다.

7. **속성** 화면의 왼쪽 창에서 **강력한 이름 지정 빌드**>를 선택한 다음 어**셈블리 검사 서명** 상자를 선택합니다. **강력한 이름 키 파일**에서 **찾아보기를** 선택합니다.

8. *TypeEquivalenceInterface* 프로젝트에서 만든 *key.snk* 파일로 이동하여 선택한 다음 **확인을** 선택합니다. 자세한 내용은 공개-프라이빗 키 쌍 만들기를 참조하세요.

9. 코드 편집기에서 *ISampleInterface* 클래스를 열고 콘텐츠를 다음 코드로 바꾸어 `ISampleInterface` 인터페이스를 만듭니다.

```
C#
```

```
using System;
using System.Runtime.InteropServices;

namespace TypeEquivalenceInterface
{
    [ComImport]
    [Guid("8DA56996-A151-4136-B474-32784559F6DF")]
    public interface ISampleInterface
    {
        void GetUserInput();
        string UserInput { get; }
    }
}
```

10. **도구** 메뉴에서 **GUID 만들기**를 선택하고 **GUID 만들기** 대화 상자에서 **레지스트리 형식**을 선택합니다. **복사**를 선택한 다음 **끝내기**를 선택합니다.

11. 코드의 `Guid` 특성에서 샘플 GUID를 복사한 GUID로 바꾸고 중괄호({ })를 제거합니다.

12. **솔루션 탐색기**에서 **속성** 폴더를 펼치고 *AssemblyInfo.cs* 또는 *AssemblyInfo.vb* 파일을 선택합니다. 코드 편집기에서 파일에 다음 특성을 추가합니다.

```
C#
```

```
[assembly: ImportedFromTypeLib("")]
```

13. **파일**>**모두 저장**을 선택하거나 `Ctrl`+`Shift`+`S`를 눌러 파일과 프로젝트를 저장합니다.

14. **솔루션 탐색기**에서 **TypeEquivalenceInterface** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **빌드**를 선택합니다. 클래스 라이브러리 DLL 파일이 컴파일되고 지정된 빌드 출력 경로에 저장됩니다(예: *C:\TypeEquivalenceSample*).

런타임 클래스 만들기

다음으로 동등한 형식의 런타임 프로젝트를 만듭니다.

1. Visual Studio에서 **파일**>**새로 만들기**>**프로젝트**를 선택합니다.

2. **새 프로젝트 만들기** 대화 상자에서 **템플릿 검색** 상자에 *class library*를 입력합니다. 목록에서 *C#* 또는 Visual Basic **클래스 라이브러리(.NET Framework)** 템플릿을 선택한 후 **다음**을 선택합니다.

3. **새 프로젝트 구성** 대화 상자에서 **프로젝트 이름**에 *TypeEquivalenceRuntime*을 입력한 다음 **만들기**를 선택합니다. 새 프로젝트가 만들어집니다.

4. **솔루션 탐색기**에서 *Class1.cs* 또는 *Class1.vb* 파일을 마우스 오른쪽 단추로 클릭하고 **이름 바꾸기**를 선택하여 파일 이름을 *Class1*에서 *SampleClass*로 바꿉니다. 클래스의 이름도 `SampleClass`로 바꿀지 묻는 메시지가 표시되면 **예**라고 답합니다. 이 클래스는 `ISampleInterface` 인터페이스를 구현합니다.

5. **솔루션 탐색기**에서 **TypeEquivalenceInterface** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **속성**을 선택합니다.

6. **속성** 화면의 왼쪽 창에서 **빌드**를 선택한 다음 출력 **경로를***TypeEquivalenceInterface* 프로젝트에 사용한 것과 동일한 위치(예*: C:\TypeEquivalenceSample*)로 설정합니다.

7. **속성** 화면의 왼쪽 창에서 **강력한 이름 지정 빌드**>를 선택한 다음 어**셈블리 검사 서명** 상자를 선택합니다. **강력한 이름 키 파일**에서 **찾아보기를** 선택합니다.

8. *TypeEquivalenceInterface* 프로젝트에서 만든 *key.snk* 파일로 이동하여 선택한 다음 **확인을** 선택합니다. 자세한 내용은 공개-프라이빗 키 쌍 만들기를 참조하세요.

9. **솔루션 탐색기**에서 **TypeEquivalenceRuntime** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **추가**>**참조**를 선택합니다.

10. **참조 관리자** 대화 상자에서 **찾아보기**를 선택하고 출력 경로 폴더를 찾습니다. *TypeEquivalenceInterface.dll* 파일을 선택하고 **추가**를 선택한 다음 **확인**을 선택합니다.

11. **솔루션 탐색기**에서 **참조** 폴더를 펼치고 TypeEquivalenceInterface 참조를 선택합니다. **속성** 창에서 아직 설정하지 않았다면 **특정 버전**을 False로 설정합니다.

12. 코드 편집기에서 *SampleClass* 클래스 파일을 열고 콘텐츠를 다음 코드로 바꾸어 `SampleClass` 클래스를 만듭니다.

```
C#

using System;
using TypeEquivalenceInterface;

namespace TypeEquivalenceRuntime
{
    public class SampleClass : ISampleInterface
    {
        private string p_UserInput;
        public string UserInput { get { return p_UserInput; } }

        public void GetUserInput()
        {
            Console.WriteLine("Please enter a value:");
            p_UserInput = Console.ReadLine();
        }
    }
}
```

13. **파일**>**모두 저장**을 선택하거나 `Ctrl`+`Shift`+`S`를 눌러 파일과 프로젝트를 저장합니다.

14. **솔루션 탐색기**에서 **TypeEquivalenceRuntime** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **빌드**를 선택합니다. 클래스 라이브러리 DLL 파일이 컴파일되고 지정된 빌드 출력 경로에 저장됩니다.

클라이언트 프로젝트 만들기

마지막으로 인터페이스 어셈블리를 참조하는 형식 동등 클라이언트 프로그램을 만듭니다.

1. Visual Studio에서 **파일**>**새로 만들기**>**프로젝트**를 선택합니다.

2. **새 프로젝트 만들기** 대화 상자에서 **템플릿 검색** 상자에 *console*을 입력합니다. 목록에서 C# 또는 Visual Basic **콘솔 앱(.NET Framework)** 템플릿을 선택한 후 **다음**을 선택합니다.

3. **새 프로젝트 구성** 대화 상자에서 **프로젝트 이름**에 *TypeEquivalenceClient*를 입력한 다음 **만들기**를 선택합니다. 새 프로젝트가 만들어집니다.

4. **솔루션 탐색기**에서 **TypeEquivalenceClient** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **속성**을 선택합니다.

5. **속성** 화면 왼쪽 창에서 **빌드**를 선택한 다음 **출력 경로**를 TypeEquivalenceInterface 프로젝트에 사용한 것과 동일한 위치로 설정합니다(예: *C:\TypeEquivalenceSample*).

6. **솔루션 탐색기**에서 **TypeEquivalenceClient** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **추가**>**참조**를 선택합니다.

7. **참조 관리자** 대화 상자에서 **TypeEquivalenceInterface.dll** 파일이 이미 나열되어 있다면 선택합니다. 그렇지 않다면 **찾아보기**를 선택해 출력 경로 폴더를 찾고, *TypeEquivalenceInterface.dll* 파일(*TypeEquivalenceRuntime.dll*이 아님)을 선택하고 **추가**를 선택합니다. **확인**을 선택합니다.

8. **솔루션 탐색기**에서 **참조** 폴더를 펼치고 **TypeEquivalenceInterface** 참조를 선택합니다. **속성** 창에서 **Interop 형식 포함**을 **True**로 설정합니다.

9. 코드 편집기에서 *Program.cs* 또는 *Module1.vb* 파일을 열고 콘텐츠를 다음 코드로 바꾸어 클라이언트 프로그램을 만듭니다.

```C#
using System;
using System.Reflection;
using TypeEquivalenceInterface;

namespace TypeEquivalenceClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Assembly sampleAssembly =
Assembly.Load("TypeEquivalenceRuntime");
            ISampleInterface sampleClass =

(ISampleInterface)sampleAssembly.CreateInstance("TypeEquivalenceRuntime
.SampleClass");
            sampleClass.GetUserInput();
            Console.WriteLine(sampleClass.UserInput);

Console.WriteLine(sampleAssembly.GetName().Version.ToString());
            Console.ReadLine();
        }
    }
}
```

10. **파일**>**모두 저장**을 선택하거나 `Ctrl`+`Shift`+`S`를 눌러 파일과 프로젝트를 저장합니다.

11. Ctrl+F5를 눌러 프로그램을 빌드하고 실행합니다. 콘솔 출력은 어셈블리 버전 **1.0.0.0**을 반환합니다.

인터페이스 수정

이제 인터페이스 어셈블리를 수정하고 해당 버전을 변경합니다.

1. Visual Studio에서 **파일**>**열기**>**프로젝트/솔루션**을 선택하고 **TypeEquivalenceInterface** 프로젝트를 엽니다.
2. **솔루션 탐색기**에서 **TypeEquivalenceInterface** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **속성**을 선택합니다.
3. **속성** 화면의 왼쪽 창에서 **애플리케이션**을 선택한 다음 **어셈블리 정보**를 선택합니다.
4. **어셈블리 정보** 대화 상자에서 **어셈블리 버전** 및 **파일 버전** 값을 *2.0.0.0*으로 변경한 다음 **확인**을 선택합니다.
5. *SampleInterface.cs* 또는 *SampleInterface.vb* 파일을 열고 다음 코드 줄을 `ISampleInterface` 인터페이스에 추가합니다.

   ```
   C#

   DateTime GetDate();
   ```

6. **파일**>**모두 저장**을 선택하거나 Ctrl+Shift+S를 눌러 파일과 프로젝트를 저장합니다.
7. **솔루션 탐색기**에서 **TypeEquivalenceInterface** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **빌드**를 선택합니다. 새 버전의 클래스 라이브러리 DLL 파일이 컴파일되고 빌드 출력 경로에 저장됩니다.

런타임 클래스 수정

또한 런타임 클래스를 수정하고 해당 버전을 업데이트합니다.

1. Visual Studio에서 **파일**>**열기**>**프로젝트/솔루션**을 선택하고 **TypeEquivalenceRuntime** 프로젝트를 엽니다.
2. **솔루션 탐색기**에서 **TypeEquivalenceRuntime** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **속성**을 선택합니다.

3. **속성** 화면의 왼쪽 창에서 **애플리케이션**을 선택한 다음 **어셈블리 정보**를 선택합니다.

4. **어셈블리 정보** 대화 상자에서 **어셈블리 버전** 및 **파일 버전** 값을 *2.0.0.0*으로 변경한 다음 **확인**을 선택합니다.

5. *SampleClass.cs* 또는 *SampleClass.vb* 파일을 열고 다음 코드를 `SampleClass` 클래스에 추가합니다.

   ```
   C#

   public DateTime GetDate()
   {
       return DateTime.Now;
   }
   ```

6. **파일**>**모두 저장**을 선택하거나 `Ctrl`+`Shift`+`S`를 눌러 파일과 프로젝트를 저장합니다.

7. **솔루션 탐색기**에서 **TypeEquivalenceRuntime** 프로젝트를 마우스 오른쪽 단추로 클릭하고 **빌드**를 선택합니다. 새 버전의 클래스 라이브러리 DLL 파일이 컴파일되고 빌드 출력 경로에 저장됩니다.

업데이트된 클라이언트 프로그램 실행

빌드 출력 폴더 위치로 이동하여 *TypeEquivalenceClient.exe*를 실행합니다. 이제 프로그램을 다시 컴파일하지 않고도 콘솔 출력에 `TypeEquivalenceRuntime` 어셈블리의 새 버전 *2.0.0.0*이 반영됩니다.

참조

- **EmbedInteropTypes** (C# 컴파일러 옵션)
- -link(Visual Basic)
- C# 프로그래밍 가이드
- 프로그래밍 개념(Visual Basic)
- .NET 어셈블리

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

방법: MetadataLoadContext를 사용하여 어셈블리 콘텐츠 검사

기본적으로 .NET의 리플렉션 API를 사용하면 개발자가 기본 실행 컨텍스트에 로드된 어셈블리의 콘텐츠를 검사할 수 있습니다. 그러나 어셈블리가 다른 플랫폼 또는 프로세서 아키텍처용으로 컴파일되었거나 참조 어셈블리이기 때문에 실행 컨텍스트에 어셈블리를 로드할 수 없는 경우도 있습니다. System.Reflection.MetadataLoadContext API를 사용하면 이러한 어셈블리를 로드하고 검사할 수 있습니다. 로드된 MetadataLoadContext 어셈블리는 메타데이터로만 처리됩니다. 즉, 어셈블리의 형식을 검사할 수 있지만 포함된 코드는 실행할 수 없습니다. 주 실행 컨텍스트 MetadataLoadContext 와 달리 현재 디렉터리에서 종속성을 자동으로 로드하지 않고 전달된 MetadataAssemblyResolver 디렉터리에서 제공하는 사용자 지정 바인딩 논리를 사용합니다.

필수 조건

MetadataLoadContext를 사용하려면 System.Reflection.MetadataLoadContext NuGet 패키지를 설치합니다. .NET Standard 2.0 규격 대상 프레임워크(예: .NET Core 2.0 또는 .NET Framework 4.6.1)에서 지원됩니다.

MetadataLoadContext용 MetadataAssemblyResolver 만들기

MetadataLoadContext 를 만들려면 .의 인스턴스를 제공해야 합니다MetadataAssemblyResolver. 이를 제공하는 가장 간단한 방법은 지정된 어셈블리 경로 문자열 컬렉션에서 어셈블리를 해결하는 PathAssemblyResolver를 사용하는 것입니다. 이 컬렉션은 직접 검사하려는 어셈블리 외에 필요한 모든 종속성도 포함해야 합니다. 예를 들어 외부 어셈블리에 있는 사용자 지정 특성을 읽으려면 해당 어셈블리를 포함해야 합니다. 그렇지 않으면 예외가 throw됩니다. 대부분의 경우 *최소한 코어 어셈블리*, 즉 기본 제공 시스템 형식을 포함하는 어셈블리(예: System.Object.)를 포함해야 합니다. 다음 코드에서는 검사된 어셈블리와 현재 런타임의 핵심 어셈블리로 구성된 컬렉션을 사용하여 만드는 PathAssemblyResolver 방법을 보여 있습니다.

C#

```
var resolver = new PathAssemblyResolver(new string[] { "ExampleAssembly.dll",
typeof(object).Assembly.Location });
```

모든 BCL 형식에 액세스해야 하는 경우 컬렉션에 모든 런타임 어셈블리를 포함할 수 있습니다. 다음 코드에서는 검사된 어셈블리와 현재 런타임의 모든 어셈블리로 구성된 컬렉션을 사용하여 만드는 PathAssemblyResolver 방법을 보여 있습니다.

C#

```
// Get the array of runtime assemblies.
string[] runtimeAssemblies =
Directory.GetFiles(RuntimeEnvironment.GetRuntimeDirectory(), "*.dll");

// Create the list of assembly paths consisting of runtime assemblies and the
inspected assembly.
var paths = new List<string>(runtimeAssemblies);
paths.Add("ExampleAssembly.dll");

// Create PathAssemblyResolver that can resolve assemblies using the created list.
var resolver = new PathAssemblyResolver(paths);
```

MetadataLoadContext 만들기

MetadataLoadContext을 만들려면 생성자 MetadataLoadContext(MetadataAssemblyResolver, String)을 호출한 후, 이전에 만든 MetadataAssemblyResolver를 첫 번째 매개 변수로, 코어 어셈블리 이름을 두 번째 매개 변수로 전달합니다. 핵심 어셈블리 이름을 생략할 수 있습니다. 이 경우 생성자는 기본 이름인 "mscorlib", "System.Runtime" 또는 "netstandard"를 사용하려고 합니다.

컨텍스트를 만든 후에는 다음과 같은 LoadFromAssemblyPath메서드를 사용하여 어셈블리를 로드할 수 있습니다. 로드된 어셈블리에서 코드 실행을 포함하는 리플렉션 API를 제외한 모든 리플렉션 API를 사용할 수 있습니다. 메서드는 GetCustomAttributes 생성자 실행을 포함하므로 MetadataLoadContext에서 사용자 정의 특성을 확인해야 할 때는 GetCustomAttributesData 메서드를 대신 사용하십시오.

다음 코드 샘플에서는 MetadataLoadContext을(를) 생성하고, 어셈블리를 그 안에 로드한 다음, 어셈블리의 특성을 콘솔에 출력합니다.

C#

```
var mlc = new MetadataLoadContext(resolver);

using (mlc)
{
    // Load assembly into MetadataLoadContext.
    Assembly assembly = mlc.LoadFromAssemblyPath("ExampleAssembly.dll");
    AssemblyName name = assembly.GetName();

    // Print assembly attribute information.
    Console.WriteLine($"{name.Name} has following attributes: ");

    foreach (CustomAttributeData attr in assembly.GetCustomAttributesData())
    {
        try
        {
            Console.WriteLine(attr.AttributeType);
        }
```

```
        catch (FileNotFoundException ex)
        {
            // We are missing the required dependency assembly.
            Console.WriteLine($"Error while getting attribute type: {ex.Message}");
        }
    }
}
```

같음 또는 할당 가능성을 위해 형식을 MetadataLoadContext 테스트해야 하는 경우 해당 컨텍스트에 로드된 형식 개체만 사용합니다. 런타임 형식과 형식을 혼합하는 MetadataLoadContext 것은 지원되지 않습니다. 예를 들어, MetadataLoadContext의 `testedType` 유형을 고려하십시오. 다른 형식을 할당할 수 있는지 테스트해야 하는 경우 다음과 같은 `typeof(MyType).IsAssignableFrom(testedType)` 코드를 사용하지 마세요. 대신 다음과 같은 코드를 사용합니다.

C#

```
Assembly matchAssembly = mlc.LoadFromAssemblyPath(typeof(MyType).Assembly.Location);
Type matchType = assembly.GetType(typeof(MyType).FullName!)!;

if (matchType.IsAssignableFrom(testedType))
{
    Console.WriteLine($"{nameof(matchType)} is assignable from
{nameof(testedType)}");
}
```

예시

전체 코드 예제는 MetadataLoadContext 샘플을 사용하여 어셈블리 콘텐츠 검사를 참조하세요.

참고하십시오

- .NET의 리플렉션

Last updated on 2026. 01. 22.

.NET의 리플렉션

네임스페이스의 System.Reflection 클래스와 함께 System.Type로드된 어셈블리 및 클래스,인터페이스 및 값 형식(즉, 구조 체 및 열거형)과 같이 그 안에 정의된 형식에 대한 정보를 얻을 수 있습니다. 리플렉션을 사용하여 런타임에 형식 인스턴스를 만들고 호출하고 액세스할 수도 있습니다.

어셈블리 에는 모듈이 포함되고, 모듈에는 형식이 포함되고, 형식에는 멤버가 포함됩니다. 리플렉션은 어셈블리, 모듈 및 형식을 캡슐화하는 개체를 제공합니다. 리플렉션을 사용하여 형식의 인스턴스를 동적으로 만들거나, 형식을 기존 개체에 바인딩하거나, 기존 개체에서 형식을 가져올 수 있습니다. 그런 다음 형식의 메서드를 호출하거나 해당 필드 및 속성에 액세스할 수 있습니다. 리플렉션의 일반적인 용도는 다음과 같습니다.

- 어셈블리를 정의 및 로드하고, 어셈블리 매니페스트에 나열된 모듈을 로드하고, 이 어셈블리에서 형식을 찾아 인스턴스를 만드는 데 사용합니다 Assembly .
- 모듈 및 모듈의 클래스를 포함하는 어셈블리와 같은 정보를 검색하는 데 사용합니다 Module . 모듈에 정의된 모든 전역 메서드 또는 기타 특정 비 전역 메서드를 가져올 수도 있습니다.
- 생성자의 이름, 매개 변수, 액세스 한정자(예: ConstructorInfo 또는) 및 구현 세부 정보(예: `public` 또는`private`)와 같은 정보를 검색하는 데 사용합니다 `abstract` `virtual`. GetConstructors 메서드 또는 GetConstructor 메서드를 사용하여 Type의 특정 생성자를 호출합니다.
- 메서드의 이름, 반환 형식, 매개 변수, 액세스 한정자 및 구현 세부 정보(예: 또는 MethodInfo)와 같은 `abstract` 정보를 검색하는 데 사용합니다`virtual`. 특정 메서드를 호출하려면 GetMethods, GetMethod 또는 Type의 메서드를 사용하십시오.
- 필드의 이름, 액세스 한정자 및 구현 세부 정보(예: FieldInfo)와 같은 정보를 검색하고 필드 값을 얻거나 설정하는 데 사용합니다`static`.
- 이름, 이벤트 처리기 데이터 형식, 사용자 지정 특성, 선언 형식 및 반영된 이벤트 형식과 같은 정보를 검색하고 이벤트 처리기를 추가하거나 제거하는 데 사용합니다 EventInfo .
- 이름, 데이터 형식, 선언 형식, 반영된 형식, 속성의 읽기 전용 또는 쓰기 가능 상태와 같은 정보를 검색하고 속성 값을 얻거나 설정하는 데 사용합니다 PropertyInfo .
- 매개 변수의 이름, 데이터 형식, 매개 변수가 입력 또는 출력 매개 변수인지 여부, 메서드 서명에서 매개 변수의 위치와 같은 정보를 검색하는 데 사용합니다 ParameterInfo .
- 사용자 지정 특성에 대한 정보를 검색할 때 CustomAttributeData를 사용하고 MetadataLoadContext 또는 리플렉션 전용 컨텍스트(.NET Framework)에서 작업합니다. CustomAttributeData 를 사용하면 인스턴스를 만들지 않고 특성을 검사할 수 있습니다.

네임스페이 System.Reflection.Emit 스의 클래스는 런타임에 형식을 빌드할 수 있는 특수한 형태의 리플렉션을 제공합니다.

리플렉션을 사용하여 *형식 브라우저를* 만들면 사용자가 형식을 선택한 다음 해당 형식에 대한 정보를 볼 수 있습니다.

리플렉션에는 다른 용도가 있습니다. JScript와 같은 언어용 컴파일러는 리플렉션을 사용하여 기호 테이블을 생성합니다. 네임스페이 System.Runtime.Serialization 스의 클래스는 리플렉션을 사용하여 데이터에 액세스하고 유지할 필드를 결정합니다. System.Runtime.Remoting 네임스페이스의 클래스는 serialization을 통해 간접적으로 리플렉션을 활용합니다.

리플렉션의 런타임 형식

리플렉션은 형식, 멤버, 매개 변수 및 기타 코드 엔터티를 나타내는 클래스(예 Type : 및 MethodInfo)를 제공합니다. 그러나 리플렉션을 사용하는 경우 이러한 클래스에서 직접 작동하지 않으며, 대부분은 추상(`MustInherit` Visual Basic)입니다. 대신 CLR(공용 언어 런타임)에서 제공하는 형식으로 작업합니다.

예를 들어, C# `typeof` 연산자(Visual Basic에서는 `GetType`)를 사용하여 Type 개체를 가져오는 경우, 그 개체는 실제로 `RuntimeType`입니다. `RuntimeType` 는 Type 모든 추상 메서드의 구현을 파생시키고 제공합니다.

이러한 런타임 클래스는 `internal`입니다 (`Friend`는 Visual Basic에서). 기본 클래스 설명서에서 동작을 설명하므로 기본 클래스와 별도로 문서화되지 않습니다.

Reference

- System.Type
- System.Reflection
- System.Reflection.Emit

Last updated on 2026. 01. 21.

형식 정보 보기

아티클 • 2025. 03. 25.

System.Type 클래스는 리플렉션의 중심입니다. 공용 언어 런타임은 리플렉션 요청 시 로드된 타입에 대한 `Type`를 생성합니다. `Type` 개체의 메서드, 필드, 속성 및 중첩 클래스를 사용하여 해당 형식에 대한 모든 것을 확인할 수 있습니다.

Assembly.GetType 또는 Assembly.GetTypes 사용하여 로드되지 않은 어셈블리에서 `Type` 개체를 가져와 원하는 형식 또는 형식의 이름을 전달합니다. Type.GetType 사용하여 이미 로드된 어셈블리에서 `Type` 개체를 가져옵니다. Module.GetType 및 Module.GetTypes 사용하여 모듈 `Type` 개체를 가져옵니다.

> ⓘ **참고**
>
> 제네릭 형식 및 메서드를 검사하고 조작하려면 **리플렉션 및 제네릭 형식** 에 제공된 추가 정보를 참조하고, **리플렉션을 사용하여 제네릭 형식을 검사하고 인스턴스화하는 방법**을 확인하세요.

다음 예제에서는 어셈블리에 대한 Assembly 개체 및 모듈을 가져오는 데 필요한 구문을 보여 있습니다.

C#

```
// Gets the mscorlib assembly in which the object is defined.
Assembly a = typeof(object).Module.Assembly;
```

다음 예제에서는 로드된 어셈블리에서 `Type` 개체를 가져오는 방법을 보여 줍니다.

C#

```
// Loads an assembly using its file name.
Assembly a = Assembly.LoadFrom("MyExe.exe");
// Gets the type names from the assembly.
Type[] types2 = a.GetTypes();
foreach (Type t in types2)
{
    Console.WriteLine(t.FullName);
}
```

`Type`가져오면 해당 형식의 멤버에 대한 정보를 검색할 수 있는 여러 가지 방법이 있습니다. 예를 들어 현재 형식의 각 멤버를 설명하는 MemberInfo 개체 배열을 가져오는 Type.GetMembers 메서드를 호출하여 모든 형식의 멤버를 확인할 수 있습니다.

`Type` 클래스의 메서드를 사용하여 이름으로 지정한 하나 이상의 생성자, 메서드, 이벤트, 필드 또는 속성에 대한 정보를 검색할 수도 있습니다. 예를 들어 Type.GetConstructor 현재 클래스의 특정 생성자를 캡슐화합니다.

`Type`있는 경우 Type.Module 속성을 사용하여 해당 형식이 포함된 모듈을 캡슐화하는 개체를 가져올 수 있습니다. Module.Assembly 속성을 사용하여 모듈이 포함된 어셈블리를 캡슐화하는 개체를 찾습니다. Type.Assembly 속성을 사용하여 형식을 직접 캡슐화하는 어셈블리를 가져올 수 있습니다.

System.Type 및 ConstructorInfo

다음 예제에서는 클래스의 생성자(이 경우 String 클래스)를 나열하는 방법을 보여줍니다.

C#

```
// This program lists all the public constructors
// of the System.String class.
using System;
using System.Reflection;

class ListMembers
{
    public static void Main()
    {
        Type t = typeof(System.String);
        Console.WriteLine($"Listing all the public constructors of the {t}
type");
        // Constructors.
        ConstructorInfo[] ci = t.GetConstructors(BindingFlags.Public |
BindingFlags.Instance);
        Console.WriteLine("//Constructors");
        PrintMembers(ci);
    }

    public static void PrintMembers(MemberInfo[] ms)
    {
        foreach (MemberInfo m in ms)
        {
            Console.WriteLine($"{"     "}{m}");
        }
        Console.WriteLine();
    }
}
```

MemberInfo, MethodInfo, FieldInfo 및 PropertyInfo

MemberInfo, MethodInfo, FieldInfo또는 PropertyInfo 개체를 사용하여 형식의 메서드, 속성, 이벤트 및 필드에 대한 정보를 가져옵니다.

다음 예제에서는 `MemberInfo` 사용하여 `System.IO.File` 클래스의 멤버 수를 나열하고 IsPublic 속성을 사용하여 클래스의 표시 여부를 확인합니다.

C#

```
using System;
using System.IO;
using System.Reflection;

class MyMemberInfo
{
    public static void Main()
    {
        Console.WriteLine ("\nReflection.MemberInfo");
        // Gets the Type and MemberInfo.
        Type myType = Type.GetType("System.IO.File");
        MemberInfo[] myMemberInfoArray = myType.GetMembers();
        // Gets and displays the DeclaringType method.
        Console.WriteLine($"\nThere are {myMemberInfoArray.Length} members
in {myType.FullName}.");
        Console.WriteLine($"{myType.FullName}.");
        if (myType.IsPublic)
        {
            Console.WriteLine($"{myType.FullName} is public.");
        }
    }
}
```

다음 예제에서는 지정된 멤버의 형식을 조사합니다. `MemberInfo` 클래스의 멤버에 대해 리플렉션을 수행하고 해당 형식을 나열합니다.

C#

```
// This code displays information about the GetValue method of FieldInfo.
using System;
using System.Reflection;

class MyMethodInfo
{
    public static int Main()
    {
        Console.WriteLine("Reflection.MethodInfo");
        // Gets and displays the Type.
        Type myType = Type.GetType("System.Reflection.FieldInfo");
        // Specifies the member for which you want type information.
        MethodInfo myMethodInfo = myType.GetMethod("GetValue");
        Console.WriteLine(myType.FullName + "." + myMethodInfo.Name);
        // Gets and displays the MemberType property.
```

```
        MemberTypes myMemberTypes = myMethodInfo.MemberType;
        if (MemberTypes.Constructor == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type All");
        }
        else if (MemberTypes.Custom == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type Custom");
        }
        else if (MemberTypes.Event == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type Event");
        }
        else if (MemberTypes.Field == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type Field");
        }
        else if (MemberTypes.Method == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type Method");
        }
        else if (MemberTypes.Property == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type Property");
        }
        else if (MemberTypes.TypeInfo == myMemberTypes)
        {
            Console.WriteLine("MemberType is of type TypeInfo");
        }
        return 0;
    }
}
```

다음 예제에서는 BindingFlags 함께 모든 리플렉션 `*Info` 클래스를 사용하여 지정된 클래스의 모든 멤버(생성자, 필드, 속성, 이벤트 및 메서드)를 나열하고 멤버를 정적 및 인스턴스 범주로 분할합니다.

C#

```
// This program lists all the members of the
// System.IO.BufferedStream class.
using System;
using System.IO;
using System.Reflection;

class ListMembers
{
    public static void Main()
    {
        // Specifies the class.
        Type t = typeof(System.IO.BufferedStream);
        Console.WriteLine("Listing all the members (public and non public)
```

```
of the {0} type", t);

        // Lists static fields first.
        FieldInfo[] fi = t.GetFields(BindingFlags.Static |
            BindingFlags.NonPublic | BindingFlags.Public);
        Console.WriteLine("// Static Fields");
        PrintMembers(fi);

        // Static properties.
        PropertyInfo[] pi = t.GetProperties(BindingFlags.Static |
            BindingFlags.NonPublic | BindingFlags.Public);
        Console.WriteLine("// Static Properties");
        PrintMembers(pi);

        // Static events.
        EventInfo[] ei = t.GetEvents(BindingFlags.Static |
            BindingFlags.NonPublic | BindingFlags.Public);
        Console.WriteLine("// Static Events");
        PrintMembers(ei);

        // Static methods.
        MethodInfo[] mi = t.GetMethods (BindingFlags.Static |
            BindingFlags.NonPublic | BindingFlags.Public);
        Console.WriteLine("// Static Methods");
        PrintMembers(mi);

        // Constructors.
        ConstructorInfo[] ci = t.GetConstructors(BindingFlags.Instance |
            BindingFlags.NonPublic | BindingFlags.Public);
        Console.WriteLine("// Constructors");
        PrintMembers(ci);

        // Instance fields.
        fi = t.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
            BindingFlags.Public);
        Console.WriteLine("// Instance Fields");
        PrintMembers(fi);

        // Instance properties.
        pi = t.GetProperties(BindingFlags.Instance | BindingFlags.NonPublic
|
            BindingFlags.Public);
        Console.WriteLine ("// Instance Properties");
        PrintMembers(pi);

        // Instance events.
        ei = t.GetEvents(BindingFlags.Instance | BindingFlags.NonPublic |
            BindingFlags.Public);
        Console.WriteLine("// Instance Events");
        PrintMembers(ei);

        // Instance methods.
        mi = t.GetMethods(BindingFlags.Instance | BindingFlags.NonPublic
            | BindingFlags.Public);
        Console.WriteLine("// Instance Methods");
```

```
        PrintMembers(mi);

        Console.WriteLine("\r\nPress ENTER to exit.");
        Console.Read();
    }

    public static void PrintMembers (MemberInfo [] ms)
    {
        foreach (MemberInfo m in ms)
        {
            Console.WriteLine ("{0}{1}", "     ", m);
        }
        Console.WriteLine();
    }
}
```

리플렉션 및 제네릭 형식

아티클 • 2025. 04. 30.

리플렉션의 관점에서 제네릭 형식과 일반 형식의 차이점은 제네릭 형식이 형식 매개 변수 집합(제네릭 형식 정의인 경우) 또는 형식 인수(생성된 형식인 경우)와 연결되었다는 것입니다. 제네릭 메서드는 동일한 방식으로 일반 메서드와 다릅니다.

리플렉션이 제네릭 형식 및 메서드를 처리하는 방법을 이해하는 두 가지 키가 있습니다.

- 제네릭 형식 정의 및 제네릭 메서드 정의의 형식 매개 변수는 클래스의 Type 인스턴스로 표시됩니다.

  > **참고**
  >
  > 개체가 제네릭 형식 매개 변수를 Type 나타내는 경우 많은 속성과 메서드 Type 의 동작이 다릅니다. 이러한 차이점은 속성 및 메서드 문서에 설명되어 있습니다. 예를 들어 IsAutoClass 및 DeclaringType을 참조하세요. 또한 일부 멤버는 개체가 제네릭 형식 매개 변수를 Type 나타내는 경우에만 유효합니다. 예를 들어 GetGenericTypeDefinition를 참조하세요.

- 인스턴스가 Type 제네릭 형식을 나타내는 경우 형식 매개 변수(제네릭 형식 정의의 경우) 또는 형식 인수(생성된 형식의 경우)를 나타내는 형식 배열이 포함됩니다. 제네릭 메서드를 나타내는 클래스의 인스턴스도 MethodInfo 마찬가지입니다.

리플렉션은 형식 매개 변수의 배열에 액세스하고 인스턴스가 형식 매개 변수 또는 실제 형식을 나타내는지 여부를 Type 결정할 수 있는 메서드 MethodInfoType 를 제공합니다.

여기에 설명된 메서드를 보여주는 예제 코드는 방법: 리플렉션을 사용하여 제네릭 형식 검사 및 인스턴스화를 참조하세요.

다음 설명에서는 형식 매개 변수와 인수 간의 차이, 개방형 또는 닫힌 생성 형식과 같은 제네릭 용어에 대해 잘 알고 있다고 가정합니다. 자세한 내용은 제네릭을 참조하세요.

제네릭 형식 또는 메서드인가요?

리플렉션을 사용하여 인스턴스 Type가 나타내는 알 수 없는 형식을 검사하는 경우 이 속성을 사용하여 IsGenericType 알 수 없는 형식이 제네릭인지 여부를 확인합니다. 형식이 제네릭이면 반환 `true` 됩니다. 마찬가지로, 클래스의 인스턴스로 표시되는 알 수 없는 메서드를 검사할 때 MethodInfo 속성을 사용하여 해당 메서드가 제네릭인지 여부를 확인하십시오.

제네릭 형식 또는 메서드 정의인가요?

속성을 사용하여 개체가 IsGenericTypeDefinition 제네릭 형식 정의를 나타내는지 여부를 Type 확인하고 메서드를 사용하여 IsGenericMethodDefinition 제네릭 메서드 정의를 나타내는지 여부를 MethodInfo 확인합니다.

제네릭 형식 및 메서드 정의는 인스턴스화 가능한 형식이 만들어지는 템플릿입니다. .NET 라이브러리의 제네릭 형식(예: Dictionary<TKey,TValue>제네릭 형식)은 제네릭 형식 정의입니다.

형식 또는 메서드가 열려 있거나 닫혀 있나요?

모든 바깥쪽 형식의 모든 형식 매개 변수를 포함하여 인스턴스화 가능한 형식이 모든 형식 매개 변수로 대체된 경우 제네릭 형식 또는 메서드가 닫힙니다. 제네릭 형식이 닫힌 경우에만 제네릭 형식의 인스턴스를 만들 수 있습니다. 형식이 Type.ContainsGenericParameters로 열린 경우 속성은 `true`를 반환합니다. MethodBase.ContainsGenericParameters 메서드는 동일한 기능을 수행합니다.

닫힌 제네릭 형식 생성

제네릭 형식이나 메서드 정의가 있으면 MakeGenericType 메서드를 사용하여 닫힌 제네릭 형식을 만들 수 있습니다. 닫힌 제네릭 메서드를 만들기 위해서는 MakeGenericMethod 메서드를 사용하고 이를 위해 MethodInfo가 필요합니다.

제네릭 형식 또는 메서드 정의 가져오기

제네릭 형식 또는 메서드 정의가 아닌 개방형 제네릭 형식 또는 메서드가 있는 경우 인스턴스를 만들 수 없으며 누락된 형식 매개 변수를 제공할 수 없습니다. 제네릭 형식 또는 메서드 정의가 있어야 합니다. GetGenericTypeDefinition 메서드를 사용하여 제네릭 타입 정의를 가져오거나 GetGenericMethodDefinition 메서드를 사용하여 제네릭 메서드 정의를 가져옵니다.

예를 들어, Type 객체가 `Dictionary<int, string>`를 나타내고, 타입 `Dictionary<string, MyClass>`을 만들고자 한다면, GetGenericTypeDefinition 메서드를 사용하여 Type를 나타내는 `Dictionary<TKey, TValue>`를 얻은 다음, MakeGenericType 메서드를 사용하여 Type를 나타내는 `Dictionary<int, MyClass>`를 생성할 수 있습니다.

제네릭 형식이 아닌 열린 제네릭 형식의 예는 Type 매개 변수 또는 형식 인수를 참조하세요.

형식 인수 및 형식 매개 변수 검사

메서드를 Type.GetGenericArguments 사용하여 제네릭 형식의 Type 형식 매개 변수 또는 형식 인수를 나타내는 개체 배열을 가져오고 메서드를 사용하여 MethodInfo.GetGenericArguments 제네릭 메서드에 대해 동일한 작업을 수행합니다.

개체가 형식 매개 변수를 Type 나타낸다는 것을 알게 되면 리플렉션에서 대답할 수 있는 추가 질문이 많이 있습니다. 형식 매개 변수의 원본, 위치 및 해당 제약 조건을 확인할 수 있습니다.

형식 매개 변수 또는 형식 인수

배열의 특정 요소가 형식 매개 변수인지 아니면 형식 인수인지 확인하려면 이 속성을 사용합니다 IsGenericParameter . 속성 IsGenericParameter 은 `true` 요소가 형식 매개 변수인 경우입니다.

제네릭 형식은 제네릭 형식 정의 없이 열 수 있으며, 이 경우 형식 인수와 형식 매개 변수가 혼합되어 있습니다. 예를 들어 다음 코드에서 클래스 `D` 는 두 번째 형식 매개 변수에 대한 첫 번째 형식 매개 변수 `D` 를 대체하여 만든 형식 `B`에서 파생됩니다.

C#

```
class B<T, U> {}
class D<V, W> : B<int, V> {}
```

Type 객체를 얻어 `D<V, W>`를 나타내고 BaseType 속성을 사용하여 기본 형식을 얻는 경우, 결과 `type B<int, V>`는 열려 있지만 제네릭 형식 정의는 아닙니다.

제네릭 매개 변수의 원본

제네릭 형식 매개 변수는 검사하는 형식, 바깥쪽 형식 또는 제네릭 메서드에서 올 수 있습니다. 다음과 같이 제네릭 형식 매개 변수의 원본을 확인할 수 있습니다.

- 먼저 이 속성을 사용하여 DeclaringMethod 형식 매개 변수가 제네릭 메서드에서 제공되는지 여부를 확인합니다. 속성 값이 null 참조가 아닌 경우 원본은 제네릭 메서드입니다.
- 원본이 제네릭 메서드가 아닌 경우 속성을 사용하여 DeclaringType 제네릭 형식 매개 변수가 속한 제네릭 형식을 확인합니다.

형식 매개 변수가 제네릭 메서드에 속하는 경우 속성은 제네릭 메서드 DeclaringType 를 선언한 형식을 반환하며 이는 관련이 없습니다.

제네릭 매개 변수의 위치

드문 경우이지만 선언 클래스의 형식 매개 변수 목록에서 형식 매개 변수의 위치를 확인해야 합니다. 예를 들어 앞의 예제에서 Type 형식을 나타내는 `B<int, V>` 개체가 있다고 가정해 보겠습니다. 이 메서드는 GetGenericArguments 형식 인수 목록을 제공하며, 검사 `V` 할 때 해당 인수와 DeclaringMethod 속성을 사용하여 DeclaringType 인수의 원본을 검색할 수 있습니다. 그런 다음, 속성을 사용하여 GenericParameterPosition 정의된 형식 매개 변수 목록에서 해당 위치를 확인할 수 있습니다. 이 예제 `V` 에서는 정의된 형식 매개 변수 목록의 위치 0(0)에 있습니다.

기본 형식 및 인터페이스 제약 조건

메서드를 GetGenericParameterConstraints 사용하여 형식 매개 변수의 기본 형식 제약 조건 및 인터페이스 제약 조건을 가져옵니다. 배열 요소의 순서는 중요하지 않습니다. 요소가 인터페이스 형식인 경우 인터페이스 제약 조건을 나타냅니다.

제네릭 매개 변수 특성

이 속성은 GenericParameterAttributes 분산(공변성 또는 반공변성) 및 형식 매개 변수의 특수 제약 조건을 나타내는 값을 가져옵니다 GenericParameterAttributes .

공변성 및 반공변성

형식 매개 변수가 공변성인지 반공변성인지 확인하려면 GenericParameterAttributes.VarianceMask 속성에서 반환된 값 GenericParameterAttributes에 GenericParameterAttributes 마스크를 적용합니다. 결과가 GenericParameterAttributes.None이면, 형식 매개변수가 불변입니다. 자세한 내용은 공변성 및 반공변성(Contravariance)을 참조하세요.

특수 제약 조건

형식 매개 변수의 특수 제약 조건을 확인하려면 GenericParameterAttributes.SpecialConstraintMask 속성에서 반환되는 값에 GenericParameterAttributes 마스크를 GenericParameterAttributes 적용합니다. 결과가 있으면 GenericParameterAttributes.None특별한 제약 조건이 없습니다. 형식 매개 변수는 참조 형식, nullable이 아닌 값 형식 및 매개 변수가 없는 생성자로 제한될 수 있습니다.

불변성

제네릭 형식의 리플렉션에 사용되는 일반적인 용어에 대한 불변 조건의 테이블은 Type.IsGenericType을 참조하세요. 제네릭 메서드와 관련된 추가 용어는 다음을 참조하세요 MethodBase.IsGenericMethod.

형식을 동적으로 로드하고 사용함

리플렉션은 언어 컴파일러에서 암시적 지연 바인딩을 구현하는 데 사용하는 인프라를 제공합니다. 바인딩은 고유하게 지정된 형식에 해당하는 선언(즉, 구현)을 찾는 프로세스입니다. 이 프로세스는 컴파일 시간이 아닌 런타임에 발생하는 경우 지연 바인딩이라고 합니다. Visual Basic을 사용하면 코드에서 암시적 지연 바인딩을 사용할 수 있습니다. Visual Basic 컴파일러는 리플렉션을 사용하여 개체 형식을 가져오는 도우미 메서드를 호출합니다. 도우미 메서드에 전달된 인수로 인해 런타임에 적절한 메서드가 호출됩니다. 이러한 인수는 메서드를 호출할 인스턴스(개체), 호출된 메서드의 이름(문자열) 및 호출된 메서드(개체 배열)에 전달된 인수입니다.

다음 예제에서 Visual Basic 컴파일러는 리플렉션을 암시적으로 사용하여 컴파일 시간에 형식을 알 수 없는 개체에서 메서드를 호출합니다. `HelloWorld` 클래스에는 `PrintHello` 메서드에 일부 텍스트를 전달하여 "Hello World"와 연결된 텍스트를 출력하는 `PrintHello` 메서드가 있습니다. 이 예제에서 호출된 `PrintHello` 메서드는 사실 Type.InvokeMember입니다. Visual Basic 코드를 사용하면 객체(`PrintHello`)의 형식을 컴파일 시간(초기 바인딩)에 이미 알고 있는 것처럼 메서드를 호출할 수 있습니다. 이는 런타임(늦은 바인딩) 대신에 이루어집니다.

VB

```
Module Hello
    Sub Main()
        ' Sets up the variable.
        Dim helloObj As Object
        ' Creates the object.
        helloObj = new HelloWorld()
        ' Invokes the print method as if it was early bound
        ' even though it is really late bound.
        helloObj.PrintHello("Visual Basic Late Bound")
    End Sub
End Module
```

사용자 지정 바인딩

지연 바인딩을 위해 컴파일러에서 암시적으로 사용되는 것 외에도 리플렉션을 코드에서 명시적으로 사용하여 지연 바인딩을 수행할 수 있습니다.

공용 언어 런타임은 여러 프로그래밍 언어를 지원하며 이러한 언어의 바인딩 규칙은 다릅니다. 초기 바인딩된 경우 코드 생성기는 이 바인딩을 완전히 제어할 수 있습니다. 그러나 리플렉션을 통한 지연 바인딩에서 바인딩은 사용자 지정된 바인딩에 의해 제어되어야 합니다. 클래스는 Binder 멤버 선택 및 호출에 대한 사용자 지정 제어를 제공합니다.

사용자 지정 바인딩을 사용하여 런타임에 어셈블리를 로드하고, 해당 어셈블리의 형식에 대한 정보를 가져오고, 원하는 형식을 지정한 다음, 메서드를 호출하거나 해당 형식의 필드 또는 속성

에 액세스할 수 있습니다. 이 기술은 개체 형식이 사용자 입력에 종속된 경우와 같이 컴파일 시간에 개체의 형식을 모르는 경우에 유용합니다.

다음 예제에서는 인수 형식 변환을 제공하지 않는 간단한 사용자 지정 바인더를 보여 줍니다. `Simple_Type.dll` 주 예제 앞에 나오는 코드입니다. `Simple_Type.dll`을(를) 먼저 빌드한 후, 빌드 시 프로젝트에 그에 대한 참조를 포함해야 합니다.

C#

```
// Code for building SimpleType.dll.
using System;
using System.Reflection;
using System.Globalization;
using Simple_Type;

namespace Simple_Type
{
    public class MySimpleClass
    {
        public void MyMethod(string str, int i)
        {
            Console.WriteLine("MyMethod parameters: {0}, {1}", str, i);
        }

        public void MyMethod(string str, int i, int j)
        {
            Console.WriteLine("MyMethod parameters: {0}, {1}, {2}",
                str, i, j);
        }
    }
}

namespace Custom_Binder
{
    class MyMainClass
    {
        static void Main()
        {
            // Get the type of MySimpleClass.
            Type myType = typeof(MySimpleClass);

            // Get an instance of MySimpleClass.
            MySimpleClass myInstance = new MySimpleClass();
            MyCustomBinder myCustomBinder = new MyCustomBinder();

            // Get the method information for the particular overload
            // being sought.
            MethodInfo myMethod = myType.GetMethod("MyMethod",
                BindingFlags.Public | BindingFlags.Instance,
                myCustomBinder, new Type[] {typeof(string),
                typeof(int)}, null);
            Console.WriteLine(myMethod.ToString());
```

```
        // Invoke the overload.
        myType.InvokeMember("MyMethod", BindingFlags.InvokeMethod,
            myCustomBinder, myInstance,
            new Object[] {"Testing...", (int)32});
    }
}

// **************************************************
//  A simple custom binder that provides no
//  argument type conversion.
// **************************************************
class MyCustomBinder : Binder
{
    public override MethodBase BindToMethod(
        BindingFlags bindingAttr,
        MethodBase[] match,
        ref object[] args,
        ParameterModifier[] modifiers,
        CultureInfo culture,
        string[] names,
        out object state)
    {
        if (match == null)
        {
            throw new ArgumentNullException("match");
        }
        // Arguments are not being reordered.
        state = null;
        // Find a parameter match and return the first method with
        // parameters that match the request.
        foreach (MethodBase mb in match)
        {
            ParameterInfo[] parameters = mb.GetParameters();

            if (ParametersMatch(parameters, args))
            {
                return mb;
            }
        }
        return null;
    }

    public override FieldInfo BindToField(BindingFlags bindingAttr,
        FieldInfo[] match, object value, CultureInfo culture)
    {
        if (match == null)
        {
            throw new ArgumentNullException("match");
        }
        foreach (FieldInfo fi in match)
        {
            if (fi.GetType() == value.GetType())
            {
                return fi;
            }
```

```
        }
        return null;
    }

    public override MethodBase SelectMethod(
        BindingFlags bindingAttr,
        MethodBase[] match,
        Type[] types,
        ParameterModifier[] modifiers)
    {
        if (match == null)
        {
            throw new ArgumentNullException("match");
        }

        // Find a parameter match and return the first method with
        // parameters that match the request.
        foreach (MethodBase mb in match)
        {
            ParameterInfo[] parameters = mb.GetParameters();
            if (ParametersMatch(parameters, types))
            {
                return mb;
            }
        }

        return null;
    }

    public override PropertyInfo SelectProperty(
        BindingFlags bindingAttr,
        PropertyInfo[] match,
        Type returnType,
        Type[] indexes,
        ParameterModifier[] modifiers)
    {
        if (match == null)
        {
            throw new ArgumentNullException("match");
        }
        foreach (PropertyInfo pi in match)
        {
            if (pi.GetType() == returnType &&
                ParametersMatch(pi.GetIndexParameters(), indexes))
            {
                return pi;
            }
        }
        return null;
    }

    public override object ChangeType(
        object value,
        Type myChangeType,
        CultureInfo culture)
```

```
{
    try
    {
        object newType;
        newType = Convert.ChangeType(value, myChangeType);
        return newType;
    }
    // Throw an InvalidCastException if the conversion cannot
    // be done by the Convert.ChangeType method.
    catch (InvalidCastException)
    {
        return null;
    }
}

public override void ReorderArgumentArray(ref object[] args,
    object state)
{
    // No operation is needed here because BindToMethod does not
    // reorder the args array. The most common implementation
    // of this method is shown below.

    // ((BinderState)state).args.CopyTo(args, 0);
}

// Returns true only if the type of each object in a matches
// the type of each corresponding object in b.
private bool ParametersMatch(ParameterInfo[] a, object[] b)
{
    if (a.Length != b.Length)
    {
        return false;
    }
    for (int i = 0; i < a.Length; i++)
    {
        if (a[i].ParameterType != b[i].GetType())
        {
            return false;
        }
    }
    return true;
}

// Returns true only if the type of each object in a matches
// the type of each corresponding entry in b.
private bool ParametersMatch(ParameterInfo[] a, Type[] b)
{
    if (a.Length != b.Length)
    {
        return false;
    }
    for (int i = 0; i < a.Length; i++)
    {
        if (a[i].ParameterType != b[i])
        {
```

```
                    return false;
                }
            }
            return true;
        }
    }
}
```

InvokeMember 및 CreateInstance

형식의 멤버를 호출하는 데 사용합니다 Type.InvokeMember . 다양한 클래스의 메서드(예: `CreateInstance` 및Activator.CreateInstance)는 Assembly.CreateInstance 지정된 형식의 `InvokeMember` 새 인스턴스를 만드는 특수 형식입니다. 이 `Binder` 클래스는 이러한 메서드에서 오버로드 확인 및 인수 강제 변환에 사용됩니다.

다음 예제에서는 인수 강제 변환(형식 변환) 및 멤버 선택의 세 가지 가능한 조합을 보여 줍니다. 케이스 1에서는 인수 강제 변환 또는 멤버 선택이 필요 없습니다. 사례 2에서는 멤버 선택만 필요합니다. 사례 3에서는 인수 강제 변환만 필요합니다.

C#

```
public class CustomBinderDriver
{
    public static void Main()
    {
        Type t = typeof(CustomBinderDriver);
        CustomBinder binder = new CustomBinder();
        BindingFlags flags = BindingFlags.InvokeMethod | BindingFlags.Instance |
            BindingFlags.Public | BindingFlags.Static;
        object[] args;

        // Case 1. Neither argument coercion nor member selection is needed.
        args = new object[] {};
        t.InvokeMember("PrintBob", flags, binder, null, args);

        // Case 2. Only member selection is needed.
        args = new object[] {42};
        t.InvokeMember("PrintValue", flags, binder, null, args);

        // Case 3. Only argument coercion is needed.
        args = new object[] {"5.5"};
        t.InvokeMember("PrintNumber", flags, binder, null, args);
    }

    public static void PrintBob()
    {
        Console.WriteLine("PrintBob");
    }

    public static void PrintValue(long value)
```

```
    {
        Console.WriteLine($"PrintValue({value})");
    }

    public static void PrintValue(string value)
    {
        Console.WriteLine("PrintValue\"{0}\")", value);
    }

    public static void PrintNumber(double value)
    {
        Console.WriteLine($"PrintNumber ({value})");
    }
}
```

이름이 같은 멤버를 둘 이상 사용할 수 있는 경우 오버로드 확인이 필요합니다. Binder.BindToMethod 및 Binder.BindToField 메서드는 단일 멤버에 대한 바인딩을 해결하는 데 사용됩니다. `Binder.BindToMethod` 또한 속성 접근자 `get`와 `set`를 통해 속성을 확인합니다.

`BindToMethod`는 호출할 수 없을 경우 null 참조(MethodBase의 Visual Basic)를 반환하거나, 호출할 수 있는 `Nothing`를 반환합니다. 반환 값은 `MethodBase` 일반적인 경우이지만 *일치* 매개 변수에 포함된 값 중 하나일 필요는 없습니다.

ByRef 인수가 있으면 호출자가 인수를 다시 가져올 수 있습니다. 따라서 `Binder` 인수 배열을 조작한 경우 `BindToMethod` 클라이언트가 인수 배열을 원래 형식으로 다시 매핑할 수 있습니다. 이렇게 하려면 호출자가 인수의 순서가 변경되지 않도록 보장해야 합니다. 인수가 이름으로 `Binder` 전달되면 인수 배열을 다시 정렬합니다. 이것이 호출자가 보는 내용입니다. 자세한 내용은 Binder.ReorderArgumentArray를 참조하세요.

사용 가능한 멤버 집합은 형식 또는 모든 기본 형식에 정의된 멤버입니다. BindingFlags이(가) 지정된 경우, 모든 접근성의 멤버가 집합에 반환됩니다. 지정하지 않은 경우 `BindingFlags.NonPublic` 바인더는 접근성 규칙을 적용해야 합니다. 반드시 `Public` 또는 `NonPublic` 바인딩 플래그를 지정할 때 `Instance` 또는 `Static` 바인딩 플래그도 함께 지정해야 합니다. 그렇지 않으면 멤버가 반환되지 않습니다.

지정된 이름의 멤버가 하나만 있는 경우 콜백이 필요하지 않으며 해당 메서드에서 바인딩이 수행됩니다. 코드 예제의 사례 1은 하나의 `PrintBob` 메서드만 사용할 수 있으므로 콜백이 필요하지 않음을 보여 줍니다.

사용 가능한 집합에 둘 이상의 멤버가 있는 경우 이러한 모든 메서드가 전달되어 `BindToMethod` 적절한 메서드를 선택하고 반환합니다. 코드 예제의 Case 2에는 두 개의 메서드가 있습니다 `PrintValue`. 에 대한 호출 `BindToMethod`에서 적절한 메서드를 선택합니다.

ChangeType 는 실제 인수를 선택한 메서드의 형식 인수 형식으로 변환하는 인수 강제 변환(형식 변환)을 수행합니다. `ChangeType` 는 형식이 정확히 일치하는 경우에도 모든 인수에 대해 호출

됩니다.

코드 예제의 Case 3에서 값이 "5.5"인 형식 `String` 의 실제 인수는 형식 `Double`의 형식 인수가 있는 메서드에 전달됩니다. 호출이 성공하려면 문자열 값 "5.5"를 double 값으로 변환해야 합니다. `ChangeType` 는 이 변환을 수행합니다.

`ChangeType` 는 다음 표와 같이 무손실 또는 확대 강제 변환만 수행합니다.

테이블 확장

원본 유형	대상 형식
모든 형식	기본 형식
모든 형식	구현하는 인터페이스
`Char`	`UInt16`, `UInt32`, `Int32`, `UInt64`, `Int64`, `Single` `Double`
`Byte`	`Char`, `UInt16`, `Int16`, `UInt32`, `Int32`, `UInt64` `Int64`, `Single` `Double`
`SByte`	`Int16`, `Int32`, `Int64`, `Single` `Double`
`UInt16`	`UInt32`, `Int32`, `UInt64`, `Int64`, `Single` `Double`
`Int16`	`Int32`, `Int64`, , `Single`, `Double`
`UInt32`	`UInt64`, `Int64`, , `Single`, `Double`
`Int32`	`Int64`, `Single` `Double`
`UInt64`	`Single`, `Double`
`Int64`	`Single`, `Double`
`Single`	`Double`
참조가 아닌 유형	참조 형식

Type 클래스에는 `Get` 형식 `Binder` 의 매개 변수를 사용하여 특정 멤버에 대한 참조를 확인하는 메서드가 있습니다. Type.GetConstructor, Type.GetMethod및 Type.GetProperty 해당 멤버에 대한 서명 정보를 제공하여 현재 형식의 특정 멤버를 검색합니다. Binder.SelectMethod 는 Binder.SelectProperty 적절한 메서드의 지정된 서명 정보를 선택하기 위해 다시 호출됩니다.

참고하십시오

- Type.InvokeMember
- Assembly.Load

- 보기 형식 정보
- .NET의 형식 변환

Last updated on 2026. 01. 22.

사용자 지정 특성에 액세스

아티클 • 2025. 04. 30.

특성이 프로그램 요소와 연결되면 리플렉션을 사용하여 해당 존재와 값을 쿼리할 수 있습니다. .NET은 실행을 위해 로드할 수 없는 코드를 검사하는 데 사용할 수 있는 기능을 제공합니다 MetadataLoadContext.

MetadataLoadContext

컨텍스트에 로드된 코드는 MetadataLoadContext 실행할 수 없습니다. 즉, 생성자를 실행해야 하므로 사용자 지정 특성의 인스턴스를 만들 수 없습니다. 컨텍스트에서 MetadataLoadContext 사용자 지정 특성을 로드하고 검사하려면 클래스를 CustomAttributeData 사용합니다. 정적 CustomAttributeData.GetCustomAttributes 메서드의 적절한 오버로드를 사용하여 이 클래스의 인스턴스를 가져올 수 있습니다. 자세한 내용은 방법: MetadataLoadContext를 사용하여 어셈블리 콘텐츠 검사를 참조하세요.

실행 컨텍스트

실행 컨텍스트에서 특성을 쿼리하는 주요 리플렉션 메서드는 MemberInfo.GetCustomAttributes 및 Attribute.GetCustomAttributes입니다.

연결된 어셈블리와 관련하여 사용자 지정 특성의 접근성을 확인합니다. 이는 사용자 지정 특성이 연결된 어셈블리의 형식에 대한 메서드가 사용자 지정 특성의 생성자를 호출할 수 있는지 여부를 확인하는 것과 같습니다.

Assembly.GetCustomAttributes(Boolean)과 같은 메서드는 형식 인수의 가시성과 접근성을 확인합니다. 사용자 정의 형식이 포함된 어셈블리의 코드만 .를 사용하여 `GetCustomAttributes` 해당 형식의 사용자 지정 특성을 검색할 수 있습니다.

다음 C# 예제는 일반적인 사용자 지정 특성 디자인 패턴입니다. 런타임 사용자 지정 특성 리플렉션 모델을 보여 줍니다.

C#

```
System.DLL
public class DescriptionAttribute : Attribute
{
}

System.Web.DLL
internal class MyDescriptionAttribute : DescriptionAttribute
{
}
```

```
public class LocalizationExtenderProvider
{
    [MyDescriptionAttribute(...)]
    public CultureInfo GetLanguage(...)
    {
    }
}
```

런타임이 DescriptionAttribute 메서드에 연결된 `GetLanguage` 공용 사용자 지정 특성 형식에 대한 사용자 지정 특성을 검색하려고 하면, 다음과 같은 작업을 수행합니다.

1. 런타임은 형식 인수 `DescriptionAttribute` `Type.GetCustomAttributes(Type type)` 가 public 이므로 표시되고 액세스할 수 있는지 확인합니다.
2. 런타임은 `MyDescriptionAttribute` 에서 파생된 사용자 정의 형식 `DescriptionAttribute` 이 `GetLanguage()` 메서드에 연결된 System.Web.dll 어셈블리 내에서 표시되고 액세스할 수 있는지 확인합니다.
3. 런타임은 생성자가 `MyDescriptionAttribute` System.Web.dll 어셈블리 내에서 표시되고 액세스할 수 있는지 확인합니다.
4. 런타임은 사용자 지정 특성 매개 변수를 사용하여 생성자를 `MyDescriptionAttribute` 호출하고 새 개체를 호출자에게 반환합니다.

사용자 지정 특성 리플렉션 모델은 형식이 정의된 어셈블리 외부의 사용자 정의 형식 인스턴스를 누수할 수 있습니다. 이는 개체 배열 반환과 같이 Type.GetMethods 사용자 정의 형식의 인스턴스를 반환하는 런타임 시스템 라이브러리의 `RuntimeMethodInfo` 멤버와 다르지 않습니다. 클라이언트가 사용자 정의 사용자 지정 특성 형식에 대한 정보를 검색하지 못하도록 하려면 형식의 멤버를 비공유식으로 정의합니다.

다음 예제에서는 리플렉션을 사용하여 사용자 지정 특성에 액세스하는 기본적인 방법을 보여 줍니다.

C#

```
using System;

public class ExampleAttribute : Attribute
{
    private string stringVal;

    public ExampleAttribute()
    {
        stringVal = "This is the default string.";
    }

    public string StringValue
    {
        get { return stringVal; }
        set { stringVal = value; }
```

```
    }
}

[Example(StringValue="This is a string.")]
class Class1
{
    public static void Main()
    {
        System.Reflection.MemberInfo info = typeof(Class1);
        foreach (object attrib in info.GetCustomAttributes(true))
        {
            Console.WriteLine(attrib);
        }
    }
}
```

참고하십시오

- MemberInfo.GetCustomAttributes
- Attribute.GetCustomAttributes
- 형식 정보 보기

완전한 정규화된 타입 이름 지정

다양한 리플렉션 작업에 유효한 입력을 하려면 형식 이름을 지정해야 합니다. 정규화된 형식 이름은 어셈블리 이름 사양, 네임스페이스 사양 및 형식 이름으로 구성됩니다. 형식 이름 사양은 Type.GetType, Module.GetType, ModuleBuilder.GetType, 및 Assembly.GetType와 같은 메서드에서 사용됩니다.

형식 이름에 대한 문법

문법은 정식 언어의 구문을 정의합니다. 다음 표에서는 유효한 입력을 인식하는 방법을 설명하는 어휘 규칙을 나열합니다. 터미널(더 이상 환원할 수 없는 요소)은 모든 대문자로 표시됩니다. 비단말 기호(더 환원 가능한 요소)는 대소문자가 혼합된 형태 또는 작은따옴표로 묶인 문자열로 표시되지만, 작은따옴표(')는 구문 자체에 포함되지 않습니다. 파이프 문자(|)는 하위 규칙이 있는 규칙을 나타냅니다.

antlr

```
TypeSpec
    : ReferenceTypeSpec
    | SimpleTypeSpec
    ;

ReferenceTypeSpec
    : SimpleTypeSpec '&'
    ;

SimpleTypeSpec
    : PointerTypeSpec
    | GenericTypeSpec
    | TypeName
    ;

GenericTypeSpec
    : SimpleTypeSpec ` NUMBER
    | SimpleTypeSpec ` NUMBER '[' GenericArguments ']'
    ;

GenericArguments
    : GenericArgument
    | GenericArguments ',' GenericArgument
    ;

GenericArgument
    : TypeSpec
    | '[' TypeSpec ']'
    ;

PointerTypeSpec
    : SimpleTypeSpec '*'
```

```
    ;

ArrayTypeSpec
    : SimpleTypeSpec '[ReflectionDimension]'
    | SimpleTypeSpec '[ReflectionEmitDimension]'
    ;

ReflectionDimension
    : '*'
    | ReflectionDimension ',' ReflectionDimension
    | NOTOKEN
    ;

ReflectionEmitDimension
    : '*'
    | Number '..' Number
    | Number '…'
    | ReflectionDimension ',' ReflectionDimension
    | NOTOKEN
    ;

Number
    : [0-9]+
    ;

TypeName
    : NamespaceTypeName
    | NamespaceTypeName ',' AssemblyNameSpec
    ;

NamespaceTypeName
    : NestedTypeName
    | NamespaceSpec '.' NestedTypeName
    ;

NestedTypeName
    : IDENTIFIER
    | NestedTypeName '+' IDENTIFIER
    ;

NamespaceSpec
    : IDENTIFIER
    | NamespaceSpec '.' IDENTIFIER
    ;

AssemblyNameSpec
    : IDENTIFIER
    | IDENTIFIER ',' AssemblyProperties
    ;

AssemblyProperties
    : AssemblyProperty
    | AssemblyProperties ',' AssemblyProperty
    ;
```

```
AssemblyProperty
    : AssemblyPropertyName '=' AssemblyPropertyValue
    ;
```

특수 문자 지정

형식 이름에서 IDENTIFIER는 언어 규칙에 따라 결정되는 유효한 이름입니다.

식별자의 일부로 사용될 때 백슬래시(\)를 이스케이프 문자로 사용하여 다음 토큰을 구분합니다.

테이블 확장

토큰	의미
`\,`	어셈블리 구분 기호
`\+`	중첩된 형식 구분 기호입니다.
`\&`	참조 형식
`\*`	포인터 형식입니다.
`\[`	배열 차원 구분 기호입니다.
`\]`	배열 차원 구분 기호입니다.
`\.`	배열 사양에서 마침표가 사용되는 경우에만 마침표 앞에 백슬래시를 사용합니다. NamespaceSpec의 기간은 백슬래시를 사용하지 않습니다.
`\\`	문자열 리터럴에서 필요할 때 백슬래시를 사용하세요.

AssemblyNameSpec을 제외한 모든 TypeSpec 구성 요소에서 공백은 관련이 있습니다. AssemblyNameSpec에서 ',' 구분 기호 앞의 공백은 관련이 있지만 ',' 구분 기호 뒤의 공백은 무시됩니다.

리플렉션 클래스들, 예를 들어 Type.FullName, GetType와 같은 경우 반환된 이름을 `MyType.GetType(myType.FullName)` 호출에 사용할 수 있도록 난독화된 이름을 반환합니다.

예를 들어 형식의 정규화된 이름은 다음과 같습니다 `Ozzy.OutBack.Kangaroo+Wallaby,MyAssembly`.

네임스페이스가 있으면 `Ozzy.Out+Back` 더하기 기호 앞에 백슬래시가 와야 합니다. 그렇지 않을 경우에는 파서가 중첩 구분 기호로 해석합니다. 리플렉션은 이 문자열을 `Ozzy.Out\+Back.Kangaroo+Wallaby,MyAssembly`로 내보낸다.

어셈블리 이름 지정

어셈블리 이름 사양에 필요한 최소 정보는 어셈블리의 텍스트 이름(IDENTIFIER)입니다. 다음 표에 설명된 대로 쉼표로 구분된 속성/값 쌍 목록으로 식별자를 따를 수 있습니다. IDENTIFIER 명명은 파일 이름 지정 규칙을 따라야 합니다. 식별자는 대/소문자를 구분하지 않습니다.

테이블 확장

속성 이름	설명	허용되는 값
버전	어셈블리 버전 번호	*Major.Minor.Build.Revision*. 여기서 *Major*, *Minor*, *Build* 및 *Revision* 은 0에서 65535 사이의 정수입니다.
PublicKey	전체 공개 키	16진수 형식의 전체 공개 키의 문자열 값입니다. 프라이빗 어셈블리를 명시적으로 나타내려면 null 참조(Visual Basic에서는 **Nothing**)를 지정합니다.
PublicKeyToken	공개 키 토큰(전체 공개 키의 8바이트 해시)	16진수 형식의 공개 키 토큰의 문자열 값입니다. 프라이빗 어셈블리를 명시적으로 나타내려면 null 참조(Visual Basic에서는 **Nothing**)를 지정합니다.
문화	어셈블리 문화	RFC-1766 형식의 어셈블리에 대한 언어 문화 또는 언어 독립적(비종속적) 어셈블리의 경우 "중립"입니다.
사용자 지정	사용자 지정 BLOB(Binary Large Object)입니다. 현재 Ngen(네이티브 이미지 생성기)에서 생성된 어셈블리에서만 사용됩니다.	네이티브 이미지 생성기 도구에서 설치되는 어셈블리가 네이티브 이미지이므로 네이티브 이미지 캐시에 설치되도록 어셈블리 캐시에 알리는 데 사용되는 사용자 지정 문자열입니다. zap 문자열이라고도 합니다.

다음 예제에서는 기본 문화권이 있는 간단히 명명된 어셈블리**AssemblyName**을 보여줍니다.

C#

```
com.microsoft.crypto, Culture=""
```

다음 예제에서는 문화권이 "en"인 강력한 이름의 어셈블리에 대해 완전히 지정된 참조를 보여줍니다.

C#

```
com.microsoft.crypto, Culture=en, PublicKeyToken=a5d015c7d5a0b012,
    Version=1.0.0.0
```

다음 예제에서는 각각 강력한 어셈블리 또는 단순히 명명된 어셈블리에 의해 충족될 수 있는 부분적으로 지정된 **AssemblyName**을 보여 줍니다.

C#

```
com.microsoft.crypto
com.microsoft.crypto, Culture=""
com.microsoft.crypto, Culture=en
```

다음 예제에서는 각각 단순히 명명된 어셈블리에 의해 충족되어야 하는 부분적으로 지정된 **AssemblyName**을 보여 줍니다.

C#

```
com.microsoft.crypto, Culture="", PublicKeyToken=null
com.microsoft.crypto, Culture=en, PublicKeyToken=null
```

다음 예제에서는 각각 강력한 이름의 어셈블리에 의해 충족되어야 하는 부분적으로 지정된 **AssemblyName**을 보여 줍니다.

C#

```
com.microsoft.crypto, Culture="", PublicKeyToken=a5d015c7d5a0b012
com.microsoft.crypto, Culture=en, PublicKeyToken=a5d015c7d5a0b012,
    Version=1.0.0.0
```

제네릭 형식 지정

`SimpleTypeSpec`NUMBER`는 1개에서 *n*개까지의 제네릭 형식 매개 변수를 갖는 제네릭 형식 정의를 나타냅니다. 예를 들어 열린 제네릭 형식 `List<T>`에 대한 참조를 얻으려면 .를 사용합니다 `Type.GetType("System.Collections.Generic.List`1")`. 열린 제네릭 형식 `Dictionary<TKey,TValue>`에 대한 참조를 얻으려면 .를 사용합니다 `Type.GetType("System.Collections.Generic.Dictionary`2")`.

형식 매개 변수가 특정 형식으로 대체된 *구성된* 제네릭 형식을 지정하려면, 차수 뒤에 대괄호로 형식 인수를 추가하십시오 `SimpleTypeSpec`NUMBER[TypeArg1,TypeArg2]`. 예를 들어 `List<String>` 에 대한 참조를 얻으려면 `Type.GetType("System.Collections.Generic.List`1[System.String]")` 을 사용합니다. `Dictionary<String, Int32>`에 대한 참조를 얻으려면 `Type.GetType("System.Collections.Generic.Dictionary`2[System.String,System.Int32]")`을(를) 사용합니다.

형식 인수가 어셈블리로 한정된 경우 어셈블리 이름의 쉼표가 형식 인수 구분 기호로 잘못 해석되지 않도록 자체 대괄호로 묶습니다. 다음은 그 예입니다.

```
```

```
"System.Collections.Generic.Dictionary`2[[System.String, System.Private.CoreLib],
[System.Int32, System.Private.CoreLib]]"
```

어떤 괄호도 포함하지 않는 형식 인수는 어셈블리 한정자를 포함할 수 없습니다. 어셈블리 정규화 형식 인수와 정규화되지 않은 인수를 혼합하려면 어셈블리로 한정된 형식 인수만 대괄호로 묶습니다.

```

"System.Collections.Generic.Dictionary`2[System.String,[MyNamespace.MyType,
MyAssembly]]"
```

이 Type.AssemblyQualifiedName 속성은 `Type.GetType`에서 허용하는 형식으로 형식의 이름을 반환합니다. 생성된 제네릭 형식에 대해 올바른 형식의 문자열을 가져오는 데 사용합니다. 예를 들어, `typeof(Dictionary<string, int>).AssemblyQualifiedName`은/는 `Type.GetType`에 전달할 수 있는 정규화되고 정확하게 이스케이프된 이름을 반환합니다.

포인터 지정

SimpleTypeSpec*은 관리되지 않는 포인터를 나타냅니다. 예를 들어 MyType 형식에 대한 포인터를 얻으려면 .를 사용합니다 `Type.GetType("MyType*")`. MyType 형식에 대한 포인터에 대한 포인터를 얻으려면 .를 사용합니다 `Type.GetType("MyType**")`.

참조 지정

SimpleTypeSpec & 는 관리되는 포인터 또는 참조를 나타냅니다. 예를 들어 MyType 형식에 대한 참조를 얻으려면 .를 사용합니다 `Type.GetType("MyType &")`. 포인터와 달리 참조는 한 수준으로 제한됩니다.

배열 지정

BNF 문법에서 ReflectionEmitDimension은 ModuleBuilder.GetType를 사용하여 불완전한 형식 정의를 검색하는 데만 적용됩니다. 불완전한 형식 정의는 TypeBuilder를 사용하여 생성되었으나 System.Reflection.Emit가 호출되지 않은 객체입니다. ReflectionDimension을 사용하여 완료된 형식 정의, 즉 로드된 형식을 검색할 수 있습니다.

배열은 배열의 순위를 지정하여 리플렉션에서 액세스됩니다.

- `Type.GetType("MyArray[]")` 는 하한이 0인 1차원 배열을 가져옵니다.
- `Type.GetType("MyArray[*]")` 알 수 없는 하한이 있는 1차원 배열을 가져옵니다.

- `Type.GetType("MyArray[][]")` 는 2차원 배열의 배열을 가져옵니다.
- `Type.GetType("MyArray[*,*]")` 및 `Type.GetType("MyArray[,]")`는 하한이 알려지지 않은 직사각형 2차원 배열을 가져옵니다.

런타임 관점에서 `MyArray[] != MyArray[*]` 볼 때, 다차원 배열의 경우 두 표기법이 같습니다. 즉, `Type.GetType("MyArray [,]") == Type.GetType("MyArray[*,*]")`**true**로 평가됩니다.

`ModuleBuilder.GetType` 의 경우 `MyArray[0..5]`크기가 6, 하한이 0인 1차원 배열을 나타냅니다. `MyArray[4…]` 는 알 수 없는 크기와 하한 4의 1차원 배열을 나타냅니다.

참고하십시오

- AssemblyName
- ModuleBuilder
- TypeBuilder
- Type.FullName
- Type.GetType
- Type.AssemblyQualifiedName
- 형식 정보 보기

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 31.

방법: 리플렉션을 사용하여 제네릭 형식 검사 및 인스턴스화

2025. 06. 17.

제네릭 형식에 대한 정보는 제네릭 형식을 나타내는 개체를 검사하여 Type 다른 형식에 대한 정보와 동일한 방식으로 가져옵니다. 원칙적 차이점은 제네릭 형식에 제네릭 형식 매개 변수를 나타내는 개체 목록이 Type 있다는 것입니다. 이 섹션의 첫 번째 절차에서는 제네릭 형식을 검사합니다.

제네릭 형식 정의의 Type 형식 매개 변수에 형식 인수를 바인딩하여 생성된 형식을 나타내는 개체를 만들 수 있습니다. 두 번째 절차에서는 이를 보여 줍니다.

제네릭 형식 및 해당 형식 매개 변수를 검사하려면

1. 제네릭 형식을 나타내는 인스턴스 Type 를 가져옵니다. 다음 코드에서는 C# `typeof` 연산자(`GetType` Visual Basic)를 사용하여 형식을 가져옵니다. 다른 방법으로 Type 개체를 얻으려면 Type을 참조하세요. 이 절차의 나머지 부분에는 형식이 명명 `t`된 메서드 매개 변수에 포함됩니다.

   C#

   ```
   Type d1 = typeof(Dictionary<,>);
   ```

2. IsGenericType 속성을 사용하여 형식이 제네릭인지 여부를 확인하고 속성을 사용하여 IsGenericTypeDefinition 형식이 제네릭 형식 정의인지 여부를 확인합니다.

   C#

   ```
   Console.WriteLine($"   Is this a generic type? {t.IsGenericType}");
   Console.WriteLine($"   Is this a generic type definition?
   {t.IsGenericTypeDefinition}");
   ```

3. GetGenericArguments 메서드를 사용하여 제네릭 형식 인수가 포함된 배열을 가져옵니다.

   C#

   ```
   Type[] typeParameters = t.GetGenericArguments();
   ```

4. 각 형식 인수에 대해 속성을 사용하여 IsGenericParameter 형식 매개 변수(예: 제네릭 형식 정의) 또는 형식 매개 변수(예: 생성된 형식)에 대해 지정된 형식인지 확인합니다.

```
C#

Console.WriteLine($"   List {typeParameters.Length} type arguments:");
foreach (Type tParam in typeParameters)
{
    if (tParam.IsGenericParameter)
    {
        DisplayGenericParameter(tParam);
    }
    else
    {
        Console.WriteLine($"      Type argument: {tParam}");
    }
}
```

5. 형식 시스템에서 제네릭 형식 매개 변수는 일반 형식과 마찬가지로 인스턴스 Type로 표시됩니다. 다음 코드는 제네릭 형식 매개 변수를 Type 나타내는 개체의 이름 및 매개 변수 위치를 표시합니다. 매개 변수 위치는 여기서 간단한 정보입니다. 다른 제네릭 형식의 형식 인수로 사용된 형식 매개 변수를 검사할 때 더 중요합니다.

```
C#

Console.WriteLine($"      Type parameter: {tp.Name} position
{tp.GenericParameterPosition}");
```

6. 메서드를 사용하여 GetGenericParameterConstraints 단일 배열의 모든 제약 조건을 가져와 제네릭 형식 매개 변수의 기본 형식 제약 조건 및 인터페이스 제약 조건을 결정합니다. 제약 조건은 특정 순서로 보장되지 않습니다.

```
C#

foreach (Type iConstraint in tp.GetGenericParameterConstraints())
{
    if (iConstraint.IsInterface)
    {
        Console.WriteLine($"         Interface constraint: {iConstraint}");
    }
}

Console.WriteLine($"         Base type constraint: {tp.BaseType ??
tp.BaseType: None}");
```

7. GenericParameterAttributes 이 속성을 사용하여 형식 매개 변수에 대한 특수 제약 조건을 검색합니다(예: 참조 형식이어야 함). 이 속성에는 다음 코드와 같이 마스킹할 수 있는 분산을 나타내는 값도 포함됩니다.

```
C#
```

```
GenericParameterAttributes sConstraints =
    tp.GenericParameterAttributes &
    GenericParameterAttributes.SpecialConstraintMask;
```

8. 특수 제약 조건 특성은 플래그이며, 특수 제약 조건이 없는 경우를 나타내는 동일한 플래그(GenericParameterAttributes.None)가 공변성이나 반공변성이 없는 경우도 나타냅니다. 따라서 이러한 조건 중 하나를 테스트하려면 적절한 마스크를 사용해야 합니다. 이 경우 특수 제약 조건 플래그를 격리하는 데 사용합니다 GenericParameterAttributes.SpecialConstraintMask .

C#

```
if (sConstraints == GenericParameterAttributes.None)
{
    Console.WriteLine("         No special constraints.");
}
else
{
    if (GenericParameterAttributes.None != (sConstraints &
        GenericParameterAttributes.DefaultConstructorConstraint))
    {
        Console.WriteLine("         Must have a parameterless constructor.");
    }
    if (GenericParameterAttributes.None != (sConstraints &
        GenericParameterAttributes.ReferenceTypeConstraint))
    {
        Console.WriteLine("         Must be a reference type.");
    }
    if (GenericParameterAttributes.None != (sConstraints &
        GenericParameterAttributes.NotNullableValueTypeConstraint))
    {
        Console.WriteLine("         Must be a non-nullable value type.");
    }
}
```

제네릭 형식의 인스턴스 생성

제네릭 형식은 템플릿과 같습니다. 제네릭 형식 매개 변수에 실제 형식을 지정하지 않으면 인스턴스를 만들 수 없습니다. 런타임에 리플렉션을 사용하려면 메서드가 MakeGenericType 필요합니다.

1. 제네릭 형식을 Type 나타내는 개체를 가져옵니다. 다음 코드는 두 가지 방법으로 제네릭 형식 Dictionary<TKey,TValue>을 가져옵니다. 하나는 형식을 설명하는 문자열을 사용하여 메서드 오버로드 Type.GetType(String)를 사용하는 것이고, 다른 하나는 생성된 형식 GetGenericTypeDefinition(Visual Basic에서는 `Dictionary\<String, Example>`)에서 메서드

`Dictionary(Of String, Example)`를 호출하여 가져오는 것입니다. 메서드에는 MakeGenericType 제네릭 형식 정의가 필요합니다.

```
C#

// Use the typeof operator to create the generic type
// definition directly. To specify the generic type definition,
// omit the type arguments but retain the comma that separates
// them.
Type d1 = typeof(Dictionary<,>);

// You can also obtain the generic type definition from a
// constructed class. In this case, the constructed class
// is a dictionary of Example objects, with String keys.
Dictionary<string, Example> d2 = [];
// Get a Type object that represents the constructed type,
// and from that get the generic type definition. The
// variables d1 and d4 contain the same type.
Type d3 = d2.GetType();
Type d4 = d3.GetGenericTypeDefinition();
```

2. 형식 매개 변수를 대체할 형식 인수 배열을 생성합니다. 배열은 형식 매개 변수 목록에 나타나는 것과 동일한 순서로 올바른 수의 Type 개체를 포함해야 합니다. 이 경우 키(첫 번째 형식 매개 변수)는 형식 String이고 사전의 값은 이름이 지정된 `Example`클래스의 인스턴스입니다.

```
C#

Type[] typeArgs = [typeof(string), typeof(Example)];
```

3. 메서드를 MakeGenericType 호출하여 형식 인수를 형식 매개 변수에 바인딩하고 형식을 생성합니다.

```
C#

Type constructed = d1.MakeGenericType(typeArgs);
```

4. 메서드 오버로드를 CreateInstance(Type) 사용하여 생성된 형식의 개체를 만듭니다. 다음 코드는 `Example` 클래스의 두 인스턴스를 결과 `Dictionary<String, Example>` 개체에 저장합니다.

```
C#

_ = Activator.CreateInstance(constructed);
```

예시

다음 코드 예제에서는 제네릭 형식 정의 및 코드에 사용된 생성된 형식을 검사하고 해당 정보를 표시하는 메서드를 정의 `DisplayGenericType` 합니다. `DisplayGenericType` 메서드는 IsGenericType, IsGenericParameter, 및 GenericParameterPosition 속성과 GetGenericArguments 메서드를 사용하는 방법을 보여줍니다.

또한 이 예제에서는 제네릭 형식 매개 변수를 검사하고 해당 제약 조건을 표시하는 메서드를 정의 `DisplayGenericParameter` 합니다.

코드 예제에서는 형식 매개 변수 제약 조건을 보여 주는 제네릭 형식을 포함하여 테스트 형식 집합을 정의하고 이러한 형식에 대한 정보를 표시하는 방법을 보여 줍니다.

예제에서는 Dictionary<TKey,TValue> 클래스를 사용하여 형식 인수의 배열을 생성하고 MakeGenericType 메서드를 호출하여 형식을 구성합니다. 이 프로그램은 Type을 사용하여 생성된 MakeGenericType 객체와 Type를 (Visual Basic에서 `typeof`) 사용하여 얻어진 `GetType` 객체를 비교하여 동일함을 보여줍니다. 마찬가지로 프로그램은 메서드를 GetGenericTypeDefinition 사용하여 생성된 형식의 제네릭 형식 정의를 가져오고 클래스를 나타내는 Type 개체와 비교합니다Dictionary<TKey,TValue>.

C#

```
using System.Reflection;

// Define an example interface.
public interface ITestArgument { }

// Define an example base class.
public class TestBase { }

// Define a generic class with one parameter. The parameter
// has three constraints: It must inherit TestBase, it must
// implement ITestArgument, and it must have a parameterless
// constructor.
public class Test<T> where T : TestBase, ITestArgument, new() { }

// Define a class that meets the constraints on the type
// parameter of class Test.
public class TestArgument : TestBase, ITestArgument
{
    public TestArgument() { }
}

public class Example
{
    // The following method displays information about a generic
    // type.
    private static void DisplayGenericType(Type t)
    {
```

```
        Console.WriteLine($"\r\n {t}");
        Console.WriteLine($"   Is this a generic type? {t.IsGenericType}");
        Console.WriteLine($"   Is this a generic type definition?
{t.IsGenericTypeDefinition}");

        // Get the generic type parameters or type arguments.
        Type[] typeParameters = t.GetGenericArguments();

        Console.WriteLine($"   List {typeParameters.Length} type arguments:");
        foreach (Type tParam in typeParameters)
        {
            if (tParam.IsGenericParameter)
            {
                DisplayGenericParameter(tParam);
            }
            else
            {
                Console.WriteLine($"      Type argument: {tParam}");
            }
        }
    }

    // Displays information about a generic type parameter.
    private static void DisplayGenericParameter(Type tp)
    {
        Console.WriteLine($"      Type parameter: {tp.Name} position
{tp.GenericParameterPosition}");

        foreach (Type iConstraint in tp.GetGenericParameterConstraints())
        {
            if (iConstraint.IsInterface)
            {
                Console.WriteLine($"         Interface constraint:
{iConstraint}");
            }
        }

        Console.WriteLine($"         Base type constraint: {tp.BaseType ??
tp.BaseType: None}");

        GenericParameterAttributes sConstraints =
            tp.GenericParameterAttributes &
            GenericParameterAttributes.SpecialConstraintMask;

        if (sConstraints == GenericParameterAttributes.None)
        {
            Console.WriteLine("         No special constraints.");
        }
        else
        {
            if (GenericParameterAttributes.None != (sConstraints &
                GenericParameterAttributes.DefaultConstructorConstraint))
            {
                Console.WriteLine("         Must have a parameterless
constructor.");
```

```
            }
            if (GenericParameterAttributes.None != (sConstraints &
                GenericParameterAttributes.ReferenceTypeConstraint))
            {
                Console.WriteLine("         Must be a reference type.");
            }
            if (GenericParameterAttributes.None != (sConstraints &
                GenericParameterAttributes.NotNullableValueTypeConstraint))
            {
                Console.WriteLine("         Must be a non-nullable value type.");
            }
        }
    }

    public static void Main()
    {
        // Two ways to get a Type object that represents the generic
        // type definition of the Dictionary class.

        // Use the typeof operator to create the generic type
        // definition directly. To specify the generic type definition,
        // omit the type arguments but retain the comma that separates
        // them.
        Type d1 = typeof(Dictionary<,>);

        // You can also obtain the generic type definition from a
        // constructed class. In this case, the constructed class
        // is a dictionary of Example objects, with String keys.
        Dictionary<string, Example> d2 = [];
        // Get a Type object that represents the constructed type,
        // and from that get the generic type definition. The
        // variables d1 and d4 contain the same type.
        Type d3 = d2.GetType();
        Type d4 = d3.GetGenericTypeDefinition();

        // Display information for the generic type definition, and
        // for the constructed type Dictionary<String, Example>.
        DisplayGenericType(d1);
        DisplayGenericType(d2.GetType());

        // Construct an array of type arguments to substitute for
        // the type parameters of the generic Dictionary class.
        // The array must contain the correct number of types, in
        // the same order that they appear in the type parameter
        // list of Dictionary. The key (first type parameter)
        // is of type string, and the type to be contained in the
        // dictionary is Example.
        Type[] typeArgs = [typeof(string), typeof(Example)];

        // Construct the type Dictionary<String, Example>.
        Type constructed = d1.MakeGenericType(typeArgs);

        DisplayGenericType(constructed);
        _ = Activator.CreateInstance(constructed);
```

```
        Console.WriteLine("\r\nCompare types obtained by different methods:");
        Console.WriteLine($"   Are the constructed types equal? {d2.GetType() ==
constructed}");
        Console.WriteLine($"   Are the generic definitions equal? {d1 ==
constructed.GetGenericTypeDefinition()}");

        // Demonstrate the DisplayGenericType and
        // DisplayGenericParameter methods with the Test class
        // defined above. This shows base, interface, and special
        // constraints.
        DisplayGenericType(typeof(Test<>));
    }
}
```

참고하십시오

- Type
- MethodInfo
- 리플렉션 및 제네릭 형식
- 형식 정보 보기
- 제네릭

방법: 리플렉션을 사용하여 형식 및 멤버 정보 가져오기

아티클 • 2025. 05. 01.

네임스페이스에는 System.Reflection 형식 및 해당 멤버에 대한 정보를 가져오는 여러 메서드가 포함되어 있습니다. 이 문서에서는 이러한 방법 Type.GetMembers중 하나를 보여 줍니다. 자세한 내용은 리플렉션 개요를 참조하세요.

예시

다음 예제에서는 리플렉션을 사용하여 형식 및 멤버 정보를 가져옵니다.

C#

```
using System;
using System.Reflection;

class Asminfo1
{
    public static void Main()
    {
        Console.WriteLine ("\nReflection.MemberInfo");

        // Get the Type and MemberInfo.
        // Insert the fully qualified class name inside the quotation marks in the
        // following statement.
        Type MyType = Type.GetType("System.IO.BinaryReader");
        MemberInfo[] Mymemberinfoarray = MyType.GetMembers(BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Static |
            BindingFlags.Instance | BindingFlags.DeclaredOnly);

        // Get and display the DeclaringType method.
        Console.Write($"\nThere are {Mymemberinfoarray.Length} documentable
members in ");
        Console.Write($"{MyType.FullName}.");

        foreach (MemberInfo Mymemberinfo in Mymemberinfoarray)
        {
            Console.Write("\n" + Mymemberinfo.Name);
        }
    }
}
```

참고하십시오

- 리플렉션

방법: 리플렉션을 사용하여 대리자 연결

아티클 • 2025. 05. 04.

리플렉션을 사용하여 어셈블리를 로드하고 실행하는 경우 C# `+=` 연산자 또는 Visual Basic AddHandler 문 과 같은 언어 기능을 사용하여 이벤트를 연결할 수 없습니다. 다음 절차에서는 리플렉션을 통해 필요한 모든 형식을 가져오고 리플렉션 내보내기를 사용하여 동적 메서드를 만들어 이벤트에 연결하는 방법을 보여 줍니다.

> ⓘ **참고**
>
> 이벤트 처리 대리자를 추가하는 다른 방법은 AddEventHandler 클래스의 EventInfo 메서드에 대한 코드 예제를 참조하세요.

리플렉션을 사용하여 대리자를 연결하려면

1. 이벤트를 발생시키는 형식이 포함된 어셈블리를 로드합니다. 어셈블리는 일반적으로 Assembly.Load 메서드로 로드됩니다. 이 예제를 단순하게 유지하기 위해 현재 어셈블리에서 파생된 양식이 사용되므로 메서드를 GetExecutingAssembly 사용하여 현재 어셈블리를 로드합니다.

   C#

   ```
   Assembly assem = typeof(Example).Assembly;
   ```

2. 형식을 Type 나타내는 개체를 가져와서 형식의 인스턴스를 만듭니다. CreateInstance(Type) 이 메서드는 양식에 매개 변수가 없는 생성자가 있으므로 다음 코드에서 사용됩니다. 만드는 형식에 매개 변수가 없는 생성자가 없는 경우 사용할 수 있는 메서드의 CreateInstance 다른 오버로드가 몇 가지 있습니다. 새 인스턴스는 어셈블리에 대해 알려진 것이 없다는 허구의 유지 관리를 위해 형식 Object 으로 저장됩니다. 리플렉션을 사용하면 이름을 미리 알지 않고도 어셈블리에서 유형을 가져올 수 있습니다.

   C#

   ```
   Type tExForm = assem.GetType("ExampleForm");
   Object exFormAsObj = Activator.CreateInstance(tExForm);
   ```

3. EventInfo 이벤트를 나타내는 개체를 가져오고 속성을 사용하여 EventHandlerType 이벤트를 처리하는 데 사용되는 대리자의 형식을 가져옵니다. 다음 코드 EventInfo 에서는 이벤트에 대한 Click 값을 가져옵니다.

   C#

```
EventInfo evClick = tExForm.GetEvent("Click");
Type tDelegate = evClick.EventHandlerType;
```

4. MethodInfo 이벤트를 처리하는 메서드를 나타내는 개체를 가져옵니다. 이 문서의 뒷부분에 있는 예제 섹션의 전체 프로그램 코드에는 이벤트를 처리하는 EventHandler 대리자의 Click 서명과 일치하는 메서드가 포함되어 있지만 런타임에 동적 메서드를 생성할 수도 있습니다. 자세한 내용은 동적 메서드를 사용하여 런타임에 이벤트 처리기를 생성하기 위한 관련 절차를 참조하세요.

```
C#

MethodInfo miHandler =
    typeof(Example).GetMethod("LuckyHandler",
        BindingFlags.NonPublic | BindingFlags.Instance);
```

5. 메서드를 사용하여 대리자의 인스턴스를 만듭니다 CreateDelegate . 이 메서드는 정적(`Shared` Visual Basic)이므로 대리자 형식을 제공해야 합니다. CreateDelegate를 사용하는 MethodInfo의 오버로드를 사용하는 것이 좋습니다.

```
C#

Delegate d = Delegate.CreateDelegate(tDelegate, this, miHandler);
```

6. 접근자 메서드를 `add` 가져와서 호출하여 이벤트를 연결합니다. 모든 이벤트에는 `add` 접근자와 `remove` 접근자가 있으며, 이는 높은 수준의 언어 구문에 의해 숨겨져 있습니다. 예를 들어 C#에서는 연산자를 `+=` 사용하여 이벤트를 연결하고 Visual Basic은 AddHandler 문을 사용합니다. 다음 코드는 `add`의 접근자를 얻어 Click 이벤트에서 대리자 인스턴스를 사용하여 런타임에 바인딩된 방식으로 호출합니다. 인수는 배열로 전달되어야 합니다.

```
C#

MethodInfo addHandler = evClick.GetAddMethod();
Object[] addHandlerArgs = { d };
addHandler.Invoke(exFormAsObj, addHandlerArgs);
```

7. 이벤트를 테스트합니다. 다음 코드는 코드 예제에 정의된 양식을 보여줍니다. 폼을 클릭하면 이벤트 처리기가 호출됩니다.

```
C#

Application.Run((Form) exFormAsObj);
```

동적 메서드를 사용하여 런타임에 이벤트 처리기 생성

1. 런타임에 간단한 동적 메서드와 리플렉션 내보내기를 사용하여 이벤트 처리기 메서드를 생성할 수 있습니다. 이벤트 처리기를 생성하려면 대리자의 반환 형식 및 매개 변수 형식이 필요합니다. 대리자의 `Invoke` 메서드를 검사하여 가져올 수 있습니다. 다음 코드는 `GetDelegateReturnType` 및 `GetDelegateParameterTypes` 메서드를 사용하여 이 정보를 얻습니다. 이러한 메서드에 대한 코드는 이 문서의 뒷부분에 있는 예제 섹션에서 찾을 수 있습니다.

   DynamicMethod에는 이름을 지정할 필요가 없으므로 빈 문자열을 사용할 수 있습니다. 다음 코드에서 마지막 인수는 동적 메서드를 현재 형식과 연결하여 클래스의 `Example` 모든 public 및 private 멤버에 대한 대리자 액세스 권한을 부여합니다.

   C#

   ```
   Type returnType = GetDelegateReturnType(tDelegate);
   if (returnType != typeof(void))
       throw new ArgumentException("Delegate has a return type.", nameof(d));

   DynamicMethod handler =
       new DynamicMethod("",
                         null,
                         GetDelegateParameterTypes(tDelegate),
                         typeof(Example));
   ```

2. 메서드 본문을 생성합니다. 이 메서드는 문자열을 로드하고, 문자열을 사용하는 메서드의 MessageBox.Show 오버로드를 호출하고, 반환 값을 스택에서 팝하고(처리기에 반환 형식이 없기 때문에) 반환합니다. 동적 메서드를 내보내는 방법에 대한 자세한 내용은 방법: 동적 메서드 정의 및 실행을 참조하세요.

   C#

   ```
   ILGenerator ilgen = handler.GetILGenerator();

   Type[] showParameters = { typeof(String) };
   MethodInfo simpleShow =
       typeof(MessageBox).GetMethod("Show", showParameters);

   ilgen.Emit(OpCodes.Ldstr,
       "This event handler was constructed at run time.");
   ilgen.Emit(OpCodes.Call, simpleShow);
   ilgen.Emit(OpCodes.Pop);
   ilgen.Emit(OpCodes.Ret);
   ```

3. CreateDelegate 메서드를 호출하여 동적 메서드를 완료합니다. 접근자를 `add` 사용하여 이벤트의 호출 목록에 대리자를 추가합니다.

```
C#

Delegate dEmitted = handler.CreateDelegate(tDelegate);
addHandler.Invoke(exFormAsObj, new Object[] { dEmitted });
```

4. 이벤트를 테스트합니다. 다음 코드는 코드 예제에 정의된 양식을 로드합니다. 양식을 클릭하면 미리 정의된 이벤트 처리기와 내보낸 이벤트 처리기가 모두 호출됩니다.

```
C#

Application.Run((Form) exFormAsObj);
```

예시

다음 코드 예제에서는 리플렉션을 사용하여 이벤트에 기존 메서드를 연결하는 방법과 런타임에 DynamicMethod 클래스를 사용하여 메서드를 생성하고 이벤트에 연결하는 방법을 보여줍니다.

```
C#

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Windows.Forms;

class ExampleForm : Form
{
    public ExampleForm() : base()
    {
        this.Text = "Click me";
    }
}

class Example
{
    public static void Main()
    {
        Example ex = new Example();
        ex.HookUpDelegate();
    }

    private void HookUpDelegate()
    {
        // Load an assembly, for example using the Assembly.Load
        // method. In this case, the executing assembly is loaded, to
        // keep the demonstration simple.
```

```
        //
        Assembly assem = typeof(Example).Assembly;

        // Get the type that is to be loaded, and create an instance
        // of it. Activator.CreateInstance has other overloads, if
        // the type lacks a default constructor. The new instance
        // is stored as type Object, to maintain the fiction that
        // nothing is known about the assembly. (Note that you can
        // get the types in an assembly without knowing their names
        // in advance.)
        //
        Type tExForm = assem.GetType("ExampleForm");
        Object exFormAsObj = Activator.CreateInstance(tExForm);

        // Get an EventInfo representing the Click event, and get the
        // type of delegate that handles the event.
        //
        EventInfo evClick = tExForm.GetEvent("Click");
        Type tDelegate = evClick.EventHandlerType;

        // If you already have a method with the correct signature,
        // you can simply get a MethodInfo for it.
        //
        MethodInfo miHandler =
            typeof(Example).GetMethod("LuckyHandler",
                BindingFlags.NonPublic | BindingFlags.Instance);

        // Create an instance of the delegate. Using the overloads
        // of CreateDelegate that take MethodInfo is recommended.
        //
        Delegate d = Delegate.CreateDelegate(tDelegate, this, miHandler);

        // Get the "add" accessor of the event and invoke it late-
        // bound, passing in the delegate instance. This is equivalent
        // to using the += operator in C#, or AddHandler in Visual
        // Basic. The instance on which the "add" accessor is invoked
        // is the form; the arguments must be passed as an array.
        //
        MethodInfo addHandler = evClick.GetAddMethod();
        Object[] addHandlerArgs = { d };
        addHandler.Invoke(exFormAsObj, addHandlerArgs);

        // Event handler methods can also be generated at run time,
        // using lightweight dynamic methods and Reflection.Emit.
        // To construct an event handler, you need the return type
        // and parameter types of the delegate. These can be obtained
        // by examining the delegate's Invoke method.
        //
        // It is not necessary to name dynamic methods, so the empty
        // string can be used. The last argument associates the
        // dynamic method with the current type, giving the delegate
        // access to all the public and private members of Example,
        // as if it were an instance method.
        //
        Type returnType = GetDelegateReturnType(tDelegate);
```

```
        if (returnType != typeof(void))
            throw new ArgumentException("Delegate has a return type.", nameof(d));

        DynamicMethod handler =
            new DynamicMethod("",
                              null,
                              GetDelegateParameterTypes(tDelegate),
                              typeof(Example));

        // Generate a method body. This method loads a string, calls
        // the Show method overload that takes a string, pops the
        // return value off the stack (because the handler has no
        // return type), and returns.
        //
        ILGenerator ilgen = handler.GetILGenerator();

        Type[] showParameters = { typeof(String) };
        MethodInfo simpleShow =
            typeof(MessageBox).GetMethod("Show", showParameters);

        ilgen.Emit(OpCodes.Ldstr,
            "This event handler was constructed at run time.");
        ilgen.Emit(OpCodes.Call, simpleShow);
        ilgen.Emit(OpCodes.Pop);
        ilgen.Emit(OpCodes.Ret);

        // Complete the dynamic method by calling its CreateDelegate
        // method. Use the "add" accessor to add the delegate to
        // the invocation list for the event.
        //
        Delegate dEmitted = handler.CreateDelegate(tDelegate);
        addHandler.Invoke(exFormAsObj, new Object[] { dEmitted });

        // Show the form. Clicking on the form causes the two
        // delegates to be invoked.
        //
        Application.Run((Form) exFormAsObj);
    }

    private void LuckyHandler(Object sender, EventArgs e)
    {
        MessageBox.Show("This event handler just happened to be lying around.");
    }

    private Type[] GetDelegateParameterTypes(Type d)
    {
        if (d.BaseType != typeof(MulticastDelegate))
            throw new ArgumentException("Not a delegate.", nameof(d));

        MethodInfo invoke = d.GetMethod("Invoke");
        if (invoke == null)
            throw new ArgumentException("Not a delegate.", nameof(d));

        ParameterInfo[] parameters = invoke.GetParameters();
        Type[] typeParameters = new Type[parameters.Length];
```

```
        for (int i = 0; i < parameters.Length; i++)
        {
            typeParameters[i] = parameters[i].ParameterType;
        }
        return typeParameters;
    }

    private Type GetDelegateReturnType(Type d)
    {
        if (d.BaseType != typeof(MulticastDelegate))
            throw new ArgumentException("Not a delegate.", nameof(d));

        MethodInfo invoke = d.GetMethod("Invoke");
        if (invoke == null)
            throw new ArgumentException("Not a delegate.", nameof(d));

        return invoke.ReturnType;
    }
}
```

참고하십시오

- Assembly.Load
- DynamicMethod
- CreateInstance
- CreateDelegate
- 방법: 동적 메서드 정의 및 실행
- 리플렉션

동적 메서드 및 어셈블리 내보내기

이 섹션에서는 컴파일러 또는 도구가 런타임에 System.Reflection.Emit 메타데이터 및 CIL(공용 중간 언어)을 내보내고 필요에 따라 디스크에 PE(이식 가능한 실행 파일) 파일을 생성할 수 있도록 하는 네임스페이스의 관리되는 형식 집합에 대해 설명합니다. 스크립트 엔진 및 컴파일러는 이 네임스페이스의 기본 사용자입니다. 이 섹션에서는 네임스페이스에서 제공하는 System.Reflection.Emit 기능을 *리플렉션 내보내기*라고 합니다.

리플렉션 내보내기 기능은 다음과 같습니다.

- 런타임에 DynamicMethod 클래스를 사용하여 경량화된 전역 메서드를 정의하고, 대리자를 사용하여 실행합니다.
- 런타임에 어셈블리를 정의한 다음, 해당 어셈블리를 실행하고 디스크에 저장합니다.
- 런타임에 어셈블리를 정의하고, 실행한 다음, 언로드하여 가비지 수집이 리소스를 회수할 수 있도록 허용합니다.
- 런타임에 새 어셈블리에서 모듈을 정의한 다음 실행 및/또는 디스크에 저장합니다.
- 런타임에 모듈에서 형식을 정의하고, 이러한 형식의 인스턴스를 만들고, 해당 메서드를 호출합니다.
- 디버거 및 코드 프로파일러와 같은 도구에서 사용할 수 있는 정의된 모듈에 대한 기호 정보를 정의합니다.

네임스페이스의 관리되는 형식 System.Reflection.Emit 외에도 메타데이터 인터페이스 (.NET Framework) 및 메타데이터 인터페이스(.NET)에 설명된 관리되지 않는 메타데이터 인터페이스가 있습니다. 관리되는 리플렉션 내보내기는 관리되지 않는 메타데이터 인터페이스보다 더 강력한 의미 체계 오류 검사 및 더 높은 수준의 메타데이터 추상화 기능을 제공합니다.

메타데이터 및 CIL을 사용하는 데 유용한 또 다른 리소스는 CLI(공용 언어 인프라) 설명서, 특히 "파티션 II: 메타데이터 정의 및 의미 체계" 및 "파티션 III: CIL 명령 집합"입니다. 이 설명서는 Ecma 웹 사이트에서 온라인으로 사용할 수 있습니다.

Reference

OpCodes
메서드 본문을 빌드하는 데 사용할 수 있는 CIL 명령 코드를 카탈로그로 작성합니다.

System.Reflection.Emit
동적 메서드, 어셈블리 및 형식을 내보내는 데 사용되는 관리되는 클래스를 포함합니다.

Type
Type 관리되는 리플렉션 및 리플렉션 내보내기의 형식을 나타내며 이러한 기술을 사용하는 데 핵심적인 클래스를 설명합니다.

System.Reflection

메타데이터 및 관리 코드를 탐색하는 데 사용되는 관리되는 클래스를 포함합니다.

Last updated on 2025. 12. 03.

방법: 동적 메서드 정의 및 실행

아티클 • 2025. 04. 10.

다음 절차에서는 간단한 동적 메서드와 클래스 인스턴스에 바인딩된 동적 메서드를 정의하고 실행하는 방법을 보여 줍니다. 동적 메서드에 대한 자세한 내용은 클래스를 참조하세요 DynamicMethod .

1. 메서드를 실행할 대리자 형식을 선언합니다. 제네릭 대리자를 사용하여 선언해야 하는 대리자 형식의 수를 최소화하는 것이 좋습니다. 다음 코드는 메서드에 사용할 수 있는 두 개의 대리자 형식을 `SquareIt` 선언하고 그 중 하나는 제네릭입니다.

   C#

   ```
   private delegate long SquareItInvoker(int input);

   private delegate TReturn OneParameter<TReturn, TParameter0>
       (TParameter0 p0);
   ```

2. 동적 메서드의 매개 변수 형식을 지정하는 배열을 만듭니다. 이 예제에서 유일한 매개 변수는 `int` (`Integer` Visual Basic의 경우) 배열에 하나의 요소만 있습니다.

   C#

   ```
   Type[] methodArgs = {typeof(int)};
   ```

3. DynamicMethod을 만듭니다. 이 예제에서는 메서드의 이름이 지정 `SquareIt`됩니다.

   > **참고**
   >
   > 동적 메서드 이름을 지정할 필요는 없으며 이름으로 호출할 수 없습니다. 여러 동적 메서드의 이름은 같을 수 있습니다. 그러나 이름은 호출 스택에 표시되며 디버깅에 유용할 수 있습니다.

   반환 값의 형식은 .로 `long`지정됩니다. 메서드는 예제 코드를 포함 하는 클래스를 `Example` 포함 하는 모듈과 연결 됩니다. 로드된 모듈을 지정할 수 있습니다. 동적 메서드는 모듈 수준 `static` 메서드처럼 작동합니다(`Shared` Visual Basic의 경우).

   C#

   ```
   DynamicMethod squareIt = new DynamicMethod(
       "SquareIt",
       typeof(long),
   ```

```
    methodArgs,
    typeof(Example).Module);
```

4. 메서드 본문을 내보낸다. 이 예제에서는 개체를 ILGenerator 사용하여 CIL(공용 중간 언어)을 내보낸다. 또는 개체를 DynamicILInfo 관리되지 않는 코드 생성기와 함께 사용하여 메서드 본문을 내보낼 수 있습니다 DynamicMethod.

   이 예제의 CIL은 인수인 인수를 `int` 스택에 로드하고, 인수를 `long` 스택으로 변환하고, 복제 `long` 하고, 두 숫자를 곱합니다. 이렇게 하면 제곱 결과가 스택에 남게 되며 모든 메서드가 반환됩니다.

   C#

```
ILGenerator il = squareIt.GetILGenerator();
il.Emit(OpCodes.Ldarg_0);
il.Emit(OpCodes.Conv_I8);
il.Emit(OpCodes.Dup);
il.Emit(OpCodes.Mul);
il.Emit(OpCodes.Ret);
```

5. 메서드를 호출하여 동적 메서드를 나타내는 대리자의 인스턴스(1단계에서 선언됨)를 CreateDelegate 만듭니다. 대리자를 만들면 메서드가 완료되고 메서드를 변경하려는 추가 시도(예: CIL 추가)는 무시됩니다. 다음 코드는 제네릭 대리자를 사용하여 대리자를 만들고 호출합니다.

   C#

```
OneParameter<long, int> invokeSquareIt =
    (OneParameter<long, int>)
    squareIt.CreateDelegate(typeof(OneParameter<long, int>));

Console.WriteLine($"123456789 squared = {invokeSquareIt(123456789)}");
```

6. 메서드를 실행할 대리자 형식을 선언합니다. 제네릭 대리자를 사용하여 선언해야 하는 대리자 형식의 수를 최소화하는 것이 좋습니다. 다음 코드는 하나의 매개 변수와 반환 값을 사용하여 메서드를 실행하는 데 사용할 수 있는 제네릭 대리자 형식 또는 대리자가 개체에 바인딩된 경우 두 개의 매개 변수와 반환 값이 있는 메서드를 선언합니다.

   C#

```
private delegate TReturn OneParameter<TReturn, TParameter0>
    (TParameter0 p0);
```

7. 동적 메서드의 매개 변수 형식을 지정하는 배열을 만듭니다. 메서드를 나타내는 대리자가 개체에 바인딩되는 경우 첫 번째 매개 변수는 대리자가 바인딩된 형식과 일치해야 합니다.

이 예제에는 형식과 형식 `Example` `int` (`Integer` Visual Basic의 경우)의 두 매개 변수가 있습니다.

```
C#

Type[] methodArgs2 = { typeof(Example), typeof(int) };
```

8. DynamicMethod을 만듭니다. 이 예제에서는 메서드에 이름이 없습니다. 반환 값의 형식은 Visual Basic에서와 `Integer` 같이 `int` 지정됩니다. 메서드는 클래스의 프라이빗 및 보호된 멤버에 액세스할 수 있습니다 `Example` .

```
C#

DynamicMethod multiplyHidden = new DynamicMethod(
    "",
    typeof(int),
    methodArgs2,
    typeof(Example));
```

9. 메서드 본문을 내보낸다. 이 예제에서는 개체를 ILGenerator 사용하여 CIL(공용 중간 언어)을 내보낸다. 또는 개체를 DynamicILInfo 관리되지 않는 코드 생성기와 함께 사용하여 메서드 본문을 내보낼 수 있습니다 DynamicMethod.

   이 예제의 CIL은 클래스의 인스턴스인 첫 번째 인수를 `Example` 로드하고 이를 사용하여 형식 `int`의 프라이빗 인스턴스 필드 값을 로드합니다. 두 번째 인수가 로드되고 두 숫자가 곱됩니다. 결과가 보다 `int`크면 값이 잘리고 가장 중요한 비트가 삭제됩니다. 메서드는 스택의 반환 값을 사용하여 반환합니다.

```
C#

ILGenerator ilMH = multiplyHidden.GetILGenerator();
ilMH.Emit(OpCodes.Ldarg_0);

FieldInfo testInfo = typeof(Example).GetField("test",
    BindingFlags.NonPublic | BindingFlags.Instance);

ilMH.Emit(OpCodes.Ldfld, testInfo);
ilMH.Emit(OpCodes.Ldarg_1);
ilMH.Emit(OpCodes.Mul);
ilMH.Emit(OpCodes.Ret);
```

10. 메서드 오버로드를 호출 CreateDelegate(Type, Object) 하여 동적 메서드를 나타내는 대리자의 인스턴스(1단계에서 선언됨)를 만듭니다. 대리자를 만들면 메서드가 완료되고 메서드를 변경하려는 추가 시도(예: CIL 추가)는 무시됩니다.

> ⓘ **참고**
>
> 메서드를 CreateDelegate 여러 번 호출하여 대상 형식의 다른 인스턴스에 바인딩된 대리자를 만들 수 있습니다.

다음 코드는 프라이빗 테스트 필드가 42로 설정된 클래스의 `Example` 새 인스턴스에 메서드를 바인딩합니다. 즉, 대리자가 호출될 때마다 인스턴스가 메서드의 `Example` 첫 번째 매개 변수에 전달됩니다.

대리 `OneParameter` 자는 메서드의 첫 번째 매개 변수가 항상 인스턴스 `Example`를 수신하기 때문에 사용됩니다. 대리자가 호출되면 두 번째 매개 변수만 필요합니다.

C#

```
OneParameter<int, int> invoke = (OneParameter<int, int>)
    multiplyHidden.CreateDelegate(
        typeof(OneParameter<int, int>),
        new Example(42)
    );

Console.WriteLine($"3 * test = {invoke(3)}");
```

예시

다음 코드 예제에서는 간단한 동적 메서드 및 클래스의 인스턴스에 바인딩된 동적 메서드를 보여 줍니다.

단순 동적 메서드는 하나의 인수인 32비트 정수로, 해당 정수의 64비트 제곱을 반환합니다. 제네릭 대리자는 메서드를 호출하는 데 사용됩니다.

두 번째 동적 메서드에는 형식 `Example` 및 형식 `int`의 두 매개 변수가 있으며, Visual Basic의 경우에는 `Integer` 형식입니다. 동적 메서드가 생성되면, 하나의 `int` 형식 인수를 가지는 제네릭 대리자를 사용하여 `Example`의 인스턴스에 바인딩됩니다. 메서드의 첫 번째 매개 변수는 항상 바인딩된 인스턴스`Example`를 받기 때문에 대리자는 형식 `Example` 의 인수가 없습니다. 대리자가 호출될 때, `int` 인수만 전달됩니다. 이 동적 메서드는 클래스의 `Example` 프라이빗 필드에 액세스하고 프라이빗 필드와 인수의 곱을 `int` 반환합니다.

코드 예제에서는 메서드를 실행하는 데 사용할 수 있는 대리자를 정의합니다.

C#

```
using System;
using System.Reflection;
```

```
using System.Reflection.Emit;

public class Example
{
    // The following constructor and private field are used to
    // demonstrate a method bound to an object.
    private int test;
    public Example(int test) { this.test = test; }

    // Declare delegates that can be used to execute the completed
    // SquareIt dynamic method. The OneParameter delegate can be
    // used to execute any method with one parameter and a return
    // value, or a method with two parameters and a return value
    // if the delegate is bound to an object.
    //
    private delegate long SquareItInvoker(int input);

    private delegate TReturn OneParameter<TReturn, TParameter0>
        (TParameter0 p0);

    public static void Main()
    {
        // Example 1: A simple dynamic method.
        //
        // Create an array that specifies the parameter types for the
        // dynamic method. In this example the only parameter is an
        // int, so the array has only one element.
        //
        Type[] methodArgs = {typeof(int)};

        // Create a DynamicMethod. In this example the method is
        // named SquareIt. It is not necessary to give dynamic
        // methods names. They cannot be invoked by name, and two
        // dynamic methods can have the same name. However, the
        // name appears in calls stacks and can be useful for
        // debugging.
        //
        // In this example the return type of the dynamic method
        // is long. The method is associated with the module that
        // contains the Example class. Any loaded module could be
        // specified. The dynamic method is like a module-level
        // static method.
        //
        DynamicMethod squareIt = new DynamicMethod(
            "SquareIt",
            typeof(long),
            methodArgs,
            typeof(Example).Module);

        // Emit the method body. In this example ILGenerator is used
        // to emit the MSIL. DynamicMethod has an associated type
        // DynamicILInfo that can be used in conjunction with
        // unmanaged code generators.
        //
        // The MSIL loads the argument, which is an int, onto the
```

```
        // stack, converts the int to a long, duplicates the top
        // item on the stack, and multiplies the top two items on the
        // stack. This leaves the squared number on the stack, and
        // all the method has to do is return.
        //
        ILGenerator il = squareIt.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Conv_I8);
        il.Emit(OpCodes.Dup);
        il.Emit(OpCodes.Mul);
        il.Emit(OpCodes.Ret);

        // Create a delegate that represents the dynamic method.
        // Creating the delegate completes the method, and any further
        // attempts to change the method (for example, by adding more
        // MSIL) are ignored. The following code uses a generic
        // delegate that can produce delegate types matching any
        // single-parameter method that has a return type.
        //
        OneParameter<long, int> invokeSquareIt =
            (OneParameter<long, int>)
            squareIt.CreateDelegate(typeof(OneParameter<long, int>));

        Console.WriteLine($"123456789 squared = {invokeSquareIt(123456789)}");

        // Example 2: A dynamic method bound to an instance.
        //
        // Create an array that specifies the parameter types for a
        // dynamic method. If the delegate representing the method
        // is to be bound to an object, the first parameter must
        // match the type the delegate is bound to. In the following
        // code the bound instance is of the Example class.
        //
        Type[] methodArgs2 = { typeof(Example), typeof(int) };

        // Create a DynamicMethod. In this example the method has no
        // name. The return type of the method is int. The method
        // has access to the protected and private data of the
        // Example class.
        //
        DynamicMethod multiplyHidden = new DynamicMethod(
            "",
            typeof(int),
            methodArgs2,
            typeof(Example));

        // Emit the method body. In this example ILGenerator is used
        // to emit the MSIL. DynamicMethod has an associated type
        // DynamicILInfo that can be used in conjunction with
        // unmanaged code generators.
        //
        // The MSIL loads the first argument, which is an instance of
        // the Example class, and uses it to load the value of a
        // private instance field of type int. The second argument is
        // loaded, and the two numbers are multiplied. If the result
```

```
        // is larger than int, the value is truncated and the most
        // significant bits are discarded. The method returns, with
        // the return value on the stack.
        //
        ILGenerator ilMH = multiplyHidden.GetILGenerator();
        ilMH.Emit(OpCodes.Ldarg_0);

        FieldInfo testInfo = typeof(Example).GetField("test",
            BindingFlags.NonPublic | BindingFlags.Instance);

        ilMH.Emit(OpCodes.Ldfld, testInfo);
        ilMH.Emit(OpCodes.Ldarg_1);
        ilMH.Emit(OpCodes.Mul);
        ilMH.Emit(OpCodes.Ret);

        // Create a delegate that represents the dynamic method.
        // Creating the delegate completes the method, and any further
        // attempts to change the method — for example, by adding more
        // MSIL — are ignored.
        //
        // The following code binds the method to a new instance
        // of the Example class whose private test field is set to 42.
        // That is, each time the delegate is invoked the instance of
        // Example is passed to the first parameter of the method.
        //
        // The delegate OneParameter is used, because the first
        // parameter of the method receives the instance of Example.
        // When the delegate is invoked, only the second parameter is
        // required.
        //
        OneParameter<int, int> invoke = (OneParameter<int, int>)
            multiplyHidden.CreateDelegate(
                typeof(OneParameter<int, int>),
                new Example(42)
            );

        Console.WriteLine($"3 * test = {invoke(3)}");
    }
}
/* This code example produces the following output:

123456789 squared = 15241578750190521
3 * test = 126
 */
```

참고하십시오

- DynamicMethod

방법: 리플렉션 내보내기를 사용하여 제네릭 형식 정의(.NET Framework)

> ⓘ Important
>
> 이 방법 문서에서는 최신 .NET에서 사용할 수 없는 .NET Framework 관련 API를 보여 줍니다. 최신 .NET에서 디스크에 동적 어셈블리를 저장하려면 형식을 PersistedAssemblyBuilder 사용합니다.

이 문서에서는 다음 방법을 보여줍니다.

- 두 개의 형식 매개 변수를 사용하여 간단한 제네릭 형식을 만듭니다.
- 클래스 제약 조건, 인터페이스 제약 조건 및 특수 제약 조건을 형식 매개 변수에 적용합니다.
- 클래스의 형식 매개 변수를 매개 변수 형식 및 반환 형식으로 사용하는 멤버를 만듭니다.

> ⓘ Important
>
> 메서드는 제네릭 형식에 속하고 해당 형식의 형식 매개 변수를 사용하기 때문에 제네릭이 아닙니다. 메서드는 고유한 형식 매개 변수 목록이 있는 경우에만 제네릭입니다. 제네릭 형식에 대한 대부분의 메서드는 이 예제와 같이 제네릭이 아닙니다. 제네릭 메서드를 내보내는 예제는 **방법: 리플렉션 내보내기를 사용하여 제네릭 메서드 정의**참조하세요.

제네릭 형식 정의

1. `GenericEmitExample1`동적 어셈블리를 정의합니다. 이 예제에서는 어셈블리가 실행되어 디스크에 저장되므로 AssemblyBuilderAccess.RunAndSave 지정됩니다.

   C#

   ```
   AppDomain myDomain = AppDomain.CurrentDomain;
   AssemblyName myAsmName = new AssemblyName("GenericEmitExample1");
   AssemblyBuilder myAssembly =
       myDomain.DefineDynamicAssembly(myAsmName,
           AssemblyBuilderAccess.RunAndSave);
   ```

2. 동적 모듈을 정의합니다. 어셈블리는 실행 가능한 모듈로 구성됩니다. 단일 모듈 어셈블리의 경우 모듈 이름은 어셈블리 이름과 같으며 파일 이름은 모듈 이름과 확장명입니다.

   C#

```
ModuleBuilder myModule =
    myAssembly.DefineDynamicModule(
        myAsmName.Name,
        $"{myAsmName.Name}.dll");
```

3. 클래스를 정의합니다. 이 예제에서 클래스의 이름은 `Sample`.

```
C#

TypeBuilder myType =
    myModule.DefineType("Sample", TypeAttributes.Public);
```

4. 매개 변수 이름이 포함된 문자열 배열을 `Sample` 메서드에 전달하여 TypeBuilder.DefineGenericParameters 제네릭 형식 매개 변수를 정의합니다. 이렇게 하면 클래스가 제네릭 형식으로 만들어집니다. 반환 값은 내보낸 코드에서 사용할 수 있는 형식 매개 변수를 나타내는 GenericTypeParameterBuilder 개체의 배열입니다.

   다음 코드에서 `Sample`이 `TFirst` 및 `TSecond`형식 매개 변수를 가진 제네릭 형식이 됩니다. 코드를 더 쉽게 읽을 수 있도록 각 GenericTypeParameterBuilder 형식 매개 변수와 이름이 같은 변수에 배치됩니다.

```
C#

string[] typeParamNames = { "TFirst", "TSecond" };
GenericTypeParameterBuilder[] typeParams =
    myType.DefineGenericParameters(typeParamNames);

GenericTypeParameterBuilder TFirst = typeParams[0];
GenericTypeParameterBuilder TSecond = typeParams[1];
```

5. 형식 매개 변수에 특수 제약 조건을 추가합니다. 이 예제에서 형식 매개 변수 `TFirst` 매개 변수가 없는 생성자가 있는 형식 및 참조 형식으로 제한됩니다.

```
C#

TFirst.SetGenericParameterAttributes(
    GenericParameterAttributes.DefaultConstructorConstraint |
    GenericParameterAttributes.ReferenceTypeConstraint);
```

6. 필요에 따라 형식 매개 변수에 클래스 및 인터페이스 제약 조건을 추가합니다. 이 예제에서 형식 매개 변수 `TFirst` 변수 Type포함된 `baseType` 개체가 나타내는 기본 클래스에서 파생되는 형식으로 제한되며 해당 형식이 `interfaceA` 및 `interfaceB`변수에 포함된 인터페이스를 구현합니다. 이러한 변수의 선언 및 할당에 대한 코드 예제를 참조하세요.

```
C#
```

```
TSecond.SetBaseTypeConstraint(baseType);
Type[] interfaceTypes = { interfaceA, interfaceB };
TSecond.SetInterfaceConstraints(interfaceTypes);
```

7. 필드를 정의합니다. 이 예제에서 필드의 형식은 형식 매개 변수 `TFirst`지정됩니다. GenericTypeParameterBuilder Type파생되므로 형식을 사용할 수 있는 모든 위치에서 제네릭 형식 매개 변수를 사용할 수 있습니다.

C#

```
_ = myType.DefineField("ExampleField", TFirst, FieldAttributes.Private);
```

8. 제네릭 형식의 형식 매개 변수를 사용하는 메서드를 정의합니다. 이러한 메서드는 고유한 형식 매개 변수 목록이 없는 한 제네릭이 아닙니다. 다음 코드는 `static` 배열을 사용하고 배열의 모든 요소가 포함된 `Shared`(Visual Basic에서는`TFirst`)를 반환하는 `List<TFirst>` 메서드(Visual Basic의`List(Of TFirst)`)를 정의합니다. 이 메서드를 정의하려면 제네릭 형식 정의 `List<TFirst>`MakeGenericType 호출하여 `List<T>` 형식을 만들어야 합니다. (`T` `typeof` 연산자(Visual Basic의`GetType`)를 사용하여 제네릭 형식 정의를 가져올 때 생략됩니다. 매개 변수 형식은 MakeArrayType 메서드를 사용하여 생성됩니다.

C#

```
Type listOf = typeof(List<>);
Type listOfTFirst = listOf.MakeGenericType(TFirst);
Type[] mParamTypes = { TFirst.MakeArrayType() };

MethodBuilder exMethod =
    myType.DefineMethod("ExampleMethod",
        MethodAttributes.Public | MethodAttributes.Static,
        listOfTFirst,
        mParamTypes);
```

9. 메서드 본문을 내보낸다. 메서드 본문은 입력 배열을 스택에 로드하고, `List<TFirst>` 사용하는 `IEnumerable<TFirst>` 생성자(입력 요소를 목록에 넣는 모든 작업을 수행)를 호출하고, 반환(스택에 새 List<T> 개체를 남겨 두는) 세 개의 opcode로 구성됩니다. 이 코드를 내보내는 데 있어 어려운 부분은 생성자를 가져오는 것입니다.

   GetConstructor 메서드는 GenericTypeParameterBuilder지원되지 않으므로 `List<TFirst>` 생성자를 직접 가져올 수 없습니다. 먼저 제네릭 형식 정의 `List<T>` 생성자를 가져와서 해당 `List<TFirst>`생성자로 변환하는 메서드를 호출해야 합니다.

   이 코드 예제에 사용되는 생성자는 `IEnumerable<T>`사용합니다. 그러나 주의할 점은 이는 IEnumerable<T> 제네릭 인터페이스의 일반 형식 정의가 아니라는 것입니다. 대신, `T`의 `List<T>` 형식 매개변수를 `T`의 `IEnumerable<T>` 형식 매개변수로 대체해야 합니다. 두 형식

모두 `T`형식 매개 변수를 가지고 있기 때문에 혼동을 하게 됩니다. 따라서 이 코드 예제에서는 이름 `TFirst` 및 `TSecond`.)를 사용합니다. 생성자 인수의 형식을 얻으려면 제네릭 형식 정의의 `IEnumerable<T>` 시작하고 MakeGenericType첫 번째 제네릭 형식 매개 변수를 사용하여 `List<T>` 호출합니다. 이 경우 하나의 인수만 사용하여 생성자 인수 목록을 배열로 전달해야 합니다.

> ⓘ **참고 항목**
>
> 제네릭 형식 정의는 C#에서 `IEnumerable<>` 연산자를 사용할 때 `typeof` 표현되거나 Visual Basic에서 `IEnumerable(Of )` 연산자를 사용할 때 `GetType`.

이제 제네릭 형식 정의에서 `List<T>` 호출하여 GetConstructor 생성자를 가져올 수 있습니다. 이 생성자를 해당 `List<TFirst>`생성자로 변환하려면 `List<TFirst>` 및 생성자를 `List<T>` 정적 TypeBuilder.GetConstructor(Type, ConstructorInfo) 메서드로 전달합니다.

C#

```
ILGenerator ilgen = exMethod.GetILGenerator();

Type ienumOf = typeof(IEnumerable<>);
Type TfromListOf = listOf.GetGenericArguments()[0];
Type ienumOfT = ienumOf.MakeGenericType(TfromListOf);
Type[] ctorArgs = { ienumOfT };

ConstructorInfo ctorPrep = listOf.GetConstructor(ctorArgs);
ConstructorInfo ctor =
    TypeBuilder.GetConstructor(listOfTFirst, ctorPrep);

ilgen.Emit(OpCodes.Ldarg_0);
ilgen.Emit(OpCodes.Newobj, ctor);
ilgen.Emit(OpCodes.Ret);
```

10. 형식을 만들고 파일을 저장합니다.

C#

```
Type finished = myType.CreateType();
myAssembly.Save(myAsmName.Name + ".dll");
```

11. 메서드를 호출합니다. `ExampleMethod` 제네릭은 아니지만 해당 형식이 제네릭이므로 호출할 수 있는 MethodInfo 얻으려면 `Sample`대한 형식 정의에서 생성된 형식을 만들어야 합니다. 생성된 형식은 참조 형식이고 기본 매개 변수가 없는 생성자를 가지므로 `Example` 제약 조건을 충족하는 `TFirst` 클래스와 `ExampleDerived`대한 제약 조건을 충족하는 `TSecond` 클래스를 사용합니다. (`ExampleDerived` 코드는 예제 코드 섹션에서 찾을 수 있습니다.) 이러

한 두 형식은 생성된 형식을 만들기 위해 MakeGenericType 전달됩니다. 그런 다음 MethodInfo 메서드를 사용하여 GetMethod 가져옵니다.

```
C#
Type[] typeArgs = { typeof(Example), typeof(ExampleDerived) };
Type constructed = finished.MakeGenericType(typeArgs);
MethodInfo mi = constructed.GetMethod("ExampleMethod");
```

12. 다음 코드는 `Example` 개체의 배열을 만들고, 호출할 메서드의 인수를 나타내는 Object 형식 배열에 배열을 배치하고, Invoke(Object, Object[]) 메서드에 전달합니다. 메서드가 Invoke때문에 `static` 메서드의 첫 번째 인수는 null 참조입니다.

```
C#
Example[] input = { new Example(), new Example() };
object[] arguments = { input };

List<Example> listX =
    (List<Example>)mi.Invoke(null, arguments);

Console.WriteLine($"\nThere are {listX.Count} elements in the List<Example>.");
```

예시

다음 코드 예제에서는 전체 프로그램을 보여줍니다. 기본 클래스 및 두 인터페이스와 함께 명명된 `Sample`클래스를 정의합니다. 이 프로그램은 `Sample`대한 두 개의 제네릭 형식 매개 변수를 정의하여 제네릭 형식으로 변환합니다. 형식 매개 변수만 제네릭 형식을 만듭니다. 프로그램에서는 형식 매개 변수의 정의 전후에 테스트 메시지를 표시하여 이를 보여 줄 수 있습니다.

`TSecond` 형식 매개 변수는 기본 클래스 및 인터페이스를 사용하여 클래스 및 인터페이스 제약 조건을 보여 주는 데 사용되며, 형식 매개 변수 `TFirst` 특수 제약 조건을 보여 주는 데 사용됩니다.

코드 예제에서는 필드 형식 및 메서드의 매개 변수 및 반환 형식에 대 한 클래스의 형식 매개 변수를 사용 하 여 필드와 메서드를 정의 합니다.

`Sample` 클래스를 만든 후 메서드가 호출됩니다.

이 프로그램에는 제네릭 형식에 대한 정보를 나열하는 메서드와 형식 매개 변수에 대한 특수 제약 조건을 나열하는 메서드가 포함됩니다. 이러한 메서드는 완성된 `Sample` 클래스에 대한 정보를 표시하는 데 사용됩니다.

프로그램은 완료된 모듈을 디스크에 `GenericEmitExample1.dll`저장하므로 Ildasm.exe(IL 디스어셈블러) 사용하여 열고 `Sample` 클래스에 대한 CIL을 검사할 수 있습니다.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Collections.Generic;

// Define a trivial base class and two trivial interfaces
// to use when demonstrating constraints.
//
public class ExampleBase { }

public interface IExampleA { }

public interface IExampleB { }

// Define a trivial type that can substitute for type parameter
// TSecond.
//
public class ExampleDerived : ExampleBase, IExampleA, IExampleB { }

public class Example
{
    public static void Main()
    {
        // Define a dynamic assembly to contain the sample type. The
        // assembly won't be run, only saved to disk, so
        // AssemblyBuilderAccess.Save is specified.
        //
        AppDomain myDomain = AppDomain.CurrentDomain;
        AssemblyName myAsmName = new AssemblyName("GenericEmitExample1");
        AssemblyBuilder myAssembly =
            myDomain.DefineDynamicAssembly(myAsmName,
                AssemblyBuilderAccess.RunAndSave);

        // An assembly is made up of executable modules. For a single-
        // module assembly, the module name and file name are the same
        // as the assembly name.
        //
        ModuleBuilder myModule =
            myAssembly.DefineDynamicModule(
                myAsmName.Name,
                $"{myAsmName.Name}.dll");

        // Get type objects for the base class trivial interfaces to
        // be used as constraints.
        //
        Type baseType = typeof(ExampleBase);
        Type interfaceA = typeof(IExampleA);
        Type interfaceB = typeof(IExampleB);
```

```
// Define the "Sample" type.
//
TypeBuilder myType =
    myModule.DefineType("Sample", TypeAttributes.Public);

Console.WriteLine($"Type 'Sample' is generic: {myType.IsGenericType}");

// Define type parameters for the type. Until you do this,
// the type is not generic, as the preceding and following
// WriteLine statements show. The type parameter names are
// specified as an array of strings. To make the code
// easier to read, each GenericTypeParameterBuilder is placed
// in a variable with the same name as the type parameter.
//
string[] typeParamNames = { "TFirst", "TSecond" };
GenericTypeParameterBuilder[] typeParams =
    myType.DefineGenericParameters(typeParamNames);

GenericTypeParameterBuilder TFirst = typeParams[0];
GenericTypeParameterBuilder TSecond = typeParams[1];

Console.WriteLine($"Type 'Sample' is generic: {myType.IsGenericType}");

// Apply constraints to the type parameters.
//
// A type that is substituted for the first parameter, TFirst,
// must be a reference type and must have a parameterless
// constructor.
TFirst.SetGenericParameterAttributes(
    GenericParameterAttributes.DefaultConstructorConstraint |
    GenericParameterAttributes.ReferenceTypeConstraint);

// A type that is substituted for the second type
// parameter must implement IExampleA and IExampleB, and
// inherit from the trivial test class ExampleBase. The
// interface constraints are specified as an array
// containing the interface types.
TSecond.SetBaseTypeConstraint(baseType);
Type[] interfaceTypes = { interfaceA, interfaceB };
TSecond.SetInterfaceConstraints(interfaceTypes);

// The following code adds a private field
// named ExampleField of type TFirst.
_ = myType.DefineField("ExampleField", TFirst, FieldAttributes.Private);

// Define a static method that takes an array of TFirst and
// returns a List<TFirst> containing all the elements of
// the array. To define this method, it's necessary to create
// the type List<TFirst> by calling MakeGenericType on the
// generic type definition, List<T>. (The T is omitted with
// the typeof operator when you get the generic type
// definition.) The parameter type is created by using the
// MakeArrayType method.
//
```

```
Type listOf = typeof(List<>);
Type listOfTFirst = listOf.MakeGenericType(TFirst);
Type[] mParamTypes = { TFirst.MakeArrayType() };

MethodBuilder exMethod =
    myType.DefineMethod("ExampleMethod",
        MethodAttributes.Public | MethodAttributes.Static,
        listOfTFirst,
        mParamTypes);

// Emit the method body.
// The method body consists of just three opcodes, to load
// the input array onto the execution stack, to call the
// List<TFirst> constructor that takes IEnumerable<TFirst>,
// which does all the work of putting the input elements into
// the list, and to return, leaving the list on the stack. The
// hard work is getting the constructor.
//
// The GetConstructor method is not supported on a
// GenericTypeParameterBuilder, so it's not possible to get
// the constructor of List<TFirst> directly. There are two
// steps: getting the constructor of List<T>, and then
// calling a method that converts it to the corresponding
// constructor of List<TFirst>.
//
// The constructor needed here is the one that takes an
// IEnumerable<T>. Note, however, that this is not the
// generic type definition of IEnumerable<T>; instead, the
// T from List<T> must be substituted for the T of
// IEnumerable<T>. (This seems confusing only because both
// types have type parameters named T. That is why this example
// uses the somewhat silly names TFirst and TSecond.) To get
// the type of the constructor argument, take the generic
// type definition IEnumerable<T> (expressed as
// IEnumerable<> when you use the typeof operator) and
// call MakeGenericType with the first generic type parameter
// of List<T>. The constructor argument list must be passed
// as an array, with just one argument in this case.
//
// Now it's possible to get the constructor of List<T>,
// using GetConstructor on the generic type definition. To get
// the constructor of List<TFirst>, pass List<TFirst> and
// the constructor from List<T> to the static
// TypeBuilder.GetConstructor method.
//
ILGenerator ilgen = exMethod.GetILGenerator();

Type ienumOf = typeof(IEnumerable<>);
Type TfromListOf = listOf.GetGenericArguments()[0];
Type ienumOfT = ienumOf.MakeGenericType(TfromListOf);
Type[] ctorArgs = { ienumOfT };

ConstructorInfo ctorPrep = listOf.GetConstructor(ctorArgs);
ConstructorInfo ctor =
    TypeBuilder.GetConstructor(listOfTFirst, ctorPrep);
```

```
        ilgen.Emit(OpCodes.Ldarg_0);
        ilgen.Emit(OpCodes.Newobj, ctor);
        ilgen.Emit(OpCodes.Ret);

        // Create the type and save the assembly.
        Type finished = myType.CreateType();
        myAssembly.Save(myAsmName.Name + ".dll");

        // Invoke the method.
        // ExampleMethod is not generic, but the type it belongs to is
        // generic, so in order to get a MethodInfo that can be invoked
        // it is necessary to create a constructed type. The Example
        // class satisfies the constraints on TFirst, because it is a
        // reference type and has a default constructor. In order to
        // have a class that satisfies the constraints on TSecond,
        // this code example defines the ExampleDerived type. These
        // two types are passed to MakeGenericMethod to create the
        // constructed type.
        //
        Type[] typeArgs = { typeof(Example), typeof(ExampleDerived) };
        Type constructed = finished.MakeGenericType(typeArgs);
        MethodInfo mi = constructed.GetMethod("ExampleMethod");

        // Create an array of Example objects, as input to the generic
        // method. This array must be passed as the only element of an
        // array of arguments. The first argument of Invoke is
        // null, because ExampleMethod is static. Display the count
        // on the resulting List<Example>.
        //
        Example[] input = { new Example(), new Example() };
        object[] arguments = { input };

        List<Example> listX =
            (List<Example>)mi.Invoke(null, arguments);

        Console.WriteLine($"\nThere are {listX.Count} elements in the
List<Example>.");

        DisplayGenericParameters(finished);
    }

    private static void DisplayGenericParameters(Type t)
    {
        if (!t.IsGenericType)
        {
            Console.WriteLine("Type '{0}' is not generic.");
            return;
        }
        if (!t.IsGenericTypeDefinition)
        {
            t = t.GetGenericTypeDefinition();
        }

        Type[] typeParameters = t.GetGenericArguments();
```

```
        Console.WriteLine($"\nListing {typeParameters.Length} type parameters for
type '{t}'.");

        foreach (Type tParam in typeParameters)
        {
            Console.WriteLine($"\nType parameter {tParam}:");

            foreach (Type c in tParam.GetGenericParameterConstraints())
            {
                if (c.IsInterface)
                {
                    Console.WriteLine($"    Interface constraint: {c}");
                }
                else
                {
                    Console.WriteLine($"    Base type constraint: {c}");
                }
            }

            ListConstraintAttributes(tParam);
        }
    }

    // List the constraint flags. The GenericParameterAttributes
    // enumeration contains two sets of attributes, variance and
    // constraints. For this example, only constraints are used.
    //
    private static void ListConstraintAttributes(Type t)
    {
        // Mask off the constraint flags.
        GenericParameterAttributes constraints =
            t.GenericParameterAttributes &
GenericParameterAttributes.SpecialConstraintMask;

        if ((constraints & GenericParameterAttributes.ReferenceTypeConstraint)
            != GenericParameterAttributes.None)
        {
            Console.WriteLine("    ReferenceTypeConstraint");
        }

        if ((constraints &
GenericParameterAttributes.NotNullableValueTypeConstraint)
            != GenericParameterAttributes.None)
        {
            Console.WriteLine("    NotNullableValueTypeConstraint");
        }

        if ((constraints & GenericParameterAttributes.DefaultConstructorConstraint)
            != GenericParameterAttributes.None)
        {
            Console.WriteLine("    DefaultConstructorConstraint");
        }
    }
}
```

```
/* This code example produces the following output:

Type 'Sample' is generic: False
Type 'Sample' is generic: True

There are 2 elements in the List<Example>.

Listing 2 type parameters for type 'Sample[TFirst,TSecond]'.

Type parameter TFirst:
    ReferenceTypeConstraint
    DefaultConstructorConstraint

Type parameter TSecond:
    Interface constraint: IExampleA
    Interface constraint: IExampleB
    Base type constraint: ExampleBase
 */
```

참고하십시오

- GenericTypeParameterBuilder
- 리플렉션 이미트 사용
- 리플렉션은 동적 어셈블리 시나리오 내보낸다.

Last updated on 2026. 01. 22.

방법: 리플렉션 내보내기를 사용하여 제네릭 메서드 정의

아티클 • 2025. 04. 03.

첫 번째 절차에서는 두 개의 형식 매개 변수를 사용하여 간단한 제네릭 메서드를 만드는 방법과 형식 매개 변수에 클래스 제약 조건, 인터페이스 제약 조건 및 특수 제약 조건을 적용하는 방법을 보여 줍니다.

두 번째 절차에서는 메서드 본문을 내보내는 방법과 제네릭 메서드의 형식 매개 변수를 사용하여 제네릭 형식의 인스턴스를 만들고 해당 메서드를 호출하는 방법을 보여 줍니다.

세 번째 프로시저는 제네릭 메서드를 호출하는 방법을 보여줍니다.

> ⓘ **중요**
>
> 메서드는 제네릭 형식에 속하고 해당 형식의 형식 매개 변수를 사용하기 때문에 제네릭이 아닙니다. 메서드는 고유한 형식 매개 변수 목록이 있는 경우에만 제네릭입니다. 제네릭 메서드는 이 예제와 같이 제네릭이 아닌 형식에 나타날 수 있습니다. 제네릭 형식에 대한 비제네릭 메서드의 예는 **방법: 리플렉션 내보내기를 사용하여 제네릭 형식 정의**를 참조하세요.

제네릭 메서드 정의

1. 시작하기 전에 상위 수준 언어를 사용하여 작성할 때 제네릭 메서드가 어떻게 표시되는지 살펴보는 것이 유용합니다. 다음 코드는 제네릭 메서드를 호출하는 코드와 함께 이 문서의 예제 코드에 포함되어 있습니다. 메서드에는 두 개의 형식 매개 변수가 있으며 `TOutput`, `TInput` 두 번째 매개 변수는 참조 형식(`class`)이어야 하고 매개 변수가 없는 생성자(`new`)가 있어야 하며 구현 `ICollection<TInput>` 해야 합니다. 이 인터페이스 제약 조건은 메서드를 ICollection<T>.Add 사용하여 메서드가 만드는 컬렉션에 `TOutput` 요소를 추가할 수 있도록 합니다. 메서드에는 하나의 형식 매개 변수가 있으며, `input` 는 `TInput` 의 배열입니다. 메서드는 형식 `TOutput` 의 컬렉션을 만들고 컬렉션의 `input` 요소를 복사합니다.

```
C#

public static TOutput Factory<TInput, TOutput>(TInput[] tarray)
    where TOutput : class, ICollection<TInput>, new()
{
    TOutput ret = new TOutput();
    ICollection<TInput> ic = ret;
```

```
    foreach (TInput t in tarray)
    {
        ic.Add(t);
    }
    return ret;
}
```

2. 제네릭 메서드가 속한 형식을 포함할 동적 어셈블리 및 동적 모듈을 정의합니다. 이 경우 어셈블리에는 이름이 지정된 `DemoMethodBuilder1` 모듈이 하나뿐이며 모듈 이름은 어셈블리 이름과 확장명과 동일합니다. 이 예제에서는 어셈블리가 디스크에 저장되고 실행되기도 하므로 AssemblyBuilderAccess.RunAndSave 지정됩니다. Ildasm.exe(IL 디스어셈블러)를 사용하여 DemoMethodBuilder1.dll 검사하고 1단계에 표시된 메서드의 CIL(공용 중간 언어)과 비교할 수 있습니다.

C#

```
AssemblyName asmName = new AssemblyName("DemoMethodBuilder1");
AppDomain domain = AppDomain.CurrentDomain;
AssemblyBuilder demoAssembly =
    domain.DefineDynamicAssembly(asmName,
        AssemblyBuilderAccess.RunAndSave);

// Define the module that contains the code. For an
// assembly with one module, the module name is the
// assembly name plus a file extension.
ModuleBuilder demoModule =
    demoAssembly.DefineDynamicModule(asmName.Name,
        asmName.Name+".dll");
```

3. 제네릭 메서드가 속한 형식을 정의합니다. 형식이 제네릭일 필요는 없습니다. 제네릭 메서드는 제네릭 또는 비제네릭 형식에 속할 수 있습니다. 이 예제에서 형식은 클래스이고 제네릭이 아니며 이름이 지정 `DemoType` 됩니다.

C#

```
TypeBuilder demoType =
    demoModule.DefineType("DemoType", TypeAttributes.Public);
```

4. 제네릭 메서드를 정의합니다. 제네릭 메서드의 정식 매개 변수 형식이 제네릭 메서드의 제네릭 형식 매개 변수에 의해 지정된 경우 메서드 오버로드를 사용하여 DefineMethod(String, MethodAttributes) 메서드를 정의합니다. 메서드의 제네릭 형식 매개 변수는 아직 정의되지 않았으므로 호출 DefineMethod시 메서드의 정식 매개 변수 형식을 지정할 수 없습니다. 이 예제에서 메서드의 이름은 `Factory`.입니다. 이 메서드는 public이며 `static` (`Shared`는 Visual Basic에서는)

```
C#

MethodBuilder factory =
    demoType.DefineMethod("Factory",
        MethodAttributes.Public | MethodAttributes.Static);
```

5. 매개 변수 이름이 포함된 문자열 배열을 `DemoMethod` 메서드에 전달하여 MethodBuilder.DefineGenericParameters 제네릭 형식 매개 변수를 정의합니다. 이렇게 하면 메서드가 제네릭 메서드로 만들어집니다. 다음 코드는 형식 매개 변수 `TInput` 및 `TOutput`.를 사용하여 제네릭 메서드를 만듭니 `Factory` 다. 코드를 더 쉽게 읽을 수 있도록 이러한 이름의 변수는 두 형식 매개 변수를 나타내는 개체를 저장 GenericTypeParameterBuilder 하도록 만들어집니다.

```
C#

string[] typeParameterNames = {"TInput", "TOutput"};
GenericTypeParameterBuilder[] typeParameters =
    factory.DefineGenericParameters(typeParameterNames);

GenericTypeParameterBuilder TInput = typeParameters[0];
GenericTypeParameterBuilder TOutput = typeParameters[1];
```

6. 필요에 따라 형식 매개 변수에 특수 제약 조건을 추가합니다. 메서드를 사용하여 SetGenericParameterAttributes 특수 제약 조건이 추가됩니다. 이 예제에서는 `TOutput` 참조 형식으로 제한되고 매개 변수가 없는 생성자를 포함하도록 제한됩니다.

```
C#

TOutput.SetGenericParameterAttributes(
    GenericParameterAttributes.ReferenceTypeConstraint |
    GenericParameterAttributes.DefaultConstructorConstraint);
```

7. 필요에 따라 형식 매개 변수에 클래스 및 인터페이스 제약 조건을 추가합니다. 이 예제에서 형식 매개 변수 `TOutput` 는 (`ICollection<TInput>` C#) 인터페이스를 구현하는 형식으로 `ICollection(Of TInput)` 제한됩니다. 이렇게 하면 메서드를 Add 사용하여 요소를 추가할 수 있습니다.

```
C#

Type icoll = typeof(ICollection<>);
Type icollOfTInput = icoll.MakeGenericType(TInput);
Type[] constraints = {icollOfTInput};
TOutput.SetInterfaceConstraints(constraints);
```

8. SetParameters 메서드를 사용하여 메서드의 형식 매개 변수를 정의합니다. 이 예제에서 `Factory` 메서드에는 하나의 매개변수인 `TInput` 배열이 있습니다. 이 형식은 `TInput`를 나타내는 GenericTypeParameterBuilder에서 MakeArrayType 메서드를 호출하여 생성됩니다. 인수 SetParameters 는 개체 배열입니다 Type .

```
C#

Type[] parms = {TInput.MakeArrayType()};
factory.SetParameters(parms);
```

9. SetReturnType 메서드를 사용하여 메서드의 반환 형식을 정의합니다. 이 예제에서는 인스턴스 `TOutput` 가 반환됩니다.

```
C#

factory.SetReturnType(TOutput);
```

10. 를 사용하여 ILGenerator메서드 본문을 내보낸다. 자세한 내용은 메서드 본문을 내보내는 데 수반되는 절차를 참조하세요.

> **ⓘ 중요**
>
> 제네릭 형식의 메서드에 대한 호출을 내보내고, 이러한 형식의 형식 인수가 제네릭 메서드의 형식 매개 변수인 경우, 메서드의 생성된 형태를 얻기 위해 `static`, GetConstructor(Type, ConstructorInfo), GetMethod(Type, MethodInfo), GetField(Type, FieldInfo) 메서드 오버로드를 TypeBuilder 클래스에서 사용해야 합니다. 메서드 본문을 내보내는 데 수반되는 프로시저가 이를 보여 줍니다.

11. 메서드가 포함된 형식을 완료하고 어셈블리를 저장합니다. 제네릭 메서드를 호출하는 프로시저는 완료된 메서드를 호출하는 두 가지 방법을 보여줍니다.

```
C#

// Complete the type.
Type dt = demoType.CreateType();
// Save the assembly, so it can be examined with Ildasm.exe.
demoAssembly.Save(asmName.Name+".dll");
```

메서드 본문 내보내기

1. 코드 생성기를 가져와서 지역 변수 및 레이블을 선언합니다. 이 DeclareLocal 메서드는 지역 변수를 선언하는 데 사용됩니다. 메서드에는 네 가지 지역 변수 `retVal`가 있습니다: `Factory` 메서드에서 반환하는 새로운 `TOutput`를 보관하기 위한 변수, `TOutput`를 `ICollection<TInput>`로 캐스팅할 때 사용하는 변수 `ic`, `TInput` 개체 배열 입력을 보관하기 위한 변수 `input`, 그리고 배열을 반복하기 위한 변수 `index`입니다. 메서드에는 루프를 입력하기 위한 레이블(`enterLoop`)과 루프의 맨 위에 위치하는 레이블(`loopAgain`)이 있습니다. 이들은 DefineLabel 메서드를 사용하여 정의됩니다.

   이 메서드가 가장 먼저 수행하는 작업은 opcode를 사용하여 Ldarg_0 인수를 로드하고 opcode를 사용하여 Stloc_S 로컬 변수 `input` 에 저장하는 것입니다.

   C#

   ```
   ILGenerator ilgen = factory.GetILGenerator();

   LocalBuilder retVal = ilgen.DeclareLocal(TOutput);
   LocalBuilder ic = ilgen.DeclareLocal(icollOfTInput);
   LocalBuilder input = ilgen.DeclareLocal(TInput.MakeArrayType());
   LocalBuilder index = ilgen.DeclareLocal(typeof(int));

   Label enterLoop = ilgen.DefineLabel();
   Label loopAgain = ilgen.DefineLabel();

   ilgen.Emit(OpCodes.Ldarg_0);
   ilgen.Emit(OpCodes.Stloc_S, input);
   ```

2. Activator.CreateInstance 메서드의 제네릭 메서드 오버로드를 사용하여 `TOutput`의 인스턴스를 생성하는 코드를 내보낸다. 이 오버로드를 사용하려면 지정된 형식에 매개 변수가 없는 생성자가 있어야 합니다. 이는 해당 제약 `TOutput`조건을 추가하는 이유입니다. `TOutput`을 MakeGenericMethod에 전달하여 생성된 제네릭 메서드를 만듭니다. 메서드를 호출하는 코드를 내보낸 후 다음을 사용하여 로컬 변수 `retVal` 에 저장하는 코드를 내보냅니다. Stloc_S

   C#

   ```
   MethodInfo createInst =
       typeof(Activator).GetMethod("CreateInstance", Type.EmptyTypes);
   MethodInfo createInstOfTOutput =
       createInst.MakeGenericMethod(TOutput);

   ilgen.Emit(OpCodes.Call, createInstOfTOutput);
   ilgen.Emit(OpCodes.Stloc_S, retVal);
   ```

3. 새 `TOutput` 개체 `ICollection(Of TInput)` 를 캐스팅하여 지역 변수 `ic`에 저장하는 코드를 내보낸다.

```
C#

ilgen.Emit(OpCodes.Ldloc_S, retVal);
ilgen.Emit(OpCodes.Box, TOutput);
ilgen.Emit(OpCodes.Castclass, icollOfTInput);
ilgen.Emit(OpCodes.Stloc_S, ic);
```

4. 메서드 ICollection<T>.Add를 나타내는 값을 MethodInfo 가져옵니다. `ICollection<TInput>`에 작동하는 메서드이므로 해당 생성된 형식에 특화된 `Add` 메서드를 가져와야 합니다. `icollOfTInput`에서 직접 MethodInfo을(를) 가져오는 GetMethod 메서드를 사용할 수 없습니다. 그 이유는 GenericTypeParameterBuilder로 생성된 형식에서는 GetMethod가 지원되지 않기 때문입니다. 대신, `icoll`에서 GetMethod를 호출하세요. 여기에는 ICollection<T> 제네릭 인터페이스의 제네릭 형식 정의가 포함됩니다. 그런 다음 메서드를 GetMethod(Type, MethodInfo)`static` 사용하여 생성된 형식에 MethodInfo 대한 값을 생성합니다. 다음 코드에서는 이를 보여 줍니다.

```
C#

MethodInfo mAddPrep = icoll.GetMethod("Add");
MethodInfo mAdd = TypeBuilder.GetMethod(icollOfTInput, mAddPrep);
```

5. 32비트 정수 0을 로드하고 변수에 저장하여 변수를 초기화하는 `index` 코드를 내보낸다. 레이블 `enterLoop`으로 분기하는 코드를 내보낸다. 이 레이블은 루프 내에 있으므로 아직 표시되지 않았습니다. 루프에 대한 코드는 다음 단계에서 내보내집니다.

```
C#

// Initialize the count and enter the loop.
ilgen.Emit(OpCodes.Ldc_I4_0);
ilgen.Emit(OpCodes.Stloc_S, index);
ilgen.Emit(OpCodes.Br_S, enterLoop);
```

6. 루프에 대한 코드를 내보냅니다. 첫 번째 단계는 MarkLabel를 `loopAgain` 레이블로 호출하여 루프의 맨 위를 표시하는 것입니다. 이제 레이블을 사용하는 분기 문이 코드의 이 지점으로 분기됩니다. 다음 단계는 개체를 `ICollection(Of TInput)`로 캐스팅한 후 `TOutput` 스택에 푸시하는 것입니다. 즉시 필요하지는 않지만 메서드를 호출 `Add` 하기 위한 위치에 있어야 합니다. 그런 다음 입력 배열이 스택으로 푸시된 다음 `index` 현재 인덱스가 포함된 변수를 배열로 푸시합니다. opcode는 Ldelem 스택에

서 인덱스 및 배열을 팝하고 인덱싱된 배열 요소를 스택에 푸시합니다. 이제 스택이 메서드를 호출할 ICollection<T>.Add 준비가 되었습니다. 그러면 컬렉션과 새 요소가 스택에서 팝되고 컬렉션에 요소가 추가됩니다.

루프의 나머지 코드는 인덱스 및 테스트를 증가하여 루프가 완료되었는지 확인합니다. 인덱스 및 32비트 정수 1은 스택에 푸시되고 추가되며 스택에 합계가 남습니다. 합계는 에 저장됩니다 `index`. MarkLabel 는 이 지점을 루프의 진입점으로 설정하기 위해 호출됩니다. 인덱스가 다시 로드됩니다. 입력 배열은 스택에 푸시되고 Ldlen 길이를 가져오기 위해 내보내집니다. 이제 인덱스와 길이가 스택에 있으며 Clt 비교하기 위해 내보내집니다. 인덱스가 길이 Brtrue_S 보다 작으면 루프의 시작 부분으로 다시 분기합니다.

```
C#

ilgen.MarkLabel(loopAgain);

ilgen.Emit(OpCodes.Ldloc_S, ic);
ilgen.Emit(OpCodes.Ldloc_S, input);
ilgen.Emit(OpCodes.Ldloc_S, index);
ilgen.Emit(OpCodes.Ldelem, TInput);
ilgen.Emit(OpCodes.Callvirt, mAdd);

ilgen.Emit(OpCodes.Ldloc_S, index);
ilgen.Emit(OpCodes.Ldc_I4_1);
ilgen.Emit(OpCodes.Add);
ilgen.Emit(OpCodes.Stloc_S, index);

ilgen.MarkLabel(enterLoop);
ilgen.Emit(OpCodes.Ldloc_S, index);
ilgen.Emit(OpCodes.Ldloc_S, input);
ilgen.Emit(OpCodes.Ldlen);
ilgen.Emit(OpCodes.Conv_I4);
ilgen.Emit(OpCodes.Clt);
ilgen.Emit(OpCodes.Brtrue_S, loopAgain);
```

7. `TOutput` 개체를 스택에 푸시하고 메서드에서 반환하는 코드를 생성한다. 지역 변수 `retVal` 와 `ic` 둘 다 새 `TOutput` `ic` 변수에 대한 참조를 포함합니다. 메서드에 액세스 ICollection<T>.Add 하는 데만 사용됩니다.

```
C#

ilgen.Emit(OpCodes.Ldloc_S, retVal);
ilgen.Emit(OpCodes.Ret);
```

제네릭 메서드 호출

1. `Factory` 는 제네릭 메서드 정의입니다. 호출하려면 제네릭 형식 매개 변수에 형식을 할당해야 합니다. 이 MakeGenericMethod 작업을 수행하려면 이 메서드를 사용합니다. 다음 코드는 String가 `TInput`에 지정되고 C#의 `List(Of String)`가 `TOutput`에 지정되는 생성된 제네릭 메서드를 만들며, 메서드의 문자열 표현을 출력합니다.

```
C#

MethodInfo m = dt.GetMethod("Factory");
MethodInfo bound =
    m.MakeGenericMethod(typeof(string), typeof(List<string>));

// Display a string representing the bound method.
Console.WriteLine(bound);
```

2. 런타임에 바인드된 메서드를 호출하려면 Invoke 메서드를 사용합니다. 다음 코드는 문자열 배열만 포함하는 배열 Object을 만들고 제네릭 메서드의 인수 목록으로 전달합니다. 첫 번째 매개 변수 Invoke는 메서드가 `static`이기 때문에 null 참조입니다. 반환 값이 `List(Of String)`로 캐스팅된 다음, 첫 번째 요소가 표시됩니다.

```
C#

object o = bound.Invoke(null, new object[]{arr});
List<string> list2 = (List<string>) o;

Console.WriteLine($"The first element is: {list2[0]}");
```

3. 대리자를 사용하여 메서드를 호출하려면 생성된 제네릭 메서드의 서명과 일치하는 대리자가 있어야 합니다. 이 작업을 수행하는 쉬운 방법은 제네릭 대리자를 만드는 것입니다. 다음 코드는 예제 코드에 정의된 제네릭 대리자 `D`의 인스턴스를 Delegate.CreateDelegate(Type, MethodInfo) 메서드 오버로드를 사용하여 만들고 대리자를 호출합니다. 대리자는 지연 바인딩된 호출보다 성능이 우수합니다.

```
C#

Type dType = typeof(D<string, List <string>>);
D<string, List <string>> test;
test = (D<string, List <string>>)
    Delegate.CreateDelegate(dType, bound);

List<string> list3 = test(arr);
Console.WriteLine($"The first element is: {list3[0]}");
```

4. 내보낸 메서드는 저장된 어셈블리를 참조하는 프로그램에서도 호출할 수 있습니다.

예시

다음 코드 예제에서는 제네릭 메서드 `Factory`를 사용하여 제네릭이 아닌 형식 `DemoType`을 만듭니다. 이 메서드에는 입력 형식을 지정하고 `TOutput` 출력 형식을 지정하는 두 개의 제네릭 형식 매개 변수 `TInput` 가 있습니다. `TOutput` 형식 매개 변수는 `ICollection<TInput>`를 구현하고 (Visual Basic의 경우 `ICollection(Of TInput)`), 참조 형식이어야 하며, 매개 변수가 없는 생성자를 가져야 합니다.

메서드에는 하나의 정식 매개 변수가 있으며, 이 매개 변수는 배열입니다 `TInput`. 이 메서드는 입력 배열의 `TOutput` 모든 요소를 포함하는 인스턴스를 반환합니다. `TOutput` 는 ICollection<T> 제네릭 인터페이스를 구현하는 모든 제네릭 컬렉션 형식일 수 있습니다.

코드가 실행되면 동적 어셈블리가 DemoGenericMethod1.dll저장되고 Ildasm.exe(IL 디스어셈블러)를 사용하여 검사할 수 있습니다.

> **참고**
>
> 코드를 내보내는 방법을 알아보는 좋은 방법은 내보내려는 작업을 수행하는 프로그램을 작성하고 디스어셈블러를 사용하여 컴파일러에서 생성된 CIL을 검사하는 것입니다.

코드 예제에는 내보낸 메서드와 동일한 소스 코드가 포함됩니다. 방출된 메서드는 지연 바인딩되어 코드 예제에 선언된 제네릭 대리자를 사용하여 호출됩니다.

C#

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;

// Declare a generic delegate that can be used to execute the
// finished method.
//
public delegate TOut D<TIn, TOut>(TIn[] input);

class GenericMethodBuilder
{
    // This method shows how to declare, in Visual Basic, the generic
    // method this program emits. The method has two type parameters,
    // TInput and TOutput, the second of which must be a reference type
    // (class), must have a parameterless constructor (new()), and must
    // implement ICollection<TInput>. This interface constraint
    // ensures that ICollection<TInput>.Add can be used to add
    // elements to the TOutput object the method creates. The method
    // has one formal parameter, input, which is an array of TInput.
```

```
    // The elements of this array are copied to the new TOutput.
    //
    public static TOutput Factory<TInput, TOutput>(TInput[] tarray)
        where TOutput : class, ICollection<TInput>, new()
    {
        TOutput ret = new TOutput();
        ICollection<TInput> ic = ret;

        foreach (TInput t in tarray)
        {
            ic.Add(t);
        }
        return ret;
    }

    public static void Main()
    {
        // The following shows the usage syntax of the C#
        // version of the generic method emitted by this program.
        // Note that the generic parameters must be specified
        // explicitly, because the compiler does not have enough
        // context to infer the type of TOutput. In this case, TOutput
        // is a generic List containing strings.
        //
        string[] arr = {"a", "b", "c", "d", "e"};
        List<string> list1 =
            GenericMethodBuilder.Factory<string, List <string>>(arr);
        Console.WriteLine($"The first element is: {list1[0]}");

        // Creating a dynamic assembly requires an AssemblyName
        // object, and the current application domain.
        //
        AssemblyName asmName = new AssemblyName("DemoMethodBuilder1");
        AppDomain domain = AppDomain.CurrentDomain;
        AssemblyBuilder demoAssembly =
            domain.DefineDynamicAssembly(asmName,
                AssemblyBuilderAccess.RunAndSave);

        // Define the module that contains the code. For an
        // assembly with one module, the module name is the
        // assembly name plus a file extension.
        ModuleBuilder demoModule =
            demoAssembly.DefineDynamicModule(asmName.Name,
                asmName.Name+".dll");

        // Define a type to contain the method.
        TypeBuilder demoType =
            demoModule.DefineType("DemoType", TypeAttributes.Public);

        // Define a public static method with standard calling
        // conventions. Do not specify the parameter types or the
        // return type, because type parameters will be used for
        // those types, and the type parameters have not been
        // defined yet.
        //
```

```
        MethodBuilder factory =
            demoType.DefineMethod("Factory",
                MethodAttributes.Public | MethodAttributes.Static);

        // Defining generic type parameters for the method makes it a
        // generic method. To make the code easier to read, each
        // type parameter is copied to a variable of the same name.
        //
        string[] typeParameterNames = {"TInput", "TOutput"};
        GenericTypeParameterBuilder[] typeParameters =
            factory.DefineGenericParameters(typeParameterNames);

        GenericTypeParameterBuilder TInput = typeParameters[0];
        GenericTypeParameterBuilder TOutput = typeParameters[1];

        // Add special constraints.
        // The type parameter TOutput is constrained to be a reference
        // type, and to have a parameterless constructor. This ensures
        // that the Factory method can create the collection type.
        //
        TOutput.SetGenericParameterAttributes(
            GenericParameterAttributes.ReferenceTypeConstraint |
            GenericParameterAttributes.DefaultConstructorConstraint);

        // Add interface and base type constraints.
        // The type parameter TOutput is constrained to types that
        // implement the ICollection<T> interface, to ensure that
        // they have an Add method that can be used to add elements.
        //
        // To create the constraint, first use MakeGenericType to bind
        // the type parameter TInput to the ICollection<T> interface,
        // returning the type ICollection<TInput>, then pass
        // the newly created type to the SetInterfaceConstraints
        // method. The constraints must be passed as an array, even if
        // there is only one interface.
        //
        Type icoll = typeof(ICollection<>);
        Type icollOfTInput = icoll.MakeGenericType(TInput);
        Type[] constraints = {icollOfTInput};
        TOutput.SetInterfaceConstraints(constraints);

        // Set parameter types for the method. The method takes
        // one parameter, an array of type TInput.
        Type[] parms = {TInput.MakeArrayType()};
        factory.SetParameters(parms);

        // Set the return type for the method. The return type is
        // the generic type parameter TOutput.
        factory.SetReturnType(TOutput);

        // Generate a code body for the method.
        // ------------------------------------
        // Get a code generator and declare local variables and
        // labels. Save the input array to a local variable.
        //
```

```
        ILGenerator ilgen = factory.GetILGenerator();

        LocalBuilder retVal = ilgen.DeclareLocal(TOutput);
        LocalBuilder ic = ilgen.DeclareLocal(icollOfTInput);
        LocalBuilder input = ilgen.DeclareLocal(TInput.MakeArrayType());
        LocalBuilder index = ilgen.DeclareLocal(typeof(int));

        Label enterLoop = ilgen.DefineLabel();
        Label loopAgain = ilgen.DefineLabel();

        ilgen.Emit(OpCodes.Ldarg_0);
        ilgen.Emit(OpCodes.Stloc_S, input);

        // Create an instance of TOutput, using the generic method
        // overload of the Activator.CreateInstance method.
        // Using this overload requires the specified type to have
        // a parameterless constructor, which is the reason for adding
        // that constraint to TOutput. Create the constructed generic
        // method by passing TOutput to MakeGenericMethod. After
        // emitting code to call the method, emit code to store the
        // new TOutput in a local variable.
        //
        MethodInfo createInst =
            typeof(Activator).GetMethod("CreateInstance", Type.EmptyTypes);
        MethodInfo createInstOfTOutput =
            createInst.MakeGenericMethod(TOutput);

        ilgen.Emit(OpCodes.Call, createInstOfTOutput);
        ilgen.Emit(OpCodes.Stloc_S, retVal);

        // Load the reference to the TOutput object, cast it to
        // ICollection<TInput>, and save it.
        //
        ilgen.Emit(OpCodes.Ldloc_S, retVal);
        ilgen.Emit(OpCodes.Box, TOutput);
        ilgen.Emit(OpCodes.Castclass, icollOfTInput);
        ilgen.Emit(OpCodes.Stloc_S, ic);

        // Loop through the array, adding each element to the new
        // instance of TOutput. Note that in order to get a MethodInfo
        // for ICollection<TInput>.Add, it is necessary to first
        // get the Add method for the generic type defintion,
        // ICollection<T>.Add. This is because it is not possible
        // to call GetMethod on icollOfTInput. The static overload of
        // TypeBuilder.GetMethod produces the correct MethodInfo for
        // the constructed type.
        //
        MethodInfo mAddPrep = icoll.GetMethod("Add");
        MethodInfo mAdd = TypeBuilder.GetMethod(icollOfTInput, mAddPrep);

        // Initialize the count and enter the loop.
        ilgen.Emit(OpCodes.Ldc_I4_0);
        ilgen.Emit(OpCodes.Stloc_S, index);
        ilgen.Emit(OpCodes.Br_S, enterLoop);
```

```
        // Mark the beginning of the loop. Push the ICollection
        // reference on the stack, so it will be in position for the
        // call to Add. Then push the array and the index on the
        // stack, get the array element, and call Add (represented
        // by the MethodInfo mAdd) to add it to the collection.
        //
        // The other ten instructions just increment the index
        // and test for the end of the loop. Note the MarkLabel
        // method, which sets the point in the code where the
        // loop is entered. (See the earlier Br_S to enterLoop.)
        //
        ilgen.MarkLabel(loopAgain);

        ilgen.Emit(OpCodes.Ldloc_S, ic);
        ilgen.Emit(OpCodes.Ldloc_S, input);
        ilgen.Emit(OpCodes.Ldloc_S, index);
        ilgen.Emit(OpCodes.Ldelem, TInput);
        ilgen.Emit(OpCodes.Callvirt, mAdd);

        ilgen.Emit(OpCodes.Ldloc_S, index);
        ilgen.Emit(OpCodes.Ldc_I4_1);
        ilgen.Emit(OpCodes.Add);
        ilgen.Emit(OpCodes.Stloc_S, index);

        ilgen.MarkLabel(enterLoop);
        ilgen.Emit(OpCodes.Ldloc_S, index);
        ilgen.Emit(OpCodes.Ldloc_S, input);
        ilgen.Emit(OpCodes.Ldlen);
        ilgen.Emit(OpCodes.Conv_I4);
        ilgen.Emit(OpCodes.Clt);
        ilgen.Emit(OpCodes.Brtrue_S, loopAgain);

        ilgen.Emit(OpCodes.Ldloc_S, retVal);
        ilgen.Emit(OpCodes.Ret);

        // Complete the type.
        Type dt = demoType.CreateType();
        // Save the assembly, so it can be examined with Ildasm.exe.
        demoAssembly.Save(asmName.Name+".dll");

        // To create a constructed generic method that can be
        // executed, first call the GetMethod method on the completed
        // type to get the generic method definition. Call MakeGenericType
        // on the generic method definition to obtain the constructed
        // method, passing in the type arguments. In this case, the
        // constructed method has string for TInput and List<string>
        // for TOutput.
        //
        MethodInfo m = dt.GetMethod("Factory");
        MethodInfo bound =
            m.MakeGenericMethod(typeof(string), typeof(List<string>));

        // Display a string representing the bound method.
        Console.WriteLine(bound);
```

```
        // Once the generic method is constructed,
        // you can invoke it and pass in an array of objects
        // representing the arguments. In this case, there is only
        // one element in that array, the argument 'arr'.
        //
        object o = bound.Invoke(null, new object[]{arr});
        List<string> list2 = (List<string>) o;

        Console.WriteLine($"The first element is: {list2[0]}");

        // You can get better performance from multiple calls if
        // you bind the constructed method to a delegate. The
        // following code uses the generic delegate D defined
        // earlier.
        //
        Type dType = typeof(D<string, List <string>>);
        D<string, List <string>> test;
        test = (D<string, List <string>>)
            Delegate.CreateDelegate(dType, bound);

        List<string> list3 = test(arr);
        Console.WriteLine($"The first element is: {list3[0]}");
    }
}

/* This code example produces the following output:

The first element is: a
System.Collections.Generic.List`1[System.String] Factory[String,List`1]
(System.String[])
The first element is: a
The first element is: a
 */
```

참고하십시오

- MethodBuilder
- 방법: 리플렉션 내보내기를 사용하여 제네릭 형식 정의

동적 형식 생성을 위해 수집 가능한 어셈블리

2025. 06. 17.

수집 가능한 어셈블리 는 생성된 애플리케이션 도메인을 언로드하지 않고 언로드할 수 있는 동적 어셈블리입니다. 수집 가능한 어셈블리에서 사용되는 모든 관리형 및 관리되지 않는 메모리와 포함된 형식을 회수할 수 있습니다. 어셈블리 이름과 같은 정보는 내부 테이블에서 제거됩니다.

언로드를 활성화하려면 동적 어셈블리를 만들 때 AssemblyBuilderAccess.RunAndCollect 플래그를 사용하십시오. 어셈블리는 일시적이며(즉, 저장할 수 없음) 수집 가능한 어셈블리 제한 섹션에 설명된 제한 사항이 적용됩니다. CLR(공용 언어 런타임)은 어셈블리와 연결된 모든 개체를 해제할 때 수집 가능한 어셈블리를 자동으로 언로드합니다. 다른 모든 면에서 수집 가능한 어셈블리는 다른 동적 어셈블리와 동일한 방식으로 만들어지고 사용됩니다.

수집 가능한 어셈블리의 수명

수집 가능한 어셈블리의 수명은 포함된 형식 및 해당 형식에서 만든 개체에 대한 참조가 존재하여 제어됩니다. 공용 언어 런타임은 다음 중 하나 이상이 있는 한 어셈블리를 언로드하지 않습니다(`T` 어셈블리에 정의된 모든 형식).

- `T`인스턴스입니다.
- 배열의 인스턴스입니다 `T`.
- 해당 형식 인수 중 하나로 있는 `T` 제네릭 형식의 인스턴스입니다. 이에는 해당 컬렉션이 비어 있더라도 제네릭 `T` 컬렉션이 포함됩니다.
- Type 또는 TypeBuilder의 인스턴스로, `T`를 나타냅니다.

  > **ⓘ 중요**
  >
  > 어셈블리의 일부를 나타내는 모든 개체를 해제해야 합니다. ModuleBuilder를 정의하는 개체는 `T`에 대한 참조를 유지하고, TypeBuilder 개체는 AssemblyBuilder에 대한 참조를 유지하므로 이러한 개체에 대한 참조를 반드시 해제해야 합니다. LocalBuilder 또는 ILGenerator가 `T` 생성에 사용될 경우에도 언로드를 방지합니다.

- 코드를 실행하여 여전히 접근 가능한 동적으로 정의된 다른 형식 `T`에 의해 이루어진 `T1`의 정적 참조입니다. 예를 들어, `T1`은 `T`에서 파생될 수 있으며, `T`는 `T1`의 메서드의 매개 변수 형식일 수 있습니다.
- `ByRef`에 속하는 정적 필드인 `T`입니다.

- RuntimeTypeHandle 또는 RuntimeFieldHandle의 구성 요소를 참조하는 RuntimeMethodHandle, `T`, 또는 `T`.
- 나타내는 Type 개체에 액세스하기 위해 간접적으로 또는 직접 사용할 수 있는 리플렉션 개체의 `T` 인스턴스입니다. 예를 들어, 요소 형식이 Type인 배열 형식이나 `T`을(를) 형식 인수로 가진 제네릭 형식에서 `T` 개체를 가져올 수 있습니다.
- 어떤 스레드의 호출 스택에 있는 `M` 메서드이며, 여기서 `M`는 `T`의 메서드이거나 어셈블리에 정의된 모듈 수준의 메서드입니다.
- 어셈블리의 모듈에 정의된 정적 메서드에 대한 대리자입니다.

이 목록의 항목이 어셈블리의 형식 하나 또는 메서드 하나에만 있는 경우 런타임에서 어셈블리를 언로드할 수 없습니다.

> **참고**
>
> 런타임은 목록의 모든 항목에 대한 종료자가 실행될 때까지 어셈블리를 실제로 언로드하지 않습니다.

수명을 추적하기 위해 수집 가능한 어셈블리 생성에 생성되고 사용되는 생성된 제네릭 형식 `List<int>` (C#) 또는 `List(Of Integer)` (Visual Basic에서) 제네릭 형식 정의가 포함된 어셈블리 또는 해당 형식 인수 중 하나의 정의가 포함된 어셈블리에서 정의된 것으로 간주됩니다. 사용되는 정확한 어셈블리는 구현 세부 정보이며 변경될 수 있습니다.

수집 가능한 어셈블리에 대한 제한 사항

수집 가능한 어셈블리에는 다음 제한 사항이 적용됩니다.

- **정적 참조**

  일반 동적 어셈블리의 형식은 수집 가능한 어셈블리에 정의된 형식에 대한 정적 참조를 가질 수 없습니다. 예를 들어 수집 가능한 어셈블리의 형식을 상속하는 일반 유형을 정의하는 경우 NotSupportedException 예외가 발생합니다. 수집 가능한 어셈블리의 형식은 다른 수집 가능한 어셈블리의 형식에 대한 정적 참조를 가질 수 있습니다. 그러나 이는 참조된 어셈블리의 수명을 참조하는 어셈블리의 수명과 동일하게 연장시킵니다.

.NET Framework의 수집 가능한 어셈블리에는 다음 제한 사항이 적용됩니다.

- **COM interop**

  수집 가능한 어셈블리 내에서 COM 인터페이스를 정의할 수 없으며, 수집 가능한 어셈블리 내의 형식 인스턴스를 COM 개체로 변환할 수 없습니다. 수집 가능한 어셈블리의 형식은

COM 호출 가능 래퍼(CCW) 또는 런타임 호출 가능 래퍼(RCW)로 사용할 수 없습니다. 그러나 수집 가능한 어셈블리의 형식은 COM 인터페이스를 구현하는 개체를 사용할 수 있습니다.

- **플랫폼 호출**

  특성이 있는 DllImportAttribute 메서드는 수집 가능한 어셈블리에서 선언될 때 컴파일되지 않습니다. 이 OpCodes.Calli 명령은 수집 가능한 어셈블리의 형식 구현에 사용할 수 없으며 이러한 형식을 관리되지 않는 코드로 마샬링할 수 없습니다. 그러나 수집할 수 없는 어셈블리에 선언된 진입점을 사용하여 네이티브 코드를 호출할 수 있습니다.

- **마샬링**

  수집 가능한 어셈블리에 정의된 개체(특히 대리자)는 마샬링할 수 없습니다. 이는 모든 일시적인 내보내기 형식에 대한 제한 사항입니다.

- **어셈블리 로딩**

  리플렉션 내보내기는 수집 가능한 어셈블리를 로드하는 데 지원되는 유일한 메커니즘입니다. 다른 형태의 어셈블리 로드를 사용하여 로드되는 어셈블리는 언로드할 수 없습니다.

- **컨텍스트 바인딩된 개체**

  컨텍스트 정적 변수는 지원되지 않습니다. 수집 가능한 어셈블리의 형식은 ContextBoundObject을(를) 확장할 수 없습니다. 그러나 수집 가능한 어셈블리의 코드는 다른 곳에서 정의된 컨텍스트 바인딩된 개체를 사용할 수 있습니다.

- **스레드 정적 데이터**

  스레드 정적 변수는 지원되지 않습니다.

.NET 9 이전의 .NET Framework 및 .NET 버전에서 수집 가능한 어셈블리에는 다음 제한 사항이 적용됩니다.

- **를 사용하는 정적 필드 `FixedAddressValueTypeAttribute`**

  수집 가능한 어셈블리에 정의된 정적 필드에는 FixedAddressValueTypeAttribute 속성을 적용할 수 없습니다.

참고하십시오

- 동적 메서드 및 어셈블리 내보내기

System.Reflection.Emit.AssemblyBuilder class

2025. 07. 26.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

동적 어셈블리는 리플렉션 내보내기 API를 사용하여 만든 어셈블리입니다. 동적 어셈블리는 다른 동적 또는 정적 어셈블리에 정의된 형식을 참조할 수 있습니다. 동일한 애플리케이션을 실행하는 동안 메모리에서 동적 어셈블리를 생성하고 해당 코드를 실행하는 데 사용할 AssemblyBuilder 수 있습니다. .NET 9에서는 어셈블리를 파일에 저장할 수 있도록 완전히 관리되는 리플렉션 내보내기 구현이 포함된 새로운 PersistedAssemblyBuilder 를 추가했습니다. .NET Framework에서 동적 어셈블리를 실행하고 파일에 저장하여 둘 다 수행할 수 있습니다. 저장을 위해 만든 동적 어셈블리를 *지속형* 어셈블리라고 하지만 일반 메모리 전용 어셈블리를 *일시적* 또는 *실행 가능* 어셈블리라고 합니다. .NET Framework에서 동적 어셈블리는 하나 이상의 동적 모듈로 구성됩니다. .NET Core 및 .NET 5 이상에서는 동적 어셈블리가 하나의 동적 모듈로만 구성됩니다.

인스턴스를 AssemblyBuilder 만드는 방법은 각 구현마다 다르지만 모듈, 형식, 메서드 또는 열거형을 정의하고 IL을 작성하는 추가 단계는 매우 유사합니다.

.NET에서 실행 가능한 동적 어셈블리

실행 가능한 AssemblyBuilder 개체를 얻으려면 메서드를 AssemblyBuilder.DefineDynamicAssembly 사용합니다. 동적 어셈블리는 다음 액세스 모드 중 하나를 사용하여 만들 수 있습니다.

- AssemblyBuilderAccess.Run

  AssemblyBuilder로 표시된 동적 어셈블리를 사용하여 보낸 코드를 실행할 수 있습니다.

- AssemblyBuilderAccess.RunAndCollect

  AssemblyBuilder로 표현된 동적 어셈블리는 생성된 코드를 실행하는 데 사용될 수 있으며, 가비지 수집기에 의해 자동으로 회수됩니다.

동적 어셈블리가 정의되고 나중에 변경할 수 없는 경우 메서드 호출 AssemblyBuilderAccess 에 적절한 AssemblyBuilder.DefineDynamicAssembly 값을 제공하여 액세스 모드를 지정해야 합니다. 런타임은 동적 어셈블리의 액세스 모드를 사용하여 어셈블리의 내부 표현을 최적화합니다.

다음 예제에서는 어셈블리를 만들고 실행하는 방법을 보여 줍니다.

```
C#
```

```
public void CreateAndRunAssembly(string assemblyPath)
{
    AssemblyBuilder ab = AssemblyBuilder.DefineDynamicAssembly(new
AssemblyName("MyAssembly"), AssemblyBuilderAccess.Run);
    ModuleBuilder mob = ab.DefineDynamicModule("MyModule");
    TypeBuilder tb = mob.DefineType("MyType", TypeAttributes.Public |
TypeAttributes.Class);
    MethodBuilder mb = tb.DefineMethod("SumMethod", MethodAttributes.Public |
MethodAttributes.Static,
                                                                typeof(int),
new Type[] {typeof(int), typeof(int)});
    ILGenerator il = mb.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Ldarg_1);
    il.Emit(OpCodes.Add);
    il.Emit(OpCodes.Ret);

    Type type = tb.CreateType();

    MethodInfo method = type.GetMethod("SumMethod");
    Console.WriteLine(method.Invoke(null, new object[] { 5, 10 }));
}
```

.NET의 지속형 동적 어셈블리

.NET에서 AssemblyBuilder로부터 파생된 PersistedAssemblyBuilder 형식을 사용하면 동적 어셈블리를 저장할 수 있습니다. 자세한 내용은 PersistedAssemblyBuilder의 사용 시나리오 및 예제를 참조하세요.

.NET Framework의 지속형 동적 어셈블리

.NET Framework에서 동적 어셈블리 및 모듈을 파일에 저장할 수 있습니다. 이 기능을 지원하기 위해 열거형은 AssemblyBuilderAccess 두 개의 추가 필드를 SaveRunAndSave선언합니다.

지속 가능한 동적 어셈블리의 동적 모듈은 메서드를 사용하여 Save 동적 어셈블리를 저장할 때 저장됩니다. 실행 파일을 생성하려면 어셈블리의 SetEntryPoint 진입점인 메서드를 식별하기 위해 메서드를 호출해야 합니다. 어셈블리는 메서드가 콘솔 애플리케이션 또는 Windows 기반 애플리케이션의 생성을 요청하지 않는 한 SetEntryPoint 기본적으로 DLL로 저장됩니다.

다음 예제에서는 .NET Framework를 사용하여 어셈블리를 만들고 저장하고 실행하는 방법을 보여 줍니다.

C#

```
public void CreateRunAndSaveAssembly(string assemblyPath)
{
```

```
    AssemblyBuilder ab = Thread.GetDomain().DefineDynamicAssembly(new
AssemblyName("MyAssembly"), AssemblyBuilderAccess.RunAndSave);
    ModuleBuilder mob = ab.DefineDynamicModule("MyAssembly.dll");
    TypeBuilder tb = mob.DefineType("MyType", TypeAttributes.Public |
TypeAttributes.Class);
    MethodBuilder meb = tb.DefineMethod("SumMethod", MethodAttributes.Public |
MethodAttributes.Static,
                                                                    typeof(int),
new Type[] {typeof(int), typeof(int)});
    ILGenerator il = meb.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Ldarg_1);
    il.Emit(OpCodes.Add);
    il.Emit(OpCodes.Ret);

    Type type = tb.CreateType();

    MethodInfo method = type.GetMethod("SumMethod");
    Console.WriteLine(method.Invoke(null, new object[] { 5, 10 }));
    ab.Save("MyAssembly.dll");
}
```

기본 Assembly 클래스의 일부 메서드, 예를 들어 `GetModules` 및 `GetLoadedModules`,는 AssemblyBuilder 개체에서 호출될 때 올바르게 작동하지 않을 수 있습니다. 정의된 동적 어셈블리를 로드하고 로드된 어셈블리에서 메서드를 호출할 수 있습니다. 예를 들어 리소스 모듈이 반환된 모듈 목록에 포함되도록 하려면 로드된 `GetModules` 개체를 호출 Assembly 합니다. 동적 어셈블리에 둘 이상의 동적 모듈이 포함된 경우 어셈블리의 매니페스트 파일 이름은 메서드의 첫 번째 인수 DefineDynamicModule 로 지정된 모듈의 이름과 일치해야 합니다.

사용 하 여 KeyPair 동적 어셈블리의 서명은 어셈블리 디스크에 저장 될 때까지 유효 하지 않습니다. 따라서 강력한 이름은 일시적인 동적 어셈블리에서 작동하지 않습니다.

동적 어셈블리는 다른 어셈블리에 정의된 형식을 참조할 수 있습니다. 임시 동적 어셈블리는 다른 임시 동적 어셈블리, 지속 가능한 동적 어셈블리 또는 정적 어셈블리에 정의된 형식을 안전하게 참조할 수 있습니다. 그러나 공용 언어 런타임에서는 지속 가능한 동적 모듈이 일시적인 동적 모듈에 정의된 형식을 참조할 수 없습니다. 이는 지속형 동적 모듈이 디스크에 저장된 후 로드될 때 런타임에서 일시적인 동적 모듈에 정의된 형식에 대한 참조를 확인할 수 없기 때문입니다.

원격 애플리케이션 도메인으로 내보내는 제한 사항

일부 시나리오에서는 원격 애플리케이션 도메인에서 동적 어셈블리를 만들고 실행해야 합니다. 리플렉션 내보내기에서는 동적 어셈블리를 원격 애플리케이션 도메인으로 직접 내보내는 것을 허용하지 않습니다. 해결 방법은 현재 애플리케이션 도메인에서 동적 어셈블리를 내보내고, 내보낸 동적 어셈블리를 디스크에 저장한 다음, 동적 어셈블리를 원격 애플리케이션 도메인에 로드하는 것입니다. 원격 및 애플리케이션 도메인은 .NET Framework에서만 지원됩니다.

System.Reflection.Emit.DynamicMethod 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

클래스를 DynamicMethod 사용하여 메서드를 포함할 동적 어셈블리 및 동적 형식을 생성하지 않고도 런타임에 메서드를 생성하고 실행할 수 있습니다. JIT 컴파일러에서 생성한 실행 코드는 개체가 DynamicMethod 회수될 때 함께 회수됩니다. 동적 메서드는 소량의 코드를 생성하고 실행하는 가장 효율적인 방법입니다.

동적 메서드는 익명으로 호스트되거나 모듈 또는 형식과 논리적으로 연결될 수 있습니다.

- 동적 메서드가 익명으로 호스트되는 경우 시스템 제공 어셈블리에 있으므로 다른 코드와 격리됩니다. 기본적으로 공용이 아닌 데이터에 대한 액세스 권한은 없습니다. 익명으로 호스트되는 동적 메서드는 ReflectionPermission 플래그로 ReflectionPermissionFlag.RestrictedMemberAccess 부여된 경우 JIT 컴파일러의 가시성 검사를 건너뛰는 기능이 제한될 수 있습니다. 동적 메서드에서 비공용 멤버에 액세스하는 어셈블리의 신뢰 수준은 동적 메서드를 내보낸 호출 스택의 신뢰 수준 또는 하위 집합과 같아야 합니다. 익명으로 호스트되는 동적 메서드에 대한 자세한 내용은 연습: 부분 신뢰 시나리오에서 코드 내보내기를 참조하세요.

- 동적 메서드가 지정한 모듈과 연결된 경우 동적 메서드는 해당 모듈에 효과적으로 전역화됩니다. 모듈의 모든 형식과 형식의 모든 `internal` (`Friend` Visual Basic) 멤버에 액세스할 수 있습니다. 코드를 포함하는 호출 스택에서 플래그에 대한 ReflectionPermission 요구를 충족할 수 있는 경우 모듈을 만들었는지 여부에 관계없이 동적 메서드를 모듈과 RestrictedMemberAccess 연결할 수 있습니다. 권한 부여에 ReflectionPermissionFlag.MemberAccess 플래그가 포함된 경우 동적 메서드는 JIT 컴파일러의 가시성 검사를 건너뛰고 모듈 또는 어셈블리의 다른 모듈에 선언된 모든 형식의 프라이빗 데이터에 액세스할 수 있습니다.

  > **참고**
  >
  > 동적 메서드가 연결된 모듈을 지정하는 경우 해당 모듈은 익명 호스팅에 사용되는 시스템 제공 어셈블리에 있으면 안 됩니다.

- 동적 메서드가 지정한 형식과 연결된 경우 액세스 수준에 관계없이 형식의 모든 멤버에 액세스할 수 있습니다. 또한 JIT 가시성 검사를 건너뛸 수 있습니다. 이렇게 하면 동적 메서드가 동일한 모듈 또는 어셈블리의 다른 모듈에 선언된 다른 형식의 프라이빗 데이터에 액세스할 수 있습니다. 동적 메서드는 모든 형식과 연결할 수 있습니다. 그러나 코드에

ReflectionPermission가 부여되려면 RestrictedMemberAccess와 MemberAccess 플래그 모두가 포함되어야 합니다.

다음 표는 플래그가 ReflectionPermission 부여된 경우와 그렇지 않은 경우(RestrictedMemberAccess) JIT 표시 여부 검사에 따라 익명으로 호스트되는 동적 메서드에서 액세스할 수 있는 형식과 멤버를 보여줍니다.

테이블 확장

가시성 검사	`RestrictedMemberAccess`이 없는 경우	`RestrictedMemberAccess`와 함께
JIT 가시성 검사를 건너뛰지 않고	모든 어셈블리에 있는 public 형식의 공용 멤버입니다.	모든 어셈블리에 있는 public 형식의 공용 멤버입니다.
제한된 상태에서 JIT 가시성 검사 건너뛰기	모든 어셈블리에 있는 public 형식의 공용 멤버입니다.	동적 메서드를 내보낸 어셈블리의 신뢰 수준과 같거나 작은 신뢰 수준을 가진 어셈블리에서만 모든 형식의 모든 멤버가 가능합니다.

다음 표에서는 모듈 또는 모듈의 형식과 연결된 동적 메서드에 액세스할 수 있는 형식과 멤버를 보여 있습니다.

테이블 확장

JIT 가시성 검사 건너뛰기	모듈과 연결됨	형식과 연결됨
아니오	모듈 내의 공용, 내부 및 프라이빗 형식의 멤버들 중 공용 및 내부 멤버입니다. 모든 어셈블리에 있는 public 형식의 공용 멤버입니다.	연관된 유형의 모든 멤버입니다. 모듈에 있는 다른 모든 형식의 공용 및 내부 멤버입니다. 모든 어셈블리에 있는 public 형식의 공용 멤버입니다.
예	모든 조립체에 있는 모든 종류의 모든 멤버입니다.	모든 조립체에 있는 모든 종류의 모든 멤버입니다.

모듈과 연결된 동적 메서드에는 해당 모듈의 권한이 있습니다. 형식과 연결된 동적 메서드에는 해당 형식을 포함하는 모듈의 권한이 있습니다.

동적 메서드 및 해당 매개 변수의 이름을 지정할 필요는 없지만 디버깅에 도움이 되는 이름을 지정할 수 있습니다. 사용자 지정 특성은 동적 메서드 또는 해당 매개 변수에서 지원되지 않습니다.

동적 메서드는 `static` 메서드(`Shared` Visual Basic의 메서드)이지만 대리자 바인딩에 대한 완화된 규칙을 사용하면 동적 메서드를 개체에 바인딩할 수 있으므로 해당 대리자 인스턴스를 사용

하여 호출할 때 인스턴스 메서드처럼 작동합니다. 이것을 설명하는 예가 CreateDelegate(Type, Object) 메서드 오버로드에 대해 제공되었습니다.

확인

다음 목록에는 동적 메서드에 확인할 수 없는 코드가 포함될 수 있는 조건이 요약되어 있습니다. 예를 들어, InitLocals 속성이 `false`으로 설정된 경우 그 동적 메서드는 검증할 수 없습니다.

- 보안에 중요한 어셈블리와 연결된 동적 메서드도 보안에 중요하며 확인을 건너뛸 수 있습니다. 예를 들어 데스크톱 애플리케이션으로 실행되는 보안 특성이 없는 어셈블리는 런타임에서 보안에 중요한 것으로 처리됩니다. 동적 메서드를 어셈블리와 연결하면 동적 메서드에 확인할 수 없는 코드가 포함될 수 있습니다.
- 확인되지 않는 코드를 포함하는 동적 메서드가 수준 1 투명도가 있는 어셈블리와 연결된 경우 JIT(Just-In-Time) 컴파일러는 보안 요구를 주입합니다. 동적 메서드가 완전히 신뢰할 수 있는 코드에 의해 실행되는 경우에만 요청이 성공합니다. Security-Transparent 코드, 수준 1을 참조하세요.
- 확인되지 않는 코드를 포함하는 동적 메서드가 수준 2 투명도(예: mscorlib.dll)가 있는 어셈블리와 연결된 경우 보안 요구 대신 예외(JIT 컴파일러에 의해 삽입됨)를 throw합니다. Security-Transparent 코드, 수준 2를 참조하세요.
- 확인되지 않는 코드를 포함하는 익명으로 호스트된 동적 메서드는 항상 예외를 throw합니다. 완전히 신뢰할 수 있는 코드에서 만들고 실행하더라도 확인을 건너뛸 수 없습니다.

확인되지 않는 코드에 대해 throw되는 예외는 동적 메서드가 호출되는 방식에 따라 달라집니다. 메서드에서 CreateDelegate 반환된 대리자를 사용하여 동적 메서드를 호출하면 throw VerificationException 됩니다. 동적 메서드를 Invoke 메서드를 사용하여 호출하면 내부 TargetInvocationException와 함께 VerificationException이 throw됩니다.

System.Reflection.Emit.DynamicMethod.CreateDelegate 메서드

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

CreateDelegate(Type)

CreateDelegate 메서드 또는 메서드를 호출하면 Invoke 동적 메서드가 완료됩니다. 매개 변수 정의를 수정하거나 MSIL(Microsoft 중간 언어)을 더 내보내는 등 동적 메서드를 변경하려는 추가 시도는 무시됩니다. 예외가 throw되지 않습니다.

고유한 MSIL 생성기가 있는 경우 동적 메서드에 대한 메서드 본문을 만들려면 메서드를 GetDynamicILInfo 호출하여 개체를 DynamicILInfo 가져옵니다. 고유한 MSIL 생성기가 없는 경우 메서드를 호출 GetILGenerator 하여 메서드 본문을 생성하는 데 사용할 수 있는 개체를 가져옵니다 ILGenerator .

예시

다음 코드 예제에서는 두 개의 매개 변수를 사용하는 동적 메서드를 만듭니다. 이 예제에서는 첫 번째 매개 변수를 콘솔에 출력하는 간단한 함수 본문을 내보내고 두 번째 매개 변수를 메서드의 반환 값으로 사용합니다. 이 예제는 대리자를 생성하여 메서드를 완성한 후, 다른 매개변수로 대리자를 호출하고, 마지막으로 Invoke 메서드를 사용하여 동적 메서드를 호출합니다.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Test
{
    // Declare a delegate that will be used to execute the completed
    // dynamic method.
    private delegate int HelloInvoker(string msg, int ret);

    public static void Main()
    {
        // Create an array that specifies the types of the parameters
        // of the dynamic method. This method has a string parameter
        // and an int parameter.
        Type[] helloArgs = {typeof(string), typeof(int)};
```

```
        // Create a dynamic method with the name "Hello", a return type
        // of int, and two parameters whose types are specified by the
        // array helloArgs. Create the method in the module that
        // defines the Test class.
        DynamicMethod hello = new DynamicMethod("Hello",
            typeof(int),
            helloArgs,
            typeof(Test).Module);

        // Create an array that specifies the parameter types of the
        // overload of Console.WriteLine to be used in Hello.
        Type[] writeStringArgs = {typeof(string)};
        // Get the overload of Console.WriteLine that has one
        // String parameter.
        MethodInfo writeString =
            typeof(Console).GetMethod("WriteLine", writeStringArgs);

        // Get an ILGenerator and emit a body for the dynamic method.
        ILGenerator il = hello.GetILGenerator();
        // Load the first argument, which is a string, onto the stack.
        il.Emit(OpCodes.Ldarg_0);
        // Call the overload of Console.WriteLine that prints a string.
        il.EmitCall(OpCodes.Call, writeString, null);
        // The Hello method returns the value of the second argument;
        // to do this, load the second argument onto the stack and return.
        il.Emit(OpCodes.Ldarg_1);
        il.Emit(OpCodes.Ret);

        // Create a delegate that represents the dynamic method. This
        // action completes the method. Further attempts to change the
        // method are ignored and no exception is thrown.
        HelloInvoker hi =
            (HelloInvoker) hello.CreateDelegate(typeof(HelloInvoker));

        // Use the delegate to execute the dynamic method. Save and
        // print the return value.
        int retval = hi("\r\nHello, World!", 42);
        Console.WriteLine("Executing delegate hi(\"Hello, World!\", 42) returned
{0}",
            retval);

        // Do it again, with different arguments.
        retval = hi("\r\nHi, Mom!", 5280);
        Console.WriteLine("Executing delegate hi(\"Hi, Mom!\", 5280) returned {0}",
            retval);

        // Create an array of arguments to use with the Invoke method.
        object[] invokeArgs = {"\r\nHello, World!", 42};
        // Invoke the dynamic method using the arguments. This is much
        // slower than using the delegate, because you must create an
        // array to contain the arguments, and ValueType arguments
        // must be boxed.
        object objRet = hello.Invoke(null, invokeArgs);
        Console.WriteLine("hello.Invoke returned {0}", objRet);
```

```
    }
}
```

CreateDelegate(Type, Object)

이 메서드 오버로드는 특정 개체에 바인딩된 대리자를 만듭니다. 이러한 대리자는 첫 번째 인수를 통해 닫혀 있다고합니다. 메서드는 정적이지만 인스턴스 메서드인 것처럼 작동합니다. 인스턴스가 .입니다 `target`.

이 메서드 오버로드는 `target` 동적 메서드의 첫 번째 매개 변수와 동일한 형식이거나 해당 형식(예: 파생 클래스)에 할당할 수 있어야 합니다. 시그니처 `delegateType` 에는 첫 번째를 제외한 동적 메서드의 모든 매개 변수가 있습니다. 예를 들어 동적 메서드에 매개 변수가 String있고 Int32Byte , 매개 변수 `delegateType`Int32 가 있고 Byte; `target` 형식String인 경우입니다.

CreateDelegate 메서드 또는 메서드를 호출하면 Invoke 동적 메서드가 완료됩니다. 매개 변수 정의를 수정하거나 MSIL(Microsoft 중간 언어)을 더 내보내는 등 동적 메서드를 변경하려는 추가 시도는 무시됩니다. 예외가 throw되지 않습니다.

고유한 MSIL 생성기가 있는 경우 동적 메서드에 대한 메서드 본문을 만들려면 메서드를 GetDynamicILInfo 호출하여 개체를 DynamicILInfo 가져옵니다. 고유한 MSIL 생성기가 없는 경우 메서드를 호출 GetILGenerator 하여 메서드 본문을 생성하는 데 사용할 수 있는 개체를 가져옵니다 ILGenerator .

예시

다음 코드 예제는 메서드가 호출될 때마다 동일한 인스턴스에서 작동하도록 형식의 인스턴스에 DynamicMethod를 바인딩하는 대리자를 생성합니다.

이 코드 예제는 프라이빗 필드를 가진 클래스 `Example` 을 정의하고, 첫 번째 클래스에서 파생된 클래스 `DerivedFromExample`, `UseLikeStatic`를 반환하고 Int32, `Example` 타입의 매개 변수를 갖는 형식의 대리자 Int32, `UseLikeInstance`를 반환하고 Int32 타입의 매개 변수를 갖는 형식의 대리자 Int32를 정의합니다.

그런 다음 예제 코드는 인스턴스 DynamicMethod 의 프라이빗 필드를 변경하고 이전 값을 반환하는 형식을 만듭니다`Example`.

> ⓘ **참고 항목**
>
> 일반적으로 클래스의 내부 필드를 변경하는 것은 개체 지향 코딩 연습에 적합하지 않습니다.

예제 코드는 인스턴스 `Example` 를 만든 다음 두 대리자를 만듭니다. 첫 번째는 동적 메서드와 동일한 매개 변수를 포함하는 형식 `UseLikeStatic`입니다. 두 번째는 첫 번째 매개 변수(형식 `UseLikeInstance`)가 없는 형식 `Example`입니다. 이 대리자는 메서드 오버로드를 사용하여 CreateDelegate(Type, Object) 만들어집니다. 해당 메서드 오버로드의 두 번째 매개 변수는 인스턴스입니다 `Example`. 이 경우 방금 만든 인스턴스는 새로 만든 대리자에게 바인딩됩니다. 해당 대리자를 호출할 때마다 동적 메서드는 바인딩된 인스턴스 `Example`에서 작동합니다.

> ⓘ **참고 항목**
>
> 이것은 .NET Framework 2.0에 도입된 대리자 바인딩의 완화된 규칙의 예이며, Delegate.CreateDelegate 메서드의 새로운 오버로드와 함께 제공됩니다. 자세한 내용은 Delegate 클래스를 참조하세요.

`UseLikeStatic` 대리자가 호출하여 바인딩된 인스턴스 `Example` 를 `UseLikeInstance` 대리자를 통해 전달합니다. `UseLikeInstance` 그런 다음 두 대리자가 동일한 인스턴스`Example`에서 작동할 수 있도록 대리자가 호출됩니다. 내부 필드 값의 변경 내용은 각 호출 후에 표시됩니다. 마지막으로 대리자는 `UseLikeInstance` 인스턴스 `DerivedFromExample`에 바인딩되고 대리자 호출은 반복됩니다.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;

// These classes are for demonstration purposes.
//
public class Example
{
    private int id = 0;
    public Example(int id)
    {
        this.id = id;
    }
    public int ID { get { return id; }}
}

public class DerivedFromExample : Example
{
    public DerivedFromExample(int id) : base(id) {}
}

// Two delegates are declared: UseLikeInstance treats the dynamic
// method as if it were an instance method, and UseLikeStatic
// treats the dynamic method in the ordinary fashion.
//
public delegate int UseLikeInstance(int newID);
```

```
public delegate int UseLikeStatic(Example ex, int newID);

public class Demo
{
    public static void Main()
    {
        // This dynamic method changes the private id field. It has
        // no name; it returns the old id value (return type int);
        // it takes two parameters, an instance of Example and
        // an int that is the new value of id; and it is declared
        // with Example as the owner type, so it can access all
        // members, public and private.
        //
        DynamicMethod changeID = new DynamicMethod(
            "",
            typeof(int),
            new Type[] { typeof(Example), typeof(int) },
            typeof(Example)
        );

        // Get a FieldInfo for the private field 'id'.
        FieldInfo fid = typeof(Example).GetField(
            "id",
            BindingFlags.NonPublic | BindingFlags.Instance
        );

        ILGenerator ilg = changeID.GetILGenerator();

        // Push the current value of the id field onto the
        // evaluation stack. It's an instance field, so load the
        // instance of Example before accessing the field.
        ilg.Emit(OpCodes.Ldarg_0);
        ilg.Emit(OpCodes.Ldfld, fid);

        // Load the instance of Example again, load the new value
        // of id, and store the new field value.
        ilg.Emit(OpCodes.Ldarg_0);
        ilg.Emit(OpCodes.Ldarg_1);
        ilg.Emit(OpCodes.Stfld, fid);

        // The original value of the id field is now the only
        // thing on the stack, so return from the call.
        ilg.Emit(OpCodes.Ret);

        // Create a delegate that uses changeID in the ordinary
        // way, as a static method that takes an instance of
        // Example and an int.
        //
        UseLikeStatic uls =
            (UseLikeStatic) changeID.CreateDelegate(
                typeof(UseLikeStatic)
            );

        // Create an instance of Example with an id of 42.
        //
```

```
        Example ex = new Example(42);

        // Create a delegate that is bound to the instance of
        // of Example. This is possible because the first
        // parameter of changeID is of type Example. The
        // delegate has all the parameters of changeID except
        // the first.
        UseLikeInstance uli =
            (UseLikeInstance) changeID.CreateDelegate(
                typeof(UseLikeInstance),
                ex
            );

        // First, change the value of id by calling changeID as
        // a static method, passing in the instance of Example.
        //
        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uls(ex, 1492)
        );

        // Change the value of id again using the delegate bound
        // to the instance of Example.
        //
        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uli(2700)
        );

        Console.WriteLine("Final value of id: {0}", ex.ID);

        // Now repeat the process with a class that derives
        // from Example.
        //
        DerivedFromExample dfex = new DerivedFromExample(71);

        uli = (UseLikeInstance) changeID.CreateDelegate(
                typeof(UseLikeInstance),
                dfex
            );

        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uls(dfex, 73)
        );
        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uli(79)
        );
        Console.WriteLine("Final value of id: {0}", dfex.ID);
    }
}

/* This code example produces the following output:
```

```
Change the value of id; previous value: 42
Change the value of id; previous value: 1492
Final value of id: 2700
Change the value of id; previous value: 71
Change the value of id; previous value: 73
Final value of id: 79
 */
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 24.

System.Reflection.Emit.DynamicMethod 생성자

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

DynamicMethod(String, Type, Type[], Boolean) 생성자

이 생성자에서 만든 동적 메서드는 기존 형식 또는 모듈 대신 익명 어셈블리와 연결됩니다. 익명 어셈블리는 동적 메서드, 즉 다른 코드에서 격리하기 위한 샌드박스 환경을 제공하기 위해서만 존재합니다. 이 환경을 사용하면 동적 메서드를 부분적으로 신뢰할 수 있는 코드로 내보내고 실행할 수 있습니다.

익명으로 호스트되는 동적 메서드는 (Visual Basic에서) `private` 형식 또는 멤버 `protected` `internal` `Friend`에 자동으로 액세스할 수 없습니다. 이는 연결된 범위의 숨겨진 멤버에 액세스할 수 있는 기존 형식 또는 모듈과 연결된 동적 메서드와 다릅니다.

`true` 동적 메서드가 `private`, `protected`, 또는 `internal`에 해당하는 형식이나 멤버에 액세스해야 하는 경우 `restrictedSkipVisibility`을(를) 지정합니다. 이렇게 하면 동적 메서드가 이러한 멤버에 대한 액세스를 제한합니다. 즉, 다음 조건이 충족되는 경우에만 멤버에 액세스할 수 있습니다.

- 대상 멤버는 동적 메서드를 내보내는 호출 스택과 같거나 낮은 신뢰 수준이 있는 어셈블리에 속합니다.
- 동적 메서드를 내보내는 호출 스택은 플래그와 함께 ReflectionPermission 부여됩니다 ReflectionPermissionFlag.RestrictedMemberAccess. 이는 코드가 완전 신뢰로 실행될 때 항상 해당됩니다. 부분적으로 신뢰할 수 있는 코드의 경우 호스트가 명시적으로 사용 권한을 부여하는 경우에만 true입니다.

  > ⓘ **Important**
  >
  > 권한이 부여되지 않은 경우, 이 생성자를 호출할 때가 아니라 CreateDelegate 또는 동적 메서드를 호출하거나 호출할 때 보안 예외가 throw됩니다. 동적 메서드를 내보내는 데 특별한 권한이 필요하지 않습니다.

예를 들어 호출 스택에 제한된 멤버 액세스 권한이 부여된 경우 호출 스택에 있는 어셈블리의 프라이빗 멤버에 액세스할 수 있도록 설정된 `restrictedSkipVisibility` 동적 메서드가 있습니다 `true`. 호출 스택에 부분적으로 신뢰할 수 있는 코드가 있는 상태에서 동적 메서드를 만든 경우, 해당 어셈블리는 완전히 신뢰할 수 있기 때문에 .NET Framework 어셈블리의 형식에 있는 private 멤버에 액세스할 수 없습니다.

`restrictedSkipVisibility`가 `false`인 경우, JIT 가시성 검사가 강제됩니다. 동적 메서드의 코드는 public 클래스의 public 메서드에 액세스할 수 있습니다. 그러나 형식 또는 멤버가 `private`, `protected`, `internal`인 경우에 액세스하려고 하면 예외가 throw됩니다.

익명으로 호스트되는 동적 메서드가 생성되면 내보내는 어셈블리의 호출 스택이 포함됩니다. 메서드가 호출될 때 내보내는 호출 스택의 사용 권한은 실제 호출자의 권한 대신 사용됩니다. 따라서 동적 메서드는 신뢰 수준이 높은 어셈블리에 전달되고 실행되는 경우에도 해당 메서드를 내보낸 어셈블리보다 높은 수준의 권한으로 실행할 수 없습니다.

이 생성자는 메서드 특성 MethodAttributes.Public 및 MethodAttributes.Static호출 규칙을 CallingConventions.Standard지정합니다.

> ⓘ **참고 항목**
>
> 이 생성자는 .NET Framework 3.5 이상에서 도입되었습니다.

DynamicMethod(String, Type, Type[], Module) 생성자

이 생성자는 메서드 특성 MethodAttributes.Public 및 MethodAttributes.Static호출 규칙을 CallingConventions.Standard지정하고 JIT(Just-In-Time) 표시 유형 검사를 건너뛰지 않습니다.

이 생성자를 사용하여 만든 동적 메서드는 모듈`internal`에 포함된 모든 형식의 public 및 `Friend` (`m`Visual Basic에서) 멤버에 액세스할 수 있습니다.

> ⓘ **참고 항목**
>
> 이전 버전과의 호환성을 위해, 이 생성자는 다음 두 가지 조건이 모두 만족될 때 SecurityPermission을(를) SecurityPermissionFlag.ControlEvidence 플래그와 함께 요구합니다: `m`가 호출 모듈이 아닌 다른 모듈일 때, 그리고 ReflectionPermission에 대해서 ReflectionPermissionFlag.MemberAccess 플래그가 실패한 경우입니다. 요청 SecurityPermission 이 성공하면 작업이 허용됩니다.

예시

다음 코드 예제에서는 두 개의 매개 변수를 사용하는 동적 메서드를 만듭니다. 이 예제에서는 첫 번째 매개 변수를 콘솔에 출력하는 간단한 함수 본문을 내보내고 두 번째 매개 변수를 메서드의 반환 값으로 사용합니다. 이 예제는 대리자를 생성하여 메서드를 완성한 후, 다른 매개변수로 대리자를 호출하고, 마지막으로 Invoke(Object, BindingFlags, Binder, Object[], CultureInfo) 메서드를 사용하여 동적 메서드를 호출합니다.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Test
{
    // Declare a delegate that will be used to execute the completed
    // dynamic method.
    private delegate int HelloInvoker(string msg, int ret);

    public static void Main()
    {
        // Create an array that specifies the types of the parameters
        // of the dynamic method. This method has a string parameter
        // and an int parameter.
        Type[] helloArgs = {typeof(string), typeof(int)};

        // Create a dynamic method with the name "Hello", a return type
        // of int, and two parameters whose types are specified by the
        // array helloArgs. Create the method in the module that
        // defines the Test class.
        DynamicMethod hello = new DynamicMethod("Hello",
            typeof(int),
            helloArgs,
            typeof(Test).Module);

        // Create an array that specifies the parameter types of the
        // overload of Console.WriteLine to be used in Hello.
        Type[] writeStringArgs = {typeof(string)};
        // Get the overload of Console.WriteLine that has one
        // String parameter.
        MethodInfo writeString =
            typeof(Console).GetMethod("WriteLine", writeStringArgs);

        // Get an ILGenerator and emit a body for the dynamic method.
        ILGenerator il = hello.GetILGenerator();
        // Load the first argument, which is a string, onto the stack.
        il.Emit(OpCodes.Ldarg_0);
        // Call the overload of Console.WriteLine that prints a string.
        il.EmitCall(OpCodes.Call, writeString, null);
        // The Hello method returns the value of the second argument;
        // to do this, load the second argument onto the stack and return.
```

```
        il.Emit(OpCodes.Ldarg_1);
        il.Emit(OpCodes.Ret);

        // Create a delegate that represents the dynamic method. This
        // action completes the method. Further attempts to change the
        // method are ignored and no exception is thrown.
        HelloInvoker hi =
            (HelloInvoker) hello.CreateDelegate(typeof(HelloInvoker));

        // Use the delegate to execute the dynamic method. Save and
        // print the return value.
        int retval = hi("\r\nHello, World!", 42);
        Console.WriteLine("Executing delegate hi(\"Hello, World!\", 42) returned
{0}",
            retval);

        // Do it again, with different arguments.
        retval = hi("\r\nHi, Mom!", 5280);
        Console.WriteLine("Executing delegate hi(\"Hi, Mom!\", 5280) returned {0}",
            retval);

        // Create an array of arguments to use with the Invoke method.
        object[] invokeArgs = {"\r\nHello, World!", 42};
        // Invoke the dynamic method using the arguments. This is much
        // slower than using the delegate, because you must create an
        // array to contain the arguments, and ValueType arguments
        // must be boxed.
        object objRet = hello.Invoke(null, invokeArgs);
        Console.WriteLine("hello.Invoke returned {0}", objRet);
    }
}
```

DynamicMethod(String, Type, Type[], Type) 생성자

이 생성자로 만든 동적 메서드는 `owner` 형식의 모든 멤버와, `owner`을(를) 포함하는 모듈의 다른 모든 형식의 public 및 `internal` (`Friend` Visual Basic에서) 멤버에 접근할 수 있습니다.

이 생성자는 메서드 특성 MethodAttributes.Public 및 MethodAttributes.Static호출 규칙을 CallingConventions.Standard지정하고 JIT(Just-In-Time) 표시 유형 검사를 건너뛰지 않습니다.

> ⓘ **참고 항목**
>
> 이전 버전과의 호환성을 위해, 이 생성자는 다음 두 조건이 모두 true일 때 SecurityPermission와 SecurityPermissionFlag.ControlEvidence 플래그를 요구합니다: `owner`가 호출 모듈이 아닌 다른 모듈에 있고, ReflectionPermissionFlag.MemberAccess 플래그와 함께 ReflectionPermission에 대한 요구가 실패한 경우. 요청 SecurityPermission 이 성공하면 작업이 허용됩니다.

예시

다음 코드 예제에서는 형식과 논리적으로 연결된 형식을 만듭니다 DynamicMethod . 이 연결은 해당 형식의 프라이빗 멤버에 대한 액세스 권한을 부여합니다.

이 코드 예제는 프라이빗 필드를 가진 클래스 `Example` 을 정의하고, 첫 번째 클래스에서 파생된 클래스 `DerivedFromExample`, Int32를 반환하고 `Example`, Int32 타입의 매개 변수를 갖는 형식의 대리자 `UseLikeStatic`, Int32를 반환하고 Int32 타입의 매개 변수를 갖는 형식의 대리자 `UseLikeInstance`를 정의합니다.

그런 다음 예제 코드는 인스턴스 DynamicMethod 의 프라이빗 필드를 변경하고 이전 값을 반환하는 형식을 만듭니다`Example`.

> ⓘ **참고 항목**
>
> 일반적으로 클래스의 내부 필드를 변경하는 것은 개체 지향 코딩 연습에 적합하지 않습니다.

예제 코드는 인스턴스 `Example` 를 만든 다음 두 대리자를 만듭니다. 첫 번째는 동적 메서드와 동일한 매개 변수를 포함하는 형식 `UseLikeStatic`입니다. 두 번째는 첫 번째 매개 변수(형식 `UseLikeInstance`)가 없는 형식 `Example`입니다. 이 대리자는 메서드 오버로드를 사용하여 CreateDelegate(Type, Object) 만들어집니다. 해당 메서드 오버로드의 두 번째 매개 변수는 인스턴스입니다 `Example`. 이 경우 방금 만든 인스턴스는 새로 만든 대리자에게 바인딩됩니다. 해당 대리자를 호출할 때마다 동적 메서드는 바인딩된 인스턴스 `Example`에서 작동합니다.

> ⓘ **참고 항목**
>
> 이것은 .NET Framework 2.0에 도입된 대리자 바인딩의 완화된 규칙의 예이며, Delegate.CreateDelegate 메서드의 새로운 오버로드와 함께 제공됩니다. 자세한 내용은 Delegate 클래스를 참조하세요.

`UseLikeStatic` 대리자가 호출하여 바인딩된 인스턴스 `Example` 를 `UseLikeInstance` 대리자를 통해 전달합니다. `UseLikeInstance` 그런 다음 두 대리자가 동일한 인스턴스`Example`에서 작동할 수 있도록 대리자가 호출됩니다. 내부 필드 값의 변경 내용은 각 호출 후에 표시됩니다. 마지막으로 대리자는 `UseLikeInstance` 인스턴스 `DerivedFromExample`에 바인딩되고 대리자 호출은 반복됩니다.

C#

```
using System;
using System.Reflection;
```

```
using System.Reflection.Emit;

// These classes are for demonstration purposes.
//
public class Example
{
    private int id = 0;
    public Example(int id)
    {
        this.id = id;
    }
    public int ID { get { return id; }}
}

public class DerivedFromExample : Example
{
    public DerivedFromExample(int id) : base(id) {}
}

// Two delegates are declared: UseLikeInstance treats the dynamic
// method as if it were an instance method, and UseLikeStatic
// treats the dynamic method in the ordinary fashion.
//
public delegate int UseLikeInstance(int newID);
public delegate int UseLikeStatic(Example ex, int newID);

public class Demo
{
    public static void Main()
    {
        // This dynamic method changes the private id field. It has
        // no name; it returns the old id value (return type int);
        // it takes two parameters, an instance of Example and
        // an int that is the new value of id; and it is declared
        // with Example as the owner type, so it can access all
        // members, public and private.
        //
        DynamicMethod changeID = new DynamicMethod(
            "",
            typeof(int),
            new Type[] { typeof(Example), typeof(int) },
            typeof(Example)
        );

        // Get a FieldInfo for the private field 'id'.
        FieldInfo fid = typeof(Example).GetField(
            "id",
            BindingFlags.NonPublic | BindingFlags.Instance
        );

        ILGenerator ilg = changeID.GetILGenerator();

        // Push the current value of the id field onto the
        // evaluation stack. It's an instance field, so load the
        // instance of Example before accessing the field.
```

```
        ilg.Emit(OpCodes.Ldarg_0);
        ilg.Emit(OpCodes.Ldfld, fid);

        // Load the instance of Example again, load the new value
        // of id, and store the new field value.
        ilg.Emit(OpCodes.Ldarg_0);
        ilg.Emit(OpCodes.Ldarg_1);
        ilg.Emit(OpCodes.Stfld, fid);

        // The original value of the id field is now the only
        // thing on the stack, so return from the call.
        ilg.Emit(OpCodes.Ret);

        // Create a delegate that uses changeID in the ordinary
        // way, as a static method that takes an instance of
        // Example and an int.
        //
        UseLikeStatic uls =
            (UseLikeStatic) changeID.CreateDelegate(
                typeof(UseLikeStatic)
            );

        // Create an instance of Example with an id of 42.
        //
        Example ex = new Example(42);

        // Create a delegate that is bound to the instance of
        // of Example. This is possible because the first
        // parameter of changeID is of type Example. The
        // delegate has all the parameters of changeID except
        // the first.
        UseLikeInstance uli =
            (UseLikeInstance) changeID.CreateDelegate(
                typeof(UseLikeInstance),
                ex
            );

        // First, change the value of id by calling changeID as
        // a static method, passing in the instance of Example.
        //
        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uls(ex, 1492)
        );

        // Change the value of id again using the delegate bound
        // to the instance of Example.
        //
        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uli(2700)
        );

        Console.WriteLine("Final value of id: {0}", ex.ID);
```

```
        // Now repeat the process with a class that derives
        // from Example.
        //
        DerivedFromExample dfex = new DerivedFromExample(71);

        uli = (UseLikeInstance) changeID.CreateDelegate(
                typeof(UseLikeInstance),
                dfex
            );

        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uls(dfex, 73)
        );
        Console.WriteLine(
            "Change the value of id; previous value: {0}",
            uli(79)
        );
        Console.WriteLine("Final value of id: {0}", dfex.ID);
    }
}

/* This code example produces the following output:

Change the value of id; previous value: 42
Change the value of id; previous value: 1492
Final value of id: 2700
Change the value of id; previous value: 71
Change the value of id; previous value: 73
Final value of id: 79
 */
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 12.

System.Reflection.Emit.DynamicMethod.GetILGenerator 메서드

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

GetILGenerator()

동적 메서드가 완료된 후 또는 CreateDelegate 메서드를 Invoke 호출하여 MSIL을 추가하려는 추가 시도는 무시됩니다. 예외가 던져지지 않습니다.

> ⓘ **참고 항목**
>
> 일부 완전 신뢰 시나리오에서도 동적 메서드에서 확인되지 않는 코드에 대한 제한이 있습니다. "비고에서 DynamicMethod에 대한 \"확인\" 섹션을 참조하세요."

예시

다음 코드 예제에서는 두 개의 매개 변수를 사용하는 동적 메서드를 만듭니다. 이 예제에서는 첫 번째 매개 변수를 콘솔에 출력하는 간단한 함수 본문을 내보내고 두 번째 매개 변수를 메서드의 반환 값으로 사용합니다. 이 예제는 대리자를 생성하여 메서드를 완성한 후, 다른 매개변수로 대리자를 호출하고, 마지막으로 Invoke 메서드를 사용하여 동적 메서드를 호출합니다.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Test
{
    // Declare a delegate that will be used to execute the completed
    // dynamic method.
    private delegate int HelloInvoker(string msg, int ret);

    public static void Main()
    {
        // Create an array that specifies the types of the parameters
        // of the dynamic method. This method has a string parameter
        // and an int parameter.
        Type[] helloArgs = {typeof(string), typeof(int)};
```

```
        // Create a dynamic method with the name "Hello", a return type
        // of int, and two parameters whose types are specified by the
        // array helloArgs. Create the method in the module that
        // defines the Test class.
        DynamicMethod hello = new DynamicMethod("Hello",
            typeof(int),
            helloArgs,
            typeof(Test).Module);

        // Create an array that specifies the parameter types of the
        // overload of Console.WriteLine to be used in Hello.
        Type[] writeStringArgs = {typeof(string)};
        // Get the overload of Console.WriteLine that has one
        // String parameter.
        MethodInfo writeString =
            typeof(Console).GetMethod("WriteLine", writeStringArgs);

        // Get an ILGenerator and emit a body for the dynamic method.
        ILGenerator il = hello.GetILGenerator();
        // Load the first argument, which is a string, onto the stack.
        il.Emit(OpCodes.Ldarg_0);
        // Call the overload of Console.WriteLine that prints a string.
        il.EmitCall(OpCodes.Call, writeString, null);
        // The Hello method returns the value of the second argument;
        // to do this, load the second argument onto the stack and return.
        il.Emit(OpCodes.Ldarg_1);
        il.Emit(OpCodes.Ret);

        // Create a delegate that represents the dynamic method. This
        // action completes the method. Further attempts to change the
        // method are ignored and no exception is thrown.
        HelloInvoker hi =
            (HelloInvoker) hello.CreateDelegate(typeof(HelloInvoker));

        // Use the delegate to execute the dynamic method. Save and
        // print the return value.
        int retval = hi("\r\nHello, World!", 42);
        Console.WriteLine("Executing delegate hi(\"Hello, World!\", 42) returned
{0}",
            retval);

        // Do it again, with different arguments.
        retval = hi("\r\nHi, Mom!", 5280);
        Console.WriteLine("Executing delegate hi(\"Hi, Mom!\", 5280) returned {0}",
            retval);

        // Create an array of arguments to use with the Invoke method.
        object[] invokeArgs = {"\r\nHello, World!", 42};
        // Invoke the dynamic method using the arguments. This is much
        // slower than using the delegate, because you must create an
        // array to contain the arguments, and ValueType arguments
        // must be boxed.
        object objRet = hello.Invoke(null, invokeArgs);
        Console.WriteLine("hello.Invoke returned {0}", objRet);
```

```
    }
}
```

GetILGenerator(Int32)

동적 메서드가 완료된 후 또는 CreateDelegate 메서드를 Invoke 호출하여 MSIL을 추가하려는 추가 시도는 무시됩니다. 예외가 던져지지 않습니다.

> ⓘ **참고 항목**
>
> 일부 완전 신뢰 시나리오에서도 동적 메서드에서 확인되지 않는 코드에 대한 제한이 있습니다. "비고에서 DynamicMethod에 대한 \"확인\" 섹션을 참조하세요."

예시

다음 코드 예제에서는 이 메서드 오버로드를 보여 줍니다. 이 코드 예제는 클래스에 제공된 더 큰 예제의 DynamicMethod 일부입니다.

C#

```
// Create an array that specifies the parameter types of the
// overload of Console.WriteLine to be used in Hello.
Type[] writeStringArgs = {typeof(string)};
// Get the overload of Console.WriteLine that has one
// String parameter.
MethodInfo writeString = typeof(Console).GetMethod("WriteLine",
    writeStringArgs);

// Get an ILGenerator and emit a body for the dynamic method,
// using a stream size larger than the IL that will be
// emitted.
ILGenerator il = hello.GetILGenerator(256);
// Load the first argument, which is a string, onto the stack.
il.Emit(OpCodes.Ldarg_0);
// Call the overload of Console.WriteLine that prints a string.
il.EmitCall(OpCodes.Call, writeString, null);
// The Hello method returns the value of the second argument;
// to do this, load the second argument onto the stack and return.
il.Emit(OpCodes.Ldarg_1);
il.Emit(OpCodes.Ret);
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 24.

System.Reflection.Emit.DynamicMethod.Invoke 메서드

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

나열된 예외 외에도, 동적 메서드가 던진 모든 예외를 catch하도록 호출 코드를 준비해야 합니다.

메서드에서 만든 대리자를 사용하여 동적 메서드를 CreateDelegate 실행하는 것이 메서드를 Invoke 사용하여 실행하는 것보다 더 효율적입니다.

Invoke 메서드 또는 메서드를 호출하면 CreateDelegate 동적 메서드가 완료됩니다. 매개 변수 정의를 수정하거나 MSIL(Microsoft 중간 언어)을 더 내보내는 등 동적 메서드를 변경하려는 추가 시도는 무시됩니다. 예외가 throw되지 않습니다.

모든 동적 메서드는 정적이므로 `obj` 매개 변수는 항상 무시됩니다. 동적 메서드를 인스턴스 메서드인 것처럼 처리하려면 개체 인스턴스를 CreateDelegate(Type, Object) 사용하는 오버로드를 사용합니다.

동적 메서드에 매개 변수가 없으면 값 `parameters` 은 .이어야 `null`합니다. 그렇지 않으면 매개 변수 배열의 요소 수, 형식 및 순서가 동적 메서드의 매개 변수 수, 형식 및 순서와 동일해야 합니다.

> ⓘ **참고 항목**
>
> 이 메서드 오버로드는 Invoke(Object, Object[]) 클래스에서 상속된 MethodBase 메서드 오버로드에 의해 호출되므로, 앞의 설명이 두 오버로드에 모두 적용됩니다.

이 메서드는 직접 권한을 요구하지는 않지만 동적 메서드를 호출하면 메서드에 따라 보안 요구가 발생할 수 있습니다. 예를 들어, `restrictedSkipVisibility` 매개 변수가 `false`로 설정된 채로 생성된 익명으로 호스팅되는 동적 메서드에 대해서는 어떠한 요구도 없습니다. 반면, 대상 어셈블리의 숨겨진 멤버에 액세스할 수 있도록 `restrictedSkipVisibility`로 설정된 `true` 메서드를 만드는 경우, 이 메서드는 ReflectionPermission 플래그와 함께 대상 어셈블리의 사용 권한에 대한 요구를 발생시킵니다.

예시

다음 코드 예제에서는 US-English 문화권을 사용하여 정확한 바인딩으로 동적 메서드를 호출합니다. 이 코드 예제는 클래스에 제공된 더 큰 예제의 DynamicMethod 일부입니다.

C#

```
Console.WriteLine("\r\nUse the Invoke method to execute the dynamic method:");
// Create an array of arguments to use with the Invoke method.
object[] invokeArgs = {"\r\nHello, World!", 42};
// Invoke the dynamic method using the arguments. This is much
// slower than using the delegate, because you must create an
// array to contain the arguments, and value-type arguments
// must be boxed.
object objRet = hello.Invoke(null, BindingFlags.ExactBinding, null, invokeArgs, new
CultureInfo("en-us"));
Console.WriteLine("hello.Invoke returned: " + objRet);
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 24.

System.Reflection.Emit.DynamicMethod.IsSecurity* 보안 속성

> **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

> **참고 항목**
>
> 이 문서의 설명은 .NET Framework에만 적용되며 최신 .NET 적용되지 않습니다.

, IsSecurityCritical및 IsSecuritySafeCritical 속성은 IsSecurityTransparentCLR(공용 언어 런타임)에 의해 결정된 동적 메서드의 투명도 수준을 보고합니다. 이러한 속성의 조합은 다음 표에 나와 있습니다.

테이블 확장

보안 수준	IsSecurityCritical	보안은 안전하게 중요합니까?	IsSecurityTransparent
중요한	`true`	`false`	`false`
안전 중요	`true`	`true`	`false`
투명한	`false`	`false`	`true`

이러한 속성을 사용하는 것은 어셈블리 및 해당 형식의 보안 주석을 검사하고, 현재 신뢰 수준을 확인하고, 런타임의 규칙을 복제하는 것보다 훨씬 간단합니다.

동적 메서드의 투명도는 연결된 모듈에 따라 달라집니다. 동적 메서드가 모듈이 아닌 형식과 연결된 경우 해당 투명도는 형식이 포함된 모듈에 따라 달라집니다. 동적 메서드에는 보안 주석이 없으므로 연결된 모듈에 대한 기본 투명도가 할당됩니다.

- 익명으로 호스트되는 동적 메서드는 포함된 시스템 제공 모듈이 투명하기 때문에 항상 투명합니다.
- 신뢰할 수 있는 어셈블리(즉, 전역 어셈블리 캐시에 설치된 강력한 이름의 어셈블리)와 연결된 동적 메서드의 투명도는 다음 표에 설명되어 있습니다.

테이블 확장

어셈블리 어노테이션	수준 1 투명도	수준 2 투명도
완전 투명	투명한	투명한
완전 위험	중요한	중요한
혼합 투명도	투명한	투명한
보안에 구애받지 않습니다.	안전 필수	중요한

예를 들어 수준 2 혼합 투명도가 있는 mscorlib.dll형식과 동적 메서드를 연결하는 경우 동적 메서드는 투명하며 중요한 코드를 실행할 수 없습니다. 투명도 수준에 대한 자세한 내용은 Security-Transparent Code, Level 1 및 Security-Transparent Code 수준 2를 참조하세요.

> ⓘ **참고 항목**
>
> 동적 메서드를 System.dll같이 보안에 구애받지 않는 신뢰할 수 있는 수준 1 어셈블리의 모듈과 연결해도 신뢰 상승이 허용되지 않습니다. 동적 메서드를 호출하는 코드의 권한 부여 집합에 System.dll의 권한 부여 집합이 포함되어 있지 않은 경우(즉, 완전한 신뢰를 의미함) 동적 메서드 호출 시 예외가 발생합니다.

- 부분적으로 신뢰할 수 있는 어셈블리와 연결된 동적 메서드의 투명도는 어셈블리가 로드되는 방식에 따라 달라집니다. 어셈블리가 부분 신뢰(예: 샌드박스 애플리케이션 도메인)로 로드되는 경우 런타임은 어셈블리의 보안 주석을 무시합니다. 어셈블리와 동적 메서드를 포함한 모든 형식 및 멤버는 투명하게 처리됩니다. 런타임은 부분 신뢰 어셈블리가 완전 신뢰로 로드되는 경우에만 보안 주석에 주의를 기울입니다(예: 데스크톱 애플리케이션의 기본 애플리케이션 도메인으로). 이 경우 런타임은 어셈블리의 주석에 따라 동적 메서드에 메서드의 기본 투명도를 할당합니다.

리플렉션 내보내기 및 투명성에 대한 자세한 내용은 리플렉션 내보내기의 보안 문제를 참조하세요. 투명성에 대한 자세한 내용은 보안 변경 내용을 참조하세요.

Last updated on 2026. 03. 11.

System.Reflection.Emit.DynamicILInfo 클래스

> ⓘ **참고 항목**
>
> 이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

자신만의 MSIL 생성기를 작성하려면 DynamicILInfo 클래스를 사용하고 ILGenerator를 사용하지 마십시오.

다른 형식, 호출 메서드, 액세스 필드 또는 참조 형식의 인스턴스를 만들려면 생성하는 MSIL에 해당 엔터티에 대한 토큰이 포함되어야 합니다. DynamicILInfo 클래스는 현재 GetTokenFor 범위에서 유효한 토큰을 반환하는 DynamicILInfo 메서드의 여러 오버로드를 제공합니다. 예를 들어, 메서드의 Console.WriteLine 오버로드를 호출해야 하는 경우, 오버로드에 대한 RuntimeMethodHandle를 획득하여 GetTokenFor 메서드에 전달함으로써 MSIL에 포함할 토큰을 얻을 수 있습니다.

지역 변수 서명, 예외 및 코드 본문에 대한 Byte 배열을 만든 후에는 SetCode, SetExceptions, 및 SetLocalSignature 메서드를 사용하여 DynamicMethod 개체와 연결된 DynamicILInfo에 배열을 삽입할 수 있습니다.

사용자 고유의 메타데이터 및 MSIL을 생성하려면 CLI(공용 언어 인프라) 설명서, 특히 "파티션 II: 메타데이터 정의 및 의미 체계" 및 "파티션 III: CIL 명령 집합"에 대해 잘 알고 있어야 합니다. 자세한 내용은 ECMA 335 CLI(공용 언어 인프라)를 참조하세요.

> ⓘ **참고 항목**
>
> 대리자 생성자를 직접 호출하여 다른 동적 메서드에 대리자를 만드는 코드를 생성하는 데 사용하지 DynamicILInfo 마세요. 대신 메서드를 CreateDelegate 사용하여 대리자를 만듭니다. 대리자 생성자를 사용하여 만든 대리자에는 대상 동적 메서드에 대한 참조가 없습니다. 대리자가 여전히 사용 중일 때 동적 메서드가 가비지 수집에 의해 회수될 수 있습니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 02. 24.

System.Reflection.Emit.MethodBuilder 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

이 MethodBuilder 클래스는 이름, 특성, 서명 및 메서드 본문을 포함하여 CIL(공용 중간 언어)의 메서드를 완전히 설명하는 데 사용됩니다. 클래스와 함께 TypeBuilder 런타임에 클래스를 만드는 데 사용됩니다.

리플렉션 내보내기를 사용하여 전역 메서드를 정의하고 메서드를 형식 멤버로 정의할 수 있습니다. API는 메서드를 정의하고 MethodBuilder 개체를 반환합니다.

전역 메서드

ModuleBuilder.DefineGlobalMethod 메서드를 사용하여 전역 메서드를 정의하면 `MethodBuilder` 개체를 반환합니다.

전역 메서드는 정적이어야 합니다. 동적 모듈에 전역 메서드 ModuleBuilder.CreateGlobalFunctions 가 포함된 경우 공용 언어 런타임이 모든 전역 함수가 정의될 때까지 동적 모듈 수정을 연기하기 때문에 동적 모듈 또는 포함된 동적 어셈블리를 유지하기 전에 메서드를 호출해야 합니다.

전역 네이티브 메서드는 ModuleBuilder.DefinePInvokeMethod 메서드를 사용하여 정의됩니다. PInvoke(플랫폼 호출) 메서드는 추상 또는 가상으로 선언해서는 안 됩니다. 런타임은 플랫폼 호출 메서드의 특성을 설정합니다 MethodAttributes.PinvokeImpl .

형식의 멤버인 메서드

메서드는 TypeBuilder.DefineMethod 메서드를 사용하여 MethodBuilder 객체를 반환하는 유형 멤버로 정의됩니다.

이 DefineParameter 메서드는 매개 변수의 이름 및 매개 변수 특성 또는 반환 값을 설정하는 데 사용됩니다. 이 메서드에서 반환된 개체는 ParameterBuilder 매개 변수 또는 반환 값을 나타냅니다. 개체를 ParameterBuilder 사용하여 마샬링을 설정하고, 상수 값을 설정하고, 사용자 지정 특성을 적용할 수 있습니다.

특성

열거형의 멤버는 MethodAttributes 동적 메서드의 정확한 문자를 정의합니다.

- 정적 메서드는 MethodAttributes.Static 속성을 사용하여 지정됩니다.
- 최종 메서드(재정의할 수 없는 메서드)는 특성을 사용하여 MethodAttributes.Final 지정됩니다.
- 가상 메서드는 특성 MethodAttributes.Virtual을(를) 사용하여 지정됩니다.
- 추상 메서드는 MethodAttributes.Abstract 특성을 사용하여 지정됩니다.
- 여러 특성에 따라 메서드 표시 여부가 결정됩니다. 열거형에 대한 MethodAttributes 설명을 참조하세요.
- 오버로드된 연산자를 구현하는 메서드는 MethodAttributes.SpecialName 특성을 설정해야 합니다.
- 종료자는 MethodAttributes.SpecialName 특성을 설정해야 합니다.

알려진 문제

- MethodBuilder는 MethodInfo에서 파생되었지만, MethodInfo 클래스에 정의된 추상 메서드 중 일부는 MethodBuilder에서 완전히 구현되지 않았습니다. MethodBuilder 메서드는 NotSupportedException을 던집니다. 예를 들어 메서드가 MethodBuilder.Invoke 완전히 구현되지 않았습니다. Type.GetType 또는 Assembly.GetType 메서드를 사용하여 바깥쪽 형식을 검색함으로써 이러한 메서드를 반영할 수 있습니다.
- 사용자 지정 변형자가 지원됩니다.

.NET의 지속형 동적 어셈블리

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

구현은 이식되지 않은 Windows 관련 네이티브 코드에 크게 의존하기 때문에 AssemblyBuilder.Save API는 원래 .NET(Core)로 이식되지 않았습니다. .NET 9는 저장을 지원하는 완전 관리 `Reflection.Emit` 구현을 제공하는 PersistedAssemblyBuilder 클래스를 추가했습니다. 이 구현은 기존 런타임별 `Reflection.Emit` 구현에 종속되지 않습니다. 즉, 이제 .NET에는 *실행 가능* 하고 지속 가능한 두 가지 구현 *이* 있습니다. 지속형 어셈블리를 실행하려면 먼저 메모리 스트림 또는 파일에 저장한 다음 다시 로드합니다.

`PersistedAssemblyBuilder`전에 생성된 어셈블리만 실행하고 저장할 수 없습니다. 어셈블리가 메모리 내 전용이므로 디버그하기가 어려웠습니다. 동적 어셈블리를 파일에 저장할 경우의 장점은 다음과 같습니다.

- ILVerify와 같은 도구를 사용하여 생성된 어셈블리를 확인하거나 디컴파일하고 ILSpy와 같은 도구를 사용하여 수동으로 검사할 수 있습니다.
- 저장된 어셈블리를 다시 컴파일할 필요 없이 직접 로드할 수 있으므로 애플리케이션 시작 시간이 단축될 수 있습니다.

`PersistedAssemblyBuilder` 인스턴스를 만들려면 PersistedAssemblyBuilder(AssemblyName, Assembly, IEnumerable<CustomAttributeBuilder>) 생성자를 사용합니다. `coreAssembly` 매개 변수는 기본 런타임 형식을 확인하는 데 사용되며 참조 어셈블리 버전 관리 확인에 사용할 수 있습니다.

- `Reflection.Emit`이(가) 컴파일러가 실행 중인 런타임 버전과 동일한 런타임 버전에서만 실행되는 어셈블리를 생성하는 데 사용되는 경우(일반적으로 인프로세스에서), 핵심 어셈블리는 단순히 `typeof(object).Assembly`이 될 수 있습니다. 다음 예제에서는 어셈블리를 만들고 스트림에 저장하고 현재 런타임 어셈블리를 사용하여 실행하는 방법을 보여 줍니다.

C#

```
public static void CreateSaveAndRunAssembly()
{
    PersistedAssemblyBuilder ab = new(new AssemblyName("MyAssembly"),
typeof(object).Assembly);
    ModuleBuilder mob = ab.DefineDynamicModule("MyModule");
    TypeBuilder tb = mob.DefineType(
        "MyType",
        TypeAttributes.Public | TypeAttributes.Class);
    MethodBuilder meb = tb.DefineMethod(
        "SumMethod",
        MethodAttributes.Public | MethodAttributes.Static,
        typeof(int), [typeof(int), typeof(int)]);
    ILGenerator il = meb.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
```

```
    il.Emit(OpCodes.Ldarg_1);
    il.Emit(OpCodes.Add);
    il.Emit(OpCodes.Ret);

    tb.CreateType();

    using var stream = new MemoryStream();
    ab.Save(stream);  // Or pass filename to save into a file.
    stream.Seek(0, SeekOrigin.Begin);
    Assembly assembly = AssemblyLoadContext.Default.LoadFromStream(stream);
    MethodInfo method = assembly.GetType("MyType").GetMethod("SumMethod");
    Console.WriteLine(method.Invoke(null, [5, 10]));
}
```

- `Reflection.Emit` 특정 TFM을 대상으로 하는 어셈블리를 생성하는 데 사용되는 경우, `MetadataLoadContext`을 사용하여 지정된 TFM에 대한 레퍼런스 어셈블리를 열고, 에 대해 `coreAssembly` 속성의 값을 사용합니다. 이 값을 사용하면 생성기가 하나의 .NET 런타임 버전에서 실행되고 다른 .NET 런타임 버전을 대상으로 할 수 있습니다. 핵심 형식을 참조할 때 `MetadataLoadContext` 인스턴스에서 반환된 형식을 사용해야 합니다. 예를 들어, `typeof(int)`대신 `System.Int32`에서 이름으로 `MetadataLoadContext.CoreAssembly` 유형을 찾습니다.

  C#

  ```
  public static void CreatePersistedAssemblyBuilderCoreAssemblyWithMetadataLoadContext(string refAssembliesPath)
  {
      PathAssemblyResolver resolver = new(Directory.GetFiles(refAssembliesPath, "*.dll"));
      using MetadataLoadContext context = new(resolver);
      Assembly coreAssembly = context.CoreAssembly;
      PersistedAssemblyBuilder ab = new(new AssemblyName("MyDynamicAssembly"), coreAssembly);
      TypeBuilder typeBuilder = ab.DefineDynamicModule("MyModule").DefineType("Test", TypeAttributes.Public);
      MethodBuilder methodBuilder = typeBuilder.DefineMethod("Method", MethodAttributes.Public, coreAssembly.GetType(typeof(int).FullName), Type.EmptyTypes);
      // .. add members and save the assembly
  }
  ```

실행 파일의 진입점 설정

실행 파일의 진입점을 설정하거나 어셈블리 파일에 대한 다른 옵션을 설정하려면 `public MetadataBuilder GenerateMetadata(out BlobBuilder ilStream, out BlobBuilder`

`mappedFieldData)` 메서드를 호출하고 채워진 메타데이터를 사용하여 원하는 옵션을 사용하여 어셈블리를 생성할 수 있습니다. 예를 들면 다음과 같습니다.

C#

```
public static void SetEntryPoint()
{
    PersistedAssemblyBuilder ab = new(new AssemblyName("MyAssembly"),
typeof(object).Assembly);
    TypeBuilder tb = ab.DefineDynamicModule("MyModule").DefineType("MyType",
TypeAttributes.Public | TypeAttributes.Class);
    // ...
    MethodBuilder entryPoint = tb.DefineMethod("Main", MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.Static);
    ILGenerator il2 = entryPoint.GetILGenerator();
    // ...
    il2.Emit(OpCodes.Ret);
    tb.CreateType();

    MetadataBuilder metadataBuilder = ab.GenerateMetadata(out BlobBuilder ilStream,
out BlobBuilder fieldData);

    ManagedPEBuilder peBuilder = new(
                    header: PEHeaderBuilder.CreateExecutableHeader(),
                    metadataRootBuilder: new MetadataRootBuilder(metadataBuilder),
                    ilStream: ilStream,
                    mappedFieldData: fieldData,
                    entryPoint:
MetadataTokens.MethodDefinitionHandle(entryPoint.MetadataToken));

    BlobBuilder peBlob = new();
    peBuilder.Serialize(peBlob);

    // Create the executable:
    using FileStream fileStream = new("MyAssembly.exe", FileMode.Create,
FileAccess.Write);
    peBlob.WriteContentTo(fileStream);
}
```

기호 내보내기 및 PDB 생성

기호 메타데이터는 `pdbBuilder` 인스턴스에서 GenerateMetadata(BlobBuilder, BlobBuilder) 메서드를 호출할 때 `PersistedAssemblyBuilder` out 매개 변수로 채워집니다. 이식 가능한 PDB를 사용하여 어셈블리를 만들려면 다음을 수행합니다.

1. ISymbolDocumentWriter 메서드를 사용하여 ModuleBuilder.DefineDocument(String, Guid, Guid, Guid) 인스턴스를 만듭니다. 메서드의 IL을 내보내는 동안 해당 기호 정보도 내보냅니다.

2. PortablePdbBuilder 메서드에서 생성된 `pdbBuilder` 인스턴스를 사용하여 GenerateMetadata(BlobBuilder, BlobBuilder) 인스턴스를 만듭니다.
3. `PortablePdbBuilder`을(를) Blob로 직렬화하고, (독립 실행형 PDB를 생성하는 경우에만) `Blob`를 PDB 파일 스트림에 기록합니다.
4. DebugDirectoryBuilder 인스턴스를 만들고 DebugDirectoryBuilder.AddCodeViewEntry(독립 실행형 PDB) 또는 DebugDirectoryBuilder.AddEmbeddedPortablePdbEntry추가합니다.
5. `debugDirectoryBuilder` 인스턴스를 만들 때 선택적 PEBuilder 인수를 설정합니다.

다음 예제에서는 기호 정보를 내보내고 PDB 파일을 생성하는 방법을 보여 줍니다.

C#

```
static void GenerateAssemblyWithPdb()
{
    PersistedAssemblyBuilder ab = new PersistedAssemblyBuilder(new
AssemblyName("MyAssembly"), typeof(object).Assembly);
    ModuleBuilder mb = ab.DefineDynamicModule("MyModule");
    TypeBuilder tb = mb.DefineType("MyType", TypeAttributes.Public |
TypeAttributes.Class);
    MethodBuilder mb1 = tb.DefineMethod("SumMethod", MethodAttributes.Public |
MethodAttributes.Static, typeof(int), [typeof(int), typeof(int)]);
    ISymbolDocumentWriter srcDoc = mb.DefineDocument("MySourceFile.cs",
SymLanguageType.CSharp);
    ILGenerator il = mb1.GetILGenerator();
    LocalBuilder local = il.DeclareLocal(typeof(int));
    local.SetLocalSymInfo("myLocal");
    il.MarkSequencePoint(srcDoc, 7, 0, 7, 11);
    ...
    il.Emit(OpCodes.Ret);

    MethodBuilder entryPoint = tb.DefineMethod("Main", MethodAttributes.HideBySig |
MethodAttributes.Public | MethodAttributes.Static);
    ILGenerator il2 = entryPoint.GetILGenerator();
    il2.BeginScope();
    ...
    il2.EndScope();
    ...
    tb.CreateType();

    MetadataBuilder metadataBuilder = ab.GenerateMetadata(out BlobBuilder ilStream,
out _, out MetadataBuilder pdbBuilder);
    MethodDefinitionHandle entryPointHandle =
MetadataTokens.MethodDefinitionHandle(entryPoint.MetadataToken);
    DebugDirectoryBuilder debugDirectoryBuilder = GeneratePdb(pdbBuilder,
metadataBuilder.GetRowCounts(), entryPointHandle);

    ManagedPEBuilder peBuilder = new ManagedPEBuilder(
                    header: new PEHeaderBuilder(imageCharacteristics:
Characteristics.ExecutableImage, subsystem: Subsystem.WindowsCui),
                    metadataRootBuilder: new MetadataRootBuilder(metadataBuilder),
                    ilStream: ilStream,
```

```
                    debugDirectoryBuilder: debugDirectoryBuilder,
                    entryPoint: entryPointHandle);

    BlobBuilder peBlob = new BlobBuilder();
    peBuilder.Serialize(peBlob);

    using var fileStream = new FileStream("MyAssembly.exe", FileMode.Create,
FileAccess.Write);
    peBlob.WriteContentTo(fileStream);
}

static DebugDirectoryBuilder GeneratePdb(MetadataBuilder pdbBuilder,
ImmutableArray<int> rowCounts, MethodDefinitionHandle entryPointHandle)
{
    BlobBuilder portablePdbBlob = new BlobBuilder();
    PortablePdbBuilder portablePdbBuilder = new PortablePdbBuilder(pdbBuilder,
rowCounts, entryPointHandle);
    BlobContentId pdbContentId = portablePdbBuilder.Serialize(portablePdbBlob);
    // In case saving PDB to a file
    using FileStream fileStream = new FileStream("MyAssemblyEmbeddedSource.pdb",
FileMode.Create, FileAccess.Write);
    portablePdbBlob.WriteContentTo(fileStream);

    DebugDirectoryBuilder debugDirectoryBuilder = new DebugDirectoryBuilder();
    debugDirectoryBuilder.AddCodeViewEntry("MyAssemblyEmbeddedSource.pdb",
pdbContentId, portablePdbBuilder.FormatVersion);
    // In case embedded in PE:
    // debugDirectoryBuilder.AddEmbeddedPortablePdbEntry(portablePdbBlob,
portablePdbBuilder.FormatVersion);
    return debugDirectoryBuilder;
}
```

먼저 CustomDebugInformation 인스턴스에서 MetadataBuilder.AddCustomDebugInformation(EntityHandle, GuidHandle, BlobHandle) 메서드를 호출하여 원본 포함 및 원본 인덱싱 고급 PDB 정보를 추가한 후, `pdbBuilder`을 추가할 수 있습니다.

C#

```
private static void EmbedSource(MetadataBuilder pdbBuilder)
{
    byte[] sourceBytes = File.ReadAllBytes("MySourceFile2.cs");
    BlobBuilder sourceBlob = new BlobBuilder();
    sourceBlob.WriteBytes(sourceBytes);
    pdbBuilder.AddCustomDebugInformation(MetadataTokens.DocumentHandle(1),
        pdbBuilder.GetOrAddGuid(new Guid("0E8A571B-6926-466E-B4AD-8AB04611F5FE")),
pdbBuilder.GetOrAddBlob(sourceBlob));
}
```

PersistedAssemblyBuilder를 사용하여 리소스 추가

MetadataBuilder.AddManifestResource(ManifestResourceAttributes, StringHandle, EntityHandle, UInt32) 호출하여 필요한 만큼 리소스를 추가할 수 있습니다. 스트림은 하나의 BlobBuilder로 연결되어 ManagedPEBuilder 인수에 전달되어야 합니다. 다음 예제에서는 리소스를 만들고 만든 어셈블리에 연결하는 방법을 보여 줍니다.

C#

```
public static void SetResource()
{
    PersistedAssemblyBuilder ab = new(new AssemblyName("MyAssembly"),
typeof(object).Assembly);
    ab.DefineDynamicModule("MyModule");
    MetadataBuilder metadata = ab.GenerateMetadata(out BlobBuilder ilStream, out _);

    using MemoryStream stream = new();
    ResourceWriter myResourceWriter = new(stream);
    myResourceWriter.AddResource("AddResource 1", "First added resource");
    myResourceWriter.AddResource("AddResource 2", "Second added resource");
    myResourceWriter.AddResource("AddResource 3", "Third added resource");
    myResourceWriter.Close();

    byte[] data = stream.ToArray();
    BlobBuilder resourceBlob = new();
    resourceBlob.WriteInt32(data.Length);
    resourceBlob.WriteBytes(data);

    metadata.AddManifestResource(
        ManifestResourceAttributes.Public,
        metadata.GetOrAddString("MyResource.resources"),
        implementation: default,
        offset: 0);

    ManagedPEBuilder peBuilder = new(
                    header: PEHeaderBuilder.CreateLibraryHeader(),
                    metadataRootBuilder: new MetadataRootBuilder(metadata),
                    ilStream: ilStream,
                    managedResources: resourceBlob);

    BlobBuilder blob = new();
    peBuilder.Serialize(blob);

    // Create the assembly:
    using FileStream fileStream = new("MyAssemblyWithResource.dll", FileMode.Create,
FileAccess.Write);
    blob.WriteContentTo(fileStream);
}
```

다음 예제에서는 만든 어셈블리에서 리소스를 읽는 방법을 보여 줍니다.

C#

```
public static void ReadResource()
{
    Assembly readAssembly = Assembly.LoadFile(Path.Combine(
        Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location),
        "MyAssemblyWithResource.dll"));

    // Use ResourceManager.GetString() to read the resources.
    ResourceManager rm = new("MyResource", readAssembly);
    Console.WriteLine("Using ResourceManager.GetString():");
    Console.WriteLine($"{rm.GetString("AddResource 1",
CultureInfo.InvariantCulture)}");
    Console.WriteLine($"{rm.GetString("AddResource 2",
CultureInfo.InvariantCulture)}");
    Console.WriteLine($"{rm.GetString("AddResource 3",
CultureInfo.InvariantCulture)}");

    // Use ResourceSet to enumerate the resources.
    Console.WriteLine();
    Console.WriteLine("Using ResourceSet:");
    ResourceSet resourceSet = rm.GetResourceSet(CultureInfo.InvariantCulture,
createIfNotExists: true, tryParents: false);
    foreach (DictionaryEntry entry in resourceSet)
    {
        Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");
    }

    // Use ResourceReader to enumerate the resources.
    Console.WriteLine();
    Console.WriteLine("Using ResourceReader:");
    using Stream stream =
readAssembly.GetManifestResourceStream("MyResource.resources")!;
    using ResourceReader reader = new(stream);
    foreach (DictionaryEntry entry in reader)
    {
        Console.WriteLine($"Key: {entry.Key}, Value: {entry.Value}");
    }
}
```

ⓘ 참고 항목

모든 멤버에 대한 메타데이터 토큰은 Save 연산에서 설정됩니다. 생성된 형식 및 해당 멤버의 토큰은 기본값을 가지거나 예외를 발생시킬 수 있으니 저장하기 전에 사용하지 마세요. 생성되지 않고 참조되는 형식에 토큰을 사용하는 것이 안전합니다.

어셈블리를 내보내는 데 중요하지 않은 일부 API는 구현되지 않습니다. 예를 들어 `GetCustomAttributes()` 구현되지 않습니다. 런타임 구현을 사용하면 형식을 만든 후 해당 API를 사용할 수 있었습니다. 지속형 `AssemblyBuilder`의 경우에는 `NotSupportedException`을 던지거나 `NotImplementedException`를 사용합니다. 이러한 API가 필요한 시나리오가 있는 경우 dotnet/runtime 리포지토리 문제를 제출합니다.

어셈블리 파일을 생성하는 다른 방법은 MetadataBuilder참조하세요.

Last updated on 2026. 01. 22.

System.Reflection.Emit.TypeBuilder 클래스

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

TypeBuilder 는 런타임에서 동적 클래스 생성을 제어하는 데 사용되는 루트 클래스입니다. 클래스를 정의하고, 메서드와 필드를 추가하고, 모듈 내에 클래스를 만드는 데 사용되는 루틴 집합을 제공합니다. 동적 모듈에서 TypeBuilder 메서드를 호출하여 ModuleBuilder.DefineType 객체를 반환하는 새 TypeBuilder을 만들 수 있습니다.

리플렉션 내보내기에서는 형식을 정의하기 위한 다음 옵션을 제공합니다.

- 지정된 이름을 사용하여 클래스 또는 인터페이스를 정의합니다.
- 지정된 이름과 특성을 사용하여 클래스 또는 인터페이스를 정의합니다.
- 지정된 이름, 특성 및 기본 클래스를 사용하여 클래스를 정의합니다.
- 지정된 이름, 특성, 기본 클래스 및 클래스가 구현하는 인터페이스 집합을 사용하여 클래스를 정의합니다.
- 지정된 이름, 특성, 기본 클래스 및 압축 크기를 사용하여 클래스를 정의합니다.
- 지정된 이름, 특성, 기본 클래스 및 클래스 크기를 전체적으로 사용하여 클래스를 정의합니다.
- 지정된 이름, 특성, 기본 클래스, 압축 크기 및 클래스 크기를 전체적으로 사용하여 클래스를 정의합니다.

불완전한 형식을 나타내는 TypeBuilder 개체에 대한 배열 형식, 포인터 형식 또는 byref 형식을 만들려면 각각 MakeArrayType 메서드, MakePointerType 메서드, 또는 MakeByRefType 메서드를 사용합니다.

형식을 사용하려면 먼저 메서드를 TypeBuilder.CreateType 호출해야 합니다. **CreateType** 은 형식 만들기를 완료합니다. **CreateType**을 호출한 후 호출자는 메서드를 사용하여 Activator.CreateInstance 형식을 인스턴스화하고 메서드를 사용하여 Type.InvokeMember 형식의 멤버를 호출할 수 있습니다. **CreateType**이 호출된 후 형식의 구현을 변경하는 메서드를 호출하는 것은 오류입니다. 예를 들어 공용 언어 런타임은 호출자가 형식에 새 멤버를 추가하려고 하면 예외를 throw합니다.

클래스 이니셜라이저는 메서드를 TypeBuilder.DefineTypeInitializer 사용하여 만들어집니다. **DefineTypeInitializer** 는 개체를 ConstructorBuilder 반환합니다.

중첩 형식은 메서드 중 하나를 호출하여 정의됩니다 TypeBuilder.DefineNestedType .

특성

클래스는 TypeBuilder 열거형을 TypeAttributes 사용하여 만들 형식의 특성을 추가로 지정합니다.

- 인터페이스는 TypeAttributes.Interface 및 TypeAttributes.Abstract 특성을 사용하여 지정됩니다.
- 구체적인 클래스(확장할 수 없는 클래스)는 특성을 사용하여 TypeAttributes.Sealed 지정됩니다.
- 몇 가지 특성은 형식 표시 여부를 결정합니다. 열거형에 대한 TypeAttributes 설명을 참조하세요.
- 지정된 경우 TypeAttributes.SequentialLayout 클래스 로더는 메타데이터에서 읽는 순서대로 필드를 배치합니다. 클래스 로더는 지정된 압축 크기를 고려하지만 지정된 필드 오프셋은 무시합니다. 메타데이터는 필드 정의가 내보내는 순서를 유지합니다. 병합을 통해서도 메타데이터는 필드 정의의 순서를 다시 지정하지 않습니다. 로더는 지정된 필드 오프셋을 지정한 경우에만 TypeAttributes.ExplicitLayout 적용합니다.

알려진 문제

- 리플렉션 내보내기가 인터페이스를 구현하는 비추상 클래스가 인터페이스에 선언된 모든 메서드를 구현했는지 여부를 확인하지 않습니다. 그러나 클래스가 인터페이스에 선언된 모든 메서드를 구현하지 않는 경우 런타임은 클래스를 로드하지 않습니다.
- TypeBuilder는 Type에서 파생되었지만, Type 클래스에 정의된 추상 메서드 중 일부가 TypeBuilder 클래스에서 완전히 구현되지 않았습니다. 이러한 TypeBuilder 메서드에 대한 호출은 NotSupportedException 예외를 발생시킵니다. 원하는 기능은 Type.GetType 또는 Assembly.GetType을(를) 사용하여 생성된 형식을 검색하고, 검색된 형식을 반영하여 얻을 수 있습니다.

메타데이터 및 자체 설명 구성 요소

이전에 한 가지 언어로 작성된 소프트웨어 구성 요소(.exe 또는 .dll)는 다른 언어로 작성된 소프트웨어 구성 요소를 쉽게 사용할 수 없었습니다. COM은 이러한 문제를 해결하기 위한 단계를 제공했습니다. .NET은 컴파일러가 모든 모듈과 어셈블리에 추가 선언 정보를 내보낼 수 있도록 하여 구성 요소 상호 운용성을 훨씬 더 쉽게 만듭니다. 메타데이터라고 하는 이 정보는 구성 요소가 아무런 문제 없이 원만하게 상호 작용할 수 있도록 하는 데 도움이 됩니다.

메타데이터는 프로그램을 기술하는 이진 정보이며 공용 언어 런타임 PE 파일 또는 메모리에 저장됩니다. 코드를 PE 파일로 컴파일하면 메타데이터가 파일의 한 부분에 삽입되고 코드는 CIL(공용 중간 언어)로 변환되어 파일의 다른 부분에 삽입됩니다. 모듈 또는 어셈블리에서 정의되고 참조된 모든 형식과 멤버는 메타데이터 내에 기술됩니다. 코드를 실행하면 런타임은 메타데이터를 메모리로 로드한 다음 참조하여 해당 코드의 클래스, 멤버, 상속 등에 대한 정보를 검색합니다.

메타데이터는 언어와 무관하게 코드에 정의된 모든 형식과 멤버를 기술하며 메타데이터는 다음과 같은 정보를 저장합니다.

- 어셈블리 기술 내용
  - ID(이름, 버전, 문화권, 공개 키)
  - 내보낸 유형
  - 이 어셈블리가 종속된 다른 어셈블리
  - 실행하는 데 필요한 보안 권한
- 유형 설명
  - 이름, 표시 여부, 기본 클래스 및 구현된 인터페이스
  - 멤버(메서드, 필드, 속성, 이벤트, 중첩 형식)
- 특성.
  - 형식과 멤버를 수정하는 추가 설명적 요소

메타데이터의 이점

메타데이터는 더욱 간편한 프로그래밍 모델의 주요 요소이며 IDL(인터페이스 정의 언어) 파일, 헤더 파일 또는 구성 요소 참조의 외부 메서드를 사용하지 않아도 되게 합니다. 메타데이터를 사용하면 .NET 언어는 개발자와 사용자 모두 알지 못한 채 언어와 무관하게 자동으로 자신을 기술할 수 있습니다. 또한 특성을 사용하여 메타데이터를 확장할 수 있습니다. 메타데이터는 다음과 같은 중요한 이점을 제공합니다.

- 자기 설명 파일

공용 언어 런타임의 모듈과 어셈블리는 자동으로 기술됩니다. 모듈의 메타데이터에는 다른 모듈과 상호 작용하는 데 필요한 모든 요소가 있습니다. 메타데이터는 하나의 파일만 사용하여 정의와 구현을 모두 수행할 수 있도록 COM의 IDL 기능을 자동으로 제공합니다. 런타임 모듈과 어셈블리를 운영 체제에 등록할 필요도 없습니다. 결과적으로 런타임에 사용된 기술 내용은 항상 컴파일된 파일의 실제 코드를 반영하므로 애플리케이션의 안정성이 높아집니다.

- 언어 상호 운용성 및 간편해진 구성 요소 기반 디자인

  메타데이터는 다른 언어로 작성된 PE 파일에서 클래스를 상속하기 위해 컴파일된 코드에 대해 필요한 모든 정보를 제공하므로 명시적 마샬링을 고려하거나 사용자 지정 interop 코드를 사용할 필요 없이 관리되는 언어(공용 언어 런타임 기능이 있는 언어)로 작성된 클래스의 인스턴스를 만들 수 있습니다.

- 특성.

  .NET에서는 특성이라고 하는 특정 종류의 메타데이터를 컴파일된 파일에 선언할 수 있습니다. 특성은 .NET 전체에서 찾을 수 있으며 런타임에 프로그램이 동작하는 방식을 자세히 제어하는 데 사용됩니다. 또한 사용자 지정 특성을 사용하여 사용자 지정 메타데이터를 .NET 파일로 내보낼 수 있습니다. 자세한 내용은 특성을 참조하세요.

메타데이터 및 PE 파일 구조

메타데이터는 .NET PE(이식 가능) 파일의 한 섹션에 저장되고, CIL(공용 중간 언어)은 PE 파일의 다른 섹션에 저장됩니다. 파일의 메타데이터 부분에는 테이블 및 힙 데이터 구조가 있습니다. CIL 부분에는 PE 파일의 메타데이터 부분을 참조하는 CIL 및 메타데이터 토큰이 포함되어 있습니다. 예를 들어, 코드의 CIL을 보기 위해 IL 디스어셈블러(Ildasm.exe)와 같은 도구를 사용할 때 메타데이터 토큰이 발생할 수 있습니다.

메타데이터 테이블 및 힙

각 메타데이터 테이블은 프로그램 요소에 대한 정보를 보유합니다. 예를 들어, 한 메타데이터 테이블은 코드의 클래스를 나타내고 다른 테이블은 필드를 나타내는 식입니다. 코드에 클래스가 10개 있는 경우 클래스 테이블은 각 클래스마다 행 하나씩, 모두 10개의 행을 가집니다. 메타데이터 테이블은 다른 테이블과 힙을 참조합니다. 예를 들어, 클래스의 메타데이터 테이블은 메서드의 테이블을 참조합니다.

또한 메타데이터는 문자열, blob, 사용자 문자열 및 GUID라고 하는 네 가지 힙 구조에 정보를 저장합니다. 형식과 멤버의 이름을 지정하는 데 사용되는 모든 문자열은 문자열 힙에 저장됩니다. 예를 들어, 메서드 테이블은 특정 메서드의 이름을 직접 저장하지 않지만 문자열 힙에 저장된 메서드의 이름을 가리킵니다.

메타데이터 토큰

메타데이터 테이블의 각 행은 메타데이터 토큰에 의해 PE 파일의 CIL 부분에서 고유하게 식별됩니다. 메타데이터 토큰은 CIL에 유지되고 특정 메타데이터 테이블을 참조하는 포인터와 개념적으로 비슷합니다.

메타데이터 토큰은 4바이트 숫자입니다. 최상위 바이트는 특정 토큰이 참조하는 메타데이터 테이블(메서드, 형식 등)을 나타내고 나머지 3바이트는 메타데이터 테이블의 행 중에서 현재 나타내고 있는 프로그래밍 요소에 해당하는 행을 지정합니다. C#에서 메서드를 정의하여 PE 파일로 컴파일하면 PE 파일의 CIL 부분에 다음과 같은 메타데이터 토큰이 존재합니다.

`0x06000004`

최상위 바이트(`0x06`)는 이 토큰이 **MethodDef** 토큰임을 나타내고, 하위 3바이트(`000004`)는 공용 언어 런타임이 **MethodDef** 테이블의 넷째 행에서 이 메서드 정의를 기술하는 정보를 찾도록 지시합니다.

PE 파일 내의 메타데이터

프로그램이 공용 언어 런타임으로 컴파일되면 해당 프로그램은 세 부분으로 구성되는 PE 파일로 변환됩니다. 다음 표에서는 각 부분의 내용을 설명합니다.

테이블 확장

PE 섹션	PE 섹션의 내용
PE 헤더	PE 파일의 주 섹션 인덱스 및 진입점 주소를 저장합니다. 런타임은 이 정보를 사용하여 파일을 PE 파일로 식별하고 프로그램을 메모리로 로드하는 경우 실행 시작 위치를 결정합니다.
CIL 지침	코드를 구성하는 Microsoft Intermediate Language 지침(CIL)입니다. 대부분의 CIL 지침에는 메타데이터 토큰이 함께 제공됩니다.
메타데이터	메타데이터 테이블과 힙. 런타임은 이 섹션을 사용하여 코드의 모든 형식과 멤버에 대한 정보를 기록합니다. 또한 사용자 지정 특성과 보안 정보도 포함되어 있습니다.

메타데이터 런타임 사용

메타데이터 및 공용 언어 런타임의 역할을 더 잘 이해하려면 간단한 프로그램을 구성하고 메타데이터가 런타임 수명에 미치는 영향을 설명하는 것이 유용할 수 있습니다. 다음 코드 예제에서는 `MyApp`라는 클래스 내에서 두 가지 메서드를 보여 줍니다. `Main` 메서드는 프로그램 진입점이고 `Add` 메서드는 두 정수 인수의 합계를 반환합니다.

C#

```
using System;
public class MyApp
{
   public static int Main()
   {
      int ValueOne = 10;
      int ValueTwo = 20;
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
      return 0;
   }
   public static int Add(int One, int Two)
   {
      return (One + Two);
   }
}
```

코드를 실행하면 런타임은 모듈을 메모리로 로드하고 이 클래스의 메타데이터를 참조합니다. 로드되면 런타임은 메서드의 CIL(공용 중간 언어) 스트림을 광범위하게 분석하여 이를 빠른 네이티브 컴퓨터 명령으로 변환합니다. 런타임은 JIT(Just-In-Time) 컴파일러를 사용하여 CIL 명령어를 필요에 따라 하나의 메서드씩 네이티브 기계 코드로 변환합니다.

다음 예에서는 이전 코드의 `Main` 함수에서 생성된 CIL 부분을 보여 줍니다. IL 디스어셈블러(Ildasm.exe)를 사용하여 모든 .NET 애플리케이션에서 CIL 및 메타데이터를 볼 수 있습니다.

콘솔

```
.entrypoint
.maxstack  3
.locals ([0] int32 ValueOne,
         [1] int32 ValueTwo,
         [2] int32 V_2,
         [3] int32 V_3)
IL_0000:  ldc.i4.s   10
IL_0002:  stloc.0
IL_0003:  ldc.i4.s   20
IL_0005:  stloc.1
IL_0006:  ldstr      "The Value is: {0}"
IL_000b:  ldloc.0
IL_000c:  ldloc.1
IL_000d:  call int32 ConsoleApplication.MyApp::Add(int32,int32) /* 06000003 */
```

JIT 컴파일러는 전체 메서드에 대한 CIL을 읽고 이를 자세히 분석하여 해당 메서드에 대해 효과적인 네이티브 명령을 생성합니다. `IL_000d`에 `Add` 메서드에 대한 메타데이터 토큰(`/* 06000003 */`)이 있고 런타임은 이 토큰을 사용하여 **MethodDef** 테이블의 셋째 행을 참조합니다.

다음 표에서는 메서드를 나타내는 메타데이터 토큰에서 참조하는 `Add` 테이블의 부분을 보여 줍니다. 이 어셈블리에는 고유한 값을 가지는 다른 메타데이터 테이블도 있지만 여기서는 이 테이

블에 대해서만 설명합니다.

테이블 확장

행	RVA(Relevant Virtual Address)	ImplFlags	플래그	이름 (문자열 힙을 가리킴)	Signature(blob 힙(heap)을 가리킴.)
1	0x00002050	일리노이주 관리형	공공 사업 재사용 슬롯 특별한 이름 RTSpecialName .ctor	.ctor (생성자)	
2	0x00002058	일리노이주 관리형	공공 사업 정적 재사용 슬롯	기본	문자열
3	0x0000208c	일리노이주 관리형	공공 사업 정적 재사용 슬롯	추가	정수, 정수, 정수

테이블의 각 열에는 코드에 대한 중요한 정보가 있습니다. **RVA**열을 사용하여 런타임은 이 메서드를 정의하는 CIL의 시작 메모리 주소를 계산할 수 있습니다. **ImplFlags** 및 **Flags** 열은 메서드를 나타내는 비트마스크(예: 해당 메서드가 공용인지 아니면 전용인지를 나타냄)를 포함합니다. **Name** 열은 문자열 힙에서 메서드의 이름을 인덱싱하고 **Signature** 열은 blob 힙에서 메서드의 시그니처 정의를 인덱싱합니다.

런타임은 셋째 행의 **RVA** 열을 사용하여 원하는 오프셋 주소를 계산하고 이 주소를 JIT 컴파일러에 반환한 다음 새 주소로 계속합니다. JIT 컴파일러는 다른 메타데이터 토큰을 발견할 때까지 새 주소에서 CIL을 계속 처리하며 프로세스를 반복합니다.

메타데이터를 사용하면 런타임은 코드를 로드하여 네이티브 기계어 명령으로 처리하는 데 필요한 모든 정보에 액세스할 수 있습니다. 이런 방식으로 메타데이터는 자체 기술 파일 및 공용 형식 시스템과 함께 교차 언어 상속을 가능하게 합니다.

System.Reflection.PortableExecutable.DebugDirectoryEntryType 열거형

2025. 06. 17.

이 문서는 이 API에 대한 참조 설명서를 보충하는 추가 설명을 제공합니다.

열거형은 DebugDirectoryEntryType 디버깅 정보의 형식을 설명합니다 DebugDirectoryEntry.

개별 열거형 멤버와 관련된 사양은 다음을 참조하세요.

테이블 확장

회원	규격
`CodeView`	CodeView 디버그 디렉터리 항목(형식 2)
`EmbeddedPortablePdb`	포함된 이식 가능한 PDB 디버그 디렉터리 항목(유형 17)
`PdbChecksum`	PDB 체크섬 디버그 디렉터리 항목(유형 19)
`Reproducible`	결정적 디버그 디렉터리 항목 참조(유형 16)

DebugDirectoryEntryType.Reproducible

결정적 PE/COFF 파일을 생성한 도구는 파일의 전체 콘텐츠가 앰비언트 환경 변수(예: 현재 시간, 운영 체제, 도구를 실행하는 프로세스의 비트)가 아닌 도구에 제공된 문서화된 입력(예: 원본 파일, 리소스 파일, 컴파일러 옵션 등)만을 기반으로 함을 보장합니다. 등).

결정적 PE/COFF 파일의 `TimeDateStamp` COFF 파일 헤더에 있는 필드 값은 파일이 생성된 날짜와 시간을 나타내지 않으며 그렇게 해석해서는 안 됩니다. 대신 필드 값은 파일 콘텐츠의 해시에서 파생됩니다. 이 값을 계산하는 알고리즘은 파일을 생성한 도구의 구현 세부 정보입니다.

형식 Reproducible 의 디버그 디렉터리 항목에는 DebugDirectoryEntry.Type을 제외한 모든 필드가 0으로 설정되어야 합니다.

.NET에서의 종속성 로드

모든 .NET 애플리케이션에는 종속성이 있습니다. 간단한 `hello world` 앱에도 .NET 클래스 라이브러리의 일부에 대한 종속성이 있습니다.

.NET에서 기본 어셈블리 로드 논리를 이해하면 일반적인 배포 문제를 해결하는 데 도움이 될 수 있습니다.

일부 애플리케이션에서는 런타임 시 종속성이 동적으로 결정됩니다. 이러한 상황에서는 관리되는 어셈블리와 관리되지 않는 종속성이 로드되는 방식을 이해하는 것이 중요합니다.

AssemblyLoadContext

AssemblyLoadContext API는 .NET 로딩 디자인의 핵심입니다. AssemblyLoadContext 이해 문서에서는 디자인에 대한 개념적 개요를 제공합니다.

세부 정보 로드

로딩 알고리즘 세부 정보는 다음과 같은 여러 문서에서 간략하게 설명합니다.

- 관리되는 어셈블리 로드 알고리즘
- 위성 어셈블리 로드 알고리즘
- 관리되지 않는(네이티브) 라이브러리 로드 알고리즘
- 기본 탐색

플러그 인을 사용하여 앱 만들기

플러그 인을 사용하여 .NET 애플리케이션 만들기 자습서에서는 사용자 지정 AssemblyLoadContext를 만드는 방법을 설명합니다. 플러그인의 종속성을 해결하는 데 AssemblyDependencyResolver를 사용합니다. 이 자습서에서는 플러그 인의 종속성을 호스팅 애플리케이션에서 올바르게 격리합니다.

어셈블리 언로드 기능

.NET 문서에서 어셈블리 언로드 기능을 사용하고 디버그하는 방법은 단계별 자습서입니다. .NET 애플리케이션을 로드하고 실행한 다음 언로드하는 방법을 보여줍니다. 이 문서에서는 디버깅 팁도 제공합니다.

자세한 어셈블리 로딩 정보 수집

자세한 어셈블리 로드 정보 수집 문서에서는 런타임에서 관리되는 어셈블리 로드에 대한 자세한 정보를 수집하는 방법을 설명합니다. dotnet 추적 도구를 사용하여 실행 중인 프로세스의 추적에서 어셈블리 로더 이벤트를 캡처합니다.

Last updated on 2025. 12. 03.

System.Runtime.Loader.AssemblyLoadContext 정보

클래스는 AssemblyLoadContext .NET Core에서 도입되었으며 .NET Framework에서 사용할 수 없습니다. 이 문서에서는 개념 정보를 사용하여 AssemblyLoadContext API 설명서를 보완합니다.

이 문서는 동적 로드를 구현하는 개발자, 특히 동적 로드 프레임워크 개발자와 관련이 있습니다.

AssemblyLoadContext란?

모든 .NET 5+ 및 .NET Core 애플리케이션은 암시적으로 사용합니다 AssemblyLoadContext. 종속성을 찾고 로드하기 위한 런타임의 공급자입니다. 종속성이 로드될 때마다 인스턴스를 AssemblyLoadContext 호출하여 찾습니다.

- AssemblyLoadContext는 관리되는 어셈블리 및 기타 종속성을 찾고, 로드하고, 캐싱하는 서비스를 제공합니다.
- 동적 코드 로드 및 언로드를 지원하기 위해 자체 인스턴스에서 코드 및 해당 종속성을 로드하기 위한 격리된 컨텍스트를 AssemblyLoadContext 만듭니다.

버전 관리 규칙

단일 AssemblyLoadContext 인스턴스는 Assembly당 정확히 하나의 버전을 로드하는 것으로 제한됩니다. 해당 이름의 어셈블리가 이미 로드된 인스턴스에 대해 AssemblyLoadContext 어셈블리 참조가 확인되면 요청된 버전이 로드된 버전과 비교됩니다. 로드된 버전이 요청된 버전과 같거나 높은 경우에만 해결이 성공합니다.

여러 AssemblyLoadContext 인스턴스가 필요한 경우는 언제인가요?

단일 AssemblyLoadContext 인스턴스가 한 버전의 어셈블리만 로드할 수 있다는 제한은 코드 모듈을 동적으로 로드할 때 문제가 될 수 있습니다. 각 모듈은 독립적으로 컴파일되며 모듈은 서로 다른 버전의 Assembly에 따라 달라질 수 있습니다. 일반적으로 사용되는 라이브러리의 다른 버전에 대해 서로 다른 모듈이 의존할 때 문제가 자주 발생합니다.

동적으로 코드 AssemblyLoadContext 로드를 지원하기 위해 API는 동일한 애플리케이션에서 충돌하는 버전의 Assembly 로드를 제공합니다. 각 AssemblyLoadContext 인스턴스는 각각 AssemblyName.Name 을 특정 Assembly 인스턴스에 매핑하는 고유한 사전을 제공합니다.

또한 나중에 언로드할 수 있도록 코드 모듈과 관련된 종속성을 그룹화하기 위한 편리한 메커니즘을 제공합니다.

AssemblyLoadContext.Default 인스턴스

인스턴스는 AssemblyLoadContext.Default 시작 시 런타임에 의해 자동으로 채워집니다. 기본 검색을 사용하여 모든 정적 종속성을 찾아 찾습니다.

가장 일반적인 종속성 로드 시나리오를 해결합니다.

동적 종속성

AssemblyLoadContext 에는 재정의할 수 있는 다양한 이벤트 및 가상 함수가 있습니다.

인스턴스는 AssemblyLoadContext.Default 이벤트 재정의만 지원합니다.

관리되는 어셈블리 로드 알고리즘, 위성 어셈블리 로드 알고리즘 및 관리되지 않는(네이티브) 라이브러리 로드 알고리즘 문서는 사용 가능한 모든 이벤트 및 가상 함수를 참조합니다. 이 문서에서는 로드 알고리즘에서 각 이벤트 및 함수의 상대 위치를 보여 줍니다. 이 문서에서는 해당 정보를 재현하지 않습니다.

이 섹션에서는 관련 이벤트 및 함수에 대한 일반적인 원칙을 설명합니다.

- **반복 가능해야 합니다**. 특정 종속성에 대한 쿼리는 항상 동일한 응답을 생성해야 합니다. 동일한 로드된 종속성 인스턴스를 반환해야 합니다. 이 요구 사항은 캐시 일관성을 위한 기본 사항입니다. 특히 관리되는 어셈블리의 경우, Assembly 캐시를 만듭니다. 캐시 키는 간단한 어셈블리 이름 AssemblyName.Name입니다.
- **일반적으로 던지지 않습니다**. 이러한 함수는 요청된 종속성을 찾을 수 없을 때 throw하지 않고 반환 `null` 될 것으로 예상됩니다. throw 연산은 검색을 조기에 종료시키고 호출자에게 예외를 전달합니다. 예외 던지기는 손상된 어셈블리 또는 메모리 부족 상태와 같은 예기치 않은 오류로 제한되어야 합니다.
- **재귀를 피하십시오**. 이러한 함수 및 처리기는 종속성을 찾기 위한 로드 규칙을 구현합니다. 구현은 재귀를 트리거하는 API를 호출해서는 안 됩니다. 코드는 일반적으로 특정 경로 또는 메모리 참조 인수가 필요한 **AssemblyLoadContext** 로드 함수를 호출해야 합니다.
- **올바른 AssemblyLoadContext에 로드합니다**. 종속성을 로드할 위치를 선택하는 것은 애플리케이션별로 다릅니다. 선택은 이러한 이벤트 및 함수에 의해 구현됩니다. 코드가 **AssemblyLoadContext** load-by-path 함수를 호출할 때 코드를 로드하려는 인스턴스에서 호출합니다. 때때로 `null`을 반환하고 AssemblyLoadContext.Default이 로드를 처리하도록 하는 것이 가장 간단한 옵션일 수 있습니다.
- **스레드 경합에 유의하세요**. 로드는 여러 스레드에 의해 트리거될 수 있습니다. AssemblyLoadContext는 스레드 경합을 방지하기 위해 어셈블리를 원자적으로 캐시에 추

가하여 처리합니다. 경주 패자의 인스턴스는 삭제됩니다. 구현 논리에서 여러 스레드를 제대로 처리하지 않는 추가 논리를 추가하지 마세요.

동적 종속성은 어떻게 격리되는가?

각 AssemblyLoadContext 인스턴스는 Assembly 인스턴스와 Type 정의에 대한 고유한 범위를 나타냅니다.

이러한 종속성 간에는 이진 격리가 없습니다. 이름으로 서로를 찾지 못하여 격리됩니다.

각 AssemblyLoadContext:

- AssemblyName.Name 는 다른 Assembly 인스턴스를 참조할 수 있습니다.
- Type.GetType 는 동일한 형식에 대해 다른 형식 인스턴스를 반환할 수 있습니다 `name`.

공유 종속성

인스턴스 간에 AssemblyLoadContext 종속성을 쉽게 공유할 수 있습니다. 일반 모델은 AssemblyLoadContext 종속성을 로드하는 모델입니다. 다른 하나는 로드된 어셈블리에 대한 참조를 사용하여 종속성을 공유합니다.

이 공유는 런타임 어셈블리에 필요합니다. 이러한 어셈블리는 AssemblyLoadContext.Default에만 로드할 수 있습니다. 프레임워크(예`ASP.NET`: 또는 `WPF` `WinForms`.)에도 동일한 것이 필요합니다.

공유 종속성을 AssemblyLoadContext.Default에 로드하는 것이 좋습니다. 이 공유는 일반적인 디자인 패턴입니다.

공유는 사용자 지정 AssemblyLoadContext 인스턴스의 코딩에서 구현됩니다. AssemblyLoadContext 에는 재정의할 수 있는 다양한 이벤트 및 가상 함수가 있습니다. 이 함수들 중 어떤 것이 다른 Assembly 인스턴스에 로드된 AssemblyLoadContext 인스턴스에 대한 참조를 반환할 때, 그 Assembly 인스턴스는 공유됩니다. 표준 부하 알고리즘은 일반적인 공유 패턴을 간소화하기 위해 AssemblyLoadContext.Default에 로딩을 맡깁니다. 자세한 내용은 관리되는 어셈블리 로드 알고리즘을 참조하세요.

형식 변환 문제

두 AssemblyLoadContext 인스턴스에 동일한 형식 정의가 포함되어 있어도 `name`형식은 서로 다릅니다. 동일한 Assembly 인스턴스에서 오는 경우에 한해서 같은 유형입니다.

문제를 복잡하게 하기 위해 이러한 일치하지 않는 형식에 대한 예외 메시지는 혼동될 수 있습니다. 형식은 단순 형식 이름으로 예외 메시지에서 참조됩니다. 이 경우 일반적인 예외 메시지는 다

음과 같은 형식입니다.

> 'IsolatedType' 형식의 개체를 'IsolatedType' 형식으로 변환할 수 없습니다.

형식 변환 문제 디버그

일치하지 않는 형식 쌍이 있는 경우 다음 사항도 알아야 합니다.

- 각 형식의 Type.Assembly.
- 각 형식의 AssemblyLoadContext는 AssemblyLoadContext.GetLoadContext(Assembly) 함수를 통해서 얻을 수 있습니다.

주어진 두 개체 `a`와 `b`를 평가하는 것이 디버거에서 도움이 됩니다.

C#

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(a.GetType().Assembly)
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(b.GetType().Assembly)
```

형식 변환 문제 해결

이러한 형식 변환 문제를 해결하기 위한 두 가지 디자인 패턴이 있습니다.

1. 일반적인 공유 형식을 사용합니다. 이 공유 형식은 기본 런타임 형식이거나 공유 어셈블리에서 새 공유 형식을 만드는 작업을 포함할 수 있습니다. 공유 형식은 애플리케이션 어셈블리에 정의된 인터페이스 인 경우가 많습니다. 자세한 내용은 종속성을 공유하는 방법에 대해 읽어보세요.

2. 마샬링 기술을 사용하여 한 형식에서 다른 형식으로 변환합니다.

정적 멤버에 접근

사용자 지정 AssemblyLoadContext 에 로드된 형식은 다른 컨텍스트의 형식과 격리되므로 리플렉션을 사용하여 컨텍스트 외부에서 정적 멤버에 액세스해야 합니다.

예를 들어 동적으로 로드된 어셈블리에서 이 정적 클래스를 고려합니다.

C#

```
namespace MyPlugin;

public static class Paths
{
    public static DirectoryInfo RootIO { get; private set; }
}
```

속성 값을 읽는 데 사용합니다 PropertyInfo.GetValue . 정적 멤버에 인스턴스가 필요하지 않으므로 첫 번째 인수로 전달 `null` 합니다. 정규화된 형식 이름(네임스페이스 포함)을 다음으로 전달합니다 Assembly.GetType(String).

C#

```
// Get the type from the loaded assembly using the fully qualified name
Type pathsType = loadedAssembly.GetType("MyPlugin.Paths")
    ?? throw new InvalidOperationException("Type 'MyPlugin.Paths' not found in
loaded assembly.");

// Use PropertyInfo to access a static property
PropertyInfo rootIoProperty = pathsType.GetProperty("RootIO")
    ?? throw new InvalidOperationException("Property 'RootIO' was not found on type
'MyPlugin.Paths'.");
DirectoryInfo rootIo = (DirectoryInfo)rootIoProperty.GetValue(null);
```

대안으로, C#은 속성 접근자를 `get_` 및 `set_` 접두사와 함께 메서드로 컴파일합니다. Type.GetMethod를 사용하여 이러한 접근자 메서드를 직접 호출할 수 있습니다. 그러나 Type.GetMethod(String) public 메서드만 반환합니다. 접근자가 public이 아닌 경우(예: `private set` 예제의 경우) 다음을 허용하는 오버로드를 사용해야 합니다.BindingFlags

C#

```
// Public getter — no BindingFlags needed
MethodInfo getRootIo = pathsType.GetMethod("get_RootIO")
    ?? throw new InvalidOperationException("Accessor method 'get_RootIO' was not
found on type 'MyPlugin.Paths'.");
DirectoryInfo rootIo = (DirectoryInfo)getRootIo.Invoke(null, null);

// Non-public setter — must use BindingFlags
MethodInfo setRootIo = pathsType.GetMethod(
    "set_RootIO",
    BindingFlags.Static | BindingFlags.NonPublic)
    ?? throw new InvalidOperationException("Accessor method 'set_RootIO' was not
found on type 'MyPlugin.Paths'.");
setRootIo.Invoke(null, new object[] { newValue });
```

동일한 패턴은 정적 필드에 적용되며, 이는 FieldInfo.GetValue를 통해 액세스할 수 있습니다. 또한, 정적 메서드는 FieldInfo.SetValue 호출하고 MethodBase.Invoke으로 실행됩니다.

ⓘ 참고 항목

로드된 어셈블리에서 형식이 정의된 값을 검색하는 경우 호출 컨텍스트에서 캐스팅하려고 할 때 형식 변환 문제가 발생할 수 있습니다. 자세한 내용은 **형식 변환 문제를 참조하세요**.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 12.

기본 탐지

인스턴스는 AssemblyLoadContext.Default 어셈블리의 종속성을 찾는 역할을 담당합니다. 이 문서에서는 인스턴스의 AssemblyLoadContext.Default 검색 논리에 대해 설명합니다.

호스트 구성 탐색 속성

런타임이 시작되면 런타임 호스트는 프로브 경로를 구성하는 AssemblyLoadContext.Default 명명된 검색 속성 집합을 제공합니다.

각 검색 속성은 선택 사항입니다. 있는 경우 각 속성은 절대 경로의 구분된 목록을 포함하는 문자열 값입니다. 구분 기호는 Windows의 경우 ';'이고 다른 모든 플랫폼에서는 ':'입니다.

테이블 확장

속성 이름	Description
`TRUSTED_PLATFORM_ASSEMBLIES`	플랫폼 및 애플리케이션 어셈블리 파일 경로 목록입니다.
`PLATFORM_RESOURCE_ROOTS`	위성 리소스 어셈블리를 검색할 디렉터리 경로 목록입니다.
`NATIVE_DLL_SEARCH_DIRECTORIES`	관리되지 않는(네이티브) 라이브러리를 검색할 디렉터리 경로 목록입니다.
`APP_PATHS`	관리되는 어셈블리를 검색할 디렉터리 경로 목록입니다.

속성은 어떻게 설정되나요?

myapp<.deps.json파일이 있는지 여부에> 따라 속성을 채우는 두 가지 주요 시나리오가 있습니다.

- *.deps.json* 파일이 있으면 구문 분석되어 검색 속성을 채웁니다.
- *.deps.json* 파일이 없으면 애플리케이션의 디렉터리에 모든 종속성이 포함된 것으로 간주됩니다. 디렉터리의 내용은 검색 속성을 채우는 데 사용됩니다.

또한 참조된 프레임워크에 대한 *.deps.json* 파일도 비슷하게 구문 분석됩니다.

환경 변수 `DOTNET_ADDITIONAL_DEPS` 를 사용하여 추가 종속성을 추가할 수 있습니다. `dotnet.exe` 에는 애플리케이션 시작 시 이 값을 설정하는 선택적 `--additional-deps` 매개 변수도 포함되어 있습니다.

> **참고 항목**

`DOTNET_ADDITIONAL_DEPS` 환경 변수 및 `--additional-deps` 명령줄 옵션은 **프레임워크 종속 애플리케이션**에만 적용됩니다. 이러한 옵션은 **자체 포함 애플리케이션에 대해 무시됩니다**. 자세한 내용은 **프레임워크 종속 배포와 자체 포함 배포를 참조하세요**.

속성은 `APP_PATHS` 기본적으로 채워지지 않으며 대부분의 애플리케이션에서 생략됩니다.

애플리케이션에서 사용하는 모든 **.deps.json* 파일 목록은 .를 통해 `System.AppContext.GetData("APP_CONTEXT_DEPS_FILES")` 액세스할 수 있습니다.

관리 코드에서 검색 속성을 보려면 어떻게 하나요?

각 속성은 위의 표에서 속성 이름으로 함수를 호출 AppContext.GetData(String) 하여 사용할 수 있습니다.

프로빙 속성의 구성을 디버그하려면 어떻게 해야 하나요?

.NET Core 런타임 호스트는 특정 환경 변수를 사용할 때 유용한 추적 메시지를 출력합니다.

테이블 확장

환경 변수	Description
`DOTNET_HOST_TRACE=1`	추적을 사용하도록 설정합니다.
`DOTNET_HOST_TRACEFILE=<path>`	기본 `stderr` 값 대신 파일 경로로 추적합니다.
`DOTNET_HOST_TRACE_VERBOSITY`	출력 수준을 1(가장 낮음)에서 4(가장 높음)으로 설정합니다.

자세한 내용은 DOTNET_HOST_TRACE 환경 변수를 참조하세요.

관리되는 어셈블리의 기본 탐색

관리 어셈블리를 찾기 위해 검색할 때, AssemblyLoadContext.Default는 다음 순서대로 확인합니다.

- 파일 확장자를 제거한 AssemblyName.Name 후 인 `TRUSTED_PLATFORM_ASSEMBLIES` 과 일치하는 파일입니다.
- `APP_PATHS` 에 공통 파일 확장명을 가진 어셈블리 파일들.

기본 AssemblyLoadContext으로 로드할 때는 `TRUSTED_PLATFORM_ASSEMBLIES` 이나 `APP_PATHS` 에 있는 어셈블리가 지정된 경로나 원시 어셈블리 개체보다 우선합니다. 예를 들어, 기본 AssemblyLoadContext에서 AssemblyLoadContext.LoadFromStream 또는

AssemblyLoadContext.LoadFromAssemblyPath을 호출하고 `TRUSTED_PLATFORM_ASSEMBLIES` 또는 `APP_PATHS`에 이름이 일치하는 어셈블리가 있을 경우, 런타임은 지정된 스트림이나 경로가 아닌 해당 위치에서 어셈블리를 로드합니다.

위성 자원 조립 탐사

특정 문화권에 대한 위성 어셈블리를 찾으려면 파일 경로 집합을 생성합니다.

각 경로에 대해 `PLATFORM_RESOURCE_ROOTS` 및 `APP_PATHS` 후에, CultureInfo.Name 문자열, 디렉토리 구분 기호, AssemblyName.Name 문자열, 그리고 확장명 '.dll'을 추가합니다.

일치하는 파일이 있으면 로드하고 반환합니다.

관리되지 않는(네이티브) 라이브러리 검색

런타임의 관리되지 않는 라이브러리 검색 알고리즘은 모든 플랫폼에서 동일합니다. 그러나 관리되지 않는 라이브러리의 실제 로드는 기본 플랫폼에서 수행되므로 관찰된 동작은 약간 다를 수 있습니다.

1. 제공된 라이브러리 이름이 절대 경로 또는 상대 경로를 나타내는지 확인합니다.
2. 이름이 절대 경로를 나타내는 경우 모든 후속 작업에 직접 이름을 사용합니다. 그렇지 않으면 이름을 사용하여 플랫폼에서 정의한 조합을 고려하십시오. 조합은 플랫폼별 접두사(예: `lib`) 및/또는 접미사(예: `.dll`, `.dylib` 및 `.so`)로 구성됩니다. 이 목록은 전체 목록이 아니며 각 플랫폼에서 수행된 정확한 노력을 나타내지 않습니다. 이는 고려되는 항목의 예일 뿐입니다. 자세한 내용은 네이티브 라이브러리 로드를 참조하세요.
3. 이름 및 경로가 상대적인 경우 각 조합은 다음 단계에서 사용됩니다. 첫 번째 성공적인 로드 시도는 로드된 라이브러리에 핸들을 즉시 반환합니다.
    - 속성에 제공된 각 경로에 `NATIVE_DLL_SEARCH_DIRECTORIES` 추가하고 로드를 시도합니다.
    - DefaultDllImportSearchPathsAttribute이 호출 어셈블리 또는 p/invoke에서 정의되지 않았거나 정의되어 `DllImportSearchPath.AssemblyDirectory`을 포함하는 경우, 이름 또는 조합을 호출 어셈블리의 디렉터리에 추가하고 로드를 시도합니다.
    - 라이브러리를 직접 로드하는 데 사용합니다.
4. 라이브러리를 로드하지 못했음을 나타냅니다.

Last updated on 2026. 01. 22.

관리되는 어셈블리 로드 알고리즘

관리되는 어셈블리는 다양한 단계가 있는 알고리즘을 사용하여 배치되고 로드됩니다.

위성 어셈블리 및 `WinRT` 어셈블리를 제외한 모든 관리되는 어셈블리는 동일한 알고리즘을 사용합니다.

관리되는 어셈블리는 언제 로드되나요?

관리되는 어셈블리 로드를 트리거하는 가장 일반적인 메커니즘은 정적 어셈블리 참조입니다. 이러한 참조는 코드가 다른 어셈블리에 정의된 형식을 사용할 때마다 컴파일러에 의해 삽입됩니다. 이러한 어셈블리는 런타임에 필요에 따라 로드됩니다(`load-by-name`). 정적 어셈블리 참조가 로드되는 정확한 타이밍은 지정되지 않습니다. 런타임 버전마다 다를 수 있으며 인라인 처리와 같은 최적화의 영향을 받습니다.

다음 API를 직접 사용하면 로드도 트리거됩니다.

테이블 확장

응용 프로그램 인터페이스 (API)	설명	`Active` **AssemblyLoadContext**
AssemblyLoadContext.LoadFromAssemblyName	`Load-by-name`	이 인스턴스입니다.
AssemblyLoadContext.LoadFromAssemblyPath AssemblyLoadContext.LoadFromNativeImagePath	경로에서 로드합니다.	이 인스턴스입니다.
AssemblyLoadContext.LoadFromStream	개체에서 불러오기.	이 인스턴스입니다.
Assembly.LoadFile	새 AssemblyLoadContext 인스턴스의 경로에서 로드	새 AssemblyLoadContext 인스턴스입니다.
Assembly.LoadFrom	인스턴스의 경로에서 로드합니다 AssemblyLoadContext.Default . AppDomain.AssemblyResolve 처리기를 추가합니다. 처리기는 해당 디렉터리에서 어셈블리의 종속성을 로드합니다.	AssemblyLoadContext.Default 인스턴스입니다.
Assembly.Load(AssemblyName) Assembly.Load(String) Assembly.LoadWithPartialName	`Load-by-name`;	호출자에서 유추됩니다. AssemblyLoadContext 메서드를 선호합니다.
Assembly.Load(Byte[]) Assembly.Load(Byte[], Byte[])	새 AssemblyLoadContext 인스턴스의 개체에서 로드합니다.	새 AssemblyLoadContext 인스턴스입니다.

응용 프로그램 인터페이스 (API)	설명	`Active` AssemblyLoadContext
Type.GetType(String) Type.GetType(String, Boolean) Type.GetType(String, Boolean, Boolean)	`Load-by-name`;	호출자에서 유추됩니다. Type.GetType 메서드를 `assemblyResolver` 인수로 선호합니다.
Assembly.GetType	형식 `name`이 어셈블리 자격을 갖춘 제네릭 형식을 설명하는 경우, `Load-by-name`을 트리거합니다.	호출자에서 유추됩니다. 어셈블리 정규화된 형식 이름을 사용할 때는 Type.GetType을(를) 선호합니다.
Activator.CreateInstance(String, String) Activator.CreateInstance(String, String, Object[]) Activator.CreateInstance(String, String, Boolean, BindingFlags, Binder, Object[], CultureInfo, Object[])	`Load-by-name`;	호출자에서 유추됩니다. Activator.CreateInstance 인수를 취하는 Type 메서드를 선호합니다.

> **Important**
>
> .NET Framework와 `assemblyFile` 달리 매개 변수 Assembly.LoadFrom 는 URI가 아닌 .NET의 파일 경로로 처리됩니다. .NET Framework에서는 파일 URI(예를 들어, `file:///C:/path/to/assembly.dll`—Assembly.CodeBase를 사용하여 생성된 것)를 전달할 수 있으며, 어셈블리를 성공적으로 로드할 수 있습니다. .NET에서 `assemblyFile` 값이 URI를 제대로 처리하지 못하는 Path.GetFullPath으로 전달되어 로드가 실패합니다. 파일 URI 문자열이 이미 있는 경우 먼저 인스턴스를 Uri 만들고 호출하기 전에 LocalPath해당 Assembly.LoadFrom 속성을 사용하여 파일 경로를 가져옵니다. 이미 로드된 어셈블리의 파일 경로를 얻으려면 `CodeBase` 대신 Assembly.Location을(를) 사용하세요.

알고리즘

다음 알고리즘은 런타임이 관리되는 어셈블리를 로드하는 방법을 설명합니다.

1. 를 확인합니다 `active`AssemblyLoadContext.

   - 정적 어셈블리 참조의 `active`AssemblyLoadContext 경우 참조 어셈블리를 로드한 인스턴스입니다.
   - 선호하는 API는 `active`AssemblyLoadContext를 명시적으로 만듭니다.
   - 다른 API들은 `active`AssemblyLoadContext를 유추합니다. 이 API에서는 AssemblyLoadContext.CurrentContextualReflectionContext 속성이 사용됩니다. 값이 `null`면 유추된 AssemblyLoadContext 인스턴스가 사용됩니다.
   - 관리되는 어셈블리가 로드되는 시기 섹션의 표를 참조하세요.

2. 메서드 `Load-by-name`의 경우 `active`AssemblyLoadContext는 어셈블리를 다음 우선 순위 순서로 로드합니다.

    - 확인하세요 `cache-by-name`.
    - AssemblyLoadContext.Load 함수를 호출합니다.
    - 인스턴스의 캐시를 AssemblyLoadContext.Default 확인하고 관리되는 어셈블리 기본 검색 논리를 실행합니다. 어셈블리가 새로 로드되면 AssemblyLoadContext.Default 인스턴스의 `cache-by-name`에 참조가 추가됩니다.
    - AssemblyLoadContext.Resolving 활성 AssemblyLoadContext에 대한 이벤트를 트리거합니다.
    - AppDomain.AssemblyResolve 이벤트를 발생시킵니다.

3. 다른 유형의 로드의 경우 어셈블리를 `active`AssemblyLoadContext 다음 우선 순위 순서로 로드합니다.

    - 확인하세요 `cache-by-name`.
    - `active` AssemblyLoadContext이 AssemblyLoadContext.Default이면 관리되는 어셈블리에 대한 기본 검색 논리를 실행합니다.
    - 지정된 경로 또는 원시 어셈블리 개체에서 로드합니다. 어셈블리가 새로 로드되면 인스턴스`active`에 참조가 AssemblyLoadContext`cache-by-name` 추가됩니다.

4. 두 경우 모두 어셈블리가 새로 로드되면 AppDomain.AssemblyLoad 이벤트가 발생합니다.

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 12.

위성 어셈블리 로드 알고리즘

위성 어셈블리는 언어 및 문화권에 맞게 사용자 지정된 지역화된 리소스를 저장하는 데 사용됩니다.

위성 어셈블리는 일반적인 관리 어셈블리와 다른 로드 알고리즘을 사용합니다.

위성 어셈블리는 언제 로드되나요?

위성 어셈블리는 지역화된 리소스를 로드할 때 로드됩니다.

지역화된 리소스를 로드하는 기본 API는 System.Resources.ResourceManager 클래스입니다. 궁극적으로 ResourceManager 클래스는 각 GetSatelliteAssembly에 대해 CultureInfo.Name 메서드를 호출합니다.

상위 수준 API는 하위 수준 API를 추상화할 수 있습니다.

알고리즘

.NET Core 리소스 대체 프로세스에는 다음 단계가 포함됩니다.

1. `active` AssemblyLoadContext 인스턴스를 확인합니다. 모든 경우에 `active` 인스턴스는 실행 중인 어셈블리의 AssemblyLoadContext입니다.
2. `active` 인스턴스는 다음 우선 순위에 따라 요청된 문화권에 대한 위성 어셈블리를 로드합니다.
    - 캐시를 확인합니다.
    - `active`가 AssemblyLoadContext.Default 인스턴스인 경우 기본 위성(리소스) 어셈블리 조사 논리를 실행합니다.
    - AssemblyLoadContext.Load 함수를 호출합니다.
    - 위성 어셈블리에 해당하는 관리 어셈블리가 파일에서 로드된 경우 관리 어셈블리의 디렉터리에서 요청된 CultureInfo.Name(예: `es-MX`)과 일치하는 하위 디렉터리를 확인합니다.

        > ⓘ **참고 항목**
        >
        > Linux 및 macOS에서 하위 디렉터리는 대/소문자를 구분하며 다음 중 하나를 수행해야 합니다.
        > ◦ 대/소문자 구분이 정확히 일치합니다.

    - 소문자여야 합니다.

- AssemblyLoadContext.Resolving 이벤트를 발생시킵니다.
- AppDomain.AssemblyResolve 이벤트를 발생시킵니다.

3. 위성 어셈블리가 로드된 경우:

- AppDomain.AssemblyLoad 이벤트가 발생합니다.
- 어셈블리에서 요청된 리소스를 검색합니다. 런타임이 어셈블리에서 리소스를 찾으면 해당 리소스가 사용됩니다. 리소스를 찾지 못하면 검색을 계속합니다.

> ⓘ **참고 항목**
>
> 위성 어셈블리 내의 리소스를 찾기 위해 런타임은 ResourceManager에서 현재 CultureInfo.Name에 대해 요청한 리소스 파일을 검색합니다. 리소스 파일 내에서 요청된 리소스 이름을 검색합니다. 둘 중 하나라도 찾지 못하면 리소스가 찾을 수 없는 것으로 처리됩니다.

4. ResourceManager는 다음에 여러 가능한 수준에서 부모 문화권의 어셈블리를 순차적으로 검색하며 매번 2단계 및 3단계를 반복합니다.

   각 문화권에는 CultureInfo.Parent 속성으로 정의되는 하나의 부모만 있습니다.

   문화권의 Parent 속성이 CultureInfo.InvariantCulture이면 부모 문화권 검색이 중지됩니다.

   InvariantCulture의 경우 2-3 단계로 돌아가지 않고 5단계로 계속 진행합니다.

5. 여전히 리소스를 찾을 수 없으면 ResourceManager는 기본(대체) 문화권에 대한 리소스를 사용합니다.

   일반적으로 기본 문화권의 리소스는 주 애플리케이션 어셈블리에 포함되어 있습니다. 그러나 UltimateResourceFallbackLocation.Satellite 속성에 대해 NeutralResourcesLanguageAttribute.Location을 지정할 수 있습니다. 이 값은 리소스의 최종 대체 위치가 주 어셈블리가 아닌 위성 어셈블리임을 나타냅니다.

> ⓘ **참고 항목**
>
> 기본 문화권이 최종 대체(fallback)입니다. 따라서 기본 리소스 파일에 완전한 리소스 세트를 항상 포함하는 것이 좋습니다. 이렇게 하면 예외가 발생하는 것을 방지하는 데 도움이 됩니다. 전체 집합을 통해 모든 리소스에 대한 대체를 제공하여, 비록 문화적으

로 특정하지 않더라도 사용자가 항상 하나 이상의 리소스를 사용할 수 있도록 보장합니다.

6. 마지막으로

    - 런타임이 기본(예비) 문화권에 대한 리소스 파일을 찾지 못하면 MissingManifestResourceException 또는 MissingSatelliteAssemblyException 예외가 발생합니다.
    - 리소스 파일이 있지만 요청된 리소스가 없는 경우 요청에서 `null`을 반환합니다.

Last updated on 2026. 02. 24.

관리되지 않는(네이티브) 라이브러리 로드 알고리즘

관리되지 않는 라이브러리는 다양한 단계를 포함하는 알고리즘으로 배치되고 로드됩니다.

다음 알고리즘에서는 `PInvoke`를 통해 네이티브 라이브러리를 로드하는 방법을 설명합니다.

`PInvoke` 로드 라이브러리 알고리즘

`PInvoke`는 관리되지 않는 어셈블리를 로드하려고 할 때 다음 알고리즘을 사용합니다.

1. `active` AssemblyLoadContext를 확인합니다. 관리되지 않는 로드 라이브러리의 경우 `active` AssemblyLoadContext는 `PInvoke`를 정의하는 어셈블리를 사용합니다.

2. `active` AssemblyLoadContext의 경우 다음을 통해 우선 순위를 기준으로 어셈블리를 찾습니다.

    - 해당 캐시를 확인하는 중입니다.
    - System.Runtime.InteropServices.DllImportResolver 함수로 설정된 현재 NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver) 대리자를 호출하는 중입니다.
    - AssemblyLoadContext.LoadUnmanagedDll AssemblyLoadContext에서 `active` 함수를 호출하는 중입니다.
    - AppDomain 인스턴스의 캐시를 확인하고 관리되지 않는(네이티브) 라이브러리 검색 논리를 실행합니다.
    - `active` AssemblyLoadContext에 대한 AssemblyLoadContext.ResolvingUnmanagedDll 이벤트를 발생시킵니다.

Last updated on 2026. 02. 24.

자세한 어셈블리 로딩 정보 수집

2025. 06. 17.

.NET 5부터 런타임은 `EventPipe`에 대한 자세한 정보를 통해 이벤트를 내보낼 수 있습니다. 이러한 이벤트는`Microsoft-Windows-DotNETRuntime` 공급자가 `AssemblyLoader` 키워드(`0x4`) 아래에서 내보냅니다.

필수 조건

- .NET 5 SDK 이상 버전
- dotnet-trace 도구

> ⓘ **참고**
>
> 기능의 `dotnet-trace` 범위는 자세한 어셈블리 로드 정보를 수집하는 것보다 큽합니다. 사용 현황 `dotnet-trace`에 대한 자세한 내용은 다음을 참조하세요 dotnet-trace.

어셈블리 로드 이벤트를 사용하여 추적 수집

기존 프로세스를 추적하거나 자식 프로세스를 시작하고 시작부터 추적하는 데 사용할 `dotnet-trace` 수 있습니다.

기존 프로세스 추적

런타임에서 어셈블리 로드 이벤트를 사용하도록 설정하고 추적을 수집하려면 다음 명령을 사용합니다 `dotnet-trace` .

콘솔

```
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id <pid>
```

이 명령은 지정한 `<pid>`의 추적을 수집하여 `AssemblyLoader` 공급자에서 `Microsoft-Windows-DotNETRuntime` 이벤트를 활성화합니다. 결과는 `.nettrace` 파일입니다.

dotnet-trace를 사용하여 자식 프로세스를 시작하고 프로세스 시작부터 추적합니다.

경우에 따라 시작에서 프로세스의 추적을 수집하는 것이 유용할 수 있습니다. .NET 5 이상을 실행하는 앱의 경우 이 작업을 수행하는 데 사용할 `dotnet-trace` 수 있습니다.

다음 명령은 와 `arg1`를 명령줄 인수로 사용하여 `arg2`를 실행하고 런타임 시작 시점에서 추적을 수집합니다.

콘솔

```
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 -- hello.exe arg1 arg2
```

`Enter` 키 또는 `Ctrl` + `C`를 눌러 추적 수집을 중지할 수 있습니다. 또한 *hello.exe*닫힙니다.

> ⓘ **참고**
>
> - *hello.exe*를 시작하면 `dotnet-trace` 입력과 출력이 리디렉션되어 디폴트로 콘솔에서 이 프로그램과 상호 작용할 수 없습니다. `--show-child-io` 스위치를 사용하여 `stdin` 및 `stdout`와 상호 작용합니다.
> - `Ctrl` + `C`를 통해 도구를 종료하거나 `SIGTERM` 도구와 자식 프로세스를 모두 안전하게 종료합니다.
> - 자식 프로세스가 툴보다 먼저 종료되면 툴도 종료되고, 추적을 안전하게 볼 수 있습니다.

추적을 확인하기

수집된 추적 파일은 PerfView의 이벤트 보기를 사용하여 Windows에서 볼 수 있습니다. 모든 어셈블리 로드 이벤트는 `Microsoft-Windows-DotNETRuntime/AssemblyLoader`로 시작됩니다.

예제(Windows)

이 예제에서는 어셈블리 로드 확장 지점 샘플을 사용합니다. 애플리케이션은 어셈블리를 로드하려고 시도합니다. 어셈블리 `MyLibrary` 는 애플리케이션에서 참조하지 않으므로 어셈블리 로드 확장 지점에서 처리가 성공적으로 로드되어야 합니다.

추적 데이터 수집

1. 다운로드한 샘플을 사용하여 디렉터리로 이동합니다. 다음을 사용하여 애플리케이션을 빌드합니다.

   콘솔

   ```
   dotnet build
   ```

2. 키 누름을 기다리며 일시 중지해야 함을 나타내는 인수를 사용하여 애플리케이션을 시작합니다. 다시 열 때 성공적으로 로드하는 데 필요한 처리 없이 기본값 `AssemblyLoadContext` 으로 어셈블리를 로드하려고 시도합니다. 출력 디렉터리로 이동하여 다음을 실행합니다.

   콘솔

   ```
   AssemblyLoading.exe /d default
   ```

3. 애플리케이션의 프로세스 ID를 찾습니다.

   콘솔

   ```
   dotnet-trace ps
   ```

   출력에는 사용 가능한 프로세스가 나열됩니다. 다음은 그 예입니다.

   콘솔

   ```
   35832 AssemblyLoading C:\src\AssemblyLoading\bin\Debug\net5.0\AssemblyLoading.exe
   ```

4. 실행 중인 애플리케이션`dotnet-trace`에 붙입니다.

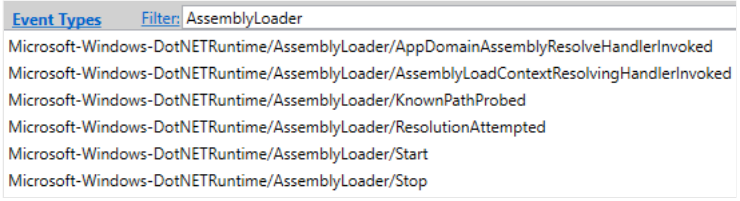
   콘솔

   ```
   dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id 35832
   ```

5. 애플리케이션을 실행하는 창에서 아무 키나 눌러 프로그램을 계속할 수 있도록 합니다. 애플리케이션이 종료되면 추적이 자동으로 중지됩니다.

추적 보기

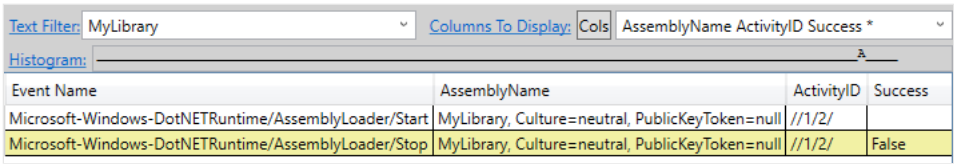
PerfView에서 수집된 추적을 열고 이벤트 보기를 엽니다. `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 이벤트로 목록을 필터링합니다.



추적이 시작된 후 애플리케이션에서 발생한 모든 어셈블리 로드가 표시됩니다. 이 예제 `MyLibrary`에 대해 관심 있는 어셈블리에 대한 부하 작업을 검사하려면 좀 더 필터링을 수행할 수 있습니다.

어셈블리 부하

왼쪽의 이벤트 목록을 사용하여 Start에 포함된 Stop 및 `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 이벤트로 뷰를 필터링하세요. 열 `AssemblyName`, `ActivityID`, 및 `Success`를 뷰에 추가합니다. `MyLibrary`을 포함하는 이벤트로 필터링합니다.



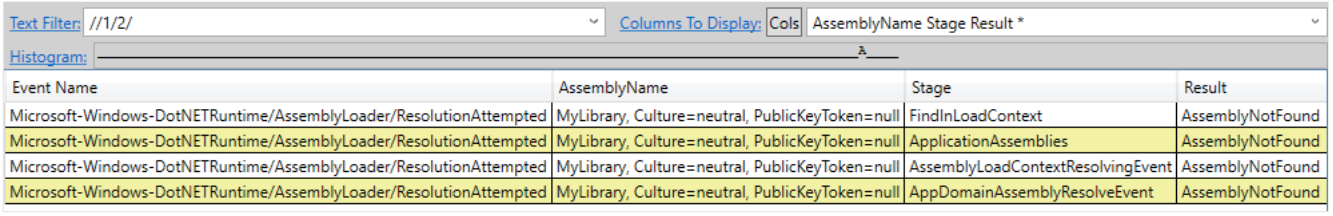
테이블 확장

이벤트 이름	어셈블리 이름	ActivityID	성공
`AssemblyLoader/Start`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	
`AssemblyLoader/Stop`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	거짓

로드 작업이 실패했음을 나타내는 `Start`와 함께하는 한 쌍의 /`Stop` `Success=False`이 `Stop` 이벤트에 표시됩니다. 두 이벤트는 동일한 활동 ID를 갖습니다. 활동 ID를 사용하여 다른 모든 어셈블리 로더 이벤트를 이 로드 작업에 해당하는 이벤트로 필터링할 수 있습니다.

로드 시도 상세 분석

부하 작업을 더 자세히 분석하려면 왼쪽에 있는 이벤트 목록을 사용하여 `ResolutionAttempted` 아래의 `Microsoft-Windows-DotNETRuntime/AssemblyLoader`로 보기를 필터링하세요. 열 `AssemblyName`, `Stage`, 및 `Result`를 뷰에 추가합니다. `Start` / `Stop` 쌍에서 활동 ID로 필터링된 이벤트를 표시합니다.



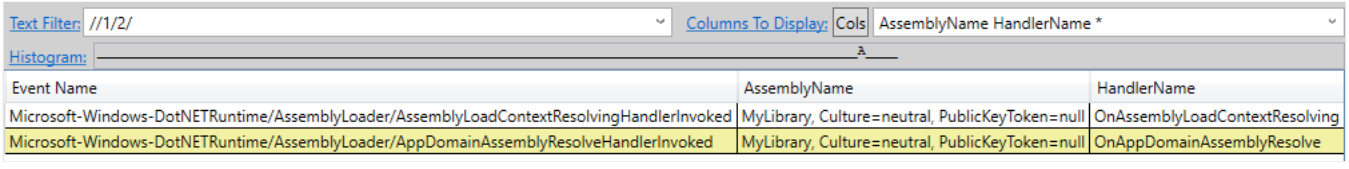
테이블 확장

이벤트 이름	어셈블리 이름	단계	결과
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`FindInLoadContext`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`ApplicationAssemblies`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AssemblyLoadContextResolvingEvent`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AppDomainAssemblyResolveEvent`	`AssemblyNotFound`

위의 이벤트는 어셈블리 로더가 현재 부하 컨텍스트를 살펴보고, 관리되는 애플리케이션 어셈블리에 대한 기본 검색 논리를 실행하고, 이벤트에 대한 AssemblyLoadContext.Resolving 처리기를 호출하고, 처리기를 AppDomain.AssemblyResolve호출하여 어셈블리를 확인하려고 했음을 나타냅니다. 이러한 모든 단계에서 어셈블리를 찾을 수 없습니다.

확장 포인트

호출된 확장 지점을 확인하려면 왼쪽의 이벤트 목록을 사용하여, AssemblyLoadContextResolvingHandlerInvoked 아래에서 AppDomainAssemblyResolveHandlerInvoked 및 `Microsoft-Windows-DotNETRuntime/AssemblyLoader`로 뷰를 필터링합니다. 뷰에 `AssemblyName` 열과 `HandlerName` 열을 추가하세요. `Start` / `Stop` 쌍에서 활동 ID로 필터링된 이벤트를 표시합니다.



테이블 확장

이벤트 이름	어셈블리 이름	핸들러 이름
`AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAssemblyLoadContextResolving`
`AssemblyLoader/AppDomainAssemblyResolveHandlerInvoked`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAppDomainAssemblyResolve`

위의 이벤트는 이벤트에 대해 `OnAssemblyLoadContextResolving` 명명 AssemblyLoadContext.Resolving 된 처리기가 호출되었고 해당 이벤트에 대해 `OnAppDomainAssemblyResolve` 명명 AppDomain.AssemblyResolve 된 처리기가 호출되었음을 나타냅니다.

또 다른 흔적을 수집하세요.

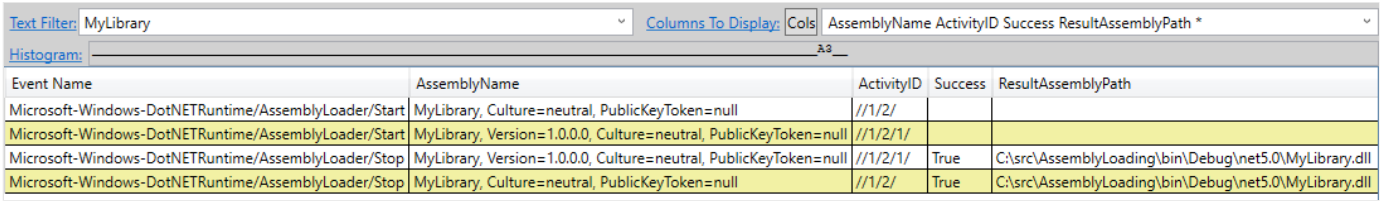
애플리케이션을 AssemblyLoadContext.Resolving 인수를 사용해 실행하여, `MyLibrary` 어셈블리를 로드할 수 있는 방식으로 이벤트의 처리기를 작동시킵니다.

콘솔

```
AssemblyLoading /d default alc-resolving
```

`.nettrace` 사용하여 다른 파일을 수집하고 엽니다.

`Start` 및 `Stop` 이벤트를 `MyLibrary`에 대해 다시 필터링합니다. 둘 사이에 다른 `Start`/`Stop` 쌍이 `Start`/`Stop` 표시됩니다. 내부 로드 작업은 호출 AssemblyLoadContext.Resolving할 때 처리기에 의해 트리거되는 부하를 AssemblyLoadContext.LoadFromAssemblyPath 나타냅니다. 이번에는 `Success=True`가 `Stop` 이벤트에 표시되어 로드 작업이 성공했음을 나타냅니다. 필드에 결과 `ResultAssemblyPath` 어셈블리의 경로가 표시됩니다.

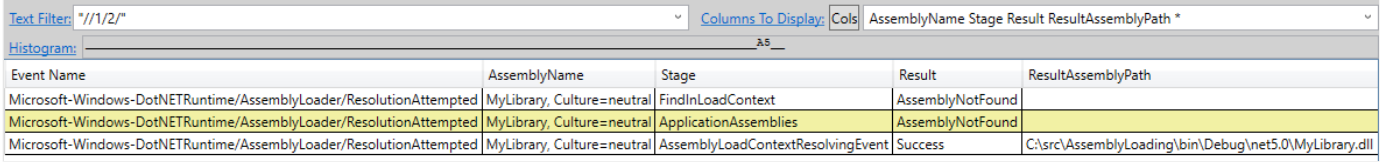

Text Filter: MyLibrary
Columns To Display: Cols AssemblyName ActivityID Success ResultAssemblyPath *
Histogram:

Event Name	AssemblyName	ActivityID	Success	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start	MyLibrary, Culture=neutral, PublicKeyToken=null	//1/2/		
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start	MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null	//1/2/1/		
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop	MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null	//1/2/1/	True	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop	MyLibrary, Culture=neutral, PublicKeyToken=null	//1/2/	True	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

테이블 확장

이벤트 이름	어셈블리 이름	ActivityID	성공	결과어셈블리경로 (ResultAssemblyPath)
`AssemblyLoader/Start`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/		
`AssemblyLoader/Start`	`MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`	//1/2/1/		
`AssemblyLoader/Stop`	`MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`	//1/2/1/	진실	*C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll*
`AssemblyLoader/Stop`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	진실	*C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll*

그런 다음, 외부 로드의 활동 ID가 있는 이벤트를 확인 `ResolutionAttempted` 하여 어셈블리가 성공적으로 해결된 단계를 확인할 수 있습니다. 이번에는, 이 이벤트들이 `AssemblyLoadContextResolvingEvent` 스테이지가 성공적이었다는 것을 보여줍니다. 필드에 결과 `ResultAssemblyPath` 어셈블리의 경로가 표시됩니다.

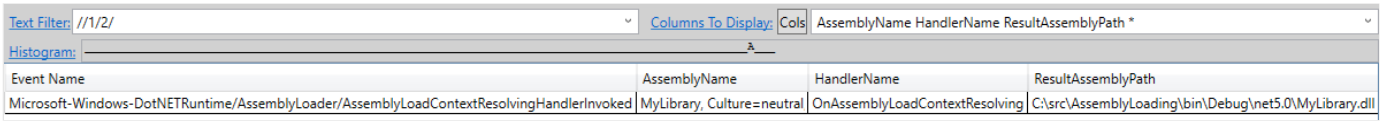

Text Filter: "//1/2/"
Columns To Display: Cols AssemblyName Stage Result ResultAssemblyPath *
Histogram:

Event Name	AssemblyName	Stage	Result	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	FindInLoadContext	AssemblyNotFound	
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	ApplicationAssemblies	AssemblyNotFound	
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	AssemblyLoadContextResolvingEvent	Success	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

테이블 확장

이벤트 이름	어셈블리 이름	단계	결과	결과어셈블리경로 (ResultAssemblyPath)
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`FindInLoadContext`	`AssemblyNotFound`	
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`ApplicationAssemblies`	`AssemblyNotFound`	
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AssemblyLoadContextResolvingEvent`	`Success`	*C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibr*

이벤트를 살펴보면 `AssemblyLoadContextResolvingHandlerInvoked` 명명 `OnAssemblyLoadContextResolving` 된 처리기가 호출되었음을 보여 줍니다. 필드에는 `ResultAssemblyPath` 처리기에서 반환된 어셈블리의 경로가 표시됩니다.



테이블 확장

이벤트 이름	어셈블리 이름	핸들러 이름	결과어셈블리경로 (ResultAssemblyPath)
`AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAssemblyLoadContextResolving`	*C:\src\AssemblyLoading\bin\Debug\net5.0\MyLib*

`ResolutionAttempted` 이벤트를 발생시키는 로드 알고리즘 단계에 도달하기 전에 어셈블리가 성공적으로 로드되었기 때문에 더 이상 `AppDomainAssemblyResolveEvent` 이벤트가 `AppDomainAssemblyResolveHandlerInvoked` 단계에서 발생하지 않으며 AppDomain.AssemblyResolve 이벤트도 발생하지 않습니다.

참고하십시오

- 어셈블리 로더 이벤트
- dotnet-trace
- PerfView

플러그 인을 사용하여 .NET Core 애플리케이션 만들기

이 자습서에서는 플러그 인을 로드하는 사용자 지정 AssemblyLoadContext 만드는 방법을 보여 줍니다. AssemblyDependencyResolver 플러그 인의 종속성을 확인하는 데 사용됩니다. 이 자습서에서는 플러그 인의 종속성에 대한 별도의 어셈블리 컨텍스트를 제공하여 플러그 인과 호스팅 애플리케이션 간에 서로 다른 어셈블리 종속성을 허용합니다. 다음을 배우게 됩니다:

- 플러그 인을 지원하도록 프로젝트를 구성합니다.
- 사용자 지정 AssemblyLoadContext를 생성하여 각 플러그인을 로드합니다.
- System.Runtime.Loader.AssemblyDependencyResolver 형식을 사용하여 플러그 인이 종속성을 가질 수 있도록 합니다.
- 빌드 아티팩트만 복사하여 쉽게 배포할 수 있는 플러그 인을 작성합니다.

> **참고 항목**
>
> 신뢰할 수 없는 코드는 신뢰할 수 있는 .NET 프로세스에 안전하게 로드할 수 없습니다. 보안 또는 안정성 경계를 제공하려면 OS 또는 가상화 플랫폼에서 제공하는 기술을 고려합니다.

필수 조건

- 최신 .NET SDK
- Visual Studio Code 편집기
- C# 개발 키트

애플리케이션 만들기

첫 번째 단계는 애플리케이션을 만드는 것입니다.

1. 새 폴더를 만들고 해당 폴더에서 다음 명령을 실행합니다.

   .NET CLI

   ```
   dotnet new console -o AppWithPlugin
   ```

2. 프로젝트를 더 쉽게 빌드하려면 동일한 폴더에 Visual Studio 솔루션 파일을 만듭니다. 다음 명령을 실행합니다.

   .NET CLI

```
dotnet new sln
```

3. 다음 명령을 실행하여 솔루션에 앱 프로젝트를 추가합니다.

.NET CLI

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

이제 애플리케이션의 구조를 채울 수 있습니다. *AppWithPlugin/Program.cs* 파일의 코드를 다음 코드로 바꿉다.

C#

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Skip command-line switches/flags (arguments starting with
'/' or '-')
                        if (commandName.StartsWith("/") || commandName.StartsWith("-
"))
                        {
```

```
                            Console.WriteLine($"Skipping command-line flag:
{commandName}");
                            continue;
                        }

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

플러그 인 인터페이스 만들기

플러그 인을 사용하여 앱을 빌드하는 다음 단계는 플러그 인이 구현해야 하는 인터페이스를 정의하는 것입니다. 앱과 플러그 인 간의 통신에 사용할 형식을 포함하는 클래스 라이브러리를 만드는 것이 좋습니다. 이 부서를 사용하면 전체 애플리케이션을 배송하지 않고도 플러그 인 인터페이스를 패키지로 게시할 수 있습니다.

프로젝트의 루트 폴더에서 `dotnet new classlib -o PluginBase`실행합니다. 또한 `dotnet sln add PluginBase/PluginBase.csproj` 실행하여 솔루션 파일에 프로젝트를 추가합니다. `PluginBase/Class1.cs` 파일을 삭제하고 다음 인터페이스 정의를 사용하여 `PluginBase` `ICommand.cs` 폴더에 새 파일을 만듭니다.

C#

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

이 `ICommand` 인터페이스는 모든 플러그 인이 구현할 인터페이스입니다.

이제 `ICommand` 인터페이스가 정의되었으므로 애플리케이션 프로젝트를 조금 더 채울 수 있습니다. 루트 폴더의 `AppWithPlugin` 명령을 사용하여 `PluginBase` 프로젝트의 참조를 `dotnet add`

`AppWithPlugin/AppWithPlugin.csproj reference PluginBase/PluginBase.csproj` 프로젝트에 추가합니다.

지정된 파일 경로에서 플러그 인을 로드할 수 있도록 `// Load commands from plugins` 주석을 다음 코드 조각으로 바꿉니다.

C#

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

그런 다음, `// Output the loaded commands` 주석을 다음 코드 조각으로 바꿉다.

C#

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

`// Execute the command with the name passed as an argument` 주석을 다음 코드 조각으로 바꿉다.

C#

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine($"No such command is known: {commandName}");
    continue;
}

command.Execute();
```

마지막으로 다음과 같이 `Program` 및 `LoadPlugin` `CreateCommands` 클래스에 정적 메서드를 추가합니다.

C#

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (var type in assembly.GetTypes().Where(t =>
typeof(ICommand).IsAssignableFrom(t)))
    {
        if (Activator.CreateInstance(type) is ICommand result)
        {
            count++;
            yield return result;
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",", assembly.GetTypes().Select(t =>
t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from
{assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

플러그 인 로드

이제 애플리케이션이 로드된 플러그 인 어셈블리에서 명령을 올바르게 로드하고 인스턴스화할 수 있지만 플러그 인 어셈블리를 로드할 수는 없습니다. 다음 내용이 포함된 *AppWithPlugin* 폴더에 *PluginLoadContext.cs* 파일을 만듭니다.

C#

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
```

```
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath =
_resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

`PluginLoadContext` 형식은 AssemblyLoadContext파생됩니다. `AssemblyLoadContext` 형식은 개발자가 로드된 어셈블리를 다른 그룹으로 격리하여 어셈블리 버전이 충돌하지 않도록 하는 런타임의 특수 형식입니다. 또한 사용자 지정 `AssemblyLoadContext` 어셈블리를 로드할 다른 경로를 선택하고 기본 동작을 재정의할 수 있습니다. `PluginLoadContext`은 .NET Core 3.0에서 도입된 `AssemblyDependencyResolver` 유형의 인스턴스를 사용하여 어셈블리 이름을 경로로 변환합니다. `AssemblyDependencyResolver` 개체는 .NET 클래스 라이브러리의 경로를 사용하여 생성됩니다. 생성자에 전달된 클래스 라이브러리 파일의 경로를 기준으로, `AssemblyDependencyResolver` 파일을 통해 어셈블리 및 네이티브 라이브러리를 해당 상대 경로로 해결합니다. 사용자 지정 `AssemblyLoadContext` 사용하면 플러그 인이 자체 종속성을 가질 수 있으며 `AssemblyDependencyResolver` 종속성을 쉽게 로드할 수 있습니다.

이제 `AppWithPlugin` 프로젝트에 `PluginLoadContext` 형식이 있으므로 `Program.LoadPlugin` 메서드를 다음 본문으로 업데이트합니다.

C#

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(
        Path.Combine(typeof(Program).Assembly.Location, "..", "..", "..", "..",
"..", ".."));
```

```
    string pluginLocation = Path.GetFullPath(Path.Combine(root,
relativePath.Replace('\\', Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new(pluginLocation);
    return
loadContext.LoadFromAssemblyName(new(Path.GetFileNameWithoutExtension(pluginLocation
)));
}
```

각 플러그 인에 대해 다른 `PluginLoadContext` 인스턴스를 사용하면 플러그 인에 문제 없이 다른 종속성 또는 충돌하는 종속성이 있을 수 있습니다.

종속성이 없는 간단한 플러그 인

루트 폴더로 돌아가서 다음을 수행합니다.

1. 다음 명령을 실행하여 `HelloPlugin`새 클래스 라이브러리 프로젝트를 만듭니다.

   .NET CLI

   ```
   dotnet new classlib -o HelloPlugin
   ```

2. 다음 명령을 실행하여 `AppWithPlugin` 솔루션에 프로젝트를 추가합니다.

   .NET CLI

   ```
   dotnet sln add HelloPlugin/HelloPlugin.csproj
   ```

3. *HelloPlugin/Class1.cs* 파일을 다음 내용으로 *HelloCommand.cs* 파일로 바꿉니다.

C#

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
```

```
    }
}
```

이제 *HelloPlugin.csproj* 파일을 엽니다. 다음과 같이 표시되어야 합니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

`<PropertyGroup>` 태그 사이에 다음 요소를 추가합니다.

XML

```xml
  <EnableDynamicLoading>true</EnableDynamicLoading>
```

`<EnableDynamicLoading>true</EnableDynamicLoading>` 플러그 인으로 사용할 수 있도록 프로젝트를 준비합니다. 무엇보다도 모든 종속성을 프로젝트의 출력에 복사합니다. 자세한 내용은 EnableDynamicLoading참조하세요.

`<Project>` 태그 사이에 다음 요소를 추가합니다.

XML

```xml
<ItemGroup>
    <ProjectReference Include="..\PluginBase\PluginBase.csproj">
        <Private>false</Private>
        <ExcludeAssets>runtime</ExcludeAssets>
    </ProjectReference>
</ItemGroup>
```

`<Private>false</Private>` 요소가 중요합니다. 그러면 MSBuild에서 HelloPlugin의 출력 디렉터리에 *PluginBase.dll* 복사하지 않도록 지시합니다. *PluginBase.dll* 어셈블리가 출력 디렉터리에 있는 경우 `PluginLoadContext` 어셈블리를 찾아서 *HelloPlugin.dll* 어셈블리를 로드할 때 로드합니다. 이 시점에서 `HelloPlugin.HelloCommand` 형식은 기본 로드 컨텍스트에 로드되는 `ICommand` 인터페이스가 아니라 프로젝트의 출력 디렉터리에 있는 `HelloPlugin` `ICommand` 인터페이스를 구현합니다. 런타임에서 이러한 두 형식을 서로 다른 어셈블리의 다른 형식으로 간주하므로 `AppWithPlugin.Program.CreateCommands` 메서드는 명령을 찾을 수 없습니다. 따라서 플러그 인 인

터페이스를 포함하는 어셈블리에 대한 참조에는 `<Private>false</Private>` 메타데이터가 필요합니다.

마찬가지로 `<ExcludeAssets>runtime</ExcludeAssets>` 다른 패키지를 참조하는 경우에도 `PluginBase` 요소가 중요합니다. 이 설정은 `<Private>false</Private>` 효과와 동일하지만 `PluginBase` 프로젝트 또는 해당 종속성 중 하나에 포함될 수 있는 패키지 참조에서 작동합니다.

이제 `HelloPlugin` 프로젝트가 완료되었으므로 `AppWithPlugin` 플러그 인을 찾을 수 있는 위치를 알 수 있도록 `HelloPlugin` 프로젝트를 업데이트해야 합니다. `// Paths to plugins to load` 주석 후에 `@"HelloPlugin\bin\Debug\net10.0\HelloPlugin.dll"` 배열의 요소로 `pluginPaths` 추가합니다(이 경로는 사용하는 .NET Core 버전에 따라 다를 수 있음).

라이브러리 종속성이 있는 플러그 인

거의 모든 플러그 인은 간단한 "Hello World"보다 더 복잡하며 많은 플러그 인은 다른 라이브러리에 종속되어 있습니다. 샘플의 `JsonPlugin` 및 `OldJsonPlugin` 프로젝트는 `Newtonsoft.Json` NuGet 패키지 종속성이 있는 플러그 인의 두 가지 예를 보여 줍니다. 이 때문에 모든 플러그 인 프로젝트는 모든 종속성을 `<EnableDynamicLoading>true</EnableDynamicLoading>` 출력에 복사하도록 프로젝트 속성에 `dotnet build` 추가해야 합니다. `dotnet publish` 사용하여 클래스 라이브러리를 게시하면 모든 종속성이 게시 출력에 복사됩니다.

샘플의 다른 예제

이 자습서의 전체 소스 코드는 dotnet/samples 리포지토리 찾을 수 있습니다. 완성된 샘플에는 `AssemblyDependencyResolver` 동작의 몇 가지 다른 예제가 포함되어 있습니다. 예를 들어 `AssemblyDependencyResolver` 개체는 NuGet 패키지에 포함된 지역화된 위성 어셈블리뿐만 아니라 네이티브 라이브러리도 확인할 수 있습니다. 샘플 리포지토리의 `UVPlugin` 및 `FrenchPlugin` 이러한 시나리오를 보여 줍니다.

NuGet 패키지에서 플러그 인 인터페이스 참조

`A.PluginBase` NuGet 패키지에 정의된 플러그 인 인터페이스가 있는 앱 A가 있다고 가정해 보겠습니다. 플러그 인 프로젝트에서 패키지를 올바르게 참조하려면 어떻게 해야 합니까? 프로젝트 참조의 경우 프로젝트 파일의 `<Private>false</Private>` 요소에 `ProjectReference` 메타데이터를 사용하면 dll이 출력에 복사되지 않습니다.

`A.PluginBase` 패키지를 올바르게 참조하려면 프로젝트 파일의 `<PackageReference>` 요소를 다음으로 변경하려고 합니다.

XML

```xml
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

이렇게 하면 `A.PluginBase` 어셈블리가 플러그 인의 출력 디렉터리에 복사되지 않고 플러그 인에서 A 버전의 `A.PluginBase`사용할 수 있습니다.

플러그 인 대상 프레임워크 권장 사항

플러그 인 종속성 로드는 *.deps.json* 파일을 사용하므로 플러그 인의 대상 프레임워크와 관련된 문제가 있습니다. 특히 플러그 인은 .NET Standard 버전 대신 .NET 10과 같은 런타임을 대상으로 해야 합니다. *.deps.json* 파일은 프로젝트가 대상으로 하는 프레임워크에 따라 생성되며, 많은 .NET Standard 호환 패키지가 특정 런타임에 대해 .NET Standard 및 구현 어셈블리에 대해 빌드하기 위한 참조 어셈블리를 제공하므로 *.deps.json* 구현 어셈블리가 올바르게 표시되지 않거나 예상하는 .NET Core 버전 대신 어셈블리의 .NET Standard 버전을 사용할 수 있습니다.

플러그 인 프레임워크 참조

현재 플러그 인은 프로세스에 새 프레임워크를 도입할 수 없습니다. 예를 들어 `Microsoft.AspNetCore.App` 프레임워크를 사용하는 플러그 인을 루트 `Microsoft.NETCore.App` 프레임워크만 사용하는 애플리케이션에 로드할 수 없습니다. 호스트 애플리케이션은 플러그 인에 필요한 모든 프레임워크에 대한 참조를 선언해야 합니다.

Last updated on 2026. 02. 04.

.NET에서 어셈블리 언로드 가능성을 사용하고 디버그하는 방법

.NET(Core)에는 어셈블리 집합을 로드하고 나중에 언로드하는 기능이 도입되었습니다. .NET Framework에서 사용자 지정 앱 도메인은 이 용도로 사용되었지만 .NET(Core)은 단일 기본 앱 도메인만 지원합니다.

언로드 기능은 AssemblyLoadContext을 통해 지원됩니다. 어셈블리 집합을 수집 가능한 `AssemblyLoadContext`으로 로드하고, 메서드를 실행하거나 리플렉션을 사용하여 검사하며, 마지막으로 `AssemblyLoadContext`을 언로드합니다. 로드된 어셈블리를 `AssemblyLoadContext`에서 언로드합니다.

AppDomains를 사용하는 `AssemblyLoadContext` 언로드와 사용 간에는 한 가지 중요한 차이점이 있습니다. AppDomains를 사용하면 언로드가 강제로 적용됩니다. 언로드 시 대상 AppDomain에서 실행되는 모든 스레드가 중단되고, 대상 AppDomain에서 만든 관리되는 COM 개체가 제거됩니다. `AssemblyLoadContext`에서는 언로드가 "협력적"입니다. 메서드를 호출하면 AssemblyLoadContext.Unload 언로드가 시작됩니다. 다음 후에 언로드가 완료됩니다.

- 스레드에는 호출 스택에 로드된 어셈블리의 `AssemblyLoadContext` 메서드가 없습니다.
- 어셈블리에 로드된 `AssemblyLoadContext`의 모든 형식, 그 형식의 인스턴스 및 어셈블리 자체는 다음에서 참조되지 않습니다.
  - `AssemblyLoadContext` 외부의 참조, 단 약한 참조(WeakReference 또는 WeakReference<T>)는 제외합니다.
  - 내부 및 외부에서 강력한 GC(가비지 수집기) 핸들(GCHandleType.Normal 또는 `AssemblyLoadContext`)을 처리합니다.

collectible AssemblyLoadContext를 사용

이 섹션에는 .NET(Core) 애플리케이션을 수집 가능한 `AssemblyLoadContext`애플리케이션으로 로드하고 진입점을 실행한 다음 언로드하는 간단한 방법을 보여 주는 자세한 단계별 자습서가 포함되어 있습니다. 에서 전체 샘플을 https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading 찾을 수 있습니다.

수집 가능한 AssemblyLoadContext 만들기

AssemblyLoadContext 클래스를 상속받아 AssemblyLoadContext.Load 메서드를 재정의하십시오. 이 메서드는 해당 `AssemblyLoadContext`에 로드된 어셈블리의 종속성인 모든 어셈블리에 대한 참조를 처리합니다.

다음 코드는 가장 간단한 사용자 지정 `AssemblyLoadContext`의 예입니다.

C#

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {
    }

    protected override Assembly? Load(AssemblyName name)
    {
        return null;
    }
}
```

보시다시피 `Load` 메서드가 `null`을 반환합니다. 즉, 모든 종속성 어셈블리가 기본 컨텍스트로 로드되고 새 컨텍스트에 명시적으로 로드된 어셈블리만 포함됩니다.

종속성의 `AssemblyLoadContext` 일부 또는 전부를 로드하려면 `Load` 메서드에서 `AssemblyDependencyResolver`을 사용할 수 있습니다. `AssemblyDependencyResolver` 어셈블리 이름을 절대 어셈블리 파일 경로로 변환합니다. 확인자는 컨텍스트에 로드된 주 어셈블리의 디렉터리에 있는 *.deps.json* 파일 및 어셈블리 파일을 사용합니다.

C#

```
using System.Reflection;
using System.Runtime.Loader;

namespace complex
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) :
base(isCollectible: true)
        {
            _resolver = new AssemblyDependencyResolver(mainAssemblyToLoadPath);
        }

        protected override Assembly? Load(AssemblyName name)
        {
            string? assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
```

```
    }
}
```

사용자 지정 수집 가능한 AssemblyLoadContext 사용

이 섹션에서는 더 간단한 버전의 `TestAssemblyLoadContext` 사용 중이라고 가정합니다.

다음과 같이 사용자 지정 `AssemblyLoadContext` 인스턴스를 만들고 어셈블리를 로드할 수 있습니다.

C#

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

로드된 어셈블리에서 참조하는 각 어셈블리에 대해 어셈블리 `TestAssemblyLoadContext.Load` 를 가져올 위치를 결정할 수 있도록 `TestAssemblyLoadContext` 메서드가 호출됩니다. 이 경우, 런타임이 기본적으로 어셈블리를 로드하는 데 사용하는 위치에서 기본 컨텍스트로 로드되어야 한다는 것을 나타내기 위해 `null`를 반환합니다.

이제 어셈블리가 로드되었으므로 해당 어셈블리에서 메서드를 실행할 수 있습니다. 메서드를 실행합니다.`Main`

C#

```
var args = new object[1] {new string[] {"Hello"}};
_ = a.EntryPoint?.Invoke(null, args);
```

`Main` 메서드가 반환되면 사용자 지정 `AssemblyLoadContext`에서 `Unload` 메서드를 호출하거나 가지고 있는 `AssemblyLoadContext`에 대한 참조를 제거하여 언로드를 시작할 수 있습니다.

C#

```
alc.Unload();
```

테스트 어셈블리를 언로드하기에 충분합니다. 다음으로, `TestAssemblyLoadContext`, `Assembly`, 및 `MethodInfo`를 스택 슬롯 참조(실제 로컬 또는 JIT에서 도입된 로컬)를 통해 활성 상태로 유지할 수 없도록 `Assembly.EntryPoint`(인라인할 수 없는 별도의 메서드)에 배치하십시오. `TestAssemblyLoadContext`을(를) 활성 상태로 유지하고 언로드를 방지할 수 있습니다.

또한, 나중에 언로드 완료를 감지하기 위해 사용할 수 있도록 `AssemblyLoadContext`에 대한 약한 참조를 반환합니다.

C#

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void ExecuteAndUnload(string assemblyPath, out WeakReference alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    _ = a.EntryPoint?.Invoke(null, args);

    alc.Unload();
}
```

이제 이 함수를 실행하여 어셈블리를 로드, 실행 및 언로드할 수 있습니다.

C#

```
WeakReference testAlcWeakRef;
ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

그러나 언로드가 즉시 완료되지는 않습니다. 앞에서 설명한 것처럼 가비지 수집기를 사용하여 테스트 어셈블리에서 모든 개체를 수집합니다. 대부분의 경우 언로드 완료를 기다릴 필요가 없습니다. 그러나 언로드가 완료되었음을 아는 것이 유용한 경우가 있습니다. 예를 들어 디스크에서 사용자 지정 `AssemblyLoadContext` 에 로드된 어셈블리 파일을 삭제할 수 있습니다. 이러한 경우 다음 코드 조각을 사용할 수 있습니다. 가비지 수집을 시작하고 사용자 지정 `AssemblyLoadContext`에 대한 약한 참조를 설정하여 `null`가 될 때까지 루프에서 대기하면서 보류 중인 종료자를 처리합니다. 이 작업은 대상 개체가 수집되었음을 나타냅니다. 대부분의 경우 루프를 한 번만 통과해야 합니다. 그러나 `AssemblyLoadContext`에서 실행되는 코드가 생성한 개체가 종료자를 가진 경우처럼 더 복잡한 사례에서는 더 많은 패스가 필요할 수 있습니다.

C#

```
for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

제한점

수집 가능한 어셈블리에 로드된 어셈블리는 수집 가능한 `AssemblyLoadContext`어셈블리에 대한 일반적인 제한을 준수해야 합니다. 다음 제한 사항이 추가로 적용됩니다.

- C++/CLI로 작성된 어셈블리는 지원되지 않습니다.
- ReadyToRun 생성 코드는 무시됩니다.

언로드 이벤트

경우에 따라 언로드가 시작될 때 사용자 지정 `AssemblyLoadContext` 에 로드된 코드가 일부 정리를 수행해야 할 수 있습니다. 예를 들어 스레드를 중지하거나 강한 GC 핸들을 처리해야 할 수 있습니다. 이러한 `Unloading` 경우 이벤트를 사용할 수 있습니다. 이 이벤트에 필요한 정리를 수행하는 처리기를 후크할 수 있습니다.

언로드 불가능 문제 해결

언로드의 협조적 특성으로 인해 물건을 수집 가능한 `AssemblyLoadContext` 상태로 유지하고 언로드를 방지할 수 있는 참조를 잊기 쉽습니다. 참조를 보유할 수 있는 엔터티는 다음과 같습니다 (일부는 명백하지 않을 수 있습니다).

- 스택 슬롯 또는 프로세서 레지스터(사용자 코드에서 명시적으로 만들거나 JIT(Just-In-Time) 컴파일러에서 암시적으로 만든 메서드 로컬), 정적 변수, 강력한(고정된) GC 핸들에 있는 일반 참조는, 수집 가능한 `AssemblyLoadContext` 외부에서 유지되며, 전이적으로 가리킵니다.
  - 어셈블리가 수집 가능한 모듈에 로드되었습니다 `AssemblyLoadContext`.
  - 이러한 어셈블리의 유형입니다.
  - 이렇게 어셈블리로부터 나온 형식의 한 인스턴스.
- 수집 가능한 `AssemblyLoadContext`어셈블리에 로드된 어셈블리에서 코드를 실행하는 스레드입니다.
- 수집할 수 있는 `AssemblyLoadContext` 내부에서 생성된 사용자 지정 `AssemblyLoadContext` 형식의 수집할 수 없는 인스턴스입니다.
- RegisteredWaitHandle 인스턴스 보류 중, 콜백이 사용자 지정 `AssemblyLoadContext`의 메서드로 설정됨.
- 로드 가능한`AssemblyLoadContext`에 로드된 형식의 어셈블리, 형식 또는 인스턴스를 참조하는 사용자 지정 `AssemblyLoadContext` 하위 클래스의 필드입니다. 언로드가 진행되는 동안 런타임은 강력한 GC 핸들을 `AssemblyLoadContext`에 보유하여 언로드를 조정합니다. 즉, GC는 직접 참조를 삭제한 후에도 해당 필드 참조를 `AssemblyLoadContext`수집하지 않습니다. 언로드를 완료할 수 있도록 이러한 필드를 지웁다.

> **팁**
>
> 스택 슬롯 또는 프로세서 레지스터에 저장되고 언로드 `AssemblyLoadContext` 를 방지할 수 있는 개체 참조는 다음과 같은 경우에 발생할 수 있습니다.

- 사용자가 만든 지역 변수가 없더라도 함수 호출 결과가 다른 함수에 직접 전달되는 경우
- JIT 컴파일러가 메서드의 특정 지점에서 사용할 수 있는 개체에 대한 참조를 유지하는 경우

언로드 문제 디버깅

언로드와 관련된 디버깅 문제는 지루할 수 있습니다. 오브젝트가 살아 있는 상태를 유지하도록 하는 것이 무엇인지 알 수 없는 상황에 빠질 수 있지만, 그러나 언로드에는 실패합니다. 이를 도와주는 가장 좋은 도구는 SOS 플러그 인을 사용하는 WinDbg(또는 Unix의 LLDB)입니다. 특정 `AssemblyLoadContext`에 속한 `LoaderAllocator`가 계속 유지되는 원인을 찾아야 합니다. SOS 플러그 인을 사용하면 GC 힙 개체, 해당 계층 구조 및 루트를 볼 수 있습니다.

SOS 플러그 인을 디버거에 로드하려면 디버거 명령줄에 다음 명령 중 하나를 입력합니다.

WinDbg에서(아직 로드되지 않은 경우):

콘솔

```
.loadby sos coreclr
```

LLDB에서:

콘솔

```
plugin load /path/to/libsosplugin.so
```

이제 언로드에 문제가 있는 예제 프로그램을 디버그합니다. 소스 코드는 예제 소스 코드 섹션에서 사용할 수 있습니다. WinDbg에서 실행하면 프로그램이 언로드 성공 여부를 확인한 직후 디버거로 중단됩니다. 그런 다음 범인을 찾고 시작할 수 있습니다.

> **팁**
>
> Unix에서 LLDB를 사용하여 디버그하는 경우, 다음 예제의 SOS 명령 앞에 `!`가 없습니다.

콘솔

```
!dumpheap -type LoaderAllocator
```

이 명령은 GC 힙에 있는 형식 이름을 가진 `LoaderAllocator` 모든 개체를 덤프합니다. 예제는 다음과 같습니다.

콘솔

```
         Address               MT     Size
000002b78000ce40 00007ffadc93a288       48
000002b78000ceb0 00007ffadc93a218       24

Statistics:
              MT    Count    TotalSize Class Name
00007ffadc93a218        1           24 System.Reflection.LoaderAllocatorScout
00007ffadc93a288        1           48 System.Reflection.LoaderAllocator
Total 2 objects
```

"'통계:' 부분에서 관심 있는 개체인 `System.Reflection.LoaderAllocator`에 속하는 `MT` (`MethodTable`)을 확인합니다." 그런 다음, 시작 부분의 목록에서 해당 항목과 `MT` 일치하는 항목을 찾고 개체 자체의 주소를 가져옵니다. 이 경우 "000002b78000ce40"입니다.

이제 개체의 `LoaderAllocator` 주소를 알게 되었으므로 다른 명령을 사용하여 해당 GC 루트를 찾을 수 있습니다.

콘솔

```
!gcroot 0x000002b78000ce40
```

이 명령은 `LoaderAllocator` 인스턴스로 이어지는 개체 참조 체인을 덤프합니다. 목록은 루트로 시작합니다. 루트는 `LoaderAllocator`의 생명력을 유지하는 엔터티로, 문제의 핵심입니다. 루트는 스택 슬롯, 프로세서 레지스터, GC 핸들 또는 정적 변수일 수 있습니다.

다음은 명령 출력의 예입니다.`gcroot`

콘솔

```
Thread 4ac:
    000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
        rbp-20: 000000cf9499dd90
            ->  000002b78000d328 System.Reflection.RuntimeMethodInfo
            ->  000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
            ->  000002b78000d1d0 System.RuntimeType
            ->  000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
    000002b7f8a81198 (strong handle)
    -> 000002b78000d948 test.Test
    -> 000002b78000ce40 System.Reflection.LoaderAllocator
```

```
    000002b7f8a815f8 (pinned handle)
    -> 000002b790001038 System.Object[]
    -> 000002b78000d390 example.TestInfo
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.
```

다음 단계는 루트가 있는 위치를 파악하여 해결할 수 있도록 하는 것입니다. 가장 쉬운 경우는 루트가 스택 슬롯 또는 프로세서 레지스터인 경우입니다. 이 경우 `gcroot` 프레임에 루트가 포함된 함수의 이름과 해당 함수를 실행하는 스레드가 표시됩니다. 어려운 경우는 루트가 정적 변수 또는 GC 핸들인 경우입니다.

이전 예제에서 첫 번째 루트는 주소 `System.Reflection.RuntimeMethodInfo` 에 있는 함수 `example.Program.Main(System.String[])` 프레임에 저장된 형식 `rbp-20` 의 로컬입니다(`rbp`프로세서 레지스터 `rbp` 이고 -20 해당 레지스터의 16진수 오프셋임).

두 번째 루트는 클래스 인스턴스 `GCHandle` 에 대한 참조를 보유하는 일반(강력한) `test.Test` 입니다.

세 번째 루트는 고정된 `GCHandle`입니다. 이것은 실제로 정적 변수이지만 불행히도 알 수있는 방법은 없습니다. 참조 형식에 대한 정적은 내부 런타임 구조의 관리되는 개체 배열에 저장됩니다.

언로드를 방지할 수 있는 또 다른 경우는 스레드의 스택에 `AssemblyLoadContext`에 로드된 어셈블리의 메서드 프레임이 있을 때입니다. 모든 스레드의 관리되는 호출 스택을 덤프하여 확인할 수 있습니다.

콘솔

```
~*e !clrstack
```

명령은 "`!clrstack` 명령을 모든 스레드에 적용"을 의미합니다. 다음은 예제에 대한 해당 명령의 출력입니다. 아쉽게도 Unix의 LLDB에는 모든 스레드에 명령을 적용할 수 있는 방법이 없으므로 스레드를 수동으로 전환하고 명령을 반복 `clrstack` 해야 합니다. 디버거가 "관리 스택을 탐색할 수 없음"이라고 표시된 모든 스레드를 무시합니다.

콘솔

```
OS Thread Id: 0x6ba8 (0)
        Child SP               IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[])
```

```
[E:\unloadability\example\Program.cs @ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
        Child SP               IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame: 0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
        Child SP               IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
        Child SP               IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc()
[E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame: 0000001fc727f7f0]
```

볼 수 있듯이 마지막 스레드에는 `test.Program.ThreadProc()`. 이 함수는 로드된 어셈블리의 `AssemblyLoadContext`로부터 온 것이므로 `AssemblyLoadContext`을(를) 활성 상태로 유지합니다.

예제 소스 코드

언로드 가능성 문제가 포함된 다음 코드는 이전 디버깅 예제에서 사용됩니다.

주 테스트 프로그램

C#

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;

namespace example
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly? Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo? mi)
        {
            _entryPoint = mi;
        }

        MethodInfo? _entryPoint;
    }

    class Program
    {
        static TestInfo? entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference
testAlcWeakRef, out MethodInfo? testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a method
```

```
            // for an assembly loaded into the TestAssemblyLoadContext in a static
variable.
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            var oResult = a.EntryPoint?.Invoke(null, args);
            alc.Unload();
            return (oResult is int result) ? result : -1;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a method for
an assembly loaded into the TestAssemblyLoadContext in a local variable
            MethodInfo? testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out
testAlcWeakRef, out testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();

            Console.WriteLine($"Test completed, result={result}, entryPoint:
{testEntryPoint} unload success: {!testAlcWeakRef.IsAlive}");
        }
    }
}
```

TestAssemblyLoadContext에 로드된 프로그램

다음 코드는 주 테스트 프로그램의 메서드에 전달된 `ExecuteAndUnload` 나타냅니다.

C#

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

namespace test
{
    class Test
    {
    }

    class Program
    {
        public static void ThreadProc()
```

```
        {
            // Issue preventing unloading #4 - a thread running method inside of
the TestAssemblyLoadContext at the unload time
            Thread.Sleep(Timeout.Infinite);
        }

        static GCHandle handle;
        static int Main(string[] args)
        {
            // Issue preventing unloading #3 - normal GC handle
            handle = GCHandle.Alloc(new Test());
            Thread t = new Thread(new ThreadStart(ThreadProc));
            t.IsBackground = true;
            t.Start();
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");

            return 1;
        }
    }
}
```

ⓘ **참고**: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 19.

MEF(관리 확장성 프레임워크)

이 문서에서는 .NET Framework 4에 도입된 관리되는 확장성 프레임워크의 개요를 제공합니다.

MEF란?

MEF(Managed Extensibility Framework)는 가볍고 확장 가능한 애플리케이션을 만들기 위한 라이브러리입니다. 이를 통해 애플리케이션 개발자는 구성 없이 확장을 검색하고 사용할 수 있습니다. 또한 확장 개발자는 코드를 쉽게 캡슐화하고 취약한 하드 종속성을 방지할 수 있습니다. MEF를 사용하면 애플리케이션 내에서뿐만 아니라 애플리케이션 간에도 확장을 다시 사용할 수 있습니다.

확장성 문제

확장성을 지원해야 하는 대규모 애플리케이션의 설계자라고 상상해 보십시오. 애플리케이션은 잠재적으로 많은 수의 더 작은 구성 요소를 포함해야 하며, 구성 요소를 만들고 실행하는 역할을 담당합니다.

문제에 대한 가장 간단한 방법은 구성 요소를 애플리케이션에 소스 코드로 포함하고 코드에서 직접 호출하는 것입니다. 여기에는 여러 가지 명백한 단점이 있습니다. 가장 중요한 것은 소스 코드를 수정하지 않고는 새 구성 요소를 추가할 수 없다는 것입니다. 예를 들어 웹 애플리케이션에서 허용될 수 있지만 클라이언트 애플리케이션에서는 작동하지 않을 수 있는 제한 사항입니다. 마찬가지로, 구성 요소가 타사에서 개발되어 소스 코드에 접근할 수 없을 수도 있고 같은 이유로 그들이 당신의 소스 코드에 접근하는 것을 허용할 수 없습니다.

약간 더 정교한 방법은 확장 지점 또는 인터페이스를 제공하여 애플리케이션과 해당 구성 요소 간의 분리를 허용하는 것입니다. 이 모델에서는 구성 요소가 구현할 수 있는 인터페이스와 애플리케이션과 상호 작용할 수 있도록 하는 API를 제공할 수 있습니다. 이렇게 하면 소스 코드 액세스가 필요한 문제가 해결되지만 여전히 고유한 문제가 있습니다.

애플리케이션은 자체적으로 구성 요소를 검색할 수 있는 용량이 부족하기 때문에 사용 가능한 구성 요소를 명시적으로 알려주어야 하며 로드해야 합니다. 이 작업은 일반적으로 구성 파일에 사용 가능한 구성 요소를 명시적으로 등록하여 수행됩니다. 즉, 구성 요소가 올바른지 확인하면 유지 관리 문제가 됩니다. 특히 업데이트를 수행할 것으로 예상되는 개발자가 아닌 최종 사용자인 경우 더욱 그러합니다.

또한 구성 요소는 애플리케이션 자체의 엄격하게 정의된 채널을 제외하고는 서로 통신할 수 없습니다. 애플리케이션 설계자가 특정 통신의 필요성을 예상하지 못한 경우 일반적으로 불가능합니다.

마지막으로 구성 요소 개발자는 구현하는 인터페이스가 포함된 어셈블리에 대한 하드 종속성을 허용해야 합니다. 따라서 둘 이상의 애플리케이션에서 구성 요소를 사용하는 것이 어렵고 구성

요소에 대한 테스트 프레임워크를 만들 때 문제가 발생할 수도 있습니다.

MEF에서 제공하는 내용

MEF는 사용 가능한 구성 요소를 명시적으로 등록하는 대신 *컴퍼지션*을 통해 암시적으로 검색할 수 있는 방법을 제공합니다. *파트*라고 하는 MEF 구성 요소는 종속성(*가져오기*라고 함)과 사용할 수 있는 기능(내보내기라고 함)을 선언적으로 지정*합니다*. 부품(컴포넌트)이 생성될 때, MEF 구성 엔진은 다른 컴포넌트에서 제공 가능한 것으로 해당 요구 사항을 충족합니다.

이 방법은 이전 섹션에서 설명한 문제를 해결합니다. MEF 파트는 해당 기능을 선언적으로 지정하기 때문에 런타임에 검색할 수 있습니다. 즉, 애플리케이션이 하드 코딩된 참조 또는 취약한 구성 파일 없이 파트를 사용할 수 있습니다. MEF를 사용하면 애플리케이션을 인스턴스화하거나 어셈블리를 로드하지 않고도 메타데이터로 파트를 검색하고 검사할 수 있습니다. 따라서 확장을 로드해야 하는 시기와 방법을 신중하게 지정할 필요가 없습니다.

제공된 내보내기 외에도 파트는 가져오기를 지정할 수 있으며, 이 항목은 다른 파트로 채워집니다. 이렇게 하면 파트 간 통신이 가능할 뿐만 아니라 쉬워지고, 코드의 좋은 분할을 가능하게 합니다. 예를 들어 많은 구성 요소에 공통된 서비스는 별도의 부분으로 인계되고 쉽게 수정되거나 교체될 수 있습니다.

MEF 모델은 특정 애플리케이션 어셈블리에 대한 하드 종속성이 필요하지 않으므로 애플리케이션에서 애플리케이션으로 확장을 다시 사용할 수 있습니다. 또한 애플리케이션과 독립적으로 테스트 하네스를 쉽게 개발하여 확장 구성 요소를 테스트할 수 있습니다.

MEF를 사용하여 작성된 확장 가능 애플리케이션은 확장 구성 요소로 채울 수 있는 가져오기를 선언하고 애플리케이션 서비스를 확장에 노출하기 위해 내보내기를 선언할 수도 있습니다. 각 확장 구성 요소는 내보내기 선언 및 가져오기를 선언할 수도 있습니다. 이러한 방식으로 확장 구성 요소 자체는 자동으로 확장할 수 있습니다.

MEF를 사용할 수 있는 위치

MEF는 .NET Framework 4 이상 버전 및 .NET 5 이상 버전에서 사용할 수 있습니다. Windows Forms, WPF 또는 기타 기술을 사용하는지 또는 ASP.NET 사용하는 서버 애플리케이션에서 클라이언트 애플리케이션에서 MEF를 사용할 수 있습니다.

MEF 및 MAF

이전 버전의 .NET Framework는 애플리케이션이 확장을 격리하고 관리할 수 있도록 설계된 MAF(관리되는 추가 기능 프레임워크)를 도입했습니다. MAF의 초점은 MEF보다 약간 더 높으며 확장 격리 및 어셈블리 로드 및 언로드에 중점을 두고 있으며 MEF의 초점은 검색 기능, 확장성

및 이식성에 있습니다. 두 프레임워크는 원활하게 상호 운용되며 단일 애플리케이션이 둘 다 활용할 수 있습니다.

SimpleCalculator: 예제 애플리케이션

MEF에서 수행할 수 있는 작업을 확인하는 가장 간단한 방법은 간단한 MEF 애플리케이션을 빌드하는 것입니다. 이 예제에서는 SimpleCalculator라는 매우 간단한 계산기를 빌드합니다. SimpleCalculator의 목표는 기본 산술 명령을 "5+3" 또는 "6-2" 형식으로 수락하고 올바른 답변을 반환하는 콘솔 애플리케이션을 만드는 것입니다. MEF를 사용하면 애플리케이션 코드를 변경하지 않고도 새 연산자를 추가할 수 있습니다.

이 예제의 전체 코드를 다운로드하려면 SimpleCalculator 샘플(Visual Basic) 참조하세요.

> **참고 항목**
>
> SimpleCalculator의 목적은 MEF의 개념과 구문을 보여 주는 것이 아니라 반드시 사용하기 위한 현실적인 시나리오를 제공하는 것입니다. MEF의 강력한 기능을 통해 가장 많은 이점을 얻을 수 있는 많은 애플리케이션은 SimpleCalculator보다 더 복잡합니다. 더 광범위한 예제는 GitHub **관리 확장성 프레임워크** 참조하세요.

- 시작하려면 Visual Studio에서 새 콘솔 애플리케이션 프로젝트를 하나 만들고 이름을 `SimpleCalculator`로 지정합니다.
- MEF가 있는 어셈블리에 `System.ComponentModel.Composition` 대한 참조를 추가합니다.
- *Module1.vb* 또는 *Program.cs*를 열고, `Imports` 및 `using` 지시문을 추가하기 위해 `System.ComponentModel.Composition` 또는 `System.ComponentModel.Composition.Hosting` 지시문을 추가하세요. 이러한 두 네임스페이스에는 확장 가능한 애플리케이션을 개발하는 데 필요한 MEF 형식이 포함되어 있습니다.
- Visual Basic 사용하는 경우 `Public` 키워드를 `Module1` 모듈을 선언하는 줄에 추가합니다.

컴퍼지션 컨테이너 및 카탈로그

MEF 컴퍼지션 모델의 핵심은 사용 가능한 모든 부분을 포함하고 *컴퍼지*션을 수행하는 컴퍼지션 컨테이너입니다. 수입과 수출의 일치를 의미합니다. 컴퍼지션 컨테이너의 가장 일반적인 형식은 CompositionContainerSimpleCalculator에 사용됩니다.

Visual Basic 사용하는 경우 `Program`이라는 공용 클래스를 추가합니다.

Module1.vb `Program`*Program.cs* 클래스에 다음 줄을 추가합니다.

C#

```
private CompositionContainer _container;
```

사용할 수 있는 파트를 검색하기 위해 컴퍼지션 컨테이너는 *카탈로그*를 사용합니다. 카탈로그는 일부 원본에서 검색된 사용 가능한 파트를 만드는 개체입니다. MEF는 제공된 형식, 어셈블리 또는 디렉터리에서 파트를 검색하는 카탈로그를 제공합니다. 애플리케이션 개발자는 웹 서비스와 같은 다른 원본에서 파트를 검색하는 새 카탈로그를 쉽게 만들 수 있습니다.

클래스에 다음 생성자를 `Program` 추가합니다.

C#

```
private Program()
{
    try
    {
        // An aggregate catalog that combines multiple catalogs.
        var catalog = new AggregateCatalog();
        // Adds all the parts found in the same assembly as the Program class.
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));

        // Create the CompositionContainer with the parts in the catalog.
        _container = new CompositionContainer(catalog);
        _container.ComposeParts(this);
    }
    catch (CompositionException compositionException)
    {
        Console.WriteLine(compositionException.ToString());
    }
}
```

ComposeParts 호출은 컴포지션 컨테이너가 특정 부분 집합을 구성하도록 지시하는 것으로, 이 경우 현재 인스턴스인 `Program`를 구성합니다. 그러나 이 시점에서는 `Program` 에 채울 가져오기가 없으므로 아무 일도 일어나지 않습니다.

특성을 사용하여 가져오기 및 내보내기

`Program` 먼저 계산기를 가져왔습니다. 이렇게 하면 콘솔 입력 및 출력과 같은 사용자 인터페이스 문제를 계산기의 논리에서 분리할 수 있습니다.

`Program` 클래스에 다음 코드를 추가합니다.

C#

```
[Import(typeof(ICalculator))]
```

```
public ICalculator calculator;
```

`calculator` 객체의 선언 자체는 특별하지 않지만 ImportAttribute 속성으로 장식되어 있습니다. 이 특성은 가져오기로 선언합니다. 즉, 개체가 작성될 때 컴퍼지션 엔진에 의해 채워집니다.

모든 가져오기에는 일치하는 내보내기를 결정하는 *계약이* 있습니다. 계약은 명시적으로 지정된 문자열이거나 지정된 형식(이 경우 인터페이스 `ICalculator`)에서 MEF에 의해 자동으로 생성될 수 있습니다. 일치하는 계약으로 선언된 모든 내보내기가 이 가져오기를 수행합니다. 실제로 `calculator` 개체의 형식이 `ICalculator`이긴 하지만, 이는 필수가 아닙니다. 계약은 수입 객체의 종류와 독립적입니다. (이 경우 `typeof(ICalculator)` 태그를 생략할 수 있습니다. MEF는 명시하지 않는 한 계약이 가져오기 유형을 기반으로 한다고 자동으로 추측합니다.)

이 매우 간단한 인터페이스를 모듈 또는 `SimpleCalculator` 네임스페이스에 추가합니다.

C#

```
public interface ICalculator
{
    string Calculate(string input);
}
```

이제 정의 `ICalculator`했으므로 이를 구현하는 클래스가 필요합니다. 모듈 또는 `SimpleCalculator` 네임스페이스에 다음 클래스를 추가합니다.

C#

```
[Export(typeof(ICalculator))]
class MySimpleCalculator : ICalculator
{

}
```

다음은 `Program`에서 가져오기와 일치하는 내보내기입니다. 내보내기가 가져오기와 일치하려면 동일한 계약 조건을 가지고 있어야 합니다. 계약 기반으로 `typeof(MySimpleCalculator)`을(를) 내보낼 경우 불일치가 발생하며, 가져오기가 완료되지 않을 것입니다. 계약이 정확히 일치해야 합니다.

컴포지션 컨테이너는 이 어셈블리에 있는 모든 파트로 채워지므로, `MySimpleCalculator` 파트도 이용 가능합니다. 생성자가 `Program` 개체에 대해 `Program` 컴포지션을 수행할 때, 해당 가져오기는 `MySimpleCalculator` 개체로 채워지게 됩니다. 이 개체는 그 목적을 위해 만들어집니다.

사용자 인터페이스 계층(`Program`)은 다른 것을 알 필요가 없습니다. 따라서 `Main` 메서드에서 나머지 사용자 인터페이스 논리를 채울 수 있습니다.

`Main` 메서드에 다음 코드를 추가합니다.

C#

```
static void Main(string[] args)
{
    // Composition is performed in the constructor.
    var p = new Program();
    Console.WriteLine("Enter Command:");
    while (true)
    {
        string s = Console.ReadLine();
        Console.WriteLine(p.calculator.Calculate(s));
    }
}
```

이 코드는 단순히 입력 줄을 읽고 그 결과에 대해 `Calculate`의 `ICalculator` 함수를 호출하여 콘솔에 출력합니다. 이것이 필요한 모든 코드입니다.`Program` 나머지 작업은 모든 각 부분에서 이루어질 것입니다.

Import 및 ImportMany 특성

SimpleCalculator를 확장하려면 작업 목록을 가져와야 합니다. 일반 ImportAttribute 특성은 하나의 ExportAttribute로만 채워집니다. 둘 이상의 사용 가능한 경우 컴퍼지션 엔진에서 오류가 발생합니다. ImportManyAttribute 속성을 사용하면 어떤 수의 내보내기로도 채울 수 있는 가져오기를 만들 수 있습니다.

클래스에 다음 작업 속성을 `MySimpleCalculator` 추가합니다.

C#

```
[ImportMany]
IEnumerable<Lazy<IOperation, IOperationData>> operations;
```

Lazy<T,TMetadata> 는 내보내기용 간접 참조를 보관하기 위해 MEF에서 제공하는 형식입니다. 여기서는 내보낸 개체 자체 외에도 *내보내기 메타데이터 또는 내*보낸 개체를 설명하는 정보도 가져옵니다. 각각 Lazy<T,TMetadata> 에는 `IOperation` 실제 작업을 나타내는 개체와 해당 메타데이터를 `IOperationData` 나타내는 개체가 포함됩니다.

모듈 또는 `SimpleCalculator` 네임스페이스에 다음 간단한 인터페이스를 추가합니다.

C#

```
public interface IOperation
{
```

```
     int Operate(int left, int right);
}

public interface IOperationData
{
    char Symbol { get; }
}
```

이 경우 각 작업에 대한 메타데이터는 +, -, *등과 같은 해당 작업을 나타내는 기호입니다. 더하기 작업을 사용할 수 있도록 모듈 또는 `SimpleCalculator` 네임스페이스에 다음 클래스를 추가합니다.

C#

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '+')]
class Add: IOperation
{
    public int Operate(int left, int right)
    {
        return left + right;
    }
}
```

ExportAttribute 속성은 이전과 동일하게 기능합니다. ExportMetadataAttribute 특성은 이름-값 쌍 형식의 메타데이터를 해당 내보내기에 연결합니다. `Add`을 구현하는 클래스 `IOperation`는 있지만, `IOperationData`을 구현하는 클래스는 명시적으로 정의되지 않습니다. 대신 제공된 메타데이터의 이름을 기반으로 하는 속성을 사용하여 MEF에서 클래스를 암시적으로 만듭니다. (MEF에서 메타데이터에 액세스하는 여러 가지 방법 중 하나입니다.)

MEF의 컴퍼지션은 *재귀적입니다*. 명시적으로 `Program` 개체를 구성했으며, 이 구성은 `ICalculator`을 가져왔는데, 이 것이 결국 `MySimpleCalculator` 형식의 것으로 판명되었습니다. `MySimpleCalculator`는 `IOperation`의 개체 모음을 가져오며, 이 가져오기는 `MySimpleCalculator`가 생성될 때 `Program`의 가져오기와 동시에 채워집니다. 클래스가 `Add` 추가 가져오기를 선언한 경우 해당 항목도 채워야 합니다. 가져오기가 완료되지 않으면 구성 오류가 발생합니다. 그러나 가져오기를 선택 사항으로 선언하거나 기본값을 할당할 수 있습니다.

계산기 논리

이러한 부분이 준비되면 계산기 논리 자체만 남아 있습니다. 클래스에 `MySimpleCalculator` 다음 코드를 추가하여 메서드를 구현합니다 `Calculate` .

C#

```
public String Calculate(string input)
{
    int left;
    int right;
    char operation;
    // Finds the operator.
    int fn = FindFirstNonDigit(input);
    if (fn < 0) return "Could not parse command.";

    try
    {
        // Separate out the operands.
        left = int.Parse(input.Substring(0, fn));
        right = int.Parse(input.Substring(fn + 1));
    }
    catch
    {
        return "Could not parse command.";
    }

    operation = input[fn];

    foreach (Lazy<IOperation, IOperationData> i in operations)
    {
        if (i.Metadata.Symbol.Equals(operation))
        {
            return i.Value.Operate(left, right).ToString();
        }
    }
    return "Operation Not Found!";
}
```

초기 단계에서는 입력 문자열을 왼쪽 및 오른쪽 피연산자와 연산자 문자로 구문 분석합니다. `foreach` 루프에서 `operations` 컬렉션의 모든 멤버가 검사됩니다. 이러한 개체는 형식 Lazy<T,TMetadata>이며 해당 메타데이터 값은 Metadata 속성을 사용하여, 내보낸 개체는 Value 속성을 사용하여 각각 액세스할 수 있습니다. 이 경우 `Symbol` 개체의 `IOperationData` 속성이 일치하는 것으로 검색되면 계산기는 개체의 메서드를 `Operate` `IOperation` 호출하고 결과를 반환합니다.

계산기를 완료하려면 문자열에서 첫 번째 숫자가 아닌 문자의 위치를 반환하는 도우미 메서드도 필요합니다. 클래스에 다음 도우미 메서드를 `MySimpleCalculator` 추가합니다.

C#

```
private int FindFirstNonDigit(string s)
{
    for (int i = 0; i < s.Length; i++)
    {
        if (!char.IsDigit(s[i])) return i;
    }
```

```
    return -1;
}
```

이제 프로젝트를 컴파일하고 실행할 수 있습니다. Visual Basic에서 `Public` 키워드를 `Module1`에 추가했는지 확인하세요. 콘솔 창에서 "5+3"과 같은 추가 작업을 입력하고 계산기는 결과를 반환합니다. 다른 연산자는 "연산을 찾을 수 없음!" 메시지를 표시합니다.

새 클래스를 사용하여 SimpleCalculator 확장

이제 계산기가 작동하므로 새 작업을 쉽게 추가할 수 있습니다. 모듈 또는 `SimpleCalculator` 네임스페이스에 다음 클래스를 추가합니다.

C#

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '-')]
class Subtract : IOperation
{
    public int Operate(int left, int right)
    {
        return left - right;
    }
}
```

프로젝트를 컴파일하고 실행합니다. "5-3"과 같은 빼기 연산을 입력합니다. 이제 계산기가 빼기 및 추가를 지원합니다.

새 어셈블리를 사용하여 SimpleCalculator 확장

소스 코드에 클래스를 추가하는 것은 간단하지만 MEF는 파트에 대한 애플리케이션의 자체 소스 외부에서 볼 수 있는 기능을 제공합니다. 이를 설명하기 위해서는 DirectoryCatalog 태그를 추가하여 디렉터리와 자신의 어셈블리 안에서 부품을 검색하도록 SimpleCalculator를 수정해야 합니다.

SimpleCalculator 프로젝트에 명명된 `Extensions` 새 디렉터리를 추가합니다. 솔루션 수준이 아닌 프로젝트 수준에서 추가해야 합니다. 그런 다음, 솔루션에 새 클래스 라이브러리 프로젝트인를 추가합니다 `ExtendedOperations`. 새 프로젝트는 별도의 어셈블리로 컴파일됩니다.

ExtendedOperations 프로젝트에 대한 프로젝트 속성 디자이너를 열고 **컴파일** 또는 **빌드** 탭을 클릭합니다. SimpleCalculator 프로젝트 디렉터리(.)의 확장 디렉터리를 가리키도록 **빌드 출력 경로** 또는 **출력 경로를** 변경합니다. *\SimpleCalculator\Extensions*).

Module1.vb 또는 *Program.cs* 생성자에 다음 줄을 `Program` 추가합니다.

C#

```
catalog.Catalogs.Add(
    new DirectoryCatalog(
        "C:\\SimpleCalculator\\SimpleCalculator\\Extensions"));
```

예제 경로를 Extensions 디렉터리의 경로로 바꿉니다. (이 절대 경로는 디버깅 용도로만 사용됩니다. 프로덕션 애플리케이션에서는 상대 경로를 사용합니다.) DirectoryCatalog 이제 확장 디렉터리의 어셈블리에 있는 모든 파트를 컴퍼지션 컨테이너에 추가합니다.

`ExtendedOperations` 프로젝트에서 `SimpleCalculator` 및 `System.ComponentModel.Composition`에 대한 참조를 추가하세요. `ExtendedOperations` 클래스 파일에서 `Imports`에 대한 `using` 또는 `System.ComponentModel.Composition` 지시문을 추가합니다. Visual Basic에서 `Imports`에 대한 `SimpleCalculator` 문을 추가합니다. 그런 다음 클래스 파일에 다음 클래스를 `ExtendedOperations` 추가합니다.

C#

```
[Export(typeof(SimpleCalculator.IOperation))]
[ExportMetadata("Symbol", '%')]
public class Mod : SimpleCalculator.IOperation
{
    public int Operate(int left, int right)
    {
        return left % right;
    }
}
```

계약이 성립되려면 ExportAttribute 속성의 유형과 ImportAttribute 속성의 유형이 같아야 합니다.

프로젝트를 컴파일하고 실행합니다. 새 Mod(%) 연산자를 테스트합니다.

결론

이 항목에서는 MEF의 기본 개념을 설명했습니다.

- 파트, 카탈로그 및 구성 컨테이너

  부품 및 컴퍼지션 컨테이너는 MEF 애플리케이션의 기본 구성 요소입니다. 파트는 값을 가져오거나 내보내는 모든 개체로, 그 자체도 포함됩니다. 카탈로그는 특정 원본의 파트 컬렉션을 제공합니다. 컴퍼지션 컨테이너는 카탈로그에서 제공하는 파트를 사용하여 컴퍼지션을 수행하고 가져오기를 내보내는 바인딩을 수행합니다.

- 가져오기 및 내보내기

가져오기 및 내보내기가 구성 요소에서 통신하는 방법입니다. 가져오기를 사용하면 구성 요소는 특정 값 또는 개체에 대한 필요성을 지정하고 내보내기를 통해 값의 가용성을 지정합니다. 각 가져오기는 계약에 따라 내보내기 항목 목록과 일치합니다.

다음 단계:

이 예제의 전체 코드를 다운로드하려면 SimpleCalculator 샘플(Visual Basic) 참조하세요.

자세한 내용 및 코드 예제는 Managed Extensibility Framework를 참조하세요. MEF 형식 목록은 네임스페이스를 System.ComponentModel.Composition 참조하세요.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 11.

MEF(특성 프로그래밍 모델 개요)

MEF(Managed Extensibility Framework)에서 *프로그래밍 모델은* MEF가 작동하는 개념 개체 집합을 정의하는 특정 메서드입니다. 이러한 개념 개체에는 파트, 가져오기 및 내보내기가 포함됩니다. MEF는 이러한 개체를 사용하지만 표현 방법을 지정하지는 않습니다. 따라서 사용자 지정된 프로그래밍 모델을 포함하여 다양한 프로그래밍 모델이 가능합니다.

MEF에서 사용되는 기본 프로그래밍 모델은 *특성이 지정된 프로그래밍 모델*입니다. 특성이 지정된 프로그래밍 모델 파트에서 가져오기, 내보내기 및 기타 개체는 일반 .NET 클래스를 데코레이트하는 특성으로 정의됩니다. 이 문서에서는 특성이 지정된 프로그래밍 모델에서 제공하는 특성을 사용하여 MEF 애플리케이션을 만드는 방법을 설명합니다.

기본 사항 가져오기 및 내보내기

*내보내기*는 컨테이너의 다른 부분에 제공하는 값이고, *가져오기*는 사용 가능한 내보내기에서 채워질 컨테이너에 대한 요구 사항을 표현하는 것입니다. 특성이 지정된 프로그래밍 모델에서 가져오기 및 내보내기가 클래스 또는 멤버를 `Import` 및 `Export` 어트리뷰트로 데코레이터를 사용하여 선언됩니다. `Export` 특성은 클래스, 필드, 속성 또는 메서드를 데코레이트할 수 있고, `Import` 특성은 필드, 속성 또는 생성자 매개변수를 데코레이트할 수 있습니다.

가져오기를 내보내기와 일치하려면 가져오기 및 내보내기가 동일한 *계약을* 가져야 합니다. 계약은 *계약 이름*이라고 하는 문자열과 계약 형식이라고 하는 내보내거나 가져온 개체의 형식으로 구성 *됩니다*. 계약 이름과 계약 유형이 모두 일치해야 내보내기가 특정 가져오기를 충족하는 것으로 간주됩니다.

계약 매개 변수 중 하나 또는 둘 다 암시적 또는 명시적일 수 있습니다. 다음 코드는 기본 가져오기를 선언하는 클래스를 보여 줍니다.

C#

```
public class MyClass
{
    [Import]
    public IMyAddin MyAddin { get; set; }
}
```

가져오기 `Import`에서 특성은 계약 형식과 계약 이름 매개 변수가 모두 부착되어 있지 않습니다. 따라서 데코레이션된 속성에서 두 요소 모두 추론됩니다. 이 경우 계약 유형이 되고 `IMyAddin` 계약 이름은 계약 형식에서 만든 고유한 문자열이 됩니다. 다시 말하면, 계약 이름은 형식 `IMyAddin`에서 유추된 이름의 내보내기만 일치합니다.

다음은 이전 가져오기와 일치하는 내보내기를 보여줍니다.

C#

```
[Export(typeof(IMyAddin))]
public class MyLogger : IMyAddin { }
```

이 내보내기에서 계약 형식은 특성의 `IMyAddin` 매개 변수로 지정되기 때문입니다`Export`. 내보낸 형식은 계약 형식과 동일하거나, 계약 형식에서 파생되거나, 인터페이스인 경우 계약 형식을 구현해야 합니다. 이 내보내기에서 실제 형식 `MyLogger` 은 인터페이스 `IMyAddin`를 구현합니다. 계약 이름은 계약 유형에서 유추됩니다. 즉, 이 내보내기가 이전 가져오기와 일치합니다.

> ⓘ **참고 항목**
>
> 내보내기 및 가져오기는 일반적으로 공용 클래스 또는 멤버에 선언되어야 합니다. 다른 선언은 지원되지만 프라이빗, 보호된 또는 내부 멤버를 내보내거나 가져오면 파트에 대한 격리 모델이 중단되므로 권장되지 않습니다.

내보내기 및 가져오기가 일치하는 것으로 간주되려면 계약 유형이 정확히 일치해야 합니다. 다음 수출을 검토합니다.

C#

```
[Export] //WILL NOT match the previous import!
public class MyLogger : IMyAddin { }
```

이 내보내기에서는 계약 유형이 `MyLogger`가 아니라 `IMyAddin`입니다. `MyLogger`가 `IMyAddin`를 구현했기 때문에 `IMyAddin` 개체로 캐스팅될 수 있지만, 계약 유형이 동일하지 않아 이 내보내기는 이전 가져오기와 일치하지 않을 것입니다.

일반적으로 계약 이름을 지정할 필요는 없으며 대부분의 계약은 계약 유형 및 메타데이터 측면에서 정의해야 합니다. 그러나 특정 상황에서 계약 이름을 직접 지정하는 것이 중요합니다. 가장 일반적인 경우는 클래스가 기본 형식과 같은 공통 형식을 공유하는 여러 값을 내보내는 경우입니다. 계약 이름은 `Import` 또는 `Export` 특성의 첫 번째 매개 변수로 지정할 수 있습니다. 다음 코드는 지정된 계약 이름을 `MajorRevision`가진 가져오기 및 내보내기를 보여 냅니다.

C#

```
public class MyClass
{
    [Import("MajorRevision")]
    public int MajorRevision { get; set; }
}

public class MyExportClass
{
```

```
    [Export("MajorRevision")] //This one will match.
    public int MajorRevision = 4;

    [Export("MinorRevision")]
    public int MinorRevision = 16;
}
```

계약 형식을 지정하지 않으면 가져오기 또는 내보내기 형식에서 계속 유추됩니다. 그러나 계약 이름을 명시적으로 지정하더라도 가져오기 및 내보내기가 일치하는 것으로 간주되려면 계약 형식도 정확히 일치해야 합니다. 예를 들어 필드가 `MajorRevision` 문자열인 경우 유추된 계약 형식은 일치하지 않으며 내보내기가 동일한 계약 이름을 가지고 있음에도 불구하고 가져오기와 일치하지 않습니다.

메서드 가져오기 및 내보내기

이 특성은 `Export` 클래스, 속성 또는 함수와 같은 방식으로 메서드를 데코레이트할 수도 있습니다. 메서드 내보내기에서는 형식을 유추할 수 없으므로 계약 형식 또는 계약 이름을 지정해야 합니다. 지정된 형식은 사용자 지정 대리자 또는 제네릭 형식(예: `Func`.)일 수 있습니다. 다음 클래스는 이름이 지정된 `DoSomething`메서드를 내보냅니다.

C#

```
public class MyAddin
{
    //Explicitly specifying a generic type.
    [Export(typeof(Func<int, string>))]
    public string DoSomething(int TheParam);
}
```

이 클래스에서 `DoSomething` 메서드는 단일 `int` 매개 변수를 받아 `string`를 반환합니다. 이 내보내기와 일치하려면 가져오기 파트가 적절한 멤버를 선언해야 합니다. 다음 클래스는 메서드를 `DoSomething` 가져옵니다.

C#

```
public class MyClass
{
    [Import] //Contract name must match!
    public Func<int, string> DoSomething { get; set; }
}
```

개체를 사용하는 `Func<T, T>` 방법에 대한 자세한 내용은 다음을 참조하세요 Func<T,TResult>.

가져오기 유형

MEF는 동적, 지연, 필수 구성 요소 및 선택 사항을 비롯한 여러 가져오기 형식을 지원합니다.

동적 가져오기

일부 경우에는 가져오기 클래스가 특정 계약 이름을 가진 모든 유형의 수출을 일치시키고자 할 수 있습니다. 이 시나리오에서 클래스는 *동적 가져오기*를 선언할 수 있습니다. 다음 가져오기는 계약 이름이 "TheString"인 내보내기와 일치하는 것입니다.

C#

```
public class MyClass
{
    [Import("TheString")]
    public dynamic MyAddin { get; set; }
}
```

계약 유형이 `dynamic` 키워드를 통해 유추되면, 모든 계약 유형과 일치합니다. 이 경우 가져오기는 **항상** 일치시킬 계약 이름을 지정해야 합니다. (계약 이름이 지정되지 않은 경우 가져오기는 내보내기 없음과 일치하는 것으로 간주됩니다.) 다음 두 내보내기 모두 이전 가져오기와 일치합니다.

C#

```
[Export("TheString", typeof(IMyAddin))]
public class MyLogger : IMyAddin { }

[Export("TheString")]
public class MyToolbar { }
```

물론 임의 형식의 개체를 처리하도록 가져오기 클래스를 준비해야 합니다.

느린 불러오기

일부 경우 가져오기 클래스는 개체가 즉시 인스턴스화되지 않도록 가져온 개체에 대한 간접 참조가 필요할 수 있습니다. 이 시나리오에서 클래스는 계약 형식을 사용하여 `Lazy<T>`를 선언할 수 있습니다. 다음 가져오기 속성은 지연 가져오기를 선언합니다.

C#

```
public class MyClass
{
    [Import]
    public Lazy<IMyAddin> MyAddin { get; set; }
}
```

컴퍼지션 엔진의 관점에서 계약 형식은 계약 형식 `Lazy<T>` `T`과 동일한 것으로 간주됩니다. 따라서 이전 가져오기는 다음 내보내기와 일치합니다.

C#

```
[Export(typeof(IMyAddin))]
public class MyLogger : IMyAddin { }
```

계약 이름과 계약 유형은 앞서 "기본 가져오기 및 내보내기" 섹션에서 설명한 대로 지연 가져오기에 대한 `Import` 속성에 지정할 수 있습니다.

필수 구성 요소 가져오기

내보낸 MEF 파트는 일반적으로 컴퍼지션 엔진에서 직접 요청 또는 일치하는 가져오기를 채울 필요성에 따라 생성됩니다. 기본적으로 파트를 만들 때 컴퍼지션 엔진은 매개 변수가 없는 생성자를 사용합니다. 엔진이 다른 생성자를 사용하도록 하려면 특성으로 `ImportingConstructor` 표시할 수 있습니다.

각 파트에는 컴퍼지션 엔진에서 사용할 생성자가 하나만 있을 수 있습니다. 매개 변수가 없는 기본 생성자와 `ImportingConstructor` 특성을 제공하지 않거나 `ImportingConstructor` 특성을 하나 이상 제공하면 오류가 발생합니다.

특성으로 표시된 생성자의 매개 변수를 `ImportingConstructor` 채우기 위해 해당 매개 변수는 모두 자동으로 가져오기로 선언됩니다. 이는 파트 초기화 중에 사용되는 가져오기를 선언하는 편리한 방법입니다. 다음 클래스는 가져오기를 선언하는 데 사용합니다 `ImportingConstructor` .

C#

```
public class MyClass
{
    private IMyAddin _theAddin;

    //Parameterless constructor will NOT be
    //used because the ImportingConstructor
    //attribute is present.
    public MyClass() { }

    //This constructor will be used.
    //An import with contract type IMyAddin is
    //declared automatically.
    [ImportingConstructor]
    public MyClass(IMyAddin MyAddin)
    {
        _theAddin = MyAddin;
    }
}
```

기본적으로 이 특성은 `ImportingConstructor` 모든 매개 변수 가져오기에 유추된 계약 형식 및 계약 이름을 사용합니다. 매개 변수를 특성으로 `Import` 데코레이팅하여 이를 재정의할 수 있으며, 그러면 계약 형식 및 계약 이름을 명시적으로 정의할 수 있습니다. 다음 코드에서는 이 구문을 사용하여 부모 클래스 대신 파생 클래스를 가져오는 생성자를 보여 줍니다.

C#

```
[ImportingConstructor]
public MyClass([Import(typeof(IMySubAddin))]IMyAddin MyAddin)
{
    _theAddin = MyAddin;
}
```

특히 컬렉션 매개 변수에 주의해야 합니다. 예를 들어, `ImportingConstructor`를 형식 `IEnumerable<int>`의 매개 변수와 함께 생성자에 지정하면, 가져오기는 형식 `IEnumerable<int>`의 내보내기 집합이 아닌 형식 `int`의 단일 내보내기와 일치합니다. 형식 `int`의 내보내기 집합과 일치하려면 매개 변수에 `ImportMany` 속성을 추가해야 합니다.

특성에서 `ImportingConstructor` 가져오기로 선언된 매개 변수도 *필수 구성 요소 가져오기*로 표시됩니다. MEF는 일반적으로 내보내기 및 가져오기가 *주기*를 형성하도록 허용합니다. 예를 들어 주기는 개체 A가 개체 B를 가져오는 위치이며, 이 개체는 개체 A를 가져옵니다. 일반적인 상황에서는 주기가 문제가 되지 않으며 컴퍼지션 컨테이너는 두 개체를 정상적으로 생성합니다.

가져온 값이 파트의 생성자에 필요한 경우 해당 개체는 주기에 참여할 수 없습니다. 개체 A는 개체 B가 먼저 생성되어야만 생성될 수 있는데, 개체 B가 개체 A를 참조하면 주기가 해결되지 않으며 컴퍼지션 오류가 발생합니다. 따라서 생성자 매개 변수에 선언된 가져오기는 필수 구성 요소 가져오기이므로 필요한 개체에서 내보내기 전에 모두 채워야 합니다.

선택적 가져오기

이 특성은 `Import` 파트가 작동하기 위한 요구 사항을 지정합니다. 가져오기를 수행할 수 없는 경우 해당 파트의 컴퍼지션이 실패하고 파트를 사용할 수 없습니다.

속성을 사용하여 가져오기가 `AllowDefault`임을 지정할 수 있습니다. 이 경우 가져오기가 사용 가능한 내보내기와 일치하지 않더라도 구성이 성공하고 가져오기 속성이 해당 속성 형식의 기본값으로 설정됩니다. 개체 속성의 경우 `null`, 부울의 경우 `false`, 숫자 속성의 경우 0으로 설정됩니다. 다음 클래스는 선택적 가져오기를 사용합니다.

C#

```
public class MyClass
{
    [Import(AllowDefault = true)]
    public Plugin thePlugin { get; set; }
```

```
    //If no matching export is available,
    //thePlugin will be set to null.
}
```

여러 개체 가져오기

성공적으로 구성되려면 `Import` 특성이 오직 하나의 내보내기와 일치해야 합니다. 다른 경우에는 컴퍼지션 오류가 발생합니다. 동일한 계약과 일치하는 여러 내보내기를 가져오려면 `ImportMany` 특성을 사용하세요. 이 특성으로 표시된 가져오기는 항상 선택 사항입니다. 예를 들어 일치하는 내보내기가 없으면 컴포지션이 실패하지 않는 상태입니다. 다음 클래스는 형식 `IMyAddin`의 내보내기를 여러 개 가져옵니다.

C#

```
public class MyClass
{
    [ImportMany]
    public IEnumerable<IMyAddin> MyAddin { get; set; }
}
```

가져온 배열은 일반 `IEnumerable<T>` 구문 및 메서드를 사용하여 액세스할 수 있습니다. 대신 일반 배열(`IMyAddin[]`)을 사용할 수도 있습니다.

이 패턴은 구문과 `Lazy<T>` 함께 사용할 때 매우 중요할 수 있습니다. 예를 들어 , `ImportMany` 및 `IEnumerable<T>`를 사용하여 `Lazy<T>` 임의의 수의 개체에 대한 간접 참조를 가져오고 필요한 개체만 인스턴스화할 수 있습니다. 다음 클래스는 이 패턴을 보여 줍니다.

C#

```
public class MyClass
{
    [ImportMany]
    public IEnumerable<Lazy<IMyAddin>> MyAddin { get; set; }
}
```

발견 회피

경우에 따라 카탈로그의 일부로 부품이 검색되지 않도록 할 수 있습니다. 예를 들어 파트는 상속할 기본 클래스일 수 있지만 사용되지는 않습니다. 이 작업을 수행하는 방법에는 두 가지가 있습니다. 먼저 파트 클래스에서 `abstract` 키워드를 사용할 수 있습니다. 추상 클래스는 직접적인 내보내기를 제공하지 않지만, 이를 파생한 클래스에는 상속된 내보내기를 제공할 수 있습니다.

클래스를 추상화할 수 없는 경우 특성으로 `PartNotDiscoverable` 데코레이트할 수 있습니다. 이 특성으로 데코레이팅된 부분은 카탈로그에 포함되지 않습니다. 다음 예제에서는 이러한 패턴을 보여 줍니다. `DataOne` 는 카탈로그에서 검색됩니다. `DataTwo` 추상적이므로 검색되지 않습니다. `DataThree`가 `PartNotDiscoverable` 특성을 사용했기 때문에 검색되지 않습니다.

C#

```
[Export]
public class DataOne
{
    //This part will be discovered
    //as normal by the catalog.
}

[Export]
public abstract class DataTwo
{
    //This part will not be discovered
    //by the catalog.
}

[PartNotDiscoverable]
[Export]
public class DataThree
{
    //This part will also not be discovered
    //by the catalog.
}
```

메타데이터 및 메타데이터 뷰

내보내기에서는 *메타데이터*라고 하는 자신에 대한 추가 정보를 제공할 수 있습니다. 메타데이터를 사용하여 내보낸 개체의 속성을 가져오기 파트로 전달할 수 있습니다. 가져오기 파트는 이 데이터를 사용하여 사용할 내보내기를 결정하거나 내보내기를 생성하지 않고도 내보내기 정보를 수집할 수 있습니다. 이러한 이유로 메타데이터를 사용하려면 가져오기가 지연되어야 합니다.

메타데이터를 사용하려면 일반적으로 사용할 수 있는 메타데이터를 선언하는 *메타데이터 뷰*라는 인터페이스를 선언합니다. 메타데이터 뷰 인터페이스에는 속성만 있어야 하며 이러한 속성에는 `get` 접근자가 있어야 합니다. 다음 인터페이스는 예제 메타데이터 뷰입니다.

C#

```
public interface IPluginMetadata
{
    string Name { get; }

    [DefaultValue(1)]
```

```
    int Version { get; }
}
```

제네릭 컬렉션을 `IDictionary<string, object>` 메타데이터 보기로 사용할 수도 있지만 형식 검사의 이점을 상실하므로 피해야 합니다.

일반적으로 메타데이터 뷰에 명명된 모든 속성이 필요하며 이를 제공하지 않는 모든 내보내기가 일치하는 것으로 간주되지 않습니다. `DefaultValue` 속성은 특성이 선택 사항임을 지정합니다. 속성이 포함되지 않으면 매개 변수 `DefaultValue`로 지정된 기본값이 할당됩니다. 다음은 메타데이터로 데코레이팅된 두 가지 클래스입니다. 이러한 두 클래스는 이전 메타데이터 뷰와 일치합니다.

C#

```
[Export(typeof(IPlugin)),
    ExportMetadata("Name", "Logger"),
    ExportMetadata("Version", 4)]
public class Logger : IPlugin
{
}

[Export(typeof(IPlugin)),
    ExportMetadata("Name", "Disk Writer")]
    //Version is not required because of the DefaultValue
public class DWriter : IPlugin
{
}
```

메타데이터는 `Export` 특성 후에 `ExportMetadata` 특성을 사용하여 표현됩니다. 메타데이터의 각 부분은 이름/값 쌍으로 구성됩니다. 메타데이터의 이름 부분은 메타데이터 보기에서 해당 속성의 이름과 일치해야 하며 값은 해당 속성에 할당됩니다.

사용 중인 메타데이터 보기(있는 경우)를 지정하는 가져오기입니다. 메타데이터가 있는 가져오기는 지연 가져오기로 선언되고 메타데이터 인터페이스는 두 번째 형식 매개 변수 `Lazy<T,T>`로 사용됩니다. 다음 클래스는 메타데이터를 사용하여 이전 부분을 가져옵니다.

C#

```
public class Addin
{
    [Import]
    public Lazy<IPlugin, IPluginMetadata> plugin;
}
```

대부분의 경우 사용 가능한 가져오기를 구문 분석하고 하나만 선택 및 인스턴스화하거나 특정 조건에 맞게 컬렉션을 필터링하기 위해 메타데이터 `ImportMany` 를 특성과 결합하려고 합니다.

다음 클래스는 값이 "Logger"인 `IPlugin` 개체만 `Name` 인스턴스화합니다.

C#

```
public class User
{
    [ImportMany]
    public IEnumerable<Lazy<IPlugin, IPluginMetadata>> plugins;

    public IPlugin InstantiateLogger()
    {
        IPlugin logger = null;

        foreach (Lazy<IPlugin, IPluginMetadata> plugin in plugins)
        {
            if (plugin.Metadata.Name == "Logger")
                logger = plugin.Value;
        }
        return logger;
    }
}
```

상속 가져오기 및 내보내기

클래스가 파트에서 상속되는 경우 해당 클래스도 파트가 될 수 있습니다. 임포트는 항상 서브클래스에 상속됩니다. 따라서 파트의 하위 클래스는 항상 부모 클래스와 동일한 가져오기를 사용하여 파트가 됩니다.

특성을 사용해 `Export`로 선언된 내보내기는 하위 클래스에 상속되지 않습니다. 그러나 파트는 `InheritedExport` 속성을 사용하여 자신을 내보낼 수 있습니다. 파트의 하위 클래스는 계약 이름 및 계약 형식을 포함하여 동일한 내보내기를 상속하고 제공합니다. 특성 `Export` 과 `InheritedExport` 달리 멤버 수준이 아닌 클래스 수준에서만 적용할 수 있습니다. 따라서 멤버 단위의 내보내기는 절대 상속될 수 없습니다.

다음 네 가지 클래스는 가져오기 및 내보내기 상속의 원칙을 보여 줍니다. `NumTwo`는 `NumOne`로부터 상속받습니다, 그래서 `NumTwo`가 `IMyData`를 가져옵니다. 일반 내보내기가 상속되지 않으므로 `NumTwo` 아무것도 내보내지 않습니다. `NumFour`는 `NumThree`로부터 상속받습니다. `NumThree`가 `InheritedExport`을 사용했기 때문에, `NumFour`에는 계약 유형이 `NumThree`인 내보내기가 하나 있습니다. 멤버 수준 내보내기는 상속되지 않으므로 `IMyData` 내보내지 않습니다.

C#

```
[Export]
public class NumOne
{
    [Import]
```

```
    public IMyData MyData { get; set; }
}

public class NumTwo : NumOne
{
    //Imports are always inherited, so NumTwo will
    //import IMyData.

    //Ordinary exports are not inherited, so
    //NumTwo will NOT export anything. As a result it
    //will not be discovered by the catalog!
}

[InheritedExport]
public class NumThree
{
    [Export]
    Public IMyData MyData { get; set; }

    //This part provides two exports, one of
    //contract type NumThree, and one of
    //contract type IMyData.
}

public class NumFour : NumThree
{
    //Because NumThree used InheritedExport,
    //this part has one export with contract
    //type NumThree.

    //Member-level exports are never inherited,
    //so IMyData is not exported.
}
```

특성과 `InheritedExport` 연결된 메타데이터가 있는 경우 해당 메타데이터도 상속됩니다. (자세한 내용은 이전의 "메타데이터 및 메타데이터 뷰" 섹션을 참조하세요.) 상속된 메타데이터는 하위 클래스에서 수정할 수 없습니다. 그러나 `InheritedExport` 특성을 동일한 계약 이름 및 계약 형식으로 다시 선언하여 새 메타데이터를 사용하면, 하위 클래스는 상속된 메타데이터를 새 메타데이터로 바꿀 수 있습니다. 다음 클래스는 이 원칙을 보여 줍니다. `MegaLogger` 부품은 `Logger` 에서 상속받고 `InheritedExport` 특성을 포함합니다. `MegaLogger` Status라는 새 메타데이터를 다시 선언하므로 이름 및 버전 메타데이터`Logger`는 상속되지 않습니다.

C#

```
[InheritedExport(typeof(IPlugin)),
    ExportMetadata("Name", "Logger"),
    ExportMetadata("Version", 4)]
public class Logger : IPlugin
{
    //Exports with contract type IPlugin and
    //metadata "Name" and "Version".
```

```
}

public class SuperLogger : Logger
{
    //Exports with contract type IPlugin and
    //metadata "Name" and "Version", exactly the same
    //as the Logger class.
}

[InheritedExport(typeof(IPlugin)),
    ExportMetadata("Status", "Green")]
public class MegaLogger : Logger        {
    //Exports with contract type IPlugin and
    //metadata "Status" only. Re-declaring
    //the attribute replaces all metadata.
}
```

메타데이터를 재정의하기 위해 `InheritedExport` 특성을 다시 선언할 때 계약 종류가 동일한지 확인합니다. 이전 예제에서 `IPlugin`는 계약 유형입니다. 이들이 다를 경우, 재정의하는 대신 두 번째 속성은 파트에서 독립적인 두 번째 내보내기를 생성합니다. 일반적으로, 이는 `InheritedExport` 특성을 재정의할 때, 이전 예제에서 보여준 것처럼 계약 유형을 명시적으로 지정해야 한다는 것을 의미합니다.

인터페이스는 직접 인스턴스화할 수 없으므로 일반적으로 특성을 사용하여 `Export` `Import` 데코레이팅할 수 없습니다. 그러나 인터페이스는 인터페이스 수준에서 `InheritedExport` 특성으로 데코레이팅할 수 있으며, 내보내기 작업과 관련된 메타데이터는 구현 클래스에 의해 상속됩니다. 그러나 인터페이스 자체는 일부로 사용할 수 없습니다.

사용자 지정 내보내기 특성

기본 내보내기 특성 `Export` 및 `InheritedExport`메타데이터를 특성 속성으로 포함하도록 확장할 수 있습니다. 이 기술은 유사한 메타데이터를 여러 부분에 적용하거나 메타데이터 특성의 상속 트리를 만드는 데 유용합니다.

사용자 지정 특성은 계약 유형, 계약 이름 또는 기타 메타데이터를 지정할 수 있습니다. 사용자 지정 특성을 정의하려면 `ExportAttribute` 또는 `InheritedExportAttribute`를 상속받는 클래스에 `MetadataAttribute` 특성을 데코레이트해야 합니다. 다음 클래스는 사용자 지정 특성을 정의합니다.

C#

```
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public class MyAttribute : ExportAttribute
{
    public MyAttribute(string myMetadata)
```

```
        : base(typeof(IMyAddin))
    {
        MyMetadata = myMetadata;
    }

    public string MyMetadata { get; private set; }
}
```

이 클래스는 계약 형식 `MyAttribute` 과 명명`IMyAddin`된 일부 메타데이터를 사용하여 명명된 `MyMetadata` 사용자 지정 특성을 정의합니다. 특성으로 표시된 클래스의 `MetadataAttribute` 모든 속성은 사용자 지정 특성에 정의된 메타데이터로 간주됩니다. 다음 두 선언은 동일합니다.

C#

```
[Export(typeof(IMyAddin)),
    ExportMetadata("MyMetadata", "theData")]
public MyAddin myAddin { get; set; }
```

C#

```
[MyAttribute("theData")]
public MyAddin myAddin { get; set; }
```

첫 번째 선언에서 계약 유형 및 메타데이터는 명시적으로 정의됩니다. 두 번째 선언에서 계약 유형 및 메타데이터는 사용자 지정된 특성에서 암시적입니다. 특히 많은 부분(예: 작성자 또는 저작권 정보)에 많은 양의 동일한 메타데이터를 적용해야 하는 경우 사용자 지정 특성을 사용하면 많은 시간과 중복을 절약할 수 있습니다. 또한 변형을 허용하도록 사용자 지정 특성의 상속 트리를 만들 수 있습니다.

사용자 지정 특성에서 선택적 메타데이터를 만들려면 이 특성을 사용할 `DefaultValue` 수 있습니다. 이 특성이 사용자 지정 특성 클래스의 속성에 적용되는 경우 데코레이팅된 속성은 선택 사항이며 내보내기자가 제공할 필요가 없음을 지정합니다. 속성 값이 제공되지 않으면 속성 형식의 기본값(일반적으로 `null`, `false`또는 0)이 할당됩니다.

생성 정책

파트가 임포트와 컴포지션을 지정하면, 컴포지션 컨테이너는 일치하는 엑스포트를 찾으려고 시도합니다. 가져오기와 내보내기가 일치하는 경우 가져오기 멤버는 내보낸 개체의 인스턴스로 설정됩니다. 이 인스턴스의 위치는 내보내기 파트의 *생성 정책에* 의해 제어됩니다.

두 가지 가능한 만들기 정책은 *공유* 되고 *공유되지 않습니다*. 공유 생성 정책이 있는 파트는 해당 계약이 있는 파트에 대해 컨테이너 내 모든 가져오기 간에 공유됩니다. 컴퍼지션 엔진이 일치 항목을 찾고 가져오기 속성을 설정해야 하는 경우 아직 없는 경우에만 파트의 새 복사본을 인스턴스화합니다. 그렇지 않으면 기존 복사본을 제공합니다. 즉, 많은 개체에 동일한 부분에 대한 참

조가 있을 수 있습니다. 이러한 부분은 여러 위치에서 변경될 수 있는 내부 상태에 의존해서는 안 됩니다. 이 정책은 정적 파트, 서비스를 제공하는 파트 및 많은 메모리 또는 기타 리소스를 사용하는 파트에 적합합니다.

비공유의 만들기 정책이 있는 파트는 해당 내보내기 중 하나에 대해 일치하는 가져오기를 찾을 때마다 만들어집니다. 따라서 해당 파트의 내보낸 계약과 일치하는 경우, 컨테이너의 모든 가져오기에 대해 새 복사본이 생성됩니다. 이러한 복사본의 내부 상태는 공유되지 않습니다. 이 정책은 각 가져오기에 자체 내부 상태가 필요한 부분에 적합합니다.

가져오기 및 내보내기 둘 다 값 `Shared` `NonShared`중에서 파트의 만들기 정책을 지정할 수 있습니다`Any`. 기본값은 `Any` 가져오기 및 내보내기 모두에 대한 것입니다. `Shared` 또는 `NonShared`을 지정하는 내보내기는 동일한 것을 지정하거나 `Any`을 지정하는 가져오기와만 일치합니다. 마찬가지로, `Shared` 또는 `NonShared`을 지정하는 가져오기는 동일한 것을 지정하거나 `Any`을 지정하는 내보내기와만 일치합니다. 호환되지 않는 만들기 정책을 사용하는 가져오기 및 내보내기에서는 계약 이름 또는 계약 형식이 일치하지 않는 가져오기 및 내보내기와 같은 방식으로 일치 항목으로 간주되지 않습니다. 가져오기와 내보내기가 모두 `Any`을(를) 지정하거나 생성 정책을 지정하지 않고 기본값이 `Any`일 경우, 생성 정책은 기본값으로 공유로 설정됩니다.

다음 예제에서는 만들기 정책을 지정하는 가져오기 및 내보내기를 모두 보여 줍니다. `PartOne` 에서는 만들기 정책을 지정하지 않으므로 기본값은 .입니다 `Any`. `PartTwo` 에서는 만들기 정책을 지정하지 않으므로 기본값은 .입니다 `Any`. 가져오기 및 내보내기 모두 기본값이므로 `Any` `PartOne` 공유됩니다. `PartThree` 는 `Shared` 만들기 정책을 지정하므로 `PartTwo` `PartThree` 동일한 복사본 `PartOne`을 공유합니다. `PartFour` 는 `NonShared` 만들기 정책을 지정하므로 `PartFour` 는 `PartFive`에서 공유되지 않습니다. `PartSix` 는 만들기 정책을 지정합니다 `NonShared` . `PartFive`와 `PartSix`는 각각 `PartFour`의 별도 복사본을 받게 됩니다. `PartSeven` 는 만들기 정책을 지정합니다 `Shared` . 생성 정책이 `PartFour`인 내보낸 `Shared`이 없으므로 `PartSeven` 가져오기는 어떤 것도 일치하지 않아 채워지지 않습니다.

C#

```
[Export]
public class PartOne
{
    //The default creation policy for an export is Any.
}

public class PartTwo
{
    [Import]
    public PartOne partOne { get; set; }

    //The default creation policy for an import is Any.
    //If both policies are Any, the part will be shared.
}
```

```
public class PartThree
{
    [Import(RequiredCreationPolicy = CreationPolicy.Shared)]
    public PartOne partOne { get; set; }

    //The Shared creation policy is explicitly specified.
    //PartTwo and PartThree will receive references to the
    //SAME copy of PartOne.
}

[Export]
[PartCreationPolicy(CreationPolicy.NonShared)]
public class PartFour
{
    //The NonShared creation policy is explicitly specified.
}

public class PartFive
{
    [Import]
    public PartFour partFour { get; set; }

    //The default creation policy for an import is Any.
    //Since the export's creation policy was explicitly
    //defined, the creation policy for this property will
    //be non-shared.
}

public class PartSix
{
    [Import(RequiredCreationPolicy = CreationPolicy.NonShared)]
    public PartFour partFour { get; set; }

    //Both import and export specify matching creation
    //policies.  PartFive and PartSix will each receive
    //SEPARATE copies of PartFour, each with its own
    //internal state.
}

public class PartSeven
{
    [Import(RequiredCreationPolicy = CreationPolicy.Shared)]
    public PartFour partFour { get; set; }

    //A creation policy mismatch.  Because there is no
    //exported PartFour with a creation policy of Shared,
    //this import does not match anything and will not be
    //filled.
}
```

수명 주기 및 폐기

파트는 컴퍼지션 컨테이너에 호스트되므로 해당 수명 주기는 일반 개체보다 더 복잡할 수 있습니다. 파트는 두 가지 중요한 수명 주기 관련 인터페이스를 구현할 수 있습니다 `IDisposable` 및 `IPartImportsSatisfiedNotification`.

종료 시 작업을 수행해야 하거나 리소스를 해제해야 하는 파트는 .NET 개체에 대해 평소와 같이 `IDisposable` 구현해야 합니다. 그러나 컨테이너는 파트에 대한 참조를 만들고 유지 관리하므로 파트를 소유하는 컨테이너만 해당 파트에 대한 메서드를 `Dispose` 호출해야 합니다. 컨테이너 자체는 `IDisposable`을(를) 구현하며, 정리 작업의 일환으로 `Dispose`에서 소유하고 있는 모든 파트에 대해 `Dispose`을(를) 호출합니다. 이러한 이유로, 컴퍼지션 컨테이너와 그에 속한 모든 부품이 더 이상 필요하지 않게 되면 항상 폐기해야 합니다.

수명이 긴 컴퍼지션 컨테이너의 경우 공유하지 않는 만들기 정책을 사용하는 파트의 메모리 사용이 문제가 될 수 있습니다. 이러한 비 공유 파트는 여러 번 만들 수 있으며 컨테이너 자체가 삭제될 때까지 삭제되지 않습니다. 이를 처리하기 위해 컨테이너는 메서드를 `ReleaseExport` 제공합니다. 공유가 아닌 내보내기에서 이 메서드를 호출하면 컴퍼지션 컨테이너에서 해당 내보내기가 제거되고 삭제됩니다. 제거된 내보내기에서만 사용되는 부분 및 트리 아래로 사용되는 부분도 제거되고 삭제됩니다. 이러한 방식으로 컴퍼지션 컨테이너 자체를 삭제하지 않고 리소스를 회수할 수 있습니다.

`IPartImportsSatisfiedNotification` 에는 이름이 1개인 `OnImportsSatisfied`메서드가 포함되어 있습니다. 이 메서드는 컴퍼지션이 완료되고 파트의 가져오기를 사용할 준비가 되었을 때 인터페이스를 구현하는 모든 파트의 컴퍼지션 컨테이너에 의해 호출됩니다. 컴포지션 엔진은 부품을 생성하여 다른 부품의 가져오기를 충족합니다. 파트 가져오기를 설정하기 전에 해당 값을 특성을 사용하여 `ImportingConstructor` 필수 구성 요소로 지정하지 않는 한 파트 생성자에서 가져온 값을 사용하거나 조작하는 초기화를 수행할 수 없습니다. 이는 일반적으로 기본 설정 방법이지만 경우에 따라 생성자 주입을 사용할 수 없는 경우도 있습니다. 이러한 경우 `OnImportsSatisfied`에서 초기화를 수행할 수 있으며, 파트는 `IPartImportsSatisfiedNotification`을 구현해야 합니다.

Last updated on 2026. 03. 11.

.NET 버전 관리 방법

.NET 런타임 및 .NET SDK는 다양한 주파수에서 새로운 기능을 추가합니다. 일반적으로 SDK는 런타임보다 더 자주 업데이트됩니다. 이 문서에서는 런타임 및 SDK 버전 번호를 설명합니다.

.NET은 매년 11월에 새 주 버전을 릴리스합니다. .NET 6 또는 .NET 8과 같은 짝수 번호가 매겨진 릴리스는 LTS(장기 지원) 릴리스입니다. LTS 릴리스는 3년 동안 무료 지원과 패치를 가져옵니다. 홀수 번호가 매겨진 릴리스는 STS(표준 기간 지원) 릴리스입니다. 표준 기간 지원 릴리스는 2년 동안 무료 지원 및 패치를 받습니다(.NET 9부터 시작).

버전 관리 세부 정보

.NET 런타임에는 시맨틱 버전을 따르는 major.minor.patch 접근 방식이 있습니다.

그러나 .NET SDK는 의미 체계 버전 지정을 따르지 않습니다. .NET SDK는 더 빠르게 릴리스되며 해당 버전 번호는 정렬된 런타임과 SDK 자체의 부 릴리스 및 패치 릴리스를 모두 전달해야 합니다.

.NET SDK 버전 번호의 처음 두 위치는 릴리스된 .NET 런타임 버전과 일치합니다. SDK의 각 버전은 이 런타임 또는 더 낮은 버전에 대한 애플리케이션을 만들 수 있습니다.

SDK 버전 번호의 세 번째 위치는 부 번호와 패치 번호를 모두 전달합니다. 소수 버전은 100으로 곱해집니다. 마지막 두 자리 숫자는 패치 번호를 나타냅니다. 부 버전 1, 패치 버전 2는 102로 표시됩니다. 예를 들어 런타임 및 SDK 버전 번호의 가능한 시퀀스는 다음과 같습니다.

테이블 확장

변화	.NET 런타임	.NET SDK(*)	비고
초기 릴리스	5.0.0	5.0.100	초기 릴리스입니다.
SDK 패치	5.0.0	5.0.101	이 SDK 패치에서는 런타임이 변경되지 않았습니다. SDK 패치는 SDK 패치에서 마지막 숫자를 증가시킵니다.
런타임 및 SDK 패치	5.0.1	5.0.102	런타임 패치는 런타임 패치 번호를 증가시킵니다. SDK 패치는 SDK 패치에서 마지막 숫자를 증가시킵니다.
SDK 기능 변경	5.0.1	5.0.200	런타임 패치는 변경되지 않았습니다. 새 SDK 기능으로 SDK 패치의 첫 번째 숫자가 증가합니다.
런타임 패치	5.0.2	5.0.200	런타임 패치는 런타임 패치 번호를 증가시킵니다. SDK는 변경되지 않습니다.

앞의 표에서 다음과 같은 여러 정책을 볼 수 있습니다.

- 런타임 및 SDK는 주 버전과 부 버전을 공유합니다. 지정된 SDK 및 런타임에 대한 처음 두 숫자는 일치해야 합니다. 앞의 모든 예제는 .NET 5.0 릴리스 스트림의 일부입니다.
- 런타임의 패치 버전은 런타임이 업데이트될 때만 변경됩니다. SDK 패치 번호는 런타임 패치에 대해 업데이트되지 않습니다.
- SDK의 패치 버전은 SDK가 업데이트될 때만 업데이트됩니다. 런타임 패치에는 SDK 패치가 필요하지 않을 수 있습니다.

참고:

- 런타임 기능 업데이트 전에 SDK에 10개의 기능 업데이트가 있는 경우 버전 번호는 1000 시리즈로 롤됩니다. 버전 5.0.1000은 버전 5.0.900을 따릅니다. 이 상황은 발생하지 않을 것으로 예상됩니다.
- 기능 릴리스가 없는 99개의 패치 릴리스는 발생하지 않습니다. 릴리스가 이 숫자에 가까워지면 기능 릴리스가 강제로 적용됩니다.

dotnet/designs 리포지토리의 초기 제안에서 자세한 내용을 볼 수 있습니다.

의미 체계 버전 관리

.NET *런타임*은 버전 번호의 다양한 부분을 사용하여 변경의 정도와 유형을 설명하는 버전 관리 사용을 채택하는 `MAJOR.MINOR.PATCH`를 대략적으로 준수합니다.

```
MAJOR.MINOR.PATCH[-PRERELEASE-BUILDNUMBER]
```

선택 사항 `PRERELEASE` 및 `BUILDNUMBER` 파트는 지원되는 릴리스에 포함되지 않으며 야간 빌드, 원본 대상의 로컬 빌드 및 지원되지 않는 미리 보기 릴리스에만 존재합니다.

런타임 버전 번호 변경

- `MAJOR` 는 1년에 한 번 증가하며 다음을 포함할 수 있습니다.
  - 제품의 중요한 변경 또는 새 제품 방향.
  - API에는 호환성을 깨는 변경이 적용되었습니다. 호환성이 손상되는 변경 내용을 수락할 수 있는 높은 기준이 있습니다.
  - 기존 종속성의 최신 `MAJOR` 버전이 채택됩니다.

  주 릴리스는 1년마다 한 번씩 발생하며, 짝수 번호가 매겨진 버전은 LTS(장기 지원) 릴리스입니다. 이 버전 관리 체계를 사용하는 첫 번째 LTS 릴리스는 .NET 6입니다. 최신 비 LTS 버전은 .NET 9입니다.

- `MINOR` 는 다음과 같은 경우 증가합니다.

    - 공용 API 노출 영역이 추가됩니다.
    - 새 동작이 추가됩니다.
    - 기존 종속성의 최신 `MINOR` 버전이 채택됩니다.
    - 새 종속성이 도입되었습니다.

- `PATCH` 는 다음과 같은 경우 증가합니다.
    - 버그 수정이 이루어집니다.
    - 최신 플랫폼에 대한 지원이 추가됩니다.
    - 기존 종속성의 최신 `PATCH` 버전이 채택됩니다.
    - 다른 변경 내용은 이전 사례 중 하나에 맞지 않습니다.

여러 변경 내용이 있는 경우 개별 변경 내용의 영향을 받는 가장 높은 요소가 증가하며 다음 요소는 0으로 다시 설정됩니다. 예를 들어, `MAJOR`가 증가하면 `MINOR.PATCH`가 0으로 재설정됩니다. `MINOR` 증가하면 `PATCH` 0으로 다시 설정되지만 `MAJOR` 동일하게 유지됩니다.

파일 이름의 버전 번호

.NET용으로 다운로드된 파일은 예를 들어 `dotnet-sdk-5.0.301-win-x64.exe`버전을 전달합니다.

미리 보기 버전

미리 보기 버전은 버전 `-preview.[number].[build]` 번호에 추가됩니다. 예: `6.0.0-preview.5.21302.13`.

서비스 버전

릴리스가 종료된 후 릴리스 분기는 일반적으로 일일 빌드 생성을 중지하고 대신 서비스 빌드를 생성하기 시작합니다. 서비스 버전은 버전에 `-servicing-[number]` 추가됩니다. 예: `5.0.1-servicing-006924`.

.NET 런타임 호환성

.NET 런타임은 버전 간에 높은 수준의 호환성을 유지합니다. .NET 앱은 일반적으로 새 주 .NET 런타임 버전으로 업그레이드한 후에도 계속 작동해야 합니다.

각 주요 .NET 런타임 버전에는 의도적이고 신중하게 검사되고 문서화된 호환성이 손상되는 변경 내용이 포함되어 있습니다. 문서화된 호환성을 깨뜨리는 변경 사항은 업그레이드 후 앱에 영향을 미칠 수 있는 문제의 유일한 원인이 아닙니다. 예를 들어 .NET 런타임의 성능 향상(호환성이 손상되는 변경으로 간주되지 않음)은 앱이 해당 버전에서 작동하지 않는 잠재적인 앱 스레딩

버그를 노출할 수 있습니다. 대형 앱은 새 .NET 런타임 주 버전으로 업그레이드한 후 몇 가지 수정이 필요합니다.

기본적으로 .NET 앱은 지정된 .NET 런타임 주 버전에서 실행되도록 구성되므로 새 .NET 런타임 주 버전에서 실행되도록 앱을 업그레이드하는 것이 좋습니다. 그런 다음 업그레이드 후 앱을 다시 테스트하여 문제를 식별합니다.

앱 다시 컴파일을 통해 업그레이드할 수 없다고 가정합니다. 이 경우 .NET 런타임은 컴파일된 버전보다 더 높은 주 .NET 런타임 버전에서 앱을 실행할 수 있도록 추가 설정을 제공합니다. 이러한 설정은 앱을 더 높은 주 .NET 런타임 버전으로 업그레이드하는 데 관련된 위험을 변경하지 않으며, 업그레이드 후에도 앱을 다시 테스트해야 합니다.

.NET 런타임은 이전 .NET 런타임 버전을 대상으로 하는 라이브러리 로드를 지원합니다. 최신 주 .NET 런타임 버전으로 업그레이드된 앱은 이전 .NET 런타임 버전을 대상으로 하는 라이브러리 및 NuGet 패키지를 참조할 수 있습니다. 앱에서 참조하는 모든 라이브러리 및 NuGet 패키지의 대상 런타임 버전을 동시에 업그레이드할 필요가 없습니다.

참고하십시오

- .NET의 주요 변경 내용
- 대상 프레임워크
- .NET 배포 패키징
- .NET 지원 수명 주기 팩트 시트
- .NET용 Docker 이미지

Last updated on 2025. 11. 07.

사용할 .NET 버전 선택

이 문서에서는 .NET 도구, SDK 및 런타임에서 버전을 선택하는 데 사용하는 정책을 설명합니다. 이러한 정책은 지정된 버전을 사용하여 애플리케이션을 실행하고 개발자 컴퓨터와 최종 사용자 컴퓨터를 쉽게 업그레이드할 수 있는 균형을 제공합니다. 이러한 정책은 다음을 사용하도록 설정합니다.

- 보안 및 안정성 업데이트를 포함하여 .NET을 쉽고 효율적으로 배포합니다.
- 대상 런타임에 관계없이 최신 도구 및 명령을 사용합니다.

버전 선택이 수행됩니다.

- SDK 명령을 실행할 때 SDK에서 설치된 최신 버전사용합니다.
- 어셈블리를 빌드할 때 대상 프레임워크 모니커가 빌드 시간 API정의합니다.
- .NET 애플리케이션을 실행할 때, 대상 프레임워크에 종속된 앱이 롤 포워드됩니다 .
- 자체 포함 애플리케이션을 게시하는 경우 자체 포함 배포에는 선택한 런타임포함됩니다.

이 문서의 나머지 단계에서는 이러한 네 가지 시나리오를 살펴봅니다.

SDK는 최신 설치된 버전을 사용합니다.

SDK 명령에는 `dotnet new` 및 `dotnet run`포함됩니다. .NET CLI는 모든 `dotnet` 명령에 대해 SDK 버전을 선택해야 합니다. 다음과 같은 경우에도 기본적으로 컴퓨터에 설치된 최신 SDK를 사용합니다.

- 이 프로젝트는 이전 버전의 .NET 런타임을 대상으로 합니다.
- 최신 버전의 .NET SDK는 미리 보기 버전입니다.

이전 .NET 런타임 버전을 대상으로 하는 동안 최신 SDK 기능 및 향상된 기능을 활용할 수 있습니다. 동일한 SDK 도구를 사용하여 여러 런타임 버전의 .NET을 대상으로 지정할 수 있습니다.

경우에 따라 특정 버전의 SDK를 사용해야 할 수 있습니다. 파일 *global.json*에서 버전을 지정합니다.

global.json 파일 계층 구조의 아무 곳에나 배치할 수 있습니다. 특정 *global.json*가 파일 시스템 내의 위치에 따라 어떤 프로젝트에 적용되는지를 제어합니다. .NET CLI는 현재 작업 디렉터리에서 위쪽으로 경로를 반복적으로 탐색하는 *global.json* 파일을 검색합니다(프로젝트 디렉터리와 동일하지는 않음). 찾은 첫 번째 *global.json* 파일은 사용된 버전을 지정합니다. 해당 SDK 버전이 설치된 경우 해당 버전이 사용됩니다. *global.json* 지정된 SDK를 찾을 수 없는 경우 .NET CLI는 일치 규칙을 사용하여 호환되는 SDK를 선택하거나 없는 경우 실패합니다.

다음 예제에서는 *global.json* 구문을 보여줍니다.

```
```

JSON

```
{
  "sdk": {
    "version": "5.0.0"
  }
}
```

SDK 버전을 선택하는 프로세스는 다음과 같습니다.

1. `dotnet` 현재 작업 디렉터리에서 위쪽으로 경로를 반복적으로 역방향으로 탐색하는 *global.json* 파일을 검색합니다.
2. 첫 번째로 발견된 `dotnet`에 지정된 SDK를 가 사용합니다.
3. `dotnet` *global.json* 없는 경우 최신 설치된 SDK를 사용합니다.

SDK 버전 선택에 대한 자세한 내용은 global.json 개요 문서의 일치 규칙 및 rollForward 섹션을 참조하세요.

SDK 버전 업데이트

최신 기능, 성능 향상 및 버그 수정을 채택하려면 SDK의 최신 버전으로 정기적으로 업데이트하는 것이 중요합니다. SDK에 대한 업데이트를 쉽게 확인하려면 `dotnet sdk check`명령사용합니다. 또한 *global.json*사용하여 특정 버전을 선택하는 경우 새 버전을 사용할 수 있게 되면 고정된 SDK 버전을 자동으로 업데이트하는 Dependabot과 같은 도구를 고려하세요.

대상 프레임워크 모니커가 빌드 시간 API를 정의합니다.

에 정의된 API에 대해 프로젝트를 대상 프레임워크 모니커(TFM)으로 빌드합니다. 프로젝트 파일에서 대상 프레임워크 지정합니다. 다음 예제와 같이 프로젝트 파일의 `TargetFramework` 요소를 설정합니다.

XML

```
<TargetFramework>net8.0</TargetFramework>
```

여러 TFM에 대해 프로젝트를 빌드할 수 있습니다. 여러 대상 프레임워크를 설정하는 것은 라이브러리에 더 일반적이지만 애플리케이션에서도 수행할 수 있습니다. `TargetFrameworks` 속성(복수 `TargetFramework`)을 지정합니다. 대상 프레임워크는 다음 예제와 같이 세미콜론으로 구분됩니다.

XML

```
<TargetFrameworks>net8.0;net47</TargetFrameworks>
```

지정된 SDK는 제공되는 런타임의 대상 프레임워크로 제한되는 고정된 프레임워크 집합을 지원합니다. 예를 들어 .NET 8 SDK에는 `net8.0` 대상 프레임워크의 구현인 .NET 8 런타임이 포함됩니다. .NET 8 SDK는 `net7.0`, `net6.0`및 `net5.0`지원하지만 `net9.0` 이상은 지원하지 않습니다. `net9.0`위해 빌드할 .NET 9 SDK를 설치합니다.

.NET Standard

.NET Standard는 다양한 .NET 구현에서 공유하는 API 표면을 대상으로 지정하는 방법이었습니다. API 표준 자체인 .NET 5 릴리스부터 .NET Standard는 한 가지 시나리오를 제외하고는 관련성이 거의 없습니다. .NET Standard는 .NET 및 .NET Framework를 모두 대상으로 지정할 때 유용합니다. .NET 5는 모든 .NET Standard 버전을 구현합니다.

자세한 내용은 .NET 5 및 .NET Standard 참조하세요.

프레임워크 종속 앱의 전진 업데이트

dotnet run원본에서 애플리케이션을 실행할 때, **프레임워크 종속 배포**dotnet myapp.dll와 연결하여 실행하거나, **프레임워크 종속 실행 파일**`myapp.exe`와 연결하여 실행하면, `dotnet` 실행 파일은 애플리케이션의 **호스트** 입니다.

호스트는 컴퓨터에 설치된 최신 패치 버전을 선택합니다. 예를 들어 프로젝트 파일에 `net5.0` 지정하고 `5.0.2` 설치된 최신 .NET 런타임인 경우 `5.0.2` 런타임이 사용됩니다.

허용되는 `5.0.*` 버전이 없으면 새 `5.*` 버전이 사용됩니다. 예를 들어 `net5.0` 지정하고 `5.1.0` 설치한 경우 애플리케이션은 `5.1.0` 런타임을 사용하여 실행됩니다. 이 동작을 "부 버전 롤 포워드"라고 합니다. 하위 버전도 고려되지 않습니다. 허용되는 런타임이 설치되지 않으면 애플리케이션이 실행되지 않습니다.

5.0을 대상으로 하는 경우의 몇 가지 사용 예제는 동작을 보여 줍니다.

- ✔ 5.0이 지정됩니다. 5.0.3은 설치된 가장 높은 패치 버전입니다. 5.0.3이 사용됩니다.
- ✘ 5.0이 지정됩니다. 5.0.* 버전이 설치되어 있지 않습니다. 3.1.1은 설치된 가장 높은 런타임입니다. 오류 메시지가 표시됩니다.
- ✔ 5.0이 지정됩니다. 5.0.* 버전이 설치되어 있지 않습니다. 5.1.0은 설치된 가장 높은 런타임 버전입니다. 5.1.0이 사용됩니다.
- ✘ 3.0이 지정됩니다. 3.x 버전이 설치되어 있지 않습니다. 5.0.0은 설치된 가장 높은 런타임입니다. 오류 메시지가 표시됩니다.

하위 버전 전진 업데이트에는 최종 사용자에게 영향을 줄 수 있는 하나의 효과가 있습니다. 다음 시나리오를 고려합니다.

1. 애플리케이션은 5.0이 필요하다고 지정합니다.
2. 실행 시 버전 5.0.*은 설치되지 않지만 5.1.0은 설치됩니다. 버전 5.1.0이 사용됩니다.
3. 나중에 사용자가 5.0.3을 설치하고 애플리케이션을 다시 실행합니다. 이제 5.0.3이 사용됩니다.

특히 이진 데이터 직렬화와 같은 시나리오에서는 5.0.3 및 5.1.0이 다르게 동작할 수 있습니다.

롤 포워드 동작 제어

> 기본 롤포워드 동작을 재정의하기 전에 .NET 런타임 호환성수준을 숙지하세요.

애플리케이션에 대한 롤 포워드 동작은 다음과 같은 네 가지 방법으로 구성할 수 있습니다.

1. `<RollForward>` 속성을 설정하여 프로젝트 수준 설정:

   XML

   ```xml
   <PropertyGroup>
     <RollForward>LatestMinor</RollForward>
   </PropertyGroup>
   ```

2. `*.runtimeconfig.json` 파일입니다.

   이 파일은 애플리케이션을 컴파일할 때 생성됩니다. 프로젝트에서 `<RollForward>` 속성이 설정된 경우 `*.runtimeconfig.json` 파일에서 `rollForward` 설정으로 재현됩니다. 사용자는 이 파일을 편집하여 애플리케이션의 동작을 변경할 수 있습니다.

   JSON

   ```json
   {
     "runtimeOptions": {
       "tfm": "net5.0",
       "rollForward": "LatestMinor",
       "framework": {
         "name": "Microsoft.NETCore.App",
         "version": "5.0.0"
       }
     }
   }
   ```

3. `dotnet` 명령의 `--roll-forward <value>` 속성입니다.

   애플리케이션을 실행할 때 명령줄을 통해 롤 포워드 동작을 제어할 수 있습니다.

   .NET CLI

```
dotnet run --roll-forward LatestMinor
dotnet myapp.dll --roll-forward LatestMinor
myapp.exe --roll-forward LatestMinor
```

4. `DOTNET_ROLL_FORWARD` 환경 변수입니다.

우선 순위

롤 포워드 동작은 앱이 실행될 때 다음 순서로 설정되며, 번호가 높은 항목이 번호가 낮은 항목보다 우선합니다.

1. 먼저 `*.runtimeconfig.json` 구성 파일이 평가됩니다.
2. 다음으로 `DOTNET_ROLL_FORWARD` 환경 변수가 고려되어 이전 검사를 재정의합니다.
3. 마지막으로 실행 중인 애플리케이션에 전달된 `--roll-forward` 매개 변수는 다른 모든 것을 재정의합니다.

가치

그러나 롤 포워드 설정을 지정하려면 다음 값 중 하나를 사용하여 동작을 설정합니다.

테이블 확장

가치	묘사
`Minor`	지정하지 않으면 기본 . 요청된 부 버전이 누락된 경우 사용 가능한 다음 상위 부 버전(및 해당 부 버전 내에서 사용 가능한 가장 높은 패치 버전)으로 롤 포워드합니다. 요청된 부 버전이 있는 경우 `LatestPatch` 정책이 사용됩니다.
`Major`	요청된 주 버전이 누락된 경우 사용 가능한 다음 상위 주 버전(사용 가능한 가장 낮은 부 버전 및 해당 부 버전 내에서 사용 가능한 가장 높은 패치 버전)으로 롤오버합니다. 요청된 주 버전이 있는 경우 `Minor` 정책이 사용됩니다.
`LatestPatch`	요청된 주 및 부 버전에 사용할 수 있는 가장 높은 패치 버전으로 롤워드합니다. 이 값은 부 버전 전진 업데이트를 사용하지 않도록 설정합니다.
`LatestMinor`	요청된 부 버전이 있는 경우에도 요청된 주 버전(및 해당 부 버전 내에서 사용 가능한 가장 높은 패치 버전)에 사용할 수 있는 가장 높은 부 버전으로 롤워드합니다.
`LatestMajor`	요청된 주 버전이 있더라도 사용 가능한 가장 높은 주 버전(및 해당 주 버전 내에서 사용 가능한 가장 높은 부 버전 및 패치 버전)으로 롤워드합니다.
`Disable`	롤 포워드하지 말고 지정된 버전에만 바인딩합니다. 이 정책은 최신 패치로 롤 포워드하는 기능을 사용하지 않도록 설정하기 때문에 일반적인 용도로는 권장되지 않습니다. 이 값은 테스트에만 권장됩니다.

예를 들어 애플리케이션이 버전을 `8.0.0`요청하는 반면 로컬로 사용 가능한 버전은 `8.2.0`, `8.2.3`, `8.4.5`, `9.0.0` `9.0.6`, . `9.7.8` 그런 다음 해결된 버전은 각 경우에 다음과 같습니다.

테이블 확장

가치	해결된 버전	`8.0.1`이(가) 사용 가능한 경우 해결된 버전
`Minor`	`8.2.3`	`8.0.1`
`Major`	`8.2.3`	`8.0.1`
`LatestPatch`	(실패)	`8.0.1`
`LatestMinor`	`8.4.5`	`8.4.5`
`LatestMajor`	`9.7.8`	`9.7.8`
`Disable`	(실패)	(실패)

자체 포함 배포에는 선택한 런타임이 포함됩니다.

자체 포함 배포로 애플리케이션을 게시할 수 있습니다. 이 방법은 .NET 런타임 및 라이브러리를 애플리케이션과 함께 포함합니다. 자체 포함 배포에는 런타임 환경에 대한 종속성이 없습니다. 런타임 버전 선택은 런타임이 아니라 게시 시간에 발생합니다.

게시할 때 발생하는 *복원* 이벤트는 지정된 런타임 패밀리의 최신 패치 버전을 선택합니다. 예를 들어 `dotnet publish` .NET 5 런타임 제품군의 최신 패치 버전인 경우 .NET 5.0.3을 선택합니다. 대상 프레임워크(최신 설치된 보안 패치 포함)는 애플리케이션과 함께 패키지됩니다.

애플리케이션에 대해 지정된 최소 버전이 충족되지 않으면 오류가 발생합니다. `dotnet publish` 지정된 major.minor 버전 제품군 내에서 최신 런타임 패치 버전에 바인딩됩니다. `dotnet publish`는 `dotnet run`의 롤 포워드 의미를 지원하지 않습니다. 패치 및 자체 포함 배포에 대한 자세한 내용은 .NET 애플리케이션 배포의 런타임 패치 선택 문서를 참조하세요.

자체 포함 배포에는 특정 패치 버전이 필요할 수 있습니다. 다음 예제와 같이 프로젝트 파일에서 최소 런타임 패치 버전(상위 또는 하위 버전)을 재정의할 수 있습니다.

XML

```xml
<PropertyGroup>
  <RuntimeFrameworkVersion>5.0.7</RuntimeFrameworkVersion>
</PropertyGroup>
```

`RuntimeFrameworkVersion` 요소는 기본 버전 정책을 재정의합니다. 자체 포함 배포의 경우 `RuntimeFrameworkVersion` *정확한* 런타임 프레임워크 버전을 지정합니다. 프레임워크 종속 애플

리케이션의 경우 `RuntimeFrameworkVersion` *최*소 필요한 런타임 프레임워크 버전을 지정합니다.

참고 항목

- Dependabot에 의해 지원되는 에코시스템 및 리포지토리.
- .NET다운로드 및 설치.
- .NET이 이미설치되어 있는지 확인하는 방법입니다.
- .NET 런타임 및 SDK제거하는 방법입니다.

ⓘ **참고:** 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2025. 10. 22.

.NET 런타임 구성 설정

.NET은 .NET 런타임의 동작을 구성하기 위한 다음 메커니즘을 제공합니다.

테이블 확장

메커니즘	비고
runtimeconfig.json 파일	특정 앱에 설정을 적용합니다. 앱의 여러 인스턴스가 단일 시스템에서 동시에 실행되고 최적의 성능을 위해 각각을 구성하려는 경우 이 파일을 사용합니다.
MSBuild 속성	특정 앱에 설정을 적용합니다. MSBuild 속성은 *runtimeconfig.json* 설정보다 우선합니다.
환경 변수	모든 .NET 앱에 설정을 적용합니다. .NET 9부터 환경 변수가 MSBuild 속성 및 *runtimeconfig.json* 설정보다 우선합니다. 자세한 내용은 앱 런타임 구성 설정에서 환경 변수가 우선하는 경우를 참조하세요.

일부 구성 값은 메서드를 호출 AppContext.SetSwitch 하여 프로그래밍 방식으로 설정할 수도 있습니다.

> **참고 항목**
>
> 이 섹션의 문서는 .NET 런타임 자체의 구성에 대해 설명합니다. .NET Framework에서 .NET으로 앱을 마이그레이션하고 *app.config* 파일을 대체하려는 경우 **.NET으로 업그레이드한 후 현대화를** 참조하세요. .NET 앱에 사용자 지정 구성 값을 제공하는 방법에 대한 자세한 내용은 **.NET의 구성을** 참조하세요.

설명서의 이 섹션에 있는 문서는 범주별로 구성됩니다(예: 디버깅 및 가비지 수집). 해당하는 경우 *runtimeconfig.json* 파일, MSBuild 속성, 환경 변수 및 .NET Framework 프로젝트의 상호 참조 *app.config* 파일에 대한 구성 옵션이 표시됩니다.

runtimeconfig.json

프로젝트를 빌드하면 출력 디렉터리에 *[appname].runtimeconfig.json* 파일이 생성됩니다. *runtimeconfig.template.json* 파일이 프로젝트 파일과 동일한 폴더에 있는 경우 포함된 모든 구성 옵션이 *[appname].runtimeconfig.json* 파일에 삽입됩니다. 앱을 직접 빌드하는 경우 *runtimeconfig.template.json* 파일에 구성 옵션을 배치합니다. 앱을 실행하는 경우 *[appname].runtimeconfig.json* 파일에 직접 삽입합니다.

> **참고 항목**
>
> - *[appname].runtimeconfig.json* 파일은 후속 빌드에서 덮어씁니다.

- 앱의 `OutputType`가 `Exe`이(가) 아니라면 구성 옵션을 *runtimeconfig.template.json*에서 *[appname].runtimeconfig.json*로 복사하려면, 프로젝트 파일에서 `GenerateRuntimeConfigurationFiles`를 `true`로 명시적으로 설정해야 합니다. *runtimeconfig.json* 파일이 필요한 앱의 경우 이 속성의 기본값은 .입니다`true`.

runtimeconfig.json 또는 *runtimeconfig.template.json* 파일의 *configProperties* 섹션에서 런타임 구성 옵션을 지정합니다. 이 섹션에는 다음과 같은 형식이 있습니다.

JSON

```
"configProperties": {
  "config-property-name1": "config-value1",
  "config-property-name2": "config-value2"
}
```

예제 [appname].runtimeconfig.json 파일

출력 JSON 파일에 옵션을 배치하는 경우, 옵션을 `runtimeOptions` 속성 아래에 배치하세요.

JSON

```
{
  "runtimeOptions": {
    "tfm": "net8.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "8.0.0"
    },
    "configProperties": {
      "System.Globalization.UseNls": true,
      "System.Net.DisableIPv6": true,
      "System.GC.Concurrent": false,
      "System.Threading.ThreadPool.MinThreads": 4,
      "System.Threading.ThreadPool.MaxThreads": 25
    }
  }
}
```

예제 runtimeconfig.template.json 파일

템플릿 JSON 파일에 옵션을 배치하는 경우 속성을 **생략**합니다`runtimeOptions`.

JSON

```
{
  "configProperties": {
```

```
    "System.Globalization.UseNls": true,
    "System.Net.DisableIPv6": true,
    "System.GC.Concurrent": false,
    "System.Threading.ThreadPool.MinThreads": "4",
    "System.Threading.ThreadPool.MaxThreads": "25"
  }
}
```

MSBuild 속성

SDK 스타일 .NET 프로젝트의 *.csproj* 또는 *.vbproj* 파일에서 MSBuild 속성을 사용하여 일부 런타임 구성 옵션을 설정할 수 있습니다. MSBuild 속성은 *runtimeconfig.template.json* 파일에 설정된 옵션보다 우선합니다.

특정 MSBuild 속성이 없는 런타임 구성 설정의 경우 MSBuild 항목을 대신 사용할 `RuntimeHostConfigurationOption` 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다.

다음은 .NET 런타임의 동작을 구성하기 위한 MSBuild 속성이 있는 SDK 스타일 프로젝트 파일의 예입니다.

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
    <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>
  </PropertyGroup>

  <ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Globalization.UseNls"
Value="true" />
    <RuntimeHostConfigurationOption Include="System.Net.DisableIPv6" Value="true" />
  </ItemGroup>

</Project>
```

런타임의 동작을 구성하기 위한 MSBuild 속성은 가 비지 수집과 같은 각 영역에 대한 개별 문서에 나와 있습니다. 또한 SDK 스타일 프로젝트에 대한 MSBuild 속성 참조의 런타임 구성 섹션에 나열됩니다.

환경 변수

환경 변수를 사용하여 일부 런타임 구성 정보를 제공할 수 있습니다. 환경 변수로 지정된 구성 노브는 일반적으로 접두 `DOTNET_`사를 갖습니다. (.NET Framework 런타임 구성의 경우 대신 접두사를 `COMPlus_` 사용합니다.)

> ⓘ **참고 항목**
>
> .NET 9부터 환경 변수가 MSBuild 속성 및 *runtimeconfig.json* 설정보다 우선합니다. 이 호환성이 손상되는 변경에 대한 자세한 내용은 **앱 런타임 구성 설정에서 환경 변수가 우선**하는 것을 참조하세요.

Windows 제어판, 명령줄 또는 Windows 및 Unix 기반 시스템에서 메서드를 호출 Environment.SetEnvironmentVariable(String, String) 하여 프로그래밍 방식으로 환경 변수를 정의할 수 있습니다.

다음 예제에서는 명령줄에서 환경 변수를 설정하는 방법을 보여 줍니다.

shell

```
# Windows
set DOTNET_GCRetainVM=1

# Powershell
$env:DOTNET_GCRetainVM="1"

# Unix
export DOTNET_GCRetainVM=1
```

참고하십시오

- .NET 환경 변수

Last updated on 2026. 01. 22.

컴파일을 위한 런타임 구성 옵션

이 문서에서는 .NET 컴파일을 구성하는 데 사용할 수 있는 설정을 자세히 설명합니다.

계층화된 컴파일

- JIT(Just-In-Time) 컴파일러에서 계층화된 컴파일을 사용하는지 여부를 구성합니다. 계층화된 컴파일은 다음 두 계층을 통해 메서드를 전환합니다.
  - 첫 번째 계층은 코드를 더 빠르게 생성하거나(빠른 JIT) 미리 컴파일된 코드(ReadyToRun)를 로드합니다.
  - 두 번째 계층은 백그라운드에서 최적화된 코드를 생성합니다("JIT 최적화").
- .NET Core 3.0 이상에서는 계층화된 컴파일이 기본적으로 사용하도록 설정됩니다.
- .NET Core 2.1 및 2.2에서는 계층화된 컴파일이 기본적으로 사용하지 않도록 설정됩니다.
- 자세한 내용은 계층화된 컴파일 가이드를 참조하세요.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Runtime.TieredCompilation`	`true` - 사용 `false` - 사용 안 함
MSBuild 속성	`TieredCompilation`	`true` - 사용 `false` - 사용 안 함
환경 변수	`DOTNET_TieredCompilation`	`1` - 사용 `0` - 사용 안 함

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation": false
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Runtime.TieredCompilation": false
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilation>false</TieredCompilation>
  </PropertyGroup>

</Project>
```

빠른 JIT

- JIT 컴파일러에서 *빠른 JIT*를 사용하는지 여부를 구성합니다. 루프를 포함하지 않고 미리 컴파일된 코드를 사용할 수 없는 메서드의 경우 빠른 JIT는 최적화 없이 더 빠르게 컴파일합니다.
- 빠른 JIT를 사용하도록 설정하면 시작 시간이 줄어들지만 성능 특성이 저하된 코드를 생성할 수 있습니다. 예를 들어 코드는 더 많은 스택 공간을 사용하고, 더 많은 메모리를 할당하고, 느리게 실행할 수 있습니다.
- 빠른 JIT를 사용하지 않도록 설정하지만 계층화된 컴파일 을 사용하는 경우 미리 컴파일된 코드만 계층화된 컴파일에 참여합니다. 메서드가 ReadyToRun을 사용하여 미리 컴파일되지 않은 경우 JIT 동작은 계층화된 컴파일을 사용하지 않도록 설정한 경우와 동일합니다.
- .NET Core 3.0 이상에서는 빠른 JIT가 기본적으로 사용하도록 설정됩니다.
- .NET Core 2.1 및 2.2에서는 빠른 JIT가 기본적으로 사용하지 않도록 설정됩니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Runtime.TieredCompilation.QuickJit`	`true` - 사용 `false` - 사용 안 함
MSBuild 속성	`TieredCompilationQuickJit`	`true` - 사용 `false` - 사용 안 함
환경 변수	`DOTNET_TC_QuickJit`	`1` - 사용 `0` - 사용 안 함

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation.QuickJit": false
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Runtime.TieredCompilation.QuickJit": false
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
  </PropertyGroup>

</Project>
```

루프에 대한 빠른 JIT

- JIT 컴파일러가 루프를 포함하는 메서드에서 빠른 JIT를 사용하는지 여부를 구성합니다.
- 루프에 대해 빠른 JIT를 사용하도록 설정하면 시작 성능이 향상될 수 있습니다. 그러나 장기 실행 루프는 오랫동안 최적화되지 않은 코드에서 중단될 수 있습니다.
- 빠른 JIT를 사용하지 않도록 설정하면 이 설정이 적용되지 않습니다.
- 이 설정을 생략하면 루프가 포함된 메서드에는 빠른 JIT가 사용되지 않습니다. 이는 값을 `false`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Runtime.TieredCompilation.QuickJitForLoops`	`false` - 사용 안 함 `true` - 사용
MSBuild 속성	`TieredCompilationQuickJitForLoops`	`false` - 사용 안 함 `true` - 사용
환경 변수	`DOTNET_TC_QuickJitForLoops`	`0` - 사용 안 함 `1` - 사용

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation.QuickJitForLoops": false
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Runtime.TieredCompilation.QuickJitForLoops": false
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilationQuickJitForLoops>true</TieredCompilationQuickJitForLoops>
  </PropertyGroup>

</Project>
```

ReadyToRun

- .NET 런타임에서 사용 가능한 ReadyToRun 데이터가 있는 이미지에 미리 컴파일된 코드를 사용할지 여부를 구성합니다. 이 옵션을 사용하지 않도록 설정하면 런타임이 JIT 컴파일 프레임워크 코드로 강제 적용됩니다.
- 자세한 내용은 실행 준비를 참조하세요.
- 이 설정을 생략하면 .NET은 사용 가능한 경우 ReadyToRun 데이터를 사용합니다. 이는 값을 `1`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	가치들
환경 변수	`DOTNET_ReadyToRun`	`1` - 사용 `0` - 사용 안 함

프로필 기반 최적화

이 설정을 사용하면 .NET 6 이상 버전에서 동적(계층화된) PGO(프로필 기반 최적화)를 사용할 수 있습니다.

테이블 확장

	설정 이름	가치들
환경 변수	`DOTNET_TieredPGO`	`1` - 사용 `0` - 사용 안 함
MSBuild 속성	`TieredPGO`	`true` - 사용 `false` - 사용 안 함

PGO(프로필 기반 최적화)는 JIT 컴파일러가 가장 자주 사용되는 형식 및 코드 경로 측면에서 최적화된 코드를 생성하는 위치입니다. *동적인* PGO는 계층화된 컴파일과 함께 작동하여 계층 0 중에 배치되는 추가 계측을 기반으로 코드를 더욱 최적화합니다.

예시

프로젝트 파일:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
    <TieredPGO>true</TieredPGO>
  </PropertyGroup>

</Project>
```

Last updated on 2025. 11. 22.

디버깅 및 프로파일링을 위한 런타임 구성 옵션

이 문서에서는 .NET 디버깅 및 프로파일링을 구성하는 데 사용할 수 있는 설정을 자세히 설명합니다.

> ⓘ **참고 항목**
>
> .NET 11부터 프로파일러 환경 변수는 접두사와 `DOTNET` 접두사를 모두 `CORECLR` 지원합니다. `DOTNET` 접두사는 새 표준 `CORECLR` 이며 이전 버전과의 호환성을 위해 유지 관리되며 나중에 제거될 수 있습니다.

진단 사용

- 디버거, 프로파일러 및 EventPipe 진단의 사용 여부를 구성합니다.
- 이 설정을 생략하면 진단이 활성화됩니다. 이는 값을 `1`으로 설정하는 것과 같습니다.

⛶ **테이블 확장**

	설정 이름	가치들
runtimeconfig.json	N/A	N/A
환경 변수	`DOTNET_EnableDiagnostics`	`1` - 사용 `0` - 사용 안 함

프로파일링 사용

- 현재 실행 중인 프로세스에 대해 프로파일링을 사용할 수 있는지 여부를 구성합니다.
- 이 설정을 생략하면 프로파일링이 비활성화됩니다. 이는 값을 `0`으로 설정하는 것과 같습니다.
- 프로파일링을 로드하려면 프로파일링을 사용하도록 설정하는 것 외에도 프로파일러 GUID 및 프로파일러 위치도 이러한 설정을 사용하여 구성해야 합니다.

⛶ **테이블 확장**

	설정 이름	가치들
runtimeconfig.json	N/A	N/A

	설정 이름	가치들
환경 변수	`DOTNET_ENABLE_PROFILING`	`0` - 사용 안 함 `1` - 사용

프로파일러 GUID

- 현재 실행 중인 프로세스에 로드할 프로파일러의 GUID를 지정합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	N/A	N/A
환경 변수	`CORECLR_PROFILER` 또는 `DOTNET_PROFILER`	*string-guid*

프로파일러 위치

프로파일링을 사용하도록 설정하면 프로파일러를 환경 변수(크로스 플라트) 또는 레지스트리(Windows 전용)를 통해 두 가지 방법으로 로드할 수 있습니다. 프로파일러 경로 환경 변수는 둘 다 지정된 경우 레지스트리의 COM 라이브러리 경로보다 우선합니다.

환경 변수(크로스 플라트)

- 현재 실행 중인 프로세스(또는 32비트 또는 64비트 프로세스)로 로드할 프로파일러 DLL의 경로를 지정합니다.
- 둘 이상의 변수가 설정되면 비트 수별 변수가 우선합니다. 로드할 프로파일러의 비트를 지정합니다.

테이블 확장

	설정 이름	가치들
환경 변수	`CORECLR_PROFILER_PATH` 또는 `DOTNET_PROFILER_PATH`	*string-path*
환경 변수	`CORECLR_PROFILER_PATH_32` 또는 `DOTNET_PROFILER_PATH_32`	*string-path*
환경 변수	`CORECLR_PROFILER_PATH_64` 또는 `DOTNET_PROFILER_PATH_64`	*string-path*
환경 변수	`CORECLR_PROFILER_PATH_ARM32` 또는 `DOTNET_PROFILER_PATH_ARM32`	*string-path*
환경 변수	`CORECLR_PROFILER_PATH_ARM64` 또는 `DOTNET_PROFILER_PATH_ARM64`	*string-path*

레지스트리를 통해(Windows에만 해당)

Windows에서 `DOTNET_PROFILER_PATH*` 실행하는 동안 환경 변수가 설정되지 않은 경우 coreclr는 레지스트리에서 `DOTNET_PROFILER` CLSID를 조회하여 프로파일러의 DLL에 대한 전체 경로를 찾습니다. COM 서버 DLL과 마찬가지로 프로파일러의 CLSID는 HKLM 및 HKCU의 클래스를 병합하는 HKEY_CLASSES_ROOT 아래에서 조회됩니다.

성능 맵 및 jit 덤프 내보내기

- 성능 맵 또는 jit 덤프를 사용하거나 사용하지 않도록 설정합니다. 이러한 파일을 사용하면 Linux `perf` 도구와 같은 타사 도구가 동적으로 생성된 코드 및 미리 컴파일된 R2R(ReadyToRun) 모듈에 대해 사람이 읽을 수 있는 이름을 제공할 수 있습니다.
- 이 설정을 생략하면 성능 맵 및 jit 덤프 파일 작성이 모두 비활성화됩니다. 이는 값을 `0`으로 설정하는 것과 같습니다.
- 성능 맵을 사용하지 않도록 설정하면 관리되는 모든 호출 사이트를 제대로 확인할 수 없습니다.
- Linux 커널 버전에 따라 두 형식 모두 도구에서 `perf` 지원됩니다.
- 성능 맵 또는 jit 덤프를 사용하도록 설정하면 최대 20개의% 오버헤드가 발생할 수 있지만 종종 훨씬 적습니다. 성능 영향을 최소화하려면 성능 맵 또는 지트 덤프를 선택적으로 사용하도록 설정하는 것이 좋지만 둘 다 사용하도록 설정하는 것이 좋습니다. 이 영향은 애플리케이션이 JITing 코드인 경우에만 발생합니다. 시작 시 발생하는 경우가 많지만 애플리케이션이 새 코드 경로를 처음으로 실행하는 경우 나중에 발생할 수 있습니다.

다음 표에서는 성능 맵과 jit 맵을 비교합니다.

테이블 확장

포맷	Description	다음에서 지원됨
성능 맵	`/tmp/perf-<pid>.map`동적으로 생성된 코드에 대한 기호 정보를 포함하는 내보내기입니다. `/tmp/perfinfo-<pid>.map`R2R(ReadyToRun) 모듈 기호 정보를 포함하고 PerfCollect에서 사용되는 내보내기	성능 맵은 모든 Linux 커널 버전에서 지원됩니다.
Jit 덤프	jit 덤프 형식은 성능 맵을 대체하고 보다 자세한 기호 정보를 포함합니다. 사용하도록 설정하면 jit 덤프가 파일에 출력 `/tmp/jit-<pid>.dump` 됩니다.	Linux 커널 버전 5.4 이상.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	N/A	N/A
환경 변수	`DOTNET_PerfMapEnabled`	`0` - 사용 안 함 `1` - 성능 맵 및 jit 덤프 모두 사용 `2` - jit 덤프 사용 `3` - 성능 맵 사용

성능 로그 표식

- 지정된 신호를 perf 로그의 표식으로 허용하고 무시하도록 설정하거나 사용하지 않도록 설정합니다.
- 이 설정을 생략하면 지정된 신호가 무시되지 않습니다. 이는 값을 `0`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	N/A	N/A
환경 변수	`DOTNET_PerfMapIgnoreSignal`	`0` - 사용 안 함 `1` - 사용

> **참고 항목**
>
> 이 설정은 DOTNET_PerfMapEnabled 생략하거나 사용하지 않도록 설정된 `0` 경우 무시됩니다.

Last updated on 2025. 11. 22.

가비지 수집을 위한 런타임 구성 옵션

이 페이지에는 .NET 런타임 GC(가비지 수집기)의 설정에 대한 정보가 포함되어 있습니다. 실행 중인 앱의 최고 성능을 달성하려는 경우 이러한 설정을 사용해 보시기 바랍니다. 그러나 기본값으로도 일반적인 상황에 있는 대부분의 애플리케이션에서 최적의 성능을 얻을 수 있습니다.

이 페이지에서는 설정이 그룹별로 정리되어 있습니다. 각 그룹에 포함된 설정은 특정 결과를 얻기 위해 서로 함께 사용되는 경우가 많습니다.

> **참고 항목**
>
> - 이러한 구성은 GC가 초기화될 때(일반적으로 프로세스 시작 시간 동안을 의미)만 런타임에서 읽습니다. 프로세스가 이미 실행 중인 상태로 환경 변수를 변경하는 경우 해당 프로세스에는 변경 내용이 반영되지 않습니다. 런타임 시 API를 통해 변경할 수 있는 설정(예: **대기 시간 수준**)은 이 페이지에서 생략됩니다.
> - GC는 프로세스별로 수행되므로 이러한 구성을 컴퓨터 수준에서 설정하는 것은 거의 의미가 없습니다. 예를 들어 컴퓨터의 모든 .NET 프로세스에서 서버 GC 또는 동일한 힙 하드 제한을 사용하도록 하지는 않을 것입니다.
> - 숫자 값의 경우, *runtimeconfig.json* 또는 *runtimeconfig.template.json* 파일에 있는 설정에는 10진수 표기법을 사용하고, 환경 변수 설정에는 16진수 표기법을 사용합니다. 16진수 값의 경우 "0x" 접두사를 사용하거나 사용하지 않고 지정할 수 있습니다.
> - 환경 변수를 사용하는 경우 .NET 6 이상 버전은 `DOTNET_` 접두사 대신 `COMPlus_`을 표준화합니다. 그러나 접두사는 `COMPlus_` 계속 작동합니다. 이전 버전의 .NET 런타임을 사용하는 경우에도 `COMPlus_` 접두사(예: `COMPlus_gcServer`)를 사용해야 합니다.

구성 지정 방법

.NET 런타임 버전이 다양한 경우 구성 값을 지정하는 방법에는 여러 가지가 있습니다. 다음 표에 요약되어 있습니다.

테이블 확장

구성 위치	이 위치가 적용되는 .NET 버전	형식	해석 방법
runtimeconfig.json 파일/ runtimeconfig.template.json 파일	.NET(Core)	n	n은 10진수 값으로 해석합니다.

구성 위치	이 위치가 적용되는 .NET 버전	형식	해석 방법
환경 변수	.NET Framework, .NET	0xn 또는 n	n은 두 서식 중 하나에서 16진수 값으로 해석합니다.
app.config 파일	.NET Framework	0xn	n은 16진수 값[1]으로 해석합니다.

[1] `0x` 접두사가 app.config 파일 설정에 없어도 값을 지정할 수는 있지만 권장하지는 않습니다. .NET Framework 4.8 이상에서는 버그로 인해 `0x` 접두사 없이 지정된 값이 16진수로 해석되지만, 이전 버전의 .NET Framework에서는 10진수로 해석됩니다. 구성을 변경할 필요가 없도록 하려면 app.config 파일에서 값을 지정할 때 `0x` 접두사를 사용합니다.

예를 들어 이름이 `GCHeapCount`인 .NET Framework 앱의 에 대한 12 힙을 지정하려면, 다음 XML을 *A.exe.config* 파일에 추가합니다.

XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    ...
    <runtime>
        <gcServer enabled="true"/>
        <GCHeapCount>0xc</GCHeapCount>
    </runtime>
</configuration>
```

.NET(Core)과 .NET Framework 모두 환경 변수를 사용할 수 있습니다.

.NET 6 이상 버전을 사용하는 Windows:

Windows 명령 프롬프트

```
SET DOTNET_gcServer=1
SET DOTNET_GCHeapCount=c
```

.NET 6 이상 버전을 사용하는 다른 운영 체제에서:

Bash

```
export DOTNET_gcServer=1
export DOTNET_GCHeapCount=c
```

.NET Framework를 사용하지 않는 경우 *runtimeconfig.json*이나 *runtimeconfig.template.json* 파일의 값도 설정할 수 있습니다.

runtimeconfig.json 파일:

JSON

```
{
  "runtimeOptions": {
   "configProperties": {
      "System.GC.Server": true,
      "System.GC.HeapCount": 12
   }
  }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
  "configProperties": {
    "System.GC.Server": true,
    "System.GC.HeapCount": 12
  }
}
```

가비지 수집 버전

가비지 수집의 2가지 주요 버전은 워크스테이션 GC와 서버 GC입니다. 둘 사이의 차이점에 대한 자세한 내용은 워크스테이션 및 가비지 수집을 참조하세요.

가비지 수집의 하위 버전은 백그라운드와 비동시입니다.

다음 설정을 사용하여 가비지 수집의 버전을 선택합니다.

- 워크스테이션과 서버 GC 비교
- 백그라운드 GC

워크스테이션과 서버 비교

- 애플리케이션이 워크스테이션 가비지 수집과 서버 가비지 수집 중 어느 것을 사용하는지 구성합니다.
- 기본값: 워크스테이션 가비지 수집. 이는 값을 `false`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.Server`	`false` - 워크스테이션 `true` - 서버	.NET Core 1.0

	설정 이름	값	도입된 버전
MSBuild 속성	`ServerGarbageCollection`	`false` - 워크스테이션 `true` - 서버	.NET Core 1.0
환경 변수	`DOTNET_gcServer`	`0` - 워크스테이션 `1` - 서버	.NET 6
.NET Framework의 app.config	GCServer	`false` - 워크스테이션 `true` - 서버	

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.Server": true
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.Server": true
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
  </PropertyGroup>

</Project>
```

백그라운드 GC

- 백그라운드(동시) 가비지 수집이 사용하도록 설정되었는지 여부를 구성합니다.
- 기본값: 백그라운드 GC를 사용합니다. 이는 값을 `true`으로 설정하는 것과 같습니다.
- 자세한 내용은 백그라운드 가비지 수집을 참조하세요.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.Concurrent`	`true` - 백그라운드 GC `false` - 비동시 GC	.NET Core 1.0
MSBuild 속성	`ConcurrentGarbageCollection`	`true` - 백그라운드 GC `false` - 비동시 GC	.NET Core 1.0
환경 변수	`DOTNET_gcConcurrent`	`1` - 백그라운드 GC `0` - 비동시 GC	.NET 6
.NET Framework의 app.config	gcConcurrent	`true` - 백그라운드 GC `false` - 비동시 GC	

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.Concurrent": false
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.Concurrent": false
   }
}
```

프로젝트 파일:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
  </PropertyGroup>

</Project>
```

독립 실행형 GC

기본 GC 구현 대신 독립 실행형 가비지 수집기를 사용하려면 경로(.NET 9 이상 버전) 또는 *GC 네이티브 라이브러리의 이름을* 지정할 *수 있습니다*.

경로

- 런타임이 기본 GC 구현 대신 로드하는 GC 네이티브 라이브러리의 전체 경로를 지정합니다. 보안을 위해 이 위치는 잠재적으로 악의적인 변조로부터 보호되어야 합니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.Path`	*string_path*	.NET 9
환경 변수	`DOTNET_GCPath`	*string_path*	.NET 9

속성

- 런타임이 기본 GC 구현 대신 로드하는 GC 네이티브 라이브러리의 이름을 지정합니다. 경로 구성이 도입되어 .NET 9에서 동작이 변경되었습니다.

  .NET 8 및 이전 버전:
  - 라이브러리 이름만 지정한 경우 라이브러리는 .NET 런타임과 동일한 디렉터리에 있어야 합니다(*Windows에서는 coreclr.dll* , Linux에서는 *libcoreclr.so* , *OSX의 경우 libcoreclr.dylib*).
  - 예를 들어 "를 지정하는 경우 값은 상대 경로일 수도 있습니다. Windows의 \clrgc.dll"이고, *clrgc.dll* .NET 런타임 디렉터리의 부모 디렉터리에서 로드됩니다.

  .NET 9 이상 버전에서 이 값은 파일 이름만 지정합니다(경로는 허용되지 않음).
  - .NET은 앱 `Main` 의 메서드가 포함된 어셈블리가 있는 디렉터리에서 지정한 이름을 검색합니다.

- 파일을 찾을 수 없으면 .NET 런타임 디렉터리가 검색됩니다.

- 경로 구성이 지정된 경우 이 구성 설정은 무시됩니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.Name`	*string_name*	.NET 7
환경 변수	`DOTNET_GCName`	*string_name*	.NET 6

LOH 특정 설정

매우 큰 개체 허용

- 64비트 플랫폼에서 총 크기가 2기가바이트(GB)보다 큰 배열에 대한 가비지 수집기 지원을 구성합니다.
- 기본값: GC가 2GB를 초과하는 배열을 지원합니다. 이는 값을 `1`으로 설정하는 것과 같습니다.
- 이 옵션은 이후 버전의 .NET에서 더 이상 사용되지 않을 수 있습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	해당 없음	해당 없음	해당 없음
환경 변수	`DOTNET_gcAllowVeryLargeObjects`	`1` - 사용 `0` - 사용 안 함	.NET 6
.NET Framework의 app.config	gcAllowVeryLargeObjects	`1` - 사용 `0` - 사용 안 함	.NET Framework 4.5

큰 개체 힙 임계값

- 개체가 큰 개체 힙(LOH)으로 이동되도록 하는 임계값 크기를 바이트 단위로 지정합니다.
- 기본 임계값은 85,000바이트입니다.
- 사용자가 지정하는 값은 기본 임계값보다 커야 합니다.
- 이 값은 런타임에 의해 현재 구성에서 가능한 최대 크기로 제한될 수 있습니다. API를 통해 런타임에 사용 중인 값을 검사할 GC.GetConfigurationVariables() 수 있습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.LOHThreshold`	*10진수 값*	.NET Core 3.0
환경 변수	`DOTNET_GCLOHThreshold`	*16진수 값*	.NET 6
.NET Framework의 app.config	GCLOHThreshold	*10진수 값*	.NET Framework 4.8

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.LOHThreshold": 120000
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.LOHThreshold": 120000
   }
}
```

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어, 임계값 크기를 120,000바이트로 설정하려면 JSON 파일의 경우 값을 120000으로 지정하고 환경 변수의 경우 값을 0x1D4C0 또는 1D4C0으로 지정합니다.

모든 GC 버전에 대한 리소스 사용량 관리

다음 설정은 GC의 모든 버전에 적용됩니다.

- 힙 하드 제한
- 힙 하드 제한 비율
- 개체별 힙 하드 제한
- 개체당 힙 하드 제한 백분율
- 큰 페이지
- 지역 범위
- 지역 크기
- 최소 메모리 비율
- VM 유지
- 메모리 절약

힙 하드 제한

- 힙 하드 제한은 GC 힙 및 GC 부기용 최대 커밋 크기(바이트)로 정의됩니다.
- 이 설정은 64비트 컴퓨터에만 적용됩니다.
- 이 제한이 구성되지 않았지만 프로세스가 메모리 제한 환경에서 실행되는 경우, 즉 지정된 메모리 제한이 있는 컨테이너 내에서 기본값이 설정됩니다. 이 기본값은 컨테이너에 대한 메모리 제한의 20MB 또는 75% 중 더 큰 값입니다.
- 개체별 힙 하드 제한이 구성된 경우 이 설정은 무시됩니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimit`	*10진수 값*	.NET Core 3.0
환경 변수	`DOTNET_GCHeapHardLimit`	*16진수 값*	.NET 6

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

```
JSON
```

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapHardLimit": 209715200
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.HeapHardLimit": 209715200
   }
}
```

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어, 힙의 하드 한도를 200메비바이트(MiB) 지정하려면 JSON 파일의 경우 값을 209715200으로 지정하고 환경 변수의 경우 값을 0xC800000 또는 C800000으로 지정합니다.

힙 하드 제한 비율

- 힙 하드 제한을 총 실제 메모리의 백분율로 지정합니다. 프로세스가 메모리 제한 환경에서 실행되는 경우, 즉 지정된 메모리 제한이 있는 컨테이너 내에서 총 실제 메모리는 메모리 제한입니다. 그렇지 않으면 컴퓨터에서 사용할 수 있는 것입니다.
- 이 설정은 64비트 컴퓨터에만 적용됩니다.
- 개체별 힙 하드 제한이 구성되거나 힙 하드 제한이 구성된 경우 이 설정은 무시됩니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitPercent`	*10진수 값*	.NET Core 3.0
환경 변수	`DOTNET_GCHeapHardLimitPercent`	*16진수 값*	.NET 6

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapHardLimitPercent": 30
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.HeapHardLimitPercent": 30
   }
}
```

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어, 힙의 사용량을 30%로 제한하려면 JSON 파일의 경우 값을 30으로 지정하고 환경 변수의 경우 값을 0x1E 또는 1E로 지정합니다.

개체별 힙 하드 제한

개체별 힙 기준으로 GC의 힙 하드 제한을 지정할 수 있습니다. LOH(대형 개체 힙), SOH(작은 개체 힙), POH(고정 개체 힙) 등의 다양한 힙이 있습니다.

- `DOTNET_GCHeapHardLimitSOH`, `DOTNET_GCHeapHardLimitLOH` 또는 `DOTNET_GCHeapHardLimitPOH` 설정 중 하나의 값을 지정하는 경우 `DOTNET_GCHeapHardLimitSOH` 및 `DOTNET_GCHeapHardLimitLOH`의 값도 지정해야 합니다. 그러지 않으면 런타임이 초기화되지 않습니다.
- `DOTNET_GCHeapHardLimitPOH`의 기본값은 0입니다. `DOTNET_GCHeapHardLimitSOH` 및 `DOTNET_GCHeapHardLimitLOH`에는 기본값이 없습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitSOH`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHeapHardLimitSOH`	*16진수 값*	.NET 6

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitLOH`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHeapHardLimitLOH`	*16진수 값*	.NET 6

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitPOH`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHeapHardLimitPOH`	*16진수 값*	.NET 6

이러한 구성 설정에는 지정된 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어, 힙의 하드 한도를 200메비바이트(MiB) 지정하려면 JSON 파일의 경우 값을 209715200으로 지정하고 환경 변수의 경우 값을 0xC800000 또는 C800000으로 지정합니다.

개체당 힙 하드 제한 백분율

개체별 힙 기준으로 GC의 힙 하드 제한을 지정할 수 있습니다. LOH(대형 개체 힙), SOH(작은 개체 힙), POH(고정 개체 힙) 등의 다양한 힙이 있습니다.

- `DOTNET_GCHeapHardLimitSOHPercent`, `DOTNET_GCHeapHardLimitLOHPercent` 또는 `DOTNET_GCHeapHardLimitPOHPercent` 설정 중 하나의 값을 지정하는 경우 `DOTNET_GCHeapHardLimitSOHPercent` 및 `DOTNET_GCHeapHardLimitLOHPercent`의 값도 지정해야 합니다. 그러지 않으면 런타임이 초기화되지 않습니다.

- `DOTNET_GCHeapHardLimitSOH`, `DOTNET_GCHeapHardLimitLOH` 및 `DOTNET_GCHeapHardLimitPOH`를 지정하면 해당 설정은 무시됩니다.
- 값이 1이면 GC가 해당 개체 힙의 총 실제 메모리 중 1%를 사용합니다.
- 각 값은 0보다 크고 100보다 작아야 합니다. 또한 백분율 값 3개의 합계는 100보다 작아야 합니다. 그렇지 않으면 런타임이 초기화되지 않습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitSOHPercent`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHeapHardLimitSOHPercent`	*16진수 값*	.NET 6

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitLOHPercent`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHeapHardLimitLOHPercent`	*16진수 값*	.NET 6

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapHardLimitPOHPercent`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHeapHardLimitPOHPercent`	*16진수 값*	.NET 6

이러한 구성 설정에는 지정된 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어, 힙의 사용량을 30%로 제한하려면 JSON 파일의 경우 값을 30으로 지정하고 환경 변수의 경우 값을 0x1E 또는 1E로 지정합니다.

큰 페이지

- 힙의 하드 한도가 설정된 경우 큰 페이지를 사용할지 여부를 지정합니다.

- 기본값: 힙 하드 한도가 설정된 경우 대용량 페이지를 사용하지 않습니다. 이는 값을 `0`으로 설정하는 것과 같습니다.
- 이것은 실험적인 설정입니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	해당 없음	해당 없음	해당 없음
환경 변수	`DOTNET_GCLargePages`	`0` - 사용 안 함 `1` - 사용	.NET 6

지역 범위

.NET 7부터 GC 힙은 64비트 Windows 및 Linux용 세그먼트에서 지역으로 물리적 표현을 전환했습니다. (자세한 내용은 Maoni Stephens의 블로그 문서를 참조하세요.) 이 변경으로 GC는 초기화 중에 다양한 가상 메모리를 예약합니다. 이는 커밋되지 않고 메모리만 예약합니다(GC 힙 크기는 커밋된 메모리). GC 힙이 커밋할 수 있는 최대 범위를 정의하는 범위일 뿐입니다. 대부분의 애플리케이션은 거의 커밋할 필요가 없습니다.

다른 구성이 없고 메모리가 제한된 환경에서 실행되지 않는 경우(일부 GC 구성이 설정됨) 기본적으로 256GB가 예약됩니다. 256GB 이상의 실제 메모리를 사용할 수 있는 경우 해당 용량의 두 배가 됩니다.

힙당 하드 제한이 설정된 경우 예약 범위는 총 하드 제한과 동일합니다. 단일 하드 제한 구성이 설정된 경우 이 범위는 해당 금액의 5배입니다.

이 범위는 총 가상 메모리의 양으로 제한됩니다. 일반적으로 64비트에서는 문제가 되지 않지만 프로세스에 설정된 가상 메모리 제한이 있을 수 있습니다. 이 범위는 해당 금액의 절반으로 제한됩니다. 예를 들어 구성을 `HeapHardLimit` 1GB로 설정하고 프로세스에 4GB 가상 메모리 제한이 설정된 경우 이 범위는 2GB입니다 `min (5x1GB, 4GB/2)`.

API를 GC.GetConfigurationVariables() 사용하여 이름 `GCRegionRange` 아래에 이 범위의 값을 볼 수 있습니다. `E_OUTOFMEMORY` 런타임 초기화 중에 이 범위를 예약해야 하는지 확인하려면 GC 초기화 중에 Windows에서 호출 `VirtualAlloc` 하거나 `MEM_RESERVE` Linux에서 호출 `mmap` 한 내용을 확인하고 `PROT_NONE` 해당 호출에서 OOM이 있는지 확인합니다. 이 예약 호출이 실패하는 경우 다음 구성 설정을 통해 변경할 수 있습니다. 예약 금액에 대한 권장 사항은 GC 힙의 커밋된 크기의 2~5배입니다. 시나리오에서 큰 할당을 많이 하지 않는 경우(UOH에 할당되거나 UOH 지역 크기보다 클 수 있음) 커밋된 크기의 두 배는 안전해야 합니다. 그렇지 않으면 더 큰 지역을 위한 공간을 만들기 위해 너무 자주 전체 압축 GC가 발생하지 않도록 더 크게 만들 수 있습니다. GC 힙의 커밋된 크기를 모르는 경우 이를 프로세스에 사용할 수 있는 실제 메모리의 두 배로 설정할 수 있습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.RegionRange`	*10진수 값*	.NET 10
환경 변수	`DOTNET_GCRegionRange`	*16진수 값*	.NET 7

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

지역 크기

.NET 7부터 GC 힙은 64비트 Windows 및 Linux용 세그먼트에서 지역으로 실제 표현을 전환했습니다. (자세한 내용은 Maoni Stephens의 블로그 문서를 참조하세요.) 기본적으로 각 지역은 SOH의 경우 4MB입니다. UOH(LOH 및 POH)의 경우 SOH 지역 크기의 8배입니다. 이 구성을 사용하여 SOH 지역 크기를 변경할 수 있으며 UOH 지역은 그에 따라 조정됩니다.

지역은 필요한 경우에만 할당되므로 일반적으로 지역 크기에 대해 걱정할 필요가 없습니다. 그러나 이 크기를 조정할 수 있는 두 가지 경우가 있습니다.

- GC 힙이 매우 작은 프로세스의 경우 지역 크기를 더 작게 변경하는 것이 GC 고유의 부기에서 네이티브 메모리 사용에 유용합니다. 권장 사항은 1MB입니다.
- Linux에서 메모리 매핑 수를 줄여야 하는 경우 지역 크기를 더 크게 변경할 수 있습니다(예: 32MB).

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.RegionSize`	*10진수 값*	.NET 10
환경 변수	`DOTNET_GCRegionSize`	*16진수 값*	.NET 7

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

최소 메모리 비율

메모리 로드는 사용 중인 실제 메모리의 비율로 표시됩니다. 기본적으로 실제 메모리 부하가 **90%**에 도달하면 가비지 수집은 페이징이 발생하지 않도록 더 적극적으로 가비지 수집을 꽉 채

우고 완전히 압축하려고 합니다. 메모리 부하가 90% 미만이면 GC는 전체 가비지 수집을 백그라운드에서 수행하여 일시 중지 시간을 줄이면서 전체 힙 크기는 크게 줄이지 않으려고 합니다. 많은 양의 메모리(80GB 이상)를 사용하는 머신에서 기본 부하 임계값은 90~97%입니다.

높은 메모리 부하 임계값은 `DOTNET_GCHighMemPercent` 환경 변수 또는 `System.GC.HighMemoryPercent` JSON 구성 설정으로 조정할 수 있습니다. 힙 크기를 제어하려면 임계값을 조정하는 것이 좋습니다. 예를 들어 메모리가 64GB인 컴퓨터의 주요 프로세스의 경우, 가용 메모리가 10%가 되면 GC가 반응하기 시작하는 것이 합당합니다. 그러나 작은 프로세스(예: 1GB 메모리만 사용하는 프로세스)의 경우 가용 메모리가 10% 미만일 때도 GC가 최적으로 실행됩니다. 이러한 작은 프로세스의 경우 임계값을 높게 설정하는 것이 좋습니다. 반면, 큰 프로세스의 힙 크기를 더 작게 유지하려는 경우(사용할 수 있는 실제 메모리가 많은 경우에도) 이 임계값을 낮추는 것은 GC가 더 빠르게 반응하여 힙을 압축하게 되는 효과적인 방법입니다.

> **참고 항목**
>
> 컨테이너에서 실행되는 프로세스의 경우 GC는 컨테이너 제한을 기준으로 실제 메모리를 고려합니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HighMemoryPercent`	*10진수 값*	.NET 5
환경 변수	`DOTNET_GCHighMemPercent`	*16진수 값*	.NET 6

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어 높은 메모리 임계값을 75%로 설정하려면 JSON 파일의 경우 값을 75로 지정하고, 환경 변수의 경우 값을 0x4B 또는 4B로 지정합니다.

VM 유지

- 삭제해야 할 세그먼트를 나중에 사용할 수 있도록 대기 목록에 둘지 아니면 해제하여 운영 체제(OS)로 돌려보낼지를 구성합니다.

- 기본값: 세그먼트를 해제하여 운영 체제로 돌려보냅니다. 이는 값을 `false`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.RetainVM`	`false` - OS로 해제 `true` - 대기 목록에 두기	.NET Core 1.0
MSBuild 속성	`RetainVMGarbageCollection`	`false` - OS로 해제 `true` - 대기 목록에 두기	.NET Core 1.0
환경 변수	`DOTNET_GCRetainVM`	`0` - OS로 해제 `1` - 대기 목록에 두기	.NET 6

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.RetainVM": true
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.RetainVM": true
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <RetainVMGarbageCollection>true</RetainVMGarbageCollection>
```

```
  </PropertyGroup>

</Project>
```

메모리 절약

- 더 빈번한 가비지 수집과 더 긴 일시 중지 시간 대신 메모리를 절약하도록 가비지 수집기를 구성합니다.
- 기본값은 0입니다. 이것은 변경되지 않음을 의미합니다.
- 기본값 0 외에 1과 9 사이의 값이 유효합니다. 값이 높을수록 더 많은 가비지 수집기가 메모리를 절약하여 힙을 작게 유지하려고 합니다.
- 값이 0이 아닐 경우 조각화가 너무 많으면 큰 개체 힙이 자동으로 압축됩니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.ConserveMemory`	`0` - `9`	.NET 6
환경 변수	`DOTNET_GCConserveMemory`	`0` - `9`	.NET 6
.NET Framework의 app.config	GCConserveMemory	`0` - `9`	.NET Framework 4.8

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

*app.config*파일 예제:

XML

```
<configuration>
  <runtime>
    <GCConserveMemory enabled="5"/>
  </runtime>
</configuration>
```

> **팁**
>
> 다른 숫자로 실험하여 가장 적합한 값을 확인합니다. 5에서 7 사이의 값으로 시작합니다.

서버 GC에 대한 리소스 사용량 관리

다음 설정은 서버 GC 스레드 수와 코어에 선호도를 지정하는 경우/방법에 영향을 줍니다. 워크스테이션 GC에는 영향을 주지 않습니다.

- 힙 수
- 선호도 지정
- 선호도 지정 마스크
- 선호도 지정 범위
- CPU 그룹
- 데이터

처음 3개 설정에 대한 자세한 내용은 워크스테이션과 서버 GC 블로그 항목 간의 중간 지점을 참조하세요.

힙 수

- 가비지 수집기가 생성하는 힙의 개수를 제한합니다.
- 서버 가비지 수집에만 적용됩니다.
- GC 프로세서 선호도가 사용하도록 설정된 경우(기본값), 힙 개수 설정은 `n` GC 힙/스레드의 선호도를 처음 `n`개의 프로세서로 지정합니다. (정확히 어떤 프로세서의 선호도를 지정하려는지 지정하려면 선호도 지정 마스크 또는 선호도 지정 범위 설정을 사용하세요.)
- GC 프로세서 선호도를 사용하지 않도록 설정하는 경우, 이 설정으로 GC 힙 개수가 제한됩니다.
- 자세한 내용은 GCHeapCount 설명을 참조하세요.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapCount`	*10진수 값*	.NET Core 3.0
환경 변수	`DOTNET_GCHeapCount`	*16진수 값*	.NET 6
.NET Framework의 app.config	GCHeapCount	*10진수 값*	.NET Framework 4.6.2

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

```
JSON
```

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapCount": 16
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.HeapCount": 16
   }
}
```

> **팁**
>
> *runtimeconfig.json*의 옵션을 설정할 때는 10진수 값을 지정합니다. 옵션을 환경 변수로 설정할 때는 16진수 값을 지정합니다. 예를 들어, 힙의 개수를 16으로 제한하려면 JSON 파일의 경우 값을 16으로 지정하고 환경 변수의 경우 값을 0x10 또는 10으로 지정합니다.

선호도 지정

- 가비지 수집 스레드가 프로세서를 *선호*하도록 지정할지 여부를 지정합니다. GC 스레드의 선호도를 지정한다는 것은 특정 CPU에서만 실행할 수 있음을 의미합니다. 각 GC 스레드에 대해 힙이 생성됩니다.
- 서버 가비지 수집에만 적용됩니다.
- 기본값: 가비지 수집 스레드가 프로세서를 선호하도록 지정합니다. 이는 값을 `false`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.NoAffinitize`	`false` - 선호 `true` - 선호 안 함	.NET Core 3.0
환경 변수	`DOTNET_GCNoAffinitize`	`0` - 선호 `1` - 선호 안 함	.NET 6
.NET Framework의 app.config	GCNoAffinitize	`false` - 선호	.NET Framework 4.6.2

	설정 이름	값	도입된 버전
		`true` - 선호 안 함	

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.NoAffinitize": true
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.NoAffinitize": true
   }
}
```

선호도 지정 마스크

- 가비지 수집기 스레드가 사용할 정확한 프로세서를 지정합니다.
- GC 프로세서 선호도를 사용하지 않도록 설정하면 설정이 무시됩니다.
- 서버 가비지 수집에만 적용됩니다.
- 값은 프로세스에서 사용할 수 있는 프로세서를 정의하는 비트 마스크입니다. 예를 들어, 10진수 값 1023(환경 변수를 사용하는 경우에는 16진수 값 0x3FF 또는 3FF)은 이진 표기법으로 0011 1111 1111입니다. 이는 처음 10개의 프로세서를 사용하도록 지정합니다. 다음 10개의 프로세서, 즉 프로세서 10~19를 지정하려면 10진수 값 1047552(또는 16진수 값 0xFFC00 또는 FFC00)을 지정합니다. 이는 이진수 값 1111 1111 1100 0000 0000과 같습니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapAffinitizeMask`	*10진수 값*	.NET Core 3.0
환경 변수	`DOTNET_GCHeapAffinitizeMask`	*16진수 값*	.NET 6
.NET Framework의 app.config	GCHeapAffinitizeMask	*10진수 값*	.NET Framework 4.6.2

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapAffinitizeMask": 1023
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.HeapAffinitizeMask": 1023
   }
}
```

선호도 지정 범위

- 가비지 수집기 스레드가 사용할 프로세서 목록을 지정합니다.
- 이 설정은 64개가 넘는 프로세서를 지정할 수 있다는 점을 제외하면 System.GC.HeapAffinitizeMask와 비슷합니다.
- Windows 운영 체제에서는 번호 또는 범위 앞에 해당하는 CPU 그룹(예: "0:1-10,0:12,1:50-52,1:7")을 붙입니다. 실제로 하나 이상의 CPU 그룹이 없는 경우 이 설정을 사용할 수 없습

니다. 선호도 지정 마스크 설정을 사용해야 합니다. 지정한 숫자는 해당 그룹 내에 있으므로 >=64는 성립할 수 없습니다.

- CPU 그룹 개념이 없는 Linux 운영 체제의 경우 이 설정과 선호도 지정 마스크 설정을 모두 사용하여 동일한 범위를 지정할 수 있습니다. 그룹 인덱스를 지정할 필요가 없으므로 "0:1-10" 대신 "1-10"을 지정합니다.
- GC 프로세서 선호도를 사용하지 않도록 설정하면 설정이 무시됩니다.
- 서버 가비지 수집에만 적용됩니다.
- 자세한 내용은 Maoni Stephens의 블로그에서 Making CPU configuration better for GC on machines with > 64 CPUs(CPU가 64개가 넘는 머신에서 GC를 위한 CPU 구성 개선하기) 게시물을 참조하세요.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.HeapAffinitizeRanges`	쉼표로 구분된 프로세서 번호 또는 프로세서 번호 범위 목록. Unix 예제: "1-10,12,50-52,70" Windows 예제: "0:1-10,0:12,1:50-52,1:7"	.NET Core 3.0
환경 변수	`DOTNET_GCHeapAffinitizeRanges`	쉼표로 구분된 프로세서 번호 또는 프로세서 번호 범위 목록. Unix 예제: "1-10,12,50-52,70" Windows 예제: "0:1-10,0:12,1:50-52,1:7"	.NET 6

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

예제

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapAffinitizeRanges": "0:1-10,0:12,1:50-52,1:7"
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.GC.HeapAffinitizeRanges": "0:1-10,0:12,1:50-52,1:7"
   }
}
```

CPU 그룹

- 가비지 수집기가 CPU 그룹을 사용하는지 여부는 구성합니다.

  64비트 Windows 컴퓨터에 여러 개의 CPU 그룹이 있는 경우, 다시 말해서 64개가 넘는 프로세서가 있는 경우, 이 요소를 사용하도록 설정하면 가비지 수집이 모든 CPU 그룹으로 확장됩니다. 가비지 수집기는 모든 코어를 사용하여 힙을 만들고 분산합니다.

  > **참고 항목**
  >
  > Windows에만 해당되는 개념입니다. 이전 Windows 버전에서 Windows는 프로세스를 하나의 CPU 그룹으로 제한했습니다. 따라서 GC는 이 설정을 사용하여 여러 CPU 그룹을 사용하도록 설정하지 않는 한 하나의 CPU 그룹만 사용했습니다. 이 OS 제한은 Windows 11 및 Server 2022에서 없어졌습니다. 또한, .NET 7부터는 GC를 Windows 11 또는 Server 2022에서 실행할 때 해당 GC가 기본적으로 모든 CPU 그룹을 사용합니다.

- 64비트 Windows 운영 체제의 서버 가비지 수집에만 적용됩니다.
- 기본값: GC가 CPU 그룹 간에 확장되지 않습니다. 이는 값을 `0`으로 설정하는 것과 같습니다.
- 자세한 내용은 Maoni Stephens의 블로그에서 Making CPU configuration better for GC on machines with > 64 CPUs(CPU가 64개가 넘는 머신에서 GC를 위한 CPU 구성 개선하기) 게시물을 참조하세요.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.CpuGroup`	`false` - 사용 안 함 `true` - 사용	.NET 5
환경 변수	`DOTNET_GCCpuGroup`	`0` - 사용 안 함 `1` - 사용	.NET 6

	설정 이름	값	도입된 버전
.NET Framework의 app.config	GCCpuGroup	`false` - 사용 안 함 `true` - 사용	

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

> ⓘ **참고 항목**
>
> 모든 CPU 그룹의 스레드 풀에서도 스레드를 분산하도록 CLR(공용 언어 런타임)을 구성하려면 Thread_UseAllCpuGroups 요소 옵션을 사용하도록 설정합니다. .NET Core 앱의 경우, `DOTNET_Thread_UseAllCpuGroups` 환경 변수의 값을 `1`로 설정하여 이 옵션을 사용하도록 설정할 수 있습니다.

애플리케이션 크기에 대한 동적 적응(DATAS)

데이터 사용 또는 사용 안 함

- DATAS를 사용하도록 가비지 수집기를 구성합니다. DATAS는 애플리케이션 메모리 요구 사항에 맞게 조정됩니다. 즉, 앱 힙 크기는 수명이 긴 데이터 크기에 거의 비례해야 합니다.
- .NET 9부터 기본적으로 사용하도록 설정됩니다.

테이블 확장

	설정 이름	값	도입된 버전
환경 변수	`DOTNET_GCDynamicAdaptationMode`	`1` - 사용 `0` - 사용 안 함	.NET 8
MSBuild 속성	`GarbageCollectionAdaptationMode`	`1` - 사용 `0` - 사용 안 함	.NET 8
runtimeconfig.json	`System.GC.DynamicAdaptationMode`	`1` - 사용 `0` - 사용 안 함	.NET 8

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

대상 TCP

DATAS는 처리량에 대한 메모리 비용 측정으로 TCP(처리량 비용 백분율)를 사용합니다. GC 일시 중지와 대기해야 하는 할당의 양을 모두 고려합니다. 일반적으로 GC 일시 중지가 지배하므로 "% GC의 일시 중지 시간"을 사용하여 이 비용을 근사화할 수 있습니다. 할당 대기 시간이 지배할 수 있는 두 가지 일시적인 경우가 있습니다.

- 프로세스가 시작되면 DATAS는 항상 하나의 힙으로 시작됩니다. 따라서 할당하는 스레드가 충분한 경우 대기가 발생합니다.
- 예를 들어 사용량이 많은 시간이 시작될 때 워크로드가 더 가볍고 무거워지면 해당 기간 동안 스레드 할당에 대기가 발생할 수 있습니다. 힙 수가 증가하기 전에 몇 가지 GC가 필요하기 때문입니다.

DATAS는 2% 기본 TCP로 사용하며 이 설정을 사용하여 조정할 수 있습니다. 백분율로 해석되는 정수입니다. 예를 들어 5는 대상 TCP가 5%의미합니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.DTargetTCP`	*10진수 값*	.NET 9
환경 변수	`DOTNET_GCDTargetTCP`	*16진수 값*	.NET 9

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

Gen0 최대 예산 설정

gen0 예산을 조정하는 것은 데이터에서 애플리케이션 크기에 맞게 조정하는 데 사용하는 주요 요소 중 하나입니다. DATAS는 애플리케이션 크기의 함수로 총 gen0 예산에 대한 BCD(DATAS를 통해 계산되는 예산)라는 상한 임계값을 정의합니다. 승수를 계산하는 수식은 다음과 같습니다.

$$f(\text{application\_size\_in\_MB}) = \frac{20 - \text{conserve\_memory}}{\sqrt{\text{application\_size\_in\_MB}}}$$

그런 다음 수식이 애플리케이션 크기에 MB로 적용되기 전에 최대값과 최소값으로 고정됩니다. 메모리 절약 설정을 지정하지 않으면 DATAS는 기본적으로 5를 사용합니다. 최대값과 최소값은 각각 10과 0.1로 기본값입니다.

예를 들어 애플리케이션 크기가 1GB이면 수식이 계산됩니다 `(20 - 5) / sqrt (1000) = 0.474`. 10에서 0.1 사이이므로 클램핑은 효과가 없습니다. 즉, 총 gen0 예산은 474MB인 1GB의% 47.4입

니다. 애플리케이션 크기가 1MB인 경우 수식은 15를 계산한 다음 10으로 조정됩니다. 즉, 허용되는 총 gen0 예산은 10MB입니다.

수식이 계산하는 내용을 조정하고 클램핑 값을 수정하기 위한 세 가지 설정이 제공됩니다.

- 계산에 적용 `f` 할 백분율입니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.DGen0GrowthPercent`	*10진수 값*	.NET 10
환경 변수	`DOTNET_GCDGen0GrowthPercent`	*16진수 값*	.NET 10

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

따라서 0.474를 계산하고 이 설정이 200이면 `f` 클램핑이 적용되기 전에 승수 값이 됩니다 `0.474 * 200% = 0.948` .

- 퍼밀의 최대 클램핑 값입니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.DGen0GrowthMaxFactor`	*10진수 값*	.NET 10
환경 변수	`DOTNET_GCDGen0GrowthMaxFactor`	*16진수 값*	.NET 10

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

이 값이 20000이면 최대 클램핑 값이 됩니다 `20000 * 0.001 = 20`.

- 퍼밀의 최소 클램핑 값입니다.

테이블 확장

	설정 이름	값	도입된 버전
runtimeconfig.json	`System.GC.DGen0GrowthMinFactor`	*10진수 값*	.NET 10

	설정 이름	값	도입된 버전
환경 변수	`DOTNET_GCDGen0GrowthMinFactor`	*16진수 값*	.NET 10

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

이 값이 200이면 최소 클램핑 값은 .입니다 `200 * 0.001 = 0.2`.

Last updated on 2025. 11. 22.

세계화를 위한 런타임 구성 옵션

고정 모드

- 문화권별 데이터 및 동작에 액세스하지 않고 .NET Core 앱이 세계화 고정 모드에서 실행되는지 여부를 결정합니다.
- 이 설정을 생략하면 앱이 문화권 데이터에 대한 액세스 권한으로 실행됩니다. 이는 값을 `false`으로 설정하는 것과 같습니다.
- 자세한 내용은 .NET Core 세계화 고정 모드를 참조하세요.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Globalization.Invariant`	`false` - 문화권 데이터에 대한 액세스 `true` - 고정 모드에서 실행
MSBuild 속성	`InvariantGlobalization`	`false` - 문화권 데이터에 대한 액세스 `true` - 고정 모드에서 실행
환경 변수	`DOTNET_SYSTEM_GLOBALIZATION_INVARIANT`	`0` - 문화권 데이터에 대한 액세스 `1` - 고정 모드에서 실행

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Globalization.Invariant": true
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Globalization.Invariant": true
```

```
    }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>

</Project>
```

연대 연도 범위

- 여러 연대를 지원하는 달력에 대한 범위 검사가 완화되는지 또는 연대의 날짜 범위를 오버플로하는 날짜가 throw되는지 여부를 결정합니다 ArgumentOutOfRangeException.
- 이 설정을 생략하면 범위 검사가 완화됩니다. 이는 값을 `false`으로 설정하는 것과 같습니다.
- 자세한 내용은 일정, 연대 및 날짜 범위: 완화된 범위 검사를 참조하세요.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`Switch.System.Globalization.EnforceJapaneseEraYearRanges`	`false` - 완화 범위 검사 `true` - 오버플로로 인해 예외가 발생합니다.
환경 변수	N/A	N/A

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

일본어 날짜 구문 분석

- 연도로 "1" 또는 "Gannen"을 포함하는 문자열이 성공적으로 구문 분석되는지 또는 "1"만 지원되는지 여부를 결정합니다.

- 이 설정을 생략하면 연도 구문 분석으로 "1" 또는 "Gannen"이 포함된 문자열이 성공적으로 구문 분석됩니다. 이는 값을 `false`으로 설정하는 것과 같습니다.
- 자세한 내용은 여러 연대가 있는 달력의 날짜 표시를 참조하세요.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`Switch.System.Globalization.EnforceLegacyJapaneseDateParsing`	`false` - "Gannen" 또는 "1"이 지원됩니다. `true` - "1"만 지원됩니다.
환경 변수	N/A	N/A

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

일본어 연도 형식

- 일본 달력 시대의 첫 해가 "Gannen"으로 형식이 지정되는지 또는 숫자로 지정되는지를 결정합니다.
- 이 설정을 생략하면 첫 해가 "Gannen"으로 형식이 지정됩니다. 이는 값을 `false`으로 설정하는 것과 같습니다.
- 자세한 내용은 여러 연대가 있는 달력의 날짜 표시를 참조하세요.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`Switch.System.Globalization.FormatJapaneseFirstYearAsANumber`	`false` - "Gannen" 형식 `true` - 숫자로 서식 지정
환경 변수	N/A	N/A

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

NLS

- .NET에서 WINDOWS 앱용 NLS(국가별 언어 지원) 또는 ICU(International Components for Unicode) 세계화 API를 사용할지 여부를 결정합니다. .NET 5 이상 버전은 Windows 10 2019년 5월 업데이트 이상 버전에서 기본적으로 ICU 세계화 API를 사용합니다.
- 이 설정을 생략하면 .NET은 기본적으로 ICU 세계화 API를 사용합니다. 이는 값을 `false`으로 설정하는 것과 같습니다.
- 자세한 내용은 Windows에서 ICU 라이브러리를 사용하는 세계화 API를 참조하세요.

테이블 확장

	설정 이름	가치들	도입
runtimeconfig.json	`System.Globalization.UseNls`	`false` - ICU 세계화 API 사용 `true` - NLS 세계화 API 사용	.NET 5
환경 변수	`DOTNET_SYSTEM_GLOBALIZATION_USENLS`	`false` - ICU 세계화 API 사용 `true` - NLS 세계화 API 사용	.NET 5

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

미리 정의된 문화권

- 세계화 고정 모드를 사용하는 경우 앱이 고정 문화권 이외의 문화권을 만들 수 있는지 여부를 구성합니다.
- 이 설정을 생략하면 .NET은 세계화 고정 모드에서 문화권 생성을 제한합니다. 이는 값을 `true`으로 설정하는 것과 같습니다.
- 자세한 내용은 세계화 고정 모드의 문화권 만들기 및 대/소문자 매핑을 참조하세요.

테이블 확장

	설정 이름	가치들	도입
runtimeconfig.json	`System.Globalization.PredefinedCulturesOnly`	`true` - 세계화 고정 모드에서는 고정 문화권을 제외한 모든 문화권의 생성을 허용하지 않습니다. `false` - 모든 문	.NET 6

	설정 이름	가치들	도입
		화권의 생성을 허용합니다.	
MSBuild 속성	`PredefinedCulturesOnly`	`true` - 세계화 고정 모드에서는 고정 문화권을 제외한 모든 문화권의 생성을 허용하지 않습니다. `false` - 모든 문화권의 생성을 허용합니다.	.NET 6
환경 변수	`DOTNET_SYSTEM_GLOBALIZATION_PREDEFINED_CULTURES_ONLY`	`true` - 세계화 고정 모드에서는 고정 문화권을 제외한 모든 문화권의 생성을 허용하지 않습니다. `false` - 모든 문화권의 생성을 허용합니다.	.NET 6

Last updated on 2025. 12. 03.

네트워킹에 대한 런타임 구성 옵션

HTTP/2 프로토콜

- HTTP/2 프로토콜에 대한 지원을 사용할 수 있는지 여부를 구성합니다.
- 이 설정을 생략하면 HTTP/2 프로토콜에 대한 지원이 활성화됩니다. 이는 값을 `true`으로 설정하는 것과 같습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Http.SocketsHttpHandler.Http2Support`	`false` - 사용 안 함 `true` - 사용
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2SUPPORT`	`0` - 사용 안 함 `1` - 사용

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

HTTP/3 프로토콜

- .NET 7부터 HTTP/3은 기본적으로 사용하도록 설정됩니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.SocketsHttpHandler.Http3Support`	`false` -비활성화 `true` - 사용
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP3SUPPORT`	`0` - 사용 안 함 `1` - 사용

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

HttpClient에서 SPN 만들기(.NET 6 이상)

- 헤더가 누락되고 대상이 기본 포트에서 실행되고 있지 않을 때 `Host` Kerberos 및 NTLM 인증에 대한 SPN(서비스 사용자 이름) 생성에 영향을 줍니다.
- .NET 6 이상 버전은 기본적으로 SPN에 포트를 포함하지 않습니다. 그러나 동작을 구성할 수 있습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Http.UsePortInSpn`	`true` - SPN에 포트 번호 포함(예: `HTTP/host:port` `false` - SPN에 포트를 포함하지 마세요. 예를 들면 다음과 같습니다. `HTTP/host`
환경 변수	`DOTNET_SYSTEM_NET_HTTP_USEPORTINSPN`	`1` - SPN에 포트 번호 포함(예: `HTTP/host:port` `0` - SPN에 포트를 포함하지 마세요. 예를 들면 다음과 같습니다. `HTTP/host`

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

HTTP/2 동적 창 크기 조정

- 흐름 제어를 위해 HTTP/2 동적 창 크기 조정 알고리즘을 사용하지 않도록 설정할지 여부를 구성합니다. 알고리즘은 기본적으로 사용하도록 설정됩니다.
- 이 값으로 `true`설정하면 동적 창 크기 조정 알고리즘이 비활성화됩니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.SocketsHttpHandler.Http2FlowControl.DisableDynamicWindowSizing`	`false` - 사용(기본값) `true` - 사용 안 함
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2FLOWCONTROL_DISABLEDYNAMICWINDOWSIZING`	`0` - 사용(기본값) `1` - 사용 안 함

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

HTTP/2 스트림 수신 창 크기

- HTTP/2 스트림 수신 창의 최대 크기를 구성합니다.
- 기본값은 16MB입니다. 65,535 미만의 값은 65,535로 고정됩니다. 하드 상한은 없지만 이 설정을 기본값 이상으로 늘리면 처리량이 높고 대기 시간이 긴 네트워크에서만 유용합니다.

테이블 확장

	설정 이름	가치들
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_FLOWCONTROL_MAXSTREAMWINDOWSIZE`	정수(기본값: 16MB, 최소값: 65,535)

HTTP/2 스트림 창 크기 조정 임계값

- HTTP/2 스트림 수신 창이 얼마나 적극적으로 증가하는지 제어하는 승수를 구성합니다. 값이 높을수록 창이 더 보수적으로 증가하여 최대 처리량이 줄어듭니다.
- 기본값은 1.0입니다. 0 미만의 값은 기본값으로 다시 설정됩니다. 하드 상한은 없지만 기본값보다 훨씬 높은 값은 요청당 처리량을 점진적으로 제한합니다.

> **참고 항목**
>
> 이 설정은 고급 진단 및 내부 튜닝을 위한 것입니다. 대부분의 개발자는 변경할 필요가 없습니다.

테이블 확장

	설정 이름	가치들
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_FLOWCONTROL_STREAMWINDOWSCALETHRESHOLDMULTIPLIER`	Float(기본값: 1.0; 최소: 0)

HTTP 활동 전파

분산 추적 작업 전파를 사용할 수 있는지 여부를 구성합니다 HttpClient. 사용하도록 설정하면 나가는 HTTP 요청은 OpenTelemetry와 같은 분산 추적 도구에 대한 추적 컨텍스트 헤더(예: `traceparent`)를 전파합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Http.EnableActivityPropagation`	`true` - 사용(기본값) `false` - 사용 안 함
환경 변수	`DOTNET_SYSTEM_NET_HTTP_ENABLEACTIVITYPROPAGATION`	`1` - 사용(기본값) `0` - 사용 안 함

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

요청 완료 시 보류 중인 연결 시간 제한

HTTP 요청 시작이 완료된 후 보류 중인 연결 시도를 완료하기 위한 시간 제한(밀리초)을 구성합니다. 요청이 완료된 후에도 연결이 설정되는 경우 이 시간 제한은 연결 시도를 중단하기 전에 대기하는 시간을 결정합니다.

- 기본값은 `5000` 5초입니다.
- 연결이 `-1` 완료될 때까지 무기한 대기하도록 설정합니다.
- 요청이 완료되면 즉시 보류 중인 연결을 취소하도록 `0` 설정합니다.
- 하드 상한은 없지만 매우 큰 값은 실용적이지 않습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.SocketsHttpHandler.PendingConnectionTimeoutOnRequestCompletion`	정수 (기본값: `5000`)
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_PENDINGCONNECTIONTIMEOUTONREQUESTCOMPLETION`	정수 (기본값: `5000`)

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

프록시 사전 인증

사용하도록 설정 SocketsHttpHandler 하면 프록시의 챌린지 응답을 기다리는 대신 첫 번째 요청에 프록시 인증 자격 증명을 `407` 사전에 보냅니 `Basic` 다. 이는 챌린지 응답을 보내지 `407` 않는 프록시에 유용합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Http.SocketsHttpHandler.ProxyPreAuthenticate`	`false` - 사용 안 함(기본값) `true` - 사용

	설정 이름	가치들
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_PROXYPREAUTHENTICATE`	`0` - 사용 안 함(기본값) `1` - 사용

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

서버당 최대 연결 수

단일 서버에 열리는 최대 동시 TCP 연결 수를 구성합니다 SocketsHttpHandler . 처리기는 보다 `1` 작은 값을 무시하고 기본값을 사용합니다.

- 기본값은 무제한(`int.MaxValue`)입니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.SocketsHttpHandler.MaxConnectionsPerServer`	정수(기본값: 무제한)
환경 변수	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_MAXCONNECTIONSPERSERVER`	정수(기본값: 무제한)

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

소켓 인라인 완성

에 디스패치 System.Threading.ThreadPool되는 대신 이벤트 스레드에서 소켓 연속을 실행할 수 있는지 여부를 구성합니다. 이 설정을 사용하도록 설정하면 일부 시나리오에서 성능이 향상될 수 있습니다. 그러나 비용이 많이 드는 작업이 필요 이상으로 I/O 스레드를 보유하는 경우 성능이 저하될 수 있습니다.

> **참고 항목**
>
> 이 설정을 사용하도록 설정하면 특정 시나리오의 성능에 도움이 되는지 테스트합니다.

테이블 확장

	설정 이름	가치들
환경 변수	`DOTNET_SYSTEM_NET_SOCKETS_INLINE_COMPLETIONS`	`0` - 사용 안 함(기본값) `1` - 사용

소켓 스레드 수

소켓 I/O에 사용되는 스레드 수를 구성합니다. 재정의되지 않으면 프로세서 수 및 아키텍처에 따라 값이 계산됩니다. 실제 값은 범위에 `[1, ProcessorCount]`있습니다. 이 범위를 벗어난 값은 거부되지 않지만 성능이 향상될 가능성은 낮습니다.

> **참고 항목**
>
> 이 설정은 극단적인 부하 시나리오를 위한 것입니다. 대부분의 개발자는 변경할 필요가 없습니다.

테이블 확장

	설정 이름	가치들
환경 변수	`DOTNET_SYSTEM_NET_SOCKETS_THREAD_COUNT`	정수

IPv6

IPv6(인터넷 프로토콜 버전 6)을 사용하지 않도록 설정할지 여부를 구성합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.DisableIPv6`	`false` - 사용(기본값) `true` - 사용 안 함
환경 변수	`DOTNET_SYSTEM_NET_DISABLEIPV6`	`0` - 사용(기본값) `1` - 사용 안 함

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

TLS 세션 재개

TLS 세션 다시 시작에서 TLS 세션 재개 SslStream를 사용하지 않도록 설정할지 여부를 제어합니다. 세션 재개를 사용하면 TLS 다시 연결에서 이전에 협상한 세션 매개 변수를 다시 사용하여 전체 핸드셰이크를 건너뛰어 대기 시간을 줄일 수 있습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Security.DisableTlsResume`	`false` - 사용(기본값) `true` - 사용 안 함

	설정 이름	가치들
환경 변수	`DOTNET_SYSTEM_NET_SECURITY_DISABLETLSRESUME`	`0` - 사용(기본값) `1` - 사용 안 함

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

서버 AIA 다운로드

사용하도록 설정하면 TLS 클라이언트는 서버 인증서의 AIA(기관 정보 액세스) 확장 URL에서 중간 인증서를 자동으로 다운로드합니다. 이렇게 하면 서버가 전체 체인을 보내지 않는 경우에도 클라이언트가 전체 인증서 체인을 빌드할 수 있습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Security.EnableServerAiaDownloads`	`false` - 사용 안 함(기본값) `true` - 사용
환경 변수	`DOTNET_SYSTEM_NET_SECURITY_ENABLESERVERAIADOWNLOADS`	`0` - 사용 안 함(기본값) `1` - 사용

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

QUIC 구성 캐싱

MsQuic 구성 개체의 캐싱을 사용하지 않도록 설정합니다. 사용하도록 설정하면(기본값) 시스템은 연결 간에 구성 개체를 캐시하고 다시 사용하므로 동일한 매개 변수를 사용하는 반복된 연결에 대한 TLS 및 QUIC 설정의 오버헤드가 줄어듭니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Quic.DisableConfigurationCache`	`false` - 캐싱 사용(기본값) `true` - 캐싱 사용 안 함
환경 변수	`DOTNET_SYSTEM_NET_QUIC_DISABLE_CONFIGURATION_CACHE`	`0` - 캐싱 사용(기본값) `1` - 캐싱 사용 안 함

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어

msBuild 속성을 참조하세요.

앱 로컬 MsQuic(Windows)

이 기능을 사용하도록 설정하면 QUIC 구현은 .NET 어셈블리와 함께 번들로 제공되는 시스템 제공 라이브러리 대신 애플리케이션 디렉터리의 MsQuic 라이브러리를 사용합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.Quic.AppLocalMsQuic`	`false` - 시스템 MsQuic 사용(기본값) `true` - 앱 로컬 MsQuic 사용

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

HttpListener 커널 응답 버퍼링(Windows)

사용하도록 설정 HttpListener 하면 HTTP.sys통해 커널의 응답 데이터를 버퍼링합니다. 커널 버퍼링은 한 번에 최대 하나의 미해결 쓰기로 동기 I/O 또는 비동기 I/O를 사용하는 애플리케이션의 대기 시간이 긴 연결에 대한 처리량을 크게 향상시킬 수 있습니다. 동시 미해결 쓰기가 여러 개인 애플리케이션에는 이 설정을 사용하도록 설정하지 마세요.

> **참고 항목**
>
> 커널 응답 버퍼링을 사용하도록 설정하면 HTTP.sysCPU 및 메모리 사용량이 높아질 수 있습니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Net.HttpListener.EnableKernelResponseBuffering`	`false` - 사용 안 함(기본값) `true` - 사용

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

참고: 작성자가 AI의 도움을 받아 이 문서를 만들었습니다. 자세한 정보

Last updated on 2026. 03. 18.

스레딩에 대한 런타임 구성 옵션

이 문서에서는 .NET에서 스레딩을 구성하는 데 사용할 수 있는 설정을 자세히 설명합니다.

Windows에서 모든 CPU 그룹 사용

- CPU 그룹이 여러 개 있는 컴퓨터에서 이 설정은 스레드 풀과 같은 구성 요소가 모든 CPU 그룹을 사용하는지 아니면 프로세스의 기본 CPU 그룹만 사용하는지 여부를 구성합니다. 이 설정은 반환되는 항목 Environment.ProcessorCount 에도 영향을 줍니다.
- 이 설정을 사용하면 모든 CPU 그룹이 사용되고 스레드도 기본적으로 CPU 그룹에 자동으로 분산됩니다.
- 이 설정은 기본적으로 Windows 11 이상 버전에서 사용하도록 설정되며 Windows 10 및 이전 버전에서는 기본적으로 사용하지 않도록 설정됩니다. 이 설정을 사용하도록 설정하면 GC도 모든 CPU 그룹을 사용하도록 구성해야 합니다. 자세한 내용은 GC CPU 그룹을 참조하세요.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	N/A	N/A
환경 변수	`DOTNET_Thread_UseAllCpuGroups`	`0` - 사용 안 함 `1` - 사용

Windows의 CPU 그룹에 스레드 할당

- 여러 CPU 그룹이 있고 모든 CPU 그룹이 사용되는 컴퓨터에서 이 설정은 스레드가 CPU 그룹에 자동으로 분산되는지 여부를 구성합니다.
- 이 설정을 사용하도록 설정하면 새 CPU 그룹을 활용하기 전에 이미 사용 중인 CPU 그룹을 완전히 채웁니다.
- 이 설정은 기본적으로 사용하도록 설정됩니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	N/A	N/A
환경 변수	`DOTNET_Thread_AssignCpuGroups`	`0` - 사용 안 함 `1` - 사용

최소 스레드

- 작업자 스레드 풀의 최소 스레드 수를 지정합니다.
- 메서드에 ThreadPool.SetMinThreads 해당합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Threading.ThreadPool.MinThreads`	최소 스레드 수를 나타내는 정수입니다.
MSBuild 속성	`ThreadPoolMinThreads`	최소 스레드 수를 나타내는 정수입니다.
환경 변수	N/A	N/A

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.MinThreads": 4
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.MinThreads": 4
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  </PropertyGroup>

</Project>
```

최대 스레드 수

- 작업자 스레드 풀의 최대 스레드 수를 지정합니다.
- 메서드에 ThreadPool.SetMaxThreads 해당합니다.

테이블 확장

	설정 이름	가치들
runtimeconfig.json	`System.Threading.ThreadPool.MaxThreads`	최대 스레드 수를 나타내는 정수입니다.
MSBuild 속성	`ThreadPoolMaxThreads`	최대 스레드 수를 나타내는 정수입니다.
환경 변수	N/A	N/A

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.MaxThreads": 20
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.MaxThreads": 20
   }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMaxThreads>20</ThreadPoolMaxThreads>
  </PropertyGroup>

</Project>
```

Windows 스레드 풀

- Windows 프로젝트의 경우 스레드 풀 스레드 관리가 Windows 스레드 풀에 위임되는지 여부를 구성합니다.

- 이 설정을 생략하거나 플랫폼이 Windows가 아닌 경우 .NET 스레드 풀이 대신 사용됩니다.
- Windows에서 Native AOT를 사용하여 게시된 애플리케이션만 기본적으로 Windows 스레드 풀을 사용하며, 이 경우 구성 설정을 사용하지 않도록 설정하여 .NET 스레드 풀을 사용하도록 선택할 수 있습니다.
- Windows 스레드 풀은 최소 스레드 수가 높은 값으로 구성된 경우 또는 Windows 스레드 풀이 이미 앱에서 많이 사용되고 있는 경우와 같은 경우에 더 나은 성능을 발휘할 수 있습니다. 더 큰 컴퓨터에서 I/O 처리가 많은 경우와 같이 .NET 스레드 풀이 더 잘 수행되는 경우도 있을 수 있습니다. 이 구성 설정을 변경할 때 성능 메트릭을 확인하는 것이 좋습니다.
- 일부 API는 Windows 스레드 풀(예: ThreadPool.SetMinThreads, ThreadPool.SetMaxThreads및 ThreadPool.BindHandle(SafeHandle))을 사용할 때 지원되지 않습니다. 최소 및 최대 스레드에 대한 스레드 풀 구성 설정도 효과적이지 않습니다. 대안 ThreadPool.BindHandle(SafeHandle) 은 클래스입니다 ThreadPoolBoundHandle .

테이블 확장

	설정 이름	가치들	도입된 버전
runtimeconfig.json	`System.Threading.ThreadPool.UseWindowsThreadPool`	`true` - 사용 `false` - 사용 안 함	.NET 8
MSBuild 속성	`UseWindowsThreadPool`	`true` - 사용 `false` - 사용 안 함	.NET 8
환경 변수	`DOTNET_ThreadPool_UseWindowsThreadPool`	`1` - 사용 `0` - 사용 안 함	.NET 8

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.UseWindowsThreadPool": true
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.UseWindowsThreadPool": true
   }
}
```

프로젝트 파일:

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <UseWindowsThreadPool>true</UseWindowsThreadPool>
  </PropertyGroup>

</Project>
```

작업 항목 차단에 대한 응답으로 스레드 주입

경우에 따라 스레드 풀은 스레드를 차단하는 작업 항목을 검색합니다. 보정하기 위해 더 많은 스레드를 삽입합니다. .NET 6 이상에서는 다음 런타임 구성 설정을 사용하여 작업 항목 차단에 대한 응답으로 스레드 삽입을 구성할 수 있습니다. 현재 이러한 설정은 일반적인 비동기 동기화 사례와 같이 다른 작업이 완료되기를 기다리는 작업 항목에만 적용됩니다.

테이블 확장

***runtimeconfig.json* 설정 이름**	**Description**	**도입된 버전**
`System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor`	기준 스레드 수에 `MinThreads` 도달하면 이 값(프로세서 수를 곱한 후)은 지연 없이 만들 수 있는 추가 스레드 수를 지정합니다.	.NET 6
`System.Threading.ThreadPool.Blocking.ThreadsPerDelayStep_ProcCountFactor`	기준 스레드 수에 `ThreadsToAddWithoutDelay` 도달하면 이 값(프로세서 수를 곱한 후)은 각 새 스레드를 만들기 전에 지연에 추가 `DelayStepMs` 되는 스레드 수를 지정합니다.	.NET 6
`System.Threading.ThreadPool.Blocking.DelayStepMs`	기준 스레드 수에 `ThreadsToAddWithoutDelay` 도달한 후 이 값은 스레드당 `ThreadsPerDelayStep` 추가할 추가 지연 시간을 지정합니다. 이 지연 시간은 각 새 스레드를 만들기 전에 적용됩니다.	.NET 6

runtimeconfig.json 설정 이름	Description	도입된 버전
`System.Threading.ThreadPool.Blocking.MaxDelayMs`	기준 스레드 수에 `ThreadsToAddWithoutDelay` 도달한 후 이 값은 각 새 스레드를 만들기 전에 사용할 최대 지연 시간을 지정합니다.	.NET 6
`System.Threading.ThreadPool.Blocking.IgnoreMemoryUsage`	기본적으로 차단에 대한 응답으로 스레드 주입 속도는 사용 가능한 실제 메모리가 충분한지 여부를 결정하는 추론에 의해 제한됩니다. 경우에 따라 메모리가 낮은 상황에서도 스레드를 더 빨리 주입하는 것이 좋습니다. 이 스위치를 끄면 메모리 사용 추론을 사용하지 않도록 설정할 수 있습니다.	.NET 7

구성 설정이 적용되는 방식

- 기반 스레드 수에 `MinThreads` 도달한 후에는 지연 없이 최대 `ThreadsToAddWithoutDelay` 추가 스레드를 만들 수 있습니다.
- 그런 다음, 각 추가 스레드가 생성되기 전에 지연이 발생합니다 `DelayStepMs`.
- 지연으로 추가되는 모든 `ThreadsPerDelayStep` 스레드에 대해 지연에 추가 `DelayStepMs` 추가됩니다.
- 지연이 초과 `MaxDelayMs`되지 않을 수 있습니다.
- 지연은 스레드를 만들기 전에만 유도됩니다. 스레드를 이미 사용할 수 있는 경우 작업 항목 차단을 보정하기 위해 지연 없이 해제됩니다.
- 실제 메모리 사용량 및 제한도 사용되며 임계값을 초과하면 시스템이 느린 스레드 주입으로 전환됩니다.

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
```

```
        "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
   }
}
```

`AutoreleasePool` 관리되는 스레드의 경우

이 옵션은 지원되는 macOS 플랫폼에서 실행할 때 각 관리되는 스레드가 암시적 NSAutoreleasePool 을 수신하는지 여부를 구성합니다.

테이블 확장

	설정 이름	가치들	도입된 버전
runtimeconfig.json	`System.Threading.Thread.EnableAutoreleasePool`	`true` 또는 `false`	.NET 6
MSBuild 속성	`AutoreleasePoolSupport`	`true` 또는 `false`	.NET 6
환경 변수	N/A	N/A	N/A

예시

runtimeconfig.json 파일:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.Thread.EnableAutoreleasePool": true
      }
   }
}
```

runtimeconfig.template.json 파일:

JSON

```
{
   "configProperties": {
```

```
      "System.Threading.Thread.EnableAutoreleasePool": true
    }
}
```

프로젝트 파일:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <AutoreleasePoolSupport>true</AutoreleasePoolSupport>
  </PropertyGroup>

</Project>
```

Last updated on 2025. 11. 22.

WPF에 대한 런타임 구성 옵션

이 문서에서는 .NET에서 WPF(Windows Presentation Framework)를 구성하는 데 사용할 수 있는 설정을 자세히 설명합니다.

RDP의 하드웨어 가속

- RDP(원격 데스크톱 프로토콜)를 통해 액세스되는 WPF 앱에 하드웨어 가속을 사용할지 여부를 구성합니다. 하드웨어 가속은 애플리케이션에서 그래픽 및 시각 효과의 렌더링 속도를 높이기 위해 컴퓨터의 GPU(그래픽 처리 장치)를 사용하는 것을 의미합니다. 이로 인해 성능이 향상되고 응답성이 뛰어난 그래픽이 향상될 수 있습니다.
- 이 설정을 생략하면 그래픽이 소프트웨어에서 렌더링됩니다. 이는 값을 `false`으로 설정하는 것과 같습니다.

테이블 확장

설정 유형	설정 이름	가치들	도입된 버전
runtimeconfig.json	`Switch.System.Windows.Media.EnableHardwareAccelerationInRdp`	`true` - 사용 `false` - 사용 안 함	.NET 8
환경 변수	N/A		

이 구성 설정에는 특정 MSBuild 속성이 없습니다. 그러나 `RuntimeHostConfigurationOption` MSBuild 항목을 대신 추가할 수 있습니다. *runtimeconfig.json* 설정 이름을 `Include` 특성 값으로 사용합니다. 예를 들어 msBuild 속성을 참조하세요.

Last updated on 2025. 11. 22.

앱 시작 실패 문제 해결

2025. 08. 22.

이 문서에서는 애플리케이션 시작 실패에 대한 몇 가지 일반적인 이유와 가능한 솔루션에 대해 설명합니다. 머신에 .NET 설치 를 사용하는 프레임워크 종속 애플리케이션과 관련이 있습니다.

필요한 .NET 버전을 이미 알고 있는 경우 .NET 다운로드에서 다운로드할 수 있습니다.

.NET 설치를 찾을 수 없음

.NET 설치를 찾을 수 없으면 애플리케이션이 다음과 유사한 메시지와 함께 시작되지 않습니다.

콘솔

```
You must install .NET to run this application.

App: C:\repos\myapp\myapp.exe
Architecture: x64
Host version: 7.0.0
.NET location: Not found
```

오류 메시지에는 .NET을 다운로드하는 링크가 포함되어 있습니다. 해당 링크를 따라 적절한 다운로드 페이지로 가져올 수 있습니다. .NET 다운로드에서 .NET 버전(지정됨 `Host version`)을 선택할 수도 있습니다.

필요한 .NET 버전의 다운로드 페이지에서 오류 메시지에 나열된 아키텍처와 일치하는 **.NET 런타임** 다운로드를 찾습니다. 그런 다음 **설치 관리자**를 다운로드하고 실행하여 설치할 수 있습니다.

또는 필요한 .NET 버전의 다운로드 페이지에서 지정된 아키텍처에 대한 **이진 파일을** 다운로드할 수 있습니다.

필수 프레임워크를 찾을 수 없음

필요한 프레임워크 또는 호환되는 버전을 찾을 수 없는 경우 애플리케이션은 다음과 유사한 메시지와 함께 시작되지 않습니다.

콘솔

```
You must install or update .NET to run this application.

App: C:\repos\myapp\myapp.exe
Architecture: x64
Framework: 'Microsoft.NETCore.App', version '5.0.15' (x64)
```

```
  .NET location: C:\Program Files\dotnet\

  The following frameworks were found:
    6.0.2 at [c:\Program Files\dotnet\shared\Microsoft.NETCore.App]
```

오류는 누락된 프레임워크의 이름, 버전 및 아키텍처와 설치될 것으로 예상되는 위치를 나타냅니다. 애플리케이션을 실행하려면 지정된 ".NET 위치"에 호환되는 런타임을 설치 할 수 있습니다. 애플리케이션이 설치한 버전보다 낮은 버전을 대상으로 하고 더 높은 버전에서 실행하려는 경우 애플리케이션에 대한 롤 포워드 동작을 구성할 수도 있습니다.

호환되는 런타임 설치

오류 메시지에는 누락된 프레임워크를 다운로드하는 링크가 포함되어 있습니다. 이 링크를 따라 적절한 다운로드 페이지로 가져올 수 있습니다.

또는 .NET 다운로드 페이지에서 런타임을 다운로드할 수 있습니다 . 여러 .NET 런타임 다운로드가 있습니다.

다음 표에서는 각 런타임에 포함된 프레임워크를 보여 줍니다.

테이블 확장

런타임 다운로드	포함된 프레임워크
ASP.NET Core 런타임	Microsoft.NETCore.App Microsoft.AspNetCore.App
.NET 데스크톱 런타임	Microsoft.NETCore.App Microsoft.WindowsDesktop.App
.NET 런타임	Microsoft.NETCore.App

누락된 프레임워크가 포함된 런타임 다운로드를 선택한 다음 설치합니다.

필요한 .NET 버전의 다운로드 페이지에서 오류 메시지에 나열된 아키텍처와 일치하는 런타임 다운로드를 찾습니다. **설치 관리자**를 다운로드하려고 할 수 있습니다.

또는 필요한 .NET 버전의 다운로드 페이지에서 지정된 아키텍처에 대한 **이진 파일을** 다운로드할 수 있습니다.

대부분의 경우 시작에 실패한 애플리케이션이 이러한 설치를 사용하는 경우 오류 메시지의 ".NET 위치"는 다음을 가리킵니다.

`%ProgramFiles%\dotnet`

기타 옵션

고려해야 할 다른 설치 및 해결 방법이 있습니다.

dotnet-install 스크립트 실행

운영 체제에 대한 dotnet-install 스크립트 를 다운로드합니다. 오류 메시지의 정보에 따라 옵션을 사용하여 스크립트를 실행합니다. dotnet-install 스크립트 참조 페이지에는 사용 가능한 모든 옵션이 표시됩니다.

PowerShell을 시작하고 다음을 실행합니다.

PowerShell

```
dotnet-install.ps1 -Architecture <architecture> -InstallDir <directory> -Runtime
<runtime> -Version <version>
```

예를 들어 이전 섹션의 오류 메시지는 다음과 같습니다.

PowerShell

```
dotnet-install.ps1 -Architecture x64 -InstallDir "C:\Program Files\dotnet\" -
Runtime dotnet -Version 5.0.15
```

실행 중인 스크립트가 비활성화되었다는 오류가 발생하는 경우 스크립트를 실행할 수 있도록 실행 정책을 설정해야 할 수 있습니다.

PowerShell

```
Set-ExecutionPolicy Bypass -Scope Process
```

스크립트를 사용하여 설치하는 방법에 대한 자세한 내용은 PowerShell 자동화를 사용하여 설치를 참조하세요.

이진 파일 다운로드

다운로드 페이지에서 .NET의 이진 보관 파일을 다운로드할 수 있습니다. 런타임 다운로드의 **이진** 열에서 필요한 아키텍처와 일치하는 이진 릴리스를 다운로드합니다. 다운로드한 보관 파일을 오류 메시지에 지정된 ".NET 위치"로 추출합니다.

수동 설치에 대한 자세한 내용은 Windows에서 .NET 설치를 참조하세요.

롤 포워드 동작 구성

필요한 프레임워크의 상위 버전이 이미 설치된 경우 롤 포워드 동작을 구성하여 더 높은 버전에서 애플리케이션을 실행할 수 있습니다.

애플리케이션을 실행할 때 명령줄 옵션을 지정--roll-forward하거나 환경 변수를 DOTNET_ROLL_FORWARD 설정할 수 있습니다. 기본적으로 애플리케이션에는 애플리케이션이 대상으로 하는 것과 동일한 주 버전과 일치하지만 더 높은 부 버전 또는 패치 버전을 사용할 수 있는 프레임워크가 필요합니다. 그러나 애플리케이션 개발자는 다른 동작을 지정했을 수 있습니다. 자세한 내용은 프레임워크 종속 앱 롤 포워드를 참조하세요.

> **참고**
>
> 이 옵션을 사용하면 애플리케이션이 디자인된 프레임워크 버전과 다른 프레임워크 버전에서 실행될 수 있으므로 프레임워크 버전 간의 변경으로 인해 의도하지 않은 동작이 발생할 수 있습니다.

파괴적 변경

.NET 7 이상에서는 다중 수준 조회를 사용하지 않도록 설정

Windows에서 .NET 7 이전에는 애플리케이션이 여러 설치 위치에서 프레임워크를 검색할 수 있습니다.

1. 다음을 기준으로 하는 하위 디렉터리:
   - `dotnet` 를 통해 `dotnet` 애플리케이션을 실행할 때 실행 가능
   - `DOTNET_ROOT` 실행 파일을 통해 애플리케이션을 실행할 때 환경 변수(`apphost` 설정된 경우)입니다.
2. 에 전역적으로 등록된 설치 위치(설정된 경우)입니다 `HKLM\SOFTWARE\dotnet\Setup\InstalledVersions\<arch>\InstallLocation`.
3. 기본 설치 위치 `%ProgramFiles%\dotnet` (또는 `%ProgramFiles(x86)%\dotnet` 64비트 Windows의 32비트 프로세스)입니다.

이 다중 수준 조회 동작은 기본적으로 사용하도록 설정되었지만 환경 변수 `DOTNET_MULTILEVEL_LOOKUP=0`를 설정하여 사용하지 않도록 설정할 수 있습니다.

.NET 7 이상을 대상으로 하는 애플리케이션의 경우 다단계 조회가 완전히 비활성화되고 .NET 설치가 발견된 첫 번째 위치인 한 위치만 검색됩니다. 애플리케이션을 실행하는 `dotnet`경우 프레

임워크는 에 상대적 `dotnet`인 하위 디렉터리에서만 검색됩니다. 애플리케이션이 실행 파일 () `apphost`을 통해 실행되는 경우 프레임워크는 .NET이 있는 이전에 나열된 위치 중 첫 번째 위치에서만 검색됩니다.

자세한 내용은 다단계 조회를 사용하지 않도록 설정됨을 참조하세요.

참고하십시오

- .NET 설치
- .NET 설치 위치
- 설치된 .NET 버전 확인
- 프레임워크 종속 애플리케이션