Документация по C#

Узнайте, как писать любое приложение с помощью языка программирования C#.

Выбор пути

НАЧАЛО РАБОТЫ

Новые возможности программирования? Начните с уроков для начинающих

Руководство по браузеру: Hello world

Серия видео для начинающих на C#

ОБЗОР

Опытный разработчик? Обзор C#

Изучение C# для разработчиков Java

Изучение C# для разработчиков JavaScript

Изучение C# для разработчиков на Python.

ОБУЧЕНИЕ

Базовая сертификация C#

Что можно создать с помощью C#

Основы

ОБЗОР

Обзор C#

Внутри программы C#

Серия видеороликов о функциональных возможностях C#

Стратегия языка C#

КОНЦЕПЦИЯ

Система типов

Объектно-ориентированное программирование

Сопоставление шаблонов

Исключения

Стиль программирования

УЧЕБНИК

Создание приложений на основе файлов

Введение в классы

Объектно-ориентированный C#

Преобразование типов

Сопоставление шаблонов

Использование LINQ для запроса данных

Новые возможности

НОВЫЕ ВОЗМОЖНОСТИ

Новые возможности C# 14

Новые возможности C# 13

Новые возможности C# 12

УЧЕБНИК

Изучение типов записей

Изучение инструкций верхнего уровня

ССЫЛКА

Критические изменения в компиляторе C#

Совместимость версий

Основные понятия

ОБЗОР

Что можно создать с помощью C#

Стратегия языка C#

Основные понятия программирования

КОНЦЕПЦИЯ

Интегрированный язык запросов (LINQ)

Асинхронное программирование

ОБУЧЕНИЕ

Изучение C# для разработчиков Java

Изучение C# для разработчиков JavaScript

Изучение C# для разработчиков на Python.

Справочник по языку C#

ССЫЛКА

Справочник по языку

Ключевые слова

Операторы и выражения

Токены

Оставайтесь на связи

ССЫЛКА

C# на Q&A

Сообщество разработчиков .NET

C# в Stack Overflow

C# в Дискорде

YouTube

Краткий обзор языка C#

В этой статье представлен общий обзор языка C#. Независимо от того, пишете ли вы первую программу или вы опытный разработчик, изучающий C#, вы найдете основные понятия и функции здесь.

> **Совет**
>
> **Новые возможности программирования?** Просмотрите эту статью, чтобы понять общую картину, а затем начните учебные пособия для начинающих по написанию кода.
>
> **Вы имеете опыт работы с Java, JavaScript или Python?** Ознакомьтесь с советами для разработчиков Java, JavaScript или Python вместе с этой статьей.
>
> **Опытный разработчик, но новичок в C#?** В этой статье описывается, что делает C# отличительным. Узнайте , что можно создать с помощью C# , чтобы найти рабочую нагрузку, которая соответствует вашим целям.

Язык C# является самым популярным языком для платформы .NET, бесплатной кроссплатформенной среды разработки открытый код. Программы C# могут выполняться на различных устройствах, от устройств Интернета вещей (IoT) до облака и между ними. Вы можете создавать приложения для телефонных, настольных и ноутбуков и серверов. Дополнительные сведения о типах приложений см. в статье " Что можно создать с помощью C# ".

C# — это кроссплатформенный язык общего назначения, который делает разработчиков продуктивным при написании высокопроизводительного кода. C# имеет широкую поддержку в экосистеме и в различных нагрузках. Основанный на объектно-ориентированных принципах, он включает множество функций из других парадигм, включая функциональное программирование. Низкоуровневые функции поддерживают сценарии высокой эффективности без написания небезопасного кода. Большинство сред выполнения и библиотек написаны на C#, и достижения в C# часто используются для всех разработчиков .NET.

C# находится в семействе языков C. Синтаксис C# знаком, если вы использовали C, C++, JavaScript, TypeScript или Java. Как и C++, запятая (`;`) определяет конец инструкций. Идентификаторы C# чувствительны к регистру. C# использует фигурные скобки так же, как `{` и `}`, а также операторы управления, такие как `if`, `else` и `switch`, и циклические конструкции, такие как `for` и `while`. C# также имеет инструкцию `foreach` для любого типа коллекции.

Hello World

Программа Hello, World традиционно представляет язык программирования. Ниже приведена программа в C#:

C#

```
// This line prints "Hello, World"
Console.WriteLine("Hello, World");
```

Строка, начатая с `//`, это *однострочный комментарий*. Одиночные строковые комментарии C# начинаются с `//` и продолжаются до конца текущей строки. C# также поддерживает *многострочные комментарии*. Многострочные комментарии начинаются с `/*` и заканчиваются `*/`. Метод `WriteLine` класса `Console`, находящегося в пространстве имен `System`, создает выходные данные программы. Библиотеки стандартных классов предоставляют этот класс, и каждая программа C# автоматически ссылается на эти библиотеки по умолчанию. Для другой формы программы необходимо объявить содержащий класс и метод для точки входа программы. Компилятор синтезирует эти элементы при использовании инструкций верхнего уровня.

Этот альтернативный формат по-прежнему действителен и содержит многие основные понятия во всех программах C#. Многие существующие примеры C# используют следующий эквивалентный формат:

C#

```
using System;
namespace TourOfCsharp;

class Program
{
    static void Main()
    {
        // This line prints "Hello, World"
        Console.WriteLine("Hello, World");
    }
}
```

Предыдущая программа Hello, World начинается с `using` директивы, которая ссылается на `System` пространство имен. Пространства имен позволяют иерархически упорядочивать программы и библиотеки C#. Пространства имен содержат типы и другие пространства имен. Например, `System` пространство имен содержит множество типов, таких как `Console` класс, на который ссылается программа, и многие другие пространства имен, например `IO` и `Collections`. Директива `using`, которая ссылается на пространство имен,

позволяет использовать типы из этого пространства имен без указания полного имени. Благодаря директиве `using` в коде программы можно использовать сокращенное имя `Console.WriteLine` вместо полного варианта `System.Console.WriteLine`. В предыдущем примере это пространство имен было неявно включено.

> ⓘ **Примечание.**
>
> Предыдущая программа объявляет `Program` класс с одним членом: методом с именем `Main`. Многие существующие примеры и учебники C# используют этот формат. По соглашению, если операторы верхнего уровня отсутствуют, статический метод с именем `Main` служит точкой **входа** программы C#. Оба формата компилируются в один код. Вам не нужно использовать этот формат— приложения на основе файлов и инструкции верхнего уровня проще приступить к работе.

Приложения на основе файлов

C# — это *скомпилированный* язык. В большинстве программ C# используется dotnet build команда для компиляции группы исходных файлов в двоичный пакет. Затем вы используете dotnet run команду для запуска программы. (Этот процесс можно упростить, так как `dotnet run` компилирует программу перед его выполнением при необходимости.) Эти средства поддерживают широкий язык параметров конфигурации и коммутаторов командной строки. Интерфейс `dotnet` командной строки (CLI), включенный в пакет SDK для .NET, предоставляет множество средств для создания и изменения файлов C#.

Начиная с C# 14 и .NET 10, можно создавать *приложения на основе файлов*, упрощающие создание и запуск программ C#. Вы используете `dotnet run` команду для запуска программы, содержащейся в одном `*.cs` файле. Например, если следующий код хранится в файле с именем `hello-world.cs`, его можно запустить, введя `dotnet run hello-world.cs` следующую команду:

C#

```
#!/usr/bin/env dotnet

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

Первая строка программы содержит `#!` последовательность (shebang) для оболочк unix. Расположение интерфейса командной `dotnet` строки может отличаться от разных дистрибутивов. В любой системе UNIX, если задать разрешение *на выполнение* (`+x`) для

такого файла C#, содержащего директиву shebang, можно запустить файл C# непосредственно из командной строки:

```Bash
./hello-world.cs
```

Источник для этих программ должен быть одним файлом, но в противном случае весь синтаксис C# является допустимым. Приложения на основе файлов можно использовать для небольших служебных программ командной строки, прототипов или других экспериментов.

Знакомые функции C#

C# подходит для начинающих, но предлагает расширенные возможности для опытных разработчиков написания специализированных приложений. Вы можете быстро стать продуктивными. Вы можете узнать больше специализированных методов, так как они нужны для ваших приложений.

Приложения C# получают преимущества от автоматического управления памятью среды выполнения. Приложения C# также используют обширные библиотеки среды выполнения, предоставляемые пакетом SDK для .NET. Некоторые компоненты являются независимыми от платформы, такими как библиотеки файловой системы, коллекции данных и математические библиотеки. Другие относятся к одной рабочей нагрузке, например к веб-библиотекам ASP.NET Core или библиотеке пользовательского интерфейса .NET MAUI. Расширенная экосистема с открытым исходным кодом в NuGet расширяет библиотеки, которые являются частью среды выполнения. Эти библиотеки предоставляют еще больше компонентов, которые можно использовать.

C# — это строго типизированный язык. Каждая объявленная переменная имеет тип, известный во время компиляции. Компилятор или средства редактирования указывают, используется ли этот тип неправильно. Эти ошибки можно исправить, прежде чем вы когда-либо запускаете программу. Основные типы данных встроены в язык и среду выполнения: такие типы значений, как , `int` `double` ссылочные типы, такие как `char` `string` массивы и другие коллекции. При написании программ вы создаете собственные типы. Эти типы могут быть `struct` типами значений или `class` типами, определяющими объектно-ориентированное поведение. Можно добавить модификатор `record` к типу `struct` или `class`, чтобы компилятор синтезировал код для сравнения равенства. Можно также создавать `interface` определения, определяющие контракт или набор элементов, которые должны предоставлять тип, реализующий этот интерфейс. Вы

также можете определить универсальные типы и методы. Обобщения используют *параметры типа* чтобы заменить фактический тип при использовании.

При написании кода вы определяете функции, также называемые методами, как элементы типов `struct` и `class`. Эти методы определяют поведение ваших типов. Можно перегрузить методы с разными числами или типами параметров. Методы могут при необходимости возвращать значение. Помимо методов, типы C# могут иметь свойства, которые являются элементами данных, поддерживаемыми функциями, называемыми *аксессорами*. Типы C# могут определять события, позволяющие типу уведомлять подписчиков о важных действиях. C# поддерживает объектно-ориентированные методы, такие как наследование и полиморфизм для `class` типов.

Приложения C# используют исключения для создания отчетов и обработки ошибок. Эта практика знакома, если вы использовали C++ или Java. Код создает исключение, если оно не может сделать то, что было предназначено. Другой код, независимо от количества уровней стека вызовов, может при необходимости восстановиться с помощью `try` - `catch` блока.

> **Совет**
>
> Дополнительные сведения о типах, методах и исключениях см. в разделе **основы C#** . Она подробно охватывает **систему типов**, **объектно-ориентированное программирование** и **обработку исключений** .

Отличительные возможности C#

Некоторые элементы C# могут быть менее знакомы.

C# предоставляет сопоставление шаблонов. Эти выражения позволяют проверять данные и принимать решения на основе его характеристик. Сопоставление шаблонов обеспечивает отличный синтаксис для потока управления на основе данных. В следующем коде показано, как можно выразить методы для логических операций *AND*, *OR* и *XOR* с помощью синтаксиса сопоставления шаблонов.

C#

```
public static bool Or(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
```

```
    };

public static bool And(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => false,
        (false, true) => false,
        (false, false) => false,
    };
public static bool Xor(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => false,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
    };
```

Выражения сопоставления шаблонов можно упростить, используя `_` как универсальный элемент для любого значения. В следующем примере показано, как упростить *и* метод.

C#

```
public static bool ReducedAnd(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (_, _) => false,
    };
```

В предыдущих примерах также объявляют *кортежи*, упрощенные структуры данных. *Кортеж* — это упорядоченная последовательность значений фиксированной длины, которая может иметь необязательные имена и отдельные типы. Заключите последовательность в символы `(` и `)`. Объявление определяет кортеж `(left, right)` с двумя логическими значениями: `left` и `right`. Каждая рукоятка переключателя объявляет значение кортежа, например `(true, true)`. Кортежи предоставляют удобный синтаксис для объявления одного значения, содержащего несколько элементов любого типа.

Выражения коллекции предоставляют общий синтаксис для предоставления значений коллекции. Вы пишете значения или выражения между символами `[` и `]`, и компилятор преобразует это выражение в требуемый тип коллекции.

C#

```
int[] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
List<string> names = ["Alice", "Bob", "Charlie", "David"];
```

```
IEnumerable<int> moreNumbers = [.. numbers, 11, 12, 13];
IEnumerable<string> empty = [];
```

В предыдущем примере показаны различные типы коллекций, которые можно инициализировать с помощью выражений коллекции. В одном примере используется пустое `[]` выражение коллекции для объявления пустой коллекции. Другой пример использует `..` *элемент spread* для расширения коллекции и добавления всех его значений в выражение коллекции.

Выражения *индекса* и *диапазона* можно использовать для получения одного или нескольких элементов из индексируемой коллекции:

C#

```
string second = names[1]; // 0-based index
string last = names[^1]; // ^1 is the last element
int[] smallNumbers = numbers[0..5]; // 0 to 4
```

Индекс `^` указывает *с конца* , а не с начала. Элемент `^0` размещён за пределами конца коллекции, таким образом, `^1` является последним элементом. В `..` выражении диапазона обозначается диапазон элементов, которые необходимо включить. Диапазон начинается с первого индекса и включает все элементы до, но не включая элемент в последнем индексе.

Дополнительные сведения об выражениях индекса и диапазона см. в статье "Изучение индексов и диапазонов ".

Языковой интегрированный запрос (LINQ) предоставляет общий синтаксис на основе шаблонов для запроса или преобразования любой коллекции данных. LINQ унифицировывает синтаксис для запроса коллекций в памяти, структурированных данных, таких как XML или JSON, хранилище баз данных и даже облачные API данных. Вы узнаете один набор синтаксиса, и вы можете искать и управлять данными независимо от его хранилища. Следующий запрос находит всех учащихся, средний показатель которого превышает 3,5:

C#

```
var honorRoll = from student in Students
                where student.GPA > 3.5
                select student;
```

Предыдущий запрос работает для многих типов хранилища, представленных с помощью `Students`. Это может быть коллекция объектов, таблица базы данных, большой двоичный

объект облачного хранилища или структура XML. Один и тот же синтаксис запроса работает для всех типов хранилища.

Модель асинхронного программирования на основе задач позволяет писать код, который считывается как будто он выполняется синхронно, даже если он выполняется асинхронно. Он использует ключевые слова `async` и `await` для описания методов, которые являются асинхронными, и когда выражение вычисляется асинхронно. В следующем примере ожидается асинхронный веб-запрос. После завершения асинхронной операции метод возвращает длину ответа:

C#

```
public static async Task<int> GetPageLengthAsync(string endpoint)
{
    var client = new HttpClient();
    var uri = new Uri(endpoint);
    byte[] content = await client.GetByteArrayAsync(uri);
    return content.Length;
}
```

C# также поддерживает `await foreach` инструкцию для итерации коллекции, поддерживаемой асинхронной операцией, например API разбиения по страницам GraphQL. Следующий пример считывает данные порциями, возвращая итератор, который предоставляет доступ к каждому элементу, когда он становится доступен.

C#

```
public static async IAsyncEnumerable<int> ReadSequence()
{
    int index = 0;
    while (index < 100)
    {
        int[] nextChunk = await GetNextChunk(index);
        if (nextChunk.Length == 0)
        {
            yield break;
        }
        foreach (var item in nextChunk)
        {
            yield return item;
        }
        index++;
    }
}
```

Вызывающие методы могут итерировать коллекцию с помощью инструкции `await foreach` :

C#

```
await foreach (var number in ReadSequence())
{
    Console.WriteLine(number);
}
```

Наконец, в рамках экосистемы .NET можно использовать Visual Studio или Visual Studio Code с C# DevKit. Эти средства предоставляют полное представление о C#, включая код, который вы пишете. Они также предоставляют возможности отладки.

> **Совет**
>
> Дополнительные сведения о сопоставлении шаблонов, LINQ и асинхронном программировании см. в **функциональных методах**, **обзоре LINQ** и **асинхронном программировании** .

Дальнейшие шаги

В этой статье представлен краткий обзор языка C#. Вот где идти дальше, в зависимости от вашего опыта:

- **Начните писать код**: Изучите C# с помощью учебников для начинающих.
- **Ознакомьтесь** с разделом "Основы C# " для подробного охвата системы типов, объектно-ориентированного программирования и обработки ошибок.
- **Создайте что-то: изучите**, что можно создать с помощью C#, чтобы найти интересующие вас идеи.
- **Вы пришли из другого языка?** Ознакомьтесь с руководствами для разработчиков Java, JavaScript или Python.
- **Создание файловых приложений**: узнайте, как создавать приложения на основе файлов для небольших программ и прототипов.

Last updated on 06.02.2026

Что можно создать с помощью C#

C# поддерживает широкий спектр типов приложений. Независимо от типа программного обеспечения, которое вы хотите создать, скорее всего, для нее есть рабочая нагрузка . В этой статье представлен обзор наиболее распространенных типов приложений с ссылками на начало работы с каждым из них.

> **Совет**
>
> **Новые возможности программирования?** Не беспокойтесь о выборе рабочей нагрузки еще. Сначала сосредоточьтесь на **обучении языка C#** . Эти типы приложений можно изучить, как только вы освоитесь с основами.
>
> **Опытный разработчик?** Перейдите непосредственно к рабочей нагрузке, которая соответствует вашим целям. Каждый раздел ссылается на необходимую документацию и учебники.

ИИ и машинное обучение

C# интегрируется с средствами искусственного интеллекта и машинного обучения:

- **Agent Framework** — создание агентов и рабочих процессов для Azure, OpenAI, Anthropic, Ollama и многое другое.
- **Инструменты Foundry** — доступ к предварительно созданным возможностям искусственного интеллекта, таким как зрение, понимание языка и распознавание речи.
- **ML.NET** . Создание пользовательских моделей машинного обучения на C# без необходимости глубокого опыта машинного обучения.

Начало работы: учебники ML.NET.

Веб-приложения

Создавайте веб-приложения с помощью ASP.NET Core, кроссплатформенной платформы для создания современных веб-приложений и API. Вы можете создать следующее:

- **Серверные веб-приложения** с помощью Razor Pages или MVC.
- **Интерактивные веб-интерфейсы** с помощью Blazor, который позволяет писать клиентскую логику на C# вместо JavaScript.
- **Веб-API** и минимальные API для внутренних служб.

Начало работы: учебники по ASP.NET Core

Настольные приложения

Используйте C# для создания классических приложений Windows и кроссплатформенных классических приложений, работающих в Windows и macOS:

- **.NET MAUI** создает кроссплатформенные классические приложения, которые работают в Windows, macOS, Android и iOS из одной базы кода.
- **Windows Presentation Foundation (WPF)** создает расширенные классические приложения только для Windows с расширенной графикой и привязкой данных.
- **Windows Forms** предоставляет простой способ создания классических приложений только для Windows с помощью визуального конструктора.

Начало работы: учебники по .NET MAUI.

Мобильные приложения

Создание собственных мобильных приложений для iOS и Android с помощью .NET MAUI. .NET MAUI позволяет совместно использовать код на мобильных и настольных платформах, а также получать доступ к собственным API устройств для камер, датчиков, GPS и т. д.

Начало работы. Создание первого приложения MAUI для .NET.

Облачные и микрослужбы

C# хорошо подходит для создания облачных приложений и микрослужб:

- **Пакет SDK azure для .NET** предоставляет библиотеки для работы со службами Azure, такими как хранилище, обмен сообщениями и базы данных.
- **Рабочие службы** позволяют создавать длительные фоновые службы, работающие в облаке или локальной среде.
- **Aspire** упрощает создание, запуск, отладку и развертывание распределенных приложений.

Начало работы. Начало работы с Azure и .NET.

Игры

C# является одним из самых популярных языков для разработки игр:

- **Unity** — наиболее широко используемый игровой механизм для 2D и трехмерных игр, использует C# в качестве языка сценариев.
- **MonoGame** — платформа с открытым кодом для создания кроссплатформенных игр.
- **CryEngine** — поддерживает C# для сценариев игр.

Вы также можете использовать Visual Studio для разработки игр с этими двигателями.

Интернет вещей (IoT)

Управляйте устройствами и считывайте данные датчика с помощью библиотек Интернета вещей. Приложения C# можно запускать на таких устройствах, как Raspberry Pi и другие компьютеры с одной платой для создания решений Интернета вещей.

Начало работы: учебники по IoT

Дальнейшие шаги

Теперь, когда вы знаете, что можно создать, выберите следующий шаг:

- **Узнайте о языке**: начните с начальных учебников по C# , чтобы узнать основы C#.
- **Ознакомьтесь с C#: ознакомьтесь** с обзором ключевых языковых возможностей C# .
- **Ознакомьтесь с основами: ознакомьтесь** с разделом основы C# для более глубокого охвата системы типов, объектно-ориентированного программирования и многое другое.
- **Вы пришли из другого языка?** Ознакомьтесь с планами разработки Java, JavaScript и TypeScript или Python .

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.02.2026

Введение в C#

Добро пожаловать на введение в учебники по C#. Эти уроки начинаются с интерактивного кода, который можно запустить в GitHub Codespaces. Вы можете изучить основы C# из видеосерии "C# для начинающих" до начала этих интерактивных уроков.

> **Совет**
>
> **Новые возможности программирования?** Начните с урока 1 (**Hello world**) и последовательно пройдите каждый урок.
>
> **Вы пришли из другого языка?** Вы можете пропустить руководства 1–2 и начать с **кортежей и типов**, чтобы изучить новые понятия.

Все учебные пособия используют приложения на базе файлов. Вы можете использовать наше пространство кода GitHub или установить пакет SDK для .NET и начать работу.

https://www.youtube-nocookie.com/embed/9THmGiSPjBQ?si=3kUKFtOMLpEzeq7J

Первые уроки объясняют основные понятия C# с помощью небольших фрагментов кода. Вы узнаете об основах синтаксиса C# и о том, как работать с типами данных, такими как строки, числа и логические элементы. Это все интерактивно, и вы будете писать и запускать код в течение нескольких минут. Эти первые уроки не предполагают предварительного знания о программировании или языке C#. Каждый урок основывается на предыдущих уроках. Вы должны сделать их по порядку. Однако если у вас есть опыт программирования, вы можете пропустить или пролистать первые уроки и начать с новых концепций.

Для использования GitHub Codespaces необходимо создать бесплатную учетную запись GitHub .

Hello World

В учебнике Hello world вы создадите самую простую программу C#. Вы исследуете тип `string` и как работать с текстом.

Числа в C#

В руководстве по числовым номерам в C# вы узнаете, как компьютеры хранят числа и как выполнять вычисления с разными числовыми типами. Вы узнаете основы округления и как выполнять математические вычисления с помощью C#.

Кортежи и типы

В руководстве Кортежи и типы вы узнаете, как создавать типы в языке C#. Вы можете создавать типы *кортежей*, *записей*, *структур* и *классов*. Возможности этих различных типов отражают их различные виды использования.

Ветви и циклы

В руководстве ветвей и циклов приведены основы выбора различных путей выполнения кода на основе значений, хранящихся в переменных. Вы узнаете основы потока управления, который является основой принятия решений и выбора различных действий.

Коллекция списков

Урок коллекции списков предоставляет обзор типа коллекции списков, в которой хранятся последовательности данных. Вы узнаете, как добавлять и удалять элементы, искать элементы и сортировать списки. Вы изучаете различные типы списков.

Сопоставление шаблонов

Урок сопоставления шаблонов содержит общие сведения о *сопоставлении шаблонов*. Сопоставление шаблонов позволяет сравнивать выражение с шаблоном. Успех совпадения определяет, какую логику программы необходимо использовать. Шаблоны могут сравнивать типы, свойства типа или содержимое списка. Можно объединить несколько шаблонов с помощью `and` и `or` `not` логики. Шаблоны предоставляют широкий словарь для проверки данных и принятия решений в программе на основе этой проверки.

Что дальше?

После завершения работы с этими учебниками перейдите к обучению:

- Создание файловых приложений: узнайте, как создавать и запускать программы C# с одним файлом из командной строки.
- Что можно создать с помощью C#: изучите типы приложений, которые можно создать с помощью C#.
- Система типов C#: более подробное представление о типах значений, ссылочных типах, универсальных и т. д.

- Основы C#: узнайте о структуре программы, объектно-ориентированном программировании и обработке ошибок.

Last updated on 24.02.2026

Руководство. Изучение языка C#

В этом руководстве описано, как научиться C#. Вы пишете первую программу C# и видите результаты компиляции и выполнения кода. Он содержит ряд уроков, которые начинаются с программы Hello World. Эти уроки учат вас основам языка C#.

> **Совет**
>
> **Новые возможности программирования?** Начните здесь - это руководство предполагает отсутствие предварительного опыта. **Вы пришли из другого языка?** Вы можете предпочесть пропустить примеры кода и перейти к **числам в C#** или **ветвях и циклах**.

Изучив это руководство, вы:

- ✓ Запустите пространство кода GitHub с помощью среды разработки C#.
- ✓ Создайте первое приложение C#.
- ✓ Создайте и используйте переменные для хранения текстовых данных.
- ✓ Используйте API среды выполнения с текстовыми данными.

Предпосылки

У вас должен быть один из следующих вариантов:

- Учетная запись GitHub для использования GitHub Codespaces. Если у вас еще нет учетной записи, вы можете создать бесплатную учетную запись на GitHub.com.
- Компьютер со следующими инструментами:
  - Пакет SDK для .NET 10.
  - Visual Studio Code.
  - C# DevKit.

Открытие пространств кода

Чтобы запустить GitHub Codespace с средой учебника, откройте окно браузера в репозитории пространства кода учебника. Нажмите зеленую кнопку "*Код*" и вкладку "*Пространства кода* ". Затем выберите `+` знак, чтобы создать новое пространство Codespace с помощью этой среды.

Запуск первой программы

1. При загрузке пространства кода создайте новый файл в папке *учебников* с именем *hello-world.cs*.

2. Откройте новый файл.

3. Введите или скопируйте следующий код в *hello-world.cs*:

   C#

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

4. В окне встроенного терминала сделайте папку *учебников* текущей папкой и запустите программу:

   Интерфейс командной строки.NET

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

Вы запустили первую программу C#. Это простая программа, которая печатает сообщение "Hello World!" Он использует Console.WriteLine метод для печати этого сообщения. `Console` — это тип, представляющий окно консоли. `WriteLine` — это метод типа `Console`, который печатает строку текста в этой текстовой консоли.

Давайте двигаться дальше и исследовать больше. Остальная часть этого урока изучает работу с типом `string`, который представляет текст в C#. Как и тип `Console`, `string` имеет методы. Методы `string` работают с текстом.

Объявление и использование переменных

Первая программа выводит на экран "Hello World!".

> **Совет**
>
> При изучении C# (или любого языка программирования) при написании кода возникают ошибки. **Компилятор** находит эти ошибки и сообщает им вам. Если выходные данные содержат сообщения об ошибках, внимательно изучите пример кода и код в `.cs` файле, чтобы узнать, что нужно исправить. Это упражнение поможет вам узнать структуру кода C#. Вы также можете попросить Copilot найти различия или обнаружить ошибки.

Первая программа ограничена печатью одного сообщения. Вы можете создавать более полезные программы с помощью *переменных*. Переменная — это символ, который можно использовать для выполнения одного кода с разными значениями. Давайте попробуем!

1. Начните со следующего кода:

   ```C#
   string aFriend = "Bill";
   Console.WriteLine(aFriend);
   ```

   Первая строка объявляет переменную, `aFriend` и присваивает ей значение "Билл". Вторая строка выводит имя.

2. Вы можете назначить разные значения любой объявленной переменной. Вы можете изменить имя на одного из ваших друзей. Добавьте эти две строки после уже добавленного кода. Убедитесь, что вы сохраняете объявление переменной `aFriend` и его первоначальное назначение.

   > **ⓘ Внимание**
   >
   > Не удаляйте объявление `aFriend`.

3. Добавьте следующий код в конце предыдущего кода:

   ```C#
   aFriend = "Maira";
   Console.WriteLine(aFriend);
   ```

   Обратите внимание, что одна строка кода выводит два разных сообщения на основе значения, хранящегося в переменной `aFriend`. Вы можете заметить, что слово "Hello" отсутствует в последних двух сообщениях. Давайте исправим это сейчас.

4. Измените строки, которые печатают сообщение в следующий код:

   ```C#
   Console.WriteLine("Hello " + aFriend);
   ```

5. Запустите приложение еще раз, используя `dotnet hello-world.cs`, чтобы просмотреть результаты.

Вы использовали `+` для создания строк из переменных и строк из констант . Есть лучший способ. Можно поместить переменную между `{` и `}` символами, чтобы сообщить C# заменить этот текст значением переменной. Этот процесс называется интерполяция строк.

6. Если добавить `$` перед открытием кавычки строки, можно включить переменные, такие как `aFriend`, внутри строки между фигурными скобками. Попробуйте:

   ```
   C#
   Console.WriteLine($"Hello {aFriend}");
   ```

7. Запустите приложение еще раз, используя `dotnet hello-world.cs`, чтобы просмотреть результаты. Вместо "Hello {aFriend}" сообщение должно быть "Hello Maira".

Работа со строками

Ваше последнее изменение было вашей первой возможностью узнать, что можно сделать со строками. Давайте исследуем больше.

Вы не ограничены одной переменной между фигурными скобками.

1. Попробуйте следующий код в нижней части приложения:

   ```
   C#
   string firstFriend = "Maria";
   string secondFriend = "Sage";
   Console.WriteLine($"My friends are {firstFriend} and {secondFriend}");
   ```

   Строки — это больше, чем коллекция букв. Длину строки можно найти с помощью `Length`. `Length` — это свойство строки и возвращает количество символов в этой строке.

2. Добавьте следующий код в нижней части приложения:

   ```
   C#
   Console.WriteLine($"The name {firstFriend} has {firstFriend.Length}
   letters.");
   Console.WriteLine($"The name {secondFriend} has {secondFriend.Length}
   letters.");
   ```

> **Совет**
>
> Теперь хорошее время для изучения самостоятельно. Вы узнали, что `Console.WriteLine()` записывает текст на экран. Вы узнали, как объявлять переменные и объединять строки вместе. Экспериментируйте в коде. В редакторе есть функция *IntelliSense* , которая предлагает рекомендации по тому, что можно сделать. Введите `.` после `d` в `firstFriend`. Вы увидите список предложений по свойствам и методам, которые можно использовать.

Вы использовали метод ,, Console.WriteLineдля печати сообщений. Метод — это блок кода, реализующий некоторые действия. У него есть имя, что позволяет получить к нему доступ.

> **Совет**
>
> **Подробнее:** Подробно изучите **строки** или ознакомьтесь с **методами и структурой программы** в разделе "Основы C#".

Удаление пробелов из строк

Предположим, что строки имеют начальные или конечные пробелы, которые вы не хотите отображать. Вы хотите **обрезать** пробелы из строк. Метод Trim и связанные методы TrimStart и TrimEnd выполняют эту задачу. Используйте эти методы для удаления начальных и конечных пробелов.

1. Попробуйте выполнить следующий код:

```C#
string greeting = "      Hello World!       ";
Console.WriteLine($"[{greeting}]");

string trimmedGreeting = greeting.TrimStart();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.TrimEnd();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.Trim();
Console.WriteLine($"[{trimmedGreeting}]");
```

Квадратные скобки `[` и `]` помогают представить себе, что делают методы `Trim`, `TrimStart`, и `TrimEnd`. Скобки показывают, где начинается и заканчивается пробел.

Этот пример подкрепляет пару важных понятий для работы со строками. Методы, которые управляют строками, возвращают новые объекты строк, а не вносят изменения. Вы видите, что каждый вызов любого из методов `Trim` возвращает новую строку, но не изменяет исходное сообщение.

Поиск и замена текста в строках

Для работы со строкой можно использовать другие методы. Например, можно использовать команду поиска и замены в редакторе или обработчике слов. Метод Replace выполняет что-то подобное в строке. Он ищет подстроку и заменяет ее другим текстом. Метод Replace принимает два параметра . Эти параметры представляют собой строки между скобками. Первая строка — это текст для поиска. Вторая строка — это текст для замены. Попробуйте это для себя.

1. Добавьте этот код. Введите текст, чтобы увидеть подсказки при вводе `.Re` после переменной `sayHello`.

   C#

   ```
   string sayHello = "Hello World!";
   Console.WriteLine(sayHello);
   sayHello = sayHello.Replace("Hello", "Greetings");
   Console.WriteLine(sayHello);
   ```

   Два других полезных метода преобразуют строку в все прописные буквы или все строчные буквы. Попробуйте выполнить следующий код.

2. Введите текст, чтобы увидеть, как **IntelliSense** предоставляет подсказки, когда вы начинаете вводить `To`:

   C#

   ```
   Console.WriteLine(sayHello.ToUpper());
   Console.WriteLine(sayHello.ToLower());
   ```

   Другая часть операции поиска *и замены* заключается в поиске текста в строке. Для поиска можно использовать метод Contains. Он указывает, содержит ли строка подстроку внутри нее.

3. Попробуйте использовать следующий код для изучения Contains:

C#

```
string songLyrics = "You say goodbye, and I say hello";
Console.WriteLine(songLyrics.Contains("goodbye"));
Console.WriteLine(songLyrics.Contains("greetings"));
```

Метод Contains возвращает *логическое* значение, указывающее, была ли найдена строка. Логическое значение сохраняет либо значение `true`, либо `false`. При отображении в виде текстового вывода они представлены заглавными символами: `True` и `False`соответственно. Вы узнаете больше о булевых значениях в дальнейшем уроке.

Вызов

Два аналогичных метода, StartsWith и EndsWith, также ищут подстроки в строке. Эти методы находят подстроку в начале или в конце строки. Попробуйте изменить предыдущий пример, чтобы использовать StartsWith и EndsWith вместо Contains. Найдите "You" или "goodbye" в начале строки. Найдите "hello" или "goodbye" в конце строки.

> ⓘ **Примечание.**
>
> Следите за знаками препинания, когда тестируете текст в конце строки. Если строка заканчивается точкой, необходимо проверить строку, которая заканчивается точкой.

Вы должны получить `true`, если начинается с "You" и заканчивается на "hello", и `false`, если начинается или заканчивается на "goodbye".

Вы придумали примерно следующий код (разверните, чтобы увидеть ответ):
▼ Details

C#

```
string songLyrics = "You say goodbye, and I say hello";
Console.WriteLine(songLyrics.StartsWith("You"));
Console.WriteLine(songLyrics.StartsWith("goodbye"));

Console.WriteLine(songLyrics.EndsWith("hello"));
Console.WriteLine(songLyrics.EndsWith("goodbye"));
```

Дополнительные сведения о типе `string`:

- руководство по программированию C# о строках.
- Советы и подсказки по работе со строками.

Очистка ресурсов

GitHub автоматически удаляет пространство Codespace через 30 дней бездействия. Если вы планируете изучать дополнительные руководства в этой серии, вы можете оставить Codespace в активном состоянии. Если вы готовы посетить сайт .NET , чтобы скачать пакет SDK для .NET, можно удалить пространство Codespace. Чтобы удалить ваш Codespace, откройте окно браузера и перейдите к вашим Codespace. В текущем окне появится список ваших codespaces (пространств кода). Выберите три точки (`...`) в записи для пространства кода учебника для обучения и выберите "Удалить".

Дальнейшие шаги

Перейдите к следующему руководству в этой серии или изучите связанные разделы в основах C#:

Изучение чисел в C#

- Строки — Узнайте больше о типе, который вы использовали `string` в этом руководстве.
- Методы и структура программы . Узнайте, как организованы программы C#.
- Приложения на основе файлов — сведения о команде, используемой `dotnet run` для запуска кода.

Last updated on 06.02.2026

Руководство. Использование целых чисел и чисел с плавающей запятой в C#

В этом руководстве описаны числовые типы в C#. Вы пишете небольшие объемы кода, а затем компилируете и запускаете этот код. В этом руководстве содержится ряд уроков, в которых рассматриваются числа и математические операции в C#. Эти уроки учат вас основам языка C#.

> **Совет**
>
> **Новые возможности программирования?** Работайте по каждому разделу поочередно. **Вы пришли из другого языка?** Возможно, вы уже знаете различия между целыми числами и числами с плавающей запятой — пробегитесь по основам и сосредоточьтесь на разделах точности и ограничений и десятичного типа.

Изучив это руководство, вы:

- ✓ Запустите пространство кода GitHub с помощью среды разработки C#.
- ✓ Изучите целочисленную математику.
- ✓ Учиться порядку выполнения операций.
- ✓ Узнайте об ограничениях целых чисел и их точности.
- ✓ Изучите типы данных с плавающей запятой.
- ✓ Узнайте о десятичном типе.

Предпосылки

У вас должен быть один из следующих вариантов:

- Учетная запись GitHub для использования GitHub Codespaces. Если у вас еще нет учетной записи, вы можете создать бесплатную учетную запись на GitHub.com.
- Компьютер со следующими инструментами:
  - Пакет SDK для .NET 10.
  - Visual Studio Code.
  - C# DevKit.

Изучение целочисленной математики

Чтобы запустить GitHub Codespace с средой учебника, откройте окно браузера в репозитории пространства кода учебника. Нажмите зеленую кнопку "*Код*" и вкладку "*Пространства кода* ". Затем выберите `+` знак, чтобы создать новое пространство

Codespace с помощью этой среды. Если вы завершили учебник hello world , вы можете открыть это пространство кода вместо создания нового.

1. При загрузке пространства кода создайте новый файл в папке *учебников* с именем *numbers.cs*.

2. Откройте новый файл.

3. Введите или скопируйте следующий код в *numbers.cs*:

   C#

   ```
   int a = 18;
   int b = 6;
   int c = a + b;
   Console.WriteLine(c);
   ```

4. Запустите этот код, введя следующие команды в интегрированном терминале:

   Интерфейс командной строки.NET

   ```
   cd ./tutorials
   dotnet numbers.cs
   ```

   Вы видели одну из фундаментальных математических операций с целыми числами. Тип `int` представляет **целое число**, ноль, положительное или отрицательное целое число. Для добавления используется символ `+`. Другие распространенные математические операции для целых чисел:

   - Символ `-` для вычитания
   - `*` для умножения
   - `/` для деления

5. Начните с изучения этих различных операций. Добавьте эти строки после строки, которая записывает значение `c`:

   C#

   ```
   // subtraction
   c = a - b;
   Console.WriteLine(c);

   // multiplication
   c = a * b;
   Console.WriteLine(c);

   // division
   ```

```
c = a / b;
Console.WriteLine(c);
```

6. Запустите этот код, введя `dotnet numbers.cs` в окне терминала.

Вы также можете поэкспериментировать, написав несколько математических операций в одной строке, если вы хотите. Попробуйте `c = a + b - 12 * 17;` , например. Допускается сочетание переменных и констант.

> **Совет**
>
> При изучении C# (или любого языка программирования) при написании кода могут возникать ошибки. **Компилятор** находит эти ошибки и сообщает им вам. Если выходные данные содержат сообщения об ошибках, внимательно изучите пример кода и код в окне, чтобы узнать, что нужно исправить. Вы также можете попросить Copilot найти различия или обнаружить ошибки. Это упражнение поможет вам узнать структуру кода C#.

Вы завершили первый шаг. Перед началом следующего раздела переместите текущий код в отдельный *метод*. Метод — это ряд инструкций, сгруппированных и присвоенных имени. Метод вызывается путем записи имени метода, за которым следует `()`. Упорядочивание кода в методах упрощает работу с новым примером.

> **Совет**
>
> **Подробнее:** Ознакомьтесь с **методами и структурой программ** в основах C#.

По завершении код должен выглядеть следующим образом:

C#

```
WorkWithIntegers();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);
```

```
    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}
```

Изучение порядка операций

1. Закомментируйте вызов `WorkingWithIntegers()`. Это изменение делает выходные данные менее загромождными при работе в этом разделе:

   C#

   ```
   //WorkWithIntegers();
   ```

   Оператор `//` начинает **комментарий** в C#. Примечания — это любой текст, который вы хотите сохранить в исходном коде, но не выполнять как код. Компилятор не создает исполняемый код из комментариев. Так как `WorkWithIntegers()` это метод, необходимо закомментировать только одну строку.

2. Язык C# определяет приоритет различных математических операций с правилами, согласованными с правилами, которые вы узнали в математике. Умножение и разделение имеют приоритет над добавлением и вычитанием. Изучите приоритет, добавив следующий код после вызова `WorkWithIntegers()` и введя `dotnet numbers.cs` его в окне терминала:

   C#

   ```
   int a = 5;
   int b = 4;
   int c = 2;
   int d = a + b * c;
   Console.WriteLine(d);
   ```

   Выходные данные показывают, что умножение выполняется перед добавлением.

3. Вы можете принудительно применить другой порядок операций, добавив круглые скобки вокруг операции или операций, которые необходимо выполнить сначала. Добавьте следующие строки и снова запустите приложение:

   C#

```
d = (a + b) * c;
Console.WriteLine(d);
```

4. Исследуйте больше, комбинируя различные операции. Добавьте примерно следующие строки. Повторите попытку `dotnet numbers` в окне терминала.

   C#

   ```
   d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
   Console.WriteLine(d);
   ```

   Вы можете заметить интересное поведение целых чисел. Целочисленное деление всегда создает целочисленный результат, даже если результат будет включать десятичную или дробную часть.

5. Если вы не видите это поведение, попробуйте выполнить следующий код:

   C#

   ```
   int e = 7;
   int f = 4;
   int g = 3;
   int h = (e + f) / g;
   Console.WriteLine(h);
   ```

6. Введите `dotnet numbers.cs` еще раз в окне терминала, чтобы просмотреть результаты.

Прежде чем перейти к этому разделу, возьмите весь код, который вы написали в этом разделе, и поместите его в новый метод. Вызовите этот новый метод `OrderPrecedence`. Код должен выглядеть следующим образом:

C#

```
// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
```

```
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}
```

Изучение целочисленной точности и ограничений

В предыдущем примере показано, что целочисленное деление усекает результат. **Остаток** можно получить с помощью оператора **остатка**, символа `%`.

1. Попробуйте выполнить следующий код после вызова `OrderPrecedence()` метода:

   C#

   ```
   int a = 7;
   int b = 4;
   int c = 3;
   int d = (a + b) / c;
   int e = (a + b) % c;
   Console.WriteLine($"quotient: {d}");
   Console.WriteLine($"remainder: {e}");
   ```

2. Целочисленный тип C# отличается от математических целых чисел другим способом: тип `int` имеет минимальные и максимальные ограничения. Попробуйте просмотреть эти ограничения в следующем коде:

   C#

   ```
   int max = int.MaxValue;
   int min = int.MinValue;
   Console.WriteLine($"The range of integers is {min} to {max}");
   ```

3. Если вычисление создает значение, превышающее эти ограничения, вы получите состояние **недополнения** или **переполнения**. Ответ, как представляется, переходит от одного предела к другому. Чтобы просмотреть пример, добавьте эти две строки в код:

   C#

   ```
   int what = max + 3;
   Console.WriteLine($"An example of overflow: {what}");
   ```

Обратите внимание, что ответ очень близок к минимальному (отрицательному) целочислению. Это то же самое, что и `min + 2`. Операция сложения **превысила** допустимые значения для целых чисел. Ответ представляет собой большое отрицательное число, поскольку при переполнении значение "перепрыгивает" с максимального возможного целого числа на минимальное.

`int` Если тип не соответствует вашим потребностям, используйте другие числовые типы с различными ограничениями и точностью. Давайте рассмотрим эти типы чисел далее.

Работа с двойным типом

Числовой тип `double` представляет число с плавающей запятой двойной точности. Эти термины могут быть новыми для вас. Число **с плавающей запятой** полезно для представления не целочисленных чисел, которые могут быть очень большими или небольшими величинами. **двойная точность** — это относительный термин, описывающий количество двоичных цифр, используемых для хранения значения. **числа** двойной точности имеют в два раза больше двоичных цифр, чем **числа одинарной точности**. На современных компьютерах чаще используется двойная точность, чем однократные числа точности. **Числа одинарной точности** объявляются с помощью ключевого слова `float`. Давайте исследуем.

1. Добавьте следующий код и просмотрите результат:

C#

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Обратите внимание, что ответ включает десятичную часть кворента.

2. Попробуйте немного более сложное выражение с двойниками. Можно использовать следующие значения или заменить другие цифры:

C#

```
double a = 19;
double b = 23;
double c = 8;
double d = (a + b) / c;
Console.WriteLine(d);
```

3. Диапазон двойного значения гораздо больше целочисленных значений. Попробуйте добавить следующий код в то, что вы написали до сих пор:

C#

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

Эти значения печатаются в научной нотации. Число слева `E` от знака. Число справа является экспонентой, как мощность 10.

4. Как и десятичные числа в математике, двойные числа в C# могут иметь ошибки округления. Попробуйте использовать следующий код:

C#

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

Вы знаете, что `0.3` - это `3/10`, а не совсем совпадает с `1/3`. Аналогичным образом `0.33` — это `33/100`. Это значение ближе к `1/3`, но по-прежнему не точно. Независимо от того, сколько десятичных разрядов вы добавляете, остается ошибка округления.

челлендж

Попробуйте использовать другие вычисления с большими числами, небольшими числами, умножением и делением с помощью `double` типа. Попробуйте более сложные вычисления. После того как вы проводите некоторое время с задачей, возьмите код, который вы написали, и поместите его в новый метод. Присвойте новому методу `WorkWithDoubles` имя.

Работа с десятичными типами

Вы видели основные числовые типы в C#: целые числа и двойные. Существует еще один тип, который нужно узнать: тип `decimal`. Тип `decimal` имеет меньший диапазон, но более высокую точность, чем `double`.

1. Давайте посмотрим:

   C#

   ```
   decimal min = decimal.MinValue;
   decimal max = decimal.MaxValue;
   Console.WriteLine($"The range of the decimal type is {min} to {max}");
   ```

   Обратите внимание, что диапазон меньше типа `double`.

2. Вы можете увидеть большую точность с десятичным типом, выполнив следующий код:

   C#

   ```
   double a = 1.0;
   double b = 3.0;
   Console.WriteLine(a / b);

   decimal c = 1.0M;
   decimal d = 3.0M;
   Console.WriteLine(c / d);
   ```

   Обратите внимание, что математика с десятичным типом имеет больше цифр справа от десятичной запятой.

   Суффикс `M` на числах указывает, что константа должна использовать `decimal` тип. В противном случае компилятор предполагает тип `double`.

> ⓘ **Примечание.**

Буква `M` — это наиболее визуально отличающаяся буква между ключевыми словами `double` и `decimal`.

челлендж

Теперь, когда вы знаете различные числовые типы, напишите код, который вычисляет область круга, радиус которого составляет 2,50 сантиметра. Помните, что область круга — это радиус, умноженный на ПИ. Одно указание: среда выполнения содержит константу для pi, Math.PI которую можно использовать для этого значения. Math.PI, как и все константы, объявленные в пространстве имен `System.Math`, является значением `double`. По этой причине, для этой задачи, следует использовать значения `double` вместо `decimal`.

Вы должны получить ответ от 19 до 20.

▼ Details

C#

```
double radius = 2.50;
double area = Math.PI * radius * radius;
Console.WriteLine(area);
```

Попробуйте использовать другие формулы, если вы хотите.

Очистка ресурсов

GitHub автоматически удаляет пространство Codespace через 30 дней бездействия. Если вы планируете ознакомиться с дополнительными руководствами в этой серии, вы можете оставить ваш Codespace активным. Если вы готовы посетить сайт .NET , чтобы скачать пакет SDK для .NET, можно удалить пространство Codespace. Чтобы удалить ваш Codespace, откройте окно браузера и перейдите к вашим Codespaces. В окне отобразится список ваших codespaces. Выберите три точки (`...`) в записи для пространства кода учебника для обучения и выберите **удалить**.

Дальнейшие шаги

Следующий урок в этой серии:

Кортежи и типы

Или изучите связанные разделы в основах C#:

- Система типов C# — подробное представление о типах, используемых в этом руководстве.
- Методы и структура программы — узнайте больше о методах, созданных для упорядочивания кода.
- Что можно создать с помощью C# — ознакомьтесь с типами приложений, которые можно создавать с помощью изучаемых вами материалов.
- целочисленные числовые типы
- Числовые типы с плавающей запятой
- встроенные числовые преобразования

Last updated on 24.02.2026

Руководство. Создание типов в C#

В этом руководстве описано, как создавать типы в C#. Вы пишете небольшие объемы кода, а затем компилируете и запускаете этот код. В этом руководстве содержится ряд уроков, которые изучают различные типы типов в C#. Эти уроки учат вас основам языка C#.

> **Совет**
>
> **Новые возможности программирования?** Работайте с каждым разделом в порядке: кортежи, записи, а затем структуры и классы. **Вы пришли из другого языка?** Если вы уже знаете классы и структуры, сосредоточьтесь на **кортежах** и **типах записей**, которые могут быть новыми для вас.

Предыдущие учебники работали с текстом и цифрами. Строки и числа — *это простые типы*: каждый из них хранит одно значение. По мере увеличения размера программ необходимо работать с более сложными структурами данных. C# предоставляет различные типы типов, которые можно определить, когда требуется структура данных с большим количеством полей, свойств или поведения. Давайте начнем изучать эти типы.

Изучив это руководство, вы:

- ✓ Создание типов кортежей и управление ими.
- ✓ Создание типов записей.
- ✓ Узнайте о структурах, классах и интерфейсах.

Предпосылки

У вас должен быть один из следующих вариантов:

- Учетная запись GitHub для использования GitHub Codespaces. Если у вас еще нет учетной записи, вы можете создать бесплатную учетную запись на GitHub.com.
- Компьютер со следующими инструментами:
  - Пакет SDK для .NET 10.
  - Visual Studio Code.
  - C# DevKit.

Чтобы использовать пространства кода, требуется учетная запись GitHub. Если у вас еще нет учетной записи, вы можете создать бесплатную учетную запись на GitHub.com.

Кортежи

Чтобы запустить GitHub Codespace с средой учебника, откройте окно браузера в репозитории пространства кода учебника . Нажмите зеленую кнопку "*Код*" и вкладку "*Пространства кода* ". Затем выберите `+` знак, чтобы создать новое пространство Codespace с помощью этой среды. Если вы выполнили другие учебники в этой серии, вы можете открыть это пространство кода вместо создания нового.

1. При загрузке пространства кода создайте новый файл в папке *учебников* с именем *tuples.cs*.

2. Откройте новый файл.

3. Введите или скопируйте следующий код в *tuples.cs*:

   C#

   ```
   var pt = (X: 1, Y: 2);

   var slope = (double)pt.Y / (double)pt.X;
   Console.WriteLine($"A line from the origin to the point {pt} has a slope of
   {slope}.");
   ```

4. Запустите программу, введя следующие команды в окне встроенного терминала:

   Интерфейс командной строки.NET

   ```
   cd tutorials
   dotnet tuples.cs
   ```

   Кортежи — это упорядоченная последовательность значений с фиксированной длиной. Каждый элемент кортежа имеет тип и необязательное имя.

   > **Совет**
   >
   > При изучении C# (или любого языка программирования) при написании кода возникают ошибки. **Компилятор** находит эти ошибки и сообщает им вам. Если выходные данные содержат сообщения об ошибках, внимательно изучите пример кода и код, чтобы узнать, что нужно исправить. Вы также можете попросить Copilot найти различия или обнаружить какие-либо ошибки. Это упражнение поможет вам узнать структуру кода C#.

5. Добавьте следующий код после предыдущего кода, чтобы изменить элемент кортежа:

   C#

```
pt.X = pt.X + 5;
Console.WriteLine($"The point is now at {pt}.");
```

6. Вы также можете создать новый кортеж, представляющий собой измененную копию исходного, с помощью `with` выражения. Добавьте следующий код после существующего кода и введите `dotnet tuples.cs` в окне терминала, чтобы просмотреть результаты:

   C#

   ```
   var pt2 = pt with { Y = 10 };
   Console.WriteLine($"The point 'pt2' is at {pt2}.");
   ```

   Кортеж `pt2` содержит значение `X` для `pt` (6), а `pt2.Y` равно 10. Кортежи являются структурными типами. Другими словами, типы кортежей не имеют имен, таких как `string` или `int`. Тип кортежа определяется числом членов, называемым *arity*, и типами этих членов. Имена членов предназначены для удобства. Кортеж можно присвоить кортежу с одинаковой арностью и типами, даже если элементы имеют разные имена.

7. Вы можете добавить следующий код после уже написанного кода:

   C#

   ```
   var subscript = (A: 0, B: 0);
   subscript = pt;
   Console.WriteLine(subscript);
   ```

8. Попробуйте его, введя `dotnet tuples.cs` в окне терминала. Переменная `subscript` имеет два члена, оба из которых являются целыми числами. И `subscript`, и `pt` представляют собой экземпляры одного и того же типа кортежа: кортежа, содержащего два элемента `int`.

   Кортежи легко создавать: вы объявляете несколько элементов, заключённых в скобки. Все приведенные ниже объявления определяют различные кортежи с разной арностью и типами элементов.

9. Добавьте следующий код для создания новых типов кортежей:

   C#

   ```
   var namedData = (Name: "Morning observation", Temp: 17, Wind: 4);
   var person = (FirstName: "", LastName: "");
   ```

```
var order = (Product: "guitar picks", style: "triangle", quantity: 500,
UnitPrice: 0.10m);
```

10. Попробуйте ввести `dotnet tuples.cs` снова в окне терминала.

Хотя кортежи легко создать, их возможности ограничены. Типы кортежей не имеют имен, поэтому вы не можете передать смысл набору значений. Типы кортежей не могут добавлять поведение. C# имеет другие типы, которые можно создать, когда ваш тип определяет поведение.

> **Совет**
>
> **Подробнее:** Исследуйте кортежи и другие варианты типов в Основах C#.

Создание типов записей

Кортежи хорошо работают, если требуется несколько значений в одной структуре. Они легкие, и вы можете определить их в процессе использования. По мере роста программы вы можете обнаружить, что в коде используется тот же тип кортежа. Если ваше приложение работает в двумерном графическом пространстве, кортежи, представляющие точки, могут встречаться часто. При поиске этого шаблона объявите `record` тип, который хранит эти значения и предоставляет дополнительные возможности.

1. Добавьте следующий код, чтобы объявить и использовать тип `record` для представления `Point`:

   C#

   ```
   public record Point(int X, int Y);
   ```

   Поместите предыдущий код в нижней части исходного файла. Объявления типов, такие как `record` объявления, должны соответствовать исполняемым инструкциям в приложении на основе файлов.

2. Добавьте следующий код перед объявлением `record` :

   C#

   ```
   Point pt3 = new Point(1, 1);
   var pt4 = pt3 with { Y = 10 };
   Console.WriteLine($"The two points are {pt3} and {pt4}");
   ```

Объявление `record` представляет собой одну строку кода для типа `Point`, с помощью которой значения `X` и `Y` сохраняются в неизменяемых свойствах. Вы используете имя `Point`, где бы вы ни использовали этот тип. Правильно именованные типы, например `Point`, предоставляют сведения о том, как используется тип. В дополнительном коде показано, как использовать `with` выражение для создания новой точки, которая является измененной копией существующей точки. `pt4 = pt3 with { Y = 10 }` Строка гласит: "`pt4` имеет те же значения, что и `pt3`, за исключением того, что `Y` присваивается значение 10." Вы можете добавить любое количество свойств для изменения в одном выражении `with`.

Предыдущее `record` объявление представляет собой одну строку кода, которая заканчивается `;`. Вы можете добавить поведение типу `record`, объявив *членов*. Элемент записи может быть функцией или несколькими элементами данных. Члены типа находятся в объявлении типов между символами `{` и `}`.

3. Удалите `;` и добавьте следующие строки кода после объявления `record`.

```
C#
```

```
{
    public double Slope() => (double)Y / (double)X;
}
```

4. Добавьте следующий код перед объявлением `record` после строки, содержащей `with` выражение:

```
C#
```

```
double slopeResult = pt4.Slope();
Console.WriteLine($"The slope of {pt4} is {slopeResult}");
```

5. Введите `dotnet tuples.cs` в окне терминала для запуска этой версии.

   Вы придали строгость *кортежу*, представляющему значения `X` и `Y`. Вы создали `record`, который определяет именованный тип и включает функцию для вычисления наклона. Тип `record` — это сокращенное обозначение для `record class`: тип `class`, который включает дополнительное поведение.

6. Вы можете изменить `Point` тип, чтобы сделать его `record struct` также:

```
C#
```

```
public record struct Point(int X, int Y)
```

`record struct` является типом `struct`, который включает дополнительное поведение, добавленное ко всем типам `record`.

7. Попробуйте использовать эту версию, введя `dotnet tuples.cs` в окне терминала.

Структура, класс и типы интерфейсов

Все конкретные именованные типы в C# являются либо типами `class`, либо `struct`, включая типы `record`. А `class` — это *ссылочный тип*. А `struct` — это *тип значения*. Переменные типа значения хранят содержимое экземпляра непосредственно в памяти. Другими словами, `record struct Point` хранит два целых числа: `X` и `Y`. Переменные ссылочного типа хранят ссылку (указатель) на хранилище для экземпляра. Другими словами, ссылка `record class Point` на блок памяти, содержащий значения для `X` и `Y`.

На практике это различие означает, что типы значений копируются при назначении, но копия экземпляра класса является копией *ссылки*. Эта скопированная ссылка ссылается на один и тот же экземпляр точки с тем же хранилищем и `X` `Y`.

Модификатор `record` указывает компилятору написать несколько элементов. Дополнительные сведения см. в статье о типах записей в разделе "Основы".

При объявлении `record` типа вы объявляете, что тип должен использовать набор по умолчанию для сравнения равенства, назначения и копирования экземпляров этого типа. Записи — лучший выбор, когда хранение связанных данных является основной функцией вашего типа. При добавлении дополнительных действий рекомендуется использовать `struct` или `class` типы без `record` модификатора.

Используйте `struct` типы значений, если требуется более сложное поведение, но основная ответственность заключается в хранении значений. Используйте `class` типы для использования объектно-ориентированных идиом, таких как инкапсуляция, наследование и полиморфизм.

Вы также можете определить `interface` типы для объявления поведенческих контрактов, которые должны реализовывать разные типы. `struct` Оба `class` типа могут реализовывать интерфейсы.

Обычно используются все эти типы в больших программах и библиотеках. После установки пакета SDK эти типы можно изучить с помощью учебников по классам в разделе основы.

Вы выполнили руководство по созданию типов в C#.

Очистка ресурсов

GitHub автоматически удаляет пространство Codespace через 30 дней бездействия. Если вы планируете ознакомиться с дополнительными руководствами в этой серии, вы можете оставить ваш Codespace активным. Если вы готовы посетить сайт .NET , чтобы скачать пакет SDK для .NET, можно удалить пространство Codespace. Чтобы удалить ваш Codespace, откройте окно браузера и перейдите к вашим Codespace. В окне отобразится список ваших codespaces. Выберите три точки (`...`) в записи для пространства кода учебника для обучения и выберите **удалить**.

Дальнейшие шаги

Перейдите к следующему руководству в этой серии:

Ветви и циклы

Или изучите связанные разделы в основах C#:

- Система типов C# — подробное представление о типах, используемых в этом руководстве.
- Типы записей — узнайте больше о записях и о том, когда их следует использовать.
- Классы — изучение объектно-ориентированного программирования в C#.
- Что можно создать с помощью C# — ознакомьтесь с типами приложений, которые можно создавать с помощью изучаемых вами материалов.

Last updated on 06.02.2026

Руководство: Операторы и циклы C# `if` — условная логика

В этом руководстве описано, как писать код C#, который проверяет переменные и изменяет путь выполнения на основе этих переменных. Вы пишете код C# и видите результаты компиляции и выполнения кода. В этом руководстве содержится ряд уроков, в котором рассматриваются конструкции ветвления и циклирования в C#. Эти уроки учат вас основам языка C#.

> **Совет**
>
> **Новые возможности программирования?** Работайте с каждым разделом по порядку. **Вы пришли из другого языка?** Синтаксис `if`/`else` цикла выглядит знакомым — сфокусируйтесь на **вложенных циклах** и **задаче** для практики шаблонов C#.

Изучив это руководство, вы:

- ✓ Запустите пространство кода GitHub с помощью среды разработки C#.
- ✓ Изучите операторы `if` и `else`.
- ✓ Используйте циклы для повторения операций.
- ✓ Работа с вложенными циклами.
- ✓ Объедините ветви и циклы.

Предпосылки

У вас должен быть один из следующих вариантов:

- Учетная запись GitHub для использования GitHub Codespaces. Если у вас еще нет учетной записи, создайте бесплатную учетную запись на GitHub.com.
- Компьютер со следующими инструментами:
  - Пакет SDK для .NET 10.
  - Visual Studio Code.
  - C# DevKit.

Использование инструкций if

Чтобы запустить GitHub Codespace с средой учебника, откройте окно браузера в репозитории пространства кода учебника. Нажмите зеленую кнопку "*Код*" и вкладку

"Пространства кода ". Затем выберите `+` знак, чтобы создать новое пространство Codespace с помощью этой среды. Если вы выполнили другие учебники в этой серии, вы можете открыть это пространство кода вместо создания нового.

1. При загрузке пространства кода создайте новый файл в папке *учебников* с именем *branches-loops.cs*.

2. Откройте новый файл.

3. Введите или скопируйте следующий код в *branches-loops.cs*:

   C#

   ```
   int a = 5;
   int b = 6;
   if (a + b > 10)
       Console.WriteLine("The answer is greater than 10.");
   ```

4. Попробуйте этот код, введя следующую команду в интегрированном терминале:

   Интерфейс командной строки.NET

   ```
   cd tutorials
   dotnet branches-loops.cs
   ```

   Вы должны увидеть сообщение «Ответ больше 10», напечатанное в вашей консоли.

5. Измените объявление `b` таким образом, чтобы сумма была меньше 10:

   C#

   ```
   int b = 3;
   ```

6. Введите `dotnet branches-loops.cs` еще раз в окне терминала.

   Так как ответ меньше 10, ничего не печатается. Условие, которое вы тестируете, ложно. У вас нет кода для выполнения, потому что вы написали только одну из возможных ветвей для оператора `if`: истинную ветвь.

> **Совет**
>
> При изучении C# (или любого языка программирования) при написании кода могут возникать ошибки. Компилятор находит и сообщает об ошибках. Внимательно изучите выходные данные ошибки и код, создающий ошибку. Вы также можете

попросить Copilot найти различия или обнаружить какие-либо ошибки. Ошибка компилятора обычно помогает найти проблему.

В этом первом примере показаны возможности `if` и логических типов данных. *Логическое* значение — это переменная, которая может иметь одно из двух значений: `true` или `false`. C# определяет специальный тип `bool` для логических переменных. Инструкция `if` проверяет значение `bool`. Если значение равно `true`, выполняется инструкция, следующая за `if`. В противном случае это будет пропущено. Этот процесс проверки условий и выполнения инструкций на основе этих условий является мощным. Давайте исследуем больше.

Сделать так, чтобы if и else работали вместе.

Чтобы выполнить другой код в истинных и ложных ветвях, создайте `else` ветвь, которая выполняется при условии false. Попробуйте ветку `else`.

1. Добавьте последние две строки в следующем фрагменте кода (у вас уже должно быть первые четыре):

   C#

   ```
   int a = 5;
   int b = 3;
   if (a + b > 10)
       Console.WriteLine("The answer is greater than 10");
   else
       Console.WriteLine("The answer is not greater than 10");
   ```

   Инструкция, следующая за ключевым словом `else`, выполняется только в том случае, если тестируемое `false` условие. Объединение `if` и `else` с булевыми условиями дает все необходимые возможности для обработки как `true`, так и `false` условий.

   > **ⓘ Внимание**
   >
   > Отступ под инструкциями `if` и `else` предназначен для человеческого восприятия. Язык C# не обрабатывает отступы или пробелы как значительные. Инструкция, следующая за ключевым словом `if` , `else` выполняется на основе условия. Все примеры в этом руководстве следуют обычной практике отступов строк в зависимости от потока выполнения операторов.

Так как отступ не является значительным, необходимо использовать `{` и `}`, чтобы указать, когда требуется, чтобы несколько операторов были частью блока условного выполнения. Программисты C# обычно используют эти фигурные скобки для всех `if` предложений и `else` предложений.

2. Следующий пример совпадает с тем, что вы создали в предыдущем примере, с добавлением `{` и `}`. Измените код, чтобы он соответствовал следующему коду:

```C#
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

> **Совет**
>
> В остальной части этого руководства все примеры кода включают фигурные скобки, следуя общепринятым практикам.

3. Вы можете протестировать более сложные условия. Добавьте следующий код после кода, который вы написали до сих пор.

```C#
int a = 5;
int b = 3;
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

Символ `==` проверяет *равенство*. Использование `==` отличает проверку на равенство от присваивания, которое вы видели ранее в `a = 5`.

Элемент `&&` представляет «и». Это означает, что оба условия должны быть true для выполнения инструкции в истинной ветви. В этих примерах также показано, что в каждой условной ветви может быть несколько операторов, при условии, что они заключены в `{` и `}`.

4. Можно также использовать `||` для представления "или". Добавьте следующий код после того, что вы написали на данный момент.

```
C#
```

```
if ((a + b + c > 10) || (a == b))
```

5. Измените значения `a`, `b`, и `c`, и переключитесь между `&&` и `||`, чтобы исследовать. Вы получите больше понимания того, как операторы `&&` и `||` работают.

6. Вы завершили первый шаг. Перед началом следующего раздела переместите текущий код в отдельный метод. Это изменение упрощает работу с новым примером. Поместите существующий код в метод с именем `ExploreIf()`. Вызовите его из верхней части программы. После завершения этих изменений код должен выглядеть следующим образом:

```
C#
```

```
ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the second");
    }
    else
```

```
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the
second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the
second");
    }
}
```

7. Закомментируйте вызов `ExploreIf()`. Это делает выходные данные менее загроможденными при работе в этом разделе:

```
C#

//ExploreIf();
```

Элемент `//` начинает **комментарий** в C#. Примечания — это любой текст, который вы хотите сохранить в исходном коде, но не выполняется как код. Компилятор не создает исполняемый код из комментариев.

Использование циклов для повторения операций

Еще одной важной концепцией создания больших программ является **циклы**. Используйте циклы для повторения инструкций, которые требуется выполнить несколько раз.

> **Совет**
>
> **Подробнее:** Ознакомьтесь с инструкциями итерации и операторами выбора в справочнике по языку C#.

1. Добавьте этот код после вызова `ExploreIf`:

C#

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

Оператор `while` проверяет условие и выполняет оператор, следующий за `while`. Он повторяет проверку условия и выполняет эти инструкции до тех пор, пока условие не станет ложным.

В этом примере есть еще один новый оператор. После `++` переменной `counter` идет оператор **инкремента**. Он добавляет 1 к значению `counter` и сохраняет это значение в переменной `counter` .

> ⓘ **Внимание**
>
> Убедитесь, что условие цикла `while` становится ложным при выполнении кода. В противном случае создается **бесконечный цикл** , в котором программа никогда не заканчивается. Этот пример не демонстрирует это поведение, так как необходимо вынудить программу завершить работу, используя сочетание клавиш CTRL-C или другие средства.

Цикл `while` проверяет условие перед выполнением кода после `while`.

2. Цикл `do` ... `while` сначала выполняет код, а затем проверяет условие. Цикл *do while* показан в следующем коде:

C#

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Этот `do` цикл и предыдущий `while` цикл создают одни и те же выходные данные.

Давайте перейдем к одному последнему оператору цикла.

Работа с циклом for

Еще одна распространенная инструкция цикла в коде C# — это цикл `for`.

1. Попробуйте использовать следующий код:

   ```C#
   for (int counter = 0; counter < 10; counter++)
   {
       Console.WriteLine($"Hello World! The counter is {counter}");
   }
   ```

   Предыдущий `for` цикл выполняет ту же работу, что и `while` цикл и `do` цикл, которые вы уже использовали. В `for` инструкции есть три части, которые управляют тем, как она работает:

   - Первая часть — это **инициализатор для**: `int counter = 0;` объявляет, что `counter` — это переменная цикла и задает для нее начальное значение `0`.
   - Средняя часть является условием: `counter < 10` объявляет, что этот `for` цикл продолжает выполняться до тех пор, пока значение `counter` меньше 10.
   - Окончательная часть — это **итератор**: `counter++` указывает, как изменить переменную цикла после выполнения блока, следующего `for` за инструкцией. Здесь указывается, что `counter` увеличивается на 1 при каждом выполнении блока.

2. Поэкспериментируйте с этими условиями самостоятельно. Попробуйте выполнить каждое из следующих изменений:

   - Измените инициализатор, чтобы начать с другого значения.
   - Измените условие, чтобы остановиться на другом значении.

Когда вы закончите, перейдите к следующему разделу, чтобы написать код самостоятельно и использовать то, что вы узнали.

Существует еще один оператор цикла, который не рассматривается в этом руководстве: оператор `foreach`. Оператор `foreach` повторяет свою инструкцию для каждого элемента в последовательности элементов. Чаще всего он используется с *коллекциями*. Это описано в следующем руководстве.

Создание вложенных циклов

Можно вложить цикл `while`, `do` или `for` внутрь другого цикла. Объединение каждого элемента во внешнем цикле с каждым элементом во внутреннем цикле создает матрицу. Давайте создадим набор буквенно-цифровых пар для представления строк и столбцов.

1. Добавьте следующий `for` цикл, создающий строки:

```C#
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

2. Добавьте еще один цикл для создания столбцов:

```C#
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

3. Наконец, вложите цикл столбцов в цикл строк, чтобы сформировать пары.

```C#
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

   Внешний цикл увеличивается один раз для каждого полного запуска внутреннего цикла. Поменяйте местами вложенные строки и столбцы и посмотрите изменения. По завершении поместите код из этого раздела в метод с именем `ExploreLoops()`.

Объединение ветвей и циклов

Теперь, когда вы использовали инструкцию `if` и конструкции циклического цикла на языке C#, узнайте, можно ли написать код C#, чтобы найти сумму всех целых чисел 1–20, которые являются делимыми на 3. Ниже приведены несколько подсказок:

- Оператор `%` дает остаток от операции деления.
- Оператор `if` определяет условие для проверки, должно ли число быть включено в сумму.
- Цикл `for` может помочь повторить ряд шагов для всех чисел 1–20.

Попробуйте самостоятельно. Затем проверьте, как вы сделали. В качестве подсказки, правильный ответ должен быть 63.

Ты придумала что-то вроде этого?

▼ Details

C#

```
int sum = 0;
for (int number = 1; number < 21; number++)
{
    if (number % 3 == 0)
    {
        sum = sum + number;
    }
}
Console.WriteLine($"The sum is {sum}");
```

Вы завершили урок "Ветви и циклы".

Очистка ресурсов

GitHub автоматически удаляет пространство Codespace через 30 дней бездействия. Если вы планируете изучать дополнительные руководства в этой серии, вы можете оставить Codespace в активном состоянии. Если вы готовы посетить сайт .NET , чтобы скачать пакет SDK для .NET, можно удалить пространство Codespace. Чтобы удалить ваш Codespace, откройте окно браузера и перейдите к вашим Codespace. В текущем окне появится список ваших codespaces (пространств кода). Выберите три точки (`...`) в записи для пространства кода учебника для обучения и выберите **удалить**.

Дальнейшие шаги

Перейдите к следующему руководству в этой серии:

Сведения о коллекциях

Или изучите связанные разделы в основах C#:

- Сопоставление шаблонов — мощная альтернатива сложным `if`/`else` цепочкам.
- Методы и структура программы . Узнайте, как упорядочить методы, созданные в этом руководстве.
- Что можно создать с помощью C# — ознакомьтесь с типами приложений, которые можно создавать с помощью изучаемых вами материалов.

- Инструкции выбора
- Инструкции итерации

Last updated on 09.02.2026

Руководство по управлению коллекциями данных с помощью list<T> в C#

В этом вводном руководстве приведены общие сведения о языке C# и основах System.Collections.Generic.List<T> класса.

В этом руководстве описано, как научиться C#. Вы пишете код C# и видите результаты компиляции и выполнения этого кода. Он содержит ряд уроков, которые создают, изменяют и изучают коллекции. Вы работаете в первую очередь с классом List<T> .

> **Совет**
>
> **Новые возможности программирования?** Работайте по каждому разделу поочередно. **Вы пришли из другого языка?** Если вы уже знаете динамические массивы или списки, пройдите основы и сосредоточьтесь на **поиске и сортировке**, **списках других типов** и **проблеме**.

Изучив это руководство, вы:

- ✓ Запустите пространство кода GitHub с помощью среды разработки C#.
- ✓ Создание различных типов списков.
- ✓ Изменение содержимого списка.
- ✓ Поиск и сортировка списков.

Предпосылки

У вас должен быть один из следующих вариантов:

- Учетная запись GitHub для использования GitHub Codespaces. Если у вас еще нет учетной записи, вы можете создать бесплатную учетную запись на GitHub.com.
- Компьютер со следующими инструментами:
  - Пакет SDK для .NET 10.
  - Visual Studio Code.
  - C# DevKit.

Базовый пример списка

Чтобы запустить GitHub Codespace с средой учебника, откройте окно браузера в репозитории пространства кода учебника . Нажмите зеленую кнопку "*Код*" и перейдите на вкладку "*Пространства кода* ". Затем выберите `+` знак, чтобы создать новое пространство Codespace с помощью этой среды. Если вы выполнили другие учебники в этой серии, вы можете открыть это пространство кода вместо создания нового.

1. При загрузке пространства кода создайте файл в папке *учебников* с именем *lists.cs*.
2. Откройте новый файл.
3. Введите или скопируйте следующий код в *lists.cs*:

   C#

   ```
   List<string> names = ["<name>", "Ana", "Felipe"];
   foreach (var name in names)
   {
       Console.WriteLine($"Hello {name.ToUpper()}!");
   }
   ```

4. Запустите код, введя следующую команду в окне терминала:

   Интерфейс командной строки.NET

   ```
   cd tutorials
   dotnet lists.cs
   ```

Вы создали список строк, добавили в этот список три имени и напечатали имена заглавными буквами. Вы используете концепции, которые вы изучили в предыдущих руководствах, чтобы перебирать список.

Код для отображения имен использует функцию интерполяции строк . Когда перед `string` поставить символ `$`, можно внедрить код на C# в строку. Фактическая строка заменяет код C# значением, которое он создает. В этом примере он заменяет каждое `{name.ToUpper()}` имя, преобразованное в прописные буквы, так как вызывается String.ToUpper метод.

Давайте продолжаем изучать.

Изменение содержимого списка

Созданная коллекция использует List<T> тип. Этот тип сохраняет последовательности элементов. Вы указываете тип элементов между угловой скобкой.

> **Совет**
>
> **Подробнее:** Изучите коллекции и универсальные шаблоны в документации по C#.

Один из важных аспектов этого List<T> типа заключается в том, что он может увеличиваться или уменьшаться, поэтому можно добавлять или удалять элементы. Вы можете увидеть результаты, изменив содержимое после его отображения.

1. Добавьте следующий код после написанного кода (цикл, который печатает содержимое):

   C#

   ```
   Console.WriteLine();
   names.Add("Maria");
   names.Add("Bill");
   names.Remove("Ana");
   foreach (var name in names)
   {
       Console.WriteLine($"Hello {name.ToUpper()}!");
   }
   ```

   Вы добавили еще два имени в конец списка. Вы также удалили один из них. Выходные данные из этого блока кода отображают начальное содержимое, а затем выводит пустую строку и новое содержимое.

2. Тип List<T> также позволяет ссылаться на отдельные элементы по **индексу**. Доступ к элементам можно получить с помощью маркеров `[` и `]`. Вставьте следующий код после написанного вами и протестируйте.

   C#

   ```
   Console.WriteLine($"My name is {names[0]}.");
   Console.WriteLine($"I've added {names[2]} and {names[3]} to the list.");
   ```

   Вы не можете получить доступ к концу списка. Вы можете проверить длину списка, используя свойство Count.

3. Добавьте следующий код:

   C#

   ```
   Console.WriteLine($"The list has {names.Count} people in it");
   ```

Введите `dotnet lists.cs` еще раз в окне терминала, чтобы просмотреть результаты. В C#индексы начинаются с 0, поэтому самый большой допустимый индекс меньше числа элементов в списке.

Дополнительные сведения об индексах см. в статье "Изучение индексов и диапазонов ".

Поиск и сортировка списков

В примерах используются относительно небольшие списки, но приложения часто могут создавать списки с большим количеством элементов, иногда нумерации в тысячах. Чтобы найти элементы в этих больших коллекциях, необходимо выполнить поиск по списку различных элементов. Метод IndexOf ищет элемент и возвращает индекс элемента. Если элемент отсутствует в списке, `IndexOf` возвращается `-1`.

1. Попробуйте узнать, как это работает. Добавьте следующий код после того, как вы написали до сих пор:

   C#

   ```
   var index = names.IndexOf("Felipe");
   if (index == -1)
   {
       Console.WriteLine($"When an item is not found, IndexOf returns {index}");
   }
   else
   {
       Console.WriteLine($"The name {names[index]} is at index {index}");
   }

   index = names.IndexOf("Not Found");
   if (index == -1)
   {
       Console.WriteLine($"When an item is not found, IndexOf returns {index}");
   }
   else
   {
       Console.WriteLine($"The name {names[index]} is at index {index}");
   }
   ```

   Возможно, вы не знаете, находится ли элемент в списке, поэтому всегда проверяйте индекс, возвращаемый IndexOf. Если `-1`, то элемент не найден.

2. Вы также можете сортировать элементы в списке. Метод Sort сортирует все элементы в списке в обычном порядке (в алфавитном порядке для строк). Добавьте этот код и запустите его еще раз:

C#

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Списки других типов

До сих пор вы в списках использовали тип `string`. Давайте сделаем List<T>, используя другой тип. Давайте создадим набор чисел.

1. Добавьте следующий код в конце исходного файла:

   C#

   ```
   List<int> fibonacciNumbers = [1, 1];
   ```

   Этот код создает список целых чисел и задает первые два целых числа значением 1. *Последовательность Fibonacci*, последовательность чисел, начинается с двух 1s. Каждое следующее число Fibonacci найдено путем принятия суммы предыдущих двух чисел.

2. Добавьте следующий код:

   C#

   ```
   var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
   var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

   fibonacciNumbers.Add(previous + previous2);

   foreach (var item in fibonacciNumbers)
   {
       Console.WriteLine(item);
   }
   ```

3. Введите `dotnet lists.cs`, чтобы увидеть результаты в окне терминала.

Вызов

Узнайте, можно ли объединить некоторые понятия из этого и более ранних уроков. Разверните то, что вы создали до сих пор с числами Фибоначчи. Попробуйте написать

код, чтобы создать первые 20 чисел в последовательности. (Как намек, 20-й номер Фибоначчи равен 6765.)

Вы придумали что-то подобное этому коду?

▼ Details

C#

```
List<int> fibonacciNumbers = [1, 1];

while (fibonacciNumbers.Count < 20)
{
    var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
    var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

    fibonacciNumbers.Add(previous + previous2);
}
foreach (var item in fibonacciNumbers)
{
    Console.WriteLine(item);
}
```

При каждой итерации цикла вы принимаете последние два целых числа в списке, суммируете их и добавляете это значение в список. Цикл повторяется до добавления в список 20 элементов.

Вы завершили руководство по списку.

Очистка ресурсов

GitHub автоматически удаляет пространство Codespace через 30 дней бездействия. Если вы планируете ознакомиться с дополнительными руководствами в этой серии, вы можете оставить ваш Codespace активным. Если вы готовы посетить сайт .NET , чтобы скачать пакет SDK для .NET, можно удалить пространство Codespace. Чтобы удалить ваш Codespace, откройте окно браузера и перейдите к вашим Codespace. В окне отобразится список ваших codespaces. Выберите три точки (`...`) в записи для пространства кода учебника для обучения и выберите **удалить**.

Дальнейшие шаги

Перейдите к следующему руководству в этой серии:

Сопоставление шаблонов

Или изучите связанные разделы в основах C#:

- Обобщения в C# — разберитесь в синтаксисе `<T>`, используемом в этом руководстве.
- Обзор LINQ — запрос и преобразование коллекций с помощью языкового интегрированного запроса.
- Что можно создать с помощью C# — ознакомьтесь с типами приложений, которые можно создавать с помощью изучаемых вами материалов.
- Выбор типа коллекции
- часто используемые типы коллекций
- Когда следует использовать универсальные коллекции

Last updated on 24.02.2026

Руководство. Использование C# для сопоставления данных с шаблонами

В этом руководстве описано, как использовать сопоставление шаблонов для проверки данных в C#. Вы пишете небольшие объемы кода, а затем компилируете и запускаете этот код. В этом руководстве содержится ряд уроков, которые изучают различные типы шаблонов, поддерживаемых C#. Эти уроки учат вас основам языка C#.

> **Совет**
>
> **Новые возможности программирования?** Работайте по каждому разделу поочередно. **Вы пришли из другого языка?** Если вы уже знакомы с `switch` инструкциями, сосредоточьтесь на **исчерпывающих совпадениях** и **шаблонах типов** — это уникальные особенности языка C#.

Изучив это руководство, вы:

- ✓ Запустите пространство кода GitHub с помощью среды разработки C#.
- ✓ Тестируйте данные для дискретных значений.
- ✓ Соотносить перечисляемые данные со значением.
- ✓ Создайте исчерпывающие совпадения с помощью `switch` выражений.
- ✓ Сопоставление типов с использованием шаблонов типов.

Предпосылки

У вас должен быть один из следующих вариантов:

- Учетная запись GitHub для использования GitHub Codespaces. Если у вас еще нет учетной записи, вы можете создать бесплатную учетную запись на GitHub.com.
- Компьютер со следующими инструментами:
  - Пакет SDK для .NET 10.
  - Visual Studio Code.
  - C# DevKit.

Сопоставление значения

В предыдущих руководствах показаны встроенные типы и типы, которые определяются как кортежи или записи. Вы можете проверить экземпляры этих типов на соответствие *шаблону*. Совпадает ли экземпляр с шаблоном, определяет действия, выполняемые

программой. В приведенных ниже примерах вы увидите `?` после имен типов. Этот символ позволяет значению этого типа иметь значение NULL (например, `bool?` может быть `true`, `false` или `null`). Дополнительные сведения см. в разделе "Типы значений, допускающих значение NULL". Начнем с изучения способа использования шаблонов.

Чтобы запустить GitHub Codespace с средой учебника, откройте окно браузера в репозитории пространства кода учебника . Нажмите зеленую кнопку "*Код*" и вкладку "*Пространства кода* ". Затем выберите `+` знак, чтобы создать новое пространство Codespace с помощью этой среды. Если вы выполнили другие уроки в этой серии, вы можете открыть это кодовое пространство.

1. При загрузке пространства кода создайте новый файл в папке *учебников* с именем *patterns.cs*.

2. Откройте новый файл.

3. Во всех примерах в этом руководстве используются текстовые входные данные, представляющие ряд банковских транзакций в виде входных данных, разделенных запятыми (CSV). В каждом из образцов каждую запись можно сопоставить с шаблоном с помощью выражения `is` или `switch`. Этот первый пример разбивает каждую строку на `,` символ, а затем *сопоставляет* первое строковое поле со значением "ДЕПОЗИТ" или "ВЫВОД" с помощью `is` выражения. При совпадении сумма транзакции добавляется или вычитается из текущего баланса счета. Чтобы увидеть его работу, добавьте следующий код в *patterns.cs*:

   C#

   ```
   string bankRecords = """
       DEPOSIT,   10000, Initial balance
       DEPOSIT,     500, regular deposit
       WITHDRAWAL, 1000, rent
       DEPOSIT,    2000, freelance payment
       WITHDRAWAL,  300, groceries
       DEPOSIT,     700, gift from friend
       WITHDRAWAL,  150, utility bill
       DEPOSIT,    1200, tax refund
       WITHDRAWAL,  500, car maintenance
       DEPOSIT,     400, cashback reward
       WITHDRAWAL,  250, dining out
       DEPOSIT,    3000, bonus payment
       WITHDRAWAL,  800, loan repayment
       DEPOSIT,     600, stock dividends
       WITHDRAWAL,  100, subscription fee
       DEPOSIT,    1500, side hustle income
       WITHDRAWAL,  200, fuel expenses
       DEPOSIT,     900, refund from store
       WITHDRAWAL,  350, shopping
   ```

```
    DEPOSIT,    2500, project milestone payment
    WITHDRAWAL,  400, entertainment
    """;

double currentBalance = 0.0;
var reader = new StringReader(bankRecords);

string? line;
while ((line = reader.ReadLine()) is not null)
{
    if (string.IsNullOrWhiteSpace(line)) continue;
    // Split the line based on comma delimiter and trim each part
    string[] parts = line.Split(',');

    string? transactionType = parts[0]?.Trim();
    if (double.TryParse(parts[1].Trim(), out double amount))
    {
        // Update the balance based on transaction type
        if (transactionType?.ToUpper() is "DEPOSIT")
            currentBalance += amount;
        else if (transactionType?.ToUpper() is "WITHDRAWAL")
            currentBalance -= amount;

        Console.WriteLine($"{line.Trim()} => Parsed Amount: {amount}, New
Balance: {currentBalance}");
    }
}
```

4. Затем введите следующий текст в окне терминала:

Интерфейс командной строки.NET

```
cd tutorials
dotnet patterns.cs
```

5. Проверьте выходные данные. Вы можете увидеть, что каждая строка обрабатывается, сравнивая значение текста в первом поле.

Можно аналогично создать предыдущий пример с помощью `==` оператора для проверки того, что два `string` значения равны. Сравнение переменной с константой является основным стандартным блоком для сопоставления шаблонов. Давайте рассмотрим больше основных элементов, которые являются частью сопоставления шаблонов.

> **Совет**
>
> **Подробнее:** Ознакомьтесь с описанием **сопоставления шаблонов** в разделе "Основы C#" для комплексного обзора всех типов шаблонов.

Совпадения элементов перечисления

Другой распространенный способ сопоставления шаблонов — сопоставление значений `enum` типа. Следующий пример обрабатывает входные записи, чтобы создать *кортеж*, где первое значение является `enum` значением, которое отмечает депозит или вывод. Второе значение — это значение транзакции.

1. Добавьте следующий код в конец исходного файла. Он определяет перечисление `TransactionType` :

   C#

   ```
   public enum TransactionType
   {
       Deposit,
       Withdrawal,
       Invalid
   }
   ```

2. Добавьте функцию для анализа банковской транзакции в кортеж, содержащий тип транзакции и значение транзакции. Добавьте следующий код перед объявлением `TransactionType` перечисления:

   C#

   ```
   static IEnumerable<(TransactionType type, double amount)>
   TransactionRecords(string inputText)
   {
       var reader = new StringReader(inputText);
       string? line;
       while ((line = reader.ReadLine()) is not null)
       {
           string[] parts = line.Split(',');

           string? transactionType = parts[0]?.Trim();
           if (double.TryParse(parts[1].Trim(), out double amount))
           {
               // Update the balance based on transaction type
               if (transactionType?.ToUpper() is "DEPOSIT")
                   yield return (TransactionType.Deposit, amount);
               else if (transactionType?.ToUpper() is "WITHDRAWAL")
                   yield return (TransactionType.Withdrawal, amount);
           }
           else {
           yield return (TransactionType.Invalid, 0.0);
           }
       }
   }
   ```

3. Добавьте новый цикл для обработки данных транзакции с помощью вами объявленного перечисления `TransactionType`.

```C#
currentBalance = 0.0;

foreach (var transaction in TransactionRecords(bankRecords))
{
    if (transaction.type == TransactionType.Deposit)
        currentBalance += transaction.amount;
    else if (transaction.type == TransactionType.Withdrawal)
        currentBalance -= transaction.amount;
    Console.WriteLine($"{transaction.type} => Parsed Amount:
{transaction.amount}, New Balance: {currentBalance}");
}
```

В предыдущем примере также используется `if` инструкция для проверки значения `enum` выражения. Другая форма сопоставления шаблонов использует `switch` выражение. Давайте рассмотрим этот синтаксис и как его можно использовать.

Исчерпывающие совпадения с `switch`

Ряд `if` инструкций может протестировать ряд условий. Но компилятор не может определить, являются ли ряд операторов `if` *исчерпывающими*, или если более поздние `if` условия *охвачены* предыдущими условиями. *Исчерпывающим* образом означает, что один из `if` `else` предложений в серии тестов обрабатывает все возможные входные данные. Если ряд инструкций `if` является исчерпывающим, все возможные входные данные удовлетворяют по крайней мере одному `if` или `else` предложению. *Подстановочное правило* означает, что позднее `if` или `else` условие не может быть достигнуто, так как предыдущие `if` или `else` условия соответствуют всем возможным входным данным. Например, в следующем примере кода одно предложение никогда не совпадает:

```C#
int n = GetNumber();

if (n < 20)
    Console.WriteLine("n is less than 20");
else if (n < 10)
    Console.WriteLine("n is less than 10"); // unreachable
else
    Console.WriteLine("n is greater than 20");
```

Условие `else if` никогда не выполняется, так как любое число меньше 10 также меньше 20. Выражение `switch` гарантирует выполнение обоих этих характеристик, что приводит к меньшим ошибкам в приложениях. Давайте попробуем и поэкспериментируем.

1. Скопируйте следующий код. Замените две инструкции `if` в вашем `foreach` цикле на скопированное выражение `switch`.

   C#

   ```
   currentBalance += transaction switch
   {
       (TransactionType.Deposit, var amount) => amount,
       (TransactionType.Withdrawal, var amount) => -amount,
       _ => 0.0,
   };
   ```

2. Введите `dotnet patterns.cs` в окне терминала, чтобы запустить новый пример.

   При запуске кода вы увидите, что он работает так же.

3. Чтобы продемонстрировать *подсумпцию*, переставьте ветви переключателя, как показано в следующем фрагменте:

   C#

   ```
   currentBalance += transaction switch
   {
       (TransactionType.Deposit, var amount) => amount,
       _ => 0.0,
       (TransactionType.Withdrawal, var amount) => -amount,
   };
   ```

   После переупорядочивания рычагов переключателя введите `dotnet patterns.cs` в окне терминала. Компилятор сообщает об ошибке, потому что ветка с `_` соответствует всем значениям. В результате, этот последний сегмент с `TransactionType.Withdrawal` никогда не выполняется. Компилятор сообщает, что в коде что-то не так.

   Компилятор сообщает предупреждение, если выражение, проверенное в `switch` выражении, может содержать значения, которые не соответствуют ни одному переключателю. Если некоторые значения не могут соответствовать условию, `switch` выражение не является *исчерпывающим*. Компилятор также сообщает предупреждение, если некоторые значения входных данных не соответствуют ни одной из ветвей оператора switch.

4. Удалите строку с `_ => 0.0,`, чтобы недопустимые значения не совпадали.

5. Введите `dotnet patterns.cs`, чтобы просмотреть результаты.

   Компилятор сообщает предупреждение. Тестовые данные допустимы, поэтому программа работает. Однако любые недопустимые данные могут привести к сбою во время выполнения.

Шаблоны типов

Для завершения работы с этим руководством изучите еще один стандартный блок для сопоставления шаблонов: *типовой шаблон*. *Шаблон типа* проверяет выражение во время выполнения, чтобы узнать, является ли он указанным типом. С выражением `is` или `switch` выражением можно использовать тест типа. Измените текущий пример двумя способами. Во-первых, вместо кортежа создайте типы записей `Deposit` и `Withdrawal`, которые представляют транзакции.

1. Добавьте следующие объявления в конце файла кода:

   C#

   ```
   public record Deposit(double Amount, string description);
   public record Withdrawal(double Amount, string description);
   ```

2. Добавьте этот метод непосредственно перед объявлением `TransactionType` перечисления. Он анализирует текст и возвращает ряд записей:

   C#

   ```
   static IEnumerable<object?> TransactionRecordType(string inputText)
   {
       var reader = new StringReader(inputText);
       string? line;
       while ((line = reader.ReadLine()) is not null)
       {
           string[] parts = line.Split(',');

           string? transactionType = parts[0]?.Trim();
           if (double.TryParse(parts[1].Trim(), out double amount))
           {
               // Update the balance based on transaction type
               if (transactionType?.ToUpper() is "DEPOSIT")
                   yield return new Deposit(amount, parts[2]);
               else if (transactionType?.ToUpper() is "WITHDRAWAL")
                   yield return new Withdrawal(amount, parts[2]);
           }
   ```

```
    }
}
```

3. Добавьте следующий код после последнего `foreach` цикла:

```C#
currentBalance = 0.0;

foreach (var transaction in TransactionRecordType(bankRecords))
{
    currentBalance += transaction switch
    {
        Deposit d => d.Amount,
        Withdrawal w => -w.Amount,
        _ => 0.0,
    };
    Console.WriteLine($" {transaction} => New Balance: {currentBalance}");
}
```

4. Введите `dotnet patterns.cs` в окне терминала, чтобы увидеть результаты. Эта окончательная версия проверяет входные данные по *muny*.

Сопоставление шаблонов предоставляет словарь для сравнения выражения с характеристиками. Шаблоны могут включать тип выражения, значения типов, значений свойств и сочетаний этих выражений. Сравнение выражений с шаблоном может быть более четким, чем несколько `if` сравнений. Вы изучили некоторые шаблоны, которые можно использовать для сопоставления выражений. Существует множество способов использования сопоставления шаблонов в приложениях. При изучении вы можете узнать больше о сопоставлении шаблонов в C# в следующих статьях:

- Сопоставление шаблонов в C#
- Изучение руководства по сопоставлению шаблонов
- Сценарий сопоставления шаблонов
- Система типов C# — Поймите типы, с которыми вы работали в этом руководстве.

Очистка ресурсов

GitHub автоматически удаляет пространство Codespace через 30 дней бездействия. Вы завершили все уроки в этой серии. Чтобы удалить ваш Codespace, откройте окно браузера и перейдите к вашим Codespaces. В окне отобразится список ваших codespaces. Выберите три точки (`...`) в записи для пространства кода учебника для обучения и выберите **удалить**.

Дальнейшие шаги

Вы выполнили все вводные руководства! Вот где идти дальше:

- Приложения на основе файлов — узнайте о команде `dotnet run` , которую вы использовали в этих руководствах.
- Основы C# — более глубокое представление о структуре программы, типах и объектно-ориентированном программировании.
- Что можно создать с помощью C# — ознакомьтесь с типами приложений, которые вы можете создать с помощью того, что вы узнали.
- Скачайте и установите пакет SDK для .NET 10 .
- Скачайте и установите Visual Studio Code .
- Скачайте и установите C# DevKit .

Last updated on 24.02.2026

Стратегия Аннотированного C#

> **Совет**
>
> Эта статья наиболее полезна для специалистов и студентов, которые хотят понять философию дизайна C#и как развивается язык. Если вы не знакомы с C#, начните с руководства по C# или начальным руководствам .

Мы будем и дальше развивать C# в соответствии с изменяющимися потребностями разработчиков и оставаться современным языком программирования. Мы будем активно и широко сотрудничать с командами, ответственными за библиотеки .NET, средства разработчика и поддержку рабочей нагрузки, при этом внимательно следить за тем, чтобы оставаться в духе языка. Признавая разнообразие доменов, в которых используется C#, мы будем предпочитать улучшения языка и производительности, которые помогут всем разработчикам или большинству разработчиков и обеспечить высокую приверженность обратной совместимости. Мы будем продолжать расширять возможности более широкой экосистемы .NET и увеличивать свою роль в будущем C#, сохраняя руководство по проектированию решений.

Практическое руководство по стратегии C#

Стратегия C# поможет нашим решениям об эволюции C# и эти заметки дают представление о том, как мы думаем о ключевых инструкциях.

> "Мы будем активно и широко внедрять инновации"

Сообщество C# продолжает расти, и язык C# продолжает развиваться в соответствии с потребностями и ожиданиями сообщества. Мы черпаем вдохновение из многих источников, чтобы выбрать функции, которые пользуются большим сегментом разработчиков C#, и обеспечивают согласованное улучшение производительности, удобочитаемости и производительности.

> быть осторожным, чтобы соответствовать духу языка

Мы оцениваем новые идеи в духе и истории языка C#. Мы приоритетим инновации, которые подходят для большинства существующих разработчиков C#.

> улучшения, которые принесут пользу всем или большинству разработчиков

Разработчики используют C# во всех рабочих нагрузках .NET. Разработчики создают веб-интерфейсы и серверные части, облачные собственные приложения и классические приложения с помощью C#. C# включает кроссплатформенные приложения. Мы сосредоточимся на новых функциях, которые оказывают наибольшее влияние напрямую или за счет улучшения общих библиотек. Разработка языковых функций включает интеграцию с нашими инструментами разработчика и ресурсами обучения.

> "высокая приверженность обратной совместимости"

Мы уважаем, что сегодня используется огромный объем кода C#. Любые потенциальные критические изменения тщательно рассматриваются в отношении масштаба и влияния дестабилизации на сообщество C#.

> поддержание ответственного управления

Разработка языка C# выполняется в открытом режиме с участием сообщества. Любой пользователь может предложить новые возможности C# в репозитории GitHub. Команда разработки языка принимает окончательные решения после взвешивания входных данных сообщества.

Last updated on 06.02.2026

Дорожная карта для разработчиков Java, изучающих C#

Если вы начинаете новую работу или присоединяетесь к команде, использующую C#, эта статья помогает быстро повысить эффективность работы. В нем описаны знакомые возможности Java и новые возможности C#.

C# и Java имеют множество сходств. По мере изучения C#вы можете применить большую часть знаний, которые у вас уже есть из программирования на Java:

1. ***Аналогичный синтаксис***: Java и C# находятся в семействе языков C. Это сходство означает, что вы уже можете читать и понимать C#. Существуют некоторые различия, но большая часть синтаксиса совпадает с Java и C. Фигурные скобки и точка с запятой знакомы. Операторы управления, такие как `if`, `else`, `switch`, одинаковы. Цикловые операторы `for`, `while` и `do`...`while` одинаковы. Одни и те же ключевые слова для `class` и `interface` присутствуют в обоих языках. Модификаторы доступа от `public` до `private` совпадают. Даже многие встроенные типы используют одни и те же ключевые слова: `int`, `string` и `double`.
2. ***Объектно-ориентированной парадигмы***: Как Java, так и C# являются объектно-ориентированными языками. Понятия полиморфизма, абстракции и инкапсуляции применяются на обоих языках. Оба добавили новые конструкции, но основные функции остаются актуальными.
3. ***Строго типизированные***: Java и C# являются строго типизированными языками. Тип данных переменных объявляется явно или неявно. Компилятор обеспечивает безопасность типов. Компилятор перехватывает ошибки, связанные с типами в вашем коде, ещё до его выполнения.
4. ***Кроссплатформенные***: Java и C# являются кроссплатформенными. Вы можете запускать средства разработки на любимой платформе. Приложение может работать на нескольких платформах. Платформа разработки не обязана совпадать с вашей целевой платформой.
5. ***Обработка*** исключений: как Java, так и C# вызывают исключения для указания ошибок. Оба блока используются `try` - `catch` - `finally` для обработки исключений. Классы Exception имеют схожие имена и иерархии наследования. Одно из различий заключается в том, что C# не имеет концепции проверенных исключений. Любой метод может (в теории) вызвать любое исключение.
6. ***Стандартные библиотеки***: среда выполнения .NET и стандартная библиотека Java (JSL) поддерживают распространенные задачи. Оба имеют обширные экосистемы для других пакетов с открытым исходным кодом. В C# диспетчер пакетов — NuGet. Это аналогично Maven.

7. ***Сборка*** мусора. Оба языка используют автоматическое управление памятью с помощью сборки мусора. Среда выполнения освобождает память от объектов, на которые нет ссылок. Одно из различий заключается в том, что C# позволяет создавать типы значений в виде `struct` типов.

Синтаксис на первый взгляд

В следующих примерах показаны несколько распространенных шаблонов параллельно. Эти сравнения не являются исчерпывающими, но они позволяют быстро почувствовать различия синтаксиса.

Объявление переменных и вывод типов:

Java

```
// Java
var name = "Hello";
final int count = 5;
```

C#

```
// C#
var name = "Hello";
const int count = 5;
```

Интерполяция строк:

Java

```
// Java
var message = "Hello, " + name + "! Count: " + count;
```

C#

```
// C#
var message = $"Hello, {name}! Count: {count}";
```

Дополнительные сведения: интерполяция строк

Лямбда-выражения:

Java

```java
// Java
list.stream().filter(x -> x > 5).collect(Collectors.toList());
```

C#

```csharp
// C#
var result = list.Where(x => x > 5).ToList();
```

Дополнительные сведения: обзор LINQ

Обработка NULL:

Java

```java
// Java
String value = optional.orElse("default");
```

C#

```csharp
// C#
string value = input ?? "default";
```

Дополнительные сведения: ссылочные типы, допускающие значение NULL

Что знакомо

Вы можете эффективно работать в C# практически сразу из-за сходства. По мере изучения вы узнаете о функциях и идиомах в C#, которые недоступны в Java.

1. Сопоставление шаблонов: сопоставление шаблонов включает краткие условные операторы и выражения на основе формы сложных структур данных. `is` оператор проверяет, соответствует ли переменная некоторому шаблону. Выражение на основе шаблонов предоставляет широкий синтаксис для проверки переменной и принятия решений на основе её характеристик.
2. ***Интерполяция строк и необработанные строковые литералы: интерполяция*** строк позволяет вставлять вычислянные выражения в строку, а не использовать позиционные идентификаторы. Необработанные строковые литералы позволяют свести к минимуму escape-последовательности в тексте.
3. *Типы, допускающие значение NULL и не допускающие значение NULL: C# поддерживает **типы** значений, допускающие значение NULL, и ссылочные типы*, допускающие значение NULL, путем добавления `?` суффикса к типу. Для типов, допускающих значение NULL, компилятор предупреждает вас, если вы не

проверяете на null прежде чем разыменовать выражение. Для типов, не допускающих значение NULL, компилятор предупреждает вас, если вы можете назначить `null` значение этой переменной. Ссылочные типы, не допускающие значения NULL, сводят к минимуму ошибки программирования, которые вызывают ошибку System.NullReferenceException.

4. ***Расширения***. В C#можно создавать элементы, *расширяющие* класс или интерфейс. Расширения обеспечивают новое поведение для типа из библиотеки или всех типов, реализующих заданный интерфейс.
5. ***LINQ: интегрированный язык запросов (LINQ)*** предоставляет общий синтаксис для запроса и преобразования данных независимо от его хранилища.
6. ***Локальные функции: в C#можно вложить функции*** в методы или другие локальные функции. Локальные функции предоставляют еще один слой инкапсуляции.

> **Совет**
>
> Дополнительные сведения о системе типов C#, включая `struct` vs. `class`, записи и интерфейсы, см. в обзоре **системы типов** в разделе "Основы".

В C# существуют другие функции, которые не находятся в Java. Функции, такие как async и await, моделируют асинхронные операции в последовательном синтаксисе. Оператор using автоматически освобождает ресурсы, не относящиеся к памяти.

Существуют также некоторые аналогичные функции между C# и Java, которые имеют тонкие, но важные различия:

1. ***Свойства*** и индексаторы: свойства и ***индексаторы*** (обработка класса, например массива или словаря), поддерживают язык. В Java существуют соглашения об именовании для методов, начинающихся с `get` и `set`.
2. ***Записи***: В C# записи могут быть либо `class` (ссылочные), либо `struct` (значимые) типы. Записи C# могут быть неизменяемыми, но не обязательно быть неизменяемыми.
3. ***Кортежи*** имеют различный синтаксис в C# и Java.
4. ***Атрибуты похожи на аннотации*** Java.

Наконец, существуют функции языка Java, недоступные в C#:

1. ***Проверенные*** исключения: в C#любой метод теоретически может вызвать любое исключение.
2. ***Проверенная ковариация*** массива: в C#массивы не являются безопасными ковариантными. Если вам нужны ковариантные структуры, следует использовать универсальные классы и интерфейсы коллекции.

В целом, обучение C# для разработчика, опытного в Java, должно быть гладким. C# имеет достаточно знакомых идиом, чтобы вы могли продуктивно работать, пока изучаете новые идиомы.

Дальнейшие шаги

- Обзор C#: ознакомьтесь с общими сведениями обо всех функциях C#.
- Учебники для начинающих: Учитесь C# шаг за шагом с помощью интерактивных уроков.
- Что можно создать с помощью C#: изучите типы приложений, которые можно создать с помощью C#.
- Основы C#: более подробное представление о системе типов, объектно-ориентированном программировании и т. д.

Last updated on 24.02.2026

Дорожная карта для разработчиков JavaScript и TypeScript, изучающих C#

Если вы присоединяетесь к команде, которая использует C#, или изучаете его для разработки серверной или полного стека, эта статья поможет вам быстро начать работать эффективно. Он выделяет знакомые возможности JavaScript и TypeScript и новые возможности C#.

C#, TypeScript и JavaScript являются членами семейства языков C. Сходство между языками помогает быстро повысить эффективность работы в C#.

1. ***Аналогичный синтаксис***: JavaScript, TypeScript и C# находятся в семействе языков C. Это сходство означает, что вы уже можете читать и понимать C#. Существуют некоторые различия, но большая часть синтаксиса совпадает с JavaScript и C. Фигурные скобки и точка с запятой знакомы. Операторы управления, такие как `if`, `else` и `switch` одинаковы. Операторы циклов `for`, `while` и `do...while` одинаковы. Одни и те же ключевые слова для `class` и `interface` находятся как в C#, так и в TypeScript. Модификаторы доступа в TypeScript и C#, от `public` до `private`, одинаковы.
2. ***Маркер*** `=>`: все языки поддерживают облегченные определения функций. В C#они называются *лямбда-выражениями*. В JavaScript они обычно называются *функциями со стрелками*.
3. ***иерархии функций***: все три языка поддерживают локальные функции, которые являются функциями, определенными в других функциях.
4. ***Async / Await***: все три языка используют одинаковые `async` и ключевые слова `await` для асинхронного программирования.
5. ***сборка мусора***: все три языка основаны на сборщике мусора для автоматического управления памятью.
6. ***модель событий***: синтаксис event C#аналогичен модели JavaScript для событий объектной модели документа (DOM).
7. ***Диспетчер пакетов***: NuGet — это наиболее распространенный диспетчер пакетов для C#, аналогичный npm для приложений JavaScript. Библиотеки C# предоставляются в сборках.

Синтаксис на первый взгляд

В следующих примерах показаны несколько распространенных шаблонов параллельно. Эти сравнения не являются исчерпывающими, но они позволяют быстро почувствовать различия синтаксиса.

Заметки типа:

TypeScript

```typescript
// TypeScript
let name: string = "Hello";
let count: number = 5;
```

C#

```csharp
// C#
string name = "Hello";
int count = 5;
```

Async / await:

TypeScript

```typescript
// TypeScript
async function fetchData(): Promise<string> {
    const response = await fetch(url);
    return await response.text();
}
```

C#

```csharp
// C#
async Task<string> FetchDataAsync() {
    var response = await client.GetAsync(url);
    return await response.Content.ReadAsStringAsync();
}
```

Дополнительные сведения см. в статье "Асинхронное программирование".

Классы:

TypeScript

```typescript
// TypeScript
class Point {
    constructor(public x: number, public y: number) {}
}
```

C#

```
// C#
record Point(int X, int Y);
```

Дополнительные сведения см. в разделе "Записи".

Сопоставление с шаблоном:

TypeScript

```
// TypeScript - manual type checking
if (typeof value === "string") { /* ... */ }
```

C#

```
// C# - pattern matching
if (value is string s) { /* use s */ }
```

Дополнительные сведения см. в разделе "Сопоставление шаблонов".

Различия в модели среды выполнения

Несмотря на то, что C# и JavaScript выглядят аналогичным образом в синтаксисе, они выполняются очень по-разному:

- JavaScript выполняется в среде выполнения, например версии 8, и использует цикл событий для обработки асинхронной работы.
- C# выполняется в среде выполнения .NET (CLR), где код компилируется на промежуточный язык (IL), а затем выполняется с помощью JIT или компиляции AOT.

Новые возможности для вас в C#

По мере изучения C#вы сталкиваетесь с понятиями, которые не являются частью JavaScript. Некоторые из этих понятий могут быть знакомы с вами, если вы используете TypeScript:

1. ***система типов C#***: C# — строго типизированный язык. Каждая переменная имеет тип, и этот тип не может измениться. Вы определяете типы `class` или `struct`. Вы можете определить определения interface, определяющие поведение, реализованное другими типами. TypeScript включает многие из этих понятий, но так как TypeScript построен на JavaScript, система типов не так строга.

2. ***шаблонное сопоставление***: шаблонное сопоставление позволяет создавать лаконичные условные операторы и выражения, основанные на форме сложных структур данных. Выражение is проверяет, соответствует ли переменная какому-либо шаблону. Выражение на основе шаблона switch предоставляет широкий синтаксис для проверки переменной и принятия решений на основе его характеристик.
3. ***интерполяция строк*** и ***необработанные строковые литералы***: интерполяция строк позволяет вставлять вычислянные выражения в строку, а не использовать позиционные идентификаторы. Сырьевые строковые литералы позволяют уменьшить необходимость в escape-последовательностях в тексте.
4. ***Типы, допускающие значение NULL и не*** допускающие значение NULL: C# поддерживает *типы значений, допускающие* значение NULL, и *ссылочные типы, допускающие значение NULL* , путем `?` добавления суффикса к типу. Для типов с возможностью значения NULL компилятор предупреждает вас, если вы не проверяете наличие `null` перед разыменованием выражения. Для типов, не допускающих значение NULL, компилятор предупреждает вас, если вы можете присвоить значение `null` этой переменной. Эти функции могут свести к минимуму возникновение System.NullReferenceExceptionв вашем приложении. Синтаксис может быть знаком благодаря использованию примера `?` в TypeScript для необязательных свойств.
5. ***LINQ***: языковой интегрированный запрос (LINQ) предоставляет общий синтаксис для запроса и преобразования данных независимо от его хранилища.

> **Совет**
>
> Дополнительные сведения о системе типов C#, включая `class` и `struct`, универсальные шаблоны и интерфейсы, см. в разделе **Обзор системы типов** в разделе "Основы".

Когда вы узнаете больше, другие различия становятся очевидными, но многие из этих различий меньше по масштабу.

Некоторые знакомые функции и идиомы из JavaScript и TypeScript недоступны в C#:

1. ***динамических типов***: C# использует статическую типизацию. Объявление переменной включает тип, и этот тип не может измениться. В C# есть тип dynamic, который предоставляет привязку среды выполнения.
2. ***прототипное наследование***: наследование C# является частью объявления типа. Объявление C# `class` указывает любой базовый класс. В JavaScript свойство `__proto__` задается для установки базового типа на любом экземпляре.

3. ***интерпретируемый язык***: код C# необходимо скомпилировать перед его запуском. Код JavaScript можно запускать непосредственно в браузере.

Кроме того, несколько дополнительных функций TypeScript недоступны в C#:

1. ***Типы союзов***: начиная с C# 15, C# поддерживает типы союзов. Объединение определяет замкнутый набор именованных вариантов, которые может представлять значение, и компилятор гарантирует исчерпывающее сопоставление с использованием шаблонов по этим вариантам.
2. ***декораторы***: C# не имеет декораторов. Некоторые распространенные декораторы, такие как `@sealed`, являются зарезервированными ключевыми словами в C#. Другие распространенные декораторы могут иметь соответствующие атрибуты. Для всех остальных декораторов вы можете создать собственные атрибуты.
3. ***более простительный синтаксис***: компилятор C# анализирует код более строго, чем требует JavaScript.

Если вы создаете веб-приложение, попробуйте использовать Blazor для создания приложения. Blazor — это веб-платформа с полным стеком, созданная для C#. Компоненты Blazor могут выполняться на сервере, как сборки .NET или на клиенте с помощью WebAssembly. Blazor поддерживает взаимодействие с любимыми библиотеками JavaScript или TypeScript.

Дальнейшие шаги

- Обзор C#: ознакомьтесь с общими сведениями обо всех функциях C#.
- Учебники для начинающих: Учитесь C# шаг за шагом с помощью интерактивных уроков.
- Что можно создать с помощью C#: изучите типы приложений, которые можно создать с помощью C#.
- Основы C#: более подробное представление о системе типов, объектно-ориентированном программировании и т. д.

Last updated on 26.05.2026

Дорожная карта для разработчиков на Python, изучающих C#

Если вы переходите с Python на C# для новой роли или проекта, эта статья помогает быстро повысить эффективность работы. Он выделяет знакомые возможности Python и отличия в C#.

C# и Python используют аналогичные понятия. Эти знакомые конструкции помогут вам узнать C#, когда вы уже знаете Python.

1. ***Объектно-ориентированное***: Python и C# — это объектно-ориентированные языки. Все понятия, связанные с классами в Python, применяются в C#, даже если синтаксис отличается.
2. ***Кроссплатформенный***: Python и C# являются кроссплатформенными языками. Приложения, написанные на любом языке, могут работать на многих платформах.
3. ***Сборка*** мусора. Оба языка используют автоматическое управление памятью с помощью сборки мусора. Среда выполнения освобождает память от объектов, на которые нет ссылок.
4. ***Строго типизированные***: Python и C# являются строго типизированными языками. Приведение типов не происходит неявно. Ниже описаны различия, так как C# статически типизированный, в то время как Python динамически типируется.
5. ***Async / Await***: Возможности `async` и `await` в Python были непосредственно вдохновлены поддержкой `async` и `await` в C#.
6. **Сопоставление шаблонов**: Сопоставление шаблонов и выражений в Python похоже на сопоставление `match` и выражений C#. Они используются для проверки сложного выражения данных, чтобы определить, соответствует ли он шаблону.
7. ***Ключевые слова инструкции***: Python и C# используют множество ключевых слов, таких как `if`, `else`, `while` и `for` многие другие. Хотя не весь синтаксис одинаков, есть достаточно сходства, чтобы вы могли понять C#, если знаете Python.

Синтаксис на первый взгляд

В следующих примерах показаны несколько распространенных шаблонов параллельно. Эти сравнения не являются исчерпывающими, но они позволяют быстро почувствовать различия синтаксиса.

Заметки типа:

Python

```python
# Python
name: str = "Hello"
count: int = 5
```

C#

```csharp
// C#
string name = "Hello";
int count = 5;
```

Фильтрация списка (понимание и LINQ):

Python

```python
# Python
result = [x for x in items if x > 5]
```

C#

```csharp
// C#
var result = items.Where(x => x > 5).ToList();
```

Дополнительные сведения: обзор LINQ

Область блокировки (отступы и фигурные скобки):

Python

```python
# Python
if count > 0:
    print("positive")
```

C#

```csharp
// C#
if (count > 0)
{
    Console.WriteLine("positive");
}
```

Определение класса:

Python

```python
# Python
class Point:
```

```
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y
```

C#

```
// C#
record Point(int X, int Y);
```

Дополнительные сведения: Записи

Основные отличия

При изучении C#вы узнаете, что эти важные понятия отличаются от C# от Python:

1. ***Отступы и маркеры***: в Python новые линии и отступы являются элементами syntactic первого класса. В C# пробелы не имеют значения. Токены, такие как `;`, разделяют операторы, а другие токены, такие как `{` и `}`, контролируют область видимости блоков для `if` и других операторов блоков. Однако для удобства чтения большинство стилей программирования (включая стиль, используемый в этих документах), используют отступ для укрепления областей блоков, объявленных `{` и `}`.
2. ***Статическая типизация***: в C#объявление переменной включает его тип. Переназначение переменной объекту другого типа создает ошибку компилятора. В Python тип может измениться при переназначении.
3. ***Типы***, допускающие значение NULL: переменные C# могут иметь *значение* NULL или *не допускающие* значение NULL. Тип, не допускающий значения null, — это тип, который не может принимать значение null (или ничто). Он всегда ссылается на допустимый объект. Тип nullable может ссылаться на допустимый объект или NULL.
4. ***LINQ: ключевые слова выражения запроса, составляющие интегрированный язык запросов (LINQ)***, не являются ключевыми словами в Python. Однако библиотеки Python, такие как `itertools`, `more-itertools` и `py-linq` предоставляют аналогичные функциональные возможности.
5. ***Обобщения***: обобщения C# используют статическую типизацию C# для утверждения об аргументах, предоставленных для параметров типа. Универсальному алгоритму может требоваться указать ограничения, которым должен удовлетворять тип аргумента.

> **Совет**

Дополнительные сведения о системе типов C#, включая `class` и `struct`, универсальные шаблоны и интерфейсы, см. в разделе **Обзор системы типов** в разделе "Основы".

Наконец, некоторые функции Python недоступны в C#:

1. ***Структурная (утки) типизация***: в C# типы имеют имена и объявления. Кроме кортежей, типы, имеющие ту же структуру, не взаимозаменяемы.
2. ***REPL***: C# не имеет интерактивной оболочки (REPL) для быстрого создания прототипов решений.
3. ***Значительные пробелы***: необходимо правильно использовать фигурные скобки `{` и `}` для обозначения области видимости блока.

Изучение C#, если вы знаете Python, является легким процессом. Языки имеют аналогичные понятия и аналогичные идиомы для использования.

Дальнейшие шаги

- Обзор C#: ознакомьтесь с общими сведениями обо всех функциях C#.
- Учебники для начинающих: Учитесь C# шаг за шагом с помощью интерактивных уроков.
- Что можно создать с помощью C#: изучите типы приложений, которые можно создать с помощью C#.
- Основы C#: более подробное представление о системе типов, объектно-ориентированном программировании и т. д.

Last updated on 24.02.2026

Общая структура программы C#

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вам помогут шаг за шагом написать вашу первую программу на C#, перед тем, как изучить структуру программы.
>
> **Есть опыт на другом языке?** Вам может потребоваться пропустить раздел "Начало работы " для синтаксиса C#, а затем вернуться сюда.

Вы создаете программы C# из этих основных стандартных блоков: пространства имен упорядочивают типы, типы (классы, структуры, интерфейсы, перечисления и делегаты) определяют поведение и данные, а также операторы и выражения выполняют работу во время выполнения. Структура точки входа зависит от выбранного стиля приложения.

Выбор стиля приложения

При создании программы C# сделайте два независимых выбора о том, как его структурировать:

- **На основе файлов или на основе проектов?**
  - Приложение на основе файлов запускается из одного `.cs` файла без файла проекта.
  - Приложение на основе проекта использует `.csproj` файл и может охватывать несколько исходных файлов.
- **Операторы верхнего уровня или `Main` метод?**
  - Инструкции верхнего уровня позволяют писать исполняемый код непосредственно в верхней части файла.
  - Метод `Main` упаковывает точку входа в явный статический метод.

Оба приложения — основанные на проектах и на файлах — поддерживают любой стиль точки входа.

Приложения на основе файлов и приложения на основе проектов

Начиная с C# 14 и .NET 10, *приложения на основе файлов* позволяют запускать программу, содержащуюся в одном `*.cs` файле без файла проекта. Сохраните

следующий код в файле с именем `hello-world.cs` и выполните его с помощью `dotnet run hello-world.cs` или `dotnet hello-world.cs`.

C#

```
#!/usr/bin/env dotnet
Console.WriteLine("Hello, World!");
```

> ⓘ **Примечание.**
>
> Строка `#!` позволяет операционным системам Unix выполнять файл напрямую (например, `./hello-world.cs`). Для этого требуется задать разрешение *на выполнение* (`chmod +x <file>`).

Приложения на основе файлов поддерживают весь синтаксис C# и могут использовать директивы препроцессора для настройки системы сборки. Используйте приложения на основе файлов для небольших служебных программ командной строки, прототипов и экспериментов. Приложение на основе файлов состоит из одного файла в каталоге:

```
my-app/
└── hello-world.cs
```

Приложения на основе проектов используют `.csproj` и команды .NET CLI, `dotnet new` `dotnet build` и `dotnet run` рабочий процесс. Выберите приложения на основе проекта, когда программа охватывает несколько файлов или требует точной настройки сборки. Приложение на основе проекта включает файл проекта вместе с одним или несколькими исходными файлами:

```
my-app/
├── my-app.csproj
├── Program.cs
├── Models/
│   └── Person.cs
└── Services/
    └── GreetingService.cs
```

Если приложение на основе файлов растет, его можно легко преобразовать в приложение на основе проекта. Запустите dotnet project convert , чтобы создать файл проекта из существующего исходного файла.

Если вы знаете, что ваше приложение нуждается в нескольких исходных файлах с самого начала, начните с приложения на основе проекта. Вы избегаете шага преобразования и можете упорядочить код в отдельные файлы сразу.

Операторы верхнего уровня и `Main` метод

С помощью инструкций верхнего уровня можно писать исполняемый код непосредственно в одном файле, не упаковывая его в класс и `Main` метод. Этот стиль используется по умолчанию при создании нового консольного приложения. `dotnet new console` В следующем примере показана современная программа C#, использующая инструкции верхнего уровня:

C#

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

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }
}
```

Только один файл в проекте может содержать инструкции верхнего уровня, а точка входа — первая строка текста программы в этом файле. При создании более крупных программ вы включаете дополнительные элементы программы.

Можно также определить явный статический Main метод в качестве точки входа программы:

C#

```
// A skeleton of a C# program using an explicit Main method
namespace YourNamespace;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Оба стиля точки входа работают с приложениями, основанными на файлах и проектах. Оба стиля поддерживают одни и те же функции.

Создание и запуск программ C#

C# — это *скомпилированный* язык. Для приложений на основе проекта используйте dotnet build команду для компиляции исходных файлов в двоичный пакет. Используйте dotnet run, чтобы выполнить сборку и запуск в один шаг. Интерфейс `dotnet` командной строки, включенный в пакет SDK для .NET, предоставляет множество средств для создания, сборки и управления проектами C#.

Для приложений `dotnet run hello-world.cs` на основе файлов компилирует и запускает один файл напрямую — файл проекта не требуется.

Выражения и инструкции

Если вы выполнили учебные пособия 'Начало работы', вы уже написали выражения и операторы. Каждая строка кода, введенная вами, была одной или другой (или то и другое). Теперь определим эти термины.

Выражения и операторы являются основными стандартными блоками программы C#. *Выражение* создает значение. *Оператор* выполняет действие и обычно заканчивается точкой запятой.

Ниже приведены выражения:

- `42` (литеральное значение)
- `x + y` (арифметическая операция)
- `Math.Max(a, b)` (вызов метода, который создает значение)
- `condition ? trueValue : falseValue` (условное выражение)
- `new Person("John")` (создание объекта)

Инструкция выполняет действие. Операторы управляют потоком программы, объявляют переменные или вызывают операции. Ниже приведены утверждения:

- `int x;` (инструкция объявления)
- `int x = 42;` (оператор объявления с инициализацией)
- `Console.WriteLine("Hello");` (оператор вызова метода)
- `if (condition) { /* code */ }` (условный оператор)
- `return result;` (оператор return)

Инструкции часто содержат выражения, а выражения могут вложиться в другие выражения. Например, следующий оператор объявления присваивает `f` результату выражения сложения. Это выражение добавления добавляет результаты двух выражений вызова метода:

C#

```
var maxResult = Math.Max(a, b) + Math.Max(c, d);
```

Подробные сведения об инструкциях см. в разделе "Инструкции". Сведения о элементах с выражением см. в разделе "Элементы с выражением".

Связанный контент

- Классы
- структуры
- пространства имен
- Интерфейсы
- перечисления
- делегаты
- Основные понятия в спецификации языка C#

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.03.2026

Пространства имен и директивы using

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Они вводят пространства имен и `using` директивы, когда вы начинаете писать свои первые программы.
>
> **Есть опыт на другом языке?** Пространства имен в C# работают похожим образом, как пакеты в Java или модули в Python. Перейдите к нужному синтаксису.

Объявления пространства имен и `using` директивы связаны с языковыми особенностями. Объявление пространства имен структурирует ваши типы данных. Пространство имен группирует связанные типы и предотвращает столкновения именования. Директива `using` позволяет вашей программе использовать эти типы по их упрощенным именам. Вам не нужно указывать полный путь к пространству имен при каждом использовании.

Вы уже использовали пространства имен в каждой программе C#, которую вы написали. Каждый тип .NET принадлежит пространству имен, и каждая `using` директива в верхней части файла ссылается на один. Например, `Console` и `Math` принадлежат к пространству имен `System`, поэтому их полные имена — это `System.Console` и `System.Math`. Типы коллекций, подобные `List<T>` и `Dictionary<TKey, TValue>`, принадлежащие `System.Collections.Generic`. `using` Одна директива для любого из этих пространств имен позволяет ссылаться на все его типы по простым именам. Вы пишете `List<T>` вместо того, чтобы `System.Collections.Generic.List<T>` везде, где вы его используете.

В этой статье содержатся дополнительные сведения о работе пространств имен и `using` директив, а также примеры шаблонов, которые вы уже обнаружили в библиотеках .NET.

Пространство имен содержит типы. Каждый тип .NET принадлежит пространству имен. Например, рассмотрим `System.Threading.Tasks.Task`: тип `Task` принадлежит пространству `System.Threading.Tasks` имен.

Рекомендуется группировать связанные или аналогичные типы в одном пространстве имен, и это то, что делает .NET с указанными типами. Пространство имен `System.Collections.Generic` содержит типы, связанные с коллекцией, а пространство имен `System.IO` содержит типы, связанные с работой с чтением и записью файлов, каталогов и данных. Пространство `System` имен содержит основные типы, такие как `Math`, `DateTime` и `Console`.

В следующем примере показано, как пространства имен работают вместе с `using` директивами в типичном файле C#:

C#

```
using System.Globalization;

namespace MyApp.Services;

class Greeter
{
    public string Greet(string name)
    {
        var culture = CultureInfo.CurrentCulture;
        return $"Hello, {name}! Culture: {culture.Name}";
    }
}
```

В предыдущем примере директива `using` означает, что вы можете использовать System.Globalization.CultureInfo по имени `CultureInfo` без необходимости указывать полное имя `System.Globalization.CultureInfo`. Директива `namespace` объявляет, что `Greeter` класс является частью `MyApp.Services` пространства имен. Полное имя `MyApp.Services.Greeter`.

Объявления пространства имен

Объявление пространства имен присваивает ваши типы к названной группе. Каждый тип, который вы записываете, должен принадлежать пространству имен. Имя пространства имен обычно отражает структуру папок проекта. Например, типы в папке `Services/Payments` часто принадлежат пространству имен `MyApp.Services.Payments`.

Пространства имен используют оператор `.` для выражения иерархии, например `System.Collections.Generic`. Имена пространств имен в C# должны быть допустимыми именами идентификаторов.

Пространства имен с областью действия файла

Используйте синтаксис *с областью действия файла* , если все типы в файле принадлежат одному пространству имен. После объявления пространства имен добавьте точку с запятой; это будет применяться ко всему файлу. Вам не нужны дополнительные отступы или фигурные скобки.

C#

```
namespace MyApp.Models;

class Customer
{
    public required string Name { get; init; }
    public string? Email { get; init; }

    public override string ToString() => $"{Name} ({Email ?? "no email"})";
}
```

Пространства имен с областью действия файлов сокращают вложенные файлы и упрощают чтение файлов. В каждом файле может быть только одно пространство имен, ограниченное файлом.

> **Совет**
>
> Используйте пространства имен, ограниченные файлом, в новом коде. Большинство шаблонов .NET и анализаторов кода рекомендуют этот стиль.

Блочные пространства имен

Используйте синтаксис *блочной области видимости*, если необходимо объявить несколько пространств имен в одном файле. Этот стиль добавляет дополнительный уровень отступа.

> **Внимание**
>
> Считается плохой практикой объявлять более одного пространства имен в одном файле. Более распространенный сценарий — использовать пространства имен *с областью действия файла* .

Следующий фрагмент кода является примером *блочного* пространства имен:

C#

```
namespace MyApp.Models
{
    class Product
    {
        public required string Name { get; init; }
        public decimal Price { get; init; }

        public override string ToString() => $"{Name}: {Price:C}";
```

```
    }
}
```

Директивы using

Без директивы необходимо ссылаться на каждый тип по `using`, полный путь пространства имен и имя типа:

C#

```
static void ShowFullyQualified()
{
    // Without a using directive, use the fully qualified name:
    System.Console.WriteLine("Hello from fully qualified name!");
}
```

На верхней части файла присутствует директива `using`, которая импортирует пространство имен, позволяя использовать его типы по их простым именам:

C#

```
static void ShowShortName()
{
    // With 'using System;' (or implicit usings enabled), use the short name:
    Console.WriteLine("Hello from short name!");
}
```

Дополнительные сведения см. в директивеusing.

Глобальные директивы using

Если вы пишете одни и те же `using` директивы в каждом файле, директивы *global using* позволяют объявить их один раз для всего проекта. Поместите их в любой файл. Многие команды создают выделенный `GlobalUsings.cs` файл:

C#

```
global using System.Text;
global using System.Text.Json;
```

После объявления глобального использования каждый файл в проекте может ссылаться на типы из этого пространства имен с помощью простых имен без дополнительной `using` директивы.

Неявное использование

SDK для .NET автоматически создает глобальные директивы using для наиболее распространенных пространств имен в зависимости от типа вашего проекта. Включите неявное использование, установив `<ImplicitUsings>enable</ImplicitUsings>` в файле проекта. Например, проект консольного приложения автоматически импортирует `System`, `System.Collections.Generic`, `System.IO`, `System.Linq`, `System.Threading` и `System.Threading.Tasks`. Текущий пакет SDK включает `ImplicitUsings` при создании нового проекта с помощью `dotnet new`.

См. раздел «Неявные директивы using» для получения дополнительной информации.

Статические директивы using

Директива `static using` импортирует статические элементы типа, чтобы их можно было вызывать без префикса имени типа:

C#

```
using static System.Math;

namespace MyApp.Utilities;

class CircleCalculator
{
    public static double CalculateArea(double radius) => PI * Pow(radius, 2);

    public static double CalculateCircumference(double radius) => 2 * PI * radius;
}
```

Статические использования хорошо работают для служебных классов, таких как Math и Console, которые вы часто вызываете.

Псевдонимы типа и пространства имен

Псевдоним `using` создает сокращенное имя для типа или пространства имен. Псевдонимы полезны для длинных общих типов, разрешения конфликтов именования и повышения удобочитаемости.

C#

```
using CustomerList = System.Collections.Generic.List<MyApp.Models.Customer>;

namespace MyApp.Services;
```

```
class CustomerService
{
    public CustomerList GetTopCustomers()
    {
        CustomerList customers = [new() { Name = "Alice" }, new() { Name = "Bob"
}];
        return customers;
    }
}
```

Начиная с C# 12, можно псевдонимить любой тип, включая кортежи и типы указателей:

C#

```
using Point = (double X, double Y);

namespace MyApp.Geometry;

class Shape
{
    public static double Distance(Point a, Point b)
    {
        var dx = a.X - b.X;
        var dy = a.Y - b.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }
}
```

Для более продвинутых сценариев, в которых две сборки определяют одно и то же полное имя типа, используйте extern alias для дизамбигуации между ними.

Связанный контент

- using директива (языковая справка)
- Пространства имен в спецификации языка C#

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.03.2026

Директивы препроцессора

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Директивы препроцессора не потребуются сразу. Сначала сосредоточьтесь на руководствах по началу работы и вернитесь сюда, когда для проектов требуется условная компиляция или конфигурация сборки.
>
> **Есть опыт на другом языке?** Если вы знакомы с `#ifdef` C/C++ или условной компиляцией на других языках, директивы препроцессора C# работают аналогично. Перейдите к нужному синтаксису.

Директивы препроцессора C# сообщают компилятору, какой код следует включать, исключать или обрабатывать по-разному при сборке приложения. Это руководство может изменить итоговую программу. Директивы препроцессора всегда начинаются с `#` и должны отображаться в собственной строке (игнорируя ведущие пробелы). После директивы можно добавить конечный комментарий. Хотя языковой справочник документирует все доступные директивы, три группы охватывают повседневное использование:

- **Приложения на основе файлов** (`#:`) — настройка файловых приложений.
- **Условная компиляция** (`#if` / `#elif` / `#else` / `#endif`) — включение или исключение кода на основе конфигурации сборки или целевой платформы.
- **Подавление предупреждений** (`#pragma warning`) — подавление или восстановление определенных предупреждений компилятора.

Директивы приложений, основанные на файлах

Начиная с C# 14 , приложения на основе файлов используют две дополнительные директивы:

- `#!` — строка *shebang* , которая позволяет выполнять файл непосредственно в Unix (например, `./Program.cs`). Для этого требуется разрешение *на выполнение* для файла (`chmod +x <file>`).
- `#:` — директивы build-system, которые настраивают пакеты, параметры пакета SDK и другие параметры для однофайловых программ.

Используйте `#:package` для добавления пакета NuGet. Например, следующее приложение на основе файлов использует пакет `Spectre.Console` для отображения стилизованного

вывода.

C#

```
#!/usr/bin/env dotnet
#:package Spectre.Console@*

AnsiConsole.MarkupLine("[bold green]Hello[/] from a file-based app!");
```

Вы можете указать точную версию с помощью `@`, и использовать `@*` для извлечения последней версии. Добавьте несколько директив для включения дополнительных `#:package` пакетов:

C#

```
#:package Serilog@3.1.1
```

Другие `#:` директивы позволяют ссылать на проекты, задавать свойства MSBuild или изменять пакет SDK:

C#

```
#:project ../SharedLibrary/SharedLibrary.csproj
#:property PublishAot=false
#:sdk Microsoft.NET.Sdk.Web
```

Полный список директив см. в разделе "Приложения на основе файлов " и справочник по языку.

Условная компиляция

Используйте `#if`, `#elif` `#else` и `#endif` для включения или исключения кода в зависимости от того, определен ли символ. Наиболее распространенными символами являются `DEBUG` (автоматически заданные для отладочных сборок) и целевые символы платформы, такие как `NET10_0_OR_GREATER`:

C#

```
static void ConfigureLogging()
{
#if DEBUG
    Console.WriteLine("Debug logging enabled — verbose output active.");
#else
    Console.WriteLine("Release logging — errors only.");
```

```
#endif
}
```

Система сборки определяет `DEBUG` символ при сборке в конфигурации Debug. Вам не нужно определять его самостоятельно. Символы целевой платформы, такие как `NET10_0_OR_GREATER` и `NET8_0_OR_GREATER` позволяют писать код, который адаптируется к различным версиям .NET в проектах с несколькими целевыми объектами.

Символы можно объединить с логическими операторами: `&&` (и) `||` (или) и `!` (не):

C#

```
static void ShowPlatformInfo()
{
#if NET10_0_OR_GREATER
    Console.WriteLine("Running on .NET 10 or later.");
#elif NET8_0_OR_GREATER
    Console.WriteLine("Running on .NET 8 or 9.");
#else
    Console.WriteLine("Running on an older .NET version.");
#endif
}
```

Используйте `#define` в верхней части файла для определения собственных символов. Вы также можете определить символы для всего проекта с помощью DefineConstants свойства в файле проекта.

Подавление предупреждений

Используется `#pragma warning disable` для подавления определенных предупреждений компилятора и `#pragma warning restore` их повторного включения. Всегда определяйте подавление как можно более узко:

C#

```
    static void ProcessData()
    {
        try
        {
            // Some operation that might fail
            var data = File.ReadAllText("config.json");
            Console.WriteLine($"Config loaded: {data.Length} characters");
        }
#pragma warning disable CS0168 // Variable is declared but never used
        catch (FileNotFoundException ex)
#pragma warning restore CS0168
        {
```

```
            // Fall back to defaults — the exception details aren't needed here
            Console.WriteLine("Config file not found, using defaults.");
        }
    }
```

> **Совет**
>
> Всегда указывайте номер предупреждения, например `CS0168`, вместо отключения всех предупреждений. Этот подход сохраняет целевое подавление и дает понять , *почему* предупреждение подавляется.

Связанный контент

- Директивы препроцессора C# (справочник по языку)
- Приложения на основе файлов

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.03.2026

Организация программы

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы узнаете о программной организации естественно по мере роста проектов.
>
> **Есть опыт на другом языке?** Если вы знакомы с решениями и проектами в Visual Studio или создаете такие системы, как Maven или Cargo, эта статья сопоставляет эти понятия с .NET.

По мере роста приложения C# необходимо упорядочить код. .NET предоставляет иерархию организационных средств — решений, проектов, сборок, пространств имен и типов, которые совместно работают для обеспечения управления большими базами кода. Описанные здесь соглашения представляют широкий консенсус в сообществе .NET. Вы можете отклоняться по определенным причинам, но следуя этим соглашениям, код становится знакомым и понятным для других разработчиков .NET.

Иерархия организации

Упорядочение типичного приложения .NET на уровнях от самых широких до наиболее конкретных:

Развернуть таблицу

Уровень	Описание	Пример
Solution	Контейнер, который группит связанные проекты.	`MyApp.slnx`
Проект	Единица сборки, которая создает одну сборку.	`MyApp.Web.csproj`
Сборка	Скомпилированный `.dll` или `.exe`, созданный проектом.	`MyApp.Web.dll`
Пространство имен	Логическое группирование типов.	`MyApp.Web.Controllers`
Тип	Класс, структура, интерфейс, перечисление или делегат.	`OrderController`

Каждый уровень служит для разных целей. Решения упорядочивают рабочий процесс разработки. Проекты определяют, что компилируется вместе, и каждый проект создает

одну сборку. Сборки — это единица развертывания и управления версиями. Пространства имен предотвращают конфликт именования и упрощают поиск типов. Одна сборка может содержать несколько пространств имен, а одно пространство имен может охватывать несколько сборок. Типы определяют фактическое поведение и данные.

Проекты и сборки

Каждый проект компилируется в одну сборку: библиотеку классов или исполняемый файл. Начните с одного проекта для небольших приложений— не разбиваясь преждевременно. Основная причина создания отдельного проекта заключается в повторном использовании этого кода в нескольких приложениях. Помимо повторного использования, добавьте проекты, если у вас есть конкретная причина:

- **Совместное использование кода между приложениями** — извлечение общей логики в библиотеку классов, на которую ссылается несколько приложений.
- **Отдельные проблемы** — сохраняйте доступ к данным, бизнес-логику и уровни презентации независимо.
- **Управление зависимостями** — проект может использовать только типы из проектов, на которые он ссылается явным образом.

Один проект хорошо подходит для многих приложений. Сопротивляйтесь призыву создать отдельные проекты "только в случае". Вы всегда можете извлечь библиотеку позже, когда второму приложению требуется тот же код.

Сопоставьте пространства имен со структурой папок

Имена пространств имен должны соответствовать структуре папок проекта. При отображении пространства имен `MyApp.Services.Payments` вы знаете, что нужно искать папку `Services/Payments`, где находится исходный код типов, определенных в этом пространстве имен. Пакет SDK для .NET поддерживает это соглашение, и оно настолько широко распространено, что нарушение его активно сбивает с толку других разработчиков.

C#

```
// File: Services/OrderService.cs
// Namespace mirrors the folder path
using MyApp.Core;

namespace MyApp.Services;
```

```
public class OrderService
{
    public Order CreateOrder(string product, int quantity, decimal price) =>
        new() { ProductName = product, Quantity = quantity, UnitPrice = price };

    public string FormatSummary(Order order) =>
        $"{order.Quantity}x {order.ProductName} = {order.Total:C}";
}
```

Корневое пространство имен автоматически присваивается имени файла проекта. Типы в вложенных папках не получают вложенные пространства имен автоматически — пространство имен нужно объявлять явно в каждом файле, при этом всегда следите за их синхронизацией.

> **Совет**
>
> Можно изменить корневое пространство имен, установив `<RootNamespace>` в файле проекта.
>
> XML
>
> ```
> <Project Sdk="Microsoft.NET.Sdk">
>  <PropertyGroup>
>    <RootNamespace>MyCompany.MyApp</RootNamespace>
>  </PropertyGroup>
> </Project>
> ```

Упорядочение пространств имен по функциям, а не по типу

Группировать связанные типы в пространства имен по функциональным возможностям или по области ответственности. Поместите интерфейс, его реализации и вспомогательные типы вместе.

C#

```
// Good: group by feature
namespace MyApp.Payments;

public interface IPaymentProcessor
{
    bool ProcessPayment(decimal amount);
}

public class CreditCardProcessor : IPaymentProcessor
```

```
{
    public bool ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing credit card payment of {amount:C}");
        return true;
    }
}

public record PaymentResult(bool Success, string? TransactionId);
```

Организация на основе функциональных возможностей хранит все необходимое в одном месте, что упрощает навигацию и понимание кода.

Модификаторы доступа и сборки

Модификаторы доступа работают со структурой проекта и сборки для управления доступом.

- public — доступен из любой сборки, ссылающейся на эту сборку.
- internal — доступен только в одной сборке (по умолчанию для типов верхнего уровня).
- private, protected, private protected, protected internal — доступен на основе содержащего типа, сборки или производных типов.

`internal` По умолчанию для типов, которые не требуются другим проектам. Эта практика скрывает детали реализации и дает возможность рефакторинга, не нарушая работу потребителей. Особенно важно для общих библиотек:

C#

```
namespace MyApp.Inventory;

// Public — other projects can use this type
public class InventoryService
{
    public int GetStockLevel(string productName) =>
        StockDatabase.Lookup(productName);
}

// Internal — only visible within this assembly
internal static class StockDatabase
{
    private static readonly Dictionary<string, int> _stock = new()
    {
        ["Widget"] = 42,
        ["Gadget"] = 17
    };
```

```
    internal static int Lookup(string productName) =>
        _stock.GetValueOrDefault(productName);
}
```

Рекомендуемые методы

- **Назовите пространства имен последовательно.** Используйте `CompanyName.ProductName.Feature` в качестве шаблона именования. Например, укажите `Contoso.Inventory.Shipping`. Согласованное именование помогает разработчикам находить типы без поиска.
- **Держите проекты в центре внимания.** Каждый проект должен иметь единую четкую ответственность. Если проект обрабатывает слишком много несвязанных проблем, разделите его.
- **Используйте пространства имен с областью действия файла.** Синтаксис `namespace MyApp.Services;` уменьшает отступ и является рекомендуемым стилем. Используйте его во всем новом коде.
- **По умолчанию используется значение `internal`.** Помечайте только типы `public`, когда они являются необходимыми для других сборок. Вы всегда можете расширить доступ позже; сужение - это обратимое изменение.

Связанный контент

- Пространства имен и директивы using
- Сборки в .NET
- Пакеты SDK для проектов .NET

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.03.2026

Выражения верхнего уровня — программы без `Main` методов

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. В этих руководствах используются инструкции верхнего уровня, поэтому вы уже знакомы с основами.
>
> **Ищете альтернативный `Main` метод?** См. точку входа основного метода для явного подхода к методу `Main`.

Используйте *инструкции верхнего уровня* для новых приложений. С помощью инструкций верхнего уровня можно написать исполняемый код непосредственно в корне файла.

Ниже приведен файл *Program.cs*, который является полной программой C#:

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

При создании консольного приложения с помощью `dotnet new console` он использует инструкции верхнего уровня по умолчанию. Они хорошо работают для программ любого размера — от небольших служебных программ и функций Azure до полных приложений. Если у вас есть существующее приложение, использующее явный `Main` метод, его не нужно преобразовать. Оба стиля компилируются в эквивалентный код.

В следующих разделах описываются правила, которые определяют использование операторов верхнего уровня.

Правила точки входа

Приложение должно иметь только одну точку входа. Проект может иметь только один файл с операторами верхнего уровня, но он может иметь любое количество файлов исходного кода, которые не имеют инструкций верхнего уровня. Вы можете явно написать `Main` метод, но он не может функционировать как точка входа. В проекте с операторами верхнего уровня нельзя использовать -main параметр компилятора для выбора точки входа, даже если проект имеет один или несколько `Main` методов.

Компилятор генерирует метод, который служит в качестве точки входа программы для проекта с входными инструкциями верхнего уровня. Сигнатура метода зависит от того, содержат ли операторы верхнего уровня ключевое слово `await` или оператор `return`. В следующей таблице показано, как выглядит сигнатура метода, используя имя `Main` метода в таблице для удобства.

Развернуть таблицу

Код верхнего уровня содержит	**Неявная сигнатура `Main`**
`await` и `return`.	`static async Task<int> Main(string[] args)`
`await`	`static async Task Main(string[] args)`
`return`	`static int Main(string[] args)`
Ни `await`, ни `return`	`static void Main(string[] args)`

Начиная с C# 14, программы могут быть *файловыми приложениями*, где один файл содержит программу. Приложения *на основе файлов* выполняются с помощью команды `dotnet <file.cs>` или непосредственно с помощью имени файла в Unix (например, `./file.cs`). Для последнего требуется включить `#!/usr/bin/env dotnet` директиву в качестве первой строки и задать разрешение на выполнение (`chmod +x <file>`).

Директивы `using`

Для одного файла, содержащего инструкции верхнего уровня, `using` директивы должны быть первыми в этом файле, как показано в следующем примере:

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("The following arguments are passed:");

foreach (var arg in args)
{
    builder.AppendLine($"Argument={arg}");
}

Console.WriteLine(builder.ToString());

return 0;
```

Пространства имен и определения типов

Операторы верхнего уровня неявно находятся в глобальном пространстве имен. Файл с операторами верхнего уровня может также содержать пространства имен и определения типов, но они должны следовать за операторами верхнего уровня. Например:

C#

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

`args`

Высокоуровневые операторы могут ссылаться на переменную `args`, чтобы получить доступ к любым аргументам командной строки, переданным приложению при его запуске. Переменная `args` никогда не `null`, но ее `Length` равно нулю, если аргументы командной строки не заданы. Например:

C#

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
```

```
    Console.WriteLine("No arguments");
}
```

`await` и код выхода

Используется `await` для вызова асинхронного метода. Когда код верхнего уровня содержит `await`, компилятор создает точку входа, возвращающую объект `Task`. Среда выполнения отслеживает `Task` до завершения, поддерживая процесс активным, пока не завершатся все асинхронные задачи. Например:

C#

```
Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

Чтобы вернуть код выхода при завершении приложения, используйте инструкцию `return` . Компилятор создает точку входа, которая возвращается `Task<int>` , когда код содержит оба `await` и `return`, или `int` когда он содержит только `return`. Например:

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

Связанный контент

- Аргументы Main() и командной строки
- Общая структура программы C#
- Спецификация функций — инструкции верхнего уровня

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.03.2026

Main() и аргументы командной строки

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. В этих руководствах используются инструкции верхнего уровня, которые проще для новых приложений.
>
> **Работа с существующей базой кода?** Многие существующие приложения используют явный `Main` метод. В этой статье объясняется, как она работает и как эффективно использовать ее.

При запуске приложения C# среда выполнения вызывает `Main` метод. Метод `Main` — это точка входа приложения C#.

Программа C# может иметь только одну точку входа. Если у вас несколько классов с методом `Main` , при компиляции программы необходимо использовать параметр компилятора **StartupObject** , чтобы указать, какой `Main` метод служит точкой входа. Дополнительные сведения см. в разделе **StartupObject** (параметры компилятора C#). В следующем примере отображается число аргументов командной строки в качестве первого действия:

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        Console.WriteLine(args.Length);
    }
}
```

Обзор

Метод `Main` — это точка входа исполняемой программы. При запуске программы среда выполнения вызывает `Main` перед выполнением любого другого кода. Когда `Main` возвращается, программа заканчивается. Вы объявляете `Main` с помощью следующих правил:

- Необходимо объявить `Main` внутри класса или структуры. Включающее `class` может быть `static`.
- Аргумент `Main` должен иметь значение static.

- `Main` может иметь любой модификатор доступа.
- `Main` может возвращать `void`, `int`, `Task` или `Task<int>`.
- Если и только если `Main` возвращает `Task` или `Task<int>`, объявление `Main` может включать модификатор async. Это правило специально исключает `async void Main` метод.
- Метод `Main` можно объявить с параметром `string[]`, содержащим аргументы командной строки, или без него. При использовании Visual Studio для создания приложений Windows можно добавить параметр вручную или использовать GetCommandLineArgs() метод для получения аргументов командной строки. Параметры — аргументы командной строки без индексирования. В отличие от C и C++, имя программы не рассматривается как первый аргумент командной строки в `args` массиве, но это первый элемент GetCommandLineArgs() метода.

В этом списке показаны пермутации объявлений `Main`:

C#

```
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
static async Task Main() { }
static async Task<int> Main() { }
static async Task Main(string[] args) { }
static async Task<int> Main(string[] args) { }
```

В предыдущих примерах не указывается модификатор доступа, поэтому они неявно `private` по умолчанию. Можно указать любой явный модификатор доступа.

В следующей таблице перечислены все допустимые `Main` подписи и когда они используются.

Развернуть таблицу

`Main` декларация	Использует `args`	Содержит `await`	Возвращает код выхода
`static void Main()`	Нет	Нет	Нет
`static int Main()`	Нет	Нет	Да
`static void Main(string[] args)`	Да	Нет	Нет
`static int Main(string[] args)`	Да	Нет	Да

`Main` декларация	Использует `args`	Содержит `await`	Возвращает код выхода
`static async Task Main()`	Нет	Да	Нет
`static async Task<int> Main()`	Нет	Да	Да
`static async Task Main(string[] args)`	Да	Да	Нет
`static async Task<int> Main(string[] args)`	Да	Да	Да

Выберите простую подпись, которая соответствует вашим потребностям. Если аргументы командной строки не нужны, опустите `string[] args` параметр. Если вам не нужно возвращать код выхода, используйте `void` или `Task`. Если необходимо вызвать асинхронные методы, используйте `async` с возвращаемым типом `Task` или `Task<int>`.

Возвращаемые значения Main()

Когда вы возвращаетесь `int` или `Task<int>`, ваша программа может отправлять сведения о состоянии в другие программы или скрипты, которые запускают исполняемый файл. Возвращаемое значение обычно означает успешность, а ненулевое значение `0` означает, что возникает ошибка.

В следующем примере возвращается код выхода:

C#

```
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

После запуска программы можно проверить код выхода. В PowerShell используйте `$LastExitCode`. В пакетном файле или скрипте оболочки используйте `%ERRORLEVEL%`.

Если ваш метод `Main` использует `await`, объявите его как `async` с возвращаемым типом `Task` или `Task<int>`. Среда выполнения вызывает `Main` и ожидает завершения `Task`, прежде чем процесс завершится. Тип возвращаемого значения не может быть `void` или `int`, потому что `async` модификатор требует возвращаемого типа, который среда

выполнения может ожидать — `void` и `int` не представляют текущую работу, из-за чего процесс может завершиться раньше, чем асинхронные операции. Используйте `Task` , если вам не нужен код выхода или `Task<int>` когда вы делаете следующее:

C#

```
class Program
{
    static async Task<int> Main(string[] args)
    {
        return await AsyncConsoleWork();
    }

    private static async Task<int> AsyncConsoleWork()
    {
        return 0;
    }
}
```

Аргументы командной строки

Включите параметр `string[] args` в объявление `Main`, чтобы принимать аргументы командной строки. Если их не требуется, опустите параметр. Параметр `args` — это String массив, который никогда не имеет значения NULL, если аргументы не указаны, его `Length` значение равно нулю.

Строковые аргументы можно преобразовать в другие типы с помощью `Parse` или Convert:

C#

```
long num = long.Parse(args[0]);
```

> **Совет**
>
> Анализ аргументов командной строки может быть сложным. Рекомендуется использовать библиотеку **System.CommandLine** для упрощения процесса.

Рабочий пример см. в разделе "Отображение аргументов командной строки".

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

Связанный контент

- System.Environment
- Отображение аргументов командной строки

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.03.2026

Система типов C#

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Они ведут вас шаг за шагом по процессу написания программ и по ходу дела объясняют типы.
>
> **Есть опыт на другом языке?** Если вы уже понимаете системы типов, пропустите различия между значением и ссылкой и выберите руководство по типам, а затем перейдите к статьям по определенным типам.

C# — это строго типизированный язык. Каждая переменная, константа и выражение имеют тип. Компилятор применяет *безопасность типов* , проверяя, является ли каждая операция в коде допустимой для используемых типов. Например, можно добавить два `int` значения, но нельзя добавить и `int` :`bool`

C#

```
int a = 5;
int b = a + 2; // OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
// int c = a + test;
```

> **Примечание.**
>
> В отличие от C и C++, в C#, `bool` не преобразуется в `int`.

Безопасность типов обнаруживает ошибки на этапе компиляции до выполнения вашего кода. Компилятор также внедряет сведения о типе в исполняемый файл в виде метаданных, которые среда CLR использует для дополнительных проверок безопасности во время выполнения.

Объявление переменных с типами

При объявлении переменной вы указываете его тип явным образом или используете `var` , чтобы компилятор выводил тип из назначенного значения:

C#

```
// Explicit type:
int count = 10;
double temperature = 36.6;

// Compiler-inferred type:
var name = "C#";
var items = new List<string> { "one", "two", "three" };
```

Параметры метода и возвращаемые значения также имеют типы. Следующий метод принимает `string` и `int` возвращает `string`:

C#

```
static string GetGreeting(string name, int visitCount)
{
    return visitCount switch
    {
        1 => $"Welcome, {name}!",
        _ => $"Welcome back, {name}! Visit #{visitCount}."
    };
}
```

После объявления переменной нельзя изменить его тип или назначить значение, несовместимое с объявленным типом. Значения можно преобразовать в другие типы. Компилятор выполняет *неявные преобразования* , которые не теряют данные автоматически. *Явные преобразования* (приведения) требуют указания преобразования в коде. Для получения дополнительной информации см. Приведение и преобразование типов.

Встроенные типы и настраиваемые типы

C# предоставляет встроенные типы для общих данных: целых чисел, чисел с плавающей запятой, `bool` `char` и `string`. Каждая программа C# может использовать эти встроенные типы без дополнительных ссылок.

Помимо встроенных типов, можно создать собственные типы с помощью нескольких конструкций:

- **Классы** — ссылочные типы для поведения моделирования и сложных объектов. Поддержка наследования и полиморфизма.
- Структуры — типы значений для **небольших, упрощенных** данных. Каждая переменная содержит собственную копию.

- **Записи** — классы или структуры с равенством, созданным компилятором, `ToString` и недеструктивная мутация с помощью `with` выражений.
- **Интерфейсы — контракты**, которые определяют элементы, которые могут быть реализованы любым классом или структурой.
- **Перечисления** — именованные наборы целочисленных констант, такие как дни недели или режимы доступа к файлам.
- **Кортежи** — упрощенные структурные типы, которые группирует связанные значения без определения именованного типа.
- **Универсальные шаблоны** — типо-параметризованные конструкции, такие как `List<T>` и `Dictionary<TKey, TValue>` обеспечивающие безопасность типов при повторном использовании одной логики для разных типов.

Типы значений и ссылочные типы

Каждый тип в C# — это *тип значения* или *ссылочный тип*. Это различие определяет, как переменные хранят данные и как работает назначение.

Типы значений хранят данные напрямую. При назначении типа значения новой переменной среда выполнения копирует данные. Изменения в одной переменной не влияют на другую. Структуры, перечисления и встроенные числовые типы являются всеми типами значений.

Ссылочные типы хранят ссылку на объект, находящийся в управляемой памяти. При назначении ссылочного типа новой переменной оба переменных указывают на один и тот же объект. Изменения через одну переменную отображаются через другую. Классы, массивы, делегаты и строки являются ссылочными типами.

В следующем примере показано различие. Первый блок показывает определение структуры `Coords` записи, которая является типом значения. Второй блок демонстрирует различное поведение для типов значений и ссылочных типов.

C#

```
public readonly record struct Coords(int X, int Y);
```

C#

```
// Value type: each variable holds its own copy
var point1 = new Coords(3, 4);
var point2 = point1;
Console.WriteLine($"point1: ({point1.X}, {point1.Y})");
Console.WriteLine($"point2: ({point2.X}, {point2.Y})");
// point1 and point2 are independent copies
```

```
// Reference type: both variables refer to the same object
var list1 = new List<int> { 1, 2, 3 };
var list2 = list1;
list2.Add(4);
Console.WriteLine($"list1 count: {list1.Count}"); // 4 — same object
```

Все типы в конечном счете являются производными от System.Object. Типы значений происходят от System.ValueType, который, в свою очередь, происходит от `object`. Эта унифицированная иерархия называется common Type System (CTS). Дополнительные сведения о наследовании см. в разделе "Наследование".

Выберите тип типа

При определении нового типа, тип, который вы выбираете, влияет на поведение вашего кода. Используйте следующие рекомендации, чтобы принять первоначальное решение:

- **Кортеж** — временное группирование значений, которое не требует именованного типа или поведения.
- `struct` или `record struct` — небольшие данные (примерно 64 байта или меньше), семантика значений или неизменяемость. Структуры записи добавляют поддержку равенства на основе значений и `with` выражений.
- `record class` — в первую очередь данные с равенством на основе значений, `ToString` а также недеструктивной мутацией. Поддерживает наследование.
- `class` — сложное поведение, полиморфизм или изменяемое состояние. Большинство пользовательских типов — это классы.
- `interface` — контракт, который могут реализовать несвязанные типы. Определяет возможности, а не личность.
- `enum` — фиксированный набор именованных констант, например коды состояния или параметры.

Более одного варианта часто является разумным.

Тип времени компиляции и тип времени выполнения

Переменная может иметь различные типы во время компиляции и во время выполнения. *Тип времени компиляции* — это объявленный или выводимый тип в исходном коде. Тип *времени выполнения* — это фактический тип экземпляра, на который ссылается переменная. Тип времени выполнения должен совпадать с типом времени компиляции или типом, производным от него или реализующим его. Назначение допустимо только в

том случае, если неявное преобразование существует из типа времени выполнения в тип времени компиляции, например удостоверение, ссылка, бокс или числовое преобразование.

C#

```
// Compile-time and run-time types match:
string message = "Hello, world!";

// Compile-time type differs from run-time type:
object boxed = "This is a string at run time";
IEnumerable<char> characters = "abcdefghijklmnopqrstuvwxyz";
```

В предыдущем примере `boxed` имеет тип `object` времени компиляции, но тип `string` времени выполнения. Задание работает, так как `string` является производным от `object`. Аналогичным образом `characters` имеет тип времени компиляции `IEnumerable<char>`, и присваивание выполняется, поскольку `string` реализует этот интерфейс. Тип времени компиляции управляет разрешением перегрузки и доступными преобразованиями. Тип времени выполнения управляет отправкой виртуальных методов, `is` выражениями и `switch` выражениями.

См. также

- типы значений
- Ссылочные типы
- Общая система типов

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.03.2026

Встроенные типы и литералы

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Они вводят типы при написании первых программ.
>
> **Есть опыт на другом языке?** Встроенные типы C# тесно соответствуют типам в Java, C++ и других языках. Просмотрите разделы синтаксиса литералов и вывод типов для получения сведений о C#.

C# предоставляет набор встроенных типов, которые можно использовать в любой программе без дополнительных ссылок. Эти типы охватывают наиболее распространенные данные, с которыми вы работаете: числа, значения true/false, отдельные символы и текст. Полная таблица ссылочного сопоставления ключевых слов C# с типами .NET доступна см. в разделе «Встроенные типы C# (справочник)».

Числовые типы

C# имеет встроенные типы для целых чисел, чисел с плавающей запятой и десятичных чисел. Наиболее часто используемые типы: `int` и `double` `decimal`:

C#

```
int population = 67_000_000;
long distance = 384_400_000L;
short temperature = -40;
byte red = 255;

double pi = 3.141592653589793;
float gravity = 9.81f;
decimal price = 19.99m;
```

Каждый числовой тип имеет фиксированный размер и диапазон. `int` хранит 32-разрядные целые числа (примерно ±2,1 млрд), `long` хранит 64-разрядные целые числа и `short` `byte` сохраняет меньшие значения. Полный список размеров и диапазонов см. в разделе " Целочисленные числовые типы " и числовые типы с плавающей запятой.

Используется `double` для общей математики с плавающей запятой, `float` если память ограничена, а `decimal` также при необходимости точной десятичной точности (например, финансовых вычислений). Добавьте суффикс `f` для литералов `float` и `m` для литералов `decimal`. Без суффикса компилятор обрабатывает число как десятичную дробь `double`.

Неподписанные типы

Каждый целочисленный тип со знаком имеет аналог без знака, который сохраняет только неотрицательные значения с удвоенным положительным диапазоном.

C#

```
uint fileSize = 4_294_967_295;
ulong totalBytes = 18_446_744_073_709_551_615;
ushort port = 443;
```

Вы можете использовать неподписанные типы, если отрицательные значения недопустимы для данных, например размеры файлов или сетевые порты. На практике многие приложения используют `int` или `long` даже для положительных значений, так как подписанные типы используются по умолчанию в API .NET.

Целые числа собственного размера

`nint` `nuint` Типы представляют целые числа, размер которых соответствует собственному размеру указателя платформы: 32 бита на 32-разрядной платформе, 64 бита на 64-разрядной платформе:

C#

```
nint bufferSize = 1024;
nuint elementCount = 256;
```

Вряд ли вам потребуется `nint` или `nuint` в повседневном коде. Они существуют для сценариев взаимодействия между системами и операций с низкоуровневой памятью, где важен соответствующий размер указателя платформы. `int` или `long` придерживайтесь, если у вас нет определенной причины использовать типы, соответствующие размеру платформы. Дополнительные сведения см. в разделах nint и nuint.

`bool`, `char` и `string`

Помимо чисел, C# предоставляет три других часто используемых встроенных типа:

C#

```
bool isValid = true;
char grade = 'A';
string greeting = "Hello, world!";
```

- `bool` — хранит `true` или `false`. Используйте его в условиях, циклах и логических выражениях.
- `char` — сохраняет один символ Юникода (единица кода UTF-16), заключенную в одинарные кавычки.
- `string` — хранит последовательность символов, заключенную в двойные кавычки. Строки *неизменяемы*. После создания строки невозможно изменить его содержимое. Операции, которые, как представляется, изменяют строку, фактически создают новую.

Строки являются одним из наиболее используемых типов в C#. Подробное покрытие строковых операций, включая интерполяцию, необработанные строковые литералы, поиск, разделение и сравнение, см. в разделе "Строки ".

Синтаксис литерала

Литерал — это значение, которое вы пишете непосредственно в коде. Компилятор назначает каждому литералу тип в зависимости от его формата и любого предоставляемого вами суффикса. C# поддерживает следующие типы литералов:

- **Целые литералы** — десятичные (`42`), шестнадцатеричные (`0x2A`) и двоичные (`0b_0010_1010`).
- **Литералы с плавающей запятой** — `double` по умолчанию (`3.14`), `float` с суффиксом `f` (`3.14f`), и `decimal` с `m` (`3.14m`).
- **Литералы символов** — один символ в одинарных кавычках (`'A'`), включая escape-последовательности (`'\n'`).
- **Строковые литералы** — обычные (`"hello"`), дословные (`@"C:\path"`), необработанные (`""" ... """`) и интерполированные (`$"value: {x}"`).
- **Логические литералы** — `true` и `false`.
- **Литерал `null`** — обозначает отсутствие значения для ссылочных типов и типов значений, допускающих значение NULL.
- **Литерал `default`** — создает значение по умолчанию для любого типа (описывается в выраженияхdefault).

В следующих разделах подробно рассматриваются наиболее распространенные литеральные формы.

Целочисленные литералы

C#

```
int dec = 42;
int hex = 0x2A;
int bin = 0b_0010_1010;
long big = 1_000_000_000L;
```

`0x` Используйте префикс для шестнадцатеричного и `0b` двоичного файла. Добавьте `L` для `long`, `U` для `uint` или `UL` для `ulong`.

`_` Поместите разделитель цифр в любое место в пределах числа, чтобы упростить чтение. Распространенные шаблоны включают разделители тысяч в десятичных литералах (`1_000_000_000`), границы байтов или слов в шестнадцатеричных числах (`0xFF_FF`), а также границы полубайтов в двоичном выражении (`0b_0010_1010`).

Литералы с плавающей запятой

C#

```
double d = 3.14;
float f = 3.14f;
decimal m = 3.14m;
double scientific = 1.5e6; // 1,500,000
```

Числовой литерал без суффикса с десятичной запятой `double`. Добавьте `f` для `float` и `m` для `decimal`. Также поддерживается научная нотация (`1.5e6`).

Символьные и строковые литералы

C#

```
char newline = '\n';
char unicode = '\u0041'; // 'A'

string message = $"Found {dec} items";            // interpolated string
string path = @"C:\Users\docs\file.txt";        // verbatim string
string json = """
    { "name": "Alice", "age": 30 }
    """;                                           // raw string literal
string raw = $"""
    Found {dec} items in "{greeting}"
    """;                                           // raw + interpolated
```

Символьные литералы используют одинарные кавычки и поддерживают escape-последовательности (`\n`, `\t`, `\u`). Строковые литералы используют двойные кавычки.

Префиксируйте строки с `$` для интерполяции. Если строка содержит кавычки, обратные слэши или внедренные JSON или XML, используйте необработанный строковый литерал (ограниченный `"""`) вместо экранирования каждого символа. Необработанные строковые литералы также объединяются с интерполяцией (`$"""`).

Старый код использует `@` (подробные строки), чтобы избежать escape-обработки. Необработанные строковые литералы проще читать и записывать, поэтому предпочитают их для нового кода.

`default` Выражения

Выражение `default` создает значение по умолчанию для типа: `0` для числовых типов, `false` для `bool`, и `null` для ссылочных типов.

C#

```
int defaultInt = default;          // 0
bool defaultBool = default;        // false
string? defaultString = default;   // null

// Use default in a conditional:
var limit = (args.Length > 0) ? int.Parse(args[0]) : default(int);
```

Выражение `default` наиболее полезно в универсальном коде, где вы не знаете конкретный тип и не можете жестко кодить определенное значение, например `0` или `null`. Напишите `default` (без аргумента типа), когда компилятор может выводить тип из контекста, либо `default(T)`, когда тип не является очевидным. Полный список значений по умолчанию по типу см. в разделе "Значения по умолчанию" типов C#.

Неявно типизированные переменные с `var`

Ключевое `var` слово указывает компилятору выводить тип локальной переменной из инициализатора.

C#

```
var count = 10;              // compiler infers int
var name = "C#";             // compiler infers string
var items = new List<int>(); // compiler infers List<int>

// var requires an initializer — the compiler needs a value to infer the type.
// The following line wouldn't compile:
// var unknown;
```

Переменная по-прежнему строго типизирована; `var` не делает его динамическим. Компилятор определяет тип во время компиляции и обеспечивает безопасность типов как обычно. Используйте `var` , когда тип очевиден с правой стороны, чтобы уменьшить визуальный шум. Прописывайте тип, когда это делает код понятнее. Дополнительные сведения см. в разделе "Неявно типизированные локальные переменные".

Типизированные по целевому типу `new` выражения

Если вы уже знаете целевой тип из контекста, например объявление переменной или параметр метода, можно опустить имя типа из `new` выражения:

C#

```
List<string> names = new() { "Alice", "Bob", "Charlie" };
Dictionary<string, int> scores = new()
{
    ["Alice"] = 95,
    ["Bob"] = 87
};
```

Целевой тип `new` уменьшает повторение, если имя типа длинное или отображается в левой части назначения. Он работает в любом месте, где компилятор может определить целевой тип, включая аргументы метода и операторы возврата. Дополнительные сведения см. в операторenew — целевой типnew.

Тип `dynamic`.

Тип `dynamic` обходит проверку типа во время компиляции. Компилятор выполняет операции с переменной `dynamic` во время выполнения.

C#

```
dynamic value = 42;
Console.WriteLine(value.GetType()); // System.Int32

value = "Now I'm a string";
Console.WriteLine(value.GetType()); // System.String

// The compiler doesn't check operations on dynamic at compile time.
// Errors surface at run time instead.
```

Используйте `dynamic` при взаимодействии с COM-API, динамическими языками или сценариями с большим количеством отражения, в которых типы не известны во время компиляции. Избегайте использования `dynamic` в большинстве кода приложения, поскольку при использовании вы теряете безопасность на этапе компиляции. Ошибки, которые компилятор обычно перехватывает, становятся исключениями во время выполнения. Дополнительные сведения см. в разделе "Динамический тип".

См. также

- Общие сведения о системе типов
- Встроенные типы (справочник по C#)
- целочисленные числовые типы
- Числовые типы с плавающей запятой
- типы значений

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.03.2026

Классы C#

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы столкнетесь с классами после того, как вам нужно моделировать объекты с поведением и состоянием.
>
> **Есть опыт на другом языке?** Классы C# похожи на классы в Java или C++. Просмотрите раздел инициализаторов объектов для шаблонов, характерных для C#, и ознакомьтесь с записями как с альтернативой, ориентированной на данные.

Класс — это ссылочный тип, определяющий схему для объектов. При создании переменной типа класса переменная содержит *ссылку на* объект в управляемой куче. Переменная не хранит сами данные объекта. Назначение переменной класса другой переменной копирует ссылку, поэтому обе переменные указывают на один и тот же объект. Классы являются наиболее распространенным способом определения пользовательских типов в C#. Используйте их, если требуется сложное поведение, наследование или общая идентичность между ссылками.

Когда следует использовать классы

Используйте класс, когда:

- Тип имеет сложное поведение или управляет изменяемым состоянием.
- Необходимо наследование для создания базового класса с производными специализациями или создания производного типа, расширяющего существующий класс.
- Экземпляры представляют общую идентичность, а не объекты, имеющие одинаковые значения. Две ссылки на один и тот же объект должны оставаться в синхронизации.
- Тип является крупным или имеет длительный срок жизни и получает преимущества от выделения в куче и ссылочной семантики.

Объявление класса

Определите класс с ключевым словом, `class` за которым следует имя типа. Необязательный модификатор доступа управляет видимостью. Значение по умолчанию — `internal`. Укажите `public`, чтобы разрешить вызывающим элементам из других сборок использовать ваши типы.

C#

```
public class Customer
{
    public string Name { get; set; }

    public Customer(string name) => Name = name;
}
```

Текст класса содержит поля, свойства, методы и события, коллективно называемые *элементами класса*. Имя должно быть допустимым именем идентификатора C#.

Создание объектов

Класс определяет тип, но он не является объектом. Вы создаете объект (*экземпляр* класса) с ключевым словом `new` :

C#

```
var customer = new Customer("Allison");
Console.WriteLine(customer.Name); // Allison
```

Переменная `customer` содержит ссылку на объект, а не сам объект. Можно назначить несколько переменных одному объекту. Изменения по одной ссылке отображаются через другую:

C#

```
var c1 = new Customer("Grace");
var c2 = c1; // both variables reference the same object

c2.Name = "Hopper";
Console.WriteLine(c1.Name); // Hopper — c1 sees the change made through c2
```

Это поведение совместного использования ссылок является одним из различий между классами и структурами. В структурах присваивание копирует данные. Более важно, классы поддерживают наследование. Вы можете создавать иерархии, в которых производные типы повторно используют и специализируются на поведении из базового

класса. Структуры не могут участвовать в иерархиях наследования. Дополнительные сведения о различиях см. в разделе "Типы значений" и ссылочные типы.

Конструкторы и инициализация

При создании экземпляра вы хотите инициализировать его поля и свойства на полезные значения. C# предлагает несколько подходов: инициализаторы полей, параметры конструктора, первичные конструкторы и обязательные свойства.

Инициализаторы полей задают значение по умолчанию непосредственно в объявлении поля:

C#

```
public class Container
{
    private int _capacity = 10;
}
```

Инициализаторы полей назначают разумное значение по умолчанию для поля или свойства. Это отличает его от следующих подходов, где вызывающие могут предоставлять начальное значение.

Параметры конструктора требуют, чтобы вызывающие пользователи предоставляли значения:

C#

```
public class Container
{
    private int _capacity;

    public Container(int capacity) => _capacity = capacity;
}
```

Основные конструкторы (C# 12+) добавляют параметры непосредственно в объявление класса. Эти параметры доступны во всем тексте класса:

C#

```
public class Container(int capacity)
{
    private int _capacity = capacity;
}
```

Основные конструкторы и инициализаторы полей могут работать вместе: инициализатор `_capacity = capacity` полей использует в качестве значения параметр первичного конструктора. Этот шаблон позволяет зафиксировать аргументы конструктора в полях с помощью одного краткого определения.

Обязательные свойства применяются для задания определенных свойств вызывающими объектами с помощью инициализатора объектов:

C#

```
public class Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}
```

C#

```
// var missing = new Person(); // Error: required properties not set
var person = new Person { FirstName = "Grace", LastName = "Hopper" };
Console.WriteLine($"{person.FirstName} {person.LastName}"); // Grace Hopper
```

Дополнительные сведения о шаблонах конструкторов, включая проверку параметров и цепочку конструкторов, см. в разделе "Конструкторы".

Статические классы

`static` класс не может быть создан и содержит только статические члены. Используйте статические классы для организации служебных методов, которые не работают с данными экземпляра:

C#

```
static class MathHelpers
{
    public static double CircleCircumference(double radius) =>
        2 * Math.PI * radius;
}
```

C#

```
double circumference = MathHelpers.CircleCircumference(5.0);
Console.WriteLine($"Circumference: {circumference:F2}"); // Circumference: 31.42
```

Библиотека классов .NET включает множество статических классов, таких как Math и Console. Статический класс неявно запечатан. Вы не можете унаследовать от него или создать экземпляр.

Инициализаторы объектов

Инициализаторы объектов позволяют задавать свойства при создании объекта без написания конструктора для каждого сочетания значений:

C#

```
class ConnectionOptions
{
    public string Host { get; init; } = "localhost";
    public int Port { get; init; } = 80;
    public bool UseSsl { get; init; }
}
```

C#

```
var options = new ConnectionOptions
{
    Host = "db.example.com",
    Port = 5432,
    UseSsl = true
};
Console.WriteLine($"{options.Host}:{options.Port} (SSL: {options.UseSsl})");
// db.example.com:5432 (SSL: True)
```

Инициализаторы объектов работают с любым доступным свойством, которое имеет `set` или init аксессор. Они объединяются естественным образом со `required` свойствами и конструкторами, которые принимают некоторые параметры, позволяя вызывающему объекту задавать другие.

Если свойство является *коллекцией*, для инициализации этого объекта можно использовать выражения коллекции (ссылка на C#).

Наследство

Классы поддерживают *наследование*. Можно определить новый класс, который повторно использует, расширяет или изменяет поведение существующего класса. Класс, от которого вы наследуете, называется *базовым классом*, а новый класс — *производным классом*.

C#

```
var manager = new Manager("Satya", "Engineering");
Console.WriteLine($"{manager.Name} manages {manager.Department}");
// Satya manages Engineering
```

Класс может наследовать от одного базового класса и реализовать несколько интерфейсов. Производные классы наследуют все члены базового класса, за исключением конструкторов. Дополнительные сведения см. в разделе "Наследование и интерфейсы".

См. также

- Общие сведения о системе типов
- Структуры
- Записи
- Интерфейсы
- Наследование

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 08.04.2026

Структуры C#

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Когда вам понадобятся легковесные типы значений в коде, вы столкнетесь с такими структурами.
>
> **Есть опыт на другом языке?** Структуры C# — это типы значений, аналогичные структурам в C++ или Swift, но они размещаются в управляемой куче при упаковке и поддерживают интерфейсы, конструкторы и методы. Просмотрите раздел только для чтения структур для C#-специфических паттернов. Инструкции записей см. в разделе "Записи".

Структура — это тип значения, который содержит свои данные непосредственно в экземпляре, а не через ссылку на объект в heap. При назначении структуры новой переменной среда выполнения копирует весь экземпляр. Изменения в одной переменной не влияют на другую, так как каждая переменная представляет другой экземпляр. Используйте структуры для небольших упрощенных типов, основная роль которых — хранение данных, а не поведение моделирования. Примеры включают координаты, цвета, измерения или параметры конфигурации.

Когда следует использовать структуры

Используйте структуру, когда ваш тип:

- Представляет одно значение или небольшую группу связанных значений (примерно 16 байтов или меньше).
- Имеет семантику значений— два экземпляра с одинаковыми данными должны быть равными.
- Это в первую очередь контейнер данных, а не модель поведения.
- Не требуется наследование от базового типа (структуры не могут наследовать от других структур или классов, но они могут реализовывать интерфейсы).

Более широкое сравнение, включающее классы, записи, кортежи и интерфейсы, см. в разделе "Выбор типа".

Объявление структуры

Определите структуру с ключевым словом `struct` . Структуру можно содержать поля, свойства, методы и конструкторы, как класс:

C#

```
struct Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public readonly double DistanceTo(Point other)
    {
        var dx = X - other.X;
        var dy = Y - other.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }

    public override string ToString() => $"({X}, {Y})";
}
```

В `Point` структуре хранятся два `double` значения и предоставляется метод для вычисления расстояния между двумя точками. Метод `DistanceTo` помечен `readonly` , так как он не изменяет состояние структуры. Этот шаблон рассматривается в элементах чтения.

Семантика значений

Структуры — это *типы значений*. Назначение копирует данные, поэтому каждая переменная содержит собственную независимую копию:

C#

```
var p1 = new Point { X = 3, Y = 4 };
var p2 = p1; // copies the data
p2.X = 10;

Console.WriteLine(p1); // (3, 4)  — p1 is unchanged
Console.WriteLine(p2); // (10, 4) — only p2 was modified
```

Поскольку структуры являются контейнерами данных, назначение копирует каждый член данных в новый независимый экземпляр. Каждая копия отличается. Изменение одного не влияет на другое. Это поведение отличается от классов, где назначение копирует только ссылку, и обе переменные используют один и тот же объект. Дополнительные сведения о различиях см. в разделе "Типы значений" и ссылочные типы.

Конструкторы структуры

Конструкторы в структурах можно определить так же, как и в классах. Структуры могут иметь *конструкторы без параметров* , которые задают пользовательские значения по умолчанию. Термин "конструктор без параметров" отличает экземпляр, созданный с `new` вызовом (который выполняет логику вашего конструктора) от экземпляра *по умолчанию*, созданного с `default` помощью выражения (который инициализирует все поля нулями):

C#

```
struct ConnectionSettings
{
    public string Host { get; set; }
    public int Port { get; set; }
    public int MaxRetries { get; set; }

    public ConnectionSettings()
    {
        Host = "localhost";
        Port = 8080;
        MaxRetries = 3;
    }
}
```

Конструктор без параметров выполняется при использовании `new` без аргументов. Выражение `default` обходит конструктор и задает всем полям значения по умолчанию (`0`, `null`, `false`). Имейте в виду разницу:

C#

```
var custom = new ConnectionSettings();
Console.WriteLine($"{custom.Host}:{custom.Port} (retries: {custom.MaxRetries})");
// localhost:8080 (retries: 3)

var defaults = default(ConnectionSettings);
Console.WriteLine($"{defaults.Host ?? "(null)"}:{defaults.Port} (retries:
{defaults.MaxRetries})");
// (null):0 (retries: 0)
```

Компилятор автоматически инициализирует все поля, которые не заданы явным образом в конструкторе. Вы можете инициализировать только поля, которые нуждаются в значениях, отличных от значений по умолчанию:

C#

```
struct GameTile
{
    public int Row { get; set; }
    public int Column { get; set; }
    public bool IsBlocked { get; set; }

    public GameTile(int row, int column)
    {
        Row = row;
        Column = column;
        // IsBlocked is automatically initialized to false
    }
}
```

В следующем примере отображается значение по умолчанию для `IsBlocked`:

C#

```
var tile = new GameTile(2, 5);
Console.WriteLine($"Tile ({tile.Row}, {tile.Column}), blocked: {tile.IsBlocked}");
// Tile (2, 5), blocked: False
```

Свойство `IsBlocked` не назначается в конструкторе, поэтому компилятор задает для него значение `false` (значение по умолчанию для `bool`). Эта функция уменьшает количество шаблонного кода в конструкторах, которые должны инициализировать только несколько полей.

Структуры только для чтения и члены только для чтения

`readonly struct` гарантирует, что ни один член экземпляра не изменяет состояние структуры. Компилятор применяет эту гарантию, требуя, чтобы все поля и автоматически реализованные свойства были доступны только для чтения:

C#

```
readonly struct Temperature
{
    public double Celsius { get; }

    public Temperature(double celsius) => Celsius = celsius;

    public double Fahrenheit => Celsius * 9.0 / 5.0 + 32.0;
```

```
    public override string ToString() => $"{Celsius:F1}°C ({Fahrenheit:F1}°F)";
}
```

В следующем примере создается экземпляр `Temperature`, и считываются его свойства.

C#

```
var temp = new Temperature(100);
Console.WriteLine(temp); // 100.0°C (212.0°F)
// temp.Celsius = 50; // Error: property is read-only
```

Если вам не нужна вся структуру, чтобы быть неизменяемой, пометьте отдельные члены как `readonly` вместо этого. Член `readonly` не может изменить состояние структуры, и компилятор проверяет это гарантии.

C#

```
struct Velocity
{
    public double X
    {
        readonly get;
        set;
    }

    public double Y
    {
        readonly get;
        set;
    }

    public readonly double Speed => Math.Sqrt(X * X + Y * Y);

    public readonly override string ToString() => $"({X}, {Y}) speed={Speed:F2}";
}
```

В следующем примере показано, что `readonly` элементы возвращают обновленные значения при изменении изменяемых свойств:

C#

```
var v = new Velocity { X = 3, Y = 4 };
Console.WriteLine(v.Speed); // 5
Console.WriteLine(v);       // (3, 4) speed=5.00
v.X = 6;
Console.WriteLine(v.Speed); // 7.211...
```

Маркировка элементов `readonly` помогает компилятору оптимизировать оборонительные копии. При передаче `readonly` структуры в метод, принимающий `in` параметр, компилятор знает, что копия не требуется.

См. также

- Общие сведения о системе типов
- Классы
- Записи
- Типы структур (справочник по C#)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 08.04.2026

Типы записей C#

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы встретите записи, когда вам нужны сжатые типы данных со встроенным равенством.
>
> **Есть опыт на другом языке?** Записи C# похожи на классы данных в Kotlin или классах регистра в Scala. Они оптимизированы для хранения данных, с равенством, генерируемым компилятором, и семантикой копирования. Просмотрите разделы record class vs record struct и выражений with для шаблонов, специфичных для C#, чтобы найти характерные шаблоны.

Ключевое слово record является модификатором, применяемым к `class` или `struct`. Он сообщает компилятору создать равенство значений, отформатированную `ToString` и недеструктивную мутацию через with выражения. Базовый тип, класс или структуру, по-прежнему определяет, используют ли экземпляры семантику ссылок или значений. Модификатор `record` добавляет понятное для данных поведение поверх этих семантик. Используйте записи, когда основная роль типа хранит данные и два экземпляра с одинаковыми значениями следует считать равными.

Когда следует использовать записи

Используйте запись, когда выполняются все следующие условия:

- Основная роль типа — хранение данных.
- Два экземпляра с одинаковыми значениями должны быть равными.
- Требуется неизменяемость (особенно для `record class` типов).
- Вы хотите удобочитаемую `ToString` , не записывая ее вручную.

При выборе между `record class` и `record struct`:

- Используйте `record class` когда требуется наследование или когда тип достаточно велик, чтобы копирование при каждом назначении было дорогостоящим.
- Используйте `record struct` для небольших автономных значений, когда полезны семантика копирования и выделение на стеке.

Избегайте записей типов сущностей в Entity Framework Core, которые зависят от ссылочного равенства для отслеживания сущностей. Более широкое сравнение параметров типа см. в разделе "Выбор типа".

Объявить запись

Можно применить `record` к классу или структуре. Простейшая форма использует *позиционные параметры* , определяющие конструктор и свойства в одной строке:

C#

```
public record Person(string FirstName, string LastName);
```

Тот же позиционный синтаксис работает для `record struct` типов:

C#

```
public record struct Coordinate(double Latitude, double Longitude);

public readonly record struct Temperature(double Celsius)
{
    public double Fahrenheit => Celsius * 9.0 / 5.0 + 32.0;
}
```

Написание `record` является сокращением для record class, который является ссылочным типом. Запись record struct создает тип значения. Компилятор создает свойства из позиционных параметров в обоих случаях, но значения по умолчанию отличаются:

- **`record class`**: свойства имеют значение `init`-only (неизменяемые после построения).
- **`record struct`**: свойства по умолчанию доступны для записи и чтения. Добавьте `readonly` (`readonly record struct`), чтобы сделать их только `init`.

Записи также можно записывать с помощью стандартного синтаксиса свойств, если вам нужен дополнительный элемент управления. Например, чтобы сделать свойство чтение и запись вместо `init`-only:

C#

```
public record Product
{
    public required string Name { get; init; }
```

```
    public decimal Price { get; set; }
}
```

`record class` и `record struct`

Поскольку модификатор сохраняет семантику базового типа, `record` и `record class` ведут себя по-другому при назначении или сравнении ссылок. Назначение `record class` копирует ссылку. Обе переменные указывают на один и тот же объект. Назначение `record struct` копирует данные, поэтому изменения одной переменной не влияют на другую.

C#

```
// Record class — assignment copies the reference
var p1 = new Person("Grace", "Hopper");
var p2 = p1; // p1 and p2 point to the same object:
Console.WriteLine(ReferenceEquals(p1, p2)); // True

// Record struct — assignment copies the data
var c1 = new Coordinate(47.6062, -122.3321);
var c2 = c1;
c2.Longitude = 0.0; // mutating c2 doesn't affect c1
Console.WriteLine(c1.Longitude); // -122.3321
Console.WriteLine(c2.Longitude); // 0
```

Структуры записей также обеспечивают равенство значений, созданных компилятором:

C#

```
var home = new Coordinate(47.6062, -122.3321);
var copy = home;

Console.WriteLine(home);           // Coordinate { Latitude = 47.6062, Longitude =
-122.3321 }
Console.WriteLine(home == copy);   // True — value equality
```

Выберите `record class` , когда требуется наследование или когда экземпляры достаточно большие, чтобы копирование было дорогостоящим. Выберите `record struct` для небольших автономных данных, где семантика копирования, характерная для типов-значений, подходит. Дополнительные сведения о семантике типа значений см. в разделе "Структуры".

Равенство значений

Модификатор `record` предоставляет как классы, так и структуры, созданные компилятором, равенство свойств по свойству. Вот как равенство работает во всех четырех типах:

- **Обычный класс**: использует *равенство ссылок* по умолчанию. Оператор `==` проверяет, указывают ли две переменные на один и тот же объект, а не совпадают ли данные.
- **Простая структура**: поддерживает равенство ValueType.Equalsзначений, но реализация по умолчанию использует отражение, которое медленнее и не создает `==`/`!=` операторы.
- **`record class`**: компилятор создает методы `Equals` и `GetHashCode`, а также операторы `==`/`!=`, которые сравнивают каждое значение свойства. Два разных объекта с одинаковыми данными равны.
- **`record struct`**: То же равенство, сгенерированное компилятором для `record class`, но без использования отражения, что делает его быстрее, чем обычное равенство структур.

В следующем примере показано равенство классов записей:

C#

```
// Person is a record type with three properties: FirstName, LastName, and
PhoneNumbers.
var phones = new string[] { "555-1234" };
var person1 = new Person("Grace", "Hopper", phones);
var person2 = new Person("Grace", "Hopper", phones);

Console.WriteLine(person1 == person2);              // True
Console.WriteLine(ReferenceEquals(person1, person2)); // False

person1.PhoneNumbers[0] = "555-9999";
Console.WriteLine(person2.PhoneNumbers[0]); // 555-9999 — same array
```

Два `Person` экземпляра являются разными объектами, но они равны, так как все их значения свойств соответствуют. Свойства массива сравниваются по ссылке, а не по содержимому. Изменение общего массива отображается с помощью обеих записей, так как сам массив не копируется.

Недеструктивная мутация с `with` выражениями

Записи часто неизменяемы, поэтому вы не можете изменить свойство после создания. `with` Выражение создает копию с одним или несколькими свойствами, оставляя

исходную запись без изменений. Этот подход работает для обоих типов: `record class` и `record struct`

C#

```
var original = new Person("Grace", "Hopper");
var modified = original with { FirstName = "Margaret" };

Console.WriteLine(original); // Person { FirstName = Grace, LastName = Hopper }
Console.WriteLine(modified); // Person { FirstName = Margaret, LastName = Hopper }
Console.WriteLine(original == modified); // False

var copy = original with { };
Console.WriteLine(original == copy); // True
```

Тот же синтаксис работает для `record struct` типов:

C#

```
var shifted = home with { Longitude = -122.0 };
Console.WriteLine(shifted);        // Coordinate { Latitude = 47.6062, Longitude =
-122 }
Console.WriteLine(home == shifted); // False
```

Выражение `with` копирует существующий экземпляр, а затем применяет указанные изменения свойств.

Позиционные записи и деконструкция

Позиционные записи генерируют метод `Deconstruct`, который вы используете для извлечения значений свойств в отдельные переменные:

C#

```
var (first, last) = person;
Console.WriteLine($"{first} {last}");
// Grace Hopper
```

Деконструкция работает с обоими `record class` и `record struct` типами. Его можно использовать в сопоставлениях назначений, `foreach` циклов и шаблонов.

Наследование записей

Один `record class` может наследоваться от другого `record class`. Запись не может наследоваться от обычного класса, и класс не может наследовать от записи:

C#

```
public record Student(string FirstName, string LastName, int GradeLevel)
    : Person(FirstName, LastName);
```

Проверки равенства значений включают тип времени выполнения, поэтому `Person` и `Student` с одинаковыми `FirstName` и `LastName` не считаются равными. Структуры записей не поддерживают наследование, так как структуры не могут наследоваться от других типов.

См. также

- Общие сведения о системе типов
- Классы
- Структуры
- Записи (справочник по C#)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 06.04.2026

Интерфейсы — определение поведения для нескольких типов

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы столкнетесь с интерфейсами, как только вам понадобится определить общее поведение для несвязанных типов.
>
> **Есть опыт на другом языке?** Интерфейсы C# похожи на интерфейсы в Java или протоколах Swift. Просмотрите явную реализацию в поисках специфических паттернов C#.

Интерфейс определяет контракт: группу связанных методов, свойств, событий и индексаторов, которые classstruct необходимо реализовать. Интерфейсы позволяют одному типу реализовать несколько контрактов, что важно, так как C# не поддерживает множественное наследование классов. Структуры не могут наследовать от других структур или классов, поэтому интерфейсы являются единственным способом добавления общего поведения между типами структур.

В следующем примере объявляется интерфейс и класс, реализующий его:

C#

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

C#

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    public bool Equals(Car? car) =>
        car is not null &&
```

```
        (Make, Model, Year) == (car.Make, car.Model, car.Year);
}
```

Любой класс или структура, реализующая IEquatable<T> , должна предоставлять метод, соответствующий `Equals` сигнатуре интерфейса. Вы можете рассчитывать на то, что любая реализация `IEquatable<T>` будет поддерживать сравнение на равенство, независимо от ее конкретного типа. Это прогнозируемость является основным значением интерфейсов.

Объявление интерфейса

Определите интерфейс с ключевым словом interface . По соглашению имена интерфейсов начинаются с буквы `I`:

C#

```
interface ILogger
{
    void Log(string message);
    string Name { get; }
}
```

Интерфейсы могут содержать методы, свойства, события и индексаторы. Интерфейс не может содержать поля экземпляра, конструкторы экземпляра или финализаторы. Члены по умолчанию `public`. При необходимости можно указать другие модификаторы специальных возможностей. Например, используйте `internal` для элементов, которые не должны быть видимы за пределами сборки.

Реализация интерфейса

Класс или структура перечисляет интерфейсы, которые он реализует после двоеточия в объявлении. Класс должен предоставить реализацию для каждого члена, объявленного в интерфейсе:

C#

```
public class ConsoleLogger : ILogger
{
    public string Name => "Console";

    public void Log(string message) =>
        Console.WriteLine($"[{Name}] {message}");
}

public class FileLogger : ILogger
```

```
{
    public string Name => "File";

    public void Log(string message)
    {
        // In a real app, write to a file
        Console.WriteLine($"[{Name}] Writing to file: {message}");
    }
}
```

Класс может реализовать несколько интерфейсов, разделенных запятыми. Он должен предоставлять реализации для всех членов из каждого интерфейса, который он перечисляет.

Явная реализация

Иногда необходимо реализовать член интерфейса, не делая его частью общедоступного API класса. *Явная реализация* квалифицирует член с именем интерфейса. Член доступен только с помощью переменной типа интерфейса:

C#

```
interface IMetric
{
    double GetDistance(); // Returns meters
}

interface IImperial
{
    double GetDistance(); // Returns feet
}

public class Runway(double meters) : IMetric, IImperial
{
    // Explicit implementation for IMetric
    double IMetric.GetDistance() => meters;

    // Explicit implementation for IImperial
    double IImperial.GetDistance() => meters * 3.28084;
}
```

Явная реализация полезна, если два интерфейса объявляют члены с одинаковым именем, или если вы хотите сохранить публичный интерфейс класса чистым. Дополнительные сведения см. в разделе "Явная реализация интерфейса".

Наследование интерфейса

Интерфейсы могут наследоваться от одного или нескольких других интерфейсов. Класс, реализующий производный интерфейс, должен реализовать все члены из производного интерфейса и всех его базовых интерфейсов:

C#

```
interface IDrawable
{
    void Draw();
}

interface IShape : IDrawable
{
    double Area { get; }
}

public class Circle(double radius) : IShape
{
    public double Area => Math.PI * radius * radius;

    public void Draw() =>
        Console.WriteLine($"Drawing circle with area {Area:F2}");
}
```

Класс, реализующий `IShape`, может быть неявно преобразован в `IDrawable`, так как `IShape` наследует от него.

Интерфейсы и абстрактные классы

Интерфейсы и абстрактные классы определяют контракты, которые должны выполнять производные типы.

- **Используйте абстрактный класс** , если связанные типы совместно используют состояние (поля), конструкторы или не открытые члены. Абстрактные классы позволяют развивать иерархию, добавляя новые члены с поведением по умолчанию, не нарушая существующие производные типы.
- **Используйте интерфейс**, когда типу требуется выполнить контракт, который пересекает несвязанные иерархии, или когда необходимо реализовать несколько контрактов. Интерфейсы не могут объявлять поля экземпляров или конструкторы, поэтому они лучше всего подходят для добавления функциональности в типы, которые уже имеют базовый класс. В расширенных сценариях интерфейсы также поддерживают реализации элементов по умолчанию.

Класс может наследоваться только от одного базового класса, но может реализовать несколько интерфейсов. Это различие часто делает интерфейсы лучшим выбором для определения возможностей, которые охватывают иерархии типов.

Работа с внутренними интерфейсами

Как правило, можно реализовать внутренний интерфейс с общедоступными элементами, если все типы в сигнатуре интерфейса общедоступны. Если интерфейс использует внутренние типы в подписях членов, необходимо использовать явную реализацию, так как реализующий член не может быть общедоступным во время предоставления внутренних типов:

C#

```
internal class InternalConfiguration
{
    public string Setting { get; set; } = "";
}

internal interface ILoggable
{
    void Log(string message);
}

internal interface IConfigurable
{
    void Configure(InternalConfiguration config);
}

public class ServiceImplementation : ILoggable, IConfigurable
{
    // Implicit implementation: ILoggable uses only public types in its signature
    public void Log(string message) =>
        Console.WriteLine($"Log: {message}");

    // Explicit implementation: IConfigurable uses internal types
    void IConfigurable.Configure(InternalConfiguration config) =>
        Console.WriteLine($"Configured with: {config.Setting}");
}
```

В предыдущем примере `IConfigurable` используется внутренний тип `InternalConfiguration` в сигнатуре метода. `ServiceImplementation` использует явную реализацию для этого элемента. В отличие от этого, `ILoggable` использует только открытые типы (`string`) в его сигнатуре и может быть реализована неявно.

Элементы интерфейса по умолчанию и статические абстрактные элементы

Интерфейсы поддерживают две расширенные функции, которые выходят за рамки основных контрактов:

- **Элементы интерфейса по умолчанию позволяют интерфейсу** предоставлять текст метода. Типы наследуют реализацию по умолчанию и могут при необходимости ее переопределить. Дополнительные сведения см. в статье о методах интерфейса по умолчанию.
- **Статические абстрактные члены** требуют реализации типов для предоставления статического элемента, который полезен для определения контрактов операторов или шаблонов фабрики. Дополнительные сведения см. в разделе статических абстрактных элементов в интерфейсах.

Оба компонента рассматриваются в статье по интерфейсам в справочнике по языку. Большинство повседневного использования интерфейса включает объявление и реализацию шаблонов, описанных ранее в этой статье.

Сводка по интерфейсам

- Интерфейс определяет контракт методов, свойств, событий и индексаторов.
- Класс или структура, реализующая интерфейс, должна предоставлять реализации для всех объявленных членов (если интерфейс не предоставляет реализацию по умолчанию).
- Невозможно создать экземпляр интерфейса напрямую.
- Класс или структура может реализовывать несколько интерфейсов. Класс может наследовать базовому классу и также реализовывать один или несколько интерфейсов.
- Имена интерфейсов обычно начинаются с `I`.

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 13.04.2026

Перечисления C#

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы увидите перечисления, когда необходимо представить фиксированный набор вариантов в коде.
>
> **Есть опыт на другом языке?** Перечисления C# работают аналогично перечислениям в Java или C++, с дополнительной поддержкой битовых флагов и сопоставления шаблонов. Просмотрите разделы флагов и выражений switch для шаблонов C#.

Тип перечисления (или *перечисление*) определяет набор именованных констант, поддерживаемых целым числом. Используйте перечисления, если значение должно быть одним из фиксированных наборов параметров, таких как дни недели, коды состояния HTTP, уровни журнала или направления. Перечисления делают код более читаемым и менее подверженным ошибкам, чем необработанные целые константы, так как компилятор применяет именованные значения.

Объявить перечисление

Определите перечисление с `enum` ключевым словом, за которым следует имя типа и его члены:

C#

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

По умолчанию базовый тип — это `int`, и значения начинаются с `0` и затем увеличиваются по одному. `Season.Spring` это `0`, `Season.Summer` это `1`, и так далее.

Указание базового типа и явных значений

Вы можете выбрать другой целочисленный тип и назначить явные значения для управления числовым представлением:

C#

```
enum HttpStatus : ushort
{
    OK = 200,
    NotFound = 404,
    InternalServerError = 500
}
```

Используйте явные значения, если числа имеют внешнее значение, например коды состояния HTTP или идентификаторы протокола. Базовый тип может быть любым целым типом , кроме `char`. Используйте `byte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`.

Использование перечислений в выражениях switch

Перечисления интуитивно работают с выражениями `switch` и сопоставлением шаблонов. Компилятор предупреждает вас, если вы не обрабатываете все элементы, что помогает предотвратить ошибки при добавлении нового значения позже:

C#

```
static string DescribeSeason(Season season) => season switch
{
    Season.Spring => "Flowers bloom and temperatures rise.",
    Season.Summer => "Long days and warm weather.",
    Season.Autumn => "Leaves change color and fall.",
    Season.Winter => "Short days and cold temperatures.",
    _ => throw new ArgumentOutOfRangeException(nameof(season))
};
```

C#

```
var today = Season.Autumn;
Console.WriteLine(DescribeSeason(today));
```

Паттерн отбрасывания (`_`) обрабатывает любое значение, которое не указано явно. Поскольку базовый тип перечисления является целым числом, переменная может содержать значение, которое не соответствует ни одному именованному элементу. Например, `(Season)99` допустимо во время выполнения. Шаблон отбрасывания гарантирует безопасную обработку выражением переключателя непредвиденных значений. *Сопоставление шаблонов* — это функция C#, которая проверяет значение для фигуры или

условия. В этом примере каждый `case` проверяет, соответствует ли перечисление конкретному члену. Выражения switch — это одна из нескольких форм сопоставления шаблонов. Дополнительные сведения о сопоставлении шаблонов см. в разделе "Сопоставление шаблонов".

Битовые флаги

Когда перечисление представляет собой комбинацию вариантов, а не единственный выбор, присвойте каждому элементу значение, равное степени двойки, и примените следующий FlagsAttribute:

C#

```
[Flags]
enum FileAccess
{
    None = 0,
    Read = 1,
    Write = 2,
    Execute = 4,
    ReadWrite = Read | Write,
    All = Read | Write | Execute
}
```

Объедините значения, используя оператор `|`, и проверьте отдельные флаги, используя HasFlag:

C#

```
var permissions = FileAccess.Read | FileAccess.Write;

Console.WriteLine(permissions);                          // ReadWrite
Console.WriteLine(permissions.HasFlag(FileAccess.Read)); // True
Console.WriteLine(permissions.HasFlag(FileAccess.Execute)); // False
```

Атрибут `[Flags]` также влияет `ToString()`. Он отображает объединенные значения как разделенные запятыми имена (например `Read, Write`) вместо необработанного числа. Дополнительные сведения см. в разделе System.FlagsAttribute.

Преобразование между перечислениями и целыми числами

Явные выражения приведения преобразуются между перечислением и его базовым целым типом. Явное приведение использует синтаксис `(Type)value`, чтобы указать компилятору на ваше намерение преобразовать.

C#

```
var status = HttpStatus.NotFound;
ushort code = (ushort)status;
Console.WriteLine($"Status: {status} ({code})"); // Status: NotFound (404)

var fromCode = (HttpStatus)200;
Console.WriteLine(fromCode); // OK
```

Помните, что приведение целого числа в перечисление не проверяет, соответствует ли значение определенному члену. Используется Enum.IsDefined для проверки допустимости при принятии числовых входных данных из внешних источников.

Анализ строк и итерации значений

Базовый Enum класс предоставляет методы для синтаксического анализа строк и итерации по всем определенным значениям:

C#

```
// Parse a string to an enum value:
var parsed = Enum.Parse<Season>("Winter");
Console.WriteLine(parsed); // Winter

// Try to parse safely. It returns false only when the input can't be parsed. Call
Enum.IsDefined to validate named members:
if (Enum.TryParse<Season>("Monsoon", out var unknown))
{
    Console.WriteLine(unknown);
}
else
{
    Console.WriteLine("'Monsoon' is not a valid Season"); // 'Monsoon' is not a valid
Season
}

// Iterate over all values in an enum:
foreach (var season in Enum.GetValues<Season>())
{
    Console.WriteLine($"{season} = {(int)season}");
}
// Spring = 0
// Summer = 1
```

```
// Autumn = 2
// Winter = 3
```

Используйте Enum.TryParse<TEnum>(String, Boolean, TEnum) вместо того, Enum.Parse<TEnum>(String) когда входные данные могут быть недопустимыми. Он возвращает `false` вместо того, чтобы вызывать исключение.

См. также

- Общие сведения о системе типов
- Типы перечисления (справочник по C#)
- Сопоставление шаблонов
- Рекомендации по проектированию перечисления

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 10.04.2026

Универсальные типы и методы

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы столкнетесь с дженериками, как только используете такие коллекции, как `List<T>`.
>
> **Есть опыт на другом языке?** Универсальные шаблоны C# похожи на универсальные шаблоны в Java или шаблонах в C++, но с полными сведениями о типе среды выполнения и без удаления типов. Просмотрите разделы выражений коллекции и ковариации и контравариации для характерных шаблонов C#.

Универсальные шаблоны позволяют писать код, который работает с любым типом при сохранении полной безопасности типов. Вместо написания отдельных классов или методов для `int`, `string` а также каждого другого типа, необходимо написать одну версию с одним или несколькими *параметрами типа* (например `T`, или `TKey`) `TValue` и указать фактические типы при его использовании. Компилятор проверяет типы во время компиляции, поэтому не требуется приведение среды выполнения или риск `InvalidCastException`.

Вы сталкиваетесь с дженериками постоянно в C# в вашей повседневной работе. Коллекции, асинхронные возвращаемые типы, делегаты и LINQ используют универсальные типы:

C#

```
List<int> scores = [95, 87, 72, 91];
Dictionary<string, decimal> prices = new()
{
    ["Widget"] = 19.99m,
    ["Gadget"] = 29.99m
};
Task<string> greeting = Task.FromResult("Hello, generics!");
Func<int, bool> isPositive = n => n > 0;

Console.WriteLine($"First score: {scores[0]}");
Console.WriteLine($"Widget price: {prices["Widget"]:C}");
Console.WriteLine($"Greeting: {await greeting}");
Console.WriteLine($"Is 5 positive? {isPositive(5)}");
```

В каждом случае аргумент типа в угловых скобках (`<int>`, `<string>`, `<Product>`) сообщает универсальному типу данных, с которыми он содержит или работает. Компилятор

обеспечивает безопасность типов. Нельзя случайно добавить `string` в `List<int>`.

Использование универсальных типов

Чаще вы используете универсальные типы из библиотеки классов .NET вместо создания собственных. В следующих разделах показаны наиболее распространенные универсальные типы, которые вы будете использовать.

Универсальные коллекции

Пространство System.Collections.Generic имен предоставляет типобезопасные классы коллекций. Всегда используйте эти коллекции вместо негенерических коллекций, таких как ArrayList:

C#

```
// A strongly typed list of strings
List<string> names = ["Alice", "Bob", "Carol"];
names.Add("Dave");
// names.Add(42); // Compile-time error: can't add an int to List<string>

// A dictionary mapping string keys to int values
var inventory = new Dictionary<string, int>
{
    ["Apples"] = 50,
    ["Oranges"] = 30
};
inventory["Bananas"] = 25;

// A set that prevents duplicates
HashSet<int> uniqueIds = [1, 2, 3, 1, 2];
Console.WriteLine($"Unique count: {uniqueIds.Count}"); // 3

// A FIFO queue
Queue<string> tasks = new();
tasks.Enqueue("Build");
tasks.Enqueue("Test");
Console.WriteLine($"Next task: {tasks.Dequeue()}"); // Build
```

Универсальные коллекции предотвращают ошибки типов во время выполнения, так как вместо этого возникают ошибки при компиляции. Эти коллекции также избегают бокса для типов значений, что повышает производительность.

Универсальные методы

Универсальный метод объявляет собственный параметр типа. Компилятор часто *вводит* аргумент типа из передаваемых значений, поэтому не нужно явно указывать его:

C#

```
static void Print<T>(T value) =>
    Console.WriteLine($"Value: {value}");

Print(42);        // Compiler infers T as int
Print("hello");   // Compiler infers T as string
Print(3.14);      // Compiler infers T as double
```

В вызове `Print(42)` компилятор определяет `T` как `int` из аргумента. Вы можете написать `Print<int>(42)` явно, но вывод типов делает код более чистым.

Выражения коллекции

Выражения коллекции (C# 12) предоставляют краткий синтаксис для создания коллекций. Используйте квадратные скобки вместо вызовов конструктора или синтаксиса инициализатора:

C#

```
// Create a list with a collection expression
List<string> fruits = ["Apple", "Banana", "Cherry"];

// Create an array
int[] numbers = [1, 2, 3, 4, 5];

// Works with any supported collection type
IReadOnlyList<double> temperatures = [72.0, 68.5, 75.3];

Console.WriteLine($"Fruits: {string.Join(", ", fruits)}");
Console.WriteLine($"Numbers: {string.Join(", ", numbers)}");
Console.WriteLine($"Temps: {string.Join(", ", temperatures)}");
```

Оператор распространения (`..`) встраивает элементы одной коллекции в другую, что полезно для объединения последовательностей:

C#

```
List<int> first = [1, 2, 3];
List<int> second = [4, 5, 6];

// Spread both lists into a new combined list
List<int> combined = [.. first, .. second];
Console.WriteLine(string.Join(", ", combined));
```

```
// Output: 1, 2, 3, 4, 5, 6

// Add extra elements alongside spreads
List<int> withExtras = [0, .. first, 99, .. second];
Console.WriteLine(string.Join(", ", withExtras));
// Output: 0, 1, 2, 3, 99, 4, 5, 6
```

Выражения коллекции работают с массивами, `List<T>`, `Span<T>`, `ImmutableArray<T>` и любым типом, поддерживающим шаблон построителя коллекций. Для полного справочника по синтаксису см. в Выражения коллекций.

Инициализация словаря

Словари можно кратко инициализировать с использованием индексаторных инициализаторов. Этот синтаксис использует квадратные скобки для задания пар "ключ-значение":

C#

```
Dictionary<string, int> scores = new()
{
    ["Alice"] = 95,
    ["Bob"] = 87,
    ["Carol"] = 92
};

foreach (var (name, score) in scores)
{
    Console.WriteLine($"{name}: {score}");
}
```

Вы можете объединить словари, скопировав один из них и применив переопределения:

C#

```
Dictionary<string, int> defaults = new()
{
    ["Timeout"] = 30,
    ["Retries"] = 3
};
Dictionary<string, int> overrides = new()
{
    ["Timeout"] = 60
};

// Merge defaults and overrides into a new dictionary
Dictionary<string, int> config = new(defaults);
foreach (var (key, value) in overrides)
```

```
{
    config[key] = value;
}

Console.WriteLine($"Timeout: {config["Timeout"]}");  // 60
Console.WriteLine($"Retries: {config["Retries"]}");   // 3
```

Ограничения типов

Ограничения задают пределы для аргументов того, какие типы может принимать универсальный тип или метод. Ограничения позволяют вызывать методы или свойства доступа к параметру типа, который не будет доступен только в `object` :

C#

```
static T Max<T>(T a, T b) where T : IComparable<T> =>
    a.CompareTo(b) >= 0 ? a : b;

Console.WriteLine(Max(3, 7));          // 7
Console.WriteLine(Max("apple", "banana")); // banana

static T CreateDefault<T>() where T : new() => new T();

var list = CreateDefault<List<int>>(); // Creates an empty List<int>
Console.WriteLine($"Empty list count: {list.Count}"); // 0
```

Наиболее распространенными ограничениями являются:

Развернуть таблицу

Ограничение	Значение
`where T : class`	`T` должен быть ссылочным типом
`where T : struct`	`T` должен быть типом значения, не допускающего значение NULL
`where T : new()`	`T` должен иметь открытый конструктор без параметров
`where T : BaseClass`	`T` должен быть производным от `BaseClass`
`where T : IInterface`	`T` должен реализовать `IInterface`

Ограничения можно объединить: `where T : class, IComparable<T>, new()` Менее распространенные ограничения включают `where T : System.Enum` и `where T : System.Delegate` `where T : unmanaged` для специализированных сценариев. Полный список см. в разделе "Ограничения" для параметров типа.

Ковариантность и контрвариантность

Ковариантность и *контравариантность* описывают поведение универсальных типов с наследованием. Они определяют, можно ли использовать более производный или менее производный аргумент типа, чем первоначально указанный:

C#

```
// Covariance: IEnumerable<Dog> can be used as IEnumerable<Animal>
// because IEnumerable<out T> is covariant
List<Dog> dogs = [new("Rex"), new("Buddy")];
IEnumerable<Animal> animals = dogs; // Allowed because Dog derives from Animal

foreach (var animal in animals)
{
    Console.WriteLine(animal.Name);
}

// Contravariance: Action<Animal> can be used as Action<Dog>
// because Action<in T> is contravariant
Action<Animal> printAnimal = a => Console.WriteLine($"Animal: {a.Name}");
Action<Dog> printDog = printAnimal; // Allowed because any Animal handler can handle
Dog

printDog(new Dog("Spot"));
```

- **Ковариантность** (`out T`): `IEnumerable<Dog>` можно использовать, где `IEnumerable<Animal>` ожидается, так как `Dog` производный от `Animal`. Ключевое `out` слово в параметре типа включает это. Ковариантные параметры типа могут появляться только в выходных позициях (возвращаемых типах).
- **Контравариантность** (`in T`): `Action<Animal>` можно использовать, где `Action<Dog>` ожидается, так как любое действие, которое обрабатывает `Animal` также может обрабатываться `Dog`. Ключевое `in` слово включает это. Параметры контравариантного типа могут отображаться только в входных позициях (параметрах).

Многие встроенные интерфейсы и делегаты уже ковариантны: `IEnumerable<out T>`, `IReadOnlyList<out T>`, `Func<out TResult>`, и `Action<in T>`. При работе с этими типами вы автоматически получаете преимущество от вариативности. Для детального рассмотрения проектирования вариантных интерфейсов и делегатов см. Ковариация и контравариация.

Создание собственных универсальных типов

Вы можете определить собственные универсальные классы, структуры, интерфейсы и методы. В следующем примере показан простой универсальный связанный список для

иллюстрации. На практике используйте List<T> или другую встроенную коллекцию:

C#

```
public class GenericList<T>
{
    private class Node(T data)
    {
        public T Data { get; set; } = data;
        public Node? Next { get; set; }
    }

    private Node? head;

    public void AddHead(T data)
    {
        var node = new Node(data) { Next = head };
        head = node;
    }

    public IEnumerator<T> GetEnumerator()
    {
        var current = head;
        while (current is not null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

C#

```
var list = new GenericList<int>();
for (var i = 0; i < 5; i++)
{
    list.AddHead(i);
}

foreach (var item in list)
{
    Console.Write($"{item} ");
}
Console.WriteLine();
// Output: 4 3 2 1 0
```

Универсальные типы не ограничиваются классами. Вы можете определить универсальные `interface` и `struct` `record` типы. Дополнительные сведения о проектировании универсальных алгоритмов и сложных сочетаний ограничений см. в разделе Generics в .NET.

См. также

- Generics в .NET
- коллекционные выражения
- Ограничения параметров типа
- Ковариантность и контравариантность
- System.Collections.Generic

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 13.04.2026

Кортежи и деконструкция

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Начните с руководств Начало работы. При необходимости возвращать несколько значений из метода или группировать значения без определения именованного типа, вы столкнетесь с кортежами.
>
> **Есть опыт на другом языке?** В языке C# кортежи представляют собой типы значений, аналогичные кортежам в Python и Swift, с возможностью использования необязательных именованных элементов и полной поддержкой деконструкции. Просмотрите разделы деконструкции и равенства, чтобы выявить шаблоны, специфичные для C#.

Кортеж группует несколько значений в одну упрощенную структуру, не требуя определения именованного типа. Кортежи — это типы значений, которые можно объявлять встроенными, возвращать из методов и деконструировать в отдельные переменные. Используйте кортежи, если требуется быстрая группирование связанных значений. Например, при возврате нескольких результатов из метода или хранения пары координат.

В следующем примере создается кортеж с именованными элементами, к которым можно обращаться по их именам.

C#

```
var location = (Latitude: 47.6062, Longitude: -122.3321);
Console.WriteLine($"Location: {location.Latitude}, {location.Longitude}");
// Output: Location: 47.6062, -122.3321
```

Кортежи хорошо подходят для кратковременных объединений, в ситуации, когда определение класса, структуры или записи вносило бы необоснованное усложнение. Для долгосрочных концепций предметной области или типов с поведением, предпочитайте записи, классы или структуры. Сравнение того, когда следует использовать каждый из них, см. в разделе "Выбор типа".

Объявление и инициализация кортежей

Объявите кортеж, перечислив типы элементов в скобках. При необходимости можно присвоить имя каждому элементу, чтобы код читался легче.

C#

```
// Tuple with named elements
(string Name, int Age) person = ("Alice", 30);
Console.WriteLine($"{person.Name} is {person.Age} years old");

// Tuple with default element names (Item1, Item2)
(string, int) unnamed = ("Bob", 25);
Console.WriteLine($"{unnamed.Item1} is {unnamed.Item2} years old");

// Tuple declared with var and inline names
var city = (Name: "Seattle", Population: 749_256);
Console.WriteLine($"{city.Name}: population {city.Population}");
```

Если имена не указаны, элементы используют имена `Item1` `Item2` по умолчанию и т. д. Именованные элементы делают ваш код самодокументируемым, что устраняет необходимость в определении отдельного типа.

Выводимые имена элементов

Компилятор выводит имена элементов из имен переменных или имен свойств, используемых для инициализации кортежа. Эта функция позволяет избежать избыточности при совпадении имен:

C#

```
var name = "Carol";
var age = 28;

// The compiler infers element names from the variable names
var person = (name, age);
Console.WriteLine($"{person.name} is {person.age}");
// Output: Carol is 28
```

Выводимые имена сохраняют краткий код. Если требуется другое имя элемента, укажите его явным образом.

Возврат нескольких значений из метода

Одним из наиболее распространенных способов использования кортежей является возврат нескольких значений из метода. Вместо определения класса или использования `out` параметров возвращает кортеж с именованными элементами:

C#

```
static (double Minimum, double Maximum, double Average) ComputeStats(List<double>
values)
{
    var min = values.Min();
    var max = values.Max();
    var avg = values.Average();
    return (min, max, avg);
}
```

Именованные элементы кортежа делают возвращаемые значения читаемыми как на месте вызова, так и в описании метода. Вызывающий может получить доступ к каждому значению по имени без необходимости запоминать позиционный порядок.

Деконструкция кортежей

Деконструкция распаковывает элементы кортежа в отдельные переменные в одном операторе. Кортежи можно деконструировать несколькими способами:

C#

```
var point = (X: 3, Y: 7);

// Deconstruct with var (infer all types)
var (x, y) = point;
Console.WriteLine($"x={x}, y={y}");

// Deconstruct with explicit types
(int px, int py) = point;
Console.WriteLine($"px={px}, py={py}");

// Deconstruct into existing variables
int a, b;
(a, b) = point;
Console.WriteLine($"a={a}, b={b}");

// Deconstruct a method return value directly
List<double> data = [10.0, 20.0, 30.0];
var (min, max, avg) = ComputeStats(data);
Console.WriteLine($"Min: {min}, Max: {max}, Avg: {avg}");
```

Деконструкция особенно полезна при получении кортежа из вызова метода и когда нужно сразу работать с его отдельными значениями.

Кортежи можно деконструировать непосредственно в `foreach` циклах, что делает итерацию по коллекциям сгруппированных значений кратким:

C#

```
List<(string Name, int Score)> results =
[
    ("Alice", 92),
    ("Bob", 87),
    ("Carol", 95)
];

foreach (var (name, score) in results)
{
    Console.WriteLine($"{name}: {score}");
}
```

Если вам не нужен каждый элемент, используйте *отмену* (`_`) вместо каждого значения, которое вы хотите игнорировать. Используйте отдельную `_` позицию для каждой отклоненной позиции:

C#

```
List<double> values = [5.0, 10.0, 15.0];
var (_, max, _) = ComputeStats(values);
Console.WriteLine($"Only need the max: {max}");
// Output: Only need the max: 15
```

Дополнительные сведения об использовании отбраковки в разных контекстах см. в разделе "Отбраковки".

Равенство кортежей

Вы можете сравнить кортежи с помощью `==` и `!=`. Эти операторы последовательно сравнивают каждый элемент, поэтому два кортежа равны, когда все их соответствующие элементы равны.

C#

```
var order1 = (Product: "Widget", Quantity: 5);
var order2 = (Product: "Widget", Quantity: 5);
var order3 = (Product: "Gadget", Quantity: 3);
```

```
Console.WriteLine(order1 == order2); // True
Console.WriteLine(order1 == order3); // False

// Element names don't affect equality—only values matter
var named = (X: 1, Y: 2);
var different = (A: 1, B: 2);
Console.WriteLine(named == different); // True
```

Определенное оператором `==` равенство кортежей используется для каждого типа элемента, что означает, что сравнение корректно работает для строк, чисел и других типов, которые определяют `==`. Имена элементов не влияют на сравнение — важны только значения и позиции.

Недеструктивная мутация с `with`

Выражение `with` создает копию кортежа с измененными одним или несколькими элементами, при этом оригинал остается без изменений.

C#

```
var original = (Name: "Widget", Price: 19.99m, InStock: true);
var discounted = original with { Price = 14.99m };

Console.WriteLine($"Original: {original.Name} at {original.Price:C}");
Console.WriteLine($"Discounted: {discounted.Name} at {discounted.Price:C}");
// Output:
// Original: Widget at $19.99
// Discounted: Widget at $14.99
```

Этот шаблон полезен, если требуется вариант существующего кортежа без изменения исходного. Выражение `with` работает с кортежами так же, как и с записями.

Кортежи в словарях и поисках

Кортежи делают значения словаря удобными, когда необходимо связать ключ с несколькими элементами данных.

C#

```
var sizeChart = new Dictionary<string, (int Min, int Max)>
{
    ["Small"] = (0, 50),
    ["Medium"] = (51, 100),
    ["Large"] = (101, 200)
};
```

```
if (sizeChart.TryGetValue("Medium", out var range))
{
    Console.WriteLine($"Medium: {range.Min}-{range.Max}");
}
// Output: Medium: 51-100
```

Кортежи могут использоваться в качестве *ключей* в словаре, предоставляя возможность создания составного ключа без необходимости определения пользовательского типа. Поскольку кортежи реализуют структурное равенство, поиск осуществляется по комбинированным значениям всех элементов.

C#

```
var grid = new Dictionary<(int Row, int Column), string>
{
    [(0, 0)] = "Origin",
    [(1, 3)] = "Sensor A",
    [(2, 5)] = "Sensor B"
};

var target = (Row: 1, Column: 3);
if (grid.TryGetValue(target, out var label))
{
    Console.WriteLine($"({target.Row}, {target.Column}): {label}");
}
// Output: (1, 3): Sensor A
```

Этот шаблон позволяет избежать необходимости в отдельном классе для простых поисков с несколькими ключами или сопоставлениями значений с несколькими ключами.

Кортежи и анонимные типы

Кортежи — это предпочтительный выбор, если вам нужна упрощенная структура данных без именованных данных. Анонимные типы по-прежнему применяются в сценариях дерева выражений и в коде, который требует ссылочных типов, но кортежи обеспечивают более высокую производительность, поддержку декомпозиции и более гибкий синтаксис. Дополнительные сведения об анонимных типах см. в разделе "Выбор между анонимными типами и типами кортежей".

См. также

- Типы кортежей (справочник по C#) для полных сведений о синтаксисе

- Деконструкция кортежей и других типов для определяемых `Deconstruct` пользователем методов
- Отбрасывания
- Записи

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 13.04.2026

Преобразования типов, приведение и упаковка

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Перед выбором метода преобразования вы практикуетесь в работе с базовыми типами данных.
>
> **Есть опыт на другом языке?** Преобразования C# работают, как и в большинстве статически типизированных языков: расширяющие преобразования выполняются неявно, сужающие преобразования требуют явного приведения типов, а для анализа текста в пользовательском коде рекомендуется использовать `TryParse`.

При написании кода на C# вы часто преобразуете значения из одного типа в другой. Например, можно преобразовать из `int` в `long`, считывать текст и преобразовывать его в число или привести базовый тип к производному типу.

Общие сведения о ключевых терминах:

- *Преобразование* — это процесс изменения значения из одного типа в другой.
- *Приведение* — это явный синтаксис преобразования, написанный с скобками, например `(int)value`.
- *Неявное преобразование* — это преобразование, которое происходит автоматически, когда компилятор может гарантировать безопасность.
- *Явное приведение типов* — это преобразование, написанное в коде, указывающее, что преобразование может потерять информацию или привести к ошибке.

Подберите стиль преобразования, соответствующий ситуации:

- Неявные преобразования происходят автоматически, когда компилятор может гарантировать безопасность — синтаксис не требуется.
- Явные приведения требуются, если данные могут быть потеряны или преобразование может завершиться ошибкой.
- Сопоставление шаблонов применяется, если требуется безопасное преобразование ссылочного типа, которое может не увенчаться успехом, или `as`.
- Api анализа применяются, если исходное значение является текстом.

Неявные и явные числовые преобразования

Неявное преобразование всегда выполняется успешно. *Явное преобразование* может завершиться с ошибкой или потерять информацию.

C#

```
int itemCount = 42;
long widened = itemCount; // Implicit conversion.

double average = 19.75;
int truncated = (int)average; // Explicit cast.

Console.WriteLine($"widened: {widened}, truncated: {truncated}");
```

Явный приведение сообщает читателям, что преобразование может потерять информацию. В примере значение `double` усечено при преобразовании в `int`.

Полные таблицы преобразования см. в разделе Встроенные числовые преобразования.

Преобразование ссылок

Классы являются ссылочными типами. Преобразование типов на них не копирует данные. Они изменяют способ просмотра одного и того же объекта.

Некоторые преобразования ссылок неявны. Компилятор гарантирует, что они безопасны. В трех ситуациях всегда происходит неявное преобразование ссылок: присваивание экземпляра производного класса переменной базового класса (при этом производный тип является подтипом базового типа), присваивание экземпляра ссылочного типа переменной интерфейса, реализуемого этим типом, и присваивание любого ссылочного типа переменной `object`.

C#

```
Mammal otter = new() { Name = "River otter" };
Animal animal = otter;           // Derived to base class: always safe.
INamed named = otter;            // Reference type to implemented interface: always
safe.
object obj = otter;              // Any reference type to object: always safe.

Console.WriteLine(named.Name);
```

Переход к другому направлению из базового типа обратно в производный тип требует явной проверки, так как объект может не быть производным типом, который вы ожидаете.

Предпочитайте сопоставление шаблонов, чтобы тест и назначение произошли вместе.

C#

```
Animal knownAnimal = new Mammal { Name = "River otter" };

if (knownAnimal is Mammal mammal)
{
    Console.WriteLine($"Pattern match succeeded: {mammal.Name}");
}

Animal unknownAnimal = new Reptile();
Console.WriteLine($"Can treat as mammal: {unknownAnimal is Mammal}");
```

Сопоставление шаблонов удерживает успешно приведённую переменную в наименьшей возможной области, что улучшает читаемость.

Если вам нужен результат null вместо условной ветви, используйте `as`:

C#

```
object boxedMammal = new Mammal { Name = "Sea lion" };
Mammal? maybeMammal = boxedMammal as Mammal;
Console.WriteLine(maybeMammal is null ? "Not a mammal" : maybeMammal.Name);

object boxedReptile = new Reptile();
Mammal? noMammal = boxedReptile as Mammal;
Console.WriteLine(noMammal is null ? "Safe null result" : noMammal.Name);
```

Используйте `as` только с ссылочными типами и типами значений, допускающими значение NULL. Возвращается `null` при сбое преобразования.

Понимание упаковки и распаковки

Боксинг преобразует структуру или другой тип значения в `object` или в реализованный интерфейсный тип. Распаковка извлекает тип значения из этой ссылки на объект.

C#

```
int temperature = 72;
object boxedTemperature = temperature; // Boxing.
int unboxedTemperature = (int)boxedTemperature; // Unboxing.

Packet packet = new(7);
ILabelled labelledPacket = packet; // Boxing through an interface reference.

Console.WriteLine($"Unboxed: {unboxedTemperature}, Label: {labelledPacket.Label}");
```

Бокс выделяет память в управляемой куче, а для распаковки требуется проверка типа. В горячих путях избегайте ненужных боксов, так как она добавляет выделения и дополнительную работу.

Анализ текста с помощью Parse и TryParse

При преобразовании входных данных пользователя или содержимого файла начните с `TryParse`. Он избегает исключений для ожидаемых недопустимых входных данных и делает обработку сбоев явной. Все API синтаксического анализа создают новый экземпляр объекта или типа значения из исходной строки; Они не изменяют источник.

C#

```
string textValue = "512";
int parsed = int.Parse(textValue);

string userInput = "12x";
bool parsedSuccessfully = int.TryParse(userInput, out int safeValue);

Console.WriteLine($"parsed: {parsed}");
Console.WriteLine(parsedSuccessfully ? $"safe value: {safeValue}" : "Input is not a
valid number.");
```

Используйте `Parse` , если входные данные гарантированно допустимы, например контролируемые тестовые данные. Используйте `TryParse` для ввода пользователем, сетевых и файловых данных.

API-интерфейсы преобразования основных компонентов

Используйте эти API чаще всего в повседневном коде:

- Int32.Parse(String)
- Int32.TryParse
- Double.TryParse
- Decimal.TryParse
- System.IConvertible

Сведения о расширенном поведении преобразования и всех перегрузках см. в справочнике по API для конкретного типа назначения.

См. также

- Общие сведения о системе типов
- Встроенные типы и литералы
- Сопоставление шаблонов
- Как безопасно приводить типы с помощью сопоставления шаблонов и операторов is и as

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.05.2026

Лямбда-выражения, делегаты и события

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Освойте основные навыки работы с типами и методами перед использованием лямбда-выражений.

Иногда требуется передать небольшую часть поведения, функцию непосредственно другому методу. Например, может потребоваться отфильтровать список, но условие фильтрации изменяется в зависимости от ситуации. Вместо написания отдельного именованного метода для каждого возможного условия вы передаете само условие в качестве аргумента.

Лямбда-выражения — это функция C#, которая делает это возможным. Лямбда-выражение — это компактная встроенная функция, которую вы пишете без указания имени. Оператор `=>` со стрелкой используется для разделения списка параметров из текста:

```
C#
x => x * 2
```

Чтение слева направо: `x` является входным параметром, `=>` означает "ведет к", и `x * 2` является телом. Он вычисляет возвращаемое значение. Если нет параметров или нескольких параметров, заключите их в скобки: `() => 42` или `(left, right) => left + right`.

Делегаты поддерживают лямбда-выражения

Чтобы использовать лямбда-выражение, компилятор C# должен знать две вещи: типы параметров и возвращаемого типа. Это описание, типы параметров и возвращаемый тип, называется *типом делегата*.

Тип делегата — это тип, представляющий сигнатуру метода. Переменная типа делегата может содержать любой подходящий метод, например, лямбда-выражение или именованный метод, если типы его параметров и возвращаемого значения совпадают.

Вы объявляете тип делегата с ключевым словом `delegate` :

C#

```
delegate int Transform(int value);
```

Это объявление говорит: "`Transform` является типом делегата для методов, которые принимают один `int` и возвращают объект `int`". Затем можно назначить лямбда-выражение или именованный метод переменной этого типа:

C#

```
Transform doubler = x => x * 2;    // assign a lambda expression
Transform squarer = Square;         // assign a named method

Console.WriteLine(doubler(5));      // 10
Console.WriteLine(squarer(5));      // 25

static int Square(int value) => value * value;
```

И `doubler`, и `squarer` имеют значение типа `Transform`. Вы вызываете их точно так же, как обычные методы. Компилятор проверяет, совпадает ли то, что вы присваиваете, с объявленной сигнатурой.

Встроенные типы делегатов: `Func` `Action`

Объявление пользовательского типа делегата для каждой ситуации может быть повторяющимся. .NET предоставляет два семейства универсальных типов делегатов, `Func` и `Action`, которые охватывают большинство сценариев, поэтому редко требуется использовать ключевое слово `delegate` самостоятельно.

Оба семейства доступны в версиях от нуля до шестнадцати входных параметров, поэтому они могут масштабироваться до любого количества входных данных. Ключевым различием между двумя семьями является:

- System.Func<T,TResult> (и `Func<T1, T2, TResult>` т. д.) представляет метод, **возвращающий значение**. Последний параметр типа всегда является возвращаемым типом; все предыдущие — это типы входных данных.
- System.Action<T> (и `Action<T1, T2>` т. д.) представляет метод, **возвращающий ничего** (`void`). Все параметры типа — входные типы. System.Action без параметров типа представляет метод без входных данных и не возвращает возвращаемое значение.

Например, `Func<int, int, int>` описывает метод с двумя `int` входными данными и одним `int` результатом. `Action<string>` описывает метод с одним `string` входным и не

возвращаемым значением.

C#

```
Func<int, int, int> add = (left, right) => left + right;
Action<string> report = message => Console.WriteLine($"Report: {message}");

int total = add(5, 9);
report($"5 + 9 = {total}");
```

Используйте описательные имена параметров в лямбда-файлах, чтобы читатели могли понять намерение без сканирования полного текста метода.

Передача лямбда-выражения методу

Когда метод объявляет параметр `Func` или `Action`, вызывающие передают лямбда-выражение, соответствующее типу делегата. Компилятор проверяет, соответствуют ли лямбда-типы параметров и тип возвращаемого значения объявленному типу делегата. Если они не соответствуют, код не компилируется.

C#

```
int[] numbers = [1, 2, 3, 4, 5, 6];
int[] evenNumbers = Filter(numbers, value => value % 2 == 0).ToArray();

Console.WriteLine(string.Join(", ", evenNumbers));
```

Метод `Filter` объявляет `Func<int, bool>` параметр с именем `predicate`. Тип `Func<int, bool>` сообщает вызывающим сторонам ожидаемую структуру: один `int` вход, один `bool` результат. Вызывающий передает `value => value % 2 == 0` в качестве этого аргумента. Этот шаблон используется во множестве API LINQ и .NET.

Делайте лямбда-выражения самодостаточными

Лямбда-выражение может ссылаться на переменные из окружающего кода. *Захват* означает, что лямбда содержит ссылку на переменную, объявленную за пределами собственного тела. Сочетание лямбда и переменных, которые она захватывает, называется *замыканием*.

Когда нет необходимости захватывать что-либо, добавьте модификатор `static` к лямбда-выражению. Статическая лямбда-функция может использовать только собственные

параметры и значения, объявленные внутри тела функции. Не удается записать локальные переменные или состояние экземпляра из включающей области.

C#

```
Func<int, bool> isEven = static value => value % 2 == 0;

Console.WriteLine(isEven(14));
Console.WriteLine(isEven(15));
```

Статические лямбды делают намерение понятным и предотвращают случайный захват.

Используйте параметры игнорирования, когда входные данные не актуальны

Иногда подпись делегата включает параметры, которые вам не нужны. Используйте параметр "отменить" `_`, чтобы явно указать на этот выбор.

Распространенные примеры включают обработчики событий, в которых вы не используете `sender` или `EventArgs`, обратные вызовы, в которых требуется только несколько входных данных, и перегрузки LINQ, которые предоставляют индекс, который не используется.

C#

```
Action<int, int, string> statusUpdate = (_, _, message) => Console.WriteLine(message);
statusUpdate(200, 42, "Operation completed");
```

Исключения улучшают удобочитаемость, так как они показывают, какие параметры имеют значение.

События предоставляют необязательные уведомления

Событие — это механизм, который один объект (*издатель*) использует для уведомления других объектов (*подписчиков*) при возникновении чего-то. Издателю не нужно знать, кто слушает или сколько подписчиков есть. Подписчики решили принять участие.

События основаны на делегатах. Событие является полем делегата с дополнительными ограничениями, применяемыми ключевым словом `event`: внешний код может только подписываться (`+=`) или отменять подписку (`-=`) от события; только класс, объявляющий событие, может инициировать его (raise).

Соглашение .NET для типов делегатов событий — System.EventHandler<TEventArgs>, где `T` — это тип данных, включенных в уведомление. Его подпись всегда имеет два параметра: `sender` (объект, который вызвал событие) и данные события типа `T`.

C#

```
MessagePublisher publisher = new();
publisher.MessagePublished += (_, message) => Console.WriteLine($"Received:
{message}");

publisher.Publish("Records updated");
```

Пошаговое руководство по коду:

- `MessagePublisher` `event EventHandler<string>? MessagePublished` объявляет . Ключевое слово `event` означает, что подписчики могут только подписаться или отменить подписку — вызывать его напрямую нельзя.
- `publisher.MessagePublished += (_, message) => ...` подписывается с использованием лямбда-выражения. Обработчик игнорирует параметр `sender`, поскольку он ему не нужен.
- `publisher.Publish("Records updated")` вызывает событие и запускает все подписанные обработчики.

Подписка является необязательной. `?.Invoke(...)` Метод `Publish` означает, что событие возникает только при подключении по крайней мере одного подписчика. Издатель поднимает событие, не зная или заботясь о том, слушает ли кто-либо.

См. также

- Общие сведения о системе типов
- Методы
- Сопоставление шаблонов
- События (руководство по программированию на C#)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.05.2026

Анонимные типы (руководство по программированию на C#)

Анонимные типы предоставляют удобный способ инкапсулировать набор свойств только для чтения в один объект без определения именованного типа. Компилятор создает имя типа во время компиляции, которое невозможно получить в исходном коде. Компилятор определяет тип каждого свойства.

Создайте анонимные типы с помощью new оператора вместе с инициализатором объектов. В следующем примере показан анонимный тип, инициализируемый двумя свойствами, `Name` и `Age`:

C#

```
var person = new { Name = "Alice", Age = 30 };
Console.WriteLine($"{person.Name} is {person.Age} years old.");
// Output:
// Alice is 30 years old.
```

Выводимые имена свойств

Имена свойств можно указать явным образом с помощью синтаксиса `Name = value` . При инициализации анонимного типа через выражение доступа к элементу или переменной, компилятор определяет имя свойства на основе этого выражения:

C#

```
string productName = "Laptop";
decimal price = 999.99m;
var product = new { productName, price };
Console.WriteLine($"{product.productName}: {product.price:C}");
// Output:
// Laptop: $999.99
```

В приведенном выше примере компилятор выводит `productName` и `price` в качестве имен свойств из имен переменных, используемых в инициализаторе.

Объявление анонимных типов с помощью var

Так как компилятор создает имя типа и не может получить доступ к нему в исходном коде, необходимо использовать var для объявления локальной переменной. Невозможно явно указать имя типа:

C#

```
// You must use var — you can't write a named type here.
var person = new { Name = "Alice", Age = 30 };
```

Использование анонимных типов в запросах LINQ

Анонимные типы чаще всего отображаются в select предложении выражения запроса, где они проектируют подмножество свойств из каждого исходного элемента:

C#

```
var words = new[] { "apple", "blueberry", "cherry" };

var results = words.Select(w => new { Word = w, Length = w.Length });

foreach (var item in results)
{
    Console.WriteLine($"{item.Word} has {item.Length} letters.");
}
// Output:
// apple has 5 letters.
// blueberry has 9 letters.
// cherry has 6 letters.
```

Равенство

Два экземпляра анонимного типа с одинаковыми именами свойств и типами в одном порядке используют один и тот же тип, созданный компилятором. Компилятор переопределяет Equals, так что GetHashCode сравнивает значения свойств, а не идентичность ссылки.

C#

```
var a = new { Name = "Alice", Age = 30 };
var b = new { Name = "Alice", Age = 30 };
var c = new { Name = "Bob", Age = 25 };

Console.WriteLine(a.Equals(b));  // True
Console.WriteLine(a.Equals(c));  // False
```

Вложенные анонимные типы

Анонимные типы могут содержать другие анонимные типы в качестве значений свойств:

C#

```
var order = new
{
    OrderId = 1,
    Customer = new { Name = "Alice", City = "Seattle" },
    Total = 150.00m
};
Console.WriteLine($"Order {order.OrderId} for {order.Customer.Name} in
{order.Customer.City}");
// Output:
// Order 1 for Alice in Seattle
```

Характеристики

Анонимные типы имеют следующие характеристики:

- Компилятор создает их как `internal sealed class` типы, производные от Object.
- Все свойства доступны `public` только для чтения.
- Анонимные типы поддерживают with выражения для недеструктивной мутации.
- Компилятор генерирует переопределения для значений Equals, GetHashCode и ToString.
- Анонимные типы поддерживают деревья выражений, в то время как кортежи не поддерживаются.

Ограничения

Анонимные типы имеют несколько ограничений:

- Их нельзя использовать в качестве типов возвращаемых методов, параметров метода или типов полей, так как не удается назвать тип.
- Они ограничены тем методом, в котором вы их объявляете.
- Нельзя добавлять методы, события или пользовательские операторы.
- Свойства всегда доступны только для чтения; анонимные типы не поддерживают изменяемые свойства.

Когда вместо этого следует использовать кортежи

Для большинства новых кодов рекомендуется использовать кортежи вместо анонимных типов. Кортежи, являясь типами значений, обеспечивают лучшую производительность. Они также обеспечивают поддержку деконструкции и более гибкий синтаксис. Анонимные типы остаются лучшим выбором, если требуется поддержка дерева выражений или семантика ссылочного типа. Подробное сравнение см. в Выбор между анонимными и кортежными типами.

См. также

- Инициализаторы объектов и коллекций
- Неявно типизированные локальные переменные
- Кортежи и деконструкция
- LINQ в C#
- Выбор между анонимными и кортежными типами

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 13.04.2026

Безопасность null в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Исходя из Java или C++?** C# обеспечивает безопасность null во время компиляции с помощью ссылочных типов, допускающих значение NULL. Цель аналогична аннотациям Java `@NonNull`, но их обеспечение осуществляется компилятором. C# также имеет выделенные операторы `?.` и `??`, которые делают null-безопасные выражения краткими.

`null` представляет отсутствие значения. При попытке получить доступ к члену в `null` ссылке путем вызова метода или чтения свойства среда выполнения выдает NullReferenceException.

C#

```
// Accessing a member on null throws NullReferenceException at runtime:
// string? name = null;
// int length = name.Length; // throws NullReferenceException

// Check before you dereference:
string? name = null;
if (name is not null)
{
    Console.WriteLine($"Name has {name.Length} characters.");
}
else
{
    Console.WriteLine("Name has no value.");
}
// Output: Name has no value.
```

C# предоставляет три дополнительных средства для написания кода, безопасного со значением NULL:

- Типы значений допускающие значение null: позволяют типу значения, например `int` или `bool`, также хранить `null`.

- **Ссылочные типы, допускающие значение NULL**, позволяют компилятору отслеживать, может ли ссылка быть `null`
- **Операторы NULL**: безопасный для null доступ и логика запасного варианта

Типы значений, допускающие значение NULL

Такие типы значений, как `int`, `double` и `bool` не могут храниться `null` по умолчанию. Добавьте `?` к имени типа, чтобы создать *nullable тип данных*, который может содержать либо значение, либо `null`:

C#

```
int? score = null;
Console.WriteLine(score.HasValue);               // False

score = 95;
Console.WriteLine(score.HasValue);               // True
Console.WriteLine(score.GetValueOrDefault());    // 95

int? missing = null;
Console.WriteLine(missing.GetValueOrDefault(-1)); // -1
```

Типы значений, допускающие значение NULL, полезны, если базовый тип значения должен представлять "нет данных". Распространенные сценарии включают столбцы базы данных, которые могут быть отсутствуют, необязательные параметры конфигурации и данные датчиков, которые еще не записаны.

Полный охват объявления, проверки и преобразования см. в разделе "Типы значений, допускающих значение NULL".

Ссылочные типы, допускающие значение null

Ссылочные типы, такие как `string` массивы и экземпляры классов, могут содержать `null` во время выполнения. *Ссылочные типы, допускающие значение NULL* — это функция компилятора, которая делает намерение null явным и выявляет ошибки во время компиляции.

Используя аннотацию `?`, вы объявляете свое намерение:

- `string?` — это *возможно* `null` ссылка; компилятор предупреждает, если вы разыменовываете её без предварительной проверки.

- `string` — эта ссылка *не должна* быть `null`; компилятор предупреждает, если ему присваивают `null`.

C#

```
// string?  means this reference might be null
// string   means this reference should not be null
string? nullableName = null;
string  nonNullName  = "Alice";

// ?. safely accesses a member when the reference might be null
string display = nullableName?.ToUpper() ?? "(no name)";
Console.WriteLine(display);          // (no name)

display = nonNullName.ToUpper();     // safe: nonNullName is never null
Console.WriteLine(display);          // ALICE
```

Все проекты .NET, которые создают современные шаблоны SDK, по умолчанию включают ссылочные типы, допускающие значение NULL. Полные рекомендации по включению и аннотации см. в разделе "Ссылочные типы, допускающие значение NULL".

Операторы NULL

C# включает в себя несколько операторов, которые позволяют писать код, безопасный в случае null, без использования ручной проверки на null везде.

Развернуть таблицу

Оператор	Name	Purpose
`?.`	Доступ к члену с условием NULL	Доступ к члену только в том случае, если объект не является NULL
`?[]`	Нулевой условный доступ к индексатору	Доступ к элементу только в том случае, если коллекция не является NULL
`??`	Объединение с значением NULL	Возвращает резервное значение, если выражение имеет значение `null`
`??=`	Присваивание по объединяющему оператору NULL	Назначайте только в том случае, если переменная `null`
`is null` / `is not null`	Шаблон NULL	Предпочтительный тест на null

C#

```
string? city = GetCity();

// ?. — access a member only when non-null
int? len = city?.Length;

// ?? — substitute a default when null
string display = city ?? "unknown";

// is null — preferred null test
if (city is null)
{
    Console.WriteLine("No city provided.");
}
else
{
    Console.WriteLine($"{display} ({len} chars)");
}
// Output: No city provided.
```

Подробные примеры каждого оператора см. в разделе Null operators.

Типы значений с поддержкой NULL и ссылочные типы с поддержкой NULL служат различным целям

Типы значений, допускающие значение NULL, и ссылочные типы, допускающие значение NULL, не являются альтернативными. Они решают различные проблемы:

- Используйте `T?` для типа значения, который должен представлять "нет значения". Например, используйте `int?` для необязательного столбца базы данных или для `DateTime?` события, которое еще не запланировано.
- Используйте `string?` и другие аннотации допускающие значение NULL, чтобы показать, что ссылка *может* быть `null`, чтобы компилятор мог предупреждать вас до того, как произойдет `NullReferenceException` во время выполнения.

Вместе эти функции и операторы NULL дают полный набор инструментов для записи кода C#, безопасного со значением NULL.

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 11.05.2026

Типы значений, допускающие значение NULL: основы C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые знают по крайней мере один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы. Дополнительные сведения см. в разделе "Типы значений, допускающие значение NULL" в справочнике по языку.

Тип `T?`, представляет все значения базового типа `T` значения, а также дополнительное `null` значение. Переменная типа `int?` может содержать любое целое число или `null`, чтобы обозначить "нет значения".

Такие типы значений, как `int`, `bool` и `DateTime` не могут храниться `null` по умолчанию. Это поведение эффективно и предотвращает множество ошибок. Однако это ограничение создает проблему, когда значение может быть действительно отсутствует. Распространенный сценарий считывается из базы данных: целочисленный столбец может содержать число или не содержать значения вообще (`NULL` в SQL). Простое `int` не может выразить это отсутствие, но `int?` может.

Объявление типа значения, принимающего NULL

Добавьте `?` к любому типу значения, чтобы сделать его пустым:

C#

```
int?    age      = null;    // integer with no value yet
double? price    = 9.99;    // nullable double with a value
bool?   isActive = null;    // boolean with no value

age = 30;                   // assign a value later

int?[] scores = [100, null, 85, null, 72]; // array with absent entries
```

Значение по умолчанию для типа значения, допускающего NULL, — это `null`, а не значение по умолчанию базового типа.

Проверка наличия значения

Рекомендуемый способ проверки типа значения, допускающего значение NULL, и извлечения его значения - с *шаблоном типа*:

C#

```
int? temperature = 72;

if (temperature is int degrees)
{
    Console.WriteLine($"Temperature is {degrees}°F.");
}
else
{
    Console.WriteLine("Temperature is not recorded.");
}
// Output: Temperature is 72°F.
```

Шаблон `is int degrees` совпадает только тогда, когда `temperature` не равно null, и при этом связывает его с `degrees`. Вы получаете как проверку NULL, так и извлечение значений на одном шаге.

В качестве альтернативы, используйте свойства `HasValue` и `Value`.

C#

```
int? count = 42;

if (count.HasValue)
{
    Console.WriteLine($"Count is {count.Value}.");
}
else
{
    Console.WriteLine("Count has no value.");
}
// Output: Count is 42.
```

Предпочитайте шаблон `is T value` для нового кода. В ней представлена новая переменная, не допускающая значения NULL, ограниченная соответствующей ветвью, что делает намерение более понятным и устраняет любой соблазн случайно использовать `Value` за пределами проверки NULL, где он вызовет InvalidOperationExceptionисключение.

Вы также можете сравнить напрямую с `null`:

C#

```
int? quantity = null;

if (quantity != null)
{
    Console.WriteLine($"Quantity: {quantity.Value}");
}
else
{
    Console.WriteLine("Quantity is not set.");
}
// Output: Quantity is not set.
```

Получение значения с резервным вариантом

Если требуется значение, не допускающее NULL, из значения, допускающего NULL, используйте `GetValueOrDefault` или null-объединяющий оператор `??`.

C#

```
int? rating = null;

int result1 = rating.GetValueOrDefault();    // 0 (default for int)
int result2 = rating.GetValueOrDefault(-1);  // -1 (specified fallback)

Console.WriteLine(result1); // 0
Console.WriteLine(result2); // -1

rating = 5;
int result3 = rating.GetValueOrDefault(-1);  // 5 (actual value)
Console.WriteLine(result3); // 5
```

Оператор `??` часто выглядит более аккуратно в строку.

C#

```
int? priority = null;

int effective = priority ?? 0;  // 0 because priority is null
Console.WriteLine(effective);   // 0

priority  = 3;
effective = priority ?? 0;      // 3 because priority has a value
Console.WriteLine(effective);   // 3
```

Оба подхода возвращают фактическое значение, когда оно присутствует, и запасное значение, которое вы указали, когда его нет.

Арифметика с типами значений, допускающими значение NULL

Арифметические операторы и операторы сравнения для типов значений, допускающих значение NULL, *повышаются*: если один из операндов `null`, результат будет `null`, а не ошибка.

C#

```
int? a = 10;
int? b = 20;
int? c = null;

int? sum     = a + b;   // both non-null: result is 30
int? product = a * c;   // one operand is null: result is null

Console.WriteLine(sum);               // 30
Console.WriteLine(product.HasValue);  // False — null propagates through arithmetic
```

Значение NULL распространяется по умолчанию с помощью арифметики. Чтобы предотвратить каскадный результат null, извлеките значение с `??` или `GetValueOrDefault` перед его использованием в вычислении.

См. также

- Общие сведения о безопасности NULL
- Операторы NULL
- Типы значений, допускающие значение NULL (справочник по языку)
- Ссылочные типы, допускающие значение NULL

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 11.05.2026

Ссылочные типы, допускающие значение null

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Начните с руководств Начало работы.
>
> **Есть опыт на другом языке?** Если вы работали с nullable-типами Kotlin, `strictNullChecks` в TypeScript или опциональными типами Swift, эта модель вам знакома. C# использует статический анализ и предупреждающую диагностику вместо отдельного типа. Бегло ознакомьтесь с выражением намерений с помощью аннотаций и анализом null-состояния, а затем перейдите к руководству: выражение проектных намерений с помощью ссылочных типов, допускающих и не допускающих значение NULL, чтобы применить эту возможность.

Ссылочные типы, допускающие значение NULL — это набор возможностей, которые сводят к минимуму вероятность того, что ваш код вызовет System.NullReferenceException. Вы объявляете, какие переменные предназначены для хранения `null` и которые не являются, и компилятор предупреждает, когда эти объявления не соответствуют их применению в коде. Поведение среды выполнения программы не изменилось. Ссылочные типы, допускающие значение NULL, полностью являются функцией времени компиляции.

Три базовых компонента работают вместе:

- *Аннотации переменных* (`string` vs. `string?`) показывают, какие ссылки должны допускать `null`.
- *Анализ состояния NULL* отслеживает, является ли значение выражения *не* null или , *возможно, null* в каждой точке кода.
- *Атрибуты* в API описывают более подробные контракты, такие как "этот аргумент может быть `null`, но возвращаемое значение равно NULL только в том случае, если аргумент имеет значение NULL".

Компилятор объединяет эти сигналы для создания диагностики. Предупреждения о переменной, не допускающей значения null, означают, что переменная может получить значение `null`. Предупреждения о переменной, допускающей значение NULL, означают,

что код может *разыменовывать* его без проверки null. *Разыменование* означает использование значения, на которое ссылается переменная. Например, чтобы вызвать у него метод (`variable.Method()`), прочитать свойство (`variable.Property`) или обратиться к нему по индексу (`variable[0]`). Разыменование переменной, которая имеет значение `null`, вызывает исключение во время выполнения программы. Любое предупреждение означает, что поведение кода не соответствует заданному дизайну.

Контекст, допускающий значение NULL

Проекты, созданные на основе новых шаблонов .NET, устанавливают `<Nullable>enable</Nullable>` в файле проекта, поэтому рекомендации, приведённые в этой статье, применимы в том виде, в котором они представлены. Если вы работаете в более старой версии проекта, откройте `.csproj` и проверьте `<PropertyGroup>` , содержит ли она следующую строку; добавьте ее, если она отсутствует:

XML

```xml
<Nullable>enable</Nullable>
```

Дополнительные сведения о переносе большого приложения см. в статье о стратегиях миграции, допускающих значение NULL , для получения дополнительных параметров и директив.

Выражение намерения с заметками

Каждая переменная ссылочного типа по умолчанию *не допускает значение NULL* . Добавьте `?`, чтобы объявить *ссылочный тип, допускающий значение NULL*:

C#

```
public static void Annotations()
{
    string required = "always set";   // non-nullable: assigning null produces a
warning
    string? optional = null;          // nullable: holding null is allowed

    Console.WriteLine(required.Length);

    if (optional is not null)
    {
        Console.WriteLine(optional.Length);
    }
}
```

Заметка не изменяет тип среды выполнения. `string` и `string?` оба являются System.String. `?` сообщает компилятору о вашем замысле. Это намерение формирует предупреждения, создаваемые компилятором:

- Переменная, не допускающая значения NULL, имеет состояние *null* по умолчанию *не-null*. Компилятор предупреждает, если вы назначите значение, которое может быть `null`.
- Переменная, допускающая значение null, по умолчанию имеет *null-состояниевозможно, null*. Компилятор предупреждает, если вы разыменовываете переменную, не проверяя ее.

Используйте заметку, чтобы сделать обязательные и необязательные значения видимыми в системе типов. В следующем типе `FirstName` и `LastName` объявлены как не допускающие значение NULL — у каждого человека должны быть оба, — а `MiddleName` как допускающий значение NULL, поскольку он есть не у всех:

C#

```
public sealed class Person(string firstName, string lastName)
{
    public string FirstName { get; } = firstName;
    public string? MiddleName { get; init; }
    public string LastName { get; } = lastName;

    public override string ToString() => MiddleName is null
        ? $"{FirstName} {LastName}"
        : $"{FirstName} {MiddleName} {LastName}";
}

public static void DesignIntent()
{
    Person p1 = new("Ada", "Lovelace") { MiddleName = "King" };
    Console.WriteLine(p1);
    // Output: Ada King Lovelace

    Person p2 = new("Grace", "Hopper");
    Console.WriteLine(p2);
    // Output: Grace Hopper
}
```

Аннотации определяют реализацию `ToString`. Так как `FirstName` и `LastName` не допускают значения null, переопределение использует их непосредственно в интерполированной строке (`$"..."` — синтаксис, который внедряет выражения в `{}` места заполнения) без проверки на null. `MiddleName` допускает значение null, поэтому переопределение сначала сравнивает его с `null` и включает его только если оно присутствует. Компилятор

обеспечивает соблюдение этого различия: код, который передаёт значение, которое может быть null, в место, где ожидается значение, не допускающее null, вызывает предупреждение, и конструктор, который оставляет неинициализированным член, не допускающий null, также вызывает предупреждение.

Анализ нулевого состояния

Компилятор отслеживает *состояние NULL* каждого выражения. Состояние является одним из двух значений:

- *not-null*: известно, что выражение не равно `null`.
- *может иметь значение NULL*: выражение может быть `null`.

Состояние null локальной переменной обновляется при анализе кода компилятором. Это меняют две вещи: **присваивания** и **проверки на null**. После присваивания null-состояние переменной соответствует выражению в правой части. Если выражение равно null или допускает значение null, переменная может иметь значение null. Если выражение является ненулевым литералом, переменная становится ненулевой. После проверки NULL состояние переменной отражает любую ветвь.

C#

```
public static void NullStateTracking()
{
    string? message = null;

    // Warning: dereference of a possibly null reference.
    Console.WriteLine(message.Length);

    message = "Hello, World!";

    // No warning: the compiler tracks that message is now not-null.
    Console.WriteLine(message.Length);
}
```

В предыдущем примере первое разыменовывание вызывает предупреждение, так как `message` имеет значение *maybe-null*. После присваивания ненулевого литерала компилятор знает, что `message` *не равно null*, поэтому второе разыменование безопасно.

Анализ нулевых состояний работает с проверками `if`, сопоставлением с образцом (выражениями, такими как `is null` или `is { }`, которые проверяют структуру значения), а также в потоке управления с циклами или ранним выходом:

C#

```
public sealed class Node(string name)
{
    public string Name { get; } = name;
    public Node? Parent { get; init; }
}

public static void FlowAnalysis(Node start)
{
    Node? current = start;
    while (current is not null)
    {
        // Inside the loop, the compiler knows current is not-null.
        Console.WriteLine(current.Name);

        current = current.Parent;
    }
}
```

Анализ не выполняется внутри тел методов. Если вам нужно сообщить вызывающему коду о возможном значении null, используйте в сигнатуре метода атрибуты анализа допускающих значение null.

Игнорируйте предупреждения с помощью `!`

Иногда вы знаете больше, чем компилятор. *Оператор,допускающий* `!` значение NULL, объявляет, что выражение *не равно NULL*, даже если анализ говорит в противном случае:

C#

```
public static void NullForgiving()
{
    // "ada" matches a switch arm that returns a non-null string,
    // but the return type is string? so the compiler treats the
    // result as maybe-null.
    string? maybeName = LookUpName("ada");

    // The ! tells the compiler "trust me, this isn't null." We just
    // passed "ada", which the switch maps to "Ada Lovelace".
    int length = maybeName!.Length;
    Console.WriteLine(length); // => 12
}

// Returns string? because the wildcard arm yields null.
private static string? LookUpName(string id) => id switch
{
    "ada" => "Ada Lovelace",
    _ => null,
};
```

Используйте `!` только в крайнем случае. Каждый такой случай — это место, где компилятор уже не сможет вас защитить. Предпочитайте добавлять проверку null, реструктуризацию кода или аннотировать соответствующий API, чтобы компилятор достиг правильного вывода самостоятельно.

Атрибуты, описывающие контракты API

Аннотации параметра или возвращаемого значения не всегда достаточно информативны. Метод может принимать аргумент, который может быть null, но гарантировать ненулевой результат. Метод теста может возвращать `true` только в том случае, если его аргумент не имеет значения NULL. Используйте атрибуты анализа, допускающие значение NULL , для передачи этих контрактов:

C#

```
public static bool IsPresent([NotNullWhen(true)] string? value) =>
    !string.IsNullOrEmpty(value);

public static void NullAnalysisAttributes()
{
    string? input = ReadInput();

    if (IsPresent(input))
    {
        // No null-forgiving operator needed: the attribute tells the compiler
        // input is not-null when IsPresent returns true.
        Console.WriteLine(input.Length);
    }
}

private static string? ReadInput() => "hello";
```

NotNullWhenAttribute сообщает компилятору, что, когда `IsPresent` возвращает `true`, аргумент *не равен null*. В блоке `if` компилятор обрабатывает `value` как *ненуловый* оператор без обязательного оператора, допускающего значение NULL. По состоянию на .NET 5 все API среды выполнения .NET аннотированы, поэтому анализ дает преимущества любому коду, который вызывает их.

Известные ошибки

Два сценария могут привести к тому, что недопускающая значения NULL ссылка будет содержать `null` без предупреждения. Оба шаблона являются ограничениями статического анализа, а не ошибок в коде.

Структуры по умолчанию

Можно создать структуру с полями ссылок, не допускающими значение NULL, с помощью `default` или `new()`. Этот подход оставляет поля структуры неинициализированными:

C#

```
public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static void DefaultStructPitfall()
{
    Student s = default;            // No warning, but FirstName and LastName are
null.
    Console.WriteLine(s.FirstName?.Length ?? -1);
}
```

Поля содержат `null` на этапе выполнения, но компилятор не выдаёт предупреждения. Если необходимо использовать структуру, отдавайте предпочтение обязательным членам — членам, которые вызывающий код должен инициализировать с помощью инициализатора объекта, — либо параметризованному конструктору, который вызывающий код обязан вызвать.

Массивы ссылок и структур

Новый массив ссылочного типа, не допускающего значения NULL, содержит все `null` элементы, пока не назначьте каждый из них:

C#

```
public static void ArrayPitfall()
{
    string[] values = new string[3];      // Elements are null at run time.
    Console.WriteLine(values[0]?.Length ?? -1);

    string[] initialized = ["a", "b", "c"]; // Collection expression initializes
every slot.
    Console.WriteLine(initialized[0].Length);
}
```

Та же ошибка применяется к массивам структур: каждый элемент начинается с значения по умолчанию структуры, поэтому все поля ссылок, не допускающие значения NULL,

начинаются с `null` каждого элемента.

Инициализируйте элементы массива при создании массива. Выражения-коллекции (литеральный синтаксис `[1, 2, 3]`) и типизируемый по целевому типу new (запись `new()`, когда компилятор может вывести тип) делают полную инициализацию краткой.

Связанный контент

- Руководство: как выразить замысел проекта с помощью ссылочных типов, допускающих и не допускающих значения NULL
- Разрешение предупреждений, допускающих значение NULL
- Стратегии миграции, допускающие значение NULL
- Атрибуты статического анализа, допускающие значение NULL
- Устранение предупреждений о допустимости значений NULL (справочник по компилятору)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.05.2026

Операторы null в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые знают по крайней мере один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы. Для полного справочника по операторам см. раздел о операторах доступа к членам и операторах объединения null в справочнике по языку.

C# предоставляет несколько операторов, которые обеспечивают краткий null-безопасный код. Вместо использования вложенных условных операторов `if (x != null)` в коде, эти операторы позволяют выразить доступ с проверкой null, значения по умолчанию и проверки на null в одном выражении.

В этой статье рассматриваются `?.` и `?[]` для условного доступа с null, `??` для объединения null, `??=` для присваивания с объединением null, а также `is null`/`is not null` для сопоставления null-шаблонов.

Доступ к члену с null-условием `?.`

Оператор `?.` обращается к члену только в том случае, если объект не имеет значения NULL. Если объект имеет значение `null`, все выражение вычисляется в `null`, вместо того чтобы выдать NullReferenceException:

C#

```
string? name = null;

// Without ?., accessing a member on null throws NullReferenceException:
// int len = name.Length; // throws if name is null

// ?. returns null instead of throwing:
int? len = name?.Length;
Console.WriteLine(len.HasValue); // False

name = "C#";
Console.WriteLine(name?.Length); // 2
```

Оператор `?.` *укороченное вычисление*: когда левая часть равна `null`, все справа пропускается. Вызовы метода не выполняются и побочные эффекты не возникают.

Вы можете связать несколько `?.` операторов в одном выражении. Цепочка останавливается при обнаружении первого `null`.

C#

```
string? input = null;

// Chain ?. across multiple method calls — short-circuits at the first null:
string? upper = input?.Trim()?.ToUpperInvariant();
Console.WriteLine(upper ?? "(none)"); // (none)

input = "  hello  ";
Console.WriteLine(input?.Trim()?.ToUpperInvariant()); // HELLO
```

Доступ к индексатору NULL `?[]`

Оператор `?[]` применяет тот же механизм сокращённого вычисления к индексатору и доступу к массиву. Используйте его, если сама коллекция может быть `null`:

C#

```
string[]? tags = null;

// ?[] accesses an element only when the collection is non-null
string? first = tags?[0];
Console.WriteLine(first ?? "(none)"); // (none)

tags = ["csharp", "dotnet", "nullable"];
Console.WriteLine(tags?[0]);          // csharp
```

Цепные null-условные операторы

Используйте цепочку нескольких операторов `?.`, чтобы обойти путь с потенциально пустыми ссылками. Цепь закорачивается на первом `null`:

C#

```
var order = new Order("ORD-001", null);

// Each ?. short-circuits when null: Customer is null, so Address and City are never
accessed
string? city = order.Customer?.Address?.City;
Console.WriteLine(city ?? "(no city)"); // (no city)
```

```
var fullOrder = new Order("ORD-002",
    new Customer("Alice", new Address("123 Main St", "Springfield", "IL")));

Console.WriteLine(fullOrder.Customer?.Address?.City); // Springfield
```

Если `Customer` равно `null`, ни `Address`, ни `City` не вычисляются. Возвращается целое выражение `null`.

Потокобезопасный вызов делегата

`?.` предоставляет чистый, потокобезопасный способ вызова делегата или вызова события. Выражение делегата вычисляется только один раз, поэтому нет окна для другого потока отмены подписки между проверкой NULL и вызовом:

C#

```
EventHandler? clicked = null;

// No subscribers — ?.Invoke does nothing instead of throwing NullReferenceException
clicked?.Invoke(null, EventArgs.Empty);

clicked += (_, _) => Console.WriteLine("Button clicked!");

// With a subscriber — ?.Invoke calls the handler
clicked?.Invoke(null, EventArgs.Empty);
// Output: Button clicked!
```

Этот шаблон заменяет старый `if (clicked != null) clicked(...)` идиом.

Объединение с значением NULL `??`

Оператор `??` возвращает свой левый операнд, когда он не имеет значения NULL, и его правый операнд, когда слева `null`. Используйте его для предоставления значения по умолчанию:

C#

```
string? username = null;

// ?? returns the right-hand value when the left-hand is null
string display = username ?? "Guest";
Console.WriteLine(display); // Guest

username = "alice";
```

```
display  = username ?? "Guest";
Console.WriteLine(display); // alice
```

`??` является правым ассоциативным, поэтому `a ?? b ?? c` интерпретируется как `a ?? (b ?? c)`. Первое значение, отличное от NULL, выигрывает. Распространенный шаблон заключается в использовании связи `?.` с `??`: сначала используется `?.` для безопасного обхода потенциальной цепочки null, а затем `??` для замены на значение по умолчанию, если цепочка вернула `null`. Полный пример см. в разделе "Объединение операторов NULL".

Операция присваивания при null-слиянии `??=`

Оператор `??=` назначает правое значение переменной только в том случае, если переменная имеет значение `null`. Используйте его для отложенной инициализации:

C#

```
List<string>? cache = null;

// ??= assigns only when the variable is null
cache ??= LoadData();
Console.WriteLine(cache.Count); // 3

// cache is already non-null, so LoadData() isn't called again
cache ??= LoadData();
Console.WriteLine(cache.Count); // 3

static List<string> LoadData() => ["alpha", "beta", "gamma"];
```

Выражение справа вычисляется только в том случае, если переменная имеет значение `null`. Если переменная уже имеет значение, правая часть не вычисляется.

Условное назначение null (C# 14)

Начиная с C# 14, можно использовать `?.` и `?[]` в качестве целевых объектов назначения. Назначение выполняется только в том случае, если левый объект не имеет значения NULL:

C#

```
AppConfig? config = new AppConfig();

// Assigns only when config is non-null (C# 14)
config?.Theme = "dark";
Console.WriteLine(config?.Theme); // dark
```

```
AppConfig? missing = null;
missing?.Theme = "light";                          // no-op: missing is null
Console.WriteLine(missing?.Theme ?? "(no config)"); // (no config)
```

Правая сторона вычисляется только в том случае, если известно, что левая сторона не является NULL.

Сопоставление шаблонов NULL: `is null` и `is not null`

Шаблоны `is null` и `is not null` проверяют, является ли выражение `null`:

C#

```
string? input = null;

// is null is the preferred test — unaffected by operator overloading
if (input is null)
{
    Console.WriteLine("No input provided.");
}

// == null also works, but a custom == operator can change its behavior
if (input == null)
{
    Console.WriteLine("Still no input.");
}
```

Предпочитайте `is null` вместо `== null` для проверки на NULL. Оператор `==` может быть перегружен, то есть `x == null` может возвратить `true` даже в том случае, если `x` не `null`, когда тип определяет настраиваемый оператор равенства. Шаблон `is null` всегда проверяет фактическую ссылку null независимо от перегрузки оператора.

C#

```
string? value = "hello";

if (value is not null)
{
    Console.WriteLine(value.ToUpper()); // HELLO
}
```

Комбинирование операторов NULL

На практике вы часто объединяете несколько из этих операторов. Одно выражение может безопасно пройти по графу глубокого объекта, применить резервный вариант, а затем защитить результат:

C#

```
Order? order = GetPendingOrder();

// Chain ?. for safe traversal, ?? for a fallback, is null for a clear guard
string city = order?.Customer?.Address?.City ?? "unknown";

if (order is null)
{
    Console.WriteLine("No pending order.");
}
else
{
    Console.WriteLine($"Shipping to: {city}");
}
// Output: No pending order.
```

Оператор, игнорирующий null `!`

Оператор `!` postfix подавляет предупреждения, допускающие значение NULL. Добавьте, `!` чтобы сообщить компилятору, что это выражение определенно не равно NULL. Оператор не действует во время выполнения. Это влияет только на анализ состояния null компилятора.

C#

```
string? name = FindUser("alice");

// Use ! only when you have information the compiler doesn't.
// FindUser guarantees a non-null result for known usernames.
int length = name!.Length;
Console.WriteLine(length); // 5
```

Используйте `!` экономно, и только в тех случаях, когда у вас есть информация, которой нет у компилятора. Примеры включают тесты, которые намеренно используют `null` для проверки логики аргументации, или вызов метода, контракт которого гарантирует ненулевой возврат для заданного входного значения. Чрезмерное использование `!` сводит на нет цель nullable ссылочных типов. Полное объяснение см. в разделе " Ссылочные типы, допускающие значение NULL".

См. также

- Общие сведения о безопасности NULL
- Обнуляемые типы значений
- Ссылочные типы, допускающие значение NULL
- Операторы доступа к элементам (справочник по языку)
- Операторы объединения null (справочник по языку)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 11.05.2026

Устранить предупреждения о допустимости значения NULL

> **Совет**
>
> **Впервые сталкиваетесь с типами-ссылками, допускающими значение NULL?** Сначала ознакомьтесь с ссылочными типами, допускающими значение NULL, чтобы понять аннотации и анализ значений NULL. В этой статье предполагается, что вы видите предупреждения в проекте, где включена эта функция.
>
> **Ищете определенный код ошибки компилятора?** Справочная статья Устранение предупреждений о допускающих значение NULL типах перечисляет все предупреждения CS86xx и соответствующие способы их устранения.

При включении ссылочных типов, допускающих значение NULL, компилятор выдает предупреждения везде, где поведение кода не соответствует его заметкам. Большинство предупреждений попадают в небольшой набор шаблонов. После распознавания шаблона исправление обычно является одним из пяти методов:

- Добавьте флажок NULL.
- Добавьте или удалите аннотацию `?` или `!`.
- Добавьте атрибут, описывающий пустой контракт.
- Правильно инициализируйте переменные.
- Проверьте параметр проекта.

В этой статье рассматриваются все методики с репрезентативным примером. Цель не в том, чтобы подавлять предупреждения. Это делает намерение обработки null кода явным, чтобы компилятор достиг те же выводы, что и у вас.

Состояние NULL: то, что компилятор отслеживает

Прежде чем перейти к этим приёмам, полезно понять, как компилятор отслеживает потенциальные нарушения null-состояния. Когда он считывает код, компилятор отслеживает *состояние null* каждого выражения: его анализ о том, может ли выражение находиться `null` в этом месте в коде. Состояние NULL является одним из двух значений:

- *not-null* — компилятор может доказать, что выражение не `null` здесь. Его можно безопасно использовать без проверки.
- *может быть null* — компилятор не может исключить `null`. Использование выражения без проверки выдает предупреждение.

Переменная в состоянии NULL изменяется, так как компилятор следует коду. Метод, который может вернуть `null`, возвращает результат *maybe-null*. Проверка `if (x is not null)` сужает `x` до *not-null* в блоке `if`. Предупреждения, которые вы видите, означают, что компилятор сообщает вам: он определил, что выражение находится в состоянии *maybe-null*, а вы собираетесь использовать его так, как будто оно находится в состоянии *not-null*. Каждый метод в остальной части этой статьи отличается от способа предоставления компилятору сведений, необходимых для обеспечения того, чтобы выражение *не равно NULL* перед его использованием.

Добавить проверку на null

Наиболее распространённое предупреждение — *возможное разыменование значения null*. Компилятор отследил null-состояние переменной до *возможно, null* и обнаружил, что переменная использовалась без проверки:

C#

```
public static int LengthOfMessageUnsafe(string? message)
{
    // Warning CS8602: dereference of a possibly null reference.
    return message.Length;
}
```

Обычно исправление — это *защитное условие*. *Защитная проверка* — это проверка в начале метода или блока, которая выполняет return или генерирует исключение, если входные данные недопустимы. Продолжается только безопасный путь. После выполнения проверки компилятор обновляет состояние NULL переменной на *ненул* в безопасном пути:

C#

```
public static int DereferenceFixed(string? message)
{
    if (message is null)
    {
        return 0;
    }

    // No warning: the compiler knows message is not-null on this path.
```

```
    return message.Length;
}
```

Сопоставление шаблонов (например, выражения, такие как `is null` или `is { }` которые проверяют форму значения), `??` и `??=` включают в себя значения NULL:

C#

```
public static int NullOperatorsFix(string? message)
{
    // ?. evaluates to null if message is null; ?? supplies the fallback value.
    int length = message?.Length ?? 0;

    // Pattern matching narrows the type on the matching branch.
    if (message is { Length: > 0 })
    {
        length = message.Length;
    }

    return length;
}
```

Шаблон `{ Length: > 0 }` свойств совпадает только в том случае, если `message` свойство не равно null , *а* его `Length` свойство больше нуля, поэтому компилятор обрабатывает `message` как *ненулевое* `if` значение внутри блока. Более простой `is not null` тест создает то же сужение состояния NULL, не проверяя какие-либо свойства.

Подробный обзор операторов см. в разделе "Операторы NULL".

Настройка заметок

Компилятор также предупреждает вас, когда в вашем коде выражение *maybe-null* присваивается переменной, не допускающей значение NULL. Это предупреждение означает одно из двух вещей:

- Переменная должна разрешать значения NULL. В этом случае добавьте `?` к типу.
- Выражение никогда не создает значение NULL. Аннотируйте API, который его создал.

C#

```
public static void AssignmentWarning()
{
    // Warning CS8600: converting null literal or possible null value to non-nullable
type.
    string name = Lookup("nobody");
```

```
    Console.WriteLine(name);
}
```

Если `Lookup` законно возвращает значение NULL, измените сайт вызова, чтобы принять отсутствующее значение:

C#

```
public static void AssignmentFixed()
{
    string? name = Lookup("somebody");
    if (name is not null)
    {
        Console.WriteLine(name);
    }
}
```

Если `Lookup` никогда не возвращает значение NULL, измените ее сигнатуру, чтобы вернуть тип ссылок, не допускающий значение NULL. Сценарии, в которых состояние NULL возвращаемого значения зависит от входных данных, см. в следующем разделе по атрибутам null-analysis.

Используйте оператор null-forgiving `!` только в тех случаях, когда вы можете гарантировать, что значение не является null, но не можете выразить эту гарантию в системе типов. Каждый `!` — это место, где компилятор больше не может вас защитить, поэтому лучше добавить проверку или аннотировать исходный API.

Добавление атрибута null-analysis

Иногда правильное исправление находится не в месте вызова. Подпись метода не фиксирует связь между входными и выходными данными достаточно точно, а компилятор выдает предупреждения внутри в противном случае безопасного кода:

C#

```
public static bool IsPresent(string? text) =>
    !string.IsNullOrEmpty(text);

public static void CallerWithoutAttribute(string? text)
{
    if (IsPresent(text))
    {
        // Warning CS8602: dereference of a possibly null reference.
        // The signature doesn't tell the compiler text is not-null here.
        Console.WriteLine(text.Length);
```

```
    }
}
```

Тело `IsPresent` доказывает, что аргумент не равен null, когда метод возвращает `true`, но в сигнатуре это не указано. Добавьте атрибут анализа, допускающий значение NULL , чтобы сделать часть контракта API:

C#

```
public static bool AttributedIsPresent([NotNullWhen(true)] string? text) =>
    !string.IsNullOrEmpty(text);

public static void CallerWithAttribute(string? text)
{
    if (AttributedIsPresent(text))
    {
        // No warning: the attribute tells the compiler text is not-null.
        Console.WriteLine(text.Length);
    }
}
```

К общим атрибутам относятся:

- NotNullWhenAttribute — аргумент *не равен NULL*, если метод возвращает указанное логическое значение.
- NotNullIfNotNullAttribute — возвращаемое значение *не равно NULL* , если именованный аргумент не имеет значения *NULL*.
- MemberNotNullAttribute — перечисленные члены *не имеют значения NULL* после возврата метода.
- DoesNotReturnAttribute — метод никогда не возвращается нормально (например, он всегда вызывает).

Полный список содержит атрибуты статического анализа, допускающие значение NULL.

Инициализация элементов, не допускающих значение NULL

Предупреждение конструктора означает ненулевое поле, свойство или автоматическое свойство (свойство, которое использует поле резервного копирования, созданное компилятором, например `public string Name { get; set; }`) выходит из конструктора без назначения ненулевого значения:

C#

```
public class PersonUninitialized
{
    // Warning CS8618: Non-nullable property 'Name' is uninitialized.
    public string Name { get; set; }
}
```

Вы можете решить эту проблему несколькими способами. Выберите тот, который лучше всего соответствует вашему намерению проектирования.

Требовать значение в качестве аргумента конструктора. Используйте основной конструктор (параметры, объявленные в самом типе, доступные в тексте) или обычный конструктор, который инициализирует свойство:

C#

```
public class PersonInjected(string name)
{
    public string Name { get; } = name;
}
```

Задайте свойство `required`. Вызывающий код должен инициализировать его с помощью инициализатора объектов (синтаксис `{ Property = value }`, следующий за `new`):

C#

```
public class PersonRequired
{
    public required string Name { get; init; }
}
```

Инициализируйте со значением по умолчанию. Если у типа есть осмысленное пустое значение, инициализируйте его при объявлении:

C#

```
public class PersonInitialized
{
    public string Name { get; set; } = "John Doe";
}
```

> **Совет**
>
> Выберите этот метод только в том случае, если тип имеет действительно хорошее значение по умолчанию: тот, который является допустимым, полнофункциональным

экземпляром для пользователей. Примеры включают пустые коллекции. Не придумывайте *sentinel* (специальное значение-заполнитель, например String.Empty, `"N/A"`, `"unknown"` или `-1`, которое вы рассматриваете как «нет значения») в качестве замены для `null`: это убирает предупреждение, но тогда каждый вызывающий код должен знать о sentinel и проверять его, а система типов здесь не поможет. Если не существует хорошего значения по умолчанию, сделайте свойство nullable вместо этого.

Сделайте свойство допускающим значение null. Если значение действительно может быть пропущено, измените тип на nullable:

C#

```
public class PersonOptional
{
    public string? Name { get; set; }
}
```

Если вспомогательный метод инициализирует поле, пометьте этот метод с помощью MemberNotNullAttribute, чтобы компилятор мог засчитывать обращения к нему.

Проверка параметра проекта

Новые проекты C# позволяют использовать ссылочные типы, допускающие значение NULL, по умолчанию, поэтому большинство кодов, которые вы пишете или читаете, уже включает функцию. Как правило, вам не нужно ничего настраивать. Если вы хотите проверить, включена ли в проекте эта функция, или вам нужно изменить этот параметр, найдите элемент `<Nullable>` в разделе `.csproj`:

XML

```
<PropertyGroup>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

Распространенные поддерживаемые значения: `enable` (по умолчанию для новых проектов) и `disable`. Если элемент отсутствует, проект использует значение по умолчанию, заданное SDK и целевой платформой.

Если необходимо включить значение NULL только для части файла с `#nullable` директивами или использовать частичные `warnings` и `annotations` режимы при переносе существующей базы кода, см. стратегии миграции, допускающие значение NULL.

Куда перейти дальше

Если предупреждение не соответствует ни одному из этих шаблонов, в справочной статье по устранению предупреждений, допускающих значение NULL , перечислены методы для каждого предупреждения CS86xxx, который компилятор выдает.

Сведения о планировании миграции, которая постепенно включает типы ссылок, допускающие значение NULL, в существующей базе кода, см. в стратегиях миграции, допускающих значение NULL.

Связанный контент

- Ссылочные типы, допускающие значение NULL
- Операторы NULL
- Стратегии миграции, допускающие значение NULL
- Атрибуты статического анализа, допускающие значение NULL
- Устранение предупреждений о допустимости значений NULL (справочник по компилятору)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.06.2026

Строки C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Исходя из Java или C++?** C# `string` — это неизменяемый ссылочный тип, основанный на String. UTF-16 — это кодировка в памяти, аналогичная `String` Java. В отличие от C/C++, строки не завершаются null и не распадаются на указатели.

Строка — это последовательность символов. В C# `string` — это ключевое слово языка для System.String типа. Каждый строковый литерал, который вы пишете, создает экземпляр `System.String`.

`string` и `String`

Ключевое `string` слово и `String` имя типа ссылаются на тот же тип. Они компилируются на идентичный промежуточный язык (IL).

C#

```
// The 'string' keyword is an alias for System.String. The two are identical.
string a = "hello";
String b = "hello";

Console.WriteLine(a == b);                // True
Console.WriteLine(typeof(string) == typeof(String)); // True
```

Предпочитайте ключевое `string` слово в собственном коде. Он согласуется с другими встроенными ключевыми словами типа (`int`, `bool`,), `double` и он работает без `using System;` директивы.

Строки неизменяемы

Неизменяемое означает, что значение невозможно изменить после его создания. После создания `string` не удается изменить его символы. Такие методы, как `ToUpperInvariant`,

`Replace` `Substring` и `Trim` возвращают *новую* строку, содержащую измененное значение. Исходный экземпляр остается неизменным.

C#

```
string greeting = "hello";

// ToUpper returns a *new* string. The original is unchanged.
string shouted = greeting.ToUpperInvariant();

Console.WriteLine(greeting);  // hello
Console.WriteLine(shouted);   // HELLO
```

Так как строки неизменяемы, их можно безопасно использовать в методах и потоках. Эта неизменяемость объясняет, почему `string` тип ведет себя как тип значения в повседневном использовании, даже если это ссылочный тип.

При формировании строки из многих небольших фрагментов в цикле используйте StringBuilder, чтобы добавлять на месте и в итоге получить одну строку:

C#

```
// For many sequential edits, StringBuilder avoids allocating a new string each time.
var builder = new StringBuilder();
for (int i = 1; i <= 3; i++)
{
    builder.Append("item ").Append(i).Append(';');
}
string result = builder.ToString();
Console.WriteLine(result);    // item 1;item 2;item 3;
```

Строковые литералы

C# предлагает четыре литеральные формы. Каждая форма подходит для разного содержимого. В качестве краткого руководства:

- Используйте **обычные литералы** для короткого простого текста с максимум несколькими экранирующими последовательностями.
- Используйте литералы **verbatim** если обратные косые скобки доминируют в содержимом, например Windows пути или шаблоны регулярных выражений.
- Используйте **необработанные строковые литералы** для многострочного или структурированного текста, например встроенных блоков JSON, SQL, XML или форматированных сообщений.

- `$` Добавьте префикс к любому из вышеупомянутых строковых литералов, чтобы получить **интерполируемую строку**, когда потребуется встроить значения.
- `u8` Добавьте суффикс в литерал для создания последовательности байтов UTF-8 (aReadOnlySpan<T>) `byte`для байтовых API. Подробные сведения см. в разделе строковые литералы UTF-8 справочника по языку.

Обычные литералы и экранирующие последовательности

Обычный строковый литерал заключён в двойные кавычки. Обратная косая черта запускает escape-последовательность:

C#

```
// Common escape sequences inside a regular string literal.
string tabbed   = "name:\tAda";          // \t  tab
string twoLines = "line 1\nline 2";      // \n  newline
string quoted   = "She said \"hi\".";    // \"  literal quote
string path     = "C:\\src\\app";        // \\  literal backslash

Console.WriteLine(tabbed);
Console.WriteLine(twoLines);
Console.WriteLine(quoted);
Console.WriteLine(path);
```

Распространённые экранирующие последовательности включают `\n` (новая строка), `\t` (табуляция), `\"` (литеральная кавычка), `\\` (литеральная обратная косая черта), `\0` (нулевой символ) и экранирующие последовательности Unicode (`\uXXXX`, `\UXXXXXXXX`).

Начиная с C# 13, `\e` представляет символ элемента управления **ESC** (U+001B). Это начальный байт для escape-последовательностей терминала ANSI:

C#

```
// Beginning in C# 13, \e represents the ESC control character (U+001B).
// It's used to start ANSI terminal escape sequences.
string esc = "\e[31mError\e[0m: file missing";

Console.WriteLine(esc);                  // ESC[31mError ESC[0m: file missing
Console.WriteLine((int)'\e');          // 27
```

Используйте обычный литерал, если текст короткий и содержит только несколько escape-последовательностей. Когда экранирующих последовательностей становится больше, чем видимых символов, переключитесь на дословный или сырой литерал.

Дословные литералы

Дословный литерал имеет префикс `@`. Обратные слэши интерпретируются буквально, что полезно для путей Windows и шаблонов регулярных выражений:

C#

```
// A verbatim string literal (@) treats backslashes literally.
// Useful for Windows paths and regular expressions.
string winPath = @"C:\src\app\readme.md";
string pattern = @"\d{3}-\d{4}";

Console.WriteLine(winPath);
Console.WriteLine(pattern);
```

Чтобы внедрить литеральные кавычки в строку подробных данных, дважды выполните следующие действия: `@"She said ""hi""."` Дословные строки также могут занимать несколько физических строк.

Подробные литералы являются правильным выбором, когда обратные косые скобки являются частью содержимого, но у вас нет много внедренных кавычки. Для многострочного текста или содержимого с кавычками необработанные строковые литералы обычно более понятны.

Необработанные строковые литералы

Для любого литерала, содержащего кавычки, обратные очки или несколько строк, предпочитают **необработанные строковые литералы**. Они полностью устраняют шум экранирования, что делает их наиболее подходящим для встроенных шаблонов JSON, SQL, XML, regex и отформатированных текстовых блоков, где источник должен выглядеть как выходные данные:

C#

```
// Raw string literals use three or more quotes and need no escaping.
// The source looks like the output, which is ideal for inline JSON, SQL, XML, and the
like.
string json = """
    {
        "name": "Ada",
        "roles": ["admin", "editor"]
    }
    """;

string sql = """
```

```
    SELECT Id, Name
    FROM   Users
    WHERE  Name = 'O''Brien'
    """;

Console.WriteLine(json);
Console.WriteLine(sql);
```

Сырые строковые литералы практически полностью устраняют необходимость в экранирующих последовательностях и поддерживают любое необходимое форматирование и использование кавычек. Полные правила см. в разделе Необработанные строковые литералы.

Интерполированные строки

Префикс `$` преобразует литерал в *интерполированную строку*. Выражения в `{}` отверстиях вычисляются и вставляются их результаты, и вы можете применить стандартные описатели формата и выравнивание внутри отверстий. Интерполяция также сочетается с другими литеральными формами — используйте `$@"..."` для интерполяции дословного литерала или `$"""..."""` для интерполяции необработанного строкового литерала для вывода с расширенным форматированием:

C#

```
// The $ prefix evaluates expressions inside { } and inserts their values.
string name = "Ada";
int score = 92;

string greeting = $"Hello, {name}! Your score is {score}.";
// Format specifiers and alignment work inside the holes.
string formatted = $"pi = {Math.PI:F3}, padded = |{name,10}|";

// Combine $ and """ for richly formatted multiline output.
string report = $"""
    Report for {name}
    -----------------
    Score : {score}
    Grade : {(score >= 90 ? "A" : "B")}
    """;

Console.WriteLine(greeting);
Console.WriteLine(formatted);
Console.WriteLine(report);
```

Интерполяция — это рекомендуемый способ создания строк из значений в повседневном коде.

Индексирование и `char`

A `string` — это последовательность *единиц кода* UTF-16. Индексатор возвращает один System.Char, представляющий одну единицу кода UTF-16, а не полную кодовую точку Юникода. `Length` возвращает количество единиц кода.

C#

```
// A string is a sequence of UTF-16 code units. s[i] returns one char.
string word = "café";

Console.WriteLine(word.Length);          // 4
Console.WriteLine(word[0]);              // c

foreach (char c in word)
{
    Console.Write($"{c} ");              // c a f é
}
Console.WriteLine();
```

Для текста, который может содержать эмодзи или символы за пределами базовой многоязычной плоскости, выполняйте итерацию по *rune* с использованием Rune или по графемным кластерам с использованием StringInfo. Простая `char` итерация подходит для большинства западноевропейских текстов и контента, в котором преобладают символы ASCII.

Равенство строк

При проверке на равенство с помощью `string` сравниваются последовательности символов, а не ссылки:

C#

```
// String equality compares character sequences, not references.
string left  = "hello";
string right = string.Concat("hel", "lo");

Console.WriteLine(left == right);                                  // True
Console.WriteLine(left.Equals(right, StringComparison.Ordinal));   // True
```

Для сравнений, которые должны учитывать языковые параметры или регистр, передайте явное значение StringComparison. Используется `StringComparison.Ordinal` для значений протокола, идентификаторов и других нелингвистических текстов.

Общие строковые операции

Используйте следующую таблицу как краткое руководство по повседневным операциям со строками в C#:

Развернуть таблицу

Категория	Что он охватывает
Search	Находить символы или подстроки, проверять префиксы и суффиксы
Разбивка	Разбить строку на подстроки по разделителям
Объединить	Объединение строк — `+`интерполяция, `Concat` `Join`
Modify	Создание преобразованной копии — `Replace`, `Trim` `Substring`
Сравнить	Проверка равенства и упорядочения с правом `StringComparison`

Полная поверхность API — каждая перегрузка, каждый метод — задокументирована в справочнике String .

См. также

- Необработанные строковые литералы
- nameof оператор
- String
- StringBuilder
- Char
- ReadOnlySpan<T>

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 27.05.2026

Необработанные строковые литералы

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы. Полный грамматический код см. в справочнике по языку.
>
> **Вы пришли из другого языка?** Необработанные строковые литералы в C# выполняют ту же функцию, что и строки `r"..."` в Python и Rust, текстовые блоки в Java (`"""..."""`), а также строки шаблонов в обратных кавычках в JavaScript, TypeScript и Go. Синтаксис C# ближе всего к текстовым блокам Java с дополнительными правилами для разделителей переменной длины и интерполяции.

Необработанный строковый литерал разделен тремя или более двойными кавычками. Внутри ограничителей каждый символ воспринимается буквально. Кавычки и обратные слеши не нужно экранировать, а символы новой строки сохраняются в том виде, в котором записаны. Используйте необработанные строки для любой строки, содержащей кавычки, обратные очки или несколько строк: JSON, XML, SQL, регулярные выражения, пути к файлам и примеры кода.

> **⚠ Предупреждение**
>
> Необработанный строковый литерал упрощает чтение SQL, но это не делает SQL более безопасным. Никогда не сцепляйте или интерполяйте предоставленные пользователем значения в команду SQL. Такая практика делает ваше приложение уязвимым для SQL-инъекций. Используйте параметризованные команды вместо этого: DbCommand.CreateParameter с DbParameterCollection.Add или вспомогательными средствами более высокого уровня в Entity Framework Core и Dapper. Эта же осторожность применяется к другим форматам, подверженным внедрению, таким как команды оболочки, фильтры LDAP и HTML.

Литерал, содержащий кавычки и обратные косые черты

Обычный литерал нуждается в обходе `"` и `\`. Дословный литерал по-прежнему должен `""` включать цитату. Сырой литерал не нуждается ни в том, ни в другом:

C#

```
// Same JSON value, three ways:
string regular  = "{ \"name\": \"Ada\", \"path\": \"C:\\\\src\" }";
string verbatim = @"{ ""name"": ""Ada"", ""path"": ""C:\\src"" }";
string raw      = """{ "name": "Ada", "path": "C:\\src" }""";

Console.WriteLine(regular  == raw);   // True
Console.WriteLine(verbatim == raw);   // True
```

Каждая форма создает ту же строку, но необработанная версия считывается точно так же, как в формате JSON, который он представляет.

Однострочные необработанные строки

Открывающий и закрывающий разделители состоят как минимум из трёх двойных кавычек, при этом закрывающий разделитель должен содержать такое же количество кавычек, что и открывающий разделитель. Содержимое находится между ними на одной строке. Кавычки и обратные косые скобки внутри содержимого являются литеральными:

C#

```
// A raw string literal starts and ends with at least three quotes.
// Inside, " and \ are literal — no escaping required.
string message = """She said "hi" and left.""";
string regex   = """\d{3}-\d{4}""";

Console.WriteLine(message);   // She said "hi" and left.
Console.WriteLine(regex);     // \d{3}-\d{4}
```

Однострочная сырая строка не может быть пустой между ограничителями. Он может заканчиваться двойной кавычкой, но не может начинаться с неё. Компилятор рассматривает двойную кавычку в начале как дополнительный открывающий символ-разделитель. Если содержимое должно начинаться с кавычки, вместо этого используйте многострочный необработанный строковый литерал, который размещает содержимое на отдельной строке, где начальная кавычка трактуется однозначно.

Многостроковые необработанные строки

Для многостроного содержимого открывающий разделитель заканчивает строку, а закрывающий разделитель запускается самостоятельно. Как и в случае с необработанными строками, разделитель имеет три или более двойных кавычки, а закрывающий разделитель должен использовать то же количество кавычек, что и открывающий. Обычно используют три кавычки, но можно использовать четыре, пять или больше, если само содержимое содержит последовательность символов `"""`. Все между двумя разделителями — это значение строки, точно как написано:

C#

```
// The opening """ and closing """ each sit on their own line.
// The content between them is the value, exactly as written.
string sql = """
    SELECT id, name
    FROM customers
    WHERE active = 1
    """;

Console.WriteLine(sql);
```

Новая строка сразу после открытия `"""` и новой линии непосредственно перед закрытием `"""` не является частью значения. Они — пробелы разделителей. Аналогичным образом компилятор удаляет все пробелы слева от закрывающего тега `"""` в каждой строке содержимого, так что вы можете добавить литералу отступ, чтобы он соответствовал окружающему блоку кода, и при этом этот отступ не попадёт в строку. В следующем разделе подробно рассматривается это правило.

Если само содержимое содержит последовательность `"""`, используйте в качестве разделителей четыре или более кавычек. Число разделителей должно просто превышать самую длинную последовательность кавычек в содержимом текста. См. полные правила в разделе Необработанные строковые литералы (справочник по языку).

Отступ: закрывающий разделитель задает поле

Столбец, в котором расположен закрывающий тег `"""`, определяет левый отступ. Компилятор удаляет пробелы до этого столбца из каждой строки содержимого. Это правило позволяет отступить литерал в соответствии с окружающим кодом, не загрязняя значение:

C#

```
// The column of the closing """ sets a left margin.
// Whitespace up to that column is stripped from every content line.
string xml = """
        <order id="42">
            <item>book</item>
        </order>
        """;

// First content line begins at column 0 of the value:
Console.WriteLine(xml);
// <order id="42">
//     <item>book</item>
```

Если строка содержимого имеет меньше ведущих символов пробелов, чем столбец закрывающего разделителя, компилятор сообщает об ошибке. Все строки содержимого должны иметь отступ не меньше, чем у закрывающего тега `"""`.

Необработанные интерполированные строки

`$` Добавьте префикс в необработанную строку, чтобы включить интерполяцию. Выражения в отверстиях вычисляются, и их результаты вставляются в `{}` значение:

C#

```
// A single $ before """ enables interpolation: single { and } mark a hole.
// Inside a single-$ raw string, literal braces aren't allowed — use $$ when
// the content also contains literal { or }.
string name = "Ada";
int    score = 95;

string report = $"""
    Player:  {name}
    Score:   {score}
    Updated: {DateTime.UtcNow:yyyy-MM-dd}
    """;

Console.WriteLine(report);
```

Если интерполированное содержимое также должно содержать литеральные символы `{` или `}`, см. статью Необработанные строковые литералы (справочник по языку).

Когда выбирать, какой литерал

Используйте **необработанный строковый литерал** всякий раз, когда содержимое содержит кавычки, обратные очки или несколько строк. В результате текст стал короче, его легче

читать, вставлять и копировать, и он не содержит ошибок, связанных с escape-последовательностями.

Используйте **обычный строковый литерал** для коротких однострочных значений без кавычек или обратных косых кавычек, таких как имена, сообщения, заполнители формата.

Используйте **дословный строковый литерал** (`@"..."`) только при работе с существующим кодом, использующим их. Для нового кода сырые строковые литералы покрывают все случаи использования, которые покрывают дословные строковые литералы, при этом обеспечивая более чистый синтаксис для встроенных кавычек.

См. также

- Общие сведения о строках
- nameof оператор
- Необработанные строковые литералы (справочник по языку)
- String

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.05.2026

Интерполяция строк в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** Интерполяция строк в C# работает так же, как литералы шаблонов в JavaScript (`` `${x}` ``) или f-strings в Python (`f"{x}"`). Выражение внутри `{}` может быть любым допустимым выражением C#, и вы можете добавлять описатели формата и выравнивания, не оставляя строку.

Интерполяция строк позволяет внедрять выражения непосредственно в строковый литерал, префиксируя литерал с помощью `$`:

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is {CalculateHypotenuse(a, b)}");
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);
// => Area of the right triangle with legs of 3 and 4 is 6
// => Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Каждое `{ }` из них является *выражением интерполяции*. C# вычисляет выражение, преобразует результат в строку путем вызова метода `ToString()` и заменяет текст результатом. Интерполяция строки для выражения `null` представляет собой пустую строку. В большинстве случаев преобразование по умолчанию создает нужные выходные данные, и вам ничего больше не нужно делать.

Интерполированные строки являются более доступной для чтения альтернативой String.Format, и они поддерживают полный составной набор функций форматирования . Все, что можно делать с помощью классической строки позиционного формата — будь то спецификаторы формата, выравнивание, форматирование с учётом региональных параметров или константные строки, — можно делать и с интерполированной строкой. В

остальной части этой статьи рассматриваются эти параметры. Обращайтесь к ним только тогда, когда вам нужен более точный контроль над результатом; в противном случае достаточно обычной формы `$"{expression}"`.

Описание синтаксиса и базовых типов обработчиков см. в справочном материале Интерполированные строки. Сведения о темах, связанных с производительностью, таких как `Span<char>` интерполяция строк и пользовательские обработчики интерполированных строк, см. в разделе Операции со строками.

Применить строку формата

Чтобы управлять форматированием результата выражения, следуйте выражению с двоеточием и строкой стандартного или настраиваемого формата:

C#

```
{<expression>:<formatString>}
```

В следующем примере форматируются значение DateTime и значение Double:

C#

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} L. Euler introduced the letter e to
denote {Math.E:F5}.");
// => On Sunday, November 25, 1731 L. Euler introduced the letter e to denote 2.71828.
```

Задайте ширину поля и выравнивание

Чтобы создать выровненные выходные данные, следуйте выражению с запятой и минимальной шириной поля. Положительные ширины выровняют значение вправо, а отрицательные ширины выравнивают его по левому краю:

C#

```
{<expression>,<width>}
```

C#

```
var titles = new Dictionary<string, string>()
{
    ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
    ["London, Jack"] = "Call of the Wild, The",
```

```
    ["Shakespeare, William"] = "Tempest, The"
};

Console.WriteLine("Author and Title List");
Console.WriteLine();
Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
foreach (var title in titles)
{
    Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
}
// => Author and Title List
// =>
// => |Author                   |                         Title|
// => |Doyle, Arthur Conan      |Hound of the Baskervilles, The|
// => |London, Jack             |         Call of the Wild, The|
// => |Shakespeare, William     |                 Tempest, The|
```

Когда нужны и выравнивание, и строка формата, сначала указывайте выравнивание:

C#

```
{<expression>,<width>:<formatString>}
```

C#

```
const int NameAlignment = -9;
const int ValueAlignment = 7;
double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
// => Three classical Pythagorean means of 3 and 4:
// => |Arithmetic|  3.500|
// => |Geometric|  3.464|
// => |Harmonic |  3.429|
```

Если форматируемое значение больше запрошенной ширины, C# игнорирует ширину и выдает полное значение.

Escape-фигурные скобки и использование escape-последовательностей

Интерполированные строки поддерживают те же экранирующие последовательности, что и обычные строковые литералы. Чтобы включить в результат буквальный символ `{` или `}`, удвойте его (`{{` или `}}`).

Для путей и других строк, содержащих обратные косые черты, предпочтите интерполируемый необработанный строковый литерал (`$"""..."""`) вместо более старой дословной формы (`$@"..."`). Необработанные строковые литералы не обрабатывают escape-последовательности, поэтому обратные косые строки отображаются as-is:

C#

```
int[] xs = [1, 2, 7, 9];
int[] ys = [7, 9, 12];
Console.WriteLine($"Find the intersection of the {{{string.Join(", ", xs)}}} and
{{{string.Join(", ", ys)}}} sets.");
// => Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var rawInterpolated = $"""C:\Users\{userName}\Documents""";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(rawInterpolated);
// => C:\Users\Jane\Documents
// => C:\Users\Jane\Documents
```

Использование условного выражения

Двоеточие имеет особое значение внутри выражения интерполяции, поэтому заключите условное выражение в скобки:

C#

```
var rand = new Random(42);
for (int i = 0; i < 3; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");
}
```

Расположить выражение на нескольких строках

Для удобства чтения разбивайте длинные выражения интерполяции на несколько строк. Выражения интерполяции с несколькими строками доступны с версии C# 11. В следующем примере добавляются новые линии между `{` символами и `}` символами для разделения интерполированных выражений из литерального текста:

C#

```
int[] numbers = [3, 1, 4, 1, 5, 9, 2, 6];
Console.WriteLine($"Total: {
        numbers.Sum()
        }, average: {numbers.Average():F2}.");
// => Total: 31, average: 3.88.
```

Создание строковых констант

Вы можете создавать интерполированные строки констант, когда каждое интерполированное выражение является константным значением. Это делает его пригодным для аргументов атрибутов, `switch` шаблонов и других контекстов, требующих констант времени компиляции:

C#

```
const string Audience = "world";
const string Greeting = $"Hello, {Audience}!";
Console.WriteLine(Greeting);
// => Hello, world!
```

Форматирование с использованием определенной культуры

По умолчанию интерполированная строка форматирует значения с помощью CultureInfo.CurrentCulture, что влияет на представление даты, чисел и денежных значений. Для детерминированного вывода передайте явно заданную культуру в String.Create(IFormatProvider, DefaultInterpolatedStringHandler):

C#

```
CultureInfo[] cultures =
[
    CultureInfo.GetCultureInfo("en-US"),
    CultureInfo.GetCultureInfo("en-GB"),
    CultureInfo.GetCultureInfo("nl-NL"),
    CultureInfo.InvariantCulture
];
var date = new DateTime(2026, 5, 21, 12, 35, 31);
var number = 31_415_926.536;
foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
```

```
}
// Output is similar to:
// => en-US       5/21/2026 12:35:31 PM      31,415,926.536
// => en-GB         21/05/2026 12:35:31      31,415,926.536
// => nl-NL         21-05-2026 12:35:31      31.415.926,536
// =>               05/21/2026 12:35:31      31,415,926.536
```

Для инвариантных выходных данных (журналы, форматы файлов, данные, доступные для чтения компьютером), передайте CultureInfo.InvariantCulture:

C#

```
var timestamp = new DateTime(2026, 5, 21, 15, 46, 24);
string message = string.Create(CultureInfo.InvariantCulture, $"Date and time in
invariant culture: {timestamp}");
Console.WriteLine(message);
// => Date and time in invariant culture: 05/21/2026 15:46:24
```

См. также

- Интерполированный строковый маркер
- Строковые операции: сопоставление шаблонов, производительность и поиск на основе диапазона
- Составное форматирование
- Типы форматирования в .NET
- String.Format
- System.FormattableString

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.06.2026

Оператор `nameof`

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы. Полное справочное описание оператора см. в разделе nameof справочника по языку.
>
> **Вы пришли из другого языка?** Другие языки имеют аналогичные функции. рефлексивный `Class.getSimpleName()` в Java, `Function.name` и `Object.keys` в JavaScript, `__name__` и `vars()` в Python и `#function`/`#keyPath` в Swift. В отличие от большинства из них, `nameof` в C# является исключительно компиляционной конструкцией. Не использует рефлексию, ничего не выделяет во время выполнения и создает константу `string`, встроенную в сборку.

Оператор `nameof` возвращает текстовый идентификатор символа, например переменную, параметр, тип, член или пространство имен в виде константы во время `string` компиляции. Везде, где иначе пришлось бы жёстко задавать идентификатор строкой, используйте `nameof`: компилятор проверяет, что символ существует, а рефакторинг переименования автоматически обновляет результат.

Что `nameof` возвращается

`nameof` принимает значение *последнего* идентификатора в своём операнде. Это происходит на этапе компиляции и не влечёт накладных расходов во время выполнения.

C#

```
// nameof produces the textual identifier of a symbol at compile time.
Console.WriteLine(nameof(Customer));        // Customer
Console.WriteLine(nameof(Customer.Name));   // Name

var customer = new Customer("Ada");
Console.WriteLine(nameof(customer));        // customer
Console.WriteLine(nameof(customer.Name));   // Name
```

Операнды также могут быть *квалифицированным выражением*, который использует оператор dot для перехода из содержащей области в элемент, например `customer.Name`,

`System.Console` или `List<int>.Enumerator`. В этом случае фиксируется только последний идентификатор: `nameof(customer.Name)` возвращается `"Name"`, а не `"customer.Name"`.

Проверка аргументов

Классический вариант использования заключается в указании имени параметра в выбрасываемом исключении. Передайте `nameof(parameter)` вместо строкового литерала `"parameter"`, чтобы будущее переименование не сделало сообщение недостоверным:

C#

```
try
{
    Greet("");
}
catch (ArgumentException ex)
{
    // The exception's ParamName is the literal "name", produced by nameof at compile
time.
    Console.WriteLine($"{ex.ParamName}: {ex.Message}");
}

static void Greet(string name)
{
    if (string.IsNullOrWhiteSpace(name))
    {
        throw new ArgumentException("Name must be non-empty.", nameof(name));
    }
    Console.WriteLine($"Hello, {name}!");
}
```

В частности для проверок на null предпочтительно использовать вспомогательные методы для создания исключений. Эти вспомогательные средства, такие как ThrowIfNull, автоматически захватывают имя аргумента с помощью CallerArgumentExpressionAttribute, поэтому отдельный `nameof` не требуется:

C#

```
// ArgumentNullException.ThrowIfNull captures the argument's name automatically
// through [CallerArgumentExpression], so a separate nameof isn't required for
// the null check. Use nameof for cases the helpers don't cover.
Customer? maybeCustomer = null;

try
{
    Save(maybeCustomer);
}
catch (ArgumentNullException ex)
```

```
{
    Console.WriteLine(ex.ParamName);   // customer
}

static void Save(Customer? customer)
{
    ArgumentNullException.ThrowIfNull(customer);
    // ...
}
```

Используйте `nameof` в случаях, когда вспомогательные средства не охватывают: ArgumentExceptionArgumentOutOfRangeException(при проверке чего-либо другого, кроме одного аргумента), а также других сообщений защиты.

Уведомления об изменении свойств

Типы, реализующие INotifyPropertyChanged, вызывают событие, полезная нагрузка которого включает имя измененного свойства. Жёсткое задание имени в виде строки создаёт скрытую ошибку, если свойство переименуют, а строку — нет. Используйте `nameof` вместо этого:

C#

```
public sealed class Person : INotifyPropertyChanged
{
    private string _name = "";

    public string Name
    {
        get => _name;
        set
        {
            if (_name == value) return;
            _name = value;
            // nameof keeps the property name and the change notification in sync.
            // Renaming the property automatically updates this argument.
            OnPropertyChanged(nameof(Name));
        }
    }

    public event PropertyChangedEventHandler? PropertyChanged;

    private void OnPropertyChanged([CallerMemberName] string? propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Сеттер вызывает `OnPropertyChanged(nameof(Name))`, поэтому имя свойства и уведомление об изменении остаются синхронизированными. Запустите пример, чтобы увидеть, как

срабатывают события:

C#

```
var person = new Person();
person.PropertyChanged += (_, e) => Console.WriteLine($"changed: {e.PropertyName}");

person.Name = "Ada";    // changed: Name
person.Name = "Grace";  // changed: Name
```

`nameof` в аргументах атрибутов

`nameof` допустим в аргументах атрибутов. Компилятор разрешает идентификаторы в окружающей области, включая параметры метода, целевые объекты атрибута. Это идиоматический способ ссылаться на параметр из атрибута, NotNullIfNotNullAttributeнапример:

C#

```
// nameof works inside attribute arguments. The compiler resolves the
// identifier even when the attribute targets a method or its parameters.
Console.WriteLine(NormalizeOrNull("  hi  ") ?? "<null>");   // hi
Console.WriteLine(NormalizeOrNull(null) ?? "<null>");        // <null>

[return: NotNullIfNotNull(nameof(input))]
static string? NormalizeOrNull(string? input) => input?.Trim();
```

Если параметр переименовывается, аргумент `nameof` обновляется при том же рефакторинге — атрибут не может устареть.

Квалифицированные имена

Для любого квалифицированного выражения `nameof` возвращает только *последний* идентификатор:

C#

```
// For a qualified expression, nameof returns only the final identifier.
Console.WriteLine(nameof(System.Collections.Generic.List<int>)); // List
Console.WriteLine(nameof(Customer.Name));                        // Name
```

Если вам нужно полное имя, используйте Type.FullName для экземпляра `Type`. `nameof` предназначен для идентификаторов, а не путей.

Используйте `nameof` вместо строковых идентификаторов

В любом месте, где вы ссылаетесь на метод, свойство, параметр, тип или пространство имен по имени в коде, используйте `nameof` вместо строкового литерала. По сравнению с жестко закодированной строкой:

- Компилятор проверяет, существует ли символ. Опечатка приводит к ошибке сборки, а не к незаметному багу во время выполнения.
- Рефакторинг переименования автоматически обновляет результат. Жестко закодированные строки не синхронизируются.
- Результат представляет собой константу на этапе компиляции, поэтому во время выполнения дополнительные затраты отсутствуют.

Эта рекомендация применяется к сообщениям журнала, аргументам исключений, аргументам атрибутов, уведомлениям об изменении свойств и константам ключей сериализации, привязанным к имени члена.

См. также

- Общие сведения о строках
- Необработанные строковые литералы
- nameof оператор (справочник по языку)
- ThrowIfNull
- INotifyPropertyChanged

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 27.05.2026

Объединение строк в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** Объединение C# с `+` параллелями Java и JavaScript. C# добавляет интерполяцию строк (`$"{x}"`), аналогичную литералам шаблонов JavaScript и Python f-strings, как предпочтительный способ построения строк из переменных. Для построения строк в цикле C# предлагает StringBuilder, подобно `StringBuilder` в Java.

Объединение добавляет одну строку в конец другой, чтобы создать новую строку. C# предоставляет несколько способов конкатенации строк, и оптимальный выбор зависит от того, объединяете ли вы фиксированный набор значений, коллекцию или собираете строку по частям в цикле.

Сцепляйте строковые литералы

При объединении строковых литералов или констант с помощью `+` компилятор объединяет их на этапе компиляции. Разделение длинного литерала на несколько строк повышает читабельность исходного кода без каких-либо затрат во время выполнения:

C#

```
// The compiler joins adjacent string literals at compile time,
// so splitting a long literal across lines has no run-time cost.
string message =
    "This is the first sentence of a longer message. " +
    "This is the second sentence. " +
    "This is the third and final sentence.";

Console.WriteLine(message);
// => This is the first sentence of a longer message. This is the second sentence.
This is the third and final sentence.
```

Использовать операторы `+` и `+=`

Чтобы объединить строковые переменные, используйте `+` оператор для создания новой строки или `+=` добавления к существующей. Оператор `+` интуитивно понятный, и компилятор копирует строковое содержимое только один раз, даже если вы привязываете несколько операторов в одном выражении:

C#

```
string name = "Alex";
string day = "Monday";

// Use + to build a string from variables and literals.
string greeting = "Hello " + name + ". Today is " + day + ".";
Console.WriteLine(greeting);
// => Hello Alex. Today is Monday.

// Use += to append to an existing string.
greeting += " How are you today?";
Console.WriteLine(greeting);
// => Hello Alex. Today is Monday. How are you today?
```

> ⓘ **Примечание.**
>
> В объединениях строк C# обрабатывает `null` строку так же, как пустую строку, поэтому объединение `null` не добавляет ничего в результат.

Использование интерполяции строк

Чтобы встраивать вычисляемые выражения в строку, предпочтительнее использовать интерполяцию строк вместо позиционных подстановок, таких как токены `{0}` и `{1}`, которые String.Format наследует из форматирования в стиле C. Интерполяция помещает каждое выражение в строку, в которой отображается его значение, поэтому результируемая строка остается удобочитаемой, и аргумент не может быть неправильно указан:

C#

```
string name = "Alex";
string day = "Monday";

// String interpolation reads better than a chain of + operators.
string greeting = $"Hello {name}. Today is {day}.";
Console.WriteLine(greeting);
// => Hello Alex. Today is Monday.
```

Когда каждое интерполированное выражение является константной строкой, можно назначить интерполированный результат строке `const` .

Присоединение к коллекции строк

Чтобы объединить элементы коллекции, используйте String.Concat для объединения их без разделителя или String.Join размещения разделителя между каждым элементом:

C#

```
string[] words = ["The", "quick", "brown", "fox"];

// Concat joins the sequence with no separator.
string runTogether = string.Concat(words);
Console.WriteLine(runTogether);
// => Thequickbrownfox

// Join places a separator between each element.
string sentence = string.Join(' ', words);
Console.WriteLine(sentence);
// => The quick brown fox
```

`string.Join` — это правильное средство, когда требуется вывод с разделителями, например значения с разделителями-запятыми или слова, разделенные пробелами.

Создать строку в цикле

Каждая операция `+` или `+=` создает новую строку, поскольку строки неизменяемы. Когда вы добавляете много фрагментов в цикле, затраты на выделение памяти накапливаются. Класс StringBuilder создает результат в одном буфере.

C#

```
// StringBuilder builds a string in place, which suits loops
// that append many pieces.
var builder = new StringBuilder();
for (int i = 1; i <= 3; i++)
{
    builder.AppendLine($"Line {i}");
}

Console.Write(builder.ToString());
// => Line 1
// => Line 2
// => Line 3
```

Используйте `StringBuilder`, если число элементов велико или неизвестно на этапе компиляции. Для небольшого фиксированного набора значений оператор `+` и интерполяция строк более понятны. Сведения о том, в каких случаях каждый подход работает лучше всего, см. в разделе Типы string и StringBuilder.

См. также

- Интерполяция строк в C#
- String.Concat
- String.Join
- StringBuilder

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.06.2026

Изменение содержимого строки в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** Как и Java и JavaScript, строки C# *неизменяемы*: такие методы, как `Replace` и `Trim` возврат новой строки вместо изменения исходного кода. Шаблоны здесь аналогичны `String` методам в этих языках.

C# `string` является *неизменяемым*, что означает, что его содержимое никогда не изменяется после его создания. Каждый метод, который, как представляется, изменяет строку, фактически возвращает новое `string` с изменениями, оставляя исходный нетронутым. Примеры, приведенные в этой статье, сохраняют каждый результат в новой переменной, чтобы вы могли видеть как источник, так и измененное значение.

Выберите способ, соответствующий вашему сценарию: замените известный текст, обрезать пробелы, удалите диапазон символов, замените текст, соответствующий шаблону, или измените отдельные символы.

Замена известного текста

Метод String.Replace заменяет каждое вхождение одной строки другой и возвращает результат в виде новой строки:

C#

```
string source = "The mountains are behind the clouds today.";

// Replace returns a new string; the original is unchanged.
string updated = source.Replace("mountains", "peaks");

Console.WriteLine(source);
// => The mountains are behind the clouds today.
Console.WriteLine(updated);
// => The peaks are behind the clouds today.
```

Исходная строка не изменяется, которая демонстрирует неизменяемость: `Replace` создает новую строку с подстановкой.

`Replace` также имеет перегрузку, которая переключает одиночные символы. Следующий пример заменяет каждое пространство символом подчеркивания:

C#

```
string source = "The mountains are behind the clouds today.";

// Replace every occurrence of one character with another.
string updated = source.Replace(' ', '_');

Console.WriteLine(updated);
// => The_mountains_are_behind_the_clouds_today.
```

Обе перегрузки заменяют *каждое* совпадение в строке, а не только первое. Независимо от того, передаёте ли вы один символ или строку, `Replace` заменяет все вхождения за один вызов.

Обрезать пробелы

Используйте String.Trim, String.TrimStart и String.TrimEnd, чтобы удалить начальные или конечные пробелы. Каждый метод возвращает новую строку:

C#

```
string source = "    I'm wider than I need to be.      ";

// Each method returns a new string with whitespace removed.
Console.WriteLine($"<{source.Trim()}>");
// => <I'm wider than I need to be.>
Console.WriteLine($"<{source.TrimStart()}>");
// => <I'm wider than I need to be.      >
Console.WriteLine($"<{source.TrimEnd()}>");
// => <    I'm wider than I need to be.>
```

Удалить фрагмент символов

Метод String.Remove удаляет ряд символов, начиная с индекса. Объедините его с String.IndexOf, чтобы найти текст для удаления:

C#

```
string source = "Many mountains are behind many clouds today.";
string toRemove = "many ";

// Find the text, then remove that span by index and length.
int index = source.IndexOf(toRemove);
string result = index >= 0
    ? source.Remove(index, toRemove.Length)
    : source;

Console.WriteLine(result);
// => Many mountains are behind clouds today.
```

Замена текста, соответствующего шаблону

Если необходимо заменить текст, который следует шаблону, а не точной строке, используйте регулярные выражения. Метод Regex.Replace принимает функцию, вычисляющую каждую замену, чтобы можно было сохранить такие сведения, как исходная заглавная буква. Шаблон `the\s` совпадает с "the", за которым следует пробельный символ, что не позволяет ему совпадать с "there":

C#

```
string source = "The mountains are still there behind the clouds today.";

// Replace "the" or "The" followed by whitespace, preserving the original case.
// The \s in the pattern keeps "there" from matching.
string result = Regex.Replace(
    source,
    """the\s""",
    match => char.IsUpper(match.Value[0]) ? "Many " : "many ",
    RegexOptions.IgnoreCase);

Console.WriteLine(result);
// => Many mountains are still there behind many clouds today.
```

Сведения о поиске по шаблону, а не о замене, см. в разделе Поиск строк в C#. Сведения о синтаксисе регулярного выражения см. в кратком справочнике по языку регулярных выражений.

Изменение отдельных символов

Чтобы изменить символы по позиции, скопируйте строку в Span<T> символы, измените диапазон и создайте новую строку из нее. Следующий пример находит слово "лиса" и заменяет его на "кот":

C#

```
string phrase = "The quick brown fox jumps over the fence.";

// A string is immutable, so copy it into a Span<char> to edit in place.
Span<char> characters = stackalloc char[phrase.Length];
phrase.CopyTo(characters);
int index = phrase.IndexOf("fox");
if (index != -1)
{
    characters[index] = 'c';
    characters[index + 1] = 'a';
    characters[index + 2] = 't';
}

// Build a new string from the modified characters.
string updated = new string(characters);
Console.WriteLine(updated);
// => The quick brown cat jumps over the fence.
```

Для высокопроизводительных сценариев, в которых удаётся избежать промежуточных выделений памяти, среда выполнения предоставляет низкоуровневые API, такие как String.Create. Эти методы являются расширенными; для повседневного кода методы в этой статье являются правильным выбором.

См. также

- Строки поиска в C#
- Регулярные выражения .NET
- String.Replace
- StringBuilder

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.06.2026

Сравнение строк в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** C# `==` в строках сравнивает *значения*, а не ссылки, например Java `equals` или JavaScript `===`. Ключевое различие заключается в том, что C# позволяет выбирать между **порядковым** (двоичным) и **сравнением с учетом языковых и региональных параметров** с помощью значения StringComparison.

Вы сравниваете строки, чтобы ответить на один из двух вопросов: *равны ли эти две строки?* Или *в каком порядке эти строки сортируются?* C# позволяет управлять двумя независимыми факторами при ответе на любой вопрос:

- **Чувствительность к регистру** — считаются ли `"Hello"` и `"hello"` равными.
- **Тип сравнения** — *порядковое* (сравнивается двоичное значение каждого символа) или *с учетом языковых и региональных параметров* (применяются лингвистические правила определенной культуры).

Перечисление StringComparison объединяет эти факторы в одно значение, которое передается в методы сравнения.

Сравнить на равенство

Метод String.Equals и оператор `==` по умолчанию выполняют **сравнение с учетом регистра и по порядковому значению**. Порядковое сравнение проверяет двоичное значение каждого символа, поэтому оно быстро и дает одинаковый результат на каждом компьютере. Вы можете явно указать свои намерения, вызвав перегрузку, которая принимает значение StringComparison:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

// Equals and == both perform a case-sensitive, ordinal comparison.
```

```
Console.WriteLine(root.Equals(root2));
// => False
Console.WriteLine(root == root2);
// => False

// Pass a StringComparison value to state the intent explicitly.
Console.WriteLine(root.Equals(root2, StringComparison.Ordinal));
// => False
```

Чтобы игнорировать регистр при сохранении порядковой семантики, передайте StringComparison.OrdinalIgnoreCase:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

// OrdinalIgnoreCase compares the binary values but ignores case.
bool equalIgnoringCase = string.Equals(root, root2,
StringComparison.OrdinalIgnoreCase);
Console.WriteLine(equalIgnoringCase);
// => True
```

Сравнение порядка сортировки

Чтобы определить порядок сортировки, а не равенство, используйте String.Compare. Он возвращает отрицательное число, ноль или положительное число, указывающее, находится ли первая строка при сортировке перед второй строкой, на той же позиции, что и вторая строка, или после неё:

C#

```
string first = "Avocado";
string second = "Banana";

// Compare returns a negative number, zero, or a positive number
// to indicate sort order. Specify the comparison type explicitly.
int order = string.Compare(first, second, StringComparison.Ordinal);
Console.WriteLine(order < 0
    ? $"'{first}' sorts before '{second}'."
    : $"'{first}' sorts at or after '{second}'.");
// => 'Avocado' sorts before 'Banana'.
```

ⓘ Важно!

`Compare` и `CompareTo` по умолчанию используют *сравнение с учетом языковых и региональных параметров*, а `Equals` и `==` — *порядковое сравнение*. Чтобы избежать сюрпризов, передайте явное StringComparison значение, чтобы код указывает, какое поведение он хочет.

Сравнение с константами с помощью сопоставления с шаблоном `is` или `switch`

Если сравниваемое значение является константой, можно использовать is оператор с константным шаблоном в качестве доступной для чтения альтернативы `==`:

C#

```
string status = "Ready";

// When the right operand is a constant, the is operator and a
// constant pattern read as an alternative to ==.
if (status is "Ready")
{
    Console.WriteLine("The system is ready.");
}
// => The system is ready.
```

Чтобы сравнить строку с несколькими константами, используйте switch выражение. Каждая ветвь проверяет шаблон-константу, а отбрасывающий шаблон (`_`) обрабатывает все значения, которые ему не соответствуют. Следующий пример сопоставляет ключевое слово направления с инструкцией по поездке:

C#

```
foreach (string heading in new[] { "North", "South", "East", "West", "NE" })
{
    // A switch expression matches each constant pattern in turn and
    // returns the first match. The discard (_) handles every other value.
    string instruction = heading switch
    {
        "North" => "Travel due North for 10 km.",
        "South" => "Travel due South for 10 km.",
        "East" => "Travel due East for 10 km.",
        "West" => "Travel due West for 10 km.",
        _ => $"Unknown heading: {heading}.",
    };
    Console.WriteLine(instruction);
}
// => Travel due North for 10 km.
// => Travel due South for 10 km.
```

```
// => Travel due East for 10 km.
// => Travel due West for 10 km.
// => Unknown heading: NE.
```

Выражение `switch`, которое проверяет строковые константы, выполняет такое же порядковое сравнение с учетом регистра, как и `==`, поэтому `"north"` не соответствует ветви `"North"`.

Выбор правильного сравнения

Выберите тип сравнения на основе данных, а не из привычки:

- Для идентификаторов, путей к файлам, токенов протокола и другого машинно-определяемого текста используйте Ordinal (или OrdinalIgnoreCase, чтобы игнорировать регистр). Порядковое сравнение выполняется быстро и дает одинаковые результаты в разных культурах.
- Для текста, который пользователи считывают и сортируют, например имена или названия продуктов, используйте CurrentCulture порядок, соответствующий ожиданиям пользователя.

Сравнение с учетом языковых и культурных особенностей применяет лингвистические правила, которые различаются в зависимости от культуры и могут приводить к неожиданным результатам. Например, в некоторых языках и региональных настройках `"ss"` и `"ß"` считаются равными, а порядок строк может различаться на разных компьютерах. Из-за этой изменчивости используйте сравнение с учётом языковых и региональных параметров только для настоящего текста на естественном языке и применяйте один и тот же тип сравнения при сортировке и поиске в коллекции.

Подробное рассмотрение культурно-зависимого сравнения, включая аспекты глобализации и различия между платформами, см. в статье Рекомендации по сравнению строк в .NET.

См. также

- Рекомендации по сравнению строк в .NET
- Строки поиска в C#
- StringComparison
- String.Compare

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.06.2026

Строки поиска в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** Методы C# `string`, такие как `Contains`, `StartsWith` и `IndexOf`, аналогичны методам в Java `String` и JavaScript `String.prototype`. Ключевое различие в том, что в некоторых операциях поиска в C# по умолчанию используется **порядковое сравнение с учетом регистра**. Другие по умолчанию используют семантику текущей культуры. Для пользовательских поисковых запросов может понадобиться передать значение StringComparison.

Класс String включает методы, которые отвечают на два повседневных вопроса:

- *Содержит ли эта строка этот текст?* — использование Contains, StartsWithили EndsWith.
- *Где происходит этот текст?* — использование IndexOf или LastIndexOf.

Более сложные алгоритмы поиска и замены можно создавать с помощью регулярных выражений. Дополнительные сведения о регулярных выражениях и других строковых операциях см. в статье справочника по языку Строковые операции.

Проверка того, содержит ли строка текст

Используйте `Contains`, `StartsWith`или `EndsWith` проверьте наличие подстроки:

C#

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Default comparisons are case sensitive.
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"""Contains "extension"? {containsSearchResult}""");
```

```
// For user-facing searches, pass a StringComparison value to control case and
culture.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"""Starts with "extension"? {ignoreCaseSearchResult} (ignoring
case)""");

bool endsWithSearchResult = factMessage.EndsWith(".", StringComparison.Ordinal);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
// => "Extension methods have all the capabilities of regular static methods."
// => Contains "extension"? False
// => Starts with "extension"? True (ignoring case)
// => Ends with '.'? True
```

В этих методах по умолчанию используется **регистрозависимое порядковое сравнение**. Чтобы принимать ввод пользователя или игнорировать регистр в отображаемом тексте, передайте значение StringComparison, например StringComparison.CurrentCultureIgnoreCase или StringComparison.OrdinalIgnoreCase.

При поиске одного символа используйте перегрузку `char` `Contains`. Это позволяет избежать выделения строки из одного символа и является более прямым способом:

C#

```
string path = "/usr/local/bin";
bool hasSlash = path.Contains('/');
Console.WriteLine($"Path contains '/': {hasSlash}");
// => Path contains '/': True
```

Найдите положение текста

`IndexOf` возвращает отсчитываемый от нуля индекс первого вхождения подстроки (или символа) и `LastIndexOf` возвращает индекс последнего вхождения. Оба возвращают `-1`, если поисковый текст отсутствует. Объедините их для извлечения текста между двумя маркерами:

C#

```
string factMessage = "Extension methods have all the capabilities of regular static
methods.";

Console.WriteLine($"\"{factMessage}\"");

// Extract the text between the first and last occurrence of "methods".
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
```

```
string between = factMessage.Substring(first, last - first);
Console.WriteLine($"""Substring between "methods" and "methods": '{between}'""");
// => "Extension methods have all the capabilities of regular static methods."
// => Substring between "methods" and "methods": ' have all the capabilities of
regular static '
```

Если требуется найти все вхождения, а не только первое или последнее, повторяйте поиск, передавая предыдущий результат плюс единицу в качестве аргумента `startIndex`, или используйте регулярное выражение.

Выбор правильного сравнения

Большинство перегрузок поиска принимают необязательное StringComparison значение. Выберите его на основе типа данных, которые вы ищете:

- Если вы ищете идентификаторы, пути к файлам, маркеры протокола или что-либо другое, определенное компьютером, используйте Ordinal.
- Если вы ищете машинно-определяемые данные того же типа, но хотите не учитывать регистр, используйте OrdinalIgnoreCase.
- Если вы ищете текст, отображаемый пользователю, для которого должны применяться правила текущей локали, используйте CurrentCulture.
- Если вы ищете тот же отображаемый пользователем текст и хотите игнорировать вариант, используйте CurrentCultureIgnoreCase.
- Если вы выполняете поиск по сохранённым данным, которые должны сравниваться одинаково на любом компьютере независимо от языка и региональных параметров, используйте InvariantCulture (требуется редко).

Порядковое сравнение является самым быстрым вариантом и правильным значением по умолчанию для всего, что не является текстом естественного языка. Сравнение с учетом культуры значительно медленнее и может привести к удивительным результатам. Например, в некоторых культурах `i` строчная буква не соответствует `I` прописной букве. Используйте это только для поисковых запросов по обычному тексту, которые выполняют пользователи.

Подробное описание сравнения с учетом языка и региональных параметров см. в рекомендациях по сравнению строк.

См. также

- Строковые операции: сопоставление шаблонов, производительность и поиск на основе диапазона
- Рекомендации по сравнению строк в .NET
- Сравнение строк
- System.String
- System.StringComparison

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.06.2026

Разделение строк на подстроки в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Переходите с другого языка?** `string.Split` В C# аналогами Java `String.split` и JavaScript `String.prototype.split` являются. В отличие от этих языков, C# возвращает массив (`string[]`), а не список, а аргумент разделителя является символом или строкой, а не регулярным выражением. Сведения о разбиении на основе шаблонов см. в разделе Regex.Split.

Метод String.Split разбивает строку на массив подстроок с помощью одного или нескольких разделителей. Это самый простой способ анализа текста с разделителями, таких как слова, значения стиля CSV или маркеры протокола.

Метод имеет много перегрузок, но они охватывают четыре независимых решения:

- **Разделители**: один `char`, массив `char`, один `string` или массив `string`.
- **Максимальное число результатов**: ограничение числа возвращаемых подстроок.
- **Обработка пустых элементов**: сохранять пустые подстроки (по умолчанию) или удалять их с помощью StringSplitOptions.RemoveEmptyEntries.
- **Обработка пробелов**: обрезать начальные и конечные пробелы у каждой записи StringSplitOptions.TrimEntries.

Разделение строки на слова

Чтобы разделить фразу на пробелы, передайте `' '` в качестве разделителя:

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
```

```
}
// => <The>
// => <quick>
// => <brown>
// => <fox>
// => <jumps>
// => <over>
// => <the>
// => <lazy>
// => <dog.>
```

Выполните итерацию возвращаемого массива, `for` чтобы восстановить позицию каждого слова:

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

for (int i = 0; i < words.Length; i++)
{
    Console.WriteLine($"Index {i}: <{words[i]}>");
}
// => Index 0: <The>
// => Index 1: <quick>
// => Index 2: <brown>
// => ...
```

Если входные данные содержат повторяющиеся экземпляры символа разделителя, `Split` создает пустые записи, по одному для каждого "пробела" между последовательными разделителями:

C#

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
// The runs of spaces produce empty entries:
// => <The>
// => <quick>
// => <brown>
// => <>
// => <>
// => <>
// => <fox>
// => ...
```

Передайте `StringSplitOptions.RemoveEmptyEntries`, чтобы отбросить эти пустые элементы, как будет показано далее в этой статье.

Разделение на несколько символов разделителя

Если несколько символов могут выступать в качестве разделителя, передайте их в виде массива. В следующем примере пробелы, запятые, точки, двоеточия и вкладки рассматриваются как границы слов:

C#

```
char[] delimiters = [' ', ',', '.', ':', '\t'];

string text = "one\ttwo three:four,five six seven";
Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiters);
Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
// => 7 words in text:
// => <one>
// => <two>
// => <three>
// => <four>
// => <five>
// => <six>
// => <seven>
```

Смежные разделители по-прежнему создают пустые записи:

C#

```
char[] delimiters = [' ', ',', '.', ':', '\t'];

string text = "one\ttwo :,five six seven";
Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiters);
Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
// => 7 words in text:
```

```
// => <one>
// => <two>
// => <>
// => <>
// => <five>
// => <six>
// => <seven>
```

Разделение по многосимвольным разделителям

Чтобы разделить целые слова или разделители с несколькими символами, передайте массив строк. Перегрузки для массива строк требуют значения StringSplitOptions. Используйте `RemoveEmptyEntries` , если повторяющиеся разделители в противном случае будут создавать пустые результаты:

C#

```
string[] separators = ["<<", "..."];

string text = "one<<two......three<four";
Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separators, StringSplitOptions.RemoveEmptyEntries);
Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    Console.WriteLine(word);
}
// => 3 substrings in text:
// => one
// => two
// => three<four
```

Ограничьте количество подстрок, которые возвращаются

Передайте аргумент `count`, чтобы ограничить количество результатов. Последняя запись содержит все, что осталось, включая остальные разделители:

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ', 4, StringSplitOptions.None);

foreach (var word in words)
{
```

```
    Console.WriteLine($"<{word}>");
}
// => <The>
// => <quick>
// => <brown>
// => <fox jumps over the lazy dog.>
```

Этот шаблон удобно подходит для `key=value` пар и других форматов, где имеет смысл только первый разделитель.

Обрезать пробелы из каждой записи

`StringSplitOptions.TrimEntries` удаляет начальные и конечные пробельные символы из каждой возвращаемой подстроки. Его можно использовать вместе с `RemoveEmptyEntries` для типичной очистки в стиле CSV:

C#

```
string numerals = "1, 2, 3, 4, 5, 6, 7, 8, 9, 10";
string[] trimmed = numerals.Split(',', StringSplitOptions.TrimEntries);

Console.WriteLine("Trimmed entries:");
foreach (var word in trimmed)
{
    Console.WriteLine($"<{word}>");
}

string[] untrimmed = numerals.Split(',', StringSplitOptions.None);
Console.WriteLine("Untrimmed entries:");
foreach (var word in untrimmed)
{
    Console.WriteLine($"<{word}>");
}
// => Trimmed entries: <1> <2> ... <10>
// => Untrimmed entries: <1> < 2> ... < 10>
```

Использование регулярных выражений

`Split` хорошо подходит для фиксированных символов или разделителей строк. Для разделения на основе шаблонов используйте Regex.Split. Общие сведения о регулярных выражениях строк см. в разделе "Строки".

См. также

- String.Split

- System.StringSplitOptions
- Строковые операции: сопоставление шаблонов, производительность и поиск на основе диапазона
- Извлечение элементов из строки

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.06.2026

Руководство. Использование интерполяции строк для создания отчета в C#

В этом руководстве вы создадите небольшое файловое приложение , которое печатает отформатированную сводку заказа. Попутно вы применяете интерполяцию строк, чтобы вставлять значения в текст, форматировать числа и денежные суммы, выравнивать столбцы и получать вывод с учетом языка и региональных параметров.

В этом руководстве предполагается, что вы знакомы с основными понятиями C#. Справочник по функциям синтаксиса интерполяции см. в разделе "Интерполяция строк".

Необходимые условия

- Пакет SDK .NET 10 или более поздняя версия. Для приложений на основе файлов требуется .NET 10 или более поздней версии.
- Редактор кода, например Visual Studio Code.

Создание приложения

Приложение на основе файлов — это один `.cs` файл, который выполняется без файла проекта. Создайте папку для приложения и перейдите в нее:

Интерфейс командной строки.NET

```
mkdir StringInterpolation
cd StringInterpolation
```

Создайте пустой файл *с именем report.cs* в этой папке и откройте его в редакторе. Вы добавляете в него код в следующих разделах, а приложение запускаете с помощью `dotnet run report.cs`.

Вставка значений в текст

Добавьте `$` перед строковым литералом, чтобы сделать его *интерполируемой строкой*. В строке поместите любое выражение C# в фигурные скобки (`{` и `}`); C# вычисляет выражение, преобразует результат в строку и вставляет его.

Добавьте следующий код в *report.cs*, а затем запустите приложение с помощью `dotnet run report.cs`:

C#

```
string name = "Maria";
int itemCount = 3;

// Prefix a string literal with $ to interpolate expressions inside braces.
Console.WriteLine($"Hello, {name}! You have {itemCount} items in your cart.");
// => Hello, Maria! You have 3 items in your cart.
```

Результат объединяет литеральный текст со значениями `name` и `itemCount`.

Форматирование чисел и валют

Сводка заказа показывает деньги и ставки, поэтому необработанные числа, как `23.5` или `0.08` выглядят незавершенными и трудными для чтения. Представить каждое значение таким образом, как клиент ожидает: цена с символом валюты и налоговой ставкой в процентах. Отформатируйте его как часть интерполяции.

Чтобы отформатировать значение, следуйте выражению с двоеточием (`:`) и строкой формата. Стандартные строки форматирования `C` и `P0` создают денежные и целые проценты. Добавьте следующий код в конец файла:

C#

```
decimal subtotal = 23.5m;
decimal taxRate = 0.08m;

// Follow an expression with :format to apply a standard or custom format string.
Console.WriteLine($"Subtotal: {subtotal:C}");
Console.WriteLine($"Tax rate: {taxRate:P0}");
Console.WriteLine($"Total:    {subtotal * (1 + taxRate):C}");
// => Subtotal: $23.50
// => Tax rate: 8%
// => Total:    $25.38
```

Строка формата учитывает параметры текущей культуры, поэтому символ валюты и разделители соответствуют настройкам компьютера.

Выровнять значения по столбцам

Когда вы печатаете несколько строк данных, значения разной длины делают столбцы неровными, и их становится трудно просматривать. Чтобы выстраивать каждое имя и число под согласованным заголовком, присвойте каждому полю фиксированную ширину, чтобы столбцы оставались прямыми независимо от того, насколько долго каждое значение.

Чтобы задать минимальную ширину поля, следуйте выражению с запятыми (`,`) и шириной. Положительная ширина выравнивает значение по правому краю; отрицательная ширина — по левому краю. Объединяет ширину и строку формата в виде `{expression,width:format}`. Добавьте следующий код для печати табличной сводки:

C#

```
(string Name, int Quantity, decimal Price)[] orders =
[
    ("Espresso", 2, 3.50m),
    ("Cappuccino", 1, 4.25m),
    ("Tea", 4, 2.00m),
];

// Follow an expression with ,width to set a minimum field width.
// A positive width right-aligns; a negative width left-aligns.
foreach (var order in orders)
{
    Console.WriteLine($"{order.Name,-12}{order.Quantity,3}{order.Price *
order.Quantity,10:C}");
}
// => Espresso      2     $7.00
// => Cappuccino    1     $4.25
// => Tea           4     $8.00
```

Выровненный по левому краю столбец с именами и выровненные по правому краю числовые столбцы образуют удобочитаемую таблицу.

Формирование вывода с учетом языковых и региональных параметров

Интерполированная строка по умолчанию использует текущие региональные параметры. Когда вам нужна определенная культура, например фиксированный формат для журналов или языковой стандарт для квитанции, передайте культуру в String.Create(IFormatProvider, DefaultInterpolatedStringHandler). Добавьте следующий код:

C#

```
decimal total = 1234.56m;
```

```
// An interpolated string uses the current culture by default.
// Use string.Create with a culture to control the formatting explicitly.
string germanReceipt = string.Create(
    new CultureInfo("de-DE"), $"Gesamt: {total:C}");
string invariantLog = string.Create(
    CultureInfo.InvariantCulture, $"total={total:F2}");

Console.WriteLine(germanReceipt);
Console.WriteLine(invariantLog);
// => Gesamt: 1.234,56 €
// => total=1234.56
```

Немецкая квитанция использует запятую в качестве десятичного разделителя и символа евро. В инвариантном журнале используется точка и не используется никакой символ, независимо от языка и региональных параметров компьютера.

Дальнейшие действия

Вы использовали интерполяцию строк для вставки, форматирования, выравнивания и локализации значений. Полный набор функций интерполяции, включая необработанные строковые литералы, экранированные фигурные скобки и интерполированные строки констант, см. в статье о концепции:

Интерполяция строк в C#

См. также

- Интерполяция строк в C#
- Составное форматирование
- Типы форматирования в .NET
- String.Format

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.06.2026

Общие сведения о сопоставлении шаблонов

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы. Опытные разработчики C# могут использовать ссылки на шаблоны для полного синтаксиса и правил.
>
> **Вы пришли из другого языка?** Шаблоны C# служат ролью, аналогичной сопоставлению шаблонов Java и случаям Python `match`. В C# используются шаблоны в `is` выражениях, `switch` инструкциях и `switch` выражениях.

Сопоставление шаблонов применяет шаблон к выражению. *Шаблон* — это условие для проверки входного выражения. *Ввод шаблона* представляет собой это выражение. C# вычисляет входное выражение; Результатом является *вычисляемое значение*. Шаблон проверяет, имеет ли значение определённый тип, равно ли оно определённому значению или имеет определённую структуру. При совпадении шаблона код может использовать сведения, полученные тестом, например более конкретный тип или значение, извлеченные из объекта.

Шаблон можно использовать в трех контекстах:

- Справа от `is` оператора.
- `case` В метке инструкции `switch`.
- В ветви выражения `switch`.

Шаблоны часто понятнее, чем последовательность выражений сравнения, так как каждая ветвь описывает данные, с которыми она работает. Например, следующий метод использует `switch` выражение для выбора сообщения доставки:

C#

```
static string GetDeliveryMessage(Delivery? delivery) =>
    delivery switch
    {
```

```
        null => "No delivery was scheduled.",
        ExpressDelivery express => $"Express package {express.TrackingCode}",
        StandardDelivery { Days: <= 2 } => "Standard delivery arriving soon",
        _ => "Standard delivery"
    };
```

Рекурсивные шаблоны имеют собственные входные выражения. В `StandardDelivery { Days: <= 2 }` внешний шаблон получает выражение `delivery`. Рекурсивный `<= 2` шаблон получает `Days` выражение свойства из соответствующего `StandardDelivery` объекта.

Выражение, прежде чем `switch` является входным выражением. Каждая строка внутри фигурных скобок — это *контакт переключателя*. Шаблон отображается раньше `=>`, и результат появляется после него. C# вычисляет входное выражение, а затем выбирает первую по порядку ветвь в тексте, шаблон которой соответствует выражению и необязательное защитное условие `when` которой равно `true`. Необязательное `when` защитное условие — это дополнительное булево условие, которое записывается после шаблона. В предыдущем примере показаны следующие шаблоны:

- `null` — это *константный шаблон*. Проверяет, вычисляется ли выражение `null` как `delivery`.
- `ExpressDelivery express` — это *шаблон объявления* с двумя частями. `ExpressDelivery` — это часть типовых испытаний. Проверяет, является ли вычисляемое значение объектом, не равным null, тип времени выполнения которого совместим с `ExpressDelivery`. `express` — *это обозначение переменной*: она объявляет переменную с именем `express` и назначает ему соответствующий `ExpressDelivery` объект.
- `StandardDelivery { Days: <= 2 }` начинается с проверки типа. `StandardDelivery` проверяет, является ли вычисляемое значение объектом, не допускающим значение NULL этого типа. Фигурные скобки содержат *шаблон свойства*. `Days` присваивает имя проверяемого свойства, поэтому `Days` выражение свойства становится входными данными в рекурсивном шаблоне. Часть `<= 2` — это *реляционный шаблон*, который проверяет, что вычисляемое значение меньше либо равно `2`.
- `_` (символ подчеркивания) — это *шаблон отбрасывания*. Он соответствует каждому вычисленному значению, включая `null`. Поскольку предыдущие ветви уже обрабатывают `null`, экспресс-доставки и стандартные доставки, поступающие в течение двух дней, эта последняя ветвь обрабатывает все остальные значения.

Рука без охранника `when` *не охраняется*. Все руки в первом примере не охраняются. Если более ранняя ветвь без защитного условия соответствует каждому проверяемому значению, которому может соответствовать более поздняя ветвь, то более поздняя ветвь *поглощается*. *Поглощённый* шаблон — это шаблон, для которого любое возможное

входное значение, соответствующее ему, уже было сопоставлено более ранней веткой switch. Он никогда не может совпадать, поэтому компилятор сообщает об ошибке. Отбрасывающая ветвь (`_`) должна идти последней, так как она соответствует любому входному выражению. Защищенная рука не подменяет более позднюю руку на основе его шаблона только потому, что охранник может быть `false`.

Switch-выражение *считается исчерпывающим*, если его ветви обрабатывают все возможные входные значения. Первый пример является исчерпывающим, поскольку его последняя отбрасывающая ветка обрабатывает всё, что не соответствует ни одной из предыдущих веток. Компилятор предупреждает, когда обнаруживает потенциальное входное значение, которое не обрабатывается рукой. Компилятор не может доказать исчерпывающее значение для каждого сочетания шаблонов, но эта диагностика помогает создавать правильный код сопоставления шаблонов. Подробные сведения о правилах сопоставления, включения и полноты см. в справочнике по шаблонам.

Проверка одного условия с помощью `is`

Используйте оператор, `is` если требуется один логический тест. Следующий код вычисляет выражение `delivery` и применяет шаблон объявления `ExpressDelivery express`. Часть, относящаяся к типу, соответствует, если вычисляемое значение не является null, а его тип во время выполнения совместим с `ExpressDelivery`. Когда шаблон совпадает, его обозначение переменной объявляет `express`:

C#

```
static void PrintTrackingCode(Delivery delivery)
{
    if (delivery is ExpressDelivery express)
    {
        Console.WriteLine($"Track express package {express.TrackingCode}");
    }
}
```

Компилятор отслеживает, назначена ли локальная переменная перед чтением кода. Такое отслеживание называется *определённым присваиванием*. `if` Внутри этого блока условие может быть `true` только в том случае, если шаблон присвоил совпавший объект переменной `express`. Поэтому компилятор знает, что `express` определенно назначается там. Код может безопасно использовать его `TrackingCode` свойство. Вам не нужен отдельный приведение. Если вам нужен только тест типа и не нужно объявлять переменную, используйте шаблон типа, например `delivery is ExpressDelivery`.

Используйте `is null` или `is not null` для проверок NULL. Эти шаблоны не вызывают определяемые пользователем операторы `==` или `!=`.

C#

```
static void PrintPackageStatus(Delivery? delivery)
{
    if (delivery is null)
    {
        Console.WriteLine("No package is available.");
    }
    else
    {
        Console.WriteLine("A package is ready to track.");
    }
}
```

Дополнительные рекомендации по безопасности null см. в разделе "Ссылочные типы, допускающие значение NULL".

Выберите между инструкцией и выражением

Шаблоны работают с обеими `switch` формами:

- Используйте switchинструкцию, если для каждого совпадения нужно выполнить одну или несколько инструкций.
- Используйте выражение `switch`, когда каждое совпадение должно возвращать значение.

Приведённый ниже оператор switch сообщает об обновлении доставки. Ветвь экспресс-доставки записывает два сообщения, поэтому инструкция подходит естественно:

C#

```
static void PrintDeliveryUpdate(Delivery? delivery)
{
    switch (delivery)
    {
        case null:
            Console.WriteLine("No delivery was scheduled.");
            break;
        case ExpressDelivery express:
            Console.WriteLine($"Express delivery {express.TrackingCode} is ready.");
            Console.WriteLine("Notify the priority desk.");
            break;
        case StandardDelivery standard:
            Console.WriteLine($"Standard delivery arrives in {standard.Days} days.");
```

```
            break;
        default:
            Console.WriteLine("Another delivery type is scheduled.");
            break;
    }
}
```

Каждая из них `case` применяет шаблон к выражению `delivery` . Соответствующий раздел может выполнять любое количество операторов перед `break` выходом из коммутатора. Раздел `default` обрабатывает всё, что не подпадает под предыдущие условия.

Следующее выражение коммутатора заменяет цепочку `if` / `else if` , которая назначает один результат:

C#

```
static string GetStatusMessage(StandardDelivery delivery) =>
    delivery.Days switch
    {
        0 => "Delivered today",
        1 => "Arriving tomorrow",
        <= 3 => "Arriving soon",
        _ => "In transit"
    };
```

Выражение переключателя является кратким, так как каждая рука имеет ту же цель: создает возвращаемое значение. Используйте инструкцию switch, когда ветви выполняют действия, и используйте switch-выражение, когда ветви вычисляют единственный результат.

Категории узоров

C# включает шаблоны для распространенных типов тестов данных:

Развернуть таблицу

Категория шаблонов	Что он проверяет
Объявление, константы и var шаблоны	Тип времени выполнения, определенное константное значение или любое значение, которое требуется записать.
Шаблоны типов	Тип времени выполнения без объявления переменной
Свойства и позиционные шаблоны	Свойства, поля или деконструированные значения
Реляционные и логические	Сравнения и сочетания, такие как `and`, `or` и `not`

Категория шаблонов	Что он проверяет
шаблоны	
Шаблоны списков	Значения и форма списка или массива
Отмена шаблонов и отмены	Любое оставшееся значение или значение, которое код намеренно игнорирует

Статьи из раздела «Основы», ссылки на которые приведены в таблице, содержат материалы по категориям, описанным в этом разделе на данный момент. Полный синтаксис и примеры для всех категорий шаблонов см. в справочнике по шаблонам.

См. также

- Объявление, константы и var шаблоны
- Шаблоны типов
- Отбрасывания
- Справочник по шаблонам
- switch Справочник по выражениям

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 18.09.2026

Объявление, константа и шаблоны `var`

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Начните с **обзора сопоставления шаблонов** , если вы раньше не использовали шаблоны C#. Полные языковые правила см. **в справочнике по шаблонам**.

Шаблон применяется к *входным выражениям*. C# вычисляет выражение, а затем тестирует шаблон или записывает полученное значение. Объявление, константа и `var` шаблоны отвечают на три практических вопроса:

- **Шаблон объявления:** Создает ли выражение ненулевое значение совместимого типа времени выполнения? Если да, объявите переменную для этого значения.
- **Шаблон констант:** Создает ли выражение одно определенное константное значение?
- `var` **шаблон:** какое значение производит выражение? Захватите его без предварительной проверки его типа или значения.

Проверить и захватить тип с помощью шаблона объявления

Шаблон объявления состоит из типа и *обозначения*. Тип указывает тип времени выполнения для тестирования. Обозначение объявляет переменную, которая получает соответствующее значение.

В следующем примере получается выражение `object`, поэтому выражение может создавать различные типы. Шаблон объявления позволяет соответствующей ветви использовать десятичную сумму без отдельного теста типа и приведения:

C#

```
static void PrintPrice(object value)
{
    if (value is decimal amount)
    {
        Console.WriteLine($"Price: {amount:C}");
```

```
    }
}
```

В `value is decimal amount`:

- `value` — это входное выражение. C# сначала вычисляет его.
- `decimal` — тестируемый тип. Шаблон совпадает, если вычисляемое значение не равно null, а его тип времени выполнения совместим с `decimal`.
- `amount` — это обозначение. Когда шаблон совпадает, объявляется `amount`, и ему присваивается значение типа decimal.

Компилятор отслеживает, получает ли локальная переменная значение, прежде чем код считывает его. Это отслеживание называется *определенным присваиванием*. Внутри блока `if` компилятор знает, что `amount` было присвоено значение, поскольку блок выполняется только в случае совпадения с шаблоном. Компилятор создает ошибку, если код пытается получить доступ `amount` за пределами `if` блока. Если `value` не является значением `decimal`, переменной `amount` не присваивается значение.

Выберите шаблон объявления, когда соответствующая ветвь должна использовать результат в качестве тестового типа. Он объединяет проверку, преобразование типов и объявление переменной, что позволяет избежать повторения выражения или выполнения отдельного приведения типов.

Вы также можете использовать шаблоны объявлений, если одно выражение может создать несколько полезных типов:

C#

```
static string FormatSensorValue(object reading) =>
    reading switch
    {
        int count => $"Count: {count}",
        double temperature => $"Temperature: {temperature:F1}°C",
        string message => $"Message: {message}",
        _ => "Unsupported reading"
    };
```

Каждая ветвь объявляет переменную совпавшего типа, поскольку результату требуется логика форматирования для этого типа. Шаблон объявления совпадает только в том случае, если вычисляемое значение не равно NULL и уже имеет тип времени выполнения, совместимый с тестируемым типом через преобразования, разрешенные для шаблонов. `null` не имеет типа времени выполнения для сопоставления шаблона. Шаблон также не

выполняет определяемые пользователем операторы преобразования: это тест типа и запись, а не запрос на преобразование значения в другой тип. Полные правила совместимости см. в разделе "Объявления" и "шаблоны типов".

Сопоставление определенного значения с константным шаблоном

Константный шаблон проверяет, создает ли выражение определенную константу, например число, строку, логический элемент, член перечисления, объявленное `const` значение или `null`.

Шаблоны констант соответствуют выражению коммутатора, когда несколько известных значений создают другой результат:

C#

```
static string GetCommandMessage(Command command) =>
    command switch
    {
        Command.Start => "Starting",
        Command.Stop => "Stopping",
        Command.Pause => "Pausing",
        _ => "Unknown command"
    };
```

`Command` — это перечисление, тип, определяющий набор именованных констант. `Command.Start`, `Command.Stop` и `Command.Pause` являются его элементами перечисления, поэтому каждая рука коммутатора использует постоянный шаблон для проверки одного именованного значения команды.

Выберите эту форму, когда команда может иметь несколько дискретных значений. Ветви switch сохраняют значения и соответствующие им результаты вместе. Для одной простой проверки на равенство инструкцию `if (command == Command.Start)`, например `if`, обычно легче читать.

Сопоставление константных шаблонов использует встроенные правила равенства языка, а не определяемый `==` пользователем оператор. Подробные сведения о правилах равенства и преобразования см. в справочнике по шаблонам констант.

Шаблон `null` константы полезен для надежной проверки null:

C#

```
static bool HasText(string? text) => text is not null;
```

Выберите `is null` или `is not null` при проверке состояния NULL. Эти шаблоны не вызывают определяемый пользователем оператор равенства, даже если для типа выражения перегружен оператор `==`.

Захватить результат для guard с шаблоном `var`

Шаблон `var` соответствует каждому результату, включая `null` и объявляет переменную, тип которой является типом входного выражения во время компиляции. Он может записать вычисленное значение, а другой шаблон уже соответствует объекту:

C#

```
static string GetDeliveryMessage(object delivery) =>
    delivery switch
    {
        ExpressDelivery express
            when EstimateDays(express) is var days && days <= 2
                => $"Arrives in {days} day{(days == 1 ? "" : "s")}",
        ExpressDelivery => "Express delivery for your location takes more than two
days",
        _ => "Standard delivery"
    };

static int EstimateDays(ExpressDelivery delivery) =>
    delivery.MilesAway <= 500 ? 1 :
    delivery.MilesAway <= 1_000 ? 2 : 3;

record ExpressDelivery(int MilesAway);
```

Шаблон `ExpressDelivery express` объявления сначала фиксирует объект доставки как `express`. Вызов `EstimateDays(express)` метода — это входное выражение для `var` шаблона. C# вычисляет этот вызов метода и `var days` записывает итоговую оценку, как `days` без тестирования типа или значения. Срок оценки может составлять один-два дня, если проверка проходит успешно. Для вывода фактического количества дней результату ветви требуется захваченное значение.

Обычную локальную переменную нельзя объявить между шаблоном ветви switch и её условием `when`. Повторный вызов `EstimateDays(express)` в результирующем выражении приведёт к повторному вычислению. Выберите эту `var` форму шаблона, когда код уже соответствует, и охранник и результат требуют вычисленного промежуточного значения.

Если захваченное значение не требуется, используйте шаблон _отбрасывания вместо объявления переменной.

См. также

- Обзор сопоставления шаблонов
- Шаблоны типов
- Отбрасывания и шаблон отбрасывания.
- Справочник по шаблонам объявления и типа.
- Ссылка на шаблон константы.
- var Ссылка на шаблон.

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 18.09.2026

Шаблоны типов

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Начните с **обзора сопоставления шаблонов** , если вы не знакомы с шаблонами. Полные правила совместимости см. в **объявлениях и шаблонах типов** в справочнике по языку.

Шаблон типа применяется к входным выражениям. C# вычисляет выражение, а затем проверяет, что результирующее значение не равно null и что его тип времени выполнения совместим с указанным типом. Шаблон типа сообщает только о том, успешно ли выполнен тест типа. Он не объявляет переменную.

Задать вопрос, предполагающий ответ «да» или «нет»

В системе доставки `IRouteStop` — это контракт, определяющий возможности для пунктов назначения, которые принимаются системой планирования маршрутов. `CanRoute` проверяет, обладает ли проверяемый пункт назначения такой возможностью, и полученное логическое значение определяет, включается ли пункт назначения в процесс планирования маршрута. Рабочему процессу нужен только ответ «да» или «нет», и он не читает никаких элементов route-stop, поэтому подходит шаблон сопоставления с типом без захватываемой переменной.

C#

```
static bool CanRoute(object? destination) =>
    destination is IRouteStop;
```

Входное выражение — `destination`, а `IRouteStop` — это проверяемый тип. Выберите тип шаблона, если возможны только ответы «да» или «нет». Если в совпадающей ветви нужно получить адрес или обратиться к другому члену через `IRouteStop`, вместо этого выберите шаблон объявления, чтобы у ветви была переменная этого типа.

> **Примечание.**

Вы также можете увидеть `destination is IRouteStop _`. Этот синтаксис представляет собой шаблон объявления, в котором `_` означает, что никакая переменная не сохраняется. Он выполняет одну и ту же проверку типа, если обе формы допустимы, но `destination is IRouteStop` более прямо указывает, что это предназначено только для тестирования.

Сопоставление классов и интерфейсов

В системе доставки `IRouteStop` определяет контракт функциональности для пунктов назначения планирования маршрутов, а `RouteStop` — это базовый класс, предоставляющий общие данные об остановках маршрута и их реализацию. `ExpressRouteStop` — это специализированный класс, производный от `RouteStop`. В примере проверяется одно вычисляемое значение, чтобы его выходные данные могли продемонстрировать все три совместимых типа. Для каждого теста требуется только логический результат, поэтому не требуется записанная переменная.

C#

```
interface IRouteStop { }

abstract class RouteStop(string street) : IRouteStop
{
    public string Street { get; } = street;

    public string GetDisplayName() => Street;
}

sealed class ExpressRouteStop(string street) : RouteStop(street)
{
}

static void ShowCompatibility()
{
    object destination = new ExpressRouteStop("8 Oak Avenue");

    Console.WriteLine($"Exact class: {destination is ExpressRouteStop}");
    Console.WriteLine($"Base class: {destination is RouteStop}");
    Console.WriteLine($"Interface: {destination is IRouteStop}");
}
```

Вычисленное значение — `ExpressRouteStop`. Выходные данные показывают, что он соответствует точному классу, его `RouteStop` базовому классу и интерфейсу `IRouteStop` , который `RouteStop` реализует.

Подробные правила совместимости и описание особых случаев см. в справочнике по шаблонам типов.

Маршрутизируйте несколько типов

В системе поддержки `SupportRequest` является базовым классом для специализированных классов запросов, таких как `PasswordResetRequest` и `BillingQuestion`. *Ветвь switch* связывает шаблон с результатом, который возвращается при совпадении с этим шаблоном. Каждая рука возвращает имя очереди обработки, а последняя рука предоставляет резервную очередь для других вычисляемых значений. Выбранная очередь зависит только от типа времени выполнения вычисляемого значения, поэтому шаблоны типов подходят, так как член запроса не считывается.

C#

```
static string RouteRequest(object request) =>
    request switch
    {
        PasswordResetRequest => "Identity queue",
        BillingQuestion => "Billing queue",
        SupportRequest => "General support queue",
        _ => "Intake queue"
    };
```

Каждая ветвь определяет тип запроса и возвращает очередь, обрабатывающую этот запрос. Если рука должна считывать элементы запроса, используйте шаблон объявления для записи соответствующего значения в переменной.

Плечи переключателя рассматриваются сверху вниз. Поместите специализированный класс перед базовым классом. В противном случае ветка базового класса может совпадать с любым экземпляром специализированного класса, из-за чего последующая ветка становится недостижимой.

Необязательно. Используйте параметр типа в качестве тестового типа.

Этот необязательный пример основан на универсальных типах и методах. *Параметр типа*, такой как `TRequest`, служит местозаполнителем для типа, который предоставляет вызывающий код. Пакет входящих запросов может содержать несколько типов запросов, а вызывающий предоставляет `ConfidentialRequest` данные для проверки необходимости конфиденциальной обработки любого запроса. Логический результат выбирает

конфиденциальную обработку для всего пакета и создает видимое сообщение о состоянии. Шаблон типа с параметром типа подходит, так как необходимо сохранить только существование соответствующего запроса, поэтому соответствующий объект не должен храниться.

C#

```
static void ShowConfidentialBatchHandling()
{
    object[] incomingRequests = [new BillingQuestion(), new ConfidentialRequest()];
    bool requiresConfidentialHandling =
        ContainsRequestOfType<ConfidentialRequest>(incomingRequests);

    Console.WriteLine(requiresConfidentialHandling
        ? "Send the entire batch to confidential handling."
        : "Send the batch to standard handling.");
}

static bool ContainsRequestOfType<TRequest>(IEnumerable<object> requests)
{
    foreach (object request in requests)
    {
        if (request is TRequest)
        {
            return true;
        }
    }

    return false;
}
```

Если вызывающей стороне требуется сам соответствующий запрос, более подходящей была бы операция поиска или фильтрации, возвращающая соответствующие элементы.

См. также

- Обзор сопоставления шаблонов
- Объявление, константы и var шаблоны
- Универсальные типы и методы
- Операторы приведения и тестирования типов
- Справочник по шаблону типов

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 18.09.2026

Свойства и позиционные шаблоны

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Начните с **обзора сопоставления шаблонов** , если вы не знакомы с шаблонами. Полные языковые правила см. в **шаблонах свойств** и **позиционных шаблонах** в справочнике по языку.

Шаблоны свойств и позиционные шаблоны оба проверяют части значения. Разница заключается в том, как они определяют эти части:

- *Шаблон свойства* указывает свойства или поля, которые нужно проверять.
- *Позиционный шаблон* определяет значения по их порядку.

Деконструкция предоставляет упорядоченный набор значений компонентов. Кортеж уже имеет порядок элементов; см. деконструкцию кортежей. Для другого типа Deconstruct метод определяет, какие значения компонентов предоставляются и их порядок.

Сравнение имен и позиций

Следующий шаблон сопоставления по свойствам проверяет два именованных свойства погодного показания со значениями температуры в градусах по Цельсию:

C#

```
static bool IsHotAndHumid(WeatherReading reading) =>
    reading is { TemperatureC: > 30, HumidityPercent: > 70 };

sealed record WeatherReading(int TemperatureC, int HumidityPercent);
```

Следующий позиционный шаблон проверяет значение сигнала, после которого следует логическое значение:

C#

```
static string GetCrossingInstruction(
    PedestrianSignal signal, bool crossingIsClear) =>
    (signal, crossingIsClear) switch
    {
```

```
        (PedestrianSignal.Walk, true) => "Cross now",
        (PedestrianSignal.Walk, false) => "Wait for the crossing to clear",
        _ => "Wait for the walk signal"
    };

enum PedestrianSignal
{
    Stop,
    Walk
}
```

Шаблон свойства определяет входные данные по именам `TemperatureC` и `HumidityPercent`. Код пересечения создает кортеж из отдельных `signal` и `crossingIsClear` значений. Затем шаблон кортежа идентифицирует эти значения по порядку: сначала `signal`, а затем `crossingIsClear`. Позиционный шаблон здесь удачно подходит, поскольку этот только что созданный кортеж содержит всего два значения, и их порядок имеет однозначный смысл при принятии решения о пересечении.

Выберите шаблон свойств, когда имена участников помогают объяснить тест. Шаблоны свойств обычно более понятны для классов, структур и записей. Выберите позиционный шаблон, если порядок уже придаёт значениям очевидный смысл. Позиционные шаблоны наиболее полезны для кортежей, которые объединяют несколько связанных значений в одно значение с фиксированным порядком.

Следуйте вложенным входным данным в рекурсивных шаблонах

Свойства и позиционные *шаблоны являются рекурсивными шаблонами*: они применяют другой шаблон к каждому свойству, полю или расположению, которые они выбирают. Выбранное значение становится входными данными в вложенный шаблон.

`reading` Выражение `IsHotAndHumid` — это входные данные в шаблон свойств. C# вычисляет это выражение перед сопоставлением. Шаблон получает два значения из результирующего объекта:

- Реляционный шаблон `> 30` проверяет значение `TemperatureC`.
- Реляционный шаблон `> 70` проверяет значение `HumidityPercent`.

Тест внешнего типа является необязательным, а предложения рекурсивных шаблонов могут быть пустыми. Например, шаблон `{ }` пустого свойства соответствует любому вычисляемому значению, отличному от NULL.

Шаблоны свойств и позиционные шаблоны сопоставляются только с вычисленными значениями, отличными от NULL. Когда `null` входит в входной домен, выберите рекурсивный шаблон, который сначала проверяет значение, отличное от NULL:

C#

```
static string DescribeNullableInput(object? value)
{
    if (value is not { } nonNullValue)
    {
        return "No value";
    }

    return nonNullValue switch
    {
        DateTime => "Date",
        string => "Text",
        _ => "Another type"
    };
}
```

Входное выражение — `value`. C# вычисляет это выражение, а шаблон свойства `{ }` проверяет, что результирующее значение не равно null, перед его присваиванием переменной `nonNullValue`. Следующее выражение коммутатора может протестировать несколько возможных типов среды выполнения. Его `DateTime` и `string` шаблоны типов не имеют обозначения, так как метод должен определять только каждый тип, а не захватывать его значение.

Если предложения рекурсивных шаблонов содержат вложенные шаблоны, каждое выбранное свойство, поле или положение становится входными данными в вложенный шаблон.

Вы можете добавить тест типа перед фигурными скобками, когда входное выражение может создавать различные типы. Вы также можете использовать путь члена для тестирования вложенного свойства:

C#

```
static string DescribeDate(object? value) =>
    value switch
    {
        DateTime { Date.DayOfWeek:
            DayOfWeek.Saturday or DayOfWeek.Sunday } => "Weekend date",
        DateTime => "Weekday date",
        null => "No date",
```

```
        _ => "Not a date"
    };
```

`value` — это входное выражение. C# сначала вычисляет это и проверяет, является ли полученное значение значением типа DateTime. Затем значение свойства `Date` используется как входные данные для обращения к члену `DayOfWeek`. Наконец, `DayOfWeek` значение становится входным в логический шаблон, который проверяет две константы. Сопоставление завершается успешно, если внешнее значение имеет указанный тип, и каждый объект, необходимый вдоль пути элемента, не имеет значения NULL.

Сравнение шаблонов с операторами ветвления

`DescribeDate` Предыдущий метод выражает четыре результата в виде шаблонов:

C#

```
static string DescribeDate(object? value) =>
    value switch
    {
        DateTime { Date.DayOfWeek:
            DayOfWeek.Saturday or DayOfWeek.Sunday } => "Weekend date",
        DateTime => "Weekday date",
        null => "No date",
        _ => "Not a date"
    };
```

Следующий метод дает тот же результат с использованием последовательности императивных операторов ветвления:

C#

```
static string DescribeDateWithBranches(object? value)
{
    if (value is DateTime date)
    {
        if (date.DayOfWeek == DayOfWeek.Saturday ||
            date.DayOfWeek == DayOfWeek.Sunday)
        {
            return "Weekend date";
        }

        return "Weekday date";
    }

    if (value is null)
    {
        return "No date";
```

```
    }

    return "Not a date";
}
```

Версия на основе шаблона сохраняет возможные результаты вместе, когда несколько ветвей проверяют тип и фигуру значения. Императивная версия делает каждую проверку и каждый шаг возврата явными. При одном условии обе формы могут выглядеть похоже. По мере роста числа связанных ветвей шаблоны могут упростить сравнение альтернативных вариантов.

См. также

- Обзор сопоставления шаблонов
- Реляционные, логические шаблоны и шаблоны в скобках.
- Деконструкция кортежей и других типов
- Справочник по шаблону свойств.
- Ссылка на позиционный шаблон.

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 25.09.2026

Реляционные, логические и заключённые в скобки шаблоны

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Начните с **обзора сопоставления шаблонов** , если вы не знакомы с шаблонами. Полные языковые правила см. в **реляционных шаблонах** и **логических шаблонах** в справочнике по языку.

Реляционные и логические шаблоны описывают диапазоны, альтернативы и исключения. Следующий метод объединяет их для классификации температуры:

C#

```
static string ClassifyTemperature(int temperature) =>
    temperature switch
    {
        < 0 => "Below freezing",
        >= 18 and <= 24 => "Comfortable",
        (>= 0 and < 10) or > 30 => "Far outside the comfortable range",
        _ => "Cool or warm"
    };
```

Выражение `temperature` — это входные данные шаблона. C# вычисляет это выражение один раз, и каждая ветвь switch проверяет полученное значение. Оружие демонстрирует следующие тесты:

- `< 0` проверяет одно граничное значение.
- `>= 18 and <= 24` проверяет диапазон.
- `(>= 0 and < 10) or > 30` проверяет два альтернативных диапазона.

В этой статье показаны как шаблоны, так и императивные условия, чтобы вы могли узнать каждую форму и сравнить, как они выражают одни и те же решения. Одно условие может выглядеть аналогично в любой форме. Шаблоны могут упростить чтение ряда связанных ветвей, сохраняя выбор рядом с их результатами. Выберите форму, которая упрощает понимание кода.

Сравнение значений с реляционными шаблонами

Реляционный шаблон сравнивает входные данные шаблона с константой времени компиляции с помощью `<`, `>` `<=`или `>=`. *Константой во время компиляции* является значение, которое компилятор может оценить при компиляции программы. Числовые и символьные литералы, а `const` также переменные совместимых числовых или символьных типов являются репрезентативными примерами. Обычные переменные, свойства, вызовы методов и `static readonly` поля не являются константами во время компиляции. В первом примере оба `>= 18` и `<= 24` проверяют одно и то же вычисленное значение `temperature`.

Тот же реляционный символ может отображаться в обычном выражении или в шаблоне. В следующем примере используются обе формы с температурой:

C#

```
static void ShowExpressionAndPattern(int temperature, int threshold)
{
    bool belowThreshold = temperature < threshold;
    bool belowFreezing = temperature is < 0;

    string description = temperature switch
    {
        < 0 => "Freezing",
        0 => "Freezing point",
        > 0 => "Above freezing"
    };

    Console.WriteLine(
        $"Below threshold: {belowThreshold}; " +
        $"below freezing: {belowFreezing}; {description}");
}
```

`temperature < threshold` — *реляционное выражение*. Вычисляет оба операнда и возвращает `bool`. Любой из операндов может быть неконстантным выражением.

В `temperature is < 0` `temperature` является входным выражением шаблона. C# вычисляет его, а реляционный шаблон `< 0` проверяет полученное значение. В ветви switch `< 0 => "Freezing"` выражение перед `switch` задаёт входное значение, поэтому образец содержит только `< 0`.

Выражение может сравниться `temperature` с переменной `threshold`. Операнд реляционного шаблона должен быть константой времени компиляции, поэтому используйте реляционное выражение, если значение сравнения является переменной. Если значение сравнения является константой, любая форма может работать.

Если правый операнд — константа, следует руководствоваться главным образом удобочитаемостью. Реляционное выражение часто соответствует одному прямому сравнению. Реляционный шаблон состоит из других шаблонов и подходит естественно, когда несколько диапазонов сопоставляются с переключением результатов.

Объединяйте условия с помощью логических шаблонов

Логические шаблоны объединяют или отрицают шаблоны с помощью операторов шаблонов `and`, `or` и `not`:

- Шаблон `and` совпадает при совпадении обоих вложенных шаблонов.
- Шаблон `or` совпадает, если совпадает любой из вложенных шаблонов.
- Шаблон `not` успешно срабатывает, когда его вложенный шаблон не срабатывает.

В открывшемся примере используется `and` для описания диапазона и `or` описания альтернативных вариантов. Шаблон `not` может исключить значение, как и в `status is not Status.Complete`. Следующие методы показывают обе формы, чтобы узнать их синтаксис и сравнить, как они выражают один и тот же тест:

C#

```
static bool IsWeekendPattern(DayOfWeek day) =>
    day is DayOfWeek.Saturday or DayOfWeek.Sunday;

static bool IsWeekendImperative(DayOfWeek day) =>
    day == DayOfWeek.Saturday || day == DayOfWeek.Sunday;
```

Шаблонная форма использует два константных шаблона, соединённых с помощью `or`. Форма императива использует два выражения равенства, соединённые оператором условного ИЛИ `||`. Обе формы являются краткими и понятными для этого одного условия. Выберите форму, которая лучше всего подходит для окружающего кода. Шаблоны часто уточняют несколько связанных вариантов в примере `switch`, как в открываемом примере.

Операторы шаблонов формируют шаблоны, а не логические выражения: `and` соответствуют сочетанию шаблонов, `or` альтернативным шаблонам и `not` отрицанию шаблона. В логических выражениях используются `&&`, `||` и `!`. Выберите `or` , когда несколько альтернативных шаблонов имеют одинаковый результат. Выбор `not` при выражении исключенного шаблона является более понятным, чем перечисление каждого принятого значения.

Групповые шаблоны с скобками

Шаблон в скобках использует скобки, чтобы показать или изменить способ группировки вложенных шаблонов. *Связывание* определяет, какие операнды шаблона оператор группирует вместе, аналогично неявной группировке, когда скобки не ставятся. C# указывает следующий порядок привязки:

1. `not`
2. `and`
3. `or`

Следующий тест принимает приоритеты от 1 до 3 или специального приоритета 9:

C#

```
static bool IsAcceptedPriority(int priority) =>
    priority is (>= 1 and <= 3) or 9;
```

Компилятор группирует `and` перед `or`. Скобки явно показывают подразумеваемую группировку и позволяют легко увидеть два варианта: диапазон от 1 до 3 или число 9. Для удобочитаемости используйте круглые скобки всякий раз, когда в шаблоне сочетаются `and` и `or`, или когда `not` применяется к составному шаблону. Круглые скобки также могут изменять группирование по умолчанию, как и в `not (>= 1 and <= 3)`.

Порядок проверок во время выполнения для вложенных шаблонов не определён, а операторы шаблонов подчиняются правилам сопоставления с шаблоном, а не правилам сокращённого вычисления булевых выражений. Пишите вложенные шаблоны так, чтобы их результат не зависел от порядка проверок.

Использование охранника `when` для отдельного условия

Логические шаблоны лучше всего работают, если вложенные шаблоны описывают само входное значение. Сторожевое условие `when` — это дополнительное логическое условие для метки `case` или ветви switch. Используйте охранник, если решение также зависит от информации, отдельной от входных данных шаблона.

Следующее предупреждение зависит от температуры и отдельного `isOutdoors` значения:

C#

```
static string GetHeatWarning(int temperature, bool isOutdoors) =>
    temperature switch
    {
        > 35 when isOutdoors => "High heat outdoors",
        > 35 => "High heat",
        _ => "No heat warning"
    };
```

Реляционный шаблон `> 35` описывает `temperature` входные данные. Защитное условие `when isOutdoors` проверяет отдельное значение. Защитное условие также предпочтительнее, если для условия нужен вызов метода или логическое выражение, которое невозможно ясно выразить с помощью синтаксиса шаблонов.

Используйте реляционные и логические паттерны, когда они помогают нагляднее увидеть допустимые формы или значения входных данных, особенно в нескольких ветвях switch. Используйте обычное логическое выражение, когда оно проще задаёт непосредственное условие. Используйте защитное условие `when`, если выбор в switch зависит от отдельного значения или от условия, которое лучше выразить логическим выражением.

См. также

- Обзор сопоставления шаблонов
- Свойства и позиционные шаблоны
- Операторы C#
- Справочник по реляционным шаблонам
- Справочник по логическим шаблонам
- Ссылка на шаблон с скобками

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 25.09.2026

Шаблоны списков и срезов

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Начните с **обзора сопоставления шаблонов** , если вы не знакомы с шаблонами. Полные правила для поддерживаемых типов и языков см. в разделе **шаблоны списков** справочника по языку.

Фигура последовательности — это набор наблюдаемых свойств, необходимых шаблону. Эти свойства могут включать количество элементов, требования, записанные для каждого элемента или последовательности элементов в целом, и значения маркеров на определенных индексах. Например, форма может требовать ровно два элемента с элементом `"Name"` в позиции 0, либо первый маркер, последний маркер и непустую последовательность между ними.

Шаблон списка описывает фигуру, сочетая требование счетчика элементов с вложенными шаблонами на упорядоченных позициях. Он проверяет только требования, указанные в шаблоне: подпаттерн элемента проверяет соответствующий ему элемент, а *шаблон среза* может допускать или проверять последовательность элементов, которые в противном случае не были бы сопоставлены. Эти части можно использовать для сопоставления точных значений счетчика и маркеров, записи значений на фиксированных позициях, разрешить несовпадную середину или применить вложенный шаблон к этой середине.

Выберите шаблон списка, если тип входных данных во время компиляции предоставляет свойство `Length` или `Count` и индексированный доступ к элементам, который позволяет получать элемент по его позиции, как в `input[index]`. Массивы, `List<T>`, строки и спаны — распространённые примеры. Компилятор определяет соответствие требованиям на основе объявленного типа переменной, известного на этапе компиляции, поскольку он должен разрешить эти обязательные члены до запуска программы. Сам IEnumerable<T> интерфейс предоставляет перечисление вместо этих элементов, даже если объект среды выполнения является индексируемой коллекцией.

Шаблоны списков используют длину или количество элементов и доступ к элементам по индексу вместо перечисления. Для распространенных встроенных типов сопоставление может проверять только позиции, именованные шаблоном, а не выполнять итерацию по

каждому элементу. Стоимость этих операций зависит от типа входных данных. Итерацию длинной коллекции может занять некоторое время, поэтому используйте цикл или LINQ, если решение требует проверки произвольного количества элементов, а не конкретных позиций.

Сопоставить точную форму

Следующий метод распознает заголовок двух столбцов:

C#

```
static bool IsHeader(string[] columns) =>
    columns is ["Name", "Score"];
```

Выражение `columns` — это входные данные шаблона. `["Name", "Score"]` содержит два константных шаблона. Точный шаблон списка исключает шаблон среза, требует ровно двух элементов и применяет каждый вложенный шаблон к элементу в одном индексе. Этот шаблон описывает форму с точным количеством и двумя значениями-маркерами: `"Name"` с индексом 0 и `"Score"` с индексом 1. Вложенные шаблоны элементов являются необязательными: шаблон `[]` пустого списка описывает и соответствует последовательности без элементов.

Выберите шаблон списка, если шаблон сочетает количество элементов с требованиями к элементам в определённых позициях. Если имеет значение только количество элементов, шаблон свойств `Count` или `Length`, например `items is { Count: 0 }`, более прямо выражает это намерение.

Использовать отбрасываемое значение для сопоставления с любым значением элемента

Следующий метод считывает победителя и участника, финишировавшего третьим, из списка из трёх имён в порядке финиша:

C#

```
static string GetAnnouncements(List<string> finishingOrder) =>
    finishingOrder switch
    {
        [var winner, _, var thirdPlace] =>
            $"Winner: {winner}; third place: {thirdPlace}",
```

```
        _ => "A complete three-runner result isn't available"
    };
```

Шаблоны `var winner` и `var thirdPlace` каждый объявляют переменную. При совпадении шаблона списка C# назначает соответствующее значение элемента в этой позиции соответствующей переменной, поэтому результат может использовать `winner` и `thirdPlace`. Шаблон отбрасывания `_` принимает второй элемент и игнорирует его значение. Для этой структуры требуется ровно три элемента, причём первой и третьей позициям придаётся особое значение, но конкретные значения ни в одной позиции не требуются.

Выберите эту форму, если фиксированные позиции имеют стабильное значение. Используйте цикл или LINQ, если необходимо проверить произвольное количество элементов, преобразовать последовательность, выполнить поиск по нему или выполнить агрегирование.

Разрешить оставшиеся элементы с шаблоном среза

В этом упрощённом примере командная строка может начинаться с `--verbose`, содержать другие аргументы и, как предполагается, заканчиваться именем входного файла. Следующий метод распознает фигуры с именем файла в последней позиции и записывает это значение:

C#

```
static string GetInputFile(string[] arguments) =>
    arguments switch
    {
        ["--verbose", .., var fileName] => $"Verbose processing: {fileName}",
        [.., var fileName] => $"Processing: {fileName}",
        [] => "No input file was provided"
    };
```

`..` допускает наличие нуля или более оставшихся элементов между шаблонами первого и последнего элементов. Шаблон списка может содержать не более одного шаблона среза. Первая переключительная рука описывает фигуру с по крайней мере двумя элементами: маркером `"--verbose"` по индексу 0, именем файла в последней позиции и любым количеством несовпаденных аргументов между ними. Автономный срез не проверяет эти средние аргументы. Вторая ветвь принимает любой непустой шаблон и связывает его с последним элементом, а пустой шаблон обрабатывает последовательность, не содержащую элементов.

Срез может отображаться в начале, середине или конце шаблона списка. Используйте его, если последовательность переменной длины является частью фигуры, но только окружающие позиции нуждаются в тестах элементов. Используйте обычную итерацию, когда нужно обработать каждый элемент.

Применение шаблона к срезу

Вы можете применить другой шаблон к части, соответствующей `..`. Следующий метод проверяет, начинается ли массив с `"BEGIN"`, заканчивается `"END"` и имеет хотя бы один элемент между ними:

C#

```
static bool HasContent(string[] entries) =>
    entries is ["BEGIN", .. { Length: > 0 }, "END"];
```

Внешний шаблон сначала требует наличия маркера `"BEGIN"` в позиции с индексом 0 и маркера `"END"` в последней позиции. Затем шаблон свойства `{ Length: > 0 }` добавляет требование к последовательности, сопоставленной со срезом: последовательность должна содержать как минимум один элемент. Вместе эти требования описывают полную форму. Шаблон не проверяет значения элементов внутри среза.

Используйте подшаблон среза только тогда, когда сама средняя часть требует проверки или захвата. Если только элементы границы имеют значение, обычный `..` проще.

См. также

- Обзор сопоставления шаблонов
- Свойства и позиционные шаблоны
- Справочник по шаблону списка
- Массивы
- Используйте инструкцию foreach для перебора элементов коллекции

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 25.09.2026

Отмена и шаблон отмены

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с шаблонами, начните с обзора сопоставления шаблонов. Полный синтаксис см. в справочнике по шаблону discard.

Маркер подчеркивания (`_`) сообщает, что значение не требуется. Его точное значение зависит от того, где он отображается:

Развернуть таблицу

Context	Значение `_`
Ветвь switch-выражения или вложенный шаблон	*Шаблон отбрасывания*, который сопоставляется без сохранения результата
Деконструкция или `out` аргумент	*Отмена*, которая игнорирует одно создаваемое значение
Назначение, например `_ = expression`	*Присваивание с отбрасыванием результата*, которое вычисляет выражение и игнорирует его результат
Два или более лямбда-параметров с именем `_`	*Отмена параметров* , входные данные которых не используются
`var _` в шаблоне	`var` Шаблон с обозначением отбрасывания

Эти формы используют орфографию и намерение, но они не взаимозаменяемы.

Сопоставление шаблонов с `switch`

В следующем примере `statusCode` — это `int`. Каждый рычаг переключателя генерирует сообщение `string`, которое программа выводит в консоль. Окончательный `_` обрабатывает каждый код состояния, отличный от `200` и `404`:

C#

```
static void ShowStatus()
{
```

```
    int statusCode = 503;
    string message = statusCode switch
    {
        200 => "Ready",
        404 => "Not found",
        _ => "Another status"
    };

    Console.WriteLine(message);
}
```

Шаблон отбрасывания применяется к входному выражению. C# вычисляет выражение, и `_` соответствует вычисленному значению, не захватывая его. Выберите `_` в качестве последней ветви switch-выражения, если для всех значений, не обработанных ранее, должен использоваться один и тот же вариант по умолчанию. Поместите его последним, поскольку оно совпадает со всем, включая `null`.

Форма `var _` является шаблоном `var` с обозначением отбрасывания. Он также соответствует каждому вычисляемому значению, но он не вводит удобочитаемую переменную. Предпочитайте более короткий `_` шаблон отбрасывания для случая switch, охватывающего все остальные значения. Дополнительные сведения о шаблонах и обозначениях см. в `var` разделе "Объявление", "Константы" и var "Шаблоны".

Объявления о деконструкции

`GetForecast` возвращает кортеж из четырёх компонентов: город `string` и три значения `int` для максимальной температуры, минимальной температуры и вероятности дождя. Объявление деконструкции сохраняет `city` и `high` потому, что программа отображает их. Для низкой температуры и вероятности дождя используется `_`, так как если дать имена этим неиспользуемым компонентам, это будет означать, что они нужны коду:

C#

```
static void ShowForecast()
{
    var (city, high, _, _) = GetForecast();
    Console.WriteLine($"{city}: high {high}°C");

    static (string City, int High, int Low, int RainChance) GetForecast() =>
        ("Portland", 18, 9, 40);
}
```

Тот же синтаксис отбрасывания работает, если метод объекта `Deconstruct` возвращает несколько значений. Сведения о них см. в разделе «Деконструкция кортежей и других

типов».

Вызовы методов с `out` параметрами

Предположим, поле ввода принимает текст только в том случае, если он представляет целое число. Метод TryParse(String, Int32) возвращает значение `bool` , указывающее, выполнен ли синтаксический анализ. Он также возвращает разобранный `int` через параметр `out`. Следующему коду требуется только логический результат, чтобы принять или отклонить ввод:

C#

```
static void CheckInput()
{
    string text = "42";

    if (IsWholeNumber(text))
    {
        Console.WriteLine($"Accepted: {text}");
    }
    else
    {
        Console.WriteLine("Enter a whole number.");
    }

    static bool IsWholeNumber(string text) => int.TryParse(text, out _);
}
```

Используйте `out _` только в случае успеха операции. Отбрасывание явно показывает, что разобранное число не нужно. Если позже коду нужен номер, присвойте `out` аргументу имя, например `out int number`, и сохраните это значение.

Присваивание с отбрасыванием, `_ = expression`, вычисляет выражение и намеренно игнорирует его результат. Иногда это бывает полезно, когда выражение в противном случае не является допустимой инструкцией.

> ⓘ **Важно!**
>
> Не используйте `_ = Task.Run(...)` или `_ = SomeAsyncMethod()`, чтобы отбрасывать задачу в коде приложения. Дождите задачу, чтобы ее завершение и исключения оставались в потоке вызовов. Присваивание в discard не делает задачу безопасной, не отслеживает ее исключение и не создает поддерживаемую операцию «запустить и забыть».

Пометка неиспользуемых лямбда-параметров

`object?` Получает EventHandler отправителя и EventArgs значение. Для следующего обработчика не нужен ни один из этих параметров; он лишь выводит `"Timer tick"` в консоль. Именование обоих параметров `_` делает их неиспользуемое состояние видимым без изобретания имен, которые тело никогда не использует:

C#

```
static void ShowLambdaDiscards()
{
    EventHandler handler = (_, _) => Console.WriteLine("Timer tick");
    handler(null, EventArgs.Empty);
}
```

Выберите отбрасываемые параметры, если сигнатура делегата требует входные параметры, которые не используются в теле лямбда-выражения. Если лямбда-файл имеет только один параметр с именем `_`, `_` остается обычным именем параметра для обратной совместимости.

Избегайте `_` в качестве идентификатора

`_` может быть обычным идентификатором в тех контекстах, где C# не распознает его как отбрасываемое значение. Переменной, находящейся в области видимости и имеющей имя `_`, можно присвоить значение выражением, которое выглядит как присваивание отбрасыванию. В контексте сопоставления с образцом доступная константа или тип с именем `_` также может изменить интерпретацию `_`. Избегайте объявления собственных переменных, констант или типов с именем `_`; используйте `_`, чтобы указать, что значение намеренно отбрасывается.

См. также

- Обзор сопоставления шаблонов
- Объявление, константы и var шаблоны
- Деконструкция кортежей и других типов
- Параметры лямбда-выражения
- Удалить ссылку на шаблон

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 18.09.2026

Выражения C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий **по началу работы**.
>
> **Вы пришли из другого языка?** Выражения в C# работают так же, как и в Java, C++и JavaScript. Стоит отметить одно различие: операторы составного присваивания, такие как `+=`, и оператор инкремента `++` в C# являются выражениями, поэтому они могут использоваться как часть более крупных выражений.

Выражение — это фрагмент кода, который компилятор вычисляет для создания значения — числа, строки, ссылки или а`bool`. Например, `3 + 4 * 2` — это выражение, которое имеет значение целого числа `11`, а `total > 10` — это выражение, которое имеет значение `true` или `false`.

Самыми простыми выражениями являются *литералы* (например`42`, или`"hello"`*) и имена переменных* (например`total`). Вы создаете более сложные выражения, сочетая более простые с операторами.

Выражения и инструкции

Выражение создает значение. *Инструкция* представляет собой полную инструкцию, выполняемую программой. Многие инструкции содержат выражения. Например, `int total = 3 + 4 * 2;` — это инструкция объявления переменной. Компилятор вычисляет выражение `3 + 4 * 2`инициализатора, которое создает `11`и присваивает это значение новой переменной `total`. В этой статье рассматриваются выражения— как они формируются, как они оцениваются и как они объединяются.

Объединение выражений

Выражения можно объединить. Рассмотрим `3 + 4 * 2`. Это одно выражение фактически содержит два небольших выражения: выражение `4 * 2`*умножения* и *сложение*`3 + <result>`. При объединении выражений C# требуется правило, чтобы решить, какой из них необходимо сначала оценить. Это правило является *приоритетом оператора*.

Вам не нужно запоминать полную иерархию приоритета. В следующем разделе кратко описаны обычные группы, а скобки всегда позволяют явно задать порядок, если вы не уверены.

Приоритет оператора

Приоритет операторов определяет, как составное выражение *группируется* в подвыражения. Эта концепция аналогична правилам порядка операций из математического класса. Оператор с более высоким приоритетом *привязывается более тесно* к операндам. Выражение построено так, как будто эти операнды заключены в скобки вместе. В `3 + 4 * 2` `*` имеет более высокий приоритет, чем `+`, поэтому выражение группируется как `3 + (4 * 2)`, а не `(3 + 4) * 2`.

C# имеет более приоритетные группы, чем четыре, приведенные здесь. Эти дополнительные группы применяют знакомые правила (например, умножение перед добавлением) и охватывают полный набор операторов. Следующие группы охватывают то, с чем вы чаще всего сталкиваетесь в повседневной работе с кодом:

1. **Первичные, унарные и диапазоны** — это три отдельные группы приоритета, все из которых привязываются более тесно, чем арифметические. Это резюме объединяет их в один этап, поскольку на практике все они применяются до выполнения арифметических операций.

    - *Первичные* операторы — доступ к члену (`x.y`), вызовы методов (`f()`), индексирование (`a[i]`) и условный доступ с проверкой на null (`?.`, `?[]`) — имеют наивысший приоритет.
    - *Унарные* операторы применяются к одному операнду: отрицание (`-x`), логическое НЕ (`!flag`), префиксный инкремент (`++i`) и постфиксный инкремент (`i++`).
    - Оператор *диапазона* (`..`) создает диапазоны индексов для выражений среза, например `array[1..4]`.

2. **Арифметика** — `*`, `/`, `%` связываются сильнее, чем `+` и `-`. Умножение и деление происходят перед добавлением и вычитанием.
3. **Сравнение** — `<`, `>`, `<=`, `>=`, `==`, `!=` имеют более низкий приоритет, чем арифметические операции, поэтому арифметические операции выполняются до сравнения.
4. **Логические операторы** — `&&` и `||` имеют самый низкий приоритет среди распространённых операторов, поэтому сравнения выполняются до логического

объединения.

Этот приоритет означает, что выражение `score + bonus > threshold && attempts < maxAttempts` вычисляется именно так, как это читается: сложить `score` и `bonus`, сравнить сумму с `attempts`, сравнить `maxAttempts` с `threshold`, а затем объединить два результата `bool` с помощью `&&`. Добавление скобок, чтобы явно обозначить каждую группировку, показывает ту же структуру: `((score + bonus) > threshold) && (attempts < maxAttempts)`.

Полные иерархии приоритета, охватывающие каждый оператор, см. в разделе операторов и выражений C#.

Используйте скобки, чтобы сделать смысл понятнее

Скобки переопределяют приоритет операций и одновременно поясняют ваше намерение. Если порядок не очевиден из указанных выше групп, добавьте скобки:

C#

```
int score = 80;
int bonus = 15;
int threshold = 90;
bool eligible = true;

// Without parentheses: || binds less tightly than &&, so this reads as:
//   eligible && (score + bonus > threshold) || bonus > 20
bool result1 = eligible && score + bonus > threshold || bonus > 20;

// With parentheses: forces the || to combine two complete conditions
bool result2 = (eligible && score + bonus > threshold) || bonus > 20;

Console.WriteLine(result1); // => True
Console.WriteLine(result2); // => True

// Parentheses can also change the result:
bool isAdmin = false;
bool isOwner = true;

// Without: && binds tighter, so: (isAdmin && isOwner) || true
bool access1 = isAdmin && isOwner || true;
// With: forces the || to run first
bool access2 = isAdmin && (isOwner || true);

Console.WriteLine(access1); // => True   (false && true is false; false || true is
true)
Console.WriteLine(access2); // => False  (false && anything is false)
```

Последние две строки показывают, что скобки могут изменить результат, а не только стиль. Когда `&&` и `||` появится вместе, добавьте скобки, чтобы определить, какое условие

объединяет сначала. Читатель, который видит `(isAdmin && isOwner) || isSuperUser`, сразу понимает замысел.

Как вычисляются выражения

Понять, как C# вычисляет составные выражения, будет проще, если использовать аналогию. Подумайте о работе с сложной математической проблемой с карандашом и бумагой: вы определяете самую внутреннюю или самую высокую приоритетность под-выражения, вычисляете его промежуточное значение, записываете результат, а затем повторяете с помощью следующего под-выражения — продолжая до тех пор, пока не достигнете окончательного ответа.

C# следует одному и тому же процессу, руководствуясь двумя правилами:

Правило 1. Операнды оценивают слева направо. Для любого двоичного выражения оба операнда должны быть полностью оценены перед применением оператора. C# сначала вычисляет левый операнд, затем правый, а затем выполняет операцию.

Правило 2: Некоторые операторы используют сокращённое вычисление. Некоторые операторы перестают оценивать, как только результат определен, пропуская все оставшиеся операнды:

- `&&` (условное И): возвращает `false`, как только значение левой части равно `false`. Правая сторона никогда не вычисляется.
- `||` (условное ИЛИ): возвращает `true`, как только левая часть равна `true`. Правая сторона никогда не вычисляется.
- `?:` (условно/ternary): вычисляет только ветвь, соответствующую условию, — другая ветвь никогда не вычисляется.
- `?.` (null-условный доступ к члену) и `?[]` (null-условный доступ к элементу): немедленно прекращают выполнение и возвращают `null`, если левая часть равна `null`, пропуская доступ к члену или обращение по индексу.
- `??=`(назначение с объединением null): назначает правую сторону только в том случае, если левая сторона .`null`

Оценка бумаги и карандаша

Рассмотрим выражение `3 + 6 / 2`. Несмотря на то что в порядке чтения первым стоит сложение, `/` имеет более высокий приоритет, чем `+`, поэтому именно `6 / 2` является

подвыражением, которое вычисляется первым. Решая это шаг за шагом — точно так же, как на бумаге:

```
3 + 6 / 2
      ↓   (evaluate 6 / 2 → 3)
3 +   3
  ↓   (evaluate 3 + 3 → 6)
  6
```

C#

```
// 3 + 6 / 2
// Step 1: 6 / 2 has higher precedence → interim value 3
// Step 2: 3 + 3 → final result 6
int result = 3 + 6 / 2;
Console.WriteLine(result); // => 6
```

Промежуточное значение `3` , созданное путем `6 / 2` , становится правым операндом `+`, а окончательный результат — `6`. Каждое под-выражение создает промежуточное значение; эти промежуточные значения питают следующее под-выражение, пока не останется только одно значение.

Ассоциативность — это связанная концепция: когда два оператора имеют одинаковый приоритет, ассоциативность решает, какой из них идет первым. Большинство операторов C# являются *левоассоциативными*, то есть группируются слева направо. Так `a - b - c` же, как `(a - b) - c`, а не `a - (b - c)`.

Оценка короткого канала на практике

Сокращённое вычисление особенно полезно при проверке на null:

C#

```
string? text = null;

// Safe: second condition runs only when text is not null
bool hasContent = text != null && text.Length > 0;
Console.WriteLine(hasContent); // => False  (short-circuits after null check; no
NullReferenceException)

text = "hello";
hasContent = text != null && text.Length > 0;
Console.WriteLine(hasContent); // => True
```

```
// || short-circuits on true: right side is never evaluated when left side is true
string word = "hello";
bool anyMatch = word.StartsWith("h") || word.StartsWith("x");
Console.WriteLine(anyMatch); // => True  (right side never evaluated)

// ?. short-circuits on null: returns null without accessing .Length
string? maybeNull = null;
int? length = maybeNull?.Length;   // length is null; no NullReferenceException
Console.WriteLine(length.HasValue); // => False
```

`text != null && text.Length > 0` безопасен, так как второе условие выполняется только если `text` — не `null`. Аналогичным образом `?.` останавливает вычисление, когда встречает ссылку `null`, что позволяет избежать `NullReferenceException` без явной проверки `if`.

Более широкий взгляд на операторы, безопасные с значением NULL, см. в разделе "Операторы NULL C#".

См. также

- Операторы и выражения C# (справочник по языку) — полная таблица приоритета и каждый оператор
- Сравнение на равенство — как работают `==`, `!=` и `Equals`
- Операторы C# NULL — `?.`, `??` и `??=`
- Булевы логические операторы

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 19.08.2026

Сравнение на равенство в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** В Java `==` для объектов и в JavaScript `===` для объектов проверяют идентичность, а не содержимое. Классы C# работают одинаково по умолчанию. В Python `==` вызывает `__eq__` и по умолчанию сравнивает содержимое, аналогично тому, как сравниваются records в C#. Структуры C# также сравниваются по значению при вызове `Equals`.

C# отличает два типа равенства. *Равенство значений* означает, что два экземпляра равны при совпадении данных. *Равенство ссылок* означает, что две переменные равны только в том случае, если они указывают на один и тот же объект в памяти. Это условие также называется *тождеством*. Типы значений обычно сравнивают по данным, а ссылочные типы — по идентичности. Авторы типов могут изменить эти значения по умолчанию, но эта умственная модель предотвращает тонкие ошибки, в которых два объекта, которые выглядят идентичными, не считаются равными, или где мутация через одну переменную автоматически изменяет то, что видит другая переменная.

Типы значений, ссылочные типы и поведение равенства по умолчанию

Каждый тип в C# — это *тип значения* или *ссылочный тип*. *Тип значения* содержит данные непосредственно в переменной. *Ссылочный тип* содержит ссылку на объект. При назначении переменной ссылочного типа другой переменной обе переменные ссылаются на один и тот же объект. Дополнительные сведения о типах значений и ссылочных типах см. в обзоре системы типов.

Поведение равенства по умолчанию обычно соответствует типу:

- **Встроенные числовые типы и перечисления** — это типы значений. Две `int` переменные равны, если их числовые значения совпадают.

- Структуры — это типы **значений**. Обычный `struct` использует равенство значений при вызове Equals.
- **Кортежи** — это типы значений. Два кортежа равны, если все значения элементов совпадают.
- **Классы** являются ссылочными типами. Обычный класс использует равенство ссылок, поэтому `==`Equals и проверяет, указывают ли две переменные на один и тот же объект.

Класс использует равенство ссылок. Два отдельных объекта с одинаковыми данными не равны, но две переменные, ссылающиеся на один и тот же объект, равны:

C#

```
var order1 = new Order(42, "Shoes");
var order2 = new Order(42, "Shoes");

Console.WriteLine(order1 == order2);                // => False
Console.WriteLine(order1.Equals(order2));           // => False
Console.WriteLine(ReferenceEquals(order1, order2)); // => False

Order order3 = order1;
Console.WriteLine(order1 == order3);                // => True
```

`struct` показывает равенство значений с помощью Equals. Два экземпляра структуры равны при совпадении полей:

C#

```
var pt1 = new Point(3, 4);
var pt2 = new Point(3, 4);

Console.WriteLine(pt1.Equals(pt2)); // => True
```

Простые структуры не получают предопределенного `==` оператора. Использование `p1 == p2` для обычной структуры компилируется, только если структура объявляет собственный `operator ==`. Если вам нужны сравнения операторов для структуры, определите `==` и `!=` как пару и сохраните их в соответствии с Equals и GetHashCode.

Кортежи также являются типами значений. Два кортежа равны, если каждое значение элемента совпадает. Имена элементов в именованном кортеже нужны лишь для удобства на этапе компиляции и не учитываются при сравнении. Важны только позиции и значения:

C#

```
var t1 = (Name: "Grace", Role: "Engineer");
var t2 = (Name: "Grace", Role: "Engineer");

Console.WriteLine(t1 == t2); // => True
```

Дополнительные сведения о синтаксисе кортежей и деконструкции см. в разделе Кортежи и деконструкция.

Используйте `Object.ReferenceEquals`, чтобы напрямую проверить личность

ReferenceEquals всегда проверяет тождественность независимо от того, как тип переопределяет Equals или перегружает `==`. Используйте это для проверки идентичности, когда нужно подтвердить, указывают ли две переменные на один и тот же объект:

C#

```
var doc1 = new Document("Report");
var doc2 = new Document("Report");
var doc3 = doc1;

Console.WriteLine(ReferenceEquals(doc1, doc2)); // => False
Console.WriteLine(ReferenceEquals(doc1, doc3)); // => True
```

Часто это используется в переопределении `Equals`, чтобы избежать полного сравнения: если оба аргумента ссылаются на один и тот же объект, они всегда равны без проверки отдельных полей.

> ⓘ **Примечание.**
>
> Когда переменные имеют тип **интерфейс**, `==` проверяет, ссылаются ли переменные интерфейсного типа на один и тот же объект. Вызов `Equals` по-прежнему выполняет реализацию базового объекта.

> ⓘ **Примечание.**
>
> ReferenceEquals всегда возвращается `false` при сравнении типов значений, даже если оба аргумента содержат одинаковые значения. Это поведение возникает, потому что каждый аргумент типа значения независимо *упаковывается* в отдельный объект в куче при передаче в `ReferenceEquals`.

Типы могут определять различные семантики равенства

Типы *могут* определять семантику равенства, отличающуюся от поведения по умолчанию. Наиболее распространенной причиной является реализация равенства значений. Если вы создаете тип, представляющий данные, такие как банковский счет, продукт в инвентаризации или пользователь в системе, рассмотрите экземпляры с теми же значениями, что и равные. *Выберите типы записей для реализации равенства значений*, а компилятор создает все необходимые элементы равенства.

> ⓘ **Примечание.**
>
> **Строки** — это классы, но `==` и Equals сравнивают содержимое строк, а не их идентичность.

- `==`: оператор равенства. Большинство типов используют этот оператор в качестве основной проверки равенства. Его поведение зависит от того, имеет ли тип встроенный или определяемый `==` пользователем оператор.
- `!=`: оператор неравенства. Если тип определяет определяемый `==` пользователем оператор, он также должен определять `!=`.
- Equals: виртуальный метод, который наследуется каждым типом. Его можно переопределить, чтобы изменить семантику равенства для типа.
- GetHashCode: виртуальный метод, используемый хэш-коллекциями. Если два значения равны, их хэш-коды также должны быть равными.
- ReferenceEquals: статический метод, который всегда проверяет идентичность.

Использование записей для равенства значений

Используйте модификатор `record`, чтобы обеспечить равенство значений для типа, ориентированного на данные, если тип может быть записью. Компилятор генерирует члены Equals, GetHashCode и `==`/`!=`, которые сравнивают значения всех объявленных свойств.

`record class` по-прежнему является ссылочным типом, но сравнивает значения, а не идентичность:

C#

```
var person1 = new Person("Ada", "Lovelace");
var person2 = new Person("Ada", "Lovelace");
```

```
Console.WriteLine(person1 == person2);                // => True
Console.WriteLine(person1.Equals(person2));           // => True
Console.WriteLine(ReferenceEquals(person1, person2)); // => False
```

ReferenceEquals подтверждает, что `person1` и `person2` — это разные объекты в памяти, тогда как `==` и Equals возвращают `True`, поскольку сгенерированная компилятором операция проверки равенства сравнивает значения свойств.

То же поколение компилятора применяется к `record struct` типам:

C#

```
var dim1 = new Dimension(1920, 1080);
var dim2 = new Dimension(1920, 1080);

Console.WriteLine(dim1 == dim2);      // => True
Console.WriteLine(dim1.Equals(dim2)); // => True
```

Типы записей создают весь набор равенства для собственного типа. Типы `record class` и `record struct` переопределяют Equals и GetHashCode. Они также создают `==` и `!=` операторы, а также типизированный `Equals` метод для типа записи. В отличие от обычного `struct`, поэтому `record struct` поддерживается `==` и `!=` автоматически. Дополнительные сведения о типах записей и их семантике равенства см. в разделе "Записи".

Записи с элементами ссылочного типа

Для равенства записей используется собственная семантика равенства членов. Каждое свойство или поле сравнивается с помощью собственного `Equals` метода. Для большинства скалярных значений, таких как `int` `string`, или `DateTime`, этот подход сравнивает значения элементов записи. Тонкость возникает с обычными изменяемыми коллекциями, такими как `List<T>` или `T[]`: эти типы сравниваются по ссылке, поэтому два экземпляра record-типа, содержащие *разные объекты списков с одинаковым содержимым*, **не** считаются равными при автоматически сгенерированном сравнении на равенство для record-типов.

C#

```
var playlist1 = new Playlist("Chill", new List<string> { "Song A", "Song B" });
var playlist2 = new Playlist("Chill", new List<string> { "Song A", "Song B" });

Console.WriteLine(playlist1.Equals(playlist2));                        // => False
```

```
(different List instances)
Console.WriteLine(playlist1.Tracks.SequenceEqual(playlist2.Tracks));    // => True
```

`playlist1` и `playlist2` являются отдельными `List<string>` экземплярами. Хотя их содержимое совпадает, `Equals` возвращает `false`.

Если вам нужно, чтобы равенство записей учитывало *содержимое* коллекции, у вас есть несколько вариантов:

- Реализуйте **`IEquatable<T>`** в записи и переопределите `Equals`, чтобы использовать Enumerable.SequenceEqual для элементов коллекции.
- **Используйте тип коллекции с равенством значений** , например настраиваемый `IEqualityComparer<T>` или тип, собственный `Equals` который сравнивает элементы.
- **Проектирование с учётом идентичности**: если запись представляет сущность, а не чистое значение, то ссылочное равенство для элементов её коллекции может быть преднамеренным.

> ⓘ **Важно!**
>
> Реализация равенства вручную редко используется сегодня в C#. Записи обрабатывают общий сценарий равенства значений автоматически. Если вам нужно вручную реализовать равенство — например, потому что ваш тип должен наследоваться от базового класса, не являющегося записью, — см. раздел **Самостоятельная реализация равенства, если тип не может быть записью** в справочнике языка.

См. также

- Общие сведения о системе типов
- Классы
- Структуры
- Записи
- Кортежи и деконструкция.
- Операторы равенства (справочник по языку).
- Равенство в иерархиях классов — расширенное руководство по полиморфическому равенству.
- Арифметические операторы, операторы сравнения, логические операторы и операторы присваивания — обзор оператора равенства наряду с арифметическими, логическими операторами и операторами присваивания.

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.08.2026

Операторы C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** Большинство операторов в этой статье (`+`, `*` `/` `&&` `||` `!` `%` `-` `!=` `<` `==` `>`операторы сравнения и `=`) работают так же, как и в Java, C++и JavaScript. Основные сюрпризы для новичков — это поведение целочисленного деления, различие между префиксной и постфиксной формами для `++`/`--`, а также то, как составное присваивание приводит значение обратно к типу левой части выражения.

Оператор объединяет один или несколько *операндов* в одно значение. Вы уже знаете о приоритете выражений и операторов из выражений C#; В этой статье подробно рассматриваются конкретные операторы, которые вы будете использовать каждый день.

Арифметические операторы

Пять арифметических операторов выполняют числовые вычисления.

Развернуть таблицу

Operator	Name	Example	Результат
`+`	Добавление	`10 + 3`	`13`
`-`	Вычитание	`10 - 3`	`7`
`*`	Умножение	`10 * 3`	`30`
`/`	Подразделение	`10 / 3`	`3`
`%`	Остаток	`10 % 3`	`1`

C#

```
int apples = 10;
int oranges = 3;

Console.WriteLine(apples + oranges);  // => 13  (addition)
Console.WriteLine(apples - oranges);  // => 7   (subtraction)
Console.WriteLine(apples * oranges);  // => 30  (multiplication)
Console.WriteLine(apples / oranges);  // => 3   (integer division: truncates toward
zero)
Console.WriteLine(apples % oranges);  // => 1   (remainder)

// Integer division always truncates toward zero — the fractional part is discarded
int result = 7 / 2;
Console.WriteLine(result); // => 3, not 3.5

// Truncation applies to negative results too: -7 / 2 is -3, not -4
int negResult = -7 / 2;
Console.WriteLine(negResult); // => -3

// To get a decimal result, at least one operand must be a double or float
double precise = 7.0 / 2;
Console.WriteLine(precise); // => 3.5

// Remainder with negative operands: the sign of the result matches the dividend
Console.WriteLine(-7 % 3);  // => -1  (-7 = 3 × -2 + (-1))
Console.WriteLine(7 % -3);  // => 1   ( 7 = -3 × -2 + 1)
```

Целочисленное деление округляется в сторону нуля. Если оба операнда — целые, `/` отбрасывает дробную часть: `7 / 2` — это `3`, а не `3.5`. Усечение выполняется к нулю, а не в сторону меньшего числа: `-7 / 2` — это `-3` (а не `-4`). Чтобы получить десятичный результат, сделайте по крайней мере один операнд типом с плавающей запятой: `7.0 / 2``3.5` Это отличается от поведения некоторых языков, где `/` всегда дает результат с плавающей точкой.

Остаток (`%`) возвращает то, что остаётся после целочисленного деления: `1` — это `10 % 3`, потому что `10 = 3 × 3 + 1`. Это полезно для перебора фиксированного диапазона (`index % length`), проверки делимости (`n % 2 == 0`), и извлечения цифр. При отрицательных операндах знак результата совпадает со знаком *делимого* (левого операнда): `-7 % 3` — `-1`, а `7 % -3` — `1`.

Унарные операторы

Унарные операторы действуют на одном операнде.

C#

```
int temperature = 20;
int windChill = -5;
```

```
int heatIndex = +temperature;   // unary +: value unchanged (rarely needed)
int coldFactor = -windChill;    // unary -: negates the value → 5

Console.WriteLine(heatIndex);   // => 20
Console.WriteLine(coldFactor);  // => 5

bool isRaining = false;
bool isSunny = !isRaining;     // logical NOT: flips true/false
Console.WriteLine(isSunny);    // => True
```

- `+x` (унарный плюс) — оставляет значение неизменным; его редко записывают явно, но это допустимо.
- `-x` (унарный минус) — отрицает значение.
- `!x` (логическое НЕ) — меняет `true` на `false`, а `false` на `true`. Часто используется `!`: `if (!list.Contains(item))`

Увеличение и уменьшение

`++` добавляет 1 и `--` вычитает 1. Оба имеют *форму префикса* и форму *постфикса* , которая отличается от возвращаемого значения:

C#

```
int counter = 5;

// Prefix: increment first, then use the new value
int a = ++counter;
Console.WriteLine(a);       // => 6
Console.WriteLine(counter); // => 6

// Postfix: use the current value first, then increment
int b = counter++;
Console.WriteLine(b);       // => 6  (value before increment)
Console.WriteLine(counter); // => 7  (incremented after)

// Decrement works the same way
int score = 10;
Console.WriteLine(score--); // => 10 (current value; score becomes 9)
Console.WriteLine(score);   // => 9
```

- **Префикс** (`++i`,): `--i` увеличивает или уменьшает переменную сначала, а затем возвращает *новое* значение.
- **Postfix** (`i++`,): `i--` сначала возвращает *текущее* значение, а затем увеличивает или уменьшает переменную.

Когда `++` или `--` отображается как автономная инструкция (не часть большего выражения), префикс и постфикс имеют тот же эффект. Различие имеет значение, только если используется результат, например в назначении или в качестве аргумента метода.

Реляционные операторы

Реляционные операторы сравнивают два значения и возвращают значение `bool`.

Развернуть таблицу

Operator	Значение	Example
`<`	Меньше	`speed < limit`
`>`	Больше чем	`speed > limit`
`<=`	Меньше или равно	`score <= 100`
`>=`	Больше или равно	`score >= 0`

C#

```
int speed = 75;
int limit = 60;

Console.WriteLine(speed > limit);   // => True   (greater than)
Console.WriteLine(speed < limit);   // => False  (less than)
Console.WriteLine(speed >= limit);  // => True   (greater than or equal)
Console.WriteLine(speed <= limit);  // => False  (less than or equal)

// Relational operators work on all numeric types and char
// char comparison uses the character's numeric Unicode code point, not alphabetical
position
// 'B' (U+0042, value 66) is less than 'A' (U+0041, value 65)? No — 'A' (65) < 'B'
(66)
char grade = 'B';
Console.WriteLine(grade >= 'A' && grade <= 'C'); // => True  ('A'=65 <= 'B'=66 <=
'C'=67)
```

Реляционные операторы работают над всеми числовыми типами и `char`. Для `char` сравнения используется числовое значение кодовой точки Unicode символа, а не какой-либо алфавитный порядок или порядок, определяемый предметной областью. В приведенном выше примере с оценками `'B'` больше или равно `'A'`, так как `'B'` имеет кодовое значение в Юникоде 66, а `'A'` имеет кодовое значение в Юникоде 65 — *числа* определяют сравнение, а не смысл буквенных оценок.

Операторы равенства

`==` и `!=` проверьте, равны ли два значения. `!=` — `true`, если операнды **не** равны, и `false`, если равны.

C#

```
int expected = 42;
int actual = 42;

Console.WriteLine(actual == expected);  // => True   (values are equal)
Console.WriteLine(actual != expected);  // => False  (true when values are not equal)

string name = "Alice";
Console.WriteLine(name == "Alice");  // => True   (string content matches)
Console.WriteLine(name == "alice");  // => False  (case-sensitive)

int x = 5;
Console.WriteLine(x == 10);  // => False
```

Для числовых типов и `string` проверка на равенство сравнивает значения. Для ссылочных типов по умолчанию используется сравнение по идентичности (то есть указывают ли две переменные на один и тот же объект), но многие типы, включая `string` и `record`, переопределяют это и выполняют сравнение по содержимому. Чтобы получить полное представление о том, как работает равенство для типов значений, ссылочных типов, записей и структур, см. раздел Сравнения на равенство.

ⓘ **Примечание.**

У C# нет `===` оператора. Использование `===` приводит к ошибке компиляции:

C#

```
// This does not compile — C# has no === operator
bool same = (x === 10);
```

Если вы переходите с JavaScript, используйте `==` для сравнения значений (в C# `==` уже сравнивает по значению примитивные типы и строки). Распространённая похожая ошибка — случайно написать `=` (присваивание) там, где имелось в виду `==` (проверка на равенство). Компилятор перехватывает наиболее распространенные формы, но дважды проверьте любое `if` условие, содержащее `=`.

Условные логические операторы

`&&` (AND) и `||` (OR) объединяют `bool` выражения.

C#

```
int age = 20;
bool hasTicket = true;

// && (AND): both sides must be true
bool canEnter = age >= 18 && hasTicket;
Console.WriteLine(canEnter); // => True

// || (OR): at least one side must be true
bool freeEntry = age < 5 || age >= 65;
Console.WriteLine(freeEntry); // => False

// Short-circuit: right side is skipped when the result is already determined
// Here, items.Count is never called if items is null
List<string>? items = null;
bool hasItems = items != null && items.Count > 0;
Console.WriteLine(hasItems); // => False  (short-circuits; no NullReferenceException)
```

Оба оператора *используют сокращённое вычисление*: они пропускают вычисление правого операнда, когда результат уже известен.

- `&&` возвращает `false`, как только левая часть становится `false`. Правая сторона никогда не вычисляется.
- `||` возвращает `true`, как только левая часть становится `true`. Правая сторона никогда не вычисляется.

Сокращённое вычисление даёт практическое преимущество: вы можете безопасно выполнить операцию в правой части выражения, если в левой части есть проверка на null, как показано в примере выше. Если `items` — `null`, то `&&` на этом прекращается — `items.Count` никогда не вызывается, поэтому исключение `NullReferenceException` не возникает.

Условный оператор `?:`

Условный оператор (также называемый *оператором ternary*) вычисляет одно из двух выражений на основе условия:

```
```

```
condition ? value-when-true : value-when-false
```

C#

```
int temperature2 = 35;

// condition ? value-when-true : value-when-false
string weather = temperature2 > 30 ? "hot" : "comfortable";
Console.WriteLine(weather); // => hot

// Only the matching branch evaluates — the other branch is never run
int divisor = 0;
// The division 10 / divisor is never evaluated because divisor == 0 is true
int safe = divisor == 0 ? -1 : 10 / divisor;
Console.WriteLine(safe); // => -1
```

Оператор `?:` всегда вычисляет ровно одну ветвь — ветвь, которая не удовлетворяет условию, никогда не вычисляется. Это позволяет безопасно использовать выражение на одной стороне, которое завершится ошибкой для других входных данных, если условие должным образом защищает его.

Используйте `?:` для простых встроенных вариантов выбора. Для многовариантных условий или блоков кода конструкция `if`/`else` обычно понятнее.

Операторы присваивания

Простой оператор `=` назначения сохраняет значение в переменной:

C#

```
int level = 1;  // declaration + initialization
level = 5;      // reassignment
```

Назначение в C# является *правым ассоциативным*, что означает, что `a = b = c = 0` вычисляется справа налево: `c` получает `0`, а затем `b` получает `0`, а затем `a` получает `0`.

Комбинированное присваивание

Операторы составных назначений объединяют двоичную операцию с назначением:

Развернуть таблицу

Operator	Эквивалентно
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

C#

```
int level = 1;
level = 5;          // simple assignment: replaces the value
Console.WriteLine(level); // => 5

// Compound assignment: short form of binary operation + assignment
int hp = 100;
hp += 20;   // same as: hp = hp + 20
Console.WriteLine(hp); // => 120
hp -= 10;   // same as: hp = hp - 10
Console.WriteLine(hp); // => 110
hp *= 2;    // same as: hp = hp * 2
Console.WriteLine(hp); // => 220
hp /= 3;    // same as: hp = hp / 3 (integer division)
Console.WriteLine(hp); // => 73
hp %= 7;    // same as: hp = hp % 7
Console.WriteLine(hp); // => 3
```

Составное назначение больше, чем просто сокращенное. Он вычисляет левую сторону **ровно один раз** , а затем преобразует результат обратно в левый тип. Это важно, когда в левой части есть побочные эффекты (например, у индексатора массива), и именно поэтому составное присваивание для переменной `byte` компилируется без явного приведения типа, а развёрнутая форма — нет:

C#

```
// Assignment is right-associative: evaluated right to left
int a2, b2, c2;
a2 = b2 = c2 = 0;     // c2 = 0 first, then b2 = 0, then a2 = 0
Console.WriteLine($"{a2} {b2} {c2}"); // => 0 0 0

// Compound assignment evaluates the left side once and converts back to the LHS type
byte small = 200;
small += 10;  // equivalent to: small = (byte)(small + 10); result is 210
Console.WriteLine(small); // => 210
```

`small += 10` компилируется, так как компилятор автоматически вставляет сужающее преобразование — результат, `210`помещается в `byte` диапазон от 0 до 255. `small = small + 10` потребует явного приведения `(byte)`, поскольку арифметические операции приводят оба операнда к `int`.

Другие операторы C#

В этой статье рассматриваются операторы, с которыми вы чаще всего будете сталкиваться в повседневной работе с кодом. Язык C# включает в себя больше операторов, полезных в конкретных сценариях:

- **Операторы shift** (`<<`, `>>`, `>>>`) — сдвиг битов целочисленного значения влево или вправо по указанному количеству позиций. Побитовые **и целые логические операторы** (`&`, `|`, `^` `~`) — объединение или инвертирование целочисленных значений за раз, полезное в флагах, масках и низкоуровневом коде: побитовые и сменные операторы
- **`checked` и `unchecked`** — определяют, вызывает ли целочисленное переполнение исключение (`checked`) или происходит тихое циклическое переполнение (`unchecked`): checked и unchecked
- **Операторы null** (`??`, `?[]`, `??=`, `?.`) — безопасно обрабатывают значения `null`, предоставляя значения по умолчанию или выполняя сокращённый доступ к членам: операторы null
- **Операторы проверки типа и преобразования** (`is`, `as`, `typeof`, приведение `(T)`) — проверяют или преобразуют тип значения во время выполнения: Операторы проверки типа и приведения
- **Операторы диапазонов и индексов** (`..`, `^`) — создают диапазоны и конечные индексы для массивов и диапазонов: доступ к членам и условные операторы null
- **Назначение деконструкции** — распаковка кортежа или типа в отдельные переменные в одном выражении: деконструкция кортежей и других типов

См. также

- Выражения C# — как формы выражений и принцип приоритета операторов работают
- Сравнения на равенство — как `==`, `!=` и `Equals` работают для разных типов
- Операторы и выражения C# (справочник по языку) — полная таблица приоритета и каждый оператор

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.08.2026

Инструкции C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий **по началу работы**. Полный синтаксис инструкций см. в **инструкциях** спецификации языка C#.
>
> **Вы пришли из другого языка?** Объявления, условные конструкции, циклы и операторы return могут быть вам знакомы. В C# используется собственный синтаксис и классификация этих функций, в которой представлена эта статья.

Инструкция — это завершённая команда: «сделай это». Вместе инструкции программы образуют рецепт, которому программа следует от начала до конца. Большинство операторов выполняются в последовательности. Ветви выбирают шаги, которые нужно выполнить, и циклы повторяются.

В этом примере объявляется количество, затем оно выводится, а после этого используется оператор `if`, чтобы определить, нужно ли пополнить запасы:

C#

```
int quantity = 5;
Console.WriteLine($"Quantity: {quantity}"); // => Quantity: 5

if (quantity < 10)
{
    quantity = 10;
    Console.WriteLine("Restocked"); // => Restocked
}

Console.WriteLine($"Quantity: {quantity}"); // => Quantity: 10
```

Сначала прочитайте пример целиком, как команду, прежде чем рассматривать его части. Объявление, первый вызов `Console.WriteLine`, вся конструкция `if` и последний вызов `Console.WriteLine` являются инструкциями. Блок внутри инструкции `if` содержит еще две инструкции.

Инструкции часто содержат выражения

Операторы часто содержат *выражения*, которые представляют собой фрагменты кода, которые создают значения. В предыдущем примере вся конструкция `if` представляет собой инструкцию. Его условие, `quantity < 10`, является выражением, которое возвращает либо `true`, либо `false`.

Инструкция *объявления* объявляет локальную переменную или константу. Выражение инициализатора может предоставить свое первое значение:

```C#
int quantity = 5;
```

Полная строка — это инструкция объявления. Инициализатор `5` — это выражение в этой инструкции.

Выражение назначения хранит значение в переменной, свойстве, индексаторе или другом расположении хранилища. C# допускает, чтобы выражение присваивания служило *инструкцией-выражением*:

```C#
quantity = 10;
```

Эта инструкция выполняет действие, а не просто вычисляет значение. Вызовы методов и операции инкремента — это ещё два распространённых вида инструкций-выражений:

```C#
Console.WriteLine("Restocking");
quantity++;
```

Только следующие формы выражений могут использоваться как операторы выражений:

- Выражения назначения
- Выражения вызова методов
- Выражения создания объектов
- Выражения с префиксным или постфиксным инкрементом и декрементом
- `await` Выражения

Не каждое выражение может использоваться как самостоятельная инструкция. Например, `quantity + 1;` вычисляет значение, но не использует его. Это некорректное утверждение.

Вот полезное рабочее определение: оператор примерно соответствует полному предложению в английском языке, в то время как выражение похоже на фразу. Это различие помогает понять официальную терминологию и объяснения, которые вы столкнетесь в документации и спецификациях, хотя обычное кодирование редко требует, чтобы вы помните различие сознательно.

Групповые инструкции в блоках

Блоки группируют ноль или больше операторов между фигурными скобками (`{` и `}`). C# обрабатывает группу как одну инструкцию. В первом примере инструкция `Console.WriteLine` может включать как присваивание, так и вызов `if`, поскольку блок объединяет их в одно тело. Блоки могут быть вложены в другие блоки.

Операторы выбора и итерации называют свое тело *вложенным оператором*. Этот текст может быть одним оператором без фигурных скобок или блоком, который группирует несколько операторов.

Предпочитайте блок, даже если текст содержит только одну инструкцию. Фигурные скобки показывают, какие инструкции принадлежат тексту и не позволяют позже изменять случайное размещение инструкции за ее пределами.

Блоки определяют область переменной

Переменные, объявленные в блоке, находятся в области видимости с момента их объявления и до конца этого блока. Вложенный блок может использовать переменные, объявленные во внешнем блоке, но внешний блок не может использовать переменные, объявленные только во вложенном блоке:

C#

```
int outerValue = 10;

if (outerValue > 0)
{
    int innerValue = outerValue * 2;
    Console.WriteLine(innerValue); // => 20
}

// innerValue isn't in scope here.
```

Выберите утверждение для задачи

После распознавания операторов как команд можно выбрать один из различных типов по назначению:

- **Объявление данных:**Инструкции объявления вводят локальные переменные и константы.
- **Выполнение действий:** Инструкции-выражения присваивают значения, вызывают методы, создают объекты, увеличивают или уменьшают значения либо ожидают завершения асинхронных операций.
- **Выберите инструкции:**Selection, например `if` и `switch`выберите, какой код выполняется.
- **Повторяющиеся шаги:**Операторы итерации, такие как `foreach`, `while` и `for`, повторяют инструкцию или блок инструкций.
- **Передача управления:**Операторы перехода, такие как `break`, `continue`, `return` и `yield`, передают управление в другую точку программы.
- **Обработка исключений:**Инструкции обработки исключений, такие как `try`, `catch` и `throw`, реагируют на ошибки или сообщают о них.
- **Управление ресурсами:**Инструкцияusing гарантирует, что ресурсы удалены.
- **Используйте специализированное поведение:** Операторы checked и unchecked, fixed и lock поддерживают определенные сценарии.

Спецификация языка C#

Дополнительные сведения см. в разделе "Инструкции " спецификации языка C#.

См. также

- Ключевые слова инструкции
- Операторы и выражения C#

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.08.2026

Операторы выбора

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий **по началу работы**. Полный синтаксис см. в **инструкциях выбора** в справочнике по языку.
>
> **Вы пришли из другого языка?** C++, Java и JavaScript во многом унаследовали C, поэтому `if`, `else` и `switch` в C# читаются одинаково. Стоит отметить одно различие: C# запрещает переход без `break` между непустыми вариантами `switch`, и каждая метка `case` может проверять *шаблон*, а не только константу. Если вы переходите с Python, то `if`/`else if`/`else` в C# соответствует `if`/`elif`/`else`, а основанный на шаблонах `switch` в C# ближе всего к оператору `match` в Python.

Операторы выбора определяют, какой блок кода будет выполняться на основе условия. C# предоставляет два варианта: `if` (с необязательным `else`) для ветвления по логическому значению и `switch` для сравнения одного значения с несколькими случаями. *Логическое выражение* — это любое выражение, которое принимает значение `true` или `false`, например выражение сравнения `temperature >= 25`.

Ветвь с `if` и `else`

Оператор `if` выполняет свой блок только тогда, когда его логическое выражение имеет значение `true`. `else` Добавьте блок для предоставления кода, выполняемого при условии `false`:

C#

```
int temperature = 28;

// An if statement runs its block only when the condition is true.
// The else block runs when the condition is false.
if (temperature >= 25)
{
    Console.WriteLine("Warm"); // => Warm
}
else
{
```

```
    Console.WriteLine("Cool");
}
```

Тело `if` или `else` состоит из одного *оператора*, например присваивания или вызова метода. *Блочная инструкция* сама по себе является одной инструкцией, которая заключает в фигурные скобки (`{ }`) ноль или более инструкций.

Многие стандарты программирования рекомендуют заключать тела ветви в фигурные скобки даже для одной инструкции. Фигурные скобки явно задают область видимости. Это предотвращает распространённую ошибку: когда позже добавляют вторую строку, рассчитывая, что она будет выполняться условно, но вместо этого она выполняется безусловно. Только фигурные скобки определяют, какие инструкции относятся к ветке. C# не считает пробельные символы значимыми, поэтому сами по себе отступы тоже никогда не имеют значения. Фигурные скобки допустимы даже для одной строки: блок представляет собой единственную инструкцию, которую выполняет эта ветка. Используйте отступы в коде для удобочитаемости, но определяйте блок с помощью фигурных скобок.

Тестирование нескольких условий с помощью `else if`

Чтобы выбрать один из более чем двух путей, объедините условия с помощью `else if`. C# вычисляет каждое условие в порядке и запускает первый блок, условие которого равно `true`, а затем пропускает остальные. Последний `else` обрабатывает каждый оставшийся случай:

C#

```
int score = 82;

// Chain conditions with else if. The first branch whose condition is
// true runs; the compiler skips the rest.
string grade;
if (score >= 90)
{
    grade = "A";
}
else if (score >= 80)
{
    grade = "B";
}
else if (score >= 70)
{
    grade = "C";
}
```

```
else
{
    grade = "F";
}

Console.WriteLine(grade); // => B
```

Порядок имеет значение в цепочке. Так как первое соответствующее условие выигрывает, поместите в первую очередь самые конкретные условия. В предыдущем примере проверка `score >= 80` перед `score >= 70` гарантирует, что оценка 82 будет сопоставлена с `"B"`, а не с `"C"`.

Сопоставить значение с помощью инструкции `switch`

Когда вы сравниваете одно значение со многими отдельными случаями, конструкция `switch` воспринимается понятнее, чем длинная цепочка `else if`. Подобно `if`, оператор `switch` является оператором ветвления: он определяет, какой код выполнять. Каждая `case` метка содержит значение, соответствующее, и каждый `case` раздел заканчивается оператором `break` . Сгруппируйте метки, чтобы они использовали один общий блок, и используйте раздел `default` для значений, которые не соответствуют ни одному варианту:

C#

```
DayOfWeek day = DayOfWeek.Saturday;

// A switch statement compares one value against several case labels.
// Stacked labels share a body. Each section ends with break, and the
// default section runs when no case matches.
switch (day)
{
    case DayOfWeek.Saturday:
    case DayOfWeek.Sunday:
        Console.WriteLine("Weekend"); // => Weekend
        break;
    default:
        Console.WriteLine("Weekday");
        break;
}
```

В отличие от C и Java, C# не допускает перехода управления из одной непустой секции `case` в следующую. Каждый раздел, содержащий код, должен заканчиваться на `break` (или другим оператором перехода, например `return` или `goto`). Это правило предотвращает случайные ошибки, распространенные на этих языках.

Тестовые шаблоны с помощью `case` и `when`

Метка `case` не ограничивается константами. Он может протестировать *шаблон*, который является правилом, описывающим форму или значение данных. Реляционный шаблон, `< 0` например соответствует любому значению меньше нуля. Добавьте конструкцию `when`, чтобы указать дополнительное условие, которое также должно быть `true`, чтобы случай совпал:

C#

```
int measurement = 42;

// A case label can test a pattern instead of a constant. A when clause
// adds a condition that must also be true for the case to match.
switch (measurement)
{
    case < 0:
        Console.WriteLine("Negative");
        break;
    case 0:
        Console.WriteLine("Zero");
        break;
    case > 0 when measurement % 2 == 0:
        Console.WriteLine("Positive and even"); // => Positive and even
        break;
    default:
        Console.WriteLine("Positive and odd");
        break;
}
```

Варианты на основе шаблонов оцениваются сверху вниз, поэтому более конкретные шаблоны относятся к более общим. Полный каталог шаблонов см. в разделе "Сопоставление шаблонов".

Выбрать значение с помощью выражения

Операторы `if` и `switch` определяют, какой код выполняется. Если вместо этого нужно выбрать *значение*, используйте *выражение*. Выражение вычисляет значение, поэтому его можно использовать в любом месте, где ожидается значение, например справа от назначения или аргумента метода. C# предлагает формы выражений, которые выбирают значение из альтернативных вариантов, таких как условный оператор и `switch` выражение.

Условный оператор `?:`

Условный оператор `?:` выбирает одно из двух значений на основе логического условия.

О его синтаксисе, поведении при коротком замыкании и рекомендациях по его выбору вместо `if`/`else` см. в разделе Условный оператор ?:.

выражение `switch`

Выражение `switch` — это эквивалент инструкции `switch` в форме выражения. Вместо выполнения кода для соответствующего случая он вычисляет значение. Это короче, чем присваивать значение в каждой ветви инструкции `switch`, и компилятор предупреждает, если ветви не охватывают все возможные входные значения. Выражение `switch` является основной частью сопоставления шаблонов. Чтобы узнать, когда и как использовать его, см. сопоставление шаблонов и ссылку switch на выражение .

См. также

- Инструкции итерации
- Сопоставление шаблонов
- Операторы выбора (справочник по языку)
- Условный оператор (справочник по языку)

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 20.08.2026

Операторы итерации

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий **по началу работы**. Полное описание синтаксиса см. в разделе **операторы итерации** в справочнике по языку.
>
> **Вы пришли из другого языка?** Все четыре цикла C# (`foreach`, , `while` `do`-`while` и `for`) имеют прямые эквиваленты в Java, C++и JavaScript. Чаще всего используется `foreach` . Он перебирает элементы коллекции без использования индекса, как расширенный `for` в Java или `for...of` в JavaScript.

Операторы итерации многократно выполняют блок кода. Каждый проход через блок является *итерацией*, и повторяющийся блок является *циклом*. C# предоставляет четыре цикла. Начните с `foreach` для коллекций, используйте `while` и `do`-`while`, когда повторение определяется условием, и используйте `for`, когда нужен явный индекс.

Итерации коллекции с помощью `foreach`

Инструкция `foreach` выполняет тело по одному разу для каждого элемента коллекции по порядку. Это наиболее распространенный выбор для чтения коллекции, так как вы не управляете индексом или проверкой границ. Инструкция `foreach` предотвращает типичные ошибки на единицу:

C#

```
string[] names = ["Ana", "Ben", "Cleo"];

// foreach reads each element in order. It's the default loop for
// collections: no index to manage and no off-by-one mistakes.
foreach (string name in names)
{
    Console.WriteLine(name); // => Ana, then Ben, then Cleo
}
```

`foreach` работает с любым типом, который C# распознает как последовательность, включая массивы, List<T>и Dictionary<TKey,TValue>. Переменная итерации (`name` в предыдущем

примере) доступна только для чтения, поэтому ее нельзя переназначить внутри цикла.

Текст цикла — это одна *инструкция*, например назначение или вызов метода. *Блочная инструкция* сама по себе является одной инструкцией, которая заключает в фигурные скобки (`{ }`) ноль или более инструкций.

Многие стандарты программирования рекомендуют заключать текст цикла в фигурные скобки даже для одной инструкции. Фигурные скобки явно задают область видимости. Это предотвращает распространённую ошибку: если позже добавить вторую строку, которую, как ожидается, будут выполнять на каждой итерации, она вместо этого выполнится один раз после завершения цикла. Только фигурные скобки определяют, какие операторы принадлежат циклу. C# не считает пробельные символы значимыми, поэтому сами по себе отступы тоже никогда не имеют значения. Фигурные скобки допустимы даже для одной строки кода: блок представляет собой единственную инструкцию, которую цикл повторяет. Используйте отступы в коде для удобочитаемости, но определяйте блок с помощью фигурных скобок.

Повторять, пока условие выполняется с `while`

Цикл `while` проверяет логическое условие *перед* каждой итерацией. Если условие `false` в начале, тело цикла ни разу не выполняется, поэтому цикл `while` выполняется ноль или более раз:

C#

```
int countdown = 3;

// A while loop checks its condition before each iteration, so the body
// runs zero or more times.
while (countdown > 0)
{
    Console.WriteLine(countdown); // => 3, then 2, then 1
    countdown--;
}
```

Убедитесь, что что-то внутри цикла изменяет условие. В предыдущем примере `countdown--` в конечном итоге приводит условие к виду `false`. Цикл, условие которого никогда не становится `false`, выполняется бесконечно.

Запуск текста по крайней мере один раз с `do`-`while`

Цикл `do-while` проверяет своё условие *после* каждой итерации, поэтому тело всегда выполняется хотя бы один раз. Используйте его, когда первый проход должен произойти, прежде чем вы сможете оценить условие, например запрашивать входные данные, а затем проверять его:

C#

```
int attempts = 0;

// A do-while loop runs its body once, then checks the condition. Use it
// when the body must run at least one time.
do
{
    attempts++;
    Console.WriteLine($"Attempt {attempts}"); // => Attempt 1, then Attempt 2, then
Attempt 3
}
while (attempts < 3);
```

Считайте с помощью `for`

Оператор `for` цикла содержит три части: *инициализатор* , который запускается один раз перед циклом, *условие* , которое проверяется перед каждой итерацией, *итератор* , который выполняется после каждой итерации. Используйте `for`, когда нужен сам индекс, а не только элементы. Как правило, требуется индекс, когда нужно изменить элемент, а не считывать его значение.

C#

```
// A for loop fits when you need an explicit index. The three parts are
// the initializer, the condition, and the iterator.
for (int i = 0; i < 3; i++)
{
    Console.WriteLine(i); // => 0, then 1, then 2
}
```

Если вы только читаете элементы и не используете позиции и не присваиваете новые значения, используйте `foreach`. Он указывает намерение более четко и избегает арифметики индекса.

Выход или пропуск с помощью `break` и `continue`

Два оператора перехода обеспечивают более точное управление в любом цикле. Оператор `break` немедленно завершает цикл, пропуская все оставшиеся итерации:

C#

```
int[] numbers = [2, 4, 7, 8];

// break stops the loop immediately, skipping any remaining elements.
foreach (int number in numbers)
{
    if (number % 2 != 0)
    {
        Console.WriteLine($"First odd number: {number}"); // => First odd number: 7
        break;
    }
}
```

Оператор `continue` пропускает остальную часть текущей итерации и переходит к следующей:

C#

```
int[] values = [1, 2, 3, 4, 5];

// continue skips the rest of the current iteration and moves to the next.
foreach (int value in values)
{
    if (value % 2 == 0)
    {
        continue; // skip even numbers
    }

    Console.WriteLine(value); // => 1, then 3, then 5
}
```

Итерацию асинхронного потока с помощью `await foreach`

Асинхронный поток — это средство чтения, использующее асинхронную задачу для создания каждого следующего элемента. В C# это представлено интерфейсом IAsyncEnumerable<T>. Данные, поступающие с течением времени, такие как страницы из веб-API или строк из базы данных, соответствуют этой модели: извлечение следующего элемента является ожидаемой операцией вместо немедленного возврата.

Чтобы перебирать асинхронный поток, поставьте ключевое слово `await` перед `foreach`. Каждая итерация ожидает следующего элемента, поэтому цикл приостанавливается во время создания этого элемента, а не блокирует поток:

C#

```
private static async Task AwaitForeachExample()
{
    // await foreach consumes an asynchronous stream. Each iteration can
    // suspend while the next element is produced.
    await foreach (int value in GenerateAsync())
    {
        Console.WriteLine(value); // => 0, then 1, then 2
    }
}

private static async IAsyncEnumerable<int> GenerateAsync()
{
    for (int i = 0; i < 3; i++)
    {
        await Task.Delay(1); // stand-in for real asynchronous work
        yield return i;
    }
}
```

Асинхронные потоки основаны на `async` и `await`. Полное пошаговое руководство см. в разделе "Создание и использование асинхронных потоков".

См. также

- Инструкции выбора
- Операторы итерации (справочник по языку)
- Создание и использование асинхронных потоков

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 21.07.2026

Common collection types

> **Совет**
>
> This article is part of the **Fundamentals** section for developers who already know at least one programming language and are learning C#. If you're new to programming, start with the Get started tutorials first. For more information about collections, see Collections in the language reference. For more information about arrays, see The array reference type in the language reference. For more information about collection expressions, see Collection expressions (Collection literals) in the language reference.
>
> **Coming from another language?** C# arrays are fixed-size ordered collections, like arrays in Java or C++. List<T> is the everyday growable sequence, like Java's `ArrayList`, JavaScript's arrays, or Python's lists. Dictionary<TKey,TValue> stores values by key, like maps or dictionaries in many languages.

A *collection* is an object that stores multiple related values. Each value in a collection is an *element*. Choose an array when the set is fixed, such as the status names your work-item tracker always uses. Choose a List<T> when items come and go, such as a backlog that gains new work and drops completed items. Choose a Dictionary<TKey,TValue> when you look up values by a key, such as finding a work item by ID or a setting by name. The next section turns those examples into a quick decision model.

Choose a collection shape

Choose the collection that matches how your code uses the data. A *sequence* stores elements in order so you can reach them by position. Use an *array*, which is represented by Array, when you know the number of positions the sequence needs. An array's length can't change after you create it, but you can replace the element stored at an existing position. Use List<T> when you need a sequence that can add or remove elements. A *dictionary* stores values that you reach by key instead of by position. Use Dictionary<TKey,TValue> when each element has a lookup key, such as a name, ID, or code. The examples use a *collection expression*, which creates a collection from expressions between square brackets. The Create collections with collection expressions section explains that syntax in more detail.

C#

```
string[] sprintPlan = ["design", "code", "test"];
List<string> backlog = ["design", "code"];
Dictionary<string, int> priorities = new()
{
    ["docs"] = 2,
    ["tests"] = 1
};

backlog.Add("test");

Console.WriteLine($"Array: {string.Join(", ", sprintPlan)}");
Console.WriteLine($"List count: {backlog.Count}");
Console.WriteLine($"Priority for docs: {priorities["docs"]}");
// This example produces the following output:
//
//    Array: design, code, test
//    List count: 3
//    Priority for docs: 2
//
```

List<T> and Dictionary<TKey,TValue> are *generic types*. A generic type uses a type argument, such as `string` or `int`, to say what kind of values it stores. Generics let you reuse the same collection shape, such as a list or dictionary, with different element types, such as `string`, `int`, or a custom type. They also provide *type safety*: the compiler guarantees every element is the declared type, so you don't need casts and can't accidentally store the wrong type. For more information about generic types, see Generic types and methods in the fundamentals.

Store fixed-size data in arrays

An array is an ordered collection with a fixed length. You access an array element by its *index*, which is its zero-based position in the array. Index `0` is the first element, index `1` is the second element, and so on.

C#

```
string[] stages = ["design", "code", "test", "review"];

Console.WriteLine($"First: {stages[0]}");
Console.WriteLine($"test index: {Array.IndexOf(stages, "test")}");
// This example produces the following output:
//
//    First: design
//    test index: 2
//
```

Use `foreach` when you want to read every element in order. Use an index when the position matters, such as when you need the first element, the last element, or the position returned by Array.IndexOf.

The length of an array is fixed, but the elements in that array can change. Assign a new value to an existing index when the position stays the same but the stored value needs an update:

C#

```
string[] stages = ["design", "code", "test"];

stages[0] = "plan";

Console.WriteLine(string.Join(", ", stages));
Console.WriteLine($"Length: {stages.Length}");
// This example produces the following output:
//
//    plan, code, test
//    Length: 3
//
```

Grow and shrink a sequence with `List<T>`

A List<T> stores elements in order and can grow or shrink as your program runs. Use Add to append an element, Remove to remove a matching element, Contains to test whether an element exists, and IndexOf to find an element's position. Use the list indexer to replace the value at an existing position.

C#

```
List<string> workItems = ["design", "code", "test"];

workItems.Add("review");
workItems.Remove("code");
workItems[1] = "verify";

Console.WriteLine(string.Join(", ", workItems));
Console.WriteLine($"Has review: {workItems.Contains("review")}");
Console.WriteLine($"Index of verify: {workItems.IndexOf("verify")}");
// This example produces the following output:
//
//    design, verify, review
//    Has review: True
//    Index of verify: 1
//
```

A List<T> keeps the remaining elements in order when you add or remove items. When you remove an element, later elements move to lower indexes.

Use Insert to add one element at a specific position, InsertRange to add several elements at a position, and RemoveAt to remove an element by index:

C#

```
List<string> workItems = ["design", "test"];

workItems.Insert(1, "code");
Console.WriteLine($"Insert middle: {string.Join(", ", workItems)}");

workItems.Insert(0, "plan");
Console.WriteLine($"Insert front: {string.Join(", ", workItems)}");

workItems.InsertRange(workItems.Count, ["review", "deploy"]);
Console.WriteLine($"Insert range at end: {string.Join(", ", workItems)}");

workItems.RemoveAt(workItems.Count - 1);
Console.WriteLine($"Remove end: {string.Join(", ", workItems)}");

workItems.RemoveAt(2);
Console.WriteLine($"Remove middle: {string.Join(", ", workItems)}");

workItems.RemoveAt(0);
Console.WriteLine($"Remove front: {string.Join(", ", workItems)}");
// This example produces the following output:
//
//    Insert middle: design, code, test
//    Insert front: plan, design, code, test
//    Insert range at end: plan, design, code, test, review, deploy
//    Remove end: plan, design, code, test, review
//    Remove middle: plan, design, test, review
//    Remove front: design, test, review
//
```

Adding or removing at the end of a List<T> is fast. Adding with Add is an O(1) operation on average, and removing the last element with RemoveAt is O(1). Inserting or removing at the front or middle is O(n) because every later element shifts to a new index. If your code frequently inserts or removes at the front, a List<T> might be the wrong collection shape.

> **Совет**
>
> Big-O notation describes how the work grows as the collection size, `n`, grows. O(1), pronounced "order one," means the operation takes about the same amount of work no

matter how many elements the collection contains. O(n), pronounced "order n," means the work grows roughly in proportion to the number of elements in the collection.

Associate a value by key with `Dictionary<TKey,TValue>`

A Dictionary<TKey,TValue> stores key/value pairs. A *key/value pair* is one key and the value associated with that key; .NET represents one pair with KeyValuePair<TKey,TValue>. Use the dictionary indexer to add or update a value by key, and use TryGetValue when the key might not exist.

C#

```
Dictionary<string, int> priorities = new()
{
    ["docs"] = 2,
    ["tests"] = 1
};

priorities["review"] = 3;
priorities.Remove("review");

if (priorities.TryGetValue("docs", out int docsPriority))
{
    Console.WriteLine($"docs priority: {docsPriority}");
}
else
{
    Console.WriteLine("docs missing");
}

if (priorities.TryGetValue("deploy", out int deployPriority))
{
    Console.WriteLine($"deploy priority: {deployPriority}");
}
else
{
    Console.WriteLine("deploy missing");
}

Console.WriteLine($"count: {priorities.Count}");
// This example produces the following output:
//
//    docs priority: 2
//    deploy missing
//    count: 2
//
```

An *indexer* lets you use bracket syntax to access a value from an object. The dictionary indexer is useful when the key must exist or when you're assigning a value. TryGetValue checks whether a key exists and gives you the value when it does. For more information about the indexer operator, see Member access operators in the language reference.

You can also change the value associated with a key that already exists. Assign through the indexer to add a key or replace the value for an existing key. If the replacement depends on the current value, read the value first, then assign the new value:

C#

```
Dictionary<string, int> priorities = new()
{
    ["docs"] = 2,
    ["tests"] = 1
};

priorities["docs"] = 1;

if (priorities.TryGetValue("tests", out int testsPriority))
{
    priorities["tests"] = testsPriority + 1;
}

priorities["deploy"] = 3;

Console.WriteLine($"docs priority: {priorities["docs"]}");
Console.WriteLine($"tests priority: {priorities["tests"]}");
Console.WriteLine($"deploy priority: {priorities["deploy"]}");
// This example produces the following output:
//
//    docs priority: 1
//    tests priority: 2
//    deploy priority: 3
//
```

Create collections with collection expressions

A *collection expression* creates a collection from expressions between square brackets. Beginning with C# 12, collection expressions can create arrays, lists, and other collection types. A *spread element* copies the elements from another collection into the new collection.

C#

```
string[] planned = ["design", "code"];
// The spread element .. planned copies each element from planned.
string[] upcoming = [.. planned, "test"];
```

```
List<string> blocked = ["docs"];

Console.WriteLine($"Upcoming: {string.Join(", ", upcoming)}");
Console.WriteLine($"Blocked: {string.Join(", ", blocked)}");
// This example produces the following output:
//
//    Upcoming: design, code, test
//    Blocked: docs
//
```

Collection expressions keep initialization concise. A collection expression has no type of its own. The compiler converts the expression into the collection shape the context calls for. The receiving variable or parameter can turn it into an array, a List<T>, or other supported collection type.

Read from positions with indexes and ranges

Indexes answer "which single element?" Ranges answer "which contiguous slice?" Use indexes when your code needs one element and ranges when the subsection is part of the data you're working with. Arrays and `List<T>` support both indexes and ranges.

An Index can count from the start or from the end of a sequence. Use the number directly to count from the start: `phases[0]` reads the first element, and `phases[1]` reads the second element. Use `^` to count from the end: `phases[^1]` reads the last element, and `phases[^2]` reads the next-to-last element.

An Range selects a slice with `..` (two dots). The range operator isn't the same as the spread element you saw earlier in collection expressions, even though both use dots. Use a range `a..b` when you want elements from index `a` up to, but ***not including***, index `b`. The expression `phases[1..3]` creates a new array that contains the elements at index `1` and index `2`.

You can omit one or both range indexes. Omit the start index when the slice starts at the beginning, as in `phases[..2]` for the first two elements. Omit the end index when the slice continues through the last element, as in `phases[2..]` for the elements from index `2` to the end. Omit both indexes when you want a copy of the whole array, as in `phases[..]`. A range can mix index kinds. The expression `phases[1..^1]` combines a from-start index and a from-end index to return the middle elements, `code` and `test`.

C#

```
string[] phases = ["design", "code", "test", "deploy"];
List<string> checklist = ["design", "test", "review"];

Console.WriteLine($"Last: {phases[^1]}");
Console.WriteLine($"Middle: {string.Join(", ", phases[1..3])}");
```

```
Console.WriteLine($"Last two: {string.Join(", ", phases[^2..])}");
Console.WriteLine($"Without ends: {string.Join(", ", phases[1..^1])}");
Console.WriteLine($"First two: {string.Join(", ", phases[..2])}");
Console.WriteLine($"From third: {string.Join(", ", phases[2..])}");
Console.WriteLine($"All phases: {string.Join(", ", phases[..])}");
Console.WriteLine($"List last: {checklist[^1]}");
Console.WriteLine($"List first two: {string.Join(", ", checklist[0..2])}");
// This example produces the following output:
//
//    Last: deploy
//    Middle: code, test
//    Last two: test, deploy
//    Without ends: code, test
//    First two: design, code
//    From third: test, deploy
//    All phases: design, code, test, deploy
//    List last: review
//    List first two: design, test
//
```

For more information about indexes and ranges, see Explore ranges of data using indices and ranges in the tutorials.

See also

- Iteration statements
- LINQ queries
- Collections
- Arrays
- Collection expressions
- Member access operators
- Explore indexes and ranges

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 21.07.2026

LINQ queries

> **Совет**
>
> This article is part of the **Fundamentals** section for developers who already know at least one programming language and are learning C#. If you're new to programming, start with the Get started tutorials first. For more information about providers, operators, and advanced scenarios, see Language Integrated Query (LINQ).
>
> **Coming from another language?** LINQ query syntax reads like a SQL-style query written inside C#. LINQ method syntax reads like chained collection operations in JavaScript, Java streams, or Python pipelines. Both forms describe the same query.

Language Integrated Query (LINQ) is the C# feature set for querying data with C# syntax. Many LINQ operators use *deferred execution*: they describe the result first and read the data later, when your code asks for the results. A *query* describes which data to read and how to shape the result. A query reads from a *data source*. A data source can be an in-memory collection, such as an array or List<T>, or an external source, such as a database or XML, exposed through a LINQ provider. A *LINQ provider* is a library that connects LINQ syntax to a specific kind of data source. This article uses in-memory collections for its examples. A *sequence* is an ordered set of elements represented by IEnumerable<T>. For more information about provider-based queries, see Language Integrated Query (LINQ).

LINQ query expression syntax

The examples in this article read in-memory collections such as arrays and List<T>. A query usually has three parts: specify the data source, describe the result, and enumerate the source to produce the result. To *enumerate* a sequence means to read its elements one at a time, often with `foreach`.

C#

```
string[] names = ["Ana", "Ben", "Cleo", "Dara"];

IEnumerable<string> query =
    from name in names
    where name.Length >= 4
    orderby name
    select name;
```

```
foreach (string name in query)
{
    Console.WriteLine(name);
}
// This example produces the following output:
//
//    Cleo
//    Dara
//
```

The query expression starts with `from` to name the data source and range variable. The `where` clause keeps only matching elements, `orderby` sorts them, and `select` shapes the result.

LINQ method syntax

Query syntax uses clauses such as `from`, `where`, `orderby`, and `select`. *Method syntax* calls LINQ methods directly. For in-memory sequences, the standard LINQ methods are the built-in Enumerable methods for filtering, projection, sorting, grouping, and related operations.

C#

```
string[] names = ["Ana", "Ben", "Cleo", "Dara"];

IEnumerable<string> query = names
    .Where(name => name.Length >= 4)
    .OrderBy(name => name)
    .Select(name => name);

foreach (string name in query)
{
    Console.WriteLine(name);
}
// This example produces the following output:
//
//    Cleo
//    Dara
//
```

Method syntax is also called *fluent syntax* because each call returns a result that the next call can use. Many queries can use either form. Use the form that makes the query easiest to read.

Query syntax often reads well when the query has several clauses. The `let` clause gives a name to an intermediate value before the query filters, sorts, and selects the final result:

C#

```
(string Area, int Priority)[] workItems =
[
    ("docs", 2),
    ("tests", 1),
    ("deploy", 4),
    ("api", 1)
];

IEnumerable<string> nextItems =
    from item in workItems
    let label = $"{item.Area}: P{item.Priority}"
    where item.Priority <= 2
    orderby item.Priority, item.Area
    select label;

foreach (string item in nextItems)
{
    Console.WriteLine(item);
}
// This example produces the following output:
//
//    api: P1
//    tests: P1
//    docs: P2
//
```

Method syntax often reads well for short operations. It's necessary for methods that don't have query-syntax keyword. For example, Count returns one value directly:

C#

```
List<string> workItems = ["design", "docs", "deploy", "review"];

int count = workItems.Count(item => item.StartsWith('d'));

Console.WriteLine($"Starts with d: {count}"); // => Starts with d: 3
```

Lambda expressions and LINQ

A *lambda expression* is an anonymous function that you can pass as an argument. LINQ method syntax commonly uses lambda expressions to say what each operator should do with each element.

C#

```
string[] names = ["Ana", "Ben", "Cleo"];

IEnumerable<string> shortNames = names
```

```
    .Where(name => name.Length == 3)
    .Select(name => name.ToUpperInvariant());

foreach (string name in shortNames)
{
    Console.WriteLine(name);
}
// This example produces the following output:
//
//    ANA
//    BEN
//
```

In `name => name.Length == 3`, `name` is the input element and `name.Length == 3` is the Boolean expression that decides whether the element belongs in the result. For more information about lambda expressions and delegate types, see Lambda expressions, delegates, and events in the fundamentals.

Query-syntax clauses use lambda expressions too. Clauses such as `where`, `orderby`, and `select` compile to method calls that take lambda expressions. The range variable becomes the lambda parameter, and the clause expression becomes the lambda body. Query syntax is a concise way to write those same lambdas:

C#

```
string[] workItems = ["docs", "test", "deploy"];

IEnumerable<string> querySyntax =
    from item in workItems
    where item.Length == 4
    select item;

IEnumerable<string> methodSyntax =
    workItems.Where(item => item.Length == 4);

foreach (string item in querySyntax)
{
    Console.WriteLine($"Result: {item}");
}

foreach (string item in methodSyntax)
{
    Console.WriteLine($"Result: {item}");
}
// This example produces the following output:
//
//    Result: docs
//    Result: test
//    Result: docs
```

```
//    Result: test
//
```

Common LINQ methods

The most common LINQ methods cover the everyday steps of a query: filter data, shape results, sort, group, and summarize. These methods are in the System.Linq namespace and work with sequences such as IEnumerable<T>.

- Where keeps only the elements that match a condition.
- Select transforms each element into a new value. C# calls this operation a *projection*.
- OrderBy sorts elements.
- GroupBy creates groups of elements that share a key.
- Aggregation methods such as Sum, Count, and Aggregate produce a single value from all elements in the data source.

> ⓘ **Примечание.**
>
> If you know the traditional functional-programming terms, **Where** is a *filter*, **Select** is a *map* (C# calls it a projection), and aggregation methods such as **Aggregate**, **Sum**, and **Count** are a *reduce*.

C#

```
Dictionary<string, int> priorities = new()
{
    ["Docs"] = 2,
    ["Code"] = 1,
    ["Test"] = 3,
    ["Deploy"] = 4
};

IEnumerable<string> plannedWork = priorities
    .Where(workItem => workItem.Value <= 3)
    .OrderBy(workItem => workItem.Value)
    .Select(workItem => $"{workItem.Key}: {workItem.Value}");

foreach (string item in plannedWork)
{
    Console.WriteLine(item);
}
// This example produces the following output:
//
//    Code: 1
//    Docs: 2
```

```
//    Test: 3
//
```

A *projection* creates a result value from each input element. In the previous example, the projection creates strings that combine a work item name and its priority.

Group related values

Use GroupBy when the result should contain groups of elements that share a key. Each group is represented by IGrouping<TKey,TElement>, which exposes the group key and the elements in that group.

C#

```
string[] labels = ["api", "auth", "docs", "deploy"];

IEnumerable<IGrouping<char, string>> groups = labels.GroupBy(label => label[0]);

foreach (IGrouping<char, string> group in groups)
{
    Console.WriteLine($"{group.Key}: {string.Join(", ", group)}");
}
// This example produces the following output:
//
//    a: api, auth
//    d: docs, deploy
//
```

Grouping is useful for summaries, reports, and menus. For more information about joins, nested groupings, and provider-specific behavior, see Language Integrated Query (LINQ).

Run a query

Many LINQ operators use *deferred execution*. Deferred execution means operators that return a sequence, such as Where, Select, and OrderBy, don't run when you define them. They build the recipe for producing results. A `foreach` loop is one way to run that recipe:

C#

```
List<string> workItems = ["design", "docs"];

IEnumerable<string> query = workItems.Where(item => item.StartsWith('d'));

workItems.Add("deploy");

// The query runs here, when foreach asks for the results.
```

```
foreach (string item in query)
{
    Console.WriteLine(item);
}
// This example produces the following output:
//
//    design
//    docs
//    deploy
//
```

Other operations run a query immediately. Operators that return a single value, such as Count, Sum, First, and Any, must read the elements when you call them so they can produce that value.

C#

```
List<string> workItems = ["design", "docs"];

IEnumerable<string> query = workItems.Where(item => item.StartsWith('d'));

// Count() reads the matching elements right when this line runs.
int count = query.Count();
Console.WriteLine($"Count before add: {count}");

workItems.Add("deploy");

Console.WriteLine($"Stored count: {count}");
Console.WriteLine($"Current count: {query.Count()}");
// This example produces the following output:
//
//    Count before add: 2
//    Stored count: 2
//    Current count: 3
//
```

There's no single trigger that runs a query. *Eager evaluation*, also called immediate evaluation, runs the query right away and stores or returns the result. Scalar and aggregate operators such as Count, Sum, First, and Any use eager evaluation. So do ToList and ToArray when you need a snapshot of the current results. For more information about deferred execution, see Introduction to LINQ Queries in the LINQ documentation.

Compose queries

Real queries often grow from smaller decisions: start with a shared filter, add a sort for one screen, or apply an optional condition from user input. By composing queries, you can keep each step readable and reuse common parts instead of repeating one large query.

Imagine an app that shows open work items in several places. One view needs all open items, and another view needs only the highest-priority open items. To *compose* a query, start with a base query stored in a variable, then build a more specific query from it. Because execution is deferred, each step describes more of the result. No work happens until you enumerate the final query.

The following example stores the open work items in one query, then reuses that query to find the highest-priority open items:

C#

```
(string Title, int Priority, bool IsOpen)[] items =
[
    ("docs", 2, true),
    ("tests", 1, true),
    ("deploy", 3, false),
    ("api", 1, true)
];

IEnumerable<(string Title, int Priority, bool IsOpen)> openItems =
    from item in items
    where item.IsOpen
    select item;

IEnumerable<string> topOpenItems =
    from item in openItems
    where item.Priority == 1
    orderby item.Title
    select item.Title;

foreach (string title in topOpenItems)
{
    Console.WriteLine(title);
}
// This example produces the following output:
//
//    api
//    tests
//
```

You can also compose queries with method syntax. The next example starts with all open items, then conditionally adds another filter before it selects the titles:

C#

```
(string Title, string Area, bool IsOpen)[] items =
[
    ("write docs", "docs", true),
    ("fix tests", "tests", true),
```

```
    ("deploy site", "deploy", false)
];

bool onlyDocs = true;

IEnumerable<(string Title, string Area, bool IsOpen)> query =
    items.Where(item => item.IsOpen);

if (onlyDocs)
{
    query = query.Where(item => item.Area == "docs");
}

foreach (string title in query.Select(item => item.Title))
{
    Console.WriteLine(title); // => write docs
}
```

Composition can use either or both *eager evaluation* and *deferred execution*. Materialize a shared intermediate result with ToList or ToArray when you want that part to run once and stay fixed. Then build another deferred query from the cached results for the final output:

C#

```
List<(string Title, int Priority, bool IsOpen)> items =
[
    ("docs", 2, true),
    ("tests", 1, true),
    ("deploy", 3, false)
];

List<(string Title, int Priority, bool IsOpen)> openItems = items
    .Where(item => item.IsOpen)
    .ToList();

items.Add(("api", 1, true));

IEnumerable<string> cachedTopOpenItems = openItems
    .Where(item => item.Priority == 1)
    .OrderBy(item => item.Title)
    .Select(item => item.Title);

foreach (string title in cachedTopOpenItems)
{
    Console.WriteLine(title); // => tests
}
```

Explore LINQ in depth

This article uses in-memory collections to teach LINQ syntax and core operators. For more information about LINQ operators and advanced scenarios, see Language Integrated Query (LINQ). That article covers joins, XML, database providers, dynamic queries, and custom operators.

See also

- Collections
- Introduction to LINQ queries
- Write LINQ queries
- Standard query operators
- Lambda expressions
- LINQ and collections

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 21.07.2026

Обзор объектно-ориентированных методов в C#

Статья • 30.04.2025

В C#определение типа — класса, структуры или записи — это схема, указывающая, что может сделать тип. Объект в основном является блоком памяти, выделенным и настроенным в соответствии с схемой. В этой статье представлен обзор этих схем и их функций. В следующей статье этой серии представлены объекты.

Инкапсуляция

Инкапсуляция иногда называется первой основой или принципом объектно-ориентированного программирования. Класс или структура может указать, насколько доступен каждый из его членов для кода вне класса или структуры. Члены, не предназначенные для использования потребителями вне класса или сборки, скрыты, чтобы ограничить возможность ошибок кодирования или вредоносных атак. Дополнительные сведения см. в руководстве по программированию на основе объектов .

Члены

Члены типа включают все методы, поля, константы, свойства и события. В C#глобальные переменные или методы отсутствуют, так как в некоторых других языках нет. Даже точка входа программы, метод `Main`, должна быть объявлена в классе или структуре (неявно при использовании операторов верхнего уровня).

Следующий список содержит все различные виды элементов, которые можно объявить в классе, структуре или записи.

- Поля
- Константы
- Свойства
- Методы
- Конструкторы
- События
- Финализаторы
- Индексаторы
- Операторы
- Вложенные типы

Дополнительные сведения см. в разделе "Участники".

Доступность

Некоторые методы и свойства предназначены для вызова или доступа из кода за пределами класса или структуры, известного как *клиентский код*. Другие методы и свойства могут использоваться только в классе или структуре. Важно ограничить доступность кода, чтобы достичь его только предполагаемого клиентского кода. Вы указываете, как доступны типы и их члены для клиентского кода с помощью следующих модификаторов доступа:

- общественный
- защищенный
- внутренний
- защищенный внутренний
- частный
- закрытый защищенный.

Доступ по умолчанию — это `private`.

Наследство

Классы (но не структуры) поддерживают концепцию наследования. Класс, производный от другого класса, автоматически содержит все общедоступные, защищенные и внутренние члены базового класса, кроме его конструкторов и финализаторов.

Классы могут быть объявлены абстрактными, что означает, что один или несколько их методов не имеют реализации. Хотя абстрактные классы нельзя создать напрямую, они могут служить базовыми классами для других классов, которые обеспечивают недостающую реализацию. Классы также можно объявить как запечатанные , чтобы предотвратить наследование других классов от них.

Дополнительные сведения см. в разделе "Наследование и полиморфизм".

Интерфейсы

Классы, структуры и записи могут реализовать несколько интерфейсов. Для реализации из интерфейса означает, что тип реализует все методы, определенные в интерфейсе. Дополнительные сведения см. в разделе "Интерфейсы".

Универсальные типы

Классы, структуры и записи можно определить с помощью одного или нескольких параметров типа. Клиентский код предоставляет тип при создании экземпляра типа. Например, List<T> класс в System.Collections.Generic пространстве имен определяется одним параметром типа. Клиентский код создает экземпляр `List<string>` или `List<int>`, чтобы указать тип, который содержит список. Дополнительные сведения см. в разделе «Обобщения».

Статические типы

Классы (но не структуры или записи) можно объявить как `static`. Статический класс может содержать только статические элементы и не может быть создан с помощью ключевого `new` слова. Одна копия класса загружается в память при загрузке программы, а его члены получают доступ через имя класса. Классы, структуры и записи могут содержать статические элементы. Дополнительные сведения см. в разделе "Статические классы" и "статические члены классов".

Вложенные типы

Класс, структуру или запись можно вложить в другой класс, структуру или запись. Дополнительные сведения см. в разделе "Вложенные типы".

Частичные типы

Вы можете определить часть класса, структуры или метода в одном файле кода и другую часть в отдельном файле кода. Дополнительные сведения см. в разделе "Частичные классы и методы".

Инициализаторы объектов

Можно создать экземпляр и инициализировать объекты класса или структуры, а также коллекции объектов, назначив значения своим свойствам. Дополнительные сведения см. в разделе "Как инициализировать объекты с помощью инициализатора объектов".

Анонимные типы

В ситуациях, когда неудобно или необязательно создавать именованный класс, используются анонимные типы. Элементы данных с именами определяют анонимные типы. Дополнительные сведения см. в разделе "Анонимные типы".

Члены расширения

Класс можно расширить, не создавая производный класс, создавая отдельный тип. Этот тип содержит методы, которые можно вызывать, как если бы они принадлежали исходному типу. Дополнительные сведения см. в разделе "Методы расширения".

Неявно типизированные локальные переменные

В методе класса или структуры можно использовать неявную типизацию, чтобы компилятор определил тип переменной во время компиляции. Дополнительные сведения см. в разделе var (справочник по C#).

Записи

Модификатор `record` можно добавить в класс или структуру. Типы записей имеют встроенное поведение для обеспечения равенства на основе значений. Запись (либо `record class` или `record struct`) предоставляет следующие возможности:

- Краткий синтаксис для создания ссылочного типа с неизменяемыми свойствами.
- Цените равенство. Две переменные типа записи равны, если они имеют одинаковый тип, и если для каждого поля значения в обеих записях равны. Классы используют равенство ссылок: два переменных типа класса равны, если они ссылаются на один и тот же объект.
- Краткий синтаксис для недеструктивной мутации. Выражение `with` позволяет создать новый экземпляр записи, который является копией существующего экземпляра, но с изменёнными указанными значениями свойств.
- Встроенное форматирование для отображения. Метод `ToString` выводит имя типа записи и имена и значения общедоступных свойств.
- Поддержка иерархий наследования в классах записей. Классы записей поддерживают наследование. Структуры записей не поддерживают наследование.

Дополнительные сведения см. в разделе "Записи".

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

Объекты — создание экземпляров типов

Определение класса или структуры похоже на схему, указывающую, что может сделать тип. Объект — это блок памяти, который программа выделяет и настраивает в соответствии с схемой. Программа может создавать множество объектов одного класса. Объекты также можно называть экземплярами. Их можно хранить в именованной переменной или в массиве или коллекции. Клиентский код использует эти переменные для вызова методов и доступа к общедоступным свойствам объекта. В объектно-ориентированном языке, например C#, типичная программа состоит из нескольких объектов, взаимодействующих динамически.

> ⓘ **Примечание.**
>
> Статические типы ведут себя не так, как описано в этой статье. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Экземпляры структур и экземпляры классов

Поскольку классы являются ссылочными типами, переменная объекта класса содержит ссылку на адрес объекта в управляемой куче. При назначении второй переменной одного типа первой переменной обе переменные ссылаются на объект по указанному адресу. В этой статье этот вопрос далее рассматривается более подробно.

Экземпляры классов создаются с помощью new оператора. В следующем примере `Person` тип и `person1` `person2` являются экземплярами или объектами этого типа.

C#

```
using System;

public class Person(string name, int age)
{
    public string Name { get; set; } = name;
    public int Age { get; set; } = age;
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
```

```
        Person person1 = new("Leopold", 6);
        Console.WriteLine($"person1 Name = {person1.Name} Age = {person1.Age}");

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine($"person2 Name = {person2.Name} Age = {person2.Age}");
        Console.WriteLine($"person1 Name = {person1.Name} Age = {person1.Age}");

        /*
            Output:
            person1 Name = Leopold Age = 6
            person2 Name = Molly Age = 16
            person1 Name = Molly Age = 16
        */
    }
}
```

Поскольку структуры являются типами значений, переменная объекта структуры содержит копию всего объекта. Вы также можете создавать экземпляры структур с помощью `new` оператора, но его не нужно использовать, как показано в следующем примере:

C#

```
using System;

namespace Example
{
    public struct Person
    {
        public string Name;
        public int Age;
        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    public class Application
    {
        static void Main()
        {
            // Create  struct instance and initialize by using "new".
            // Memory is allocated on thread stack.
            Person p1 = new("Alex", 9);
            Console.WriteLine($"p1 Name = {p1.Name} Age = {p1.Age}");
```

```
            // Create  new struct object. Note that  struct can be initialized
            // without using "new".
            Person p2 = p1;

            // Assign values to p2 members.
            p2.Name = "Spencer";
            p2.Age = 7;
            Console.WriteLine($"p2 Name = {p2.Name} Age = {p2.Age}");

            // p1 values remain unchanged because p2 is  copy.
            Console.WriteLine($"p1 Name = {p1.Name} Age = {p1.Age}");
        }
    }
    /*
        Output:
        p1 Name = Alex Age = 9
        p2 Name = Spencer Age = 7
        p1 Name = Alex Age = 9
    */
}
```

Стек потока выделяет память как для `p1`, так и для `p2`. Программа освобождает эту память вместе с типом или методом, в котором она объявлена. Такое управление памятью — одна из причин, по которой структуры копируются при присваивании. В отличие от этого, общеязыковая среда выполнения (CLR) автоматически освобождает память, которую она выделила для экземпляра класса, когда все ссылки на объект выходят из области видимости. Невозможно детерминированно уничтожить объект класса, как можно в C++. Дополнительные сведения о сборке мусора в .NET см. в статье "Сборка мусора".

> ⓘ **Примечание.**
>
> Общеязыковая среда выполнения значительно оптимизирует выделение и освобождение памяти в управляемой куче. В большинстве случаев нет существенной разницы в затратах на производительность между выделением экземпляра класса в куче и экземпляра структуры в стеке.

Удостоверение объекта и равенство значений

При сравнении двух объектов для равенства сначала определите, представляют ли эти две переменные один и тот же объект в памяти или если значения одного или нескольких их полей эквивалентны. Если вы хотите сравнить значения, рассмотрите, являются ли объекты экземплярами типов значений (структур) или ссылочных типов (классы, делегаты, массивы).

- Используйте статический Object.ReferenceEquals метод, чтобы определить, относятся ли два экземпляра класса к одному расположению в памяти (что означает, что они имеют одно и то же *удостоверение*). (System.Object является неявным базовым классом для всех типов значений и ссылочных типов, включая определяемые пользователем структуры и классы.)

- По умолчанию метод ValueType.Equals определяет, имеют ли поля экземпляров в двух экземплярах структур одинаковые значения. Так как все структуры неявно наследуются System.ValueType, метод вызывается непосредственно в объекте, как показано в следующем примере:

  C#

  ```
  // Person is defined in the previous example.

  //public struct Person(string name, int age)
  //{
  //    public string Name { get; set; } = name;
  //    public int Age { get; set; } = age;
  //}

  Person p1 = new("Wallace", 75);
  Person p2 = new("", 42);
  p2.Name = "Wallace";
  p2.Age = 75;

  if (p2.Equals(p1))
      Console.WriteLine("p2 and p1 have the same values.");

  // Output: p2 and p1 have the same values.
  ```

  Реализация по умолчанию System.ValueType `Equals` использует упаковку и отражение в некоторых случаях. Сведения о том, как реализовать эффективный алгоритм проверки на равенство для конкретного типа, см. в разделе Самостоятельная реализация равенства, если тип не может быть записью. Записи — это ссылочные типы, которые используют семантику значений для определения равенства.

- Чтобы определить, равны ли значения полей в двух экземплярах класса, можно использовать Equals метод или оператор == . Однако используйте их только в том случае, если класс переопределяет или перегружает их, чтобы предоставить пользовательское определение того, что означает "равенство" для объектов этого типа. Класс также может реализовать IEquatable<T> интерфейс или IEqualityComparer<T> интерфейс. Оба интерфейса предоставляют методы, которые можно использовать для проверки равенства значений. При проектировании

собственных классов, которые переопределяют `Equals`, обязательно следуйте рекомендациям, приведённым в руководстве Реализуйте равенство самостоятельно, если тип не может быть записью и Object.Equals(Object).

Связанные разделы

Дополнительные сведения можно найти здесь

- Классы
- Конструкторы
- Финализаторы
- События
- объекта
- Наследование
- класс
- типы структур
- новый оператор
- Общая система типов

Last updated on 24.08.2026

Наследование — создание производных типов для более специализированного поведения

17.06.2025

Наследование вместе с инкапсуляцией и полиморфизмом является одной из трех основных характеристик объектно-ориентированного программирования. Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах. Класс, члены которого наследуются, называют *базовым классом*, а класс, наследующий эти элементы, называется *производным классом*. Производный класс может иметь только один прямой базовый класс. Однако наследование является транзитивным. Если `ClassC` происходит от `ClassB`, а `ClassB` происходит от `ClassA`, то `ClassC` наследует члены, объявленные в `ClassB` и `ClassA`.

> **Примечание**
>
> Структуры не поддерживают наследование, но могут реализовывать интерфейсы.

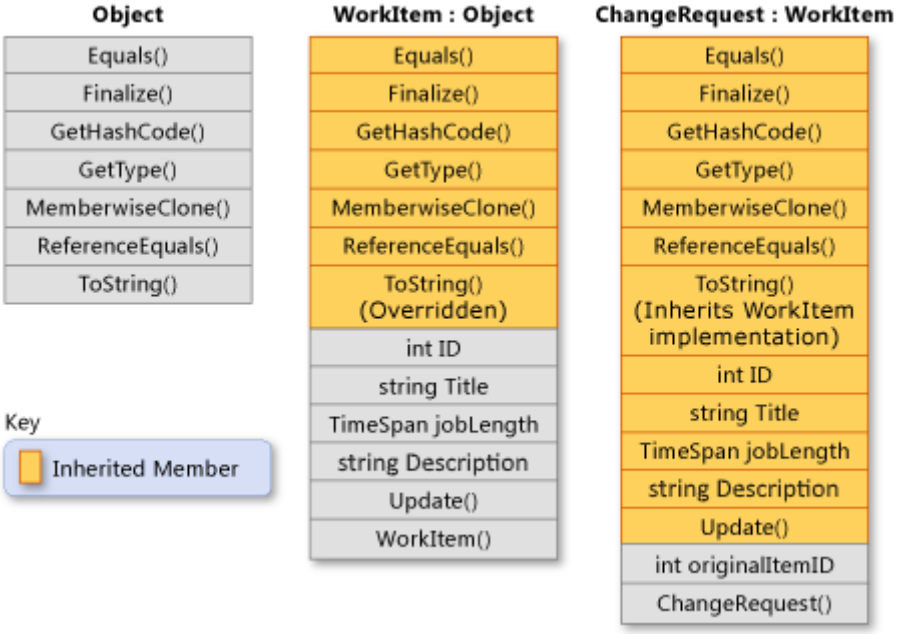
Концептуально производный класс является специализацией базового класса. Например, если у вас есть базовый класс `Animal`, у вас может быть один производный класс, который называется `Mammal` `Reptile` и другой производный класс. `Mammal` является `Animal`, а `Reptile` является `Animal`, но каждый производный класс представляет разные специализации базового класса.

Объявления интерфейса могут определять реализацию по умолчанию для своих членов. Эти реализации наследуются производными интерфейсами и классами, реализующими эти интерфейсы. Дополнительные сведения о методах интерфейса по умолчанию см. в статье об интерфейсах.

При определении класса как производного другого класса, производный класс автоматически получает все элементы базового класса, за исключением его конструкторов и финализаторов. Производный класс повторно использует код в базовом классе без необходимости повторного выполнения. Вы можете добавить дополнительные элементы в производный класс. Производный класс расширяет функциональные возможности базового класса.

На следующем рисунке показан класс `WorkItem` , представляющий элемент работы в некоторых бизнес-процессах. Как и все классы, он происходит от System.Object и наследует все его методы. `WorkItem` добавляет шесть своих членов. Эти члены включают

конструктор, поскольку конструкторы не передаются по наследству. Класс `ChangeRequest` наследует от `WorkItem` и представляет определенный вид рабочего элемента. `ChangeRequest` добавляет еще два члена к тем, которые он наследует от `WorkItem` и Object. Он должен добавить собственный конструктор, а также должен добавить `originalItemID`. Свойство `originalItemID` позволяет `ChangeRequest` экземпляру связываться с оригиналом `WorkItem` , к которому применяется запрос на изменение.



В следующем примере показано, как отношения классов, показанные на предыдущем рисунке, выражаются в C#. В примере также показано, как `WorkItem` переопределяет виртуальный метод Object.ToString, и как класс `ChangeRequest` наследует реализацию метода `WorkItem`. Первый блок определяет классы:

C#

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
```

```
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
```

```
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemID is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}
```

В следующем блоке показано, как использовать базовые и производные классы:

C#

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Абстрактные и виртуальные методы

Если базовый класс объявляет метод как virtual, производный класс может override метод с собственной реализацией. Если базовый класс объявляет член как abstract, этот метод

должен быть переопределен в любом не абстрактном классе, который напрямую наследует от этого класса. Если производный класс является абстрактным, он наследует абстрактные члены без их реализации. Абстрактные и виртуальные члены являются основой полиморфизма, которая является второй основной характеристикой объектно-ориентированного программирования. Дополнительные сведения см. в разделе Полиморфизм.

Абстрактные базовые классы

Класс можно объявить абстрактным , если вы хотите предотвратить прямое создание экземпляров с помощью нового оператора. Абстрактный класс можно использовать только в том случае, если новый класс является производным от него. Абстрактный класс может содержать одну или несколько подписей методов, которые сами объявляются как абстрактные. Эти сигнатуры указывают параметры и возвращаемое значение, но не имеют реализации (текст метода). Абстрактный класс не должен содержать абстрактные члены; Однако если класс содержит абстрактный член, сам класс должен быть объявлен как абстрактный. Производные классы, не абстрактные сами по себе, должны предоставлять реализацию для абстрактных методов из абстрактного базового класса.

Интерфейсы

Интерфейс — это ссылочный тип, определяющий набор элементов. Все классы и структуры, реализующие этот интерфейс, должны реализовать этот набор элементов. Интерфейс может определить реализацию по умолчанию для любого или всех этих элементов. Класс может реализовать несколько интерфейсов, даже если он может быть производным только от одного прямого базового класса.

Интерфейсы используются для определения конкретных функций или возможностей для классов, которые не обязательно связаны отношением типа "является". Например, интерфейс можно реализовать любым классом или структурой, System.IEquatable<T> чтобы определить, эквивалентны ли два объекта типа (однако тип определяет эквивалентность). IEquatable<T> не подразумевает того же рода отношения «является», которое существует между базовым классом и производным классом (например, `Mammal` — это `Animal`). Дополнительные сведения см. в разделе "Интерфейсы".

Предотвращение дальнейшего получения

Класс может запретить другим классам наследовать от него или от любого из его элементов, объявив себя или элемент как sealed.

Скрытие членов базового класса членами производного класса

Производный класс может скрывать элементы базового класса, объявляя элементы с одинаковым именем и подписью. new Модификатор можно использовать явным образом, чтобы указать, что элемент не предназначен для переопределения базового элемента. Использование new не является обязательным, но предупреждение компилятора будет создано, если new не используется. Чтобы узнать больше, см. версионирование с ключевыми словами Override и New и когда использовать ключевые слова Override и New.

Полиморфизм

13.10.2025

Полиморфизм часто называется третьим столпом объектно-ориентированного программирования после инкапсуляции и наследования. Полиморфизм — слово греческого происхождения, означающее "многообразие форм" и имеющее несколько аспектов.

- Во время выполнения объекты производного класса можно рассматривать как объекты базового класса в таких местах, как параметры метода и коллекции или массивы. Когда возникает полиморфизм, объявленный тип объекта перестает соответствовать своему типу во время выполнения.
- Базовые классы могут определять и реализовывать виртуальные*методы*, а производные классы могут переопределять их, что означает, что они предоставляют собственное определение и реализацию. Во время выполнения, когда клиент вызывает метод, CLR выполняет поиск типа объекта во время выполнения и вызывает перезапись виртуального метода. В исходном коде можно вызвать метод в базовом классе и обеспечить выполнение версии метода, относящейся к производному классу.

Виртуальные методы позволяют работать с группами связанных объектов универсальным способом. Представим, например, приложение, позволяющее пользователю создавать различные виды фигур на поверхности для рисования. Во время компиляции не известно, какие типы фигур будут создаваться пользователем. При этом приложению необходимо отслеживать все различные типы создаваемых фигур и обновлять их в ответ на движения мыши. Для решения этой проблемы можно использовать полиморфизм, выполнив два основных действия.

1. Создать иерархию классов, в которой каждый отдельный класс фигур является производным из общего базового класса.
2. Применить виртуальный метод для вызова соответствующего метода на любой производный класс через единый вызов в метод базового класса.

Для начала создайте базовый класс с именем `Shape` и производные классы, например `Rectangle`, `Circle` и `Triangle`. Присвойте классу `Shape` виртуальный метод с именем `Draw` и переопределите его в каждом производном классе для рисования конкретной фигуры, которую этот класс представляет. Создайте объект `List<Shape>` и добавьте в него `Circle`, `Triangle` и `Rectangle`.

```
C#
```

```
public class Shape
{
    // A few example members
    public int X { get; init; }
    public int Y { get; init; }
    public int Height { get; init; }
    public int Width { get; init; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

Для обновления поверхности рисования используйте цикл foreach, чтобы выполнить итерацию списка и вызвать метод `Draw` на каждом объекте `Shape` в списке. Несмотря на то, что каждый объект в списке имеет объявленный тип `Shape`, будет вызван тип времени выполнения (переопределенная версия метода в каждом производном классе).

```
C#
```

```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
List<Shape> shapes =
[
    new Rectangle(),
    new Triangle(),
    new Circle()
];

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

В C# каждый тип является полиморфным, так как все типы, включая пользовательские, наследуют Object.

Обзор полиморфизма

Виртуальные члены

Если производный класс наследует от базового класса, он включает все члены базового класса. Все поведение, объявленное в базовом классе, является частью производного класса. Это позволяет рассматривать объекты производного класса как объекты базового класса. Модификаторы доступа (`public`, , `protected` `private` и т. д.) определяют, доступны ли эти члены из реализации производного класса. Виртуальные методы предоставляют конструктору различные варианты поведения производного класса:

- Производный класс может переопределить виртуальные элементы в базовом классе, определяя новое поведение.
- Производный класс может наследовать ближайший метод базового класса, не переопределяя его, сохраняя существующее поведение, но позволяя дальнейшим производным классам переопределять метод.

- Производный класс может определить новую не-виртуальную реализацию этих элементов, которые скрывают реализации базового класса.

Производный класс может переопределить член базового класса, только если последний будет объявлен виртуальным или абстрактным. Производный член должен использовать ключевое слово override, указывающее, что метод предназначен для участия в виртуальном вызове. Примером является следующий код:

C#

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty => 0;
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Поля не могут быть виртуальными; только методы, свойства, события и индексаторы могут быть виртуальными. Когда производный класс переопределяет виртуальный член, он вызывается даже в то случае, если доступ к экземпляру этого класса осуществляется в качестве экземпляра базового класса. Примером является следующий код:

C#

```
DerivedClass B = new();
B.DoWork();  // Calls the new method.

BaseClass A = B;
A.DoWork();  // Also calls the new method.
```

Виртуальные методы и свойства позволяют производным классам расширять базовый класс без необходимости использовать реализацию базового класса метода. Дополнительные сведения см. в разделе Управление версиями с помощью ключевых слов Override и New. Еще одну возможность определения метода или набора методов, реализация которых оставлена производным классам, дает интерфейс.

Сокрытие членов базового класса новыми членами

Если вы хотите, чтобы производный класс имел член с тем же именем, что и член в базовом классе, можно использовать ключевое слово new, чтобы скрыть член базового класса. Ключевое `new` слово помещается перед типом возвращаемого элемента класса, который заменяется. Примером является следующий код:

C#

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

При использовании ключевого `new` слова создается метод, который *скрывает* метод базового класса, а не *переопределяет* его. Это отличается от виртуальных методов. При скрытии метода вызываемый метод зависит от типа переменной во время компиляции, а не типа времени выполнения объекта.

Скрытые члены базового класса можно получить из клиентского кода, приведя экземпляр производного класса к экземпляру базового класса. Например:

C#

```
DerivedClass B = new();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

В этом примере оба переменных ссылаются на один и тот же экземпляр объекта, но метод, который вызывается, зависит от объявленного типа переменной: `DerivedClass.DoWork()` при доступе `DerivedClass` через переменную и `BaseClass.DoWork()` при доступе `BaseClass` через переменную.

Защита виртуальных членов от переопределения производными классами

Виртуальные члены остаются виртуальными, независимо от числа классов между ними и классом, их объявившим. Если класс `A` объявляет виртуальный член, и класс `B` является производным от `A`, а класс `C` производным от `B`, класс `C` наследует виртуальный член и может переопределить его, независимо от того, объявил ли класс `B` переопределение для этого члена. Примером является следующий код:

C#

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

Производный класс может остановить виртуальное наследование, объявив переопределение как запечатанное. Для остановки наследования в объявление члена класса нужно вставить ключевое слово `sealed` перед ключевым словом `override`. Примером является следующий код:

C#

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

В предыдущем примере метод `DoWork` более не является виртуальным ни для одного класса, производного от класса `C`. Он по-прежнему является виртуальным для экземпляров класса `C`, даже если они приводятся к типу `B` или типу `A`. Запечатанные методы можно заменить производными классами с помощью ключевого слова `new`, как показано в следующем примере:

C#

```
public class D : C
{
    public new void DoWork() { }
}
```

В этом случае, если `DoWork` вызывается для `D` с помощью переменной типа `D`, вызывается новый `DoWork`. Если переменная типа `C`, `B` или `A` используется для доступа к экземпляру `D`, вызов `DoWork` следует правилам виртуального наследования, направляя эти вызовы в реализацию `DoWork` в классе `C`.

Доступ к виртуальным членам базового класса из производных классов

Производный класс, заменяющий или переопределяет метод или свойство, по-прежнему может получить доступ к методу или свойству базового класса с помощью ключевого `base` слова. Примером является следующий код:

C#

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

Для получения дополнительной информации см. базу.

> **Примечание**
>
> Рекомендуется, чтобы разработчики при реализации виртуальных членов использовали `base` для вызова реализации базового класса этого члена. Разрешение поведения базового класса позволяет производному классу концентрироваться на реализации поведения, характерного для производного класса. Если реализация базового класса не вызывается, то он до производного класса, чтобы обеспечить совместимость их поведения с поведением базового класса.

Деконструкция кортежей и других типов

Статья • 22.11.2024

Кортеж позволяет вам легко получить несколько значений при вызове метода. Но после получения кортежа вам нужно будет обработать его отдельные элементы. Работа с элементом по элементу является громоздкой, как показано в следующем примере. Метод `QueryCityData` возвращает три кортежа, и каждое из его элементов назначается переменной в отдельной операции.

C#

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Получение нескольких значений полей и свойств из объекта может быть равно громоздким: необходимо назначить значение поля или свойства переменной на основе элемента.

Можно извлечь несколько элементов из кортежа или получить несколько полей, свойств и вычисляемых значений из объекта в одной *деконструкционной* операции. Чтобы деконструировать кортеж, необходимо назначить его элементы отдельным переменным. При деконструкции объекта вы присваиваете отдельным переменным выбранные значения.

Кортежи

Язык C# имеет встроенную поддержку деконструкции кортежей, которая позволяет извлекать из кортежа все элементы за одну операцию. Общий синтаксис деконструкции кортежа напоминает синтаксис его определения: переменные, которым будут присвоены элементы кортежа, указываются в круглых скобках в левой части оператора присваивания. Например, следующая инструкция назначает элементы четырех кортежей четырем отдельным переменным:

```
C#

var (name, address, city, zip) = contact.GetAddressInfo();
```

Существует три способа деконструкции кортежа:

- Вы можете явно объявить тип каждого поля в скобках. В следующем примере этот подход используется для деконструкции трех кортежей, возвращаемых методом `QueryCityData` .

  ```
  C#

  public static void Main()
  {
      (string city, int population, double area) = QueryCityData("New
  York City");

      // Do something with the data.
  }
  ```

- Вы можете использовать ключевое слово `var`, чтобы C# определил тип каждой переменной. Ключевое слово `var` помещается за пределами скобок. В следующем примере используется вывод типа при деконструкции трех кортежей, возвращаемых методом `QueryCityData` .

  ```
  C#

  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

Кроме того, вы можете использовать ключевое слово `var` при объявлении отдельных или всех переменных внутри скобок.

```
C#

public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York
City");

    // Do something with the data.
}
```

Предыдущий пример является громоздким и не рекомендуется.

- Наконец, можно разложить кортеж на существующие переменные.

```
C#

public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- Объявление и назначение переменных можно смешивать в деконструкции.

```
C#

public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Нельзя указать определенный тип вне круглых скобок, даже если каждое поле в кортеже имеет одинаковый тип. Это приводит к возникновению ошибки компилятора CS8136, "Деконструкция "var (...)" формы запрещает определенный тип для var".

Необходимо назначить каждому элементу кортежа переменной. Если вы опустите элементы, компилятор создает ошибку CS8132: "Не удается деконструировать кортеж элементов x в переменные y".

Элементы кортежа с отменой

При деконструкции кортежа нас часто интересуют значения только некоторых элементов. Вы можете воспользоваться поддержкой C# для *отбрасывания*, которые являются переменными, предназначенными только для записи, значения которых вы решили игнорировать. Вы объявляете отмену символом подчеркивания ("_") в назначении. Вы можете отбросить столько значений, сколько вам нравится; один отброс, `_`, представляет все отброшенные значения.

В следующем примере показано использование кортежей с пустыми переменными. Метод `QueryCityDataForYears` возвращает шесть кортежей с именем города, его районом, годом, население города в течение этого года, второй год и население города в течение этого второго года. В примере показано изменение численности населения за эти два года. Из доступных в кортеже данных нас не интересует площадь города, а название города и две даты известны нам уже на этапе разработки. Следовательно, нас интересуют только два значения численности населения, которые хранятся в кортеже. Остальные значения можно обработать как пустые переменные.

C#

```
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York
City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
```

```
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Определяемые пользователем типы

C# предоставляет встроенную поддержку деконструкции типов кортежей, recordи типов DictionaryEntry . Тем не менее, если вы являетесь создателем класса, структуры или интерфейса, вы можете разрешить деконструкцию экземпляров определенного типа, реализовав один или несколько методов `Deconstruct`. Метод возвращает ничего (void). Параметр в сигнатуре метода представляет каждое значение для деконструирования. Например, следующий метод `Deconstruct` класса `Person` возвращает первое, среднее и семейное имя:

C#

```
public void Deconstruct(out string fname, out string mname, out string
lname)
```

Затем можно деконструировать экземпляр `Person` класса с именем `p` назначения, как показано в следующем коде:

C#

```
var (fName, mName, lName) = p;
```

В следующем примере показана перегрузка метода `Deconstruct` для возвращения различных сочетаний свойств объекта `Person`. Отдельные перегрузки возвращают следующие значения:

- Имя первого и семейного имени.
- Первое, среднее и семейное имя.

- Имя, имя семьи, имя города и имя штата.

C#

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string
lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");
```

```
        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Несколько методов `Deconstruct` с одинаковым числом параметров вносят путаницу. Старайтесь определять методы `Deconstruct` с разным числом параметров или аргументов. `Deconstruct` методы с одинаковым числом параметров нельзя различить при разрешении перегрузки.

Определяемый пользователем тип с отменами

Как и с кортежами, пустые переменные можно применять с пользовательскими типами, чтобы игнорировать определенные элементы, возвращаемые методом `Deconstruct`. Переменная с именем "_" представляет отмену. Одна операция деконструкции может включать одно или несколько отбрасываний.

Следующий пример деконструирует объект `Person` на четыре строки (имя и фамилию, город и штат), но отбрасывает фамилию и штат.

C#

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Методы деконструкции расширения

Если вы не являетесь создателем класса, структуры или интерфейса, вы все равно можете выполнять деконструкцию объектов этого типа, реализовав один или несколько `Deconstruct`методов расширения, которые будут возвращать интересующие вас значения.

В приведенном ниже примере определены два метода расширения `Deconstruct` для класса System.Reflection.PropertyInfo. Первый возвращает набор значений,

указывающих характеристики свойства. Второй метод показывает уровень доступа свойства. Логические значения указывают, имеет ли свойство отдельные методы получения и задания доступа или разные специальные возможности. Если есть только один метод доступа или как метод получения, так и набор доступа имеют одинаковые специальные возможности, `access` переменная указывает на доступность свойства в целом. В противном случае доступность методов чтения и записи указывается переменными `getAccess` и `setAccess`.

C#

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool
hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string
setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;

        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod;
```

```
        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
```

```
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.
{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item
property: public
```

Метод расширения для системных типов

Некоторые системные типы предоставляют `Deconstruct` метод в качестве удобства. Например, тип System.Collections.Generic.KeyValuePair<TKey,TValue> предоставляет эту функцию. При итерации по System.Collections.Generic.Dictionary<TKey,TValue>каждый элемент является `KeyValuePair<TKey, TValue>` и может быть деконструенен. Рассмотрим следующий пример:

C#

```
Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
```

```
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November
10th, 2021.");
}
```

`record` Типы

При объявлении типа record с помощью двух позиционных параметров или более компилятор создает метод `Deconstruct` с параметром `out` для каждого позиционного параметра в объявлении `record`. Дополнительные сведения см. в разделах Позиционный синтаксис для определения свойства и Поведение деконструктора в производных записях.

См. также

- Объявление переменных деконструкция (правило стиля IDE0042)
- Пустые переменные
- Типы кортежей

Исключения и обработка исключений

Статья • 26.03.2025

Функции обработки исключений языка C# помогают справиться с непредвиденными или исключительными ситуациями, возникающими при выполнении программы. Обработка исключений использует `try`, `catch` и ключевые слова `finally` для проверки действий, которые могут не выполниться, для обработки сбоев при принятии решения о том, что это разумно сделать, и для очистки ресурсов после этого. Исключения могут создаваться средой CLR, библиотеками .NET или сторонними библиотеками или кодом приложения. Исключения создаются с помощью ключевого слова `throw`.

Во многих случаях исключение может быть вызвано не методом, который ваш код вызвал напрямую, а другим методом в стеке вызовов. При возникновении исключения среда CLR раскроет стек, ищет метод с блоком `catch` для конкретного типа исключения, и будет выполнять первый такой `catch` блок, который он находит. Если он не находит подходящий блок `catch` где-либо в стеке вызовов, он завершит процесс и отобразит сообщение пользователю.

В этом примере метод проверяет деление по нулю и перехватывает ошибку. Без обработки исключений эта программа завершалась бы с ошибкой **DivideByZeroException был необработан**.

C#

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine($"{a} divided by {b} = {result}");
        }
```

```
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Обзор исключений

Исключения имеют следующие свойства:

- Исключения — это типы, производные от `System.Exception`.
- Используйте блок `try` вокруг инструкций, которые могут вызывать исключения.
- После возникновения исключения в блоке `try` поток управления переходит к первому связанному обработчику исключений, который присутствует в любом месте стека вызовов. В C#ключевое слово `catch` используется для определения обработчика исключений.
- Если обработчик исключений для данного исключения отсутствует, программа перестает выполняться с сообщением об ошибке.
- Не перехватывайте исключение, если не можете обработать его и оставить приложение в предсказуемом состоянии. Если вы поймаете `System.Exception`, повторно выбросьте его, используя ключевое слово `throw` в конце блока `catch`.
- Если блок `catch` определяет переменную исключений, ее можно использовать для получения дополнительных сведений о типе исключения, которое произошло.
- Исключения можно явно создать программой с помощью ключевого слова `throw`.
- Объекты исключений содержат подробные сведения об ошибке, такие как состояние стека вызовов и текстовое описание ошибки.
- Код в блоке `finally` выполняется независимо от того, возникает ли исключение. Используйте блок `finally` для освобождения ресурсов, например для закрытия всех потоков или файлов, открытых в блоке `try`.
- Управляемые исключения в .NET реализованы поверх структурированного механизма обработки исключений Win32. Дополнительные сведения см. в структурированной обработке исключений (C/C++) и кратком курсе по углубленному изучению структурированной обработки исключений Win32.

Спецификация языка C#

Дополнительные сведения см. в исключениях в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования C#.

См. также

- System.Exception
- инструкции обработки исключений
- Исключения

Использование исключений

17.06.2025

В C#ошибки в программе во время выполнения распространяются через программу с помощью механизма, называемого исключениями. Исключения вызываются кодом, который встречает ошибку, и перехватываются кодом, который может ее исправить. Исключения могут вызываться средой выполнения .NET или кодом в программе. После генерации исключения он распространяется по стеку вызовов до тех пор, пока `catch` не будет найдена инструкция для исключения. Необработанные исключения обрабатываются универсальным обработчиком исключений, предоставляемым системой, в котором отображается диалоговое окно.

Исключения представлены классами, производными от Exception. Этот класс определяет тип исключения и содержит свойства, содержащие сведения об исключении. Создание исключения включает создание экземпляра класса, производного от класса исключения, при необходимости настраивая свойства исключения, а затем выбрасывание объекта с помощью ключевого слова `throw`. Рассмотрим пример.

C#

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

После возникновения исключения среда выполнения проверяет текущую инструкцию, чтобы узнать, находится ли она в блоке `try` . Если это так, все `catch` блоки, связанные с блоком `try`, проверяются, чтобы определить, способны ли они перехватывать исключение. `Catch` блоки обычно указывают типы исключений; Если тип `catch` блока совпадает с типом исключения или базовым классом исключения, `catch` блок может обрабатывать метод. Рассмотрим пример.

C#

```
try
{
    TestThrow();
}
```

```
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

Если инструкция, вызывающая исключение, не находится в пределах блока `try` или если блок `try`, заключающий её, не имеет соответствующего блока `catch`, среда выполнения проверяет вызывающий метод на наличие инструкции `try` и блоков `catch`. Среда выполнения продолжает двигаться вверх по стеку вызовов, выполняя поиск совместимого блока `catch`. После обнаружения и выполнения блока `catch` управление передается следующему оператору после этого блока `catch`.

Оператор `try` может содержать более одного блока `catch`. Первая `catch` инструкция, которая может обрабатывать исключение, выполняется. Все следующие `catch` инструкции, даже если они совместимы, игнорируются. Расположите блоки catch от самых конкретных (или более производных) до наименее конкретных. Рассмотрим пример.

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
```

```
            }
            Console.WriteLine("Done");
        }
    }
}
```

Перед выполнением `catch` блока среда выполнения проверяет наличие `finally` блоков. `Finally` блоки позволяют программисту устранить любое неоднозначное состояние, которое может быть оставлено из прекращённого `try` блока, или освободить внешние ресурсы (например, графические дескрипторы, подключения к базам данных или потоки файлов) без ожидания завершения объектов сборщиком мусора в среде выполнения. Рассмотрим пример.

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

Если `WriteByte()` выдало исключение, код в блоке `try`, который пытается повторно открыть файл, завершится ошибкой, если `file.Close()` не будет вызван, и файл останется заблокированным. Так как `finally` блоки выполняются даже при возникновении

исключения, `finally` блок в предыдущем примере позволяет правильно закрыть файл и избежать ошибки.

Если в стеке вызовов не найден совместимый `catch` блок после исключения, происходит одно из трех событий.

- Если исключение находится в методе завершения, метод завершения прерван, а базовый метод завершения, если таковой есть, вызывается.
- Если стек вызовов содержит статический конструктор или инициализатор статических полей, происходит исключение TypeInitializationException, причем исходное исключение присваивается свойству InnerException нового исключения.
- Если достигается начало потока, поток завершается.

Обработка исключений (руководство по программированию на C#)

17.06.2025

Блок try используется программистами C# для разделения кода, который может быть затронут исключением. Связанные блоки перехвата используются для обработки всех результирующих исключений. Finally блок содержит код, который выполняется независимо от того, возникает ли исключение в блоке `try`, например, освобождение ресурсов, выделенных в блоке `try`. Для `try` блока требуется один или несколько связанных `catch` блоков, или `finally` блок или оба блока.

В следующих примерах показаны оператор `try-catch`, оператор `try-finally` и оператор `try-catch-finally`.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
```

```
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

Блок `try`, не имеющий блока `catch` или блока `finally`, вызывает ошибку компилятора.

Перехват блоков

Блок `catch` может указать тип исключения для перехвата. Спецификация типа называется *фильтром исключений*. Тип исключения должен быть производным от Exception. Как правило, не указывайте Exception в качестве фильтра исключений, если вы не знаете, как обрабатывать все исключения, которые могут быть вызваны в `try` блоке, или если вы добавили инструкцию throw в конце блока `catch`.

Несколько `catch` блоков с разными классами исключений можно объединить. Блоки `catch` вычисляются сверху вниз в коде, но для каждого исключения, которое создается, выполняется только один `catch` блок. `catch` Первый блок, указывающий точный тип или базовый класс создаваемого исключения, выполняется. Если ни один блок `catch` не указывает соответствующий класс исключений, будет выбран блок `catch`, который не имеет типа, если такой блок присутствует в инструкции. Важно сначала разместить `catch` блоки с наиболее конкретными классами исключений (т. е. наиболее производными).

Перехватывайте исключения, если выполняются следующие условия:

- У вас есть хорошее представление о том, почему исключение может быть создано, и вы можете реализовать определенное восстановление, например предложить пользователю ввести новое имя файла при перехвате FileNotFoundException объекта.
- Вы можете создать и бросить новое, более конкретное исключение.

  C#

  ```
  int GetInt(int[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (IndexOutOfRangeException e)
  ```

```
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- Вы хотите частично обработать исключение перед его передачей для дальнейшей обработки. В следующем примере блок `catch` используется для добавления записи в журнал ошибок перед перебросом исключения.

```
C#

try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Можно также указать *фильтры исключений* , чтобы добавить логическое выражение в предложение catch. Фильтры исключений указывают, что конструкция catch срабатывает только в том случае, если условие истинно. В следующем примере оба предложения catch используют один и тот же класс исключений, но для создания другого сообщения об ошибке проверяется дополнительное условие:

```
C#

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

Фильтр исключений, который всегда возвращается `false` , можно использовать для проверки всех исключений, но не обработки их. Обычно используется журнал исключений:

C#

```
public class ExceptionFilter
{
    public static void Main()
    {
        try
        {
            string? s = null;
            Console.WriteLine(s.Length);
        }
        catch (Exception e) when (LogException(e))
        {
        }
        Console.WriteLine("Exception must have been handled");
    }

    private static bool LogException(Exception e)
    {
        Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
        Console.WriteLine($"\tMessage: {e.Message}");
        return false;
    }
}
```

Метод `LogException` всегда возвращает `false`, ни одно предложение с использованием этого фильтра исключений не соответствует. Предложение catch может быть общим, используя System.Exception, и последующие предложения могут обрабатывать более конкретные классы исключений.

Наконец, блоки

Блок `finally` позволяет очистить действия, выполняемые в блоке `try` . Если блок `finally` присутствует, он выполняется последним, после блока `try` и любого соответствующего блока `catch`. `finally` блок всегда выполняется: возникает ли исключение или обнаружен `catch` блок, соответствующий типу исключения.

Этот `finally` блок можно использовать для освобождения таких ресурсов, как потоки файлов, подключения к базе данных и графические дескрипторы, без ожидания завершения объектов сборщиком мусора в среде выполнения.

В следующем примере `finally` блок используется для закрытия файла, открываемого в блоке `try` . Обратите внимание, что состояние дескриптора файла проверяется до закрытия файла. `try` Если блок не может открыть файл, дескриптор файла по-прежнему имеет значение `null`, и `finally` блок не пытается закрыть его. Вместо этого, если файл успешно открыт в блоке `try` , `finally` блок закрывает открытый файл.

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе "Исключения " и инструкции try в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

См. также

- Справочник по C#
- инструкции обработки исключений
- инструкции using

Создавать и выбрасывать исключения

Статья • 27.03.2025

Исключения используются для указания того, что при запуске программы произошла ошибка. Объекты исключений, описывающие ошибку, создаются, а затем *выбрасываются* с помощью инструкции throw или выражения. Затем среда выполнения выполняет поиск наиболее совместимого обработчика исключений.

Программисты должны создавать исключения, если одно или несколько следующих условий являются истинными:

- Метод не может осуществить свою определенную функциональность. Например, если параметр метода имеет недопустимое значение:

  C#

  ```
  static void CopyObject(SampleClass original)
  {
      _ = original ?? throw new ArgumentException("Parameter cannot be
  null", nameof(original));
  }
  ```

- Неуместный вызов объекта выполняется на основе состояния объекта. Одним из примеров может быть попытка записи в файл, доступный только для чтения. В случаях, когда состояние объекта не разрешает операцию, бросьте экземпляр InvalidOperationException или объект, основанный на производной от этого класса. Следующий код является примером метода, который создает объект InvalidOperationException:

  C#

  ```
  public class ProgramLog
  {
      FileStream logFile = null!;
      public void OpenLog(FileInfo fileName, FileMode mode) { }

      public void WriteLog()
      {
          if (!logFile.CanWrite)
          {
              throw new InvalidOperationException("Logfile cannot be
  read-only");
          }
          // Else write data to the log and return.
  ```

```
    }
}
```

- Если аргумент метода вызывает исключение. В этом случае исходное исключение должно быть поймано, и необходимо создать экземпляр ArgumentException. Исходное исключение должно быть передано конструктору ArgumentException в качестве параметра InnerException:

```C#
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

> ⓘ **Примечание**
>
> В предыдущем примере показано, как использовать свойство `InnerException`. Это намеренно упрощено. На практике необходимо проверить, находится ли индекс в диапазоне перед его использованием. Можно применять эту технику для оборачивания исключения в случаях, когда член параметра выбрасывает исключение, которое невозможно было предвидеть перед вызовом.

Исключения содержат свойство с именем StackTrace. Эта строка содержит имя методов в текущем стеке вызовов вместе с именем файла и номером строки, в котором исключение было создано для каждого метода. Объект StackTrace создается автоматически средой CLR из точки инструкции `throw`, поэтому исключения должны быть вызваны с точки начала трассировки стека.

Все исключения содержат свойство с именем Message. Эта строка должна быть задана, чтобы объяснить причину исключения. Сведения, конфиденциальные для безопасности, не должны быть помещены в текст сообщения. Помимо Message, ArgumentException содержит свойство с именем ParamName, которое должно быть задано в качестве имени аргумента, вызвавшего исключение. В сеттере свойства ParamName должно быть установлено значение `value`.

Общедоступные и защищенные методы создают исключения всякий раз, когда они не могут завершить свои предполагаемые функции. Создаваемый класс исключений — это наиболее конкретное исключение, которое соответствует условиям ошибки. Эти исключения должны быть задокументированы как часть функциональных возможностей класса, а производные классы или обновления исходного класса должны сохранять то же поведение для обратной совместимости.

Вещи, которых следует избегать при генерации исключений

В следующем списке указаны практики, которых следует избегать при выбрасывании исключений.

- Не используйте исключения для изменения потока программы в рамках обычного выполнения. Используйте исключения для создания отчетов и обработки условий ошибок.
- Исключения не следует возвращать как возвращаемое значение или параметр, вместо того чтобы выбрасываться.
- Не выбрасывайте System.Exception, System.SystemException, System.NullReferenceExceptionили System.IndexOutOfRangeException из собственного исходного кода намеренно.
- Не создавайте исключения, которые можно создавать в режиме отладки, но не в режиме выпуска. Чтобы выявлять ошибки во время выполнения на этапе разработки, используйте Debug Assert.

Исключения в методах возврата задач

Методы, объявленные модификатором `async`, имеют некоторые особые аспекты, когда речь идет об исключениях. Исключения, создаваемые в методе `async`, хранятся в возвращаемой задаче и не возникают до тех пор, пока, например, задача ожидается. Дополнительные сведения о хранимых исключениях см. в разделе Асинхронные исключения.

Рекомендуется проверить аргументы и вызвать все соответствующие исключения, например ArgumentException и ArgumentNullException, перед вводом асинхронных частей методов. То есть эти исключения проверки должны возникать синхронно перед началом работы. В следующем фрагменте кода показан пример, в котором при возникновении исключений ArgumentException исключения будут синхронно возникать, в то время как InvalidOperationException будут храниться в возвращаемой задаче.

C#

```
// Non-async, task-returning method.
// Within this method (but outside of the local function),
// any thrown exceptions emerge synchronously.
public static Task<Toast> ToastBreadAsync(int slices, int toastTime)
{
    if (slices is < 1 or > 4)
    {
        throw new ArgumentException(
            "You must specify between 1 and 4 slices of bread.",
            nameof(slices));
    }

    if (toastTime < 1)
    {
        throw new ArgumentException(
            "Toast time is too short.", nameof(toastTime));
    }

    return ToastBreadAsyncCore(slices, toastTime);

    // Local async function.
    // Within this function, any thrown exceptions are stored in the task.
    static async Task<Toast> ToastBreadAsyncCore(int slices, int time)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        // Start toasting.
        await Task.Delay(time);

        if (time > 2_000)
        {
            throw new InvalidOperationException("The toaster is on fire!");
        }

        Console.WriteLine("Toast is ready!");

        return new Toast();
    }
}
```

Определение классов исключений

Программы могут создавать предопределенный класс исключений в пространстве имен System (за исключением тех, где ранее было указано), или создавать собственные классы исключений, производные от Exception. Производные классы должны определять по крайней мере три конструктора: один конструктор без

параметров, один из них задает свойство сообщения и задает свойства Message и InnerException. Например:

```
C#
```

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
base(message, inner) { }
}
```

Добавьте новые свойства в класс исключений, когда предоставленные данные полезны для разрешения исключения. Если новые свойства добавляются в производный класс исключений, `ToString()` следует переопределить, чтобы вернуть добавленную информацию.

Спецификация языка C#

Дополнительные сведения см. в исключениях и операторе throw в Спецификации языка C#. Спецификация языка является окончательным источником для синтаксиса И использования C#.

См. также

- Иерархия Исключений

Исключения, создаваемые компилятором

17.06.2025

Некоторые исключения создаются автоматически средой выполнения .NET при сбое базовых операций. Эти исключения и их условия ошибки перечислены в следующей таблице.

Развернуть таблицу

Исключение	Описание
ArithmeticException	Базовый класс для исключений, возникающих во время арифметических операций, таких как DivideByZeroException и OverflowException.
ArrayTypeMismatchException	Возникает, если массив не может хранить заданный элемент, так как фактический тип элемента несовместим с фактическим типом массива.
DivideByZeroException	Выбрасывается при попытке разделить целочисленное значение на ноль.
IndexOutOfRangeException	Возникает при попытке индексировать массив, если индекс меньше нуля или за пределами границ массива.
InvalidCastException	Выбрасывается при ошибке явного преобразования из базового типа в интерфейс или производный тип во время выполнения программы.
NullReferenceException	Возникает при попытке ссылаться на объект, значение которого равно NULL.
OutOfMemoryException	Возникает, когда попытка выделить память с помощью нового оператора завершается ошибкой. Это исключение означает, что память, доступная для среды CLR, была исчерпана.
OverflowException	Вызывается при переполнении арифметической операции в контексте `checked`.
StackOverflowException	Возникает при исчерпании стека выполнения из-за слишком большого количества невыполненных вызовов методов; обычно связано с очень глубокой или бесконечной рекурсией.
TypeInitializationException	Возникает, когда статический конструктор вызывает исключение, и не существет совместимого `catch` обработчика для его перехвата.

См. также

- инструкции обработки исключений

Правила и соглашения об именовании идентификаторов C#

10.10.2025

Идентификатор — это имя, которое вы назначаете типу (классу, интерфейсу, структуре, делегату или перечислению), члену, переменной или пространству имен.

В этой статье рассматриваются основные правила для допустимых идентификаторов C# и соглашений об именовании, используемых для создания согласованного профессионального кода.

Правила именования

Допустимые идентификаторы должны соответствовать этим правилам. Компилятор C# выдает ошибку для любого идентификатора, который не соответствует этим правилам:

- Идентификаторы должны начинаться с буквы или подчеркивания (`_`).
- Идентификаторы могут содержать буквенные символы Юникода, десятичные цифры, соединительные символы Юникода, комбинируемые символы Юникода или символы форматирования Юникода. Дополнительные сведения о категориях Юникода см. в базе данных категорий Юникода.

Можно объявить идентификаторы, соответствующие ключевым словам C#, с помощью префикса `@` идентификатора. `@` не является частью имени идентификатора. Например, `@if` объявляет идентификатор с именем `if`. Эти дословные идентификаторы в основном предназначены для взаимодействия с идентификаторами, объявленными другими языками.

Полное определение допустимых идентификаторов см. в статье Идентификаторов в спецификации языка C#.

> **Важно!**
>
> спецификация языка C# допускает только букву (Lu, Ll, Lt, Lm или Nl), цифру (Nd), подключение (Pc), объединение (Mn или Mc) и категории форматирования (Cf). Все, что находится вне этого, автоматически заменяется с помощью `_`. Это может повлиять на определенные символы Юникода.

Соглашения об именовании

Помимо правил, соглашения об именах идентификаторов используются во всех API .NET. Эти соглашения обеспечивают согласованность имен, но компилятор не применяет их. Вы свободны в использовании различных соглашений в ваших проектах.

По соглашению, в программах на C# `PascalCase` используется для имен типов, пространств имен и всех общедоступных элементов. Кроме того, команда `dotnet/docs` использует следующие соглашения, принятые из стиля программирования команды среды выполнения .NET:

- Имена интерфейсов начинаются с `I` заглавной буквы.
- Типы атрибутов заканчиваются словом `Attribute`.
- Типы перечисления используют единственное существительное для нефлагов и множественное существительное для флагов.
- Идентификаторы не должны содержать два последовательных символа подчеркивания (`_`) . Эти имена зарезервированы для идентификаторов, созданных компилятором.
- Используйте значимые и описательные имена для переменных, методов и классов.
- Предпочитайте ясность вместо краткости.
- Используйте PascalCase для имен классов и имен методов.
- Используйте camelCase для параметров метода и локальных переменных.
- Используйте PascalCase для имен констант, как полей, так и локальных констант.
- Поля приватного экземпляра начинаются с подчеркивания (`_`), а оставшийся текст оформлен в верблюжем регистре.
- Статические поля начинаются с `s_`. Это соглашение не является поведением Visual Studio по умолчанию, а также не частью рекомендаций по проектированию Framework, но настраивается в редакторе.
- Избегайте использования аббревиаций или акронимов в именах, за исключением широко известных и принятых аббревиаций.
- Используйте значимые и описательные пространства имен, которые соответствуют нотации обратного доменного имени.
- Выберите имена сборок, представляющие основную цель сборки.

- Избегайте использования однобуквенных имен переменных, за исключением простых счетчиков циклов. Кроме того, в примерах синтаксиса, описывающих синтаксис конструкций C#, часто используются следующие однобуквные имена, соответствующие соглашению, используемому в спецификации языка C#. Примеры синтаксиса являются исключением из правила.
  - Используйте `S` для структур, `C` для классов.
  - Используйте `M` для методов.
  - Используйте `v` для переменных, `p` для параметров.
  - Используйте `r` для параметров `ref`.

> **Совет**
>
> Вы можете применять соглашения об именовании, касающиеся заглавных и строчных букв, префиксов, суффиксов и разделителей слов с помощью правил именования в стиле написания кода .

В следующих примерах рекомендации, относящиеся к элементам, обозначенным как `public`, также применимы при работе с элементами `protected` и `protected internal`, которые должны быть видимы внешним вызывающим.

PascalCase

При именовании типа `class`, `interface`, `struct` или `delegate` используйте Pascal-регистр (PascalCasing).

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

При именовании `interface` используйте регистр pascal в дополнение к префиксу имени с `I`. Этот префикс четко указывает потребителям, что это `interface`.

C#

```
public interface IWorkerQueue
{
}
```

При именовании членов `public` таких типов, как поля, свойства, события, используйте стиль ПаскальКейс. Кроме того, используйте pascal casing для всех методов и локальных функций.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

При написании позиционных записей используйте регистр pascal для параметров, так как они — общедоступные свойства записи.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

Для получения дополнительной информации о позиционных записях см. позиционный синтаксис для определения свойств.

Дело верблюда

При именовании полей `private` или `internal` используйте верблюжий стиль ("camelCasing") и добавляйте префикс `_`. Используйте стиль CamelCase при именовании локальных переменных, включая экземпляры типа делегата.

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

> **Совет**
>
> При редактировании кода C#, который следует этим соглашениям об именовании, в интегрированной среде разработки, поддерживающей автодополнение инструкций, ввод `_` будет отображать всех членов области видимости объекта.

При работе с полями `static`, которые `private` или `internal`, используйте префикс `s_`, а для статических потоков — префикс `t_`.

C#

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

При написании параметров метода используйте регистр верблюда.

```C#
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Параметры основного конструктора

Имена параметров первичного конструктора зависят от объявленного типа.

- Для типов `class` и `struct`: используйте camelCase, согласующийся с другими параметрами метода.

```C#
public class DataService(IWorkerQueue workerQueue, ILogger logger)
{
    public void ProcessData()
    {
        // Use the parameters directly
        logger.LogInformation("Processing data");
        workerQueue.Enqueue("data");
    }
}
```

```C#
public struct Point(double x, double y)
{
    public double Distance => Math.Sqrt(x * x + y * y);
}
```

- Для `record` типов: используйте регистр Pascal, так как параметры становятся общедоступными свойствами.

```C#
public record Person(string FirstName, string LastName);
public record Address(string Street, string City, string PostalCode);
```

Дополнительные сведения о первичных конструкторах см. в разделе "Основные конструкторы".

Дополнительные сведения о соглашениях об именовании C#см. в стиле программирования команды среды выполнения .NET.

Рекомендации по именованию параметров типа

Следующие рекомендации применяются к параметрам типа для параметров универсального типа. Параметры типа — это заполнители аргументов в универсальном типе или универсальном методе. Вы можете узнать больше о параметрах универсального типа в руководстве по программированию на C#.

- **Назначайте** параметрам универсального типа описательные имена, если только односимвольное имя полностью не объясняет их значение, и описательное имя не добавит дополнительной ценности.

  ```
  C#

  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **рассмотрим** использование `T` в качестве имени параметра типа для типов с одним параметром типа буквы.

  ```
  C#

  public int IComparer<T>() => 0;
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Используйте префикс "T" для имен параметров описательного типа**.

  ```
  C#

  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **рассмотрим**, указывающие ограничения, помещенные в параметр типа в имени параметра. Например, параметр, ограниченный `ISession`, может назваться `TSession`.

Правило анализа кода CA1715 можно использовать для обеспечения соответствующего именованности параметров типа.

Дополнительные правила именования

- Примеры, не содержащие с помощью директив, используйте квалификацию пространства имен. Если вы знаете, что пространство имен импортируется по умолчанию в проекте, вам не нужно полностью указать имена из этого пространства имен. Полные имена можно разделить после точки (.), если они слишком длинные для одной строки, как показано в следующем примере.

  C#

  ```
  var currentPerformanceCounterCategory = new System.Diagnostics.
      PerformanceCounterCategory();
  ```

- Вам не нужно изменять имена объектов, созданных с помощью средств конструктора Visual Studio, чтобы они соответствовали другим рекомендациям.

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Общие соглашения о коде C#

Статья • 15.01.2025

Соглашения по программированию необходимы для поддержания удобочитаемости кода, согласованности и совместной работы в команде разработчиков. Код, который соответствует отраслевым методикам и установленным рекомендациям, проще понимать, поддерживать и расширять. Большинство проектов применяют согласованный стиль с помощью соглашений о коде. И dotnet/docs dotnet/samples проекты не являются исключением. В этой серии статей вы узнаете о наших соглашениях по программированию и средствах, которые мы используем для их применения. Вы можете принять наши соглашения как есть или изменить их в соответствии с потребностями вашей команды.

Мы выбрали наши соглашения на основе следующих целей:

1. *Правильность*. Наши примеры копируются и вставляются в приложения. Мы ожидаем, что поэтому нам нужно сделать код устойчивым и правильным, даже после нескольких изменений.
2. *Обучение*. Цель наших примеров — научить все .NET и C#. По этой причине мы не размещаем ограничения на какие-либо языковые функции или API. Вместо этого эти примеры учат, когда функция является хорошим выбором.
3. *Согласованность. Читатели* ожидают согласованный интерфейс в нашем содержимом. Все образцы должны соответствовать одному и тому же стилю.
4. *Внедрение*: мы агрессивно обновляем наши примеры, чтобы использовать новые языковые функции. Эта практика повышает осведомленность о новых функциях и делает их более знакомыми для всех разработчиков C#.

> **ⓘ Важно!**
>
> Эти рекомендации используются корпорацией Майкрософт для разработки примеров и документации. Они были приняты из **правил среды выполнения .NET, стиля** написания кода C# и **компилятора C# (roslyn).** Мы выбрали эти рекомендации из-за их внедрения в течение нескольких лет разработки с открытым кодом. Эти рекомендации помогают членам сообщества участвовать в проектах среды выполнения и компилятора. Они предназначены быть примером общепринятых соглашений C#, а не авторитетным списком (см. **«Руководства по проектированию фреймворков»** для получения подробных рекомендаций).
>
> Цели *обучения* и внедрения *определяют, почему соглашение о кодировании документов отличается от соглашений среды выполнения и* компилятора.

> Среда выполнения и компилятор имеют строгие метрики производительности для горячих путей. Многие другие приложения не делают. Наша *цель обучения* требует, чтобы мы не запрещали какую-либо конструкцию. Вместо этого примеры показывают, когда должны использоваться конструкции. Мы обновляем примеры более агрессивно, чем большинство рабочих приложений. Наша *цель внедрения* требует, чтобы мы отображали код, который вы должны написать сегодня, даже если код, написанный в прошлом году, не нуждается в изменениях.

В этой статье описываются наши рекомендации. Рекомендации развиваются со временем, и вы найдете примеры, которые не соответствуют нашим рекомендациям. Мы приветствуем PR, которые приносят эти образцы в соответствие или проблемы, которые обращают внимание на примеры, которые мы должны обновить. Наши рекомендации — это Открытый исходный код, и мы приветствуем PR и проблемы. Однако если ваша отправка изменит эти рекомендации, сначала откройте вопрос для обсуждения. Вы можете использовать наши рекомендации или адаптировать их к вашим потребностям.

Средства и анализаторы

Средства могут помочь вашей команде обеспечить соблюдение соглашений. Вы можете включить анализ кода для применения предпочитаемого правила. Вы также можете создать конфигурацию редактора , чтобы Visual Studio автоматически применяла рекомендации по стилю. В качестве отправной точки можно скопировать dotnet/docs*.editorconfig* , чтобы использовать наш стиль.

Эти средства упрощают работу вашей команды по принятию предпочитаемых рекомендаций. Visual Studio применяет правила во всех файлах *.editorconfig* в области для форматирования кода. Можно использовать несколько конфигураций для применения корпоративных соглашений, соглашений о команде и даже детализированных соглашений о проекте.

Анализ кода создает предупреждения и диагностику при обнаружении нарушений правил. Вы настраиваете правила, примененные к проекту. Затем каждая сборка CI уведомляет разработчиков, когда они нарушают какие-либо правила.

Идентификаторы диагностики

- Выбор соответствующих диагностических идентификаторов при создании собственных анализаторов

Рекомендации по использованию языка

В следующих разделах описаны рекомендации, которые команда документации .NET следует для подготовки примеров кода и примеров. Как правило, следуйте приведенным ниже рекомендациям.

- По возможности используйте современные языковые функции и версии C#.
- Избегайте устаревших конструкций языка.
- Перехватывайте только те исключения, которые можно правильно обработать; избегайте перехвата общих исключений. Например, пример кода не должен перехватывать тип System.Exception без фильтра исключений.
- Используйте определенные типы исключений для предоставления значимых сообщений об ошибках.
- Используйте запросы и методы LINQ для обработки коллекций, чтобы улучшить удобочитаемость кода.
- Используйте асинхронное программирование с асинхронным и ожиданием операций с привязкой ввода-вывода.
- Будьте осторожны с взаимоблокировками и используйте Task.ConfigureAwait при необходимости.
- Используйте ключевые слова языка для типов данных вместо типов среды выполнения. Например, используйте `string` вместо System.Stringнее или `int` вместо System.Int32. Эта рекомендация включает использование типов `nint` и `nuint`.
- Используйте `int` вместо неподписанных типов. Использование `int` часто используется на C#, и при использовании `int`проще взаимодействовать с другими библиотеками. Исключения предназначены для документации, конкретной для типов данных без знака.
- Используйте `var` только в том случае, если средство чтения может вывести тип из выражения. Читатели просматривают наши примеры на платформе документов. У них нет подсказок по наведении указателя мыши или инструментов, отображающих тип переменных.
- Напишите код с четкостью и простотой.
- Избегайте чрезмерно сложной и запутанной логики кода.

Более конкретные рекомендации следуют.

Строковые данные

- Для сцепления коротких строк рекомендуется использовать интерполяцию строк, как показано в следующем коде.

```
```

```
C#
```

```
string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- Для добавления строк в циклах, особенно при работе с текстами больших размеров, используйте объект System.Text.StringBuilder.

```
C#
```

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Предпочитайте необработанные строковые литералы вместо экранированных последовательностей или дословных строк.

```
C#
```

```
var message = """
    This is a long message that spans across multiple lines.
    It uses raw string literals. This means we can
    also include characters like \n and \t without escaping them.
    """;
```

- Используйте интерполяцию строк на основе выражений, а не интерполяцию позиционных строк.

```
C#
```

```
// Execute the queries.
Console.WriteLine("scoreQuery:");
foreach (var student in scoreQuery)
{
    Console.WriteLine($"{student.Last} Score: {student.score}");
}
```

Конструкторы и инициализация

- Используйте вариант Pascal для основных параметров конструктора для типов записей:

C#

```
public record Person(string FirstName, string LastName);
```

- Используйте запись в стиле CamelCase для основных параметров конструктора в типах классов и структур.
- Используйте свойства `required` вместо конструкторов для принудительной инициализации значений свойств:

C#

```
public class LabelledContainer<T>(string label)
{
    public string Label { get; } = label;
    public required T Contents
    {
        get;
        init;
    }
}
```

Массивы и коллекции

- Используйте выражения коллекции для инициализации всех типов коллекций:

C#

```
string[] vowels = [ "a", "e", "i", "o", "u" ];
```

Делегаты

- Используйте Func<> и Action<> вместо определения типов делегатов. В классе определите метод делегата.

C#

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Вызывайте метод с помощью сигнатуры, которую определяет делегат `Func<>` или `Action<>`.

```
C#

actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- Если вы создаете экземпляры типа делегата, используйте сокращенный синтаксис. В классе определите тип делегата и метод с соответствующей сигнатурой.

```
C#

public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- Создайте экземпляр типа делегата и вызовите его. В следующем объявлении используется сокращенный синтаксис.

```
C#

Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- В следующем объявлении используется полный синтаксис.

```
C#

Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

Операторы `try-catch` и `using` при обработке исключений

- Рекомендуется использовать оператор try-catch для обработки большей части исключений.

```C#
static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmeticException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        throw;
    }
}
```

- Использование оператора C# using упрощает код. При наличии оператора try-finally, код которого в блоке `finally` содержит только вызов метода Dispose, вместо него рекомендуется использовать оператор `using`.

  В следующем примере оператор `try-finally` вызывает `Dispose` только в блоке `finally`.

```C#
Font bodyStyle = new Font("Arial", 10.0f);
try
{
    byte charset = bodyStyle.GdiCharSet;
}
finally
{
    bodyStyle?.Dispose();
}
```

  То же самое можно сделать с помощью оператора `using`.

```C#
using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}
```

Используйте новый using синтаксис , который не требует фигурных скобок:

```
C#
```

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

Операторы `&&` и `||`

- Используйте && вместо того, &|| чтобы вместо | выполнения сравнений, как показано в следующем примере.

```
C#
```

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor) is var result)
{
    Console.WriteLine("Quotient: {0}", result);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Если делитель равен нулю, второе условие в операторе `if` вызовет ошибку времени выполнения. При этом оператор && сразу прекращает выполнение, если первое выражение ложно. Это означает, что второе выражение не будет вычисляться. Оператор & всегда вычисляет оба выражения, и если значение `divisor` равно 0, возникает ошибка времени выполнения.

Оператор `new`

- Используйте одну из кратких форм создания экземпляра объекта, если тип переменной соответствует типу объекта, как показано в следующих определениях. Эта форма не является допустимой, если переменная является типом интерфейса или базовым классом типа среды выполнения.

```
C#
```

```
var firstExample = new ExampleClass();
```

```
C#
ExampleClass instance2 = new();
```

Предыдущие объявления эквивалентны следующему объявлению.

```
C#
ExampleClass secondExample = new ExampleClass();
```

- Используйте инициализаторы объектов, чтобы упростить создание объектов, как показано в следующем примере.

```
C#
var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

В следующем примере задаются точно такие же свойства, как и в предыдущем, но без использования инициализаторов.

```
C#
var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

Обработка событий

- Используйте лямбда-выражение для определения обработчика событий, который не требуется удалить позже:

```
C#
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
```

```
        };
}
```

Лямбда-выражение сокращает приведенное ниже традиционное определение.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Статические участники

Для вызова статических членов следует использовать имя класса: *ClassName.StaticMember*. В этом случае код становится более удобочитаемым за счет четкого доступа. Не присваивайте статическому элементу, определенному в базовом классе, имя производного класса. Хотя этот код компилируется, удобочитаемость кода вводит в заблуждение, и код может прерваться в будущем, если добавить статический член с тем же именем в производный класс.

Запросы LINQ

- Используйте значимые имена для переменных запроса. В следующем примере используется `seattleCustomers` для клиентов, находящихся в Сиэтле.

  C#

  ```
  var seattleCustomers = from customer in Customers
                         where customer.City == "Seattle"
                         select customer.Name;
  ```

- Рекомендуется использовать псевдонимы для уверенности в том, что в именах свойств анонимных типов верно используются прописные буквы при помощи правил использования прописных и строчных букв языка Pascal.

  C#

```
var localDistributors =
    from customer in Customers
    join distributor in Distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Переименуйте свойства, если имена свойств в результате могут быть неоднозначными. Например, если запрос возвращает имя клиента и имя распространителя, вместо того чтобы оставлять их в виде `Name` в результате, переименуйте их, чтобы уточнить, `CustomerName` — это имя клиента, а `DistributorName` — имя распространителя.

```
C#

var localDistributors2 =
    from customer in Customers
    join distributor in Distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorName =
distributor.Name };
```

- Рекомендуется использовать неявное типизирование в объявлении переменных запроса и переменных диапазона. Это руководство по неявным вводу в запросах LINQ переопределяет общие правила для неявно типизированных локальных переменных. Запросы LINQ часто используют проекции, которые создают анонимные типы. Другие выражения запросов создают результаты с вложенными универсальными типами. Неявные типизированные переменные часто доступны для чтения.

```
C#

var seattleCustomers = from customer in Customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Выровняйте предложения запросов в соответствии с from предложением, как показано в предыдущих примерах.

- Используйте where предложения перед другими предложениями запросов, чтобы гарантировать, что более поздние предложения запросов работают на сокращенном отфильтрованном наборе данных.

```
C#
```

```
var seattleCustomers2 = from customer in Customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Доступ к внутренним коллекциям с несколькими условиями `from` вместо условия join. Например, коллекция объектов `Student` может содержать коллекцию результатов тестирования. При выполнении следующего запроса возвращается каждая оценка, которая превышает 90, а также имя семьи учащегося, который получил оценку.

```
C#

var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Неявно типизированные локальные переменные

- Используйте неявное ввод для локальных переменных, если тип переменной очевиден с правой стороны назначения.

```
C#

var message = "This is clearly a string.";
var currentTemperature = 27;
```

- Не используйте var , если тип не отображается в правой части назначения. Не полагайтесь на то, что имя метода предоставляет информацию о типе. Тип переменной считается понятным, если это оператор `new`, явное приведение или присваивание литерального значения.

```
C#

int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- Не используйте имена переменных для указания типа переменной. Имя может быть неверным. Вместо этого используйте тип, чтобы указать тип и использовать имя переменной, чтобы указать семантические сведения переменной. Следующий пример должен использоваться `string` для типа и

что-то подобное `iterations` , чтобы указать значение информации, считываемой из консоли.

```
C#

var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Рекомендуется избегать использования `var` вместо dynamic. Используйте `dynamic`, если требуется определение типа во время выполнения. Дополнительные сведения см. в статье Использование типа dynamic (руководство по программированию на C#).

- Используйте неявное ввод для переменной цикла в for циклах.

  В следующем примере неявное типизирование используется в операторе `for`.

```
C#

var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Не используйте неявное ввод, чтобы определить тип переменной цикла в foreach циклах. В большинстве случаев тип элементов в коллекции не сразу очевиден. Имя коллекции не должно быть исключительно зависеть от вывода типа его элементов.

  В следующем примере явное типизирование используется в операторе `foreach`.

```
C#

foreach (char ch in laugh)
{
    if (ch == 'h')
    {
        Console.Write("H");
    }
    else
    {
        Console.Write(ch);
```

```
    }
}
Console.WriteLine();
```

- используйте неявный тип для последовательностей результатов в запросах LINQ. В разделе LINQ объясняется, что многие запросы LINQ приводят к анонимным типам, в которых должны использоваться неявные типы. Другие запросы приводят к вложенным универсальным типам, где `var` более удобочитаемым.

  > ⓘ **Примечание**
  >
  > Следите за тем, чтобы случайно не изменить тип элемента итерируемой коллекции. Например, легко переключаться с System.Linq.IQueryable на System.Collections.IEnumerable в инструкции `foreach`, которая изменяет выполнение запроса.

Некоторые из наших примеров объясняют *естественный тип* выражения. Эти примеры должны использовать `var` , чтобы компилятор выбрал естественный тип. Несмотря на то, что эти примеры менее очевидны, для примера требуется использование `var` . Текст должен объяснить поведение.

Объявления пространства имен в пределах файла

Большинство файлов кода объявляют одно пространство имен. Поэтому в наших примерах следует использовать объявления пространства имен, ограниченные файлом.

C#

```
namespace MySampleCode;
```

Размещение директив using за пределами объявления пространства имен

Если директива `using` находится за пределами объявления пространства имен, импортированное пространство имен является его полным именем. Полное имя является более понятным. Если директива `using` находится внутри пространства имен, она может быть либо относительно этого пространства имен, либо его полное имя.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Предположим, что класс имеет ссылку (прямую или непрямую).WaitUntil

Теперь давайте немного изменим его:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

И он компилируется сегодня. И завтра. Но когда-нибудь на следующей неделе предыдущий код (нетронутый) завершается сбоем с двумя ошибками:

Консоль

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

Одна из зависимостей, введенная в этот класс в пространстве имен, заканчивается на `.Azure`.

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

Директива, размещенная `using` в пространстве имен, учитывает контекст и усложняет разрешение имен. В этом примере это первое пространство имен, которое он находит.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Добавление нового пространства имен, соответствующего либо `CoolStuff.Azure` `CoolStuff.AwesomeFeature.Azure` совпадающего с глобальным `Azure` пространством имен. Его можно устранить, добавив `global::` модификатор в `using` объявление. Однако вместо этого проще размещать `using` объявления за пределами пространства имен.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Рекомендации по стилю

В общем случае используйте следующий формат для примеров кода:

- Используйте четыре пробела для отступа. Не используйте вкладки.
- Согласованное выравнивание кода для повышения удобства чтения.

- Ограничить строки до 65 символов, чтобы повысить удобочитаемость кода в документах, особенно на мобильных экранах.
- Улучшайте ясность и взаимодействие с пользователем, разбивая длинные инструкции на несколько строк.
- Используйте стиль Allman для фигурных скобок: открытый и закрывающий фигурные скобки своей собственной новой линии. Фигурные скобки соответствуют текущему уровню отступа.
- Разрывы строк должны возникать перед двоичными операторами при необходимости.

Стиль комментариев

- Используйте одно строковый комментарий (`//`) для кратких объяснений.

- Избегайте нескольких строковый комментарий (`/* */`) для более длинных объяснений.
  Комментарии в примерах кода не локализованы. Это означает, что объяснения, внедренные в код, не переводятся. Более длинный, пояснительный текст должен быть помещен в компаньон статью, чтобы его можно было локализовать.

- Для описания методов, классов, полей и всех общедоступных элементов используются xml-комментарии.

- Комментарий размещается на отдельной строке, а не в конце строки кода.

- Текст комментария начинается с заглавной буквы.

- Текст комментария завершается точкой.

- Вставьте одно пространство между разделителем комментариев (`//`) и текстом комментария, как показано в следующем примере.

  C#

  ```
  // The following declaration creates a query. It does not run
  // the query.
  ```

Соглашения о макете

Чтобы выделить структуру кода и облегчить чтение кода, в хорошем макете используется форматирование. Примеры и образцы корпорации Майкрософт соответствуют следующим соглашениям.

- Использование параметров редактора кода по умолчанию (логичные отступы, отступы по четыре символа, использование пробелов для табуляции). Дополнительные сведения см. в разделе "Параметры", "Текстовый редактор", C#, "Форматирование".
- Запись только одного оператора в строке.
- Запись только одного объявления в строке.
- Если строки продолжения не отступываются автоматически, отступите их на одну остановку табуляции (четыре пробела).
- Добавление по крайней мере одной пустой строки между определениями методов и свойств.
- Использование скобок для ясности предложений в выражениях, как показано в следующем коде.

```
C#
```

```
if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

Исключения возникают, когда в примере объясняется приоритет оператора или выражения.

Безопасность

Следуйте указаниям, изложенным в правилах написания безопасного кода.

Руководство. Создание программ C# на основе файлов

Приложения на основе файлов — это программы, содержащиеся в одном `*.cs` файле, который вы создаете и запускаете без соответствующего файла проекта (`*.csproj`). Приложения на основе файлов идеально подходят для обучения C#, так как они имеют меньшую сложность: вся программа хранится в одном файле. Приложения на основе файлов также полезны для создания служебных программ командной строки. На платформах Unix можно запускать приложения на основе файлов с помощью `#!` директив (shebang). Изучив это руководство, вы:

- ✓ Создайте файловую программу.
- ✓ Добавьте поддержку Unix Shebang (`#!`).
- ✓ Чтение аргументов командной строки.
- ✓ Обработка стандартных входных данных.
- ✓ Запись выходных данных искусства ASCII.
- ✓ Обработка аргументов командной строки.
- ✓ Используйте проанализированные результаты командной строки.
- ✓ Протестируйте окончательное приложение.

Вы создаете файловую программу, которая записывает текст как искусство ASCII. Приложение содержится в одном файле, использует пакеты NuGet и реализует основные функции.

Предпосылки

- Пакет SDK для .NET 10. Скачайте его с сайта загрузки .NET.
- Visual Studio Code. Скачайте его на домашней странице Visual Studio Code.
- (Необязательно) Расширение C# DevKit для Visual Studio Code. Скачайте его из Visual Studio Code Marketplace.

Создание файловой программы

1. Откройте Visual Studio Code и создайте новый файл с именем `AsciiArt.cs`. Введите следующий текст:

   C#

   ```
   Console.WriteLine("Hello, world!");
   ```

2. Сохраните файл. Затем откройте интегрированный терминал в Visual Studio Code и введите следующее:

```
Интерфейс командной строки.NET
dotnet AsciiArt.cs
```

При первом запуске этой программы `dotnet` хост создает исполняемый файл из вашего исходного файла, сохраняет артефакты сборки во временной папке, а затем запускает созданный исполняемый файл. Этот интерфейс можно проверить, введя `dotnet AsciiArt.cs` еще раз. На этот раз `dotnet` хост определяет, что исполняемый файл является актуальным и запускает его без повторной сборки. Выходные данные сборки не отображаются.

В предыдущих шагах показано, что приложения на основе файлов не являются файлами скриптов. Это исходные файлы C#, которые `dotnet` хост создает с помощью сгенерированного файла проекта во временной папке. Одна из строк выходных данных, отображаемых при сборке программы, должна выглядеть примерно так (в Windows):

```
Интерфейс командной строки.NET
AsciiArt succeeded (7.3s) → AppData\Local\Temp\dotnet\runfile\AsciiArt-85c58ae0cd68371711f06f297fa0d7891d0de82afde04d8c64d5f910ddc04ddc\bin\debug\AsciiArt.dll
```

На платформах Unix выходная папка похожа на следующее:

```
Интерфейс командной строки.NET
AsciiArt succeeded (7.3s) → Library/Application Support/dotnet/runfile/AsciiArt-85c58ae0cd68371711f06f297fa0d7891d0de82afde04d8c64d5f910ddc04ddc/bin/debug/AsciiArt.dll
```

Эти выходные данные сообщают о том, где размещаются временные файлы и выходные данные сборки. В этом руководстве в любое время, когда вы редактируете исходный файл, `dotnet` узел обновляет исполняемый файл перед его запуском.

Приложения на основе файлов — это обычные программы C#. Единственное ограничение заключается в том, что их необходимо записать в один исходный файл. Операторы верхнего уровня или классический `Main` метод можно использовать как точку входа. Можно объявить любые типы: классы, интерфейсы и структуры. Алгоритмы можно структурировать в файловой программе так же, как и в любой программе C#. Можно даже объявить несколько пространств имен для упорядочивания кода. Если файловая программа становится слишком большой для одного файла, ее можно преобразовать в

проектную программу и разделить исходный код на несколько файлов. Приложения на основе файлов — это отличный инструмент прототипа. Вы можете начать экспериментировать с минимальными затратами, чтобы доказать концепции и алгоритмы сборки.

Поддержка Unix Shebang (`#!`)

> ⓘ **Примечание.**
>
> Поддержка директив `#!` применяется только на платформах Unix. Для Windows нет аналогичной директивы для непосредственного выполнения программы C#. В Windows необходимо использовать `dotnet` в командной строке.

В Unix запустите приложения на основе файлов непосредственно, введя только имя исходного файла. Вместо использования `dotnet AsciiArt.cs` введите имя исходного файла в командной строке. Необходимо внести два изменения:

1. Задайте разрешения *на выполнение* исходного файла:

   Bash

   ```
   chmod +x AsciiArt.cs
   ```

2. Добавьте директиву shebang (`#!`) в качестве первой строки `AsciiArt.cs` файла:

   C#

   ```
   #!/usr/local/share/dotnet/dotnet
   ```

Расположение `dotnet` может отличаться в разных установках Unix. Используйте команду `which dotnet`, чтобы найти хост в вашей `dotnet` среде.

Кроме того, можно использовать `#!/usr/bin/env dotnet` для автоматического разрешения пути dotnet из переменной среды PATH:

C#

```
#!/usr/bin/env dotnet
```

После внесения этих двух изменений вы можете запустить программу из командной строки напрямую:

Bash

```
./AsciiArt.cs
```

Если вы предпочитаете, вы можете удалить расширение, чтобы вместо этого ввести `./AsciiArt` его. Вы можете добавить его в исходный `#!` файл, даже если используется Windows. Командная строка Windows не поддерживается `#!`, но компилятор C# разрешает директиву в приложениях на основе файлов на всех платформах.

Чтение аргументов командной строки

Теперь напишите все аргументы в командной строке в выходные данные.

1. Замените текущее содержимое `AsciiArt.cs` следующим кодом:

   C#

   ```
   if (args.Length > 0)
   {
       string message = string.Join(' ', args);
       Console.WriteLine(message);
   }
   ```

2. Чтобы запустить эту версию, введите следующую команду:

   Интерфейс командной строки.NET

   ```
   dotnet AsciiArt.cs -- This is the command line.
   ```

   Параметр `--` указывает, что все следующие аргументы команд должны передаваться в программу AsciiArt. Аргументы `This is the command line.` передаются в виде массива строк, где каждая строка — одно слово: `This`, `is`, `the` и `command` `line.`.

Эта версия демонстрирует следующие новые понятия:

- Предопределенная переменная `args` передает аргументы командной строки программе. Переменная `args` представляет собой массив строк: `string[]` Если длина `args` равна 0, аргументы не были предоставлены. В противном случае каждое слово в списке аргументов хранится в соответствующей записи в массиве.
- Метод string.Join объединяет несколько строк в одну строку с указанным разделителем. В этом случае разделитель является одним пробелом.
- Console.WriteLine записывает строку в стандартную выходную консоль, за которой следует новая строка.

Обработка стандартных входных данных

Предыдущий код правильно обрабатывает аргументы командной строки. Теперь добавьте код для обработки входных данных из стандартных входных данных (`stdin`) вместо аргументов командной строки.

1. Добавьте следующее `else` предложение в инструкцию `if` , добавленную в предыдущем коде:

   C#

   ```
   else
   {
       while (Console.ReadLine() is string line && line.Length > 0)
       {
           Console.WriteLine(line);
       }
   }
   ```

   Предыдущий код считывает входные данные консоли до тех пор, пока не будет прочитана пустая строка или чтение `null` . (Метод возвращает, Console.ReadLine `null` если входной поток закрыт, введя `ctrl+C` .)

2. Проверьте стандартные входные данные, создав текстовый файл в той же папке. Назовите файл `input.txt` и добавьте следующие строки:

   txt

   ```
   Hello from ...
   dotnet!

   You can create
   file-based apps
   in .NET 10 and
   C# 14

   Have fun writing
   useful utilities
   ```

   Оставьте строки короткими, чтобы они правильно форматируются при добавлении функции для использования искусства ASCII.

3. Снова запустите программу.

   С помощью bash:

   Bash

```
cat input.txt | dotnet AsciiArt.cs
```

Или с помощью PowerShell:

PowerShell

```
Get-Content input.txt | dotnet AsciiArt.cs
```

Теперь программа может принимать аргументы командной строки или стандартные входные данные.

Написать ASCII-арт

Затем добавьте пакет, поддерживающий искусство ASCII, Colorful.Console. Чтобы добавить пакет в файловую программу, используйте директиву `#:package` .

1. Добавьте следующую директиву после директивы `#!` в `AsciiArt.cs` файле:

   C#

   ```
   #:package Colorful.Console@1.2.15
   ```

   > **Внимание**
   >
   > Последняя версия `1.2.15` пакета была последней `Colorful.Console` версией пакета, когда это руководство было последнее обновление. Проверьте страницу NuGet пакета для последней версии, чтобы убедиться, что вы используете версию пакета с последними исправлениями безопасности.

2. Измените строки, вызываемые `Console.WriteLine` методом `Colorful.Console.WriteAscii` , вместо этого:

   C#

   ```
   Colorful.Console.WriteAscii(line);
   ```

3. Запустите программу. Вы видите вывод ASCII-арта вместо отображаемого текста.

Параметры команды обработки

Затем добавьте синтаксический анализ командной строки. Текущая версия записывает каждое слово в виде другой строки выходных данных. Аргументы командной строки, которые вы добавляете, поддерживают две функции:

1. Процитировать несколько слов, которые должны быть написаны на одной строке:

   Интерфейс командной строки.NET

   ```
   AsciiArt.cs "This is line one" "This is another line" "This is the last line"
   ```

2. Добавьте параметр для приостановки между каждой `--delay` строкой:

   Интерфейс командной строки.NET

   ```
   AsciiArt.cs --delay 1000
   ```

Пользователи могут использовать оба аргумента вместе.

Большинство приложений командной строки должны анализировать аргументы командной строки для эффективной обработки параметров, команд и ввода пользователем. Библиотека System.CommandLine предоставляет комплексные возможности для обработки команд, вложенных команд, параметров и аргументов. Вы можете сосредоточиться на том, что делает ваше приложение, а не механика синтаксического анализа входных данных командной строки.

Библиотека `System.CommandLine` предлагает несколько ключевых преимуществ:

- Автоматическое создание и проверка текста справки.
- Поддержка соглашений о командной строке POSIX и Windows.
- Встроенные возможности завершения вкладок.
- Согласованное поведение синтаксического анализа между приложениями.

1. `System.CommandLine` Добавьте пакет. Добавьте эту директиву после существующей директивы пакета:

   C#

   ```
   #:package System.CommandLine@2.0.0
   ```

   > **ⓘ Внимание**
   >
   > Версия была последней версией `2.0.0` , когда это руководство было последнее обновление. Если доступна более новая версия, используйте последнюю версию, чтобы обеспечить наличие последних пакетов безопасности.

> Проверьте [страницу NuGet](#) пакета для последней версии, чтобы убедиться, что вы используете версию пакета с последними исправлениями безопасности.

2. Добавьте необходимые инструкции using в верхней части файла (после `#!` и `#:package` директив):

```C#
using System.CommandLine;
using System.CommandLine.Parsing;
```

3. Определите параметр задержки и аргумент сообщений. Добавьте следующий код для создания `CommandLine.Option` и `CommandLine.Argument` объектов для представления параметра командной строки и аргумента:

```C#
Option<int> delayOption = new("--delay")
{
    Description = "Delay between lines, specified as milliseconds.",
    DefaultValueFactory = parseResult => 100
};

Argument<string[]> messagesArgument = new("Messages")
{
    Description = "Text to render."
};
```

   В приложениях командной строки параметры обычно начинаются с `--` (двойной дефис) и могут принимать аргументы. Параметр `--delay` принимает целый аргумент, указывающий задержку в миллисекундах. Определяет `messagesArgument` , как все остальные маркеры после анализа параметров анализируются как текст. Каждый маркер становится отдельной строкой в массиве, но текст может быть кавычек, чтобы включить несколько слов в один маркер. Например, `"This is one message"` становится одним маркером, а `This is four tokens` становится четырьмя отдельными маркерами.

   Предыдущий код определяет тип аргумента для `--delay` параметра и что аргументы являются массивом значений `string` . Это приложение имеет только одну команду, поэтому вы используете *корневую команду*.

4. Создайте корневую команду и настройте ее с помощью параметра и аргумента. Добавьте аргумент и параметр в корневую команду:

C#

```
RootCommand rootCommand = new("Ascii Art file-based program sample");

rootCommand.Options.Add(delayOption);
rootCommand.Arguments.Add(messagesArgument);
```

5. Добавьте код для синтаксического анализа аргументов командной строки и обработки ошибок. Этот код проверяет аргументы командной строки и сохраняет проанализированные аргументы в объекте System.CommandLine.ParseResult :

C#

```
ParseResult result = rootCommand.Parse(args);
foreach (ParseError parseError in result.Errors)
{
    Console.Error.WriteLine(parseError.Message);
}
if (result.Errors.Count > 0)
{
    return 1;
}
```

Предыдущий код проверяет все аргументы командной строки. Если проверка завершается ошибкой, приложение записывает ошибки в консоль и завершает работу.

Использование проанализированных результатов командной строки

Теперь завершите работу приложения для использования параметров синтаксического анализа и записи выходных данных. Сначала определите запись для хранения параметров синтаксического анализа. Приложения на основе файлов могут включать объявления типов, такие как записи и классы. Они должны быть после всех инструкций верхнего уровня и локальных функций.

1. record Добавьте объявление для хранения сообщений и значения параметра задержки:

C#

```
public record AsciiMessageOptions(string[] Messages, int Delay);
```

2. Добавьте следующую локальную функцию перед объявлением записи. Этот метод обрабатывает аргументы командной строки и стандартные входные данные и возвращает новый экземпляр записи:

C#

```
async Task<AsciiMessageOptions> ProcessParseResults(ParseResult result)
{
    int delay = result.GetValue(delayOption);
    List<string> messages = [.. result.GetValue(messagesArgument) ??
Array.Empty<string>()];

    if (messages.Count == 0)
    {
        while (Console.ReadLine() is string line && line.Length > 0)
        {
            Colorful.Console.WriteAscii(line);
            await Task.Delay(delay);
        }
    }
    return new([.. messages], delay);
}
```

3. Создайте локальную функцию для записи искусства ASCII с указанной задержкой. Эта функция записывает каждое сообщение в записи с указанной задержкой между каждым сообщением:

C#

```
async Task WriteAsciiArt(AsciiMessageOptions options)
{
    foreach (string message in options.Messages)
    {
        Colorful.Console.WriteAscii(message);
        await Task.Delay(options.Delay);
    }
}
```

4. Замените `if` предложение, написанного ранее, следующим кодом, который обрабатывает аргументы командной строки и записывает выходные данные:

C#

```
var parsedArgs = await ProcessParseResults(result);

await WriteAsciiArt(parsedArgs);
return 0;
```

Вы создали `record` тип, который предоставляет структуру для параметров и аргументов командной строки синтаксического анализа. Новые локальные функции создают экземпляр записи и используют запись для записи выходных данных искусства ASCII.

Тестирование окончательного приложения

Протестируйте приложение, выполнив несколько различных команд. Если у вас возникли проблемы, вот готовый пример для сравнения с тем, что вы создали:

C#

```
#!/usr/bin/env dotnet

#:package Colorful.Console@1.2.15
#:package System.CommandLine@2.0.0

using System.CommandLine;
using System.CommandLine.Parsing;

Option<int> delayOption = new("--delay")
{
    Description = "Delay between lines, specified as milliseconds.",
    DefaultValueFactory = parseResult => 100
};

Argument<string[]> messagesArgument = new("Messages")
{
    Description = "Text to render."
};

RootCommand rootCommand = new("Ascii Art file-based program sample");

rootCommand.Options.Add(delayOption);
rootCommand.Arguments.Add(messagesArgument);

ParseResult result = rootCommand.Parse(args);
foreach (ParseError parseError in result.Errors)
{
    Console.Error.WriteLine(parseError.Message);
}
if (result.Errors.Count > 0)
{
    return 1;
}

var parsedArgs = await ProcessParseResults(result);

await WriteAsciiArt(parsedArgs);
return 0;

async Task<AsciiMessageOptions> ProcessParseResults(ParseResult result)
{
    int delay = result.GetValue(delayOption);
    List<string> messages = [.. result.GetValue(messagesArgument) ??
Array.Empty<string>()];

    if (messages.Count == 0)
```

```
    {
        while (Console.ReadLine() is string line && line.Length > 0)
        {
            // <WriteAscii>
            Colorful.Console.WriteAscii(line);
            // </WriteAscii>
            await Task.Delay(delay);
        }
    }
    return new([.. messages], delay);
}

async Task WriteAsciiArt(AsciiMessageOptions options)
{
    foreach (string message in options.Messages)
    {
        Colorful.Console.WriteAscii(message);
        await Task.Delay(options.Delay);
    }
}

public record AsciiMessageOptions(string[] Messages, int Delay);
```

В этом руководстве вы узнали, как создать файловую программу, в которой вы создаете программу в одном файле C#. Эти программы не используют файл проекта и могут использовать директиву `#!` в системах Unix. Учащиеся могут создавать эти программы после работы с нашими онлайн-учебниками и перед созданием более крупных приложений на основе проектов. Приложения на основе файлов также являются отличной платформой для служебных программ командной строки.

Связанный контент

- Справочник по приложениям на основе файлов
- Оператор верхнего уровня
- Директивы препроцессора
- Новые возможности C# 14

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 12.01.2026

Практическое руководство. Отображение аргументов командной строки (Руководство по программированию на C#)

Статья • 08.06.2023

Аргументы, предоставляемые исполняемому файлу в командной строке, доступны через инструкции верхнего уровня или через необязательный параметр для метода `Main`. Аргументы предоставляются в форме массива строк. Каждый элемент массива содержит один аргумент. Пробелы между аргументами удаляются. Например, рассмотрим следующие вызовы из командной строки вымышленного исполняемого файла:

Развернуть таблицу

Входные данные в командной строке	Массив строк, передаваемых в метод Main
executable.exe a b c	"a" "b" "c"
executable.exe one two	"one" "two"
executable.exe "one two" three	"один два" "три"

> **Примечание**
>
> При выполнении приложения в Visual Studio аргументы командной строки можно указать на странице "Отладка" в конструкторе проектов.

Пример

Этот пример отображает аргументы командной строки, передаваемые в приложение командной строки. Показанные выходные данные — первая запись в таблице выше.

C#

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");

for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}

/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

См. также

- Обзор System.CommandLine
- Руководство. Начало работы с System.CommandLine

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Руководство. Создание приложения командной строки с помощью System.CommandLine

> **Совет**
>
> Эта статья является частью раздела **"Основы** ", написанного для разработчиков, которые знают по крайней мере один язык программирования и учат C#. Если вы не знакомы с программированием, начните с **начала работы**. Если вам нужен полный охват библиотеки, ознакомьтесь с **документацией по библиотеке System.CommandLine**.

Библиотека System.CommandLine обрабатывает синтаксический анализ командной строки, создание справки и проверку входных данных, чтобы сосредоточиться на логике приложения. В этом руководстве вы создадите интерфейс командной строки отслеживания задач, демонстрирующий основные понятия: команды, подкоманда, параметры и аргументы.

Изучив это руководство, вы:

- ✓ Создайте файловое приложение с помощью `System.CommandLine` пакета.
- ✓ Определите параметры и аргументы с типизированными значениями.
- ✓ Создавайте подкоманды и прикрепите к ним параметры и аргументы.
- ✓ Обработайте каждую подкоманда с помощью действия.
- ✓ Протестируйте приложение с различными входными данными командной строки.

Необходимые условия

- Установите пакет SDK для .NET 10 или более поздней версии.

Создание приложения

Начните с создания файловой программы C# и добавления `System.CommandLine` пакета.

1. Создайте файл с именем `TaskCli.cs` со следующим содержимым:

```
C#
#!/usr/bin/env dotnet
```

Строка `#!` (shebang) позволяет запускать файл непосредственно в системах Unix. На Windows запустите файл с помощью `dotnet run TaskCli.cs`.

2. Добавьте директиву `System.CommandLine` пакета и необходимые `using` инструкции:

C#

```
#:package System.CommandLine@2.0.0
```

C#

```
using System.CommandLine;
using System.CommandLine.Parsing;
using System.Text.Json;
```

> **ⓘ Внимание**
>
> Версия `2.0.0` является последней версией во время записи. Проверьте страницу NuGet пакета для последней версии, чтобы убедиться, что у вас есть последние исправления безопасности.

Общие сведения о структуре команд

Прежде чем писать любой код синтаксического анализа, рассмотрим, как выглядит интерфейс командной строки с точки зрения пользователя. Средство отслеживания задач поддерживает четыре операции:

Консоль

```
dotnet TaskCli.cs -- add "Write documentation" --priority High --due 2026-04-01
dotnet TaskCli.cs -- list --all
dotnet TaskCli.cs -- complete 3
dotnet TaskCli.cs -- remove 3
dotnet TaskCli.cs -- --verbose list
```

> **ⓘ Примечание.**
>
> Примеры выше показывают, что `--` после `TaskCli.cs` говорит dotnet run, что все остальные аргументы передаются вашему приложению, а не интерпретируются самим `dotnet` интерфейсом командной строки.

Каждая строка использует несколько концепций командной строки:

- **Подкоманда** — это команды, которые сообщают приложению, что делать. Средство отслеживания задач имеет четыре: `add`, `list`, `complete` и `remove`. Каждая подкоманда может определять собственные параметры.
- **Аргументы** — это позиционные значения, которые следуют подкомандам. В `add "Write documentation"` строке `"Write documentation"` — это аргумент, указывающий описание задачи. В `complete 3` число `3` является аргументом, указывающим идентификатор задачи.
- **Параметры** — это именованные значения с префиксом `--`. В `add --priority High --due 2026-04-01` и `--priority`, и `--due` являются вариантами со своими значениями. В `list --all`, параметре `--all` используется логический флаг, который не требует значения.
- **Глобальные параметры** применяются к каждому подкомандам. Опция `--verbose` определена в корневой команде `Recursive = true`, поэтому она работает с любой подкомандой. В `--verbose list`, подробный флаг отображается перед подкоманда, но `list --verbose` работает одинаково хорошо.

В следующих разделах вы создаете эти составные части по принципу 'снизу вверх'. Сначала вы определяете отдельные параметры (например `--priority` , и `--all`) и аргументы (например, описание и идентификатор задачи). Затем вы создадите четыре подкоманда и прикрепите соответствующие параметры и аргументы к каждому из них. Затем вы настроите действие для каждой подкоманды. Действие — это код, который выполняется при вызове этой команды пользователем. Наконец, вы собираете корневую команду, анализируете входные данные и вызываете соответствующее действие.

Более подробное описание концепций синтаксиса командной строки см. в обзоре синтаксиса командной строки.

Определение параметров и аргументов

Параметры представляют собой именованные значения, которые пользователи указывают с префиксом `--`. Аргументы представляют позиционные значения. Оба являются строго типизированными. `System.CommandLine` парсит входную строку в тип, который вы указали.

Библиотека `System.CommandLine` использует *универсальные типы* для обеспечения безопасности типов. При записи `Option<int>` `int` между угловыми скобками это *аргумент типа*. Он сообщает библиотеке, какой тип значения содержит параметр. Сам класс объявляет *параметр* `T` типа (как и в System.CommandLine.Option<T>), и при создании

экземпляра предоставляется конкретный тип. Библиотека анализирует входную строку пользователя и преобразует ее в этот тип автоматически. Если пользователь предоставляет `--delay abc` для `Option<int>`, `System.CommandLine` сообщает об ошибке синтаксического анализа, вместо передачи некорректных данных в ваш код. Вы увидите этот шаблон с `Option<bool>`, `Option<Priority>` `Option<DateOnly?>` и System.CommandLine.Argument<T> в следующих шагах. Дополнительные сведения об универсальных классах см. в разделе Generics.

1. Определите параметры. Каждый `Option<T>` указывает тип значения, имя и описание. Параметр `--priority` использует тип `enum`, и `System.CommandLine` автоматически сравнивает входные данные с допустимыми значениями перечисления.

   C#

   ```
   var verboseOption = new Option<bool>("--verbose")
   {
       Description = "Show detailed output",
       Recursive = true
   };

   var priorityOption = new Option<Priority>("--priority")
   {
       Description = "Task priority level",
       DefaultValueFactory = _ => Priority.Medium
   };

   var dueOption = new Option<DateOnly?>("--due")
   {
       Description = "Due date (uses current culture date format)"
   };

   var allOption = new Option<bool>("--all")
   {
       Description = "Include completed tasks"
   };
   ```

   Параметр `Recursive = true` в `--verbose` доступен для каждой подкоманды. `DefaultValueFactory` на `--priority` предоставляет значение по умолчанию, чтобы пользователи могли опустить параметр.

2. Определите аргументы. Каждый `Argument<T>` указывает тип значения и имя:

   C#

   ```
   var descriptionArgument = new Argument<string>("description")
   {
       Description = "Task description"
   };
   ```

```
var taskIdArgument = new Argument<int>("id")
{
    Description = "Task ID"
};
```

Команды сборки и вложенные команды

A System.CommandLine.Command представляет действие, которое пользователь может вызвать. Добавьте соответствующие параметры и аргументы в каждую команду, чтобы `System.CommandLine` знало, какие параметры куда принадлежат.

1. Создайте четыре подкоманды. Каждая команда получает собственное сочетание параметров и аргументов:

C#

```
var addCommand = new Command("add", "Add a new task")
{
    Arguments = { descriptionArgument },
    Options = { priorityOption, dueOption }
};
```

C#

```
var listCommand = new Command("list", "List all tasks")
{
    Options = { allOption }
};
```

C#

```
var completeCommand = new Command("complete", "Mark a task as complete")
{
    Arguments = { taskIdArgument }
};
```

C#

```
var removeCommand = new Command("remove", "Remove a task")
{
    Arguments = { taskIdArgument }
};
```

2. Соберите корневую команду. Это System.CommandLine.RootCommand точка входа для интерфейса командной строки. Добавьте глобальный `--verbose` параметр и все подкоманды.

```C#
var rootCommand = new RootCommand("A simple task tracker CLI")
{
    Options = { verboseOption },
    Subcommands = { addCommand, listCommand, completeCommand, removeCommand }
};
```

Автоматически созданный текст справки отображает корневое описание команды при запуске пользователем `TaskCli --help`.

Обработка команд с помощью действий

Каждой подкоманде нужно *действие*. Действие — это делегат , который запускается при вызове этой команды пользователем. Делегат — это тип, представляющий ссылку на метод. Здесь вы передаете лямбда-выражение (анонимная встроенная функция, определенная с `=>`) в качестве делегата. Вызвать `SetAction`, чтобы назначить каждое действие. Делегат получает ParseResult, который предоставляет доступ к проанализированным значениям через `GetValue`.

1. Задайте действие для `add` команды. Это действие вводит интерполяцию строк (`$"..."` строки, которые внедряют выражения в фигурные скобки), условный оператор (`?:`) и шаблон теста типа (`due is DateOnly dueDate`), который проверяет, имеет ли значение null значение и назначает его новой переменной на одном шаге:

```C#
addCommand.SetAction(parseResult =>
{
    var description = parseResult.GetValue(descriptionArgument)!;
    var priority = parseResult.GetValue(priorityOption);
    var due = parseResult.GetValue(dueOption);
    var verbose = parseResult.GetValue(verboseOption);

    var tasks = LoadTasks();
    var id = tasks.Count > 0 ? tasks.Max(t => t.Id) + 1 : 1;
    var task = new TaskItem(id, description, priority, due, false);
    tasks.Add(task);
    SaveTasks(tasks);

    Console.WriteLine($"Added task {id}: {description}");
    if (verbose)
```

```
    {
        Console.WriteLine($"  Priority: {priority}");
        if (due is DateOnly dueDate)
        {
            Console.WriteLine($"  Due: {dueDate}");
        }
    }
});
```

2. Задайте действие для `list` команды. LINQ (языковой интегрированный запрос) предоставляет стандартные операторы запросов для коллекций в памяти. В этом действии Enumerable.Where фильтрует задачи только по элементам, соответствующим условию, и Enumerable.ToList материализует отфильтрованную последовательность в список. Затем действие использует foreach цикл для итерации результатов и условного оператора для выбора символа состояния:

C#

```
listCommand.SetAction(parseResult =>
{
    var showAll = parseResult.GetValue(allOption);
    var verbose = parseResult.GetValue(verboseOption);

    var tasks = LoadTasks();
    var filtered = showAll ? tasks : tasks.Where(t => !t.IsComplete).ToList();

    if (filtered.Count == 0)
    {
        Console.WriteLine("No tasks found.");
        return;
    }

    foreach (var task in filtered)
    {
        var status = task.IsComplete ? "✓" : " ";
        Console.WriteLine($"  [{status}] {task.Id}: {task.Description}");
        if (verbose)
        {
            Console.WriteLine($"       Priority: {task.Priority}");
            if (task.Due is DateOnly dueDate)
            {
                Console.WriteLine($"       Due: {dueDate}");
            }
        }
    }
});
```

Дополнительные сведения см. в разделе LINQ.

3. Задайте действие для `complete` команды. Это действие использует Enumerable.FirstOrDefault объекта LINQ для поиска:

   - Соответствующая задача.
   - Шаблонis null для проверки наличия задачи.
   - with Выражение для создания нового экземпляра записи путем копирования существующих значений сначала, а затем применения свойств, заданных в `with` инициализаторе (здесь). `IsComplete = true` Записи неизменяемы по умолчанию, поэтому этот шаблон копирования и обновления является способом создания измененного значения.

   Так как действие может завершиться ошибкой (например, идентификатор задачи не существует), действие возвращает целый код ошибки, который становится кодом выхода приложения:

   C#

   ```
   completeCommand.SetAction(parseResult =>
   {
       var id = parseResult.GetValue(taskIdArgument);
       var verbose = parseResult.GetValue(verboseOption);

       var tasks = LoadTasks();
       var task = tasks.FirstOrDefault(t => t.Id == id);

       if (task is null)
       {
           Console.Error.WriteLine($"Task {id} not found.");
           return -1;
       }

       tasks[tasks.IndexOf(task)] = task with { IsComplete = true };
       SaveTasks(tasks);

       Console.WriteLine($"Completed task {id}: {task.Description}");
       if (verbose)
       {
           Console.WriteLine($"  Priority: {task.Priority}");
       }
       return 0;
   });
   ```

4. Задайте действие для `remove` команды. Это действие следует тому же шаблону поиска и проверки, что и `complete`. Используется `FirstOrDefault` для поиска задачи и `is null` обработки отсутствующих случаев:

   C#

```
removeCommand.SetAction(parseResult =>
{
    var id = parseResult.GetValue(taskIdArgument);
    var verbose = parseResult.GetValue(verboseOption);

    var tasks = LoadTasks();
    var task = tasks.FirstOrDefault(t => t.Id == id);

    if (task is null)
    {
        Console.Error.WriteLine($"Task {id} not found.");
        return -1;
    }

    tasks.Remove(task);
    SaveTasks(tasks);

    Console.WriteLine($"Removed task {id}: {task.Description}");
    if (verbose)
    {
        Console.WriteLine($"  Priority: {task.Priority}");
    }
    return 0;
});
```

5. Выполните анализ командной строки и вызовите соответствующее действие:

```
C#

return rootCommand.Parse(args).Invoke();
```

`rootCommand.Parse(args)` анализирует входные данные в ParseResult, а `.Invoke()` выполняет действие для соответствующей команды. Возвращаемое значение — это код выхода (0 для успешного выполнения).

Добавление вспомогательных типов и вспомогательных данных

Приложению требуется несколько вспомогательных частей: локальные функции, enum, record и контекст сериализации. В следующих разделах представлены все понятия, объясните, почему вы выберете его и покажите код. Приложение использует System.Text.Json для хранения задач в формате JSON (нотация объектов JavaScript). Приложения на основе файлов требуют, чтобы объявления типов отображались после всех инструкций верхнего уровня и локальных функций.

1. Добавьте локальные функции, которые загружают и сохраняют задачи. Локальная функция — это метод, объявленный внутри другой функции, в том числе внутри других локальных функций. Локальные функции сохраняют вспомогательные логики близко к коду, который вызывает его, что повышает удобочитаемость, так как читатель не должен переходить к отдельному классу или файлу для понимания потока. `LoadTasks` и `SaveTasks` инкапсулируют операции ввода-вывода файлов, которые используют несколько команд, так что приложение записывает логику загрузки и сохранения однократно, а затем повторно использует её.

C#

```
static string GetTaskFilePath() => Path.Combine(
    Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
    "taskcli-sample",
    "tasks.json");

static List<TaskItem> LoadTasks()
{
    var path = GetTaskFilePath();
    if (!File.Exists(path))
    {
        return [];
    }

    var json = File.ReadAllText(path);
    return JsonSerializer.Deserialize(json,
TaskJsonContext.Default.ListTaskItem) ?? [];
}

static void SaveTasks(List<TaskItem> tasks)
{
    var path = GetTaskFilePath();
    Directory.CreateDirectory(Path.GetDirectoryName(path)!);
    var json = JsonSerializer.Serialize(tasks,
TaskJsonContext.Default.ListTaskItem);
    File.WriteAllText(path, json);
}
```

2. Добавьте перечисление `Priority` и запись `TaskItem` в конце файла.

C#

```
public enum Priority { Low, Medium, High }
```

Это enum тип значения, определяющий фиксированный набор именованных констант, поддерживаемых целочисленным типом. Вы можете представлять уровни приоритета с обычными целыми числами (0, 1, 2), но `enum` лучше выбрать по нескольким причинам: компилятор ограничивает назначения определенными

именами, поэтому опечатка, например `Hihg`, вызывает ошибку во время компиляции, а не молчаливую ошибку; имена `Low`, `Medium`, и `High` делают код более читаемым; и `System.CommandLine` автоматически проверяет ввод пользователя на соответствие элементам перечисления `Low`, `Medium`, и `High`, поэтому вы получаете бесплатную проверку входных данных.

C#

```
public record TaskItem(int Id, string Description, Priority Priority,
DateOnly? Due, bool IsComplete);
```

Это record тип, который компилятор снабжает свойством равенства на основе значений и безразрушающей мутацией. `record` подходит к `TaskItem` лучше, так как данные задач представляют собой простое состояние без сложного поведения — задачи сравниваются по значениям, а не по идентичности ссылок. Компилятор создает поддержку выражений `Equals`, `GetHashCode`, `ToString` и `with` на основе параметров конструктора, что позволяет проводить корректные проверки равенства, легко получать данные для отладки и осуществлять неизменяемые обновления, не прибегая к написанию шаблонного кода. Поскольку записи неизменяемы по умолчанию, вы используете `with` выражение для создания измененной копии, как это делает действие `complete`, а не изменяете исходный объект, что предотвращает непредвиденные эффекты при одновременном чтении и записи одного списка различными действиями.

3. Добавьте контекст сериализации JSON для сериализации, совместимой с AOT:

C#

```
[System.Text.Json.Serialization.JsonSourceGenerationOptions(WriteIndented =
true)]
[System.Text.Json.Serialization.JsonSerializable(typeof(List<TaskItem>))]
internal partial class TaskJsonContext :
System.Text.Json.Serialization.JsonSerializerContext;
```

Этот класс использует два атрибута (заметки метаданных, помещенные в квадратные скобки над объявлением). Атрибут [JsonSourceGenerationOptions(WriteIndented = true)] уведомляет генератору исходного кода выводить JSON с отступами для удобства чтения. Атрибут [JsonSerializable(typeof(List<TaskItem>))] сообщает генератору, для какого типа необходимо создать код сериализации. Вместе эти атрибуты обеспечивают сериализацию, созданную источником, которая позволяет избежать отражения во время выполнения и поддерживает компиляцию перед временем (AOT).

Тестирование приложения

Запустите приложение с различными входными данными, чтобы выполнить каждую подкоманда.

1. Просмотрите автоматически сгенерированную справку:

   Интерфейс командной строки.NET

   ```
   dotnet run TaskCli.cs -- --help
   ```

   Выходные данные

   ```
   Description:
     A simple task tracker CLI

   Usage:
     TaskCli [command] [options]

   Options:
     --verbose       Show detailed output
     -?, -h, --help  Show help and usage information
     --version       Show version information

   Commands:
     add <description>  Add a new task
     list               List all tasks
     complete <id>      Mark a task as complete
     remove <id>        Remove a task
   ```

2. Добавьте задачи с различными параметрами:

   Интерфейс командной строки.NET

   ```
   dotnet run TaskCli.cs -- add "Write documentation" --priority High --due 2026-04-01
   dotnet run TaskCli.cs -- add "Review pull request"
   dotnet run TaskCli.cs -- add "Fix build errors"
   ```

   Выходные данные

   ```
   Added task 1: Write documentation
   Added task 2: Review pull request
   Added task 3: Fix build errors
   ```

3. Вывод списка задач и использование `--verbose` для отображения дополнительных сведений:

Интерфейс командной строки.NET

```
dotnet run TaskCli.cs -- --verbose list
```

Выходные данные

```
  [ ] 1: Write documentation
       Priority: High
       Due: 4/1/2026
  [ ] 2: Review pull request
       Priority: Medium
  [ ] 3: Fix build errors
       Priority: Medium
```

4. Выполните задачу, а затем убедитесь, что по умолчанию в списке отображаются завершенные задачи.

Интерфейс командной строки.NET

```
dotnet run TaskCli.cs -- complete 2
dotnet run TaskCli.cs -- list
```

Выходные данные

```
Completed task 2: Review pull request
  [ ] 1: Write documentation
  [ ] 3: Fix build errors
```

5. Используйте `--all` для включения завершенных задач:

Интерфейс командной строки.NET

```
dotnet run TaskCli.cs -- list --all
```

Выходные данные

```
  [ ] 1: Write documentation
  [✓] 2: Review pull request
  [ ] 3: Fix build errors
```

6. Удаление задачи:

Интерфейс командной строки.NET

```
dotnet run TaskCli.cs -- remove 3
```

Выходные данные

```
Removed task 3: Fix build errors
```

Очистите ресурсы

Средство отслеживания задач хранит данные в JSON-файле в локальной папке данных приложения. Удалите папку `taskcli-sample` , чтобы удалить примеры данных:

- **Windows**: удаление `%LOCALAPPDATA%\taskcli-sample`.
- **macOS**: удаление `~/Library/Application Support/taskcli-sample`.
- **Linux**: удаление `~/.local/share/taskcli-sample`.

Связанный контент

- Обзор System.CommandLine
- Руководство. Начало работы с System.CommandLine
- Общие сведения о синтаксисе командной строки
- Руководство по проектированию System.CommandLine
- Руководство по созданию файловых приложений
- Отображение аргументов командной строки

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.03.2026

Руководство: выразите свой замысел с помощью ссылочных типов, допускающих и не допускающих значения NULL

> **Совет**
>
> **Впервые сталкиваетесь с типами-ссылками, допускающими значение NULL?** Сначала прочтите ссылочные типы, допускающие значения NULL. В этом руководстве предполагается, что вы понимаете разницу между ссылочными типами, не допускающими значения NULL, и ссылочными типами, допускающими значение NULL, а также то, как компилятор отслеживает состояние NULL.
>
> **Вы пришли из другого языка?** Если вы использовали nullable-типы в Kotlin, `strictNullChecks` в TypeScript или optionals в Swift, то концептуальная модель здесь та же. Здесь вы *узнаете, как выразить намерение проектирования*, а не изучать синтаксис.

В этом руководстве вы создадите небольшую библиотеку, которая моделирует опрос. В данных есть два различных сценария, которые можно различать с помощью ссылочных типов, допускающих значение NULL:

- *Вопрос опроса* всегда должен присутствовать. Список вопросов и текст каждого вопроса никогда не может быть `null`.
- *Ответ на вопрос* может отсутствовать. Респонденты могут отказаться от ответа на некоторые или все вопросы, и модель должна сделать это явным.

Эти правила объявляют с помощью ссылочных типов, не допускающих значение NULL, и ссылочных типов, допускающих значение NULL. Затем компилятор предупреждает, когда поведение кода не соответствует дизайну.

Изучив это руководство, вы:

- ✓ Создайте приложение.
- ✓ Создайте вопросы опроса.

- ✓ Создайте опрос.
- ✓ Проверьте требование без значения NULL.
- ✓ Создание типов ответов.
- ✓ Создание респондентов.
- ✓ Создайте один ответ на опрос.
- ✓ Создайте набор ответов на опросы.
- ✓ Изучите результаты опроса.

Три класса моделируют опрос:

- `SurveyQuestion`: один вопрос. Требуется текст и тип вопроса.
- `SurveyRun`: коллекция вопросов и список респондентов.
- `SurveyResponse`: ответы одного респондента, которые могут отсутствовать.

Каждый тип использует ссылочные типы, не допускающие значение NULL, для обязательных значений и ссылочных типов, допускающих значение NULL, для отсутствующих значений.

Prerequisites

- Последняя версия .NET SDK
- редактор Visual Studio Code
- C# DevKit

В этом руководстве предполагается, что вы знакомы с C# и Visual Studio или .NET CLI.

Создайте приложение и включите допускающие значение null ссылочные типы

Создайте консольное приложение с именем `NullableIntroduction`:

Интерфейс командной строки.NET

```
dotnet new console -n NullableIntroduction
cd NullableIntroduction
```

Создание вопросов опроса

Добавьте новый файл с именем `SurveyQuestion.cs` в проект и замените его содержимое следующим кодом. Текст и тип вопроса не могут иметь значение NULL, поэтому конструктор должен инициализировать оба:

C#

```
namespace NullableIntroduction;

public enum QuestionType
{
    YesNo,
    Number,
    Text
}

public class SurveyQuestion(QuestionType typeOfQuestion, string text)
{
    public string QuestionText { get; } = text;
    public QuestionType TypeOfQuestion { get; } = typeOfQuestion;
}
```

Параметры конструктора являются ненуляемыми ссылочными типами, поэтому компилятор предупреждает вызывающего объекта, если любой из аргументов может быть `null`.

Создание опроса вопросов

Затем добавьте новый файл с именем `SurveyRun.cs` в проект и определите `SurveyRun` класс для хранения списка вопросов:

C#

```
namespace NullableIntroduction;

public class SurveyRun
{
    private List<SurveyQuestion> surveyQuestions = [];

    public void AddQuestion(QuestionType type, string question) =>
        AddQuestion(new SurveyQuestion(type, question));

    public void AddQuestion(SurveyQuestion surveyQuestion) =>
        surveyQuestions.Add(surveyQuestion);
}
```

Поле `surveyQuestions` имеет тип `List<SurveyQuestion>`, не допускающий значения NULL. В нем используется выражение коллекции для инициализации пустого списка. Обе перегрузки `AddQuestion` принимают параметры, не допускающие значения null, поэтому компилятор гарантирует, что вызывающий код не передаёт `null`.

В `Program.cs` создайте `SurveyRun` и добавьте три вопроса:

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the
nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Проверка требования, не допускающего значение NULL

Чтобы узнать, как компилятор применяет параметры, не допускающие значение NULL, попробуйте добавить следующую строку и перестроить:

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Компилятор выдает предупреждение *CS8625*, так как `default` для ссылочного типа имеет значение `null`, а `AddQuestion` ожидает значение `string`, не допускающее NULL. Удалите строку перед продолжением.

Создать типы ответов

Респонденты могут отказаться от опроса, и даже если они участвуют, они могут пропустить отдельные вопросы. Оба вида "отсутствующие" являются допустимыми результатами, и система типов должна сделать их видимыми. Обе формы обозначаются с помощью `null`.

Добавьте новый файл с именем `SurveyResponse.cs` в проект и определите `SurveyResponse` класс. Используйте основной конструктор (параметры, объявленные в самом типе, доступные в теле) для записи всегда обязательных `Id` элементов:

C#

```
namespace NullableIntroduction;

public class SurveyResponse(int id)
{
    public int Id { get; } = id;
}
```

Создание респондентов

Добавьте *статический фабричный метод* (`static` метод, который создаёт и возвращает новый экземпляр типа, то есть служит альтернативой прямому вызову конструктора), создающий объекты respondent со случайным идентификатором:

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

Создание одного ответа на опрос

Затем добавьте метод, который запрашивает опрос респонденту. Сохраните ответы в словаре с возможностью значения null, чтобы сам тип показывал, что респондент может отказаться отвечать:

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
```

```
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

Поле `surveyResponses` равно `Dictionary<int, string>?`. Если вы разыменовываете поле, не проверив сначала значение на `null`, компилятор выдаст предупреждение. Внутри `AnswerSurvey`, компилятор отслеживает, что `surveyResponses` имеет значение *not-null* сразу после выражения `new`, поэтому тело цикла не требует дополнительных проверок.

Создание набора ответов на опросы

Добавьте в `SurveyRun` метод, который формирует список респондентов, пока не наберётся достаточно согласившихся на участие:

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = [];
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Поле `respondents`: `List<SurveyResponse>?` — оно `null`, пока опрос не запущен.

Звонок `PerformSurvey` из `Main`:

C#

```
surveyRun.PerformSurvey(50);
```

Изучение результатов опроса

Чтобы сообщать о результатах, экспортируйте несколько вспомогательных функций из `SurveyResponse` и `SurveyRun`. В `SurveyResponse` добавьте члены, реализованные с помощью выражений (члены, определённые с помощью `=>` и одного выражения вместо блока `{ ... }`), которые обрабатывают словарь, допускающий значение null:

C#

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No
answer";
```

`AnsweredSurvey` проверяет поле с `null`. `Answer` использует ?. оператор условного доступа по null-ссылке (который вычисляется как `null`, если левая часть имеет значение `null`, вместо выбрасывания исключения) для безопасного обращения по ссылке, а также ?? оператор объединения с null (который подставляет правый операнд, если левый имеет значение `null`), чтобы обеспечить запасное значение, отличное от null. Тип возвращаемого метода не допускает `string`значение NULL, поэтому вызывающие элементы не нуждаются в проверке null.

Добавьте `SurveyRun`элементы с выражением, предоставляющие список участников и вопросов:

C#

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

`AllParticipants` возвращает последовательность, не допускающую значение null, даже если `respondents` может быть `null`. Оператор `??` заменяет `Enumerable.Empty<SurveyResponse>()` , когда поле еще не заполнено. При удалении `??` предложения компилятор предупреждает, что метод может возвращать `null` , несмотря на тип возвращаемого значения, не допускающего значение NULL.

Наконец, напишите отчет в нижней части `Main`:

C#

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} :
{answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

Обратите внимание, что для `participant`, `surveyRun.Questions` или `surveyRun.GetQuestion(i)` не требуется проверка на null. Типы объявляют эти значения как не допускающие значение NULL, поэтому компилятор обрабатывает их как *ненулевое* значение во всем цикле.

Запустите приложение:

Интерфейс командной строки.NET

```
dotnet run
```

Выходные данные отличаются при каждом запуске, поскольку участники опроса генерируются случайным образом, но в каждой строке либо приводятся ответы участника, либо указывается, что он отказался отвечать.

Conclusion

Готовый пример находится в папке csharp/NullableIntroduction репозитория dotnet/samples . Поэкспериментируйте путем изменения типов, допускающих значение NULL и не допускающих значение NULL. Удаление `?` там, где по замыслу допускаются отсутствующие значения, приводит к предупреждениям компилятора, указывающим на каждое место, где отсутствие значения имеет значение.

Связанный контент

- Ссылочные типы, допускающие значение NULL

- Разрешение предупреждений, допускающих значение NULL
- Стратегии миграции, допускающие значение NULL
- Работа со ссылочными типами, допускающими значение NULL, в EF Core

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.05.2026

Исследуйте объектно-ориентированное программирование с помощью классов и объектов

10.10.2025

В этом руководстве вы создадите консольное приложение и увидите основные объектно-ориентированные функции, которые являются частью языка C#.

Предварительные условия

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Инструкции по установке

В Windows этот файл конфигурации WinGet используется для установки всех необходимых компонентов. Если у вас уже установлено что-то, WinGet пропустит этот шаг.

1. Скачайте файл и дважды щелкните его, чтобы запустить его.
2. Прочитайте лицензионное соглашение, введите `и`, и выберите `ввод` при появлении запроса на принятие.
3. Если на панели задач появится мигающий запрос контроля учетных записей пользователей (UAC), разрешите установку продолжить.

На других платформах необходимо установить каждый из этих компонентов отдельно.

1. Скачайте рекомендуемый установщик на странице загрузки пакета SDK для .NET и дважды щелкните его, чтобы запустить его. Страница загрузки обнаруживает платформу и рекомендует последний установщик для вашей платформы.
2. Скачайте последнюю версию установщика на домашней странице Visual Studio Code и дважды щелкните его, чтобы запустить его. Эта страница также обнаруживает платформу, а ссылка должна быть правильной для вашей системы.
3. Нажмите кнопку "Установить" на странице расширения C# DevKit. Откроется код Visual Studio и запрашивается, нужно ли установить или включить расширение. Выберите "Установить".

Создание приложения

С помощью окна терминала создайте каталог с именем *"Классы"*. Вы создадите приложение там. Перейдите в этот каталог и введите `dotnet new console` в окне консоли. При помощи этой команды создается приложение. Откройте *Program.cs*. Он должен выглядеть так:

```
C#
```

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

При работе с этим руководством вы создадите новые типы, представляющие банковский счет. Обычно разработчики определяют каждый класс в отдельном текстовом файле. Благодаря этому программой легче управлять, когда ее размер увеличивается. Создайте файл с именем *BankAccount.cs* в каталоге *классов* .

Этот файл содержит определение ***банковского счета***. Средства объектно-ориентированного программирования обеспечивают упорядочение кода. При этом создаются типы в виде ***классов***. Классы содержат код, который представляет отдельную сущность. Класс `BankAccount` представляет банковский счет. Этот код реализует определенные операции с помощью методов и свойств. В этом учебном пособии банковский счет поддерживает такое поведение:

1. Представляет собой число из 10 цифр, которое однозначно определяет банковский счет.
2. Содержит строку, в которой хранятся имена владельцев.
3. Баланс можно получить.
4. Принимает депозиты.
5. Принимает выводы средств.
6. Начальное сальдо должно было положительным.
7. Вывод не может привести к отрицательному балансу.

Определение типа банковского счета

Сначала можно создать основы класса, который определяет такой режим работы. Создайте новый файл с помощью команды **File:New**. Присвойте ему имя *BankAccount.cs*. Добавьте в файл *BankAccount.cs* следующий код:

```
C#
```

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

Прежде чем идти дальше, давайте рассмотрим то, что вы создали. Объявление `namespace` предоставляет способ логического упорядочения кода. Это руководство относительно небольшое, поэтому вы помещаете весь код в одно пространство имен.

`public class BankAccount` определяет класс или тип, который вы создаете. Весь код в скобках `{` и `}`, который следует за объявлением класса, определяет состояние и поведение класса. Есть пять ***элементов*** класса `BankAccount`. Первые три элемента представляют собой ***свойства***. Свойства являются элементами данных и могут содержать код для выполнения проверки или других правил. Последние два элемента являются ***методами***. Методы представляют собой блоки кода, которые выполняют только одну функцию. Чтение имен каждого из членов должно предоставить достаточно информации, чтобы вы или другой разработчик смогли понять, чем занимается класс.

Открытие нового счета

Сначала нужно открыть банковский счет. Когда клиент открывает счет, он должен указать начальное сальдо и сведения о владельцах этого счета.

Для создания нового объекта `BankAccount` типа требуется определить ***конструктор*** , который назначает эти значения. ***Конструктор*** — это элемент, имя которого совпадает с классом. Он используется для инициализации объектов этого типа класса. Добавьте указанный ниже конструктор в тип `BankAccount`. Добавьте следующий код непосредственно перед объявлением `MakeDeposit`:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Код, представленный выше, определяет свойства создаваемого объекта с использованием квалификатора `this`. Обычно этот квалификатор является необязательным и может быть опущен. Вы также можете написать:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
    Balance = initialBalance;
}
```

Квалификатор `this` требуется только в том случае, если локальная переменная или параметр имеет то же имя, что и поле или свойство. Квалификатор `this` опущен в оставшейся части этой статьи, если это не необходимо.

Конструкторы вызываются при создании объекта с помощью new. Замените строку `Console.WriteLine("Hello World!");` в файле *Program.cs* следующим кодом (замените `<name>` своим именем):

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with
{account.Balance} initial balance.");
```

Давайте запустите то, что вы создали до сих пор. Если вы работаете в Visual Studio, выберите **Запуск без отладки** в меню **Отладка**. Если вы используете командную строку, введите `dotnet run` каталог, в котором вы создали проект.

Вы заметили, что номер счета не указан? Нужно решить эту проблему. Номер счета следует назначить при создании объекта. Но создавать этот номер не входит в обязанности вызывающего. Код класса `BankAccount` должен иметь информацию о том, как присвоить номера новым счетам. Простой способ — начать с 10-цифрного числа. Увеличивайте его при создании каждого нового счета. Затем при создании объекта сохраните номер текущего счета.

Добавьте объявление члена в класс `BankAccount`. Поместите следующую строку кода после открывающей скобки `{` в начале класса `BankAccount`:

```
C#

private static int s_accountNumberSeed = 1234567890;
```

`accountNumberSeed` является элементом данных. Он имеет свойство `private`, то есть к нему может получить доступ только код внутри класса `BankAccount`. Этот метод разделяет публичные обязанности (например, наличие номера счета) и частную реализацию (как создаются номера счетов). Это также `static` означает, что все `BankAccount` объекты совместно используют один экземпляр этой переменной. Значение нестатической переменной является уникальным для каждого экземпляра объекта `BankAccount`. Поле `accountNumberSeed` является полем `private static` `s_` и таким образом имеет префикс в соответствии с соглашениями об именовании C#. Поле, обозначающее `s` `static` и `_`, обозначающее `private`. Чтобы инициализировать каждый номер учетной записи, добавьте в конструктор следующие две строки. Они должны располагаться за строкой с текстом `this.Balance = initialBalance`.

```
C#

Number = s_accountNumberSeed.ToString();
s_accountNumberSeed++;
```

Введите `dotnet run`, чтобы просмотреть результаты.

Создание депозитов и списаний

Для надлежащей работы ваш класс банковского счета должен принимать депозиты и списания. Чтобы реализовать депозиты и списания, создадим журнал для каждой транзакции на счете. Отслеживание каждой транзакции имеет несколько преимуществ по сравнению с простым обновлением баланса в каждой транзакции. Журнал можно использовать для аудита всех транзакций и управления ежедневным сальдо. Вычисление баланса из истории всех транзакций при необходимости гарантирует, что все ошибки в одной транзакции, исправленные, будут правильно отражены в балансе на следующем вычислении.

Начните с создания нового типа, представляющего транзакцию. Транзакция является типом `record`, который не имеет никаких обязанностей. Ему нужно несколько свойств. Создайте новый файл с именем *Transaction.cs*. Добавьте следующий код:

C#

```
namespace Classes;

public record Transaction(decimal Amount, DateTime Date, string Notes);
```

Теперь добавим List<T> объектов `Transaction` в класс `BankAccount`. Добавьте следующее объявление после конструктора в ваш файл *BankAccount.cs*:

C#

```
private List<Transaction> _allTransactions = new List<Transaction>();
```

Далее мы соответствующим образом вычислим `Balance`. Чтобы вычислить текущий баланс, нужно суммировать значения всех транзакций. На данный момент с помощью кода можно получить только начальный баланс аккаунта, поэтому необходимо обновить свойство `Balance`. В файле `public decimal Balance { get; }` замените строку следующим кодом:

C#

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in _allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

В этом примере показан важный аспект ***свойств***. Вы вычисляете сальдо, когда другой программист запрашивает значение. В результате вашего вычисления выводится список всех транзакций и сумма в виде текущего сальдо.

Теперь реализуйте методы `MakeDeposit` и `MakeWithdrawal`. Эти методы применяют последние два правила: начальный баланс должен быть положительным, и любой вывод не должен создавать отрицательный баланс.

Эти правила представляют концепцию исключений. Стандартный способ, указывающий, что метод не может успешно завершить свою работу, заключается в том, чтобы создать исключение. В типе исключения и связанном с ним сообщении описывается ошибка.

Здесь метод `MakeDeposit` выбрасывает исключение, если сумма депозита не превышает 0. Метод `MakeWithdrawal` вызывает исключение, если сумма вывода не превышает 0, или если применение вывода приводит к отрицательному балансу. Добавьте следующий код после объявления списка `_allTransactions`:

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit
must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    _allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

`throw` создает исключение. Выполнение текущего блока завершается и управление передается в первый подходящий блок `catch` из стека вызовов. Вы добавите `catch` блок для тестирования этого кода немного позже.

Чтобы вместо непосредственного обновления сальдо добавлялась начальная транзакция, конструктор должен получить одно изменение. Так как вы уже написали метод `MakeDeposit`, вызовите его из конструктора. Готовый конструктор должен выглядеть так:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;
```

```
    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now — это свойство, которое возвращает текущие дату и время. Протестируйте этот код, добавив несколько депозитов и снятий в метод `Main`, следуя за кодом, который создает новый `BankAccount`.

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Затем проверьте, что вы обнаруживаете ошибочные состояния, пытаясь создать учетную запись с отрицательным балансом. Добавьте следующий код после только что добавленного блока кода:

C#

```
// Test that the initial balances must be positive.
BankAccount invalidAccount;
try
{
    invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
    return;
}
```

Вы используете оператор try-catch для пометки блока кода, который может вызвать исключения и для перехвата ожидаемых ошибок. Так же можно проверять код, который вызывает исключение при получении отрицательного баланса. Добавьте следующий код перед объявлением `invalidAccount` в `Main` методе:

C#

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
```

```
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Сохраните файл и введите `dotnet run` для проверки.

Задача — регистрация всех транзакций

В завершение вы создадите метод `GetAccountHistory`, который создает `string` для журнала транзакций. Добавьте этот метод в тип `BankAccount`:

C#

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in _allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

История использует класс StringBuilder для форматирования строки, в которой содержится по одной строке для каждой транзакции. Вы использовали форматирование строк в предыдущих руководствах. Новый символ — это `\t`. Он позволяет вставить вкладку для форматирования выходных данных.

Добавьте следующую строку, чтобы проверить его в файле *Program.cs*:

C#

```
Console.WriteLine(account.GetAccountHistory());
```

Запустите программу и проверьте результаты.

Следующие шаги

Если у вас возникли проблемы, изучите исходный код для этого руководства, размещенный в репозитории GitHub.

Вы можете продолжить, перейдя к учебнику по объектно-ориентированному программированию.

Дополнительные сведения об этих понятиях см. в следующих статьях:

- Инструкции выбора
- Инструкции итерации

программирование Object-Oriented (C#)

10.10.2025

C# — это объектно-ориентированный язык программирования. Ниже приведены четыре основных принципа объектно-ориентированного программирования:

- *Абстракция* Моделирование соответствующих атрибутов и взаимодействий сущностей в качестве классов для определения абстрактного представления системы.
- *Инкапсуляция* Скрытие внутреннего состояния и функциональных возможностей объекта и разрешение доступа только через общедоступный набор функций.
- *Наследство* Возможность создавать новые абстракции на основе существующих абстракций.
- *Полиморфизм* Возможность реализовать унаследованные свойства или методы различными способами в нескольких абстракциях.

В предыдущем руководстве введение в классы вы познакомились как с *абстракцией*, так и с *инкапсуляцией*. Класс `BankAccount` предоставил абстракцию для концепции банковского счета. Ее реализацию можно изменить, не затрагивая какой-либо код, который использовал `BankAccount` класс. `BankAccount` `Transaction` Оба класса обеспечивают инкапсуляцию компонентов, необходимых для описания этих понятий в коде.

В этом руководстве описано, как расширить это приложение, чтобы использовать *наследование* и *полиморфизм* для добавления новых функций. Вы также добавляете функции в `BankAccount` класс, используя методы *абстракции* и *инкапсуляции* , которые вы узнали в предыдущем руководстве.

Создание различных типов учетных записей

После создания этой программы вы получите запросы на добавление в него функций. Это хорошо работает в ситуации, когда существует только один тип банковского счета. Со временем потребности меняются, и запрашиваются связанные типы учетных записей.

- Процентный счет, который начисляет проценты в конце каждого месяца.
- Линия кредита, которая может иметь отрицательный баланс, но когда баланс положительный, взимается процентная ставка каждый месяц.
- Это предоплаченная подарочная карта, начинающаяся с одного депозита и которую можно только погашать. Он может быть перезаполнен один раз в начале каждого месяца.

Все эти разные учетные записи похожи на `BankAccount` класс, определенный в предыдущем руководстве. Вы можете скопировать этот код, переименовать классы и внести изменения. Этот метод будет работать в краткосрочной перспективе, но это будет больше работы с течением времени. Все изменения будут скопированы во всех затронутых классах.

Вместо этого можно создать новые типы банковских счетов, наследующие методы и данные из класса, созданного `BankAccount` в предыдущем руководстве. Эти новые классы могут расширить `BankAccount` класс с определенным поведением, необходимым для каждого типа:

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Каждый из этих классов *наследует* общее поведение от общего *базового класса*, `BankAccount` класса. Написание реализаций для новых и различных функций в каждом *из производных классов*. Эти производные классы уже имеют все поведение, определенное в классе `BankAccount`.

Рекомендуется создать каждый класс в другом исходном файле. В Visual Studio щелкните проект правой кнопкой мыши и выберите класс, чтобы *добавить* новый класс в новый файл. В Visual Studio Code выберите "*Файл*" , а затем "*Создать*" , чтобы создать исходный файл. В любом инструменте назовите файл, соответствующий классу: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs* и *GiftCardAccount.cs*.

При создании классов, как показано в предыдущем примере, вы обнаружите, что ни один из производных классов не компилируется. Конструктор отвечает за инициализацию объекта. Конструктор производных классов должен инициализировать производный класс и указать инструкции по инициализации объекта базового класса, включенного в производный класс. Правильная инициализация обычно происходит без дополнительного кода. Класс `BankAccount` объявляет один открытый конструктор со следующей подписью:

```
C#
```

```
public BankAccount(string name, decimal initialBalance)
```

Компилятор не создает конструктор по умолчанию при определении конструктора самостоятельно. Это означает, что каждый производный класс должен явно вызывать этот конструктор. Вы объявляете конструктор, который может передавать аргументы конструктору базового класса. В следующем коде показан конструктор для `InterestEarningAccount`:

```
C#
```

```
public InterestEarningAccount(string name, decimal initialBalance) : base(name,
initialBalance)
{
}
```

Параметры для этого нового конструктора соответствуют типу параметров и именам конструктора базового класса. Синтаксис `: base()` используется для указания вызова конструктора базового класса. Некоторые классы определяют несколько конструкторов, и этот синтаксис позволяет выбрать вызывающий конструктор базового класса. После обновления конструкторов можно разработать код для каждого производных классов. Требования к новым классам можно указать следующим образом:

- Счёт с начислением процентов
  - Получает кредит в размере 2% от баланса на конец месяца.
- Линия кредита:
  - Может иметь отрицательный баланс, но величина его абсолютного значения не должна превышать кредитный предел.
  - Каждый месяц взимается процентная плата, если остаток на конец месяца не равен 0.
  - Взимается плата за каждое снятие, которое выходит за кредитный лимит.
- Учетная запись подарочной карты:
  - Может быть заполнено указанной суммой один раз в месяц, в последний день месяца.

Вы можете увидеть, что все три этих типа учетных записей имеют действие, которое выполняется в конце каждого месяца. Однако каждый тип учетной записи выполняет разные задачи. Для реализации этого кода используется *полиморфизм* . Создайте один `virtual` метод в `BankAccount` классе:

```
C#
```

```
public virtual void PerformMonthEndTransactions() { }
```

В приведенном выше коде показано, как использовать `virtual` ключевое слово для объявления метода в базовом классе, для который производный класс может предоставить другую реализацию. `virtual` Метод — это метод, в котором любой производный класс может выбрать повторную активацию. Когда производный класс определяет новую реализацию, он вызывает *переопределение реализации базового класса*. Производные классы используют ключевое `override` слово для определения новой реализации. Ключевое `virtual` слово указывает, что производные классы могут переопределить поведение. Также можно объявлять методы `abstract`, в которых производные классы должны переопределить поведение. Базовый класс не предоставляет реализацию `abstract` для метода. Затем необходимо определить реализацию для двух созданных новых классов. Начните с `InterestEarningAccount`:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.02m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Добавьте следующий код в `LineOfCreditAccount` файл . Код делает баланс отрицательным, чтобы вычислить положительный процент, который списывается со счета.

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

Классу `GiftCardAccount` требуется два изменения для реализации функциональных возможностей по окончании месяца. Сначала измените конструктор, чтобы включить необязательный объем для добавления каждого месяца:

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal monthlyDeposit
= 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

Конструктор предоставляет значение по умолчанию для `monthlyDeposit`, поэтому вызывающие могут пропускать внесение ежемесячного `0`. Затем переопределите `PerformMonthEndTransactions` метод для добавления ежемесячного депозита, если он был задан как ненулевое значение в конструкторе:

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

Переопределение применяет ежемесячный набор депозитов в конструкторе. Добавьте следующий код в метод `Main`. Этот код проверяет эти изменения для `GiftCardAccount` и :`InterestEarningAccount`

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Проверьте результаты. Теперь добавьте аналогичный набор тестового кода для `LineOfCreditAccount`:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Добавьте предыдущий код и запустите программу. Выходные данные должны быть примерно такими, как следующая ошибка:

Консоль

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be
positive (Parameter 'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String
note) in BankAccount.cs:line 42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance) in
BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance)
in LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

> ⓘ **Примечание**
>
> Фактические выходные данные включают полный путь к папке с проектом. Имена папок были пропущены для краткости. Кроме того, в зависимости от формата кода номера строк могут немного отличаться.

Этот код завершается ошибкой, так как `BankAccount` предполагает, что начальный баланс должен быть больше 0. Другое предположение, встроенное в `BankAccount` класс, заключается в том, что баланс не может быть отрицательным. Вместо этого любой вывод средств, который превышает баланс счета, отклоняется. Оба этих предположения должны измениться. Счет кредитной линии начинается с 0 и обычно имеет отрицательный баланс. Кроме того, если клиент занимает слишком много денег, он облагается сбором. Транзакция принимается, она просто стоит больше. Первое правило можно реализовать, добавив необязательный аргумент в `BankAccount` конструктор, указывающий минимальный баланс. Значение по умолчанию — `0`. Во втором правиле требуется механизм, позволяющий производным классам изменять алгоритм по умолчанию. В некотором смысле базовый класс "спрашивает" производный тип о том, что должно

произойти в случае перерасхода средств. Поведение по умолчанию заключается в отклонении транзакции путем создания исключения.

Начнем с добавления второго конструктора, который включает необязательный `minimumBalance` параметр. Этот новый конструктор выполняет все действия, выполненные существующим конструктором. Кроме того, он задает свойство минимального баланса. Можно скопировать тело существующего конструктора, но это означает, что в будущем вам придется вносить изменения в двух местах. Вместо этого можно использовать *цепочку конструкторов* для вызова одного конструктора другого. В следующем коде показаны два конструктора и новое поле:

```C#
private readonly decimal _minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

В приведенном выше коде показаны два новых метода. Во-первых, `minimumBalance` поле помечается как `readonly`. Это означает, что значение невозможно изменить после создания объекта. После создания `BankAccount`, `minimumBalance` нельзя изменить. Во-вторых, конструктор, принимающий два параметра, использует `: this(name, initialBalance, 0) { }` в качестве своей реализации. Выражение `: this()` вызывает другой конструктор, тот, который имеет три параметра. Этот метод позволяет использовать одну реализацию для инициализации объекта, даже если клиентский код может выбрать один из многих конструкторов.

Эта реализация вызывает `MakeDeposit` только в том случае, если начальный баланс больше `0`. Это сохраняет правило, что депозиты должны быть положительными, но позволяет открыть кредитный счет с балансом `0`.

Теперь, когда `BankAccount` класс имеет поле с доступом только для чтения для минимального баланса, окончательное изменение заключается в замене жестко заданного кода `0` на `minimumBalance` в методе `MakeWithdrawal`.

C#

```
if (Balance - amount < _minimumBalance)
```

Можно изменить конструктор `BankAccount`, чтобы вызвать новый базовый конструктор, после расширения класса `LineOfCreditAccount`, как показано в следующем коде.

C#

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal
creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

Обратите внимание, что `LineOfCreditAccount` конструктор изменяет знак `creditLimit` параметра, чтобы он соответствовал значению `minimumBalance` параметра.

Разные правила овердрафта

Последняя функция, добавляемая, позволяет `LineOfCreditAccount` взимать плату за переход на кредитный лимит вместо отказа транзакции.

Одним из способов является определение виртуальной функции, в которой реализуется требуемое поведение. Класс `BankAccount` рефакторингирует `MakeWithdrawal` метод двумя методами. Новый метод выполняет указанное действие, когда снятие приводит баланс ниже минимального уровня. Существующий `MakeWithdrawal` метод имеет следующий код:

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

Замените его следующим кодом:

```C#
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    ArgumentOutOfRangeException.ThrowIfNegativeOrZero(amount);
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance - amount <
_minimumBalance);
    Transaction? withdrawal = new(-amount, date, note);
    _allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        _allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    else
    {
        return default;
    }
}
```

Добавленный метод — это `protected`, что означает, что его можно вызывать только из производных классов. Это декларация запрещает другим клиентам вызывать метод. Кроме того, это сделано для того, чтобы производные классы могли изменить поведение. Тип возвращаемого `Transaction?` значения — . Заметка `?` указывает, что метод может возвращать `null`. Добавьте следующую реализацию в `LineOfCreditAccount`. Этот код взимает плату за превышение лимита вывода:

```C#
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

Переопределение возвращает транзакцию с комиссией при превышении лимита учетной записи. Если вывод не превышает предел, метод возвращает транзакцию `null` . Это означает, что плата не взимается. Проверьте эти изменения, добавив следующий код в `Main` метод в `Program` классе:

```C#
```

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Запустите программу и проверьте результаты.

Сводка

Если у вас возникли проблемы, изучите исходный код для этого руководства, размещенный в репозитории GitHub.

В этом руководстве показано множество методов, используемых в программировании Object-Oriented:

- Вы использовали *абстракцию при определении* классов для каждого из различных типов учетных записей. Эти классы описали поведение для этого типа учетной записи.
- Вы использовали *инкапсуляцию* , когда вы сохранили много сведений `private` в каждом классе.
- Вы использовали *наследование* для использования уже созданной в `BankAccount` классе реализации, чтобы сэкономить код.
- Вы использовали *Полиморфизм* при создании `virtual` методов, которые производные классы могли переопределить для создания определенного поведения для этого типа учетной записи.

Наследование в C# и .NET

Статья • 03.04.2025

В этом руководстве вы познакомитесь с наследованием в C#. Наследование — это функция объектно-ориентированных языков программирования, которая позволяет определить базовый класс, предоставляющий определенные функциональные возможности (данные и поведение), а также определять производные классы, наследующие или переопределяют эту функциональность.

Предпосылки

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Инструкции по установке

В Windows этот файл конфигурации WinGet используется для установки всех необходимых компонентов. Если у вас уже установлено что-то, WinGet пропустит этот шаг.

1. Скачайте файл и дважды щелкните его, чтобы запустить его.
2. Прочитайте лицензионное соглашение, введите `и`, и выберите `ввод` при появлении запроса на принятие.
3. Если на панели задач появится мигающий запрос контроля учетных записей пользователей (UAC), разрешите установку продолжить.

На других платформах необходимо установить каждый из этих компонентов отдельно.

1. Скачайте рекомендуемый установщик на странице загрузки пакета SDK для .NET и дважды щелкните его, чтобы запустить его. Страница загрузки обнаруживает платформу и рекомендует последний установщик для вашей платформы.
2. Скачайте последнюю версию установщика на домашней странице Visual Studio Code и дважды щелкните его, чтобы запустить его. Эта страница также обнаруживает платформу, а ссылка должна быть правильной для вашей системы.
3. Нажмите кнопку "Установить" на странице расширения C# DevKit. Откроется код Visual Studio и запрашивается, нужно ли установить или включить

расширение. Выберите "Установить".

Выполнение примеров

Чтобы создать и запустить примеры в этом руководстве, используйте утилиту dotnet из командной строки. Выполните следующие действия для каждого примера:

1. Создайте каталог для хранения примера.
2. Введите команду dotnet new console в командной строке, чтобы создать проект .NET Core.
3. Скопируйте и вставьте код из примера в редактор кода.
4. Введите команду dotnet restore из командной строки, чтобы загрузить или восстановить зависимости проекта.

   Вам не нужно выполнять команду dotnet restore, так как она выполняется неявно всеми командами, которые требуют восстановления, например `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` и `dotnet pack`. Чтобы отключить неявное восстановление, используйте параметр `--no-restore`.

   Команду `dotnet restore` по-прежнему удобно использовать в некоторых сценариях, где необходимо явное восстановление, например в сборках с использованием непрерывной интеграции в Azure DevOps Services или системах сборки, где требуется явно контролировать время восстановления.

   В документации по dotnet restore приведены сведения об управлении каналами NuGet.
5. Введите команду dotnet run для компиляции и выполнения примера.

Контекст: Что такое наследование?

наследование является одной из фундаментальных черт объектно-ориентированного программирования. Он позволяет определить дочерний класс, который повторно использует (наследует), расширяет или изменяет поведение родительского класса. Класс, члены которого наследуются, называется *базовым классом*. Класс, наследующий члены базового класса, называется *производным классом*.

C# и .NET поддерживают только *одно наследование*. То есть класс может наследоваться только от одного класса. Однако наследование является транзитивным, что позволяет определить иерархию наследования для набора типов. Другими словами, тип `D` может наследовать от типа `C`, который наследует от типа `B`, который наследует от типа базового класса `A`. Так как наследование является транзитивным, члены типа `A` доступны для типа `D`.

Не все члены базового класса наследуются производными классами. Следующие члены не наследуются:

- Статические конструкторы, которые инициализируют статические данные класса.
- конструкторы экземпляров, которые вызываются для создания нового экземпляра класса. Каждый класс должен определять собственные конструкторы.
- Финализаторы, которые вызываются сборщиком мусора рантайма для уничтожения экземпляров класса.

Хотя все остальные члены базового класса наследуются производными классами, видимость их зависит от уровня доступа. Доступность члена влияет на его видимость для производных классов следующим образом:

- Закрытые члены доступны только в производных классах, которые вложены в свой базовый класс. В противном случае они не отображаются в производных классах. В следующем примере `A.B` является вложенным классом, производным от `A`, а `C` производным от `A`. В A.B отображается частное поле `A._value`. Однако если удалить комментарии из метода `C.GetValue` и попытаться скомпилировать пример, он создает ошибку компилятора CS0122: "A._value" недоступна из-за его уровня защиты".

```C#
public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}
```

```
public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- защищенные члены видны только в производных классах.
- Внутренние элементы видны только в производных классах, находящихся в той же сборке, что и базовый класс. Они не видны в производных классах, расположенных в другой сборке базового класса.
- открытые члены видны в производных классах и являются частью публичного интерфейса производного класса. Общедоступные унаследованные члены можно вызывать так же, как будто они определены в производном классе. В следующем примере класс `A` определяет метод с именем `Method1`, а класс `B` наследует от класса `A`. В этом примере `Method1` вызывается как метод экземпляра `B`.

C#

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
```

```
        B b = new ();
        b.Method1();
    }
}
```

Производные классы также могут *переопределять* унаследованные члены, обеспечивая альтернативную реализацию. Чтобы иметь возможность переопределить элемент, элемент в базовом классе должен быть помечен ключевым словом виртуальным. По умолчанию члены базового класса не помечены как `virtual` и не могут быть переопределены. Попытка переопределить невиртуационный член, как и в следующем примере, создает ошибку компилятора CS0506: "<член> не может переопределить унаследованный член <член>, так как он не помечен как виртуальный, абстрактный или переопределенный".

C#

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

В некоторых случаях производный класс *должен* переопределить реализацию базового класса. Члены базового класса, помеченные ключевым словом абстрактный, требуют, чтобы производные классы их переопределяли. При попытке скомпилировать следующий пример возникает ошибка компилятора CS0534, "<класс> не реализует унаследованный абстрактный элемент <член>", так как класс `B` не обеспечивает реализацию для `A.Method1`.

C#

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
```

```
{
    public void Method3()
    {
        // Do something.
    }
}
```

Наследование применяется только к классам и интерфейсам. Другие категории типов (структуры, делегаты и перечисления) не поддерживают наследование. Из-за этих правил попытка компиляции кода, как в следующем примере, приводит к ошибке компилятора CS0527: "Тип "ValueType" в списке интерфейсов не является интерфейсом". Сообщение об ошибке указывает, что, хотя можно определить интерфейсы, реализующие структуру, наследование не поддерживается.

C#

```
public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Неявное наследование

Помимо любых типов, которые они могут наследовать через одно наследование, все типы в системе типов .NET неявно наследуются от Object или типа, производного от него. Общие функциональные возможности Object доступны для любого типа.

Чтобы узнать, что означает неявное наследование, давайте определим новый класс, `SimpleClass`, то есть просто пустое определение класса:

C#

```
public class SimpleClass
{ }
```

Затем можно использовать отражение (что позволяет проверять метаданные типа для получения сведений об этом типе) для получения списка элементов, принадлежащих типу `SimpleClass`. Хотя в классе `SimpleClass` не определены элементы, выходные данные из примера указывают на то, что на самом деле у него девять элементов. Один из этих элементов — это конструктор без параметров (или по умолчанию), который автоматически предоставляется для типа `SimpleClass` компилятором C#. Остальные восемь являются членами Object, типа, от которого

все классы и интерфейсы в системе типов .NET в конечном итоге наследуются неявно.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.Public |
                             BindingFlags.NonPublic |
BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}
{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
//  Type SimpleClass has 9 members:
//  ToString (Method):  Public, Declared by System.Object
//  Equals (Method):  Public, Declared by System.Object
//  Equals (Method):  Public Static, Declared by System.Object
//  ReferenceEquals (Method):  Public Static, Declared by System.Object
//  GetHashCode (Method):  Public, Declared by System.Object
//  GetType (Method):  Public, Declared by System.Object
//  Finalize (Method):  Internal, Declared by System.Object
```

```
//  MemberwiseClone (Method):  Internal, Declared by System.Object
//  .ctor (Constructor):  Public, Declared by SimpleClass
```

Неявное наследование из класса Object делает эти методы доступными для класса `SimpleClass`:

- Открытый метод `ToString`, который преобразует объект `SimpleClass` в строковое представление, возвращает полное имя типа. В этом случае метод `ToString` возвращает строку SimpleClass.
- Три метода, которые проверяют равенство двух объектов: общедоступный метод экземпляра `Equals(Object)`, статический общедоступный метод `Equals(Object, Object)` и статический общедоступный метод `ReferenceEquals(Object, Object)`. По умолчанию эти методы проверяются на равенство ссылок; То есть, чтобы быть равным, две переменные объекта должны ссылаться на один и тот же объект.
- Общедоступный метод `GetHashCode`, который вычисляет значение, позволяющее экземпляру типа использоваться в хэшированных коллекциях.
- Открытый метод `GetType`, который возвращает объект Type, представляющий тип `SimpleClass`.
- Защищенный метод Finalize, предназначенный для освобождения неуправляемых ресурсов перед восстановлением памяти объекта сборщиком мусора.
- Защищенный метод MemberwiseClone, который создает неглубокий клон текущего объекта.

Из-за неявного наследования можно вызывать любой унаследованный элемент из объекта `SimpleClass` так же, как если бы он был фактически членом, определенным в классе `SimpleClass`. Например, в следующем примере вызывается метод `SimpleClass.ToString`, который `SimpleClass` наследует от Object.

C#

```
public class EmptyClass
{ }

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new();
```

```
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        EmptyClass
```

В следующей таблице перечислены категории типов, которые можно создать в C# и типы, от которых они неявно наследуются. Каждый базовый тип делает другой набор элементов доступным через наследование для неявно производных типов.

Развернуть таблицу

Тип категории	Неявно наследуется от
класс	Object
Структура	ValueType, Object
перечисление	Enum, ValueType, Object
делегат	MulticastDelegate, Delegate, Object

Наследование и отношение "это"

Обычно наследование используется для выражения "является" связью между базовым классом и одним или несколькими производными классами, где производные классы являются специализированными версиями базового класса; Производный класс — это тип базового класса. Например, класс `Publication` представляет публикацию любого вида, а классы `Book` и `Magazine` представляют определенные типы публикаций.

> **Примечание**
>
> Класс или структура могут реализовать один или несколько интерфейсов. Хотя реализация интерфейса часто представляется как обходное решение для одиночного наследования или как способ использования наследования со структурами, она предназначена для выражения другой связи ("может сделать") между интерфейсом и типом, который его реализует, в отличие от наследования. Интерфейс определяет подмножество функциональных возможностей (например, возможность проверки равенства, сравнения или сортировки объектов или поддержки анализа и форматирования с учетом языка и региона), которые интерфейс предоставляет реализующим его типам.

Обратите внимание, что "является" также выражает связь между типом и конкретным экземпляром этого типа. В следующем примере `Automobile` — это класс, имеющий три уникальных свойства только для чтения: `Make`, производитель автомобиля; `Model`, тип автомобиля; и `Year`, его год производства. Класс `Automobile` также имеет конструктор, аргументы которого назначаются значениям свойств, и он переопределяет метод Object.ToString для создания строки, которая однозначно определяет экземпляр `Automobile`, а не класс `Automobile`.

C#

```
public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot
be null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or
have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot
be null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or
have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

В этом случае вы не должны полагаться на наследование для представления конкретных марок и моделей автомобилей. Например, вам не нужно определять тип `Packard` для представления автомобилей, производимых компанией Packard Motor Car. Вместо этого можно представить их, создав объект `Automobile` с

соответствующими значениями, переданными конструктору класса, как показано в следующем примере.

```C#
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//        1948 Packard Custom Eight
```

Связь "является" на основе наследования наиболее подходит для базового класса и производных классов, которые добавляют дополнительные члены или требуют дополнительных функциональностей, отсутствующих в базовом классе.

Проектирование базового класса и производных классов

Рассмотрим процесс проектирования базового класса и его производных классов. В этом разделе вы определите базовый класс, `Publication`, который представляет публикацию любого вида, например книгу, журнал, газету, научный журнал, статью и т. д. Вы также определите класс `Book`, являющийся производным от `Publication`. Вы можете легко расширить пример для определения других производных классов, таких как `Magazine`, `Journal`, `Newspaper` и `Article`.

Базовый класс Publication

При проектировании класса `Publication` необходимо принять несколько решений по проектированию:

- Какие члены включаются в базовый класс `Publication`, а также предоставляют ли члены `Publication` реализации методов или является ли `Publication` абстрактным базовым классом, который служит шаблоном для производных классов.

В этом случае класс `Publication` будет предоставлять реализации методов. В разделе Проектирование абстрактных базовых классов и производных классов содержится пример, использующий абстрактный базовый класс для определения методов, которые производные классы должны переопределять. Производные классы свободны для предоставления любой реализации, подходящей для производного типа.

Возможность повторного использования кода (то есть несколько производных классов совместно используют объявление и реализацию методов базового класса и не нужно переопределять их) — это преимущество не абстрактных базовых классов. Поэтому необходимо добавить элементы в `Publication`, если их код, скорее всего, будет совместно использоваться некоторыми или большинством специализированных типов `Publication`. Если вы не сможете эффективно предоставлять реализации базового класса, в конечном итоге придется предоставлять в основном идентичные реализации членов в производных классах, а не одну реализацию в базовом классе. Необходимость поддерживать повторяющийся код в нескольких расположениях является потенциальным источником ошибок.

Чтобы развернуть повторное использование кода и создать логическую и интуитивно понятную иерархию наследования, необходимо убедиться, что вы включаете в класс `Publication` только данные и функциональные возможности, которые являются общими для всех или большинства публикаций. Затем производные классы реализуют элементы, уникальные для конкретных типов публикаций, которые они представляют.

- Как далеко расширить иерархию классов. Вы хотите разработать иерархию из трех или нескольких классов, а не просто базовый класс и один или несколько производных классов? Например, `Publication` может быть базовым классом `Periodical`, который, в свою очередь, является базовым классом `Magazine`, `Journal` и `Newspaper`.

  Например, вы будете использовать небольшую иерархию класса `Publication` и одного производного класса, `Book`. Вы можете легко расширить пример, чтобы создать ряд дополнительных классов, производных от `Publication`, таких как `Magazine` и `Article`.

- Имеет ли смысл создать экземпляр базового класса. Если это не так, следует применить ключевое слово абстрактное к классу. В противном случае объект класса `Publication` создается через вызов конструктора. Если предпринята попытка создать экземпляр класса, помеченного ключевым словом `abstract`

прямым вызовом конструктора класса, компилятор C# создает ошибку CS0144, "Не удается создать экземпляр абстрактного класса или интерфейса". Если выполняется попытка создать экземпляр класса с помощью отражения, метод отражения создает MemberAccessException.

По умолчанию базовый класс можно создать, вызвав конструктор класса. Не нужно явно определять конструктор класса. Если он отсутствует в исходном коде базового класса, компилятор C# автоматически предоставляет конструктор по умолчанию (без параметров).

Например, вы пометите класс `Publication` как абстрактный, чтобы он не мог быть создан. Класс `abstract` без методов `abstract` указывает, что этот класс представляет абстрактную концепцию, общую между несколькими конкретными классами (например, `Book`, `Journal`).

- Обязаны ли производные классы наследовать реализацию базового класса конкретных членов, имеют ли они возможность её переопределить, или они должны предоставить свою собственную реализацию. Вы используете ключевое слово abstract, чтобы принудить производные классы предоставить реализацию. Вы используете виртуальное ключевое слово для того чтобы разрешить производным классам переопределить метод базового класса. По умолчанию методы, определенные в базовом классе, *не* переопределяются.

  Класс `Publication` не имеет методов `abstract`, но сам класс `abstract`.

- Является ли производный класс конечным классом в иерархии наследования и сам по себе не может использоваться в качестве базового класса для дополнительных производных классов. По умолчанию любой класс может служить базовым классом. Можно применить ключевое слово , чтобы указать, что класс не может служить базовым классом для других классов. Попытка наследовать от запечатанного класса вызвала ошибку компилятора CS0509, "нельзя наследовать от запечатанного типа <typeName>".

  В вашем примере вы пометите ваш производный класс как `sealed`.

В следующем примере показан исходный код для класса `Publication`, а также перечисление `PublicationType`, возвращаемое свойством `Publication.PublicationType`. Помимо элементов, наследующих от Object, класс `Publication` определяет следующие уникальные элементы и переопределения элементов:

```
C#
```

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The
number of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
            return _datePublished.ToString("d");
    }

    public void Publish(DateTime datePublished)
    {
```

```
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder is
required.");
        CopyrightName = copyrightName;

        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear
+ 2)
            throw new ArgumentOutOfRangeException($"The copyright year must
be between {currentYear - 10} and {currentYear + 1}");
        CopyrightDate = copyrightDate;
    }

    public override string ToString() => Title;
}
```

- Конструктор

  Так как класс `Publication` `abstract`, он не может быть создан непосредственно из кода, как показано в следующем примере:

  C#

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and
  Games",
                                    PublicationType.Book);
  ```

  Однако его конструктор экземпляра можно вызывать непосредственно из производных конструкторов классов, как показано в исходном коде для класса `Book`.

- Два свойства, связанные с публикацией

  `Title` — это свойство только для чтения String, значение которого предоставляется путем вызова конструктора `Publication`.

  `Pages` — это свойство Int32 чтения и записи, которое указывает общее количество страниц в публикации. Значение хранится в частном поле с именем `totalPages`. Это должно быть положительным числом, иначе выбрасывается ArgumentOutOfRangeException.

- Члены, связанные с издателем

Два свойства только для чтения, `Publisher` и `Type`. Значения изначально предоставляются вызовом конструктора класса `Publication`.

- Члены, связанные с публикацией

  Два метода, `Publish` и `GetPublicationDate`, задают и возвращают дату публикации. Метод `Publish` при вызове устанавливает закрытый флаг `published` в значение `true` и назначает дату, переданную ему как аргумент, в частное поле `datePublished`. Метод `GetPublicationDate` возвращает строку "NYP", если флаг `published` `false`, а значение поля `datePublished`, если оно `true`.

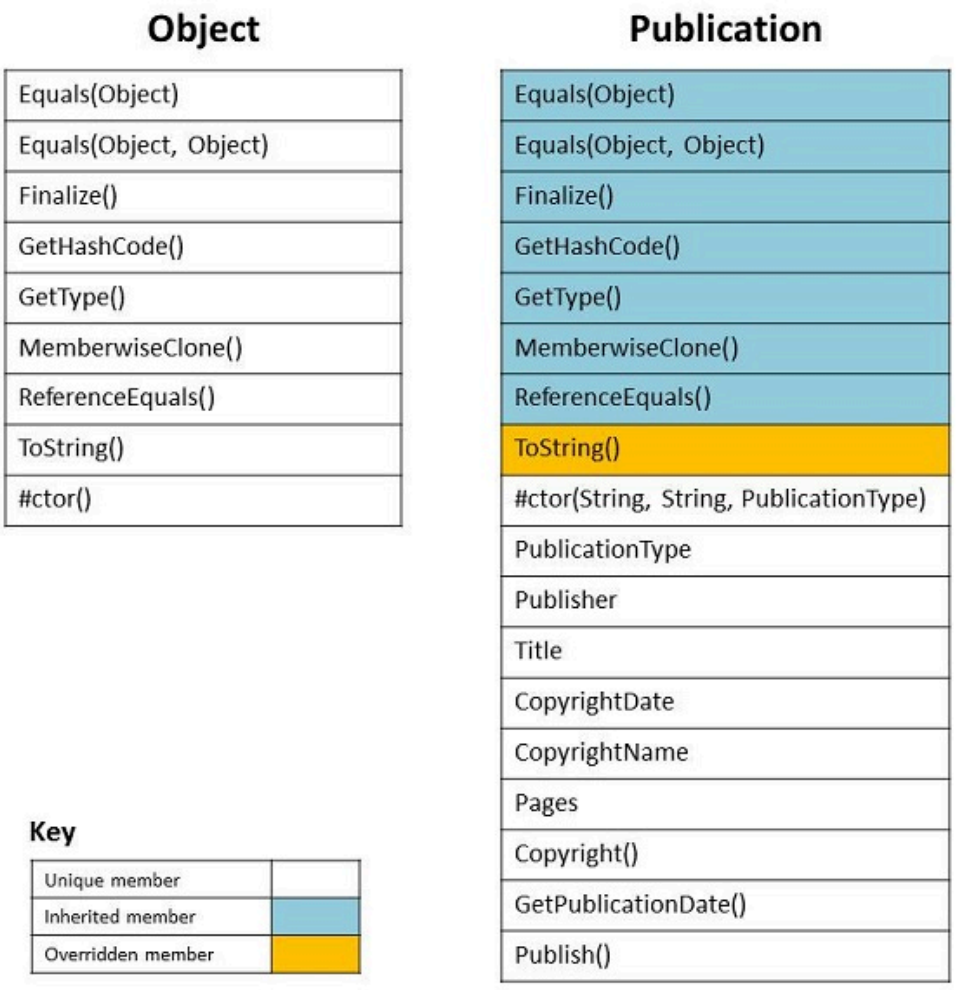
- Члены, связанные с авторским правом

  Метод `Copyright` принимает имя владельца авторских прав и год авторских прав в качестве аргументов и назначает их `CopyrightName` и `CopyrightDate` свойствам.

- Переопределение метода `ToString`

  Если тип не переопределяет метод Object.ToString, то возвращается полное имя типа, которое мало полезно для различения одного экземпляра от другого. Класс `Publication` переопределяет Object.ToString, чтобы вернуть значение свойства `Title`.

На следующем рисунке показана связь между базовым `Publication` классом и его неявно унаследованным Object классом.



Класс `Book`

Класс `Book` представляет книгу как специализированный тип публикации. В следующем примере показан исходный код для класса `Book`.

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
           this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher)
: base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string
without "-" characters.
        // We could also determine whether the ISBN is valid by comparing
its checksum digit
        // with a computed checksum.
```

```
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-
character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException("The ISBN can consist of numeric
characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price), "The price
cannot be negative.");
        decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-
character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Book book)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{(string.IsNullOrEmpty(Author) ?
"" : Author + ", ")}{Title}";
}
```

Помимо элементов, наследующих от `Publication`, класс `Book` определяет следующие уникальные элементы и переопределения элементов:

- Два конструктора

  Два конструктора `Book` используют три общих параметра. Два параметра: *заголовок* и *издатель*соответствуют параметрам конструктора `Publication`. Третий — это *автор*, который хранится в общедоступном неизменяемом `Author` свойстве. Конструктор включает параметр *isbn*, который сохраняется в автоматическом свойстве `ISBN`.

  Первый конструктор использует ключевое слово this для вызова другого конструктора. Цепочка конструкторов — это распространенный шаблон в определении конструкторов. Конструкторы с меньшим количеством параметров предоставляют значения по умолчанию при вызове конструктора с наибольшим количеством параметров.

  Второй конструктор использует ключевое слово base для передачи названия и имени издателя конструктору базового класса. Если в исходном коде не выполняется явный вызов конструктора базового класса, компилятор C# автоматически предоставляет вызов конструктору по умолчанию или без параметров базового класса.

- Свойство только для чтения `ISBN`, которое возвращает Международный стандартный номер книги объекта `Book`, уникальное 10- или 13-значное число. ISBN предоставляется в качестве аргумента одного из конструкторов `Book`. ISBN хранится в закрытом поле резервной копии, которое автоматически создается компилятором.

- Свойство `Author` только для чтения. Имя автора передается в качестве аргумента обоим конструкторам `Book` и сохраняется в свойстве.

- Два свойства, относящиеся к цене, только для чтения: `Price` и `Currency`. Их значения предоставляются в качестве аргументов в вызове метода `SetPrice`. Свойство `Currency` является трехзначным символом валюты ISO (например, USD для доллара США). Символы валюты ISO можно получить из свойства ISOCurrencySymbol. Оба этих свойства являются внешними только для чтения, но оба могут быть заданы кодом в классе `Book`.

- Метод `SetPrice`, который задает значения свойств `Price` и `Currency`. Эти же свойства возвращают эти значения.

- Переопределяет метод `ToString` (унаследованный от `Publication`) и методы Object.Equals(Object) и GetHashCode (унаследованные от Object).

  Если он не переопределен, метод Object.Equals(Object) проверяет равенство ссылок. То есть два переменных объекта считаются равными, если они ссылаются на один и тот же объект. С другой стороны, в классе `Book` два объекта `Book` должны быть равными, если они имеют одинаковый ISBN.

  При переопределении метода Object.Equals(Object) необходимо также переопределить метод GetHashCode, который возвращает значение, которое среда выполнения использует для хранения элементов в хэшированных коллекциях для эффективного извлечения. Хэш-код должен возвращать значение, соответствующее тесту на равенство. Так как вы переопределили Object.Equals(Object) для возврата `true`, когда свойства ISBN двух объектов `Book` равны, вы возвращаете хэш-код, вычисленный с помощью вызова метода GetHashCode строки, возвращаемой свойством `ISBN`.

На следующем рисунке показана связь между классом `Book` и `Publication`его базовым классом.



Теперь можно создать экземпляр объекта `Book`, вызвать как его уникальные, так и унаследованные элементы, и передать его в качестве аргумента методу, который ожидает параметр типа `Publication` или типа `Book`, как показано в следующем примере.

C#

```
public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare,
```

```
William",
                              "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press",
"Shakespeare, William");
        Console.Write($"{book.Title} and {book2.Title} are the same
publication: " +
                $"{((Publication)book).Equals(book2)}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        Console.WriteLine($"{pub.Title}, " +
                  $"{(pubDate == "NYP" ? "Not Yet Published" : "published on
" + pubDate):d} by {pub.Publisher}");
    }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

Проектирование абстрактных базовых классов и их производных классов

В предыдущем примере вы определили базовый класс, предоставляющий реализацию для ряда методов, позволяющих производным классам совместно использовать код. Однако во многих случаях не предполагается, что базовый класс будет предоставлять реализацию. Вместо этого базовый класс является *абстрактным классом*, который объявляет *абстрактные методы*; он служит шаблоном, определяющим элементы, которые должны реализовать каждый производный класс. Обычно в абстрактном базовом классе реализация каждого производного типа уникальна для этого типа. Вы пометили класс ключевым словом abstract, так как не имело смысла создавать экземпляр объекта `Publication`, хотя класс предоставил реализации функциональных возможностей, общих для публикаций.

Например, каждая закрытая двухмерная геометрическая фигура содержит два свойства: область, внутренняя степень фигуры; и периметр или расстояние вдоль краев фигуры. Однако способ вычисления этих свойств полностью зависит от конкретной фигуры. Формула вычисления периметра (или окружности) окружности, например, отличается от формулы квадрата. Класс `Shape` — это класс

`abstract` с методами `abstract`. Это означает, что производные классы имеют одинаковые функциональные возможности, но производные классы реализуют эту функциональность по-разному.

В следующем примере определяется абстрактный базовый класс с именем `Shape`, который определяет два свойства: `Area` и `Perimeter`. Помимо маркировки класса с помощью ключевого слова абстрактного, каждый член экземпляра также помечается с помощью ключевого слова абстрактного. В этом случае `Shape` также переопределяет метод Object.ToString, возвращая название типа, а не полное имя. И он определяет два статических члена, `GetArea` и `GetPerimeter`, которые позволяют вызывающим легко извлекать площадь и периметр экземпляра любого производного класса. При передаче экземпляра производного класса в любой из этих методов среда выполнения вызывает переопределение метода производного класса.

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Затем можно унаследовать некоторые классы от `Shape`, которые представляют конкретные фигуры. В следующем примере определяются три класса, `Square`, `Rectangle` и `Circle`. Каждая из них использует формулу, уникальную для конкретной фигуры для вычисления области и периметра. Некоторые производные классы также определяют свойства, такие как `Rectangle.Diagonal` и `Circle.Diameter`, которые являются уникальными для фигуры, которую они представляют.

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
```

```
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2),
2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}
```

В следующем примере используются объекты, производные от `Shape`. Он создает экземпляр массива объектов, производных от `Shape`, и вызывает статические методы класса `Shape`, который предоставляет доступ к значениям свойств `Shape`. Среда выполнения извлекает значения из переопределенных свойств производных типов. В этом примере каждый объект `Shape` в массиве преобразуется в его производный тип. Если преобразование проходит успешно, извлекаются свойства конкретного подкласса `Shape`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (Shape shape in shapes)
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"   Is Square: {rect.IsSquare()},
Diagonal: {rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                Console.WriteLine($"   Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

Руководство. Создание XML-документации

В этом руководстве вы берете существующий объектно-ориентированный пример (из предыдущего руководства) и улучшите его с помощью комментариев xml-документации. Примечания XML-документации предоставляют полезные подсказки IntelliSense и могут участвовать в автоматически созданных справочных документах API. Вы узнаете, какие элементы заслуживают комментариев, как использовать основные теги, такие как `<summary>`, `<param>`, `<returns>`, `<value>`, `<remarks>`, `<example>`, `<seealso>`, `<exception>` и `<inheritdoc>`, и как согласованные, целенаправленные комментарии повышают удобство обслуживания, индексацию и совместную работу без добавления шума. К концу вы аннотировали общедоступную поверхность примера, создали проект для выдачи XML-файла документации и видели, как эти комментарии будут передаваться непосредственно в интерфейс разработчика и средства документации ниже.

Изучив это руководство, вы:

- ✓ Включите выходные данные XML-документации в проекте C#.
- ✓ Добавьте и структурируйте комментарии на XML-документации для типов и членов.
- ✓ Создайте проект и проверьте созданный XML-файл документации.

Предпосылки

- Пакет SDK для .NET
- Visual Studio или Visual Studio Code с набором средств разработки на C#.

Включение XML-документации

Загрузите проект, созданный в предыдущем объектно-ориентированном руководстве. Если вы предпочитаете начать с чистого листа, скопируйте пример из `dotnet/docs` репозитория в папку snippets/object-oriented-programming.

Затем включите вывод документации в формате XML, чтобы компилятор создавал файл `.xml` вместе со сборкой. Измените файл проекта и добавьте (или подтвердите) следующее свойство внутри `<PropertyGroup>` элемента:

XML

```xml
<GenerateDocumentationFile>True</GenerateDocumentationFile>
```

Если вы используете Visual Studio, вы можете включить это с помощью страницы свойств build.

Создайте проект. Компилятор создает XML-файл, который собирает все комментарии `///` из типов и членов, видимых для общественности. Этот файл поставляет подсказки IntelliSense, статические средства анализа и дочерние системы создания документации.

Теперь создайте проект. Вы увидите предупреждения для всех открытых членов, если отсутствуют комментарии `<summary>`. Эти предупреждения рассматриваются как список задач, который помогает предоставить полную, преднамеренную документацию. Откройте созданный XML-файл (рядом с выходными данными сборки) и проверьте начальную структуру. Сначала раздел пуст, `<members>` так как вы еще не добавили примечания:

XML

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>oo-programming</name>
    </assembly>
    <members>
    </members>
</doc>
```

Когда файл размещён, начните добавлять целевые комментарии XML и сразу же проверяйте, как каждый из них отображается в созданном выходе.

Добавление комментариев к документации

Теперь вы перебираете предупреждения при сборке, чтобы добавить краткое и полезное описание к типу `BankAccount`. Каждое предупреждение указывает общедоступный элемент, который не имеет `<summary>` элемента (или другого обязательного). Относитесь к списку предупреждений как к контрольному списку. Избегайте добавления шума: сосредоточьтесь на описании намерений, инвариантов и значимых ограничений на использование, избегая явного повторения имен типов или типов параметров.

1. Создайте проект еще раз. В Visual Studio или Visual Studio Code откройте панель "Список ошибок/ проблемы" и отфильтруйте предупреждения документации (CS1591). В командной строке запустите сборку и просмотрите предупреждения, выводимые в консоль.
2. Перейдите к первому предупреждению (`BankAccount` класс). В строке над объявлением введите `///`. Редактор формирует `<summary>` элемент. Замените

заполнитель одним предложением, ориентированным на действия. Предложение описывает роль учетной записи в домене. Например, он отслеживает транзакции и обеспечивает минимальный баланс.

3. Добавьте `<remarks>` только в том случае, если необходимо объяснить поведение. Примеры включают, как осуществляется контроль минимального баланса или как создаются номера счетов. Держите примечания короткими.
4. Для каждого свойства (`Number`, `Owner`, `Balance`) введите `///` и напишите `<summary>`, которая указывает, что именно представляет значение, а не то, как его возвращает тривиальный метод получения. Если свойство вычисляет значение (например `Balance`), добавьте `<value>` элемент, который уточняет вычисление.
5. Для каждого конструктора добавьте элементы `<summary>` и `<param>`, описывающие значение каждого аргумента, а не просто повторяющие имя параметра. Если одна перегруженная функция делегирует другой, добавьте краткий элемент `<remarks>`.
6. Для методов, которые могут вызывать ошибки, добавьте `<exception>` теги для каждого типа исключения, возникающего целенаправленно. Описать условие, которое активирует его. Не документируйте исключения, вызванные вспомогательными средствами проверки аргументов, если они не являются частью общедоступного контракта.
7. Для методов, возвращающих значение, добавьте `<returns>` с кратким описанием того, что получают вызывающие. Избегайте повторения имени метода или управляемого типа.
8. Сначала обратитесь к базовому классу `BankAccount` .

По завершении откройте вновь созданный XML-файл и убедитесь, что каждый участник отображается с вашими новыми элементами. Обрезанный фрагмент может выглядеть следующим образом:

XML

```xml
<member name="T:OOProgramming.BankAccount">
  <summary>Represents a bank account that records transactions and enforces an
optional minimum balance.</summary>
  <remarks>Account numbers are generated sequentially when each instance is
constructed.</remarks>
</member>
<member name="P:OOProgramming.BankAccount.Balance">
  <summary>Gets the current balance based on all recorded transactions.</summary>
  <value>The net sum of deposits and withdrawals.</value>
</member>
```

Совет

Сохраняйте сводки в виде одного предложения. Если вам нужно более одного, переместите дополнительный контекст в `<remarks>`.

Использование `<inheritdoc/>` в производных классах

Если вы наследуете от `BankAccount` (например, `SavingsAccount`, который применяет процентную ставку), вы можете наследовать базовую документацию вместо ее копирования. Добавьте самозакрывающийся элемент `<inheritdoc/>` внутрь блока документации производного члена. После `<inheritdoc/>` вы можете добавить дополнительные элементы (например, дополнительные сведения `<remarks>`), чтобы документировать специализированное поведение.

C#

```
/// <inheritdoc/>
/// <remarks>Adds monthly interest during month-end processing.</remarks>
public class SavingsAccount : BankAccount { /* ... */ }
```

Примечание

`<inheritdoc/>` уменьшает дублирование и помогает обеспечить согласованность при обновлении документации базового типа позже.

После завершения документирования общедоступных интерфейсов выполните компиляцию еще раз, чтобы подтвердить отсутствие предупреждений CS1591. Теперь проект создает полезные данные IntelliSense и структурированный XML-файл, готовый для публикации рабочих процессов.

Создание выходных данных из комментариев

Вы можете узнать больше, попробовав любой из этих инструментов для создания выходных данных из XML-комментариев.

- DocFX: *DocFX* — это генератор документации по API для .NET, который в настоящее время поддерживает C#, Visual Basic и F#. Он также позволяет настроить созданную справочную документацию. DocFX создает статический HTML-сайт из исходного кода и файлов Markdown. Кроме того, DocFX обеспечивает гибкость в

настройке макета и стиля веб-сайта с помощью шаблонов. Вы также можете создавать пользовательские шаблоны.

- Sandcastle: *средства Sandcastle* создают файлы справки для библиотек управляемых классов, содержащих как концептуальные, так и справочные страницы API. Средства Sandcastle используют командную строку и не имеют графического интерфейса, функций управления проектами или автоматизированного процесса сборки. Построитель файлов справки Sandcastle предоставляет автономные графические интерфейсы и средства на основе командной строки для создания файла справки автоматически. Пакет интеграции Visual Studio также доступен для него, чтобы помочь проектам создавать и управлять ими полностью из Visual Studio.
- Doxygen: *Doxygen* создает веб-браузер документации (в HTML) или автономное справочное руководство (в LaTeX) из набора документированных исходных файлов. Кроме того, поддерживается создание выходных данных в RTF (MS Word), PostScript, гиперссылочных PDF, сжатого HTML, DocBook и Unix мануалов. Можно настроить Doxygen для извлечения структуры кода из незадокументированных исходных файлов.

Связанный контент

- Справочник по языку документов XML
- Рекомендуемые XML-теги
- Примеры XML-документации

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 17.10.2025

Как безопасно выполнить приведение типов, используя сопоставление шаблонов и операторы is и as

17.06.2025

Так как объекты являются полиморфными, переменная базового класса может содержать производный тип. Чтобы получить доступ к членам экземпляра производного типа, необходимо привести значение в производный тип. Однако гипс создает риск бросания InvalidCastException. C# предоставляет операторы шаблонного сопоставления, которые выполняют преобразование условно только в том случае, если оно завершится успешно. C# также предоставляет операторы is и as, которые позволяют проверить, является ли значение определенного типа.

В следующем примере показано, как использовать оператор сопоставления `is` шаблонов:

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);
// Output:
// I am an animal.
// SuperNova is not a Mammal

static void FeedMammals(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
    else
    {
        // variable 'm' is not in scope here, and can't be used.
        Console.WriteLine($"{a.GetType().Name} is not a Mammal");
    }
}

static void TestForMammals(object o)
```

```
{
    // You also can use the as operator and test for null
    // before referencing the variable.
    var m = o as Mammal;
    if (m != null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

В предыдущем примере показаны некоторые функции синтаксиса сопоставления шаблонов. Инструкция `if (a is Mammal m)` объединяет тест с назначением инициализации. Назначение происходит только при успешном выполнении теста. Переменная `m` находится только в области видимости внедренной `if` инструкции, где ей было присвоено значение. Вы не можете получить доступ `m` позже в том же методе. В предыдущем примере также показано, как использовать as оператор для преобразования объекта в указанный тип.

Вы также можете использовать тот же синтаксис для тестирования, если тип значения, допускающий значение NULL , имеет значение, как показано в следующем примере:

C#

```
int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
```

```
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);

static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
    {
        Console.WriteLine(j);
    }
    else if (val is null) // If val is a nullable type with no value, this
expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}
```

В предыдущем примере показаны другие функции сопоставления шаблонов для использования с преобразованиями. Вы можете протестировать переменную для шаблона NULL, проверив конкретно значение `null` . Когда значение переменной в среде выполнения равно `null`, инструкция `is`, проверяющая тип, всегда возвращает `false`.

Оператор сопоставления шаблонов не разрешает тип значения с возможным значением NULL, например `is` или `int?`, но можно тестировать любой другой тип значения. `is` Шаблоны из предыдущего примера не ограничиваются типами значений, допускающих значение NULL. Эти шаблоны также можно использовать для проверки того, имеет ли переменная ссылочного типа значение или она `null`.

В предыдущем примере также показано, как использовать шаблон типа в операторе `switch`, где переменная может быть одного из нескольких различных типов.

Если вы хотите проверить, является ли переменная заданным типом, но не назначьте ее новой переменной, можно использовать `is` `as` операторы для ссылочных типов и типов значений, допускающих значение NULL. В следующем коде показано, как использовать `is` и `as` инструкции, которые были частью языка C# перед введением сопоставления с шаблонами, чтобы проверить, является ли переменная заданного типа.

C#

```
// Use the is operator to verify the type.
// before performing a cast.
Giraffe g = new();
UseIsOperator(g);

// Use the as operator and test for null
// before referencing the variable.
UseAsOperator(g);

// Use pattern matching to test for null
// before referencing the variable
UsePatternMatchingIs(g);

// Use the as operator to test
// an incompatible type.
SuperNova sn = new();
UseAsOperator(sn);

// Use the as operator with a value type.
// Note the implicit conversion to int? in
// the method body.
int i = 5;
UseAsWithNullable(i);

double d = 9.78654;
UseAsWithNullable(d);

static void UseIsOperator(Animal a)
{
    if (a is Mammal)
    {
        Mammal m = (Mammal)a;
        m.Eat();
```

```
    }
}

static void UsePatternMatchingIs(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
}

static void UseAsOperator(object o)
{
    Mammal? m = o as Mammal;
    if (m is not null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

Как видно, сравнивая этот код с кодом сопоставления шаблонов, синтаксис сопоставления шаблонов обеспечивает более надежные функции, сочетая тест и назначение в одной инструкции. По возможности используйте синтаксис сопоставления шаблонов.

Руководство. Выбор между кортежами, записями, структурой и классами

> **Совет**
>
> Эта статья является частью раздела **"Основы** ", написанного для разработчиков, которые знают по крайней мере один язык программирования и учат C#. Если вы не знакомы с программированием, начните с **начала работы**. Сведения о краткой справочной таблице см. в разделе **"Выбор типа**".

Одним из первых решений по проектированию в любом приложении C# является выбор типа для создания. Должен ли элемент меню быть `class` или `record`? Должен ли быстрый расчет возвращать `tuple` тип или именованный тип? Каждый выбор определяет, как код обрабатывает равенство, мутируемость и полиморфизм. Неправильный выбор приводит к шаблонному коду, ошибкам или и тому и другому.

В этом руководстве вы создадите небольшую модель кафе, в которой используются пункты меню, заказы, чтения датчиков и политики скидок. Вы анализируете характеристики и определяете лучший тип C# для каждой концепции. По мере продвижения вы узнаете, как распознавать дизайнерские факторы, которые склоняют выбор в пользу одного типа против другого.

Изучив это руководство, вы:

- ✓ Определите, когда кортеж является подходящим выбором для возврата нескольких значений.
- ✓ Модель неизменяемых данных с классом записи и понимание равенства на основе значений.
- ✓ Представлять небольшие, копируемые данные с помощью структуры записи.
- ✓ Управляйте изменяемым состоянием и поведением с помощью класса.
- ✓ Расширение класса путем наследования для добавления или ужесточения правил.
- ✓ Определите общие возможности для несвязанных типов с помощью интерфейса.

Необходимые условия

- Установите SDK .NET.

Использование кортежа для временной группировки

Кафе нуждается в методе, который возвращает как общее количество заказов, так и доход за день. Вы можете определить класс или структуру для этого, но два значения из одного метода не всегда оправдывают новый тип.

C#

```
(int TotalOrders, decimal Revenue) GetDailySummary(int orders, decimal revenue)
    => (orders, revenue);

Console.WriteLine("=== Tuple: daily summary ===");
var summary = GetDailySummary(42, 1234.50m);
Console.WriteLine($"Orders: {summary.TotalOrders}, Revenue: {summary.Revenue:F2}");

var (orders, revenue) = summary;
Console.WriteLine($"Deconstructed: {orders} orders, {revenue:F2}");
```

`GetDailySummary` возвращает `(int TotalOrders, decimal Revenue)` **кортеж**. Вызывающий объект обращается к каждому элементу по имени или деконструирует их в локальные переменные. Не требуется определение класса или структуры.

Почему кортеж лучше всего работает в этом примере

Кортеж работает здесь, так как группирование является локальным: один метод создает его, и один вызывающий использует его. Именованные элементы делают намерение понятным без церемонии полного типа. Если вы обнаружите, что используете одну и ту же структуру кортежа в нескольких методах, это сигнал к преобразованию её в запись или класс. Далее в этом руководстве вы увидите эту эволюцию. Для получения подробной информации о синтаксисе и возможностях кортежа см. в разделе "Типы кортежей".

Использование записи для неизменяемых данных

Каждый кафе нуждается в меню. Элемент меню имеет имя, цену и питательную заметку. Эти значения не изменяются после перечисления элемента. Две системы, которые обе ссылаются на "Латте за $4,50", должны согласиться, что они говорят о том же, даже если они создают отдельные объекты.

Объявите позиционные записи:

C#

```
record class MenuItem(string Name, decimal Price, string NutritionalNote);
```

Компилятор создает конструктор, деконструктор, `Equals`, `GetHashCode`, и `ToString` из этой одной строки. Используйте запись по назначению:

C#

```
Console.WriteLine("\n=== Record class: MenuItem ===");
var latte = new MenuItem("Latte", 4.50m, "Contains dairy");
var latte2 = new MenuItem("Latte", 4.50m, "Contains dairy");
var seasonal = latte with { Name = "Pumpkin Spice Latte", Price = 5.25m };

Console.WriteLine(latte);
Console.WriteLine(seasonal);
Console.WriteLine($"Same reference (latte vs latte2): {ReferenceEquals(latte,
latte2)}");
Console.WriteLine($"Value equal (latte vs latte2):    {latte == latte2}");
Console.WriteLine($"Value equal (latte vs seasonal):  {latte == seasonal}");
```

Два `MenuItem` экземпляра с одинаковыми данными равны, даже если они являются отдельными объектами. Это поведение иллюстрирует равенство на основе значений. Выражение `with` создает сезонный вариант, не изменяя исходный.

Класс записи является подходящим, если идентичность определяется из данных, а не из ссылки на объект, и созданные экземпляры редко изменяются. Вы получаете читаемые `ToString()` выходные данные, структурное равенство и `with` поддержку из коробки. Более подробное пошаговое руководство см. в разделе "Записи " и учебнике по записям.

Использование структуры записи для небольших типов значений

В кофемашине есть встроенный термометр, который сообщает о показаниях температуры. Каждое измерение это небольшое значение — число и единица — и копируется в журналы, оповещения и панели мониторинга. Вы не хотите, чтобы изменение в одной копии распространилось на другие.

Объявите структуру записи:

C#

```
record struct Measurement(double Value, string Unit);
```

Используйте структуру записи:

C#

```
Console.WriteLine("\n=== Record struct: Measurement ===");
var temp = new Measurement(72.5, "°F");
var copy = temp;

copy = copy with { Value = 23.0, Unit = "°C" };

Console.WriteLine($"Original: {temp.Value}{temp.Unit}");
Console.WriteLine($"Copy (converted): {copy.Value}{copy.Unit}");
```

Назначение `temp` для `copy` создаёт независимое значение. Выражение `with` создает новое значение без касания исходного шаблона, аналогичного классу записей, но с поведением копирования по назначению вместо копирования по ссылке.

Структура записи подходит, если данные небольшие (несколько примитивных полей) и копирование дешевле, чем выделение кучи. Вы получаете равенство значений и поддержку `with` подобно классам записей, с истинной семантикой значений. Измерения, координаты и аналогичные упрощенные данные являются естественными кандидатами. Дополнительные сведения см. в разделе "Записи " и "Типы структур".

Используйте класс, если требуется изменяемое состояние и поведение

Когда клиент подходит к кассе, бариста принимает заказ и добавляет товары по одному. Общая сумма увеличивается, состояние изменяется с "Ожидание" на "Готово", и два заказа, размещенные одновременно (даже с идентичными элементами), по-прежнему являются отдельными заказами.

C#

```
class Order : IOrder
{
    public virtual string Status { get; set; } = "Pending";
    private readonly List<(string Name, decimal Price)> _items = [];

    public void AddItem(string name, decimal price) => _items.Add((name, price));

    public decimal Total => _items.Sum(i => i.Price);

    public override string ToString() =>
        $"Order [{Status}]: {string.Join(", ", _items.Select(i => i.Name))} - Total:
{Total:F2}";
}
```

C#

```
Console.WriteLine("\n=== Class: Order ===");
var order = new Order();
order.AddItem("Latte", 4.50m);
order.AddItem("Croissant", 3.25m);
order.Status = "Ready";

Console.WriteLine(order);
```

Класс `Order` отслеживает элементы, вычисляет текущий итог и предоставляет возможность задавать `Status`. **Класс** — это правильный инструмент здесь, так как объект несет изменяемое состояние, которое изменяется на протяжении его жизненного цикла, поведение (методы) играет ключевую роль в назначении типа, и идентичность имеет значение — два заказа с одинаковыми элементами все равно считаются разными. Дополнительные сведения см. в разделе "Классы", "Структуры" и "Записи".

Используйте наследование, когда нужно расширить класс.

Кафе начинает оказывать услуги кейтеринга. Заказ на кейтеринг по-прежнему остаётся заказом — он имеет позиции и общую сумму, но также учитывает количество гостей и требует утверждения менеджера до того, как кухня отмечает его как готовый. Вместо повторения логики `Order`, создайте специализированный класс.

C#

```
class CateringOrder : Order
{
    public int MinimumGuests { get; }
    public string? ApprovedBy { get; private set; }

    public CateringOrder(int minimumGuests) => MinimumGuests = minimumGuests;

    public void Approve(string manager) => ApprovedBy = manager;

    public override string Status
    {
        get => base.Status;
        set
        {
            if (value == "Ready" && ApprovedBy is null)
                throw new InvalidOperationException(
                    "A catering order requires manager approval before it can be
marked ready.");
            base.Status = value;
        }
```

```
    }

    public override string ToString() =>
        $"Catering [{Status}] for {MinimumGuests}+ guests, approved by: {ApprovedBy ??
"(none)"} - Total: {Total:F2}";
}
```

`CateringOrder` использует повторно `AddItem` и `Total` из базового класса. Переопределение `Status` ужесточает контракт— вызов `Status = "Ready"` без предварительного утверждения вызывает исключение:

C#

```
Console.WriteLine("\n=== Inheritance: CateringOrder ===");
var catering = new CateringOrder(minimumGuests: 20);
catering.AddItem("Coffee (serves 20)", 45.00m);
catering.AddItem("Pastry platter", 60.00m);

try
{
    catering.Status = "Ready";
}
catch (InvalidOperationException ex)
{
    Console.WriteLine($"Blocked: {ex.Message}");
}

catering.Approve("Sam");
catering.Status = "Ready";
Console.WriteLine(catering);
```

Этот производный класс иллюстрирует три понятия наследования:

- **Добавлено состояние**: `MinimumGuests` и `ApprovedBy` существует только в производном классе.
- **Добавлено поведение**: `Approve` это новое— база `Order` не знает о утверждениях.
- **Переопределенное поведение**: `Status` сеттер применяет бизнес-правило, которое отсутствует в базовом классе.

Наследование подходит, когда новый тип *является специализированной* версией базового типа, и необходимо повторно использовать существующее состояние и поведение при добавлении или ужесточении правил. Общий базовый класс является более естественным, чем интерфейс, когда типы совместно используют реализацию, а не только контракт.

Использование интерфейса для определения общих возможностей

Кофейня проводит различные акции — время скидок, бонусы для постоянных клиентов, сезонные предложения. Процесс оформления заказа должен применять доступные сегодня скидки, не зная особенности каждого правила. Вам нужен способ сказать "что-нибудь, что может применить скидку" без привязки к одному классу.

C#

```
interface IDiscountPolicy
{
    decimal Apply(decimal total);
}

class HappyHourDiscount : IDiscountPolicy
{
    public decimal Apply(decimal total) => total * 0.80m;
}

class LoyaltyDiscount : IDiscountPolicy
{
    public decimal Apply(decimal total) => total - 1.00m;
}
```

Метод `Checkout` принимает любой `IDiscountPolicy`, поэтому вы можете вводить новые политики без изменения логики оформления заказа:

C#

```
static decimal Checkout(decimal total, IDiscountPolicy policy) => policy.Apply(total);

Console.WriteLine("\n=== Interface: discount policy ===");
decimal subtotal = 12.00m;
Console.WriteLine($"Happy hour (20% off): {Checkout(subtotal, new
HappyHourDiscount()):F2}");
Console.WriteLine($"Loyalty ($1 off):     {Checkout(subtotal, new
LoyaltyDiscount()):F2}");
```

Интерфейс объявляет контракт — набор элементов, которые должен предоставлять любой тип реализации. Интерфейс работает здесь, так как типы скидок не связаны (они не имеют общего базового класса), но при этом оформлению заказа необходимо обработать их единообразно. Интерфейсы также упрощают тестирование: замена политики на заглушку без изменения продуктового кода. Дополнительные сведения см. в разделе "Интерфейсы".

Развивайте свои варианты выбора типов

Ни одно из этих решений не является постоянным. На самом деле, вы можете легко изменить их перед выпуском библиотеки, в которой критические изменения становятся важными. По мере роста требований улучшайте простой тип до более сложного. Вот три распространенных эволюции.

Кортеж → запись: группировка продолжает появляться

Кортеж `GetDailySummary` работает хорошо внутри одного метода, но когда вы начнете передавать его в отчеты, панели мониторинга и тесты, именованный тип данных оправдывает свое использование. Повышение кортежа до записи и добавление вычисляемых свойств:

C#

```
record class DailySummary(int TotalOrders, decimal Revenue)
{
    public decimal AverageTicket => TotalOrders > 0 ? Revenue / TotalOrders : 0m;
}
```

Вызывающие, которые ранее деконструировали кортеж, теперь получают `ToString()` бесплатно, равенство значений и естественное место для производных данных, таких как `AverageTicket`:

C#

```
Console.WriteLine("\n=== Evolve: tuple -> record ===");
var daily = new DailySummary(120, 525.75m);
Console.WriteLine(daily);
Console.WriteLine($"Average ticket: {daily.AverageTicket:F2}");
```

Структура → класс: необходимо наследование

Команда по обслуживанию магазина просит калиброванные показания: значение датчика, скорректированное смещением. Структуры `Measurement` записи отлично подходят для необработанных данных, но структуры не поддерживают наследование, поэтому вы не можете произвести калиброванный вариант. Повышение уровня иерархии классов:

C#

```
class SensorReading(double value, string unit)
{
    public double Value { get; } = value;
    public string Unit { get; } = unit;
```

```
    public virtual string Display() => $"{Value}{Unit}";
}

class CalibratedReading(double value, string unit, double offset)
    : SensorReading(value, unit)
{
    public double Offset { get; } = offset;

    public override string Display() => $"{Value + Offset}{Unit} (offset
{Offset:+0.0;-0.0})";
}
```

`CalibratedReading` наследует от `SensorReading` и переопределяет `Display()` чтобы включить смещение. Этот шаблон невозможно использовать с структурой или структурой записей:

C#

```
Console.WriteLine("\n=== Evolve: struct -> class ===");
var raw = new SensorReading(72.5, "°F");
var calibrated = new CalibratedReading(72.5, "°F", offset: -0.3);

Console.WriteLine($"Raw:        {raw.Display()}");
Console.WriteLine($"Calibrated: {calibrated.Display()}");
```

Класс → класс + интерфейс: требуется полиморфизм между типами

Класс `Order` хорошо работает сам по себе, но как только `CateringOrder` существует, оформление заказа, отчетность и печать должны обрабатывать *любой* заказ без учета того, к какому конкретному типу он принадлежит. Извлечение интерфейса с элементами, от которые вызывающие абоненты фактически зависят:

C#

```
interface IOrder
{
    string Status { get; set; }
    decimal Total { get; }
}
```

Как `Order`, так и `CateringOrder` уже удовлетворяют этому контракту. Теперь один метод обрабатывает любой тип:

C#

```
Console.WriteLine("\n=== Evolve: class -> class + interface ===");
static void PrintOrderSummary(IOrder o) =>
    Console.WriteLine($"  {o.Total:F2} [{o.Status}]");

var walkIn = new Order();
walkIn.AddItem("Mocha", 5.00m);
walkIn.Status = "Ready";

var banquet = new CateringOrder(minimumGuests: 50);
banquet.AddItem("Coffee service", 90.00m);
banquet.Approve("Alex");
banquet.Status = "Ready";

Console.WriteLine("All orders:");
foreach (IOrder o in new IOrder[] { walkIn, banquet })
    PrintOrderSummary(o);
```

Извлечение интерфейса не изменяет `Order` или `CateringOrder` — оно просто делает их общую структуру явной, что также упрощает тестирование.

Краткое руководство по принятию решений

Используйте эту таблицу в качестве отправной точки, если вы не уверены, какой тип следует выбрать:

Развернуть таблицу

Question	Оптимальный сценарий
Возврат нескольких значений из одного метода?	Tuple
Неизменяемые данные, где равенство определяется значениями?	Класс Record
Маленькие, копируемые данные значений с проверкой на равенство?	Структура записи
Изменяемое состояние, поведение или идентичность ссылки?	Class
Специализированная версия существующего класса?	Производный класс
Общая функция у несвязанных типов?	Interface

Если ни один из этих подходов не подходит полностью, рассмотрите возможность объединения типов. Например, класс может реализовать интерфейс, а запись может быть структурой. Полное сравнение см. в разделе "Выбор типа".

Связанный контент

- Типы кортежей
- Записи
- Типы структур
- Классы, структуры и записи
- Интерфейсы
- Выберите тип типа

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 07.05.2026

Руководство. Использование сопоставления шаблонов для создания управляемых типами и управляемых данными алгоритмов

Статья • 25.03.2025

Вы можете написать функции, которые работают так, будто вы расширили типы, которые могут быть в других библиотеках. Еще один вариант использования шаблонов — создать функции, которые требуются для приложения и не являются фундаментальными функциями для расширяемого типа.

Из этого руководства вы узнаете, как выполнять следующие задачи:

- ✓ распознавать случаи, в которых следует использовать сопоставление шаблонов;
- ✓ использовать выражения сопоставления шаблонов для реализации поведения с учетом типов и значений свойств;
- ✓ комбинировать сопоставление шаблонов и другие методы для создания полных алгоритмов.

Предварительные условия

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Инструкции по установке

В Windows этот файл конфигурации WinGet используется для установки всех необходимых компонентов. Если у вас уже установлено что-то, WinGet пропустит этот шаг.

1. Скачайте файл и дважды щелкните его, чтобы запустить его.
2. Прочитайте лицензионное соглашение, введите `и`, и выберите `ввод` при появлении запроса на принятие.
3. Если на панели задач появится мигающий запрос контроля учетных записей пользователей (UAC), разрешите установку продолжить.

На других платформах необходимо установить каждый из этих компонентов отдельно.

1. Скачайте рекомендуемый установщик на странице загрузки пакета SDK для .NET и дважды щелкните его, чтобы запустить его. Страница загрузки обнаруживает платформу и рекомендует последний установщик для вашей платформы.
2. Скачайте последнюю версию установщика на домашней странице Visual Studio Code и дважды щелкните его, чтобы запустить его. Эта страница также обнаруживает платформу, а ссылка должна быть правильной для вашей системы.
3. Нажмите кнопку "Установить" на странице расширения C# DevKit. Откроется код Visual Studio и запрашивается, нужно ли установить или включить расширение. Выберите "Установить".

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Сценарии для сопоставления шаблонов

Современная разработка часто предусматривает использование данных из нескольких источников, а также представление информации и идей на основе этих данных в одном связном приложении. У вас и вашей команды не всегда будет возможность контроля над всеми типами входящих данных или доступа к ним.

Для классического объектно-ориентированного приложения необходимо создавать в приложении типы данных, которые представляют каждый тип данных из этих нескольких источников. Затем ваше приложение будет работать с этими новыми типами, создавать иерархии наследования и виртуальные методы, а также реализовывать абстракции. Эти методы работают, а иногда они лучшие инструменты. Но в некоторых случаях можно писать меньше кода. Вы можете писать более понятный код, используя методы, которые разделяют сами данные и операции с этими данными.

В этом руководстве описано, как создать и оценить приложение, которое принимает входящие данные из нескольких внешних источников для одного сценария. Вы увидите, что **сопоставление шаблонов** позволяет эффективно использовать и обрабатывать данные такими способами, которые изначально не были частью системы.

Рассмотрим крупный город, в котором для управления трафиком используются дорожные сборы и тарификация на основе пиковой загрузки. Вы напишете

приложение, которое рассчитывает плату за автомобиль в зависимости от его типа. Затем усовершенствования включают цены, основанные на количестве пассажиров в автомобиле. Дальнейшие улучшения добавляют ценообразование на основе времени и дня недели.

Из этого краткого описания вы можете быстро составить иерархию объектов для моделирования этой системы. Но ваши данные поступают из разных источников, включая другие системы управления регистрацией транспортных средств. Эти системы предоставляют разные классы для моделирования таких данных, и у вас нет единой объектной модели, которую можно использовать. При работе с этим руководством для моделирования данных автомобиля вы будете использовать упрощенные классы из этих внешних систем, как показано в следующем примере кода:

C#

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

Скачать начальный код можно из репозитория GitHub dotnet/samples. Вы можете видеть, что классы транспортных средств принадлежат разным системам и

находятся в разных пространствах имен. Другой общий базовый класс, кроме `System.Object`, использовать нельзя.

Схемы сопоставления шаблонов

Сценарий, используемый в этом руководстве, позволяет выделить виды проблем, для решения которых подходит сопоставление шаблонов.

- Объекты, с которыми вам нужно работать, не находятся в иерархии объектов, которая соответствует вашим целям. Вам может потребоваться работать с классами, которые являются частью несвязанных систем.
- Функции, которые вы добавляете, не является частью основной абстракции для этих классов. Плата за автомобиль *изменяется* в зависимости от его типа. При этом плата не является основной функцией этого автомобиля.

Если фигура данных и *операции* с данными не описаны вместе, функции сопоставления шаблонов в C# упрощают работу.

Реализация расчетов базового сбора

Самый простой расчет базового сбора выполняется с учетом типа автомобиля:

- `Car` — 2,00 дол. США.
- `Taxi` — $3,50 дол. США.
- `Bus` — 5,00 долларов.
- `DeliveryTruck` — 10,00 долларов США.

Создайте класс `TollCalculator` и реализуйте сопоставление шаблонов по типу автомобиля, чтобы получить сумму сбора. В приведенном ниже примере кода показана начальная реализация класса `TollCalculator`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
```

```
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

В предыдущем коде используется switch выражение (не то же самое, что switch инструкция), которое проверяет шаблон объявления. **Выражение switch** начинается с переменной `vehicle` в приведенном выше коде, за которой следует ключевое слово `switch`. Далее следуют все **ветви switch** внутри фигурных скобок. Выражение `switch` вносит другие уточнения в синтаксис, который окружает оператор `switch`. Ключевое слово `case` опущено, и результатом каждой ветви является выражение. Последние две ветви демонстрируют новую функцию языка. Случай `{ }` соответствует любому ненулевому объекту, который не совпадает с предыдущей ветвью. Этот блок перехватывает неправильные типы, переданные этому методу. `{ }` случай должен соответствовать случаям для каждого типа транспортного средства. Если порядок был бы обратным, случай `{ }` имел бы преимущество. Наконец, `null` определяет, когда этому методу передается `null`. Шаблон `null` может быть последним, так как другие шаблоны соответствуют только ненулевому объекту правильного типа.

Этот код можно проверить с помощью следующего кода в файле `Program.cs`:

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
```

```
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}
```

Этот код включен в начальный проект, но закомментирован. Удалите комментарии, и вы можете проверить, что вы написали.

На этом примере можно понять, как шаблоны помогают создавать алгоритмы, в которых код и данные разделены. Выражение `switch` проверяет тип и создает различные значения в зависимости от результатов. Но это только начало.

Добавить ценообразование в зависимости от заполненности

Орган, взимающий сбор, поощряет передвижение автомобилей с максимальным количеством пассажиров. Следовательно, плата за проезд возрастает, когда в автомобиле находится меньше пассажиров, и действуют скидки для полных автомобилей, чтобы стимулировать их использование.

- Автомобили и такси без пассажиров должны заплатить дополнительные 0,50 долл. США.
- Автомобили и такси с двумя пассажирами получают скидку 0,50 долл. США.
- Автомобили и такси с тремя пассажирами получают скидку 1,00 долл. США.
- Если автобус заполнен менее чем на половину, взимается дополнительная плата — 2,00 дол. США.
- Если автобус заполнен более чем на 90 %, действует скидка 1,00 дол. США.

Эти правила можно реализовать с помощью шаблона свойств в этом же выражении switch. Шаблон свойства позволяет сравнивать значение свойства с

постоянным значением. Шаблон свойств проверяет свойства объекта после определения его типа. Один случай для `Car` включает четыре разных случая:

```
C#

vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                 => 2.00m - 1.0m,

    // ...
};
```

Первые три случая проверяют тип как `Car`, а затем проверяют значение свойства `Passengers`. Если оба совпадают, вычисленное выражение возвращается.

Вам также следует расширить случаи для такси аналогичным образом:

```
C#

vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
};
```

Затем реализуйте правила заполняемости, расширив случаи для автобусов, как показано в следующем примере:

```
C#

vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,
```

```
    // ...
};
```

Орган, взимающий сборы, не учитывает количество пассажиров в грузовых автомобилях, выполняющих доставку. Вместо этого сумма сбора корректируется с учетом веса грузовых автомобилей указанным ниже образом.

- Для грузовых автомобилей весом более 2268 кг взимается дополнительная плата в размере 5,00 долл. США.
- Для легких грузовиков весом до 1361 кг применяется скидка 2,00 долл. США.

Это правило, реализуется с помощью следующего кода:

C#

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

В этом коде показано предложение `when` ветви switch. Используйте оператор `when`, чтобы проверять условия свойства, отличные от равенства. Завершив работу, вы получите метод, который выглядит примерно как в следующем коде:

C#

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car                        => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
```

```
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

Многие из этих переключающих рычагов являются примерами **рекурсивных шаблонов**. Например, в `Car { Passengers: 1}` показан шаблон константы внутри шаблона свойств.

Вы можете уменьшить количество повторяемых участков кода, использовав вложенные операторы switch. В предыдущих примерах объекты `Car` и `Taxi` имеют по четыре разные руки. В обоих случаях вы можете создать шаблон объявления, который передается в шаблон константы. Пример использования этого способа показан в следующем коде:

C#

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
```

```
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

В предыдущем примере применение рекурсивного выражения позволяет не использовать повторно ветви `Car` и `Taxi` с дочерними ветвями, которые проверяют значение свойства. Этот метод не используется для ветвей `Bus` и `DeliveryTruck`, так как они представляют собой проверочные диапазоны для свойства, а не дискретные значения.

Добавить тарифы на пиковые периоды

Для последней функции властный орган по взиманию сборов хочет добавить ценообразование, зависящее от времени пиковых нагрузок. Утром и вечером, когда дороги наиболее загружены, сумма сбора удваивается. Это правило влияет только на движение в одном направлении: при въезде в город утром и на выезде вечером в час пик. В другое время в течение рабочего дня плата увеличивается на 50 %. Поздно ночью и ранним утром плата уменьшается на 25 %. В выходные дни взимается стандартная плата, независимо от времени суток. Вы можете использовать ряд `if` и `else` операторов, чтобы выразить это с помощью следующего кода:

C#

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
```

```
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

Приведенный выше код работает правильно, но трудно читается. Чтобы понять код, необходимо последовательно пройти через все входные примеры и вложенные операторы `if`. Вместо этого для данной функции вы будете использовать сопоставление шаблонов, но вместе с другими методами. Вы можете создать одно выражение для сопоставления шаблонов, учитывающее все комбинации направления, дня недели и времени. Результат будет представлен в виде сложного выражения. Было бы трудно читать и трудно понять. В таком случает будет сложнее обеспечить правильность результатов. Но вы можете объединить эти методы для создания набора значений, который кратко описывает все эти состояния. Затем, используйте сопоставление шаблонов, чтобы вычислить множитель для платы за проезд. Кортеж содержит три дискретные условия:

- День является либо будним, либо выходным.
- Диапазон времени, за который взимается плата.
- Направление в город или за город.

В таблице ниже показаны комбинации входных значений и множителя для цены в часы пик:

Развернуть таблицу

день	Время	Направление	Премиум
Будний день	Утренний час пик	входящий трафик	x 2,00

день	Время	Направление	Премиум
Будний день	Утренний час пик	Исходящий	x 1,00
будний день	Дневное время	входящий трафик	x 1,50
Будний день	Дневное время	Исходящий	x 1,50
Будний день	Вечерний час пик	входящий трафик	x 1,00
будний день	Вечерний час пик	Исходящий	x 2,00
Будний день	Ночное время	входящий трафик	x 0,75
Будний день	Ночное время	Исходящий	x 0,75
Выходные	Утренний час пик	входящий трафик	x 1,00
Выходные	Утренний час пик	Исходящий	x 1,00
Выходные	Дневное время	входящий трафик	x 1,00
Выходные	Дневное время	Исходящий	x 1,00
Выходные	Вечерний час пик	входящий трафик	x 1,00
Выходные	Вечерний час пик	Исходящий	x 1,00
Выходные	Ночное время	входящий трафик	x 1,00
Выходные	Ночное время	Исходящий	x 1,00

Для этих трех переменных существует 16 разных комбинаций. Комбинируя некоторые условия, вы упростите окончательное выражение switch.

В системе для сбора платы структура DateTime используется для определения времени сбора. Создайте методы-члены, которые создают переменные на основе предыдущей таблицы. В следующей функции для сопоставления шаблонов используется выражение switch, позволяющее определить, соответствует ли DateTime выходным или рабочим дням недели:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
```

```
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

Этот метод работает правильно, но фрагменты кода повторяются. Вы можете упростить его, как показано в следующем примере:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Затем добавьте аналогичную функцию, чтобы категоризировать время на блоки:

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

Чтобы преобразовать каждый диапазон времени в дискретное значение, добавьте приватное `enum`. `GetTimeBand` Затем метод использует реляционные шаблоны и конъюнктивные orшаблоны. Реляционный шаблон позволяет проверить числовое значение с помощью операторов `<`, `>`, `<=` или `>=`. Шаблон `or` проверяет, соответствует ли выражение одному или нескольким шаблонам. Можно также использовать шаблон `and`, чтобы проверить, что выражение соответствует двум различным шаблонам, и шаблон `not` для проверки того, что выражение не соответствует шаблону.

Создав эти методы, можно использовать другое выражение `switch` с **шаблоном кортежа** для вычисления премиум-цены. Вы можете записать выражение `switch` со всеми 16 ветвями.

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

Код выше работает, но его можно упростить. Плата за все восемь комбинаций на выходные дни одинаковая. Вы можете заменить все восемь вариантов одной строкой:

C#

```
(false, _, _) => 1.0m,
```

Для входящего и исходящего трафика используется одинаковый множитель в дневное и ночное время в рабочие дни. Эти четыре рычага переключателя можно заменить следующими двумя строками.

C#

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

После этих двух изменений код должен выглядеть как показано ниже:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Наконец, вы можете удалить два часа пик, за которые взимается обычная плата. Удалив эти ветви, можно заменить `false` пустой переменной (`_`) в последней ветви switch. У вас получится следующий законченный метод:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
```

В этом примере продемонстрировано одно из преимуществ сопоставления шаблонов: ветви шаблона вычисляются по порядку. Если поменять их местами так, чтобы более ранняя ветвь обрабатывала один из более поздних случаев, компилятор предупредит вас о недостижимом коде. Эти правила языка облегчают реализацию упрощений, выполненных выше, и гарантируют, что код не изменится.

Сопоставление шаблонов делает код некоторых типов более удобочитаемым и предлагает альтернативу объектно-ориентированным методам, когда добавление кода в классы невозможно. Разделение облака приводит к тому, что данные и функциональность существуют отдельно. *Форма* данных и *операции* с ними не обязательно описываются вместе. В этом руководстве вы использовали существующие данные совершенно другим образом, чем они предназначены для использования изначально. Сопоставление шаблонов дало вам возможность создавать функциональность, которая переопределяла эти типы, даже если вы не могли их расширить.

Следующие шаги

Скачать готовый код можно из репозитория GitHub dotnet/samples. Изучите шаблоны самостоятельно и используйте эту методику во время написания кода. Изучение этих методов дает вам возможность по-новому подойти к решению проблем и создавать новую функциональность.

См. также

- Шаблоны
- Выражение switch

Использование типов записей

> **Совет**
>
> **Вы новичок в разработке программного обеспечения?** Сначала начните с учебников для начинающих. Вы освоитесь с классами, методами и циклами.
>
> **Есть опыт на другом языке?** В этом руководстве рассматриваются функции записи C#, которые вы используете каждый день: равенство значений, позиционный синтаксис и `with` выражения.

В этом руководстве вы создадите консольное приложение, которое моделирует ежедневные температуры с помощью записей и структур записей.

В этом руководстве вы узнаете, как:

- Объявляйте позиционные записи и структуры записей.
- Создайте небольшую иерархию записей.
- Используйте равенство и форматирование, созданные компилятором.
- Используйте `with` выражения для недеструктивной мутации.

Необходимые условия

- Последняя версия .NET SDK
- редактор Visual Studio Code
- C# DevKit

Инструкции по установке

В Windows данный файл конфигурации WinGet используется для установки всех необходимых компонентов. Если у вас уже установлено что-то, WinGet пропустит этот шаг.

1. Скачайте файл и дважды щелкните его, чтобы запустить его.
2. Прочитайте лицензионное соглашение, введите `и`, и выберите `ввод` при появлении запроса на принятие.
3. Если на панели задач появится мигающий запрос контроля учетных записей пользователей (UAC), разрешите установку продолжить.

На других платформах необходимо установить каждый из этих компонентов отдельно.

1. Скачайте рекомендуемый установщик на странице загрузки пакета SDK для .NET и дважды щелкните его, чтобы запустить его. Страница загрузки обнаруживает платформу и рекомендует последний установщик для вашей платформы.
2. Скачайте последнюю версию установщика на домашней странице Visual Studio Code и дважды щелкните его, чтобы запустить его. Эта страница также обнаруживает платформу, а ссылка должна быть правильной для вашей системы.
3. Нажмите кнопку "Установить" на странице расширения C# DevKit. Откроется код Visual Studio и запрашивается, нужно ли установить или включить расширение. Выберите "Установить".

Создание приложения и первой записи

Создайте папку для приложения, запустите `dotnet new console` и откройте созданный проект.

Добавьте файл с именем `DailyTemperature.cs` и добавьте позиционный `readonly record struct` параметр для значений температуры:

C#

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Добавьте файл с именем `Program.cs` и создайте примеры данных температуры:

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
```

```
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
];
```

Этот синтаксис обеспечивает краткое моделирование данных с неизменяемой семантикой значений.

Добавьте поведение в структуру записи

В `DailyTemperature.cs` структуре записи уже есть вычисленное `Mean` свойство:

C#

```
public double Mean => (HighTemp + LowTemp) / 2.0;
```

Структура record хорошо работает здесь, так как каждое значение небольшое и независимое.

Создайте типы записей для расчетов градусо-дней

> ⓘ **Примечание.**
>
> **Градусо-дни отопления** и **градусо-дни охлаждения** измеряют, насколько средняя дневная температура отклоняется от базовой температуры (обычно 65°F/18°C). Градусо-дни отопления накапливаются в холодные дни, когда среднее значение ниже базы, в то время как градусо-дни охлаждения накапливаются в теплые дни, когда среднее значение выше базы. Эти вычисления помогают оценить потребление энергии для нагрева или охлаждения зданий, что делает их полезными для коммунальных компаний, руководителей зданий и анализа климата.

Создайте файл с именем `DegreeDays.cs` с иерархией для расчета градусо-дней нагрева и охлаждения.

C#

```
public abstract record DegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature,
```

```
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean < BaseTemperature).Sum(s
=> BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean > BaseTemperature).Sum(s
=> s.Mean - BaseTemperature);
}
```

Теперь вычислите суммы с помощью метода `Main` в `Program.cs`:

C#

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

Сгенерированные `ToString` данные полезны для быстрой диагностики в процессе итерации.

Переопределите `PrintMembers`, чтобы настроить выход.

Если выходные данные по умолчанию содержат слишком много шума, переопределите `PrintMembers` в базовой записи:

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

Переопределение сохраняет выходные данные, ориентированные на необходимые сведения.

Использование с выражениями для недеструктивной мутации

Используется `with` для создания измененных копий без изменения исходной записи:

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Расширьте эту идею, чтобы вычислить скользящие итоги из срезов входных данных:

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start +
rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

Этот подход полезен при необходимости преобразований при сохранении исходных значений.

Дальнейшие действия

- Ознакомьтесь с типами записей C# , чтобы получить более подробные рекомендации.
- Продолжайте использовать объектно-ориентированный C# для более широких шаблонов проектирования.
- Изучите типы преобразования для объединения записей с безопасными шаблонами преобразования.

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 01.05.2026

Обработка исключения с помощью try/catch

Блок try-catch- предназначен для перехвата и обработки исключений, происходящих в исполняемом коде. Некоторые исключения могут обрабатываться в блоке `catch`, и проблема решается без повторного вызова исключения. Но в большинстве случаев на этом этапе можно только проверить, что вызывается подходящее исключение.

Пример

В этом примере IndexOutOfRangeException не является наиболее подходящим исключением. Для данного метода больше подходит исключение ArgumentOutOfRangeException, так как ошибка вызывается аргументом `index`, который передает вызывающая функция.

C#

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)  // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of range.",
e);
    }
}
```

Комментарии

Код, вызывающий исключение, находится в блоке `try`. Оператор `catch` добавляется сразу после него для обработки `IndexOutOfRangeException`, если она возникает. Блок `catch` обрабатывает `IndexOutOfRangeException` и выбрасывает более подходящий `ArgumentOutOfRangeException`. Чтобы вызывающий объект получил максимально подробную информацию, рекомендуется указать исходное исключение в качестве значения InnerException нового исключения. Так как свойство InnerException доступно только для чтения, его значение необходимо присваивать только в конструкторе нового исключения.

Last updated on 30.03.2026

Практическое руководство. Выполнение кода очистки с использованием блока finally

Статья • 22.04.2023

Оператор `finally` позволяет гарантировать, что необходимая очистка объектов, как правило, объектов, занимающих внешние ресурсы, возникает немедленно, даже при создании исключения. Примером подобной очистки является вызов Close для FileStream сразу после использования вместо ожидания сборки мусора, выполняемой для объекта средой CLR, следующим образом:

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Пример

Чтобы преобразовать предыдущий код в оператор `try-catch-finally`, код очистки отделяется от рабочего кода следующим образом:

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
```

```
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        file?.Close();
    }
}
```

Так как исключение в блоке `try` может возникнуть в любой момент до вызова `OpenWrite()` или может произойти сбой самого вызова `OpenWrite()`, мы не можем гарантировать, что файл, который мы попытаемся закрыть, будет в это время открыт. Блок `finally` добавляет проверку того, что объект FileStream — не `null`, перед вызовом метода Close. Без проверки `null` блок `finally` может вызывать собственное исключение NullReferenceException, однако вызова исключений в блоках `finally` следует по возможности избегать.

Подключение к базе данных — еще один подходящий кандидат для заключения в блок `finally`. Так как количество разрешенных подключений к серверу базы данных иногда ограничено, закрывать подключения к базе данных рекомендуется как можно быстрее. Если перед закрытием подключения возникает исключение, рекомендуется использовать блок `finally`, а не ожидать сборки мусора.

См. также раздел

- Оператор using
- Операторы обработки исключений

Новые возможности C# 15

C# 15 включает следующие новые функции. Попробуйте эти функции, используя последнюю инсайдерскую версию Visual Studio 2026 или предварительный пакет SDK .NET 11:

- Аргументы выражения коллекции
- Типы объединения
- Закрытые иерархии
- Индексаторы расширений
- Помечены break и continue
- Безопасность памяти

C# 15 — это последний выпуск предварительной версии C#. Предварительная версия .NET 11 поддерживает C# 15. Дополнительные сведения см. в разделе версии языка C# .

Вы можете скачать последний пакет SDK предварительной версии .NET 11 на странице загрузки .NET. Вы также можете скачать инсайдерскую версию Visual Studio 2026, включая предварительную версию SDK для .NET 11.

Страница "Новые возможности C#" добавляет новые функции, когда они доступны в общедоступных предварительных версиях. Раздел рабочего набора на странице статуса функций Roslyn отслеживает, когда предстоящие функции объединяются в основную ветвь.

Критические изменения, внесенные в C# 15, можно найти в нашей статье о критических изменениях.

> ⓘ **Примечание.**
>
> Мы заинтересованы в ваших отзывах об этих функциях. Если вы найдете проблемы с любым из этих новых функций, создайте **новую проблему** в репозитории **dotnet/roslyn**.

Аргументы выражения коллекции

Аргументы можно передать в конструктор подлежащей коллекции или фабричный метод, используя элемент `with(...)` в качестве первого элемента в выражении коллекции. Эта

возможность позволяет вам указывать емкость, компараторы или другие параметры конструктора непосредственно в синтаксисе выражения коллекции.

В следующем примере показано, как передать аргумент емкости конструктору `List<T>` и компаратору `HashSet<T>`:

C#

```
string[] values = ["one", "two", "three"];

// Pass capacity argument to List<T> constructor
List<string> names = [with(capacity: values.Length * 2), .. values];

// Pass comparer argument to HashSet<T> constructor
HashSet<string> set = [with(StringComparer.OrdinalIgnoreCase), "Hello", "HELLO",
"hello"];
// set contains only one element because all strings are equal with OrdinalIgnoreCase
```

Дополнительные сведения о аргументах выражений коллекции см. в статье справочника по языку о выражениях коллекции или спецификации компонентов. Сведения об использовании аргументов выражения коллекции в инициализаторах коллекции см. в разделе "Инициализаторы объектов и коллекций".

Типы объединения

C# 15 представляет *типы объединения*, которые означают значение, являющееся одним из нескольких *типов-кейсов*. Объявите объединение с ключевым словом `union` :

C#

```
public record class Cat(string Name);
public record class Dog(string Name);
public record class Bird(string Name);

public union Pet(Cat, Dog, Bird);
```

Объединения предоставляют неявные преобразования из каждого типа варианта, и компилятор гарантирует, что `switch` выражения охватывают все варианты типа:

C#

```
Pet pet = new Dog("Rex");

string name = pet switch
{
    Dog d => d.Name,
```

```
    Cat c => c.Name,
    Bird b => b.Name,
};
```

Среда выполнения включает типы `UnionAttribute` и `IUnion` начиная с .NET 11 preview 5. Некоторые функции из спецификации предложения пока не реализованы. Эти функции будут доступны в будущих предварительных версиях.

Дополнительные сведения см. в разделе "Типы профсоюзов " в справочнике по языку или спецификации компонентов.

Закрытые иерархии

Начиная с C# 15, к классу можно применить модификатор `closed`, чтобы объявить *закрытую иерархию*. Закрытый класс может быть производным только от декларируемой сборки, которая исправляет набор прямых потомков во время компиляции:

C#

```
public closed record class GateState;
public record class Closed : GateState;
public record class Open(float Percent) : GateState;
```

Поскольку компилятор знает всех прямых потомков, `switch` выражение, которое обрабатывает все из них, является полным и не требует ветви по умолчанию:

C#

```
string Describe(GateState state) => state switch
{
    Closed => "closed",
    Open(var percent) => $"{percent}% open",
    // No warning: every direct descendant of 'GateState' is handled.
};
```

Модификатор `closed` — это контекстное ключевое слово. Класс `closed` неявно является `abstract` и не может сочетаться с `sealed`, `static` или явным модификатором `abstract`. Производный не является транзитивным: не закрытый потомок закрытого класса по-прежнему может быть производным от других сборок. Чтобы расширить исчерпывающую проверку иерархии, пометьте промежуточных потомков `closed` .

Дополнительные сведения см. в шаблонах закрытой модификатора и закрытой иерархии в справочнике по языку или спецификации функции.

Индексаторы расширений

Начиная с C# 15, можно объявить *индексаторы* в блоке `extension` . Индексаторы расширений позволяют индексировать приемник, как будто индексатор был объявлен в типе приемника. Так как индексаторы всегда являются элементами экземпляров, блок расширения, объявляющий индексатор, должен предоставить именованный параметр приемника.

В следующем примере объявляется индексатор `IEnumerable<int>`, доступный только для чтения, который возвращает элемент в указанной позиции:

C#

```
public static class SequenceIndexer
{
    extension(IEnumerable<int> sequence)
    {
        public int this[int index] => sequence.ElementAt(index);
    }
}
```

Индексировать приемник, как будто индексатор был членом типа приемника:

C#

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
int third = numbers[2];
```

Дополнительные сведения см. в объявлении расширения в справочнике по языку или спецификации компонентов.

Помечены `break` и `continue`

Начиная с C# 15, операторы `break` и `continue` могут указывать метку во внешней конструкции. Используйте метку `break` , чтобы выйти из включающего цикла или `switch` инструкции. Используйте метку `continue` , чтобы начать следующую итерацию включающего цикла.

Помеченные `break` и `continue` заменяют обходные приёмы, к которым иначе пришлось бы прибегнуть для управления потоком выполнения во вложенных циклах, например логический флаг, который устанавливают во внутреннем цикле, а затем проверяют на каждом внешнем уровне, или `goto`, который позволяет перескочить через циклы. Если

указать целевой цикл непосредственно в операторе перехода, это избавляет от лишних вспомогательных действий и делает подразумеваемый поток управления более понятным.

C#

```
outer: for (int row = 0; row < grid.Height; row++)
{
    for (int column = 0; column < grid.Width; column++)
    {
        if (grid[row, column].IsBlocked)
        {
            continue outer;
        }

        if (grid[row, column].IsGoal)
        {
            break outer;
        }
    }
}
```

Поместите метку непосредственно в цикл или оператор `switch`, к которым она относится. Без метки `break` и `continue` сохраняют своё исходное поведение и относятся к ближайшей применимой инструкции.

Правило стиля IDE0410 помечает логический флаг и `goto` шаблоны, которые может заменить помеченная инструкция перехода, и показывает примеры до и после каждого из них.

Дополнительные сведения см. в инструкциях Jump в справочнике по языку или спецификации компонентов.

Безопасность памяти

C# 15 начинает многорелизную инициативу по переосмыслению безопасности работы с памятью в языке. Цель заключается в том, чтобы связать `unsafe` контекст с операциями, которые фактически обращаются к неуправляемой памяти, а не к существованию типов указателей. Большинство уязвимостей безопасности памяти происходят из этих операций доступа, поэтому язык выделяет их для рецензентов и аудиторов.

При включении обновлённых правил `unsafe` у члена помечает его как *requires-unsafe*: обязанность по аудиту переходит к вызывающему коду, который должен использовать этот член в контексте `unsafe`. Компилятор записывает использование обновленных правил сборки с атрибутом `System.Runtime.CompilerServices.MemorySafetyRulesAttribute` . Модель

также добавляет контекстное ключевое слово `safe`, которое помечает члены и поля `extern` в типах с явной или расширенной компоновкой как безопасные. Вместе эти правила явно определяют границы потенциально небезопасных операций с памятью во всей программе.

Первый шаг включает в себя расслабление указателя. При компиляции с языковой `preview` версией следующие операции больше не требуют контекста `unsafe` :

- Объявление типа указателя и взятие адреса переменной с помощью оператора `&`.
- Инструкция `fixed` , которая закрепляет переменную.
- Преобразование `stackalloc` выражения в указатель.
- Оператор `sizeof` применяется к любому неуправляемому типу.

В следующем примере создается и закрепляется указатель без контекста `unsafe` :

C#

```
int number = 42;
int* pointer = &number;

int[] numbers = [10, 20, 30];
fixed (int* first = numbers)
{
    // Dereferencing the pointer still requires an unsafe context.
}
```

Операции, обращающиеся к памяти, на которую указывает указатель, такие как разыменование указателя (`*p`), доступ к члену через указатель (`p->member`), доступ к элементу через указатель (`p[i]`), а также вызов функции через указатель, по-прежнему требуют контекста `unsafe`.

C# 15 также добавляет выражение, `unsafe(expression)` которое устанавливает небезопасный `unsafe` контекст для одного выражения. Полезно, когда `unsafe` блок не может отображаться синтаксически, например инициализатор полей, инициализатор конструктора или `catch` фильтр:

C#

```
class Header
{
    // A field initializer can't contain an unsafe block, but it can contain an unsafe
expression.
    static readonly int Signature = unsafe(ReadSignature());

    static unsafe int ReadSignature()
    {
```

```
        int rawValue = 0x1234;
        int* pointer = &rawValue;
        return *pointer;
    }
}
```

Как и для остальных компонентов предварительной версии функции безопасности памяти, выражения `unsafe` требуют версию языка `preview` и параметр компилятора `AllowUnsafeBlocks`.

Компилятор также распознает `safe` контекстное ключевое слово как модификатор элементов `extern` и полей в явных или расширенных типах макетов. Установите `LangVersion` в значение `preview`, чтобы включить новый синтаксис и ослабление ограничений для указателей. Чтобы также включить обновленные правила, включая *небезопасные* обязательства вызывающего абонента, включите функцию `updated-memory-safety-rules` компилятора. Синтаксис программы на основе проекта и файлов см. в разделе "Включить обновленные правила безопасности памяти".

Дополнительные сведения см. в разделе "Небезопасный код", "типы указателей" и указатели функций в справочнике по языку или спецификации функций.

См. также

- Что нового в .NET 11

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 11.09.2026

Новые возможности C# 14

C# 14 включает следующие новые функции. Эти функции можно попробовать с помощью последней версии Visual Studio 2026 или пакета SDK для .NET 10:

- Члены расширения
- Условное присваивание null
- nameof поддерживает несвязанные универсальные типы
- Более неявные преобразования для Span<T> и ReadOnlySpan<T>
- Модификаторы простых лямбда-параметров
- field поддерживаемые свойства
- partial события и конструкторы
- определяемые пользователем операторы составных назначений
- новые директивы препроцессора для приложений на основе файлов

C# 14 — это последний выпуск C#. C# 14 поддерживается в **.NET 10**. Дополнительные сведения см. в разделе версии языка C# .

Скачать последний пакет SDK для .NET 10 можно на странице скачивания .NET. Вы также можете скачать Visual Studio 2026, который включает пакет SDK для .NET 10.

Критические изменения, внесенные в C# 14, можно найти в нашей статье о критических изменениях.

> **Заметка**
>
> Мы заинтересованы в ваших отзывах об этих функциях. Если вы найдете проблемы с любым из этих новых функций, создайте **новую проблему** в репозитории **dotnet/roslyn**.

Члены расширения

C# 14 добавляет новый синтаксис для определения *элементов расширения*. Новый синтаксис позволяет объявлять *свойства расширения* в дополнение к методам расширения. Можно также объявить члены расширения, которые расширяют сам тип, а не экземпляры этого типа. Другими словами, эти новые члены расширения могут отображаться как статические элементы расширенного типа. Эти расширения могут включать определяемые пользователем операторы, реализованные как статические методы расширения. В следующем примере кода показан пример различных типов элементов расширения, которые можно объявить:

C#

```
public static class Enumerable
{
    // Extension block
    extension<TSource>(IEnumerable<TSource> source) // extension members for
IEnumerable<TSource>
    {
        // Extension property:
        public bool IsEmpty => !source.Any();

        // Extension method:
        public IEnumerable<TSource> Where(Func<TSource, bool> predicate) { ... }
    }

    // extension block, with a receiver type only
    extension<TSource>(IEnumerable<TSource>) // static extension members for
IEnumerable<Source>
    {
        // static extension method:
        public static IEnumerable<TSource> Combine(IEnumerable<TSource> first,
IEnumerable<TSource> second) { ... }

        // static extension property:
        public static IEnumerable<TSource> Identity => Enumerable.Empty<TSource>();

        // static user defined operator:
        public static IEnumerable<TSource> operator + (IEnumerable<TSource> left,
IEnumerable<TSource> right) => left.Concat(right);
    }
}
```

Элементы в первом блоке расширения вызываются так, как будто они являются членами экземпляра `IEnumerable<TSource>`, например `sequence.IsEmpty`. Элементы во втором блоке расширения вызываются так, как будто они являются статическими элементами `IEnumerable<TSource>`, например `IEnumerable<int>.Identity`.

Дополнительные сведения см. в статье о членах расширений в руководстве по программированию, справочной статье extensionпо языку ключевого слова и спецификации компонентов для новых элементов расширения.

Ключевое слово `field`

Токен `field` позволяет вам написать тело доступа к свойству, не объявляя явное поле для хранения данных. Маркер `field` заменяется на поле синтезированной резервной копии компилятора.

Например, ранее, если вы хотели убедиться, что свойство `string` не может быть установлено в `null`, вам нужно было объявить резервное поле и реализовать оба аксессора.

C#

```
private string _msg;
public string Message
{
    get => _msg;
    set => _msg = value ?? throw new ArgumentNullException(nameof(value));
}
```

Теперь вы можете упростить код следующим образом:

C#

```
public string Message
{
    get;
    set => field = value ?? throw new ArgumentNullException(nameof(value));
}
```

Вы можете определить тело для одного или обоих акцессоров свойства, связанного с полем.

Возможны существенные изменения или путаница при чтении кода в типах, которые также содержат символ с именем `field`. Вы можете использовать `@field` или `this.field`, чтобы разрешить неоднозначность между `field` ключевым словом и идентификатором, или переименовать текущий `field` символ, чтобы обеспечить чёткое различие.

Неявные преобразования диапазона

C# 14 предоставляет первоклассную поддержку для System.Span<T> и System.ReadOnlySpan<T> в языке. Эта поддержка включает новые неявные преобразования, позволяющие более естественное программирование с этими типами.

`Span<T>` и `ReadOnlySpan<T>` используются различными ключевыми способами в C# и среде выполнения. Их введение повышает производительность без риска безопасности. C# 14 распознает связь и поддерживает некоторые преобразования между `ReadOnlySpan<T>`, `Span<T>` и `T[]`. Типы span могут выступать в качестве получателей методов расширения, комбинироваться с другими преобразованиями и помогать в ситуациях вывода универсальных типов.

Список неявных преобразований диапазона можно найти в статье о встроенных типах в разделе справочника по языку. Дополнительные сведения см. в спецификации функции для типов диапазонов первого класса.

Несвязанные обобщенные типы и `nameof`

Начиная с C# 14 аргумент `nameof` может быть несвязанным универсальным типом. Например, `nameof(List<>)` принимает значение `List`. В более ранних версиях C# для возврата `List<int>` имени можно использовать только закрытые универсальные типы, например `List`.

Простые лямбда-параметры с модификаторами

Модификаторы параметров, например `scoped`, , `ref`, `in` `out` или `ref readonly` лямбда-выражения, можно добавить без указания типа параметра:

C#

```
delegate bool TryParse<T>(string text, out T result);
// ...
TryParse<int> parse1 = (text, out result) => Int32.TryParse(text, out result);
```

Ранее добавление модификаторов было разрешено только в том случае, если объявления параметров включали типы параметров. В предыдущем объявлении требуется указывать типы для всех параметров.

C#

```
TryParse<int> parse2 = (string text, out int result) => Int32.TryParse(text, out
result);
```

Модификатор `params` по-прежнему требует явно типизированного списка параметров.

Дополнительные сведения об этих изменениях см. в статье об лямбда-выражениях в справочнике по языку C#.

Частично задействованные члены

Теперь вы можете объявлять конструкторы экземпляра и события как частичные элементы.

Частичные конструкторы и частичные события должны включать ровно одно *определение объявления* и одно *реализующее объявление*.

Только реализующее объявление частичного конструктора может включать инициализатор конструктора: `this()` или `base()`. Только одно объявление частичного типа может включать синтаксис основного конструктора.

Объявление реализации частичного события должно включать аксессоры `add` и `remove`. Определяющее объявление декларирует событие, похоже на поле.

Определяемое пользователем составное назначение

Подробнее см. в спецификации функции для пользовательского составного присваивания.

null-условное присваивание

Операторы условного доступа к члену `?.` и `?[]` теперь можно использовать на левой стороне назначения или составного назначения.

До C# 14 необходимо было проверять переменную на null, прежде чем присваивать значение свойству.

C#

```
if (customer is not null)
{
    customer.Order = GetCurrentOrder();
}
```

Вы можете упростить предыдущий код, используя оператор `?.`.

C#

```
customer?.Order = GetCurrentOrder();
```

Правая сторона оператора вычисляется только в том случае, если левая сторона `=` не имеет значения NULL. Если `customer` является null, код не вызывает `GetCurrentOrder`.

В дополнение к назначению можно использовать операторы доступа к условному члену NULL с составными операторами назначения (`+=`, `-=` и другими). Однако инкремент и

декремент, `++` и `--`, не разрешены.

Дополнительные сведения см. в справочной статье по языку о доступе к условному члену и спецификации компонентов для условного назначения null.

См. также

- Новые возможности .NET 10

Last updated on 18.11.2025

Новые возможности C# 13

C# 13 включает следующие новые функции. Эти функции можно попробовать с помощью последней версии Visual Studio 2022 или пакета SDK для .NET 9:

- коллекции params
- новые lock типа и семантики.
- новая последовательность выхода — \e.
- улучшения натурального типа группы методов
- неявное обращение к индексатору в инициализации объектов
- Включить ref локальные и unsafe контексты в итераторах и асинхронных методах
- Включить типы ref struct для реализации интерфейсов.
- Разрешить типы структур ссылок в качестве аргументов для параметров типа в универсальных шаблонах.
- Частичные свойства и индексаторы теперь разрешены в `partial` типах.
- приоритет разрешения перегрузки позволяет авторам библиотек определить одну перегрузку как предпочтительную по сравнению с другими.

Начиная с Visual Studio 17.12, C# 13 включает контекстное ключевое слово field в качестве функции предварительного просмотра.

C# 13 поддерживается в **.NET 9**. Дополнительные сведения см. в версии языка C# .

Вы можете скачать последний пакет SDK для .NET 9 на странице загрузки .NET . Вы также можете скачать Visual Studio 2022 , которая включает пакет SDK для .NET 9.

Критические изменения, внесенные в C# 13, можно найти в нашей статье о критических изменениях.

> **Заметка**
>
> Мы заинтересованы в ваших отзывах об этих функциях. Если вы найдете проблемы с любым из этих новых функций, создайте **новую проблему** в репозитории dotnet/roslyn .

коллекции `params`

Модификатор `params` не ограничивается типами массивов. Теперь можно использовать `params` с любым распознаваемым типом коллекции, включая System.Span<T>, System.ReadOnlySpan<T>и типы, реализующие System.Collections.Generic.IEnumerable<T> и имеющие метод `Add`. Помимо конкретных типов, интерфейсы

System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>, System.Collections.Generic.ICollection<T> и System.Collections.Generic.IList<T> также можно использовать.

При использовании типа интерфейса компилятор синтезирует хранилище для предоставленных аргументов. Дополнительные сведения см. в спецификации функций для коллекций params.

Например, объявления методов могут объявлять диапазоны как параметры `params`:

C#

```
public void Concat<T>(params ReadOnlySpan<T> items)
{
    for (int i = 0; i < items.Length; i++)
    {
        Console.Write(items[i]);
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

Новый объект блокировки

Среда выполнения .NET 9 включает новый тип для синхронизации потоков, тип System.Threading.Lock. Этот тип обеспечивает лучшую синхронизацию потоков через свой API. Метод Lock.EnterScope() входит в эксклюзивную область. `ref struct`, возвращенный из него, поддерживает шаблон `Dispose()` для выхода из эксклюзивной области.

Оператор C# lock распознает, является ли объект блокировки целевым объектом `Lock`. Если да, используется обновленный API вместо традиционного API, использующего System.Threading.Monitor. Компилятор также распознает, если преобразовать объект `Lock` в другой тип, а код на основе `Monitor` будет создан. Дополнительные сведения см. в спецификации функций для нового объекта блокировки .

Эта функция позволяет вам воспользоваться преимуществами нового типа библиотеки, изменив тип объекта `lock`, который вы используете. Никакого другого кода не требуется изменять.

Новая escape-последовательность

Вы можете использовать `\e` в качестве символьного литерала последовательности экранирования для `ESCAPE` символа Юникод `U+001B`. Ранее вы использовали `\u001b` или `\x1b`. Использование `\x1b` не рекомендуется, поскольку если следующие символы после `1b` являются допустимыми шестнадцатеричными цифрами, они становятся частью escape-последовательности.

Естественный тип группы методов

Эта функция делает небольшие оптимизации для разрешения перегрузки с участием групп методов. Группа методов — это совокупность метода и всех его перегрузок с одинаковым именем. Предыдущее поведение компилятора состояло в создании полного набора кандидатных методов для группы методов. Если требовался естественный тип, его определяли из полного набора кандидатных методов.

Новое поведение заключается в том, чтобы сужать набор кандидатных методов в рамках каждой области, удаляя те из них, которые не применимы. Как правило, удаленные методы — это универсальные методы с неправильной арностью или с ограничениями, которые не удовлетворены. Процесс продолжается до следующей внешней области, только если методы-кандидаты не найдены. Этот процесс более тесно соответствует общему алгоритму разрешения перегрузки. Если все кандидатные методы, найденные в заданной области, не соответствуют критериям, кандидатная группа методов не имеет естественного типа.

Подробные сведения об изменениях в спецификации предложения можно прочитать.

Неявный доступ к индексу

Неявный оператор индекса "from end" `^` теперь допускается в выражении инициализатора объекта для одномерных коллекций. Например, теперь можно инициализировать массив с одним измерением с помощью инициализатора объектов, как показано в следующем коде:

C#

```
public class TimerRemaining
{
    public int[] buffer { get; set; } = new int[10];
}

var countdown = new TimerRemaining()
{
    buffer =
    {
```

```
        [^1] = 0,
        [^2] = 1,
        [^3] = 2,
        [^4] = 3,
        [^5] = 4,
        [^6] = 5,
        [^7] = 6,
        [^8] = 7,
        [^9] = 8,
        [^10] = 9
    }
};
```

Класс `TimerRemaining` включает массив `buffer`, инициализированный до длины 10. В приведенном выше примере значения присваиваются этому массиву с помощью оператора индекса "с конца" (`^`), что в результате создаёт массив, который отсчитывается от 9 до 0.

В версиях до C# 13 оператор `^` нельзя использовать в инициализаторе объектов. Необходимо индексировать элементы с начала.

`ref` и `unsafe` в итераторах и `async` методах

Эта функция и следующие две функции позволяют `ref struct` типам использовать новые конструкции. Вы не будете использовать эти функции, если не создадите собственные `ref struct` типы. Скорее всего, вы видите косвенное преимущество по мере того как System.Span<T> и System.ReadOnlySpan<T> приобретают больше функционала.

До C# 13 методы итератора (методы, использующие `yield return`) и методы `async` не могли объявлять локальные `ref` переменные, а также не могли бы иметь контекст `unsafe`.

В C# 13 методы `async` могут объявлять `ref` локальные переменные или локальные переменные типа `ref struct`. Однако эти переменные нельзя получить через `await` границу. Они не могут быть доступны через границу `yield return`.

Это ослабленное ограничение позволяет компилятору разрешить безопасное использование `ref` локальных переменных и `ref struct` типов в большем числе мест. В этих методах можно безопасно использовать такие типы, как System.ReadOnlySpan<T>. Компилятор сообщает вам, если вы нарушаете правила безопасности.

Таким же образом C# 13 позволяет использовать контексты `unsafe` в методах итераторов. Однако все операторы `yield return` и `yield break` должны находиться в безопасных контекстах.

`allows ref struct`

До C# 13 `ref struct` типы не могут быть объявлены в качестве аргумента типа для универсального типа или метода. Теперь объявления универсальных типов могут добавлять анти-ограничение, `allows ref struct`. Это анти-ограничение объявляет, что аргумент типа, предоставленный для этого параметра типа, может быть типом `ref struct`. Компилятор применяет правила ссылочной безопасности ко всем экземплярам этого параметра типа.

Например, можно объявить универсальный тип, например следующий код:

C#

```
public class C<T> where T : allows ref struct
{
    // Use T as a ref struct:
    public void M(scoped T p)
    {
        // The parameter p must follow ref safety rules
    }
}
```

Это позволяет использовать такие типы, как System.Span<T> и System.ReadOnlySpan<T> с универсальными алгоритмами, где это применимо. Вы можете узнать больше в обновлениях для where и в статье руководства по программированию об универсальных ограничениях .

`ref struct` интерфейсы

До C# 13 `ref struct` типы не были разрешены для реализации интерфейсов. Начиная с C# 13, они могут. Можно объявить, что тип `ref struct` реализует интерфейс. Однако для обеспечения соответствия правилам безопасности не удается преобразовать тип `ref struct` в тип интерфейса. Это преобразование является преобразованием бокса и может нарушить безопасность ссылок. Явные объявления методов интерфейса в `ref struct` можно получить только через параметр типа, где этот параметр типа allows ref struct. Кроме того, `ref struct` типы должны реализовать все методы, объявленные в интерфейсе, включая те, которые имеют реализацию по умолчанию.

Узнайте больше об обновлениях для типов `ref struct`.

Частично задействованные члены

Вы можете объявить свойства `partial` и индексаторы `partial` в C# 13. Частичные свойства и индексаторы обычно следуют тем же правилам, что и методы `partial`: создается один *объявление объявления* и один *реализации объявления*. Подписи двух объявлений должны совпадать. Одно из ограничений заключается в том, что нельзя использовать объявление автоматического свойства для реализации частичного свойства . Свойства, которые не объявляют тело, считаются *декларативным объявлением*.

C#

```
public partial class C
{
    // Declaring declaration
    public partial string Name { get; set; }
}

public partial class C
{
    // implementation declaration:
    private string _name;
    public partial string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

Дополнительные сведения см. в статье о частичных членах .

Приоритет разрешения перегрузки

В C# 13 компилятор распознает OverloadResolutionPriorityAttribute, чтобы отдавать предпочтение одной перегрузке над другой. Авторы библиотеки могут использовать этот атрибут, чтобы гарантировать, что новая, более эффективная перегрузка предпочтительна по сравнению с существующей перегрузкой. Например, можно добавить новую перегрузку, которая более эффективна. Вы не хотите прерывать существующий код, использующий библиотеку, но вы хотите, чтобы пользователи обновляли новую версию при повторной компиляции. Приоритет разрешения перегрузки можно использовать для информирования компилятора о том, какая перегрузка должна быть предпочтительна. Перегрузки с наивысшим приоритетом предпочтительнее.

Эта функция предназначена для авторов библиотек, чтобы избежать неоднозначности при добавлении новых перегрузок. Авторы библиотеки должны проявлять осторожность с этим атрибутом, чтобы избежать путаницы.

Ключевое слово `field`

Контекстное ключевое слово field представлено в C# 13 в качестве предварительной функции. Маркер `field` обращается к синтезированному полю резервного копирования компилятора в методе доступа к свойствам. Он позволяет создавать текст метода доступа без объявления явного поля резервного копирования в объявлении типа. Вы можете определить тело для одного или обоих акцессоров свойства, связанного с полем.

Функция `field` выпущена как предварительная версия функции. Мы хотим узнать о своем опыте использования. Существует потенциальное изменение, приводящее к сбоям, или путаница при чтении кода в типах, которые также включают поле с именем `field`. Вы можете использовать `@field` или `this.field` для диамбигуации между ключевым словом `field` и идентификатором.

> **ⓘ Важный**
>
> Следует тщательно использовать функцию ключевого слова `field` в классе с полем с именем `field`. Новое ключевое слово `field` затмевает поле с именем `field` в аксессоре свойства. Можно изменить имя переменной `field` или использовать маркер `@` для ссылки на идентификатор `field` как `@field`. Вы можете узнать больше, прочитав спецификацию характеристики для ключевого слова **field**.

Если вы попробуете эту функцию и у вас будет отзыв, добавьте его в проблему в репозитории `csharplang`.

См. также

- Новые возможности .NET 9

Last updated on 18.11.2025

Новые возможности C# 12

Статья • 29.01.2025

C# 12 включает следующие новые функции. Эти функции можно попробовать с помощью последней версии Visual Studio 2022 или пакета SDK для .NET 8.

- Первичные конструкторы - Представлено в Visual Studio 2022 версии 17.6 (предварительный выпуск 2).
- Коллекционные выражения, представленные в Visual Studio 2022 версии 17.7 (предварительная сборка 5).
- Встроенные массивы - Представлено в Visual Studio 2022 версии 17.7 предварительная версия 3.
- необязательные параметры в лямбда-выражениях . Представлено в Visual Studio 2022 версии 17.5 (предварительная версия 2).
- ref readonly параметры . Представлено в Visual Studio 2022 версии 17.8 (предварительная версия 2).
- Псевдоним любого типа . Представлен в Visual Studio 2022 версии 17.6 (предварительная версия 3).
- экспериментальный атрибут — представлен в Visual Studio 2022 версии 17.7(предварительная версия 3).
- перехватчики - *предварительная функция* была введена в Visual Studio 2022 версии 17.7 предварительная версия 3.

C# 12 поддерживается в **.NET 8**. Дополнительные сведения см. в разделе версии языка C# .

Вы можете скачать последний пакет SDK для .NET 8 на странице загрузки .NET . Вы также можете скачать Visual Studio 2022 , которая включает пакет SDK для .NET 8.

> **Примечание**
>
> Мы заинтересованы в ваших отзывах об этих функциях. Если вы найдете проблемы с любым из этих новых функций, создайте **новую проблему** в репозитории **dotnet/roslyn** .

Основные конструкторы

Теперь вы можете создавать основные конструкторы в любом `class` и `struct`. Основные конструкторы больше не ограничиваются типами `record`. Основные параметры конструктора доступны в пределах всего тела класса. Чтобы убедиться, что все параметры первичного конструктора определенно назначены, все явно объявленные конструкторы должны вызывать основной конструктор с помощью синтаксиса `this()`. Добавление основного конструктора в `class` запрещает компилятору объявлять неявный конструктор без параметров. В `struct` неявный конструктор без параметров инициализирует все поля, включая основные параметры конструктора в 0-разрядный шаблон.

Компилятор создает общедоступные свойства для параметров первичного конструктора только в типах `record`, `record class` или `record struct`. Классы и структуры, не являющиеся записями, могут не всегда хотеть такое поведение для параметров первичного конструктора.

Дополнительные сведения об основных конструкторах см. в руководстве по изучению первичных конструкторов и в статье о конструкторах экземпляров .

Выражения коллекции

Выражения коллекции вводят новый лаконичный синтаксис для создания общих значений коллекций. Встраивание других коллекций в эти значения возможно с помощью оператора расширения `..e`.

Можно создать несколько типов, похожих на коллекции, без поддержки внешней библиотеки базовых классов (BCL). Ниже приведены следующие типы:

- Типы массивов, например `int[]`.
- System.Span<T> и System.ReadOnlySpan<T>.
- Типы, поддерживающие инициализаторы коллекций, например System.Collections.Generic.List<T>.

В следующих примерах показано использование коллекционных выражений.

C#

```
// Create an array:
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];

// Create a list:
List<string> b = ["one", "two", "three"];
```

```
// Create a span
Span<char> c  = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];

// Create a jagged 2D array:
int[][] twoD = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

// Create a jagged 2D array from variables:
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[][] twoDFromVariables = [row0, row1, row2];
```

Элемент распространения , `..e` в выражении коллекции добавляет все элементы в этом выражении. Аргумент должен быть типом коллекции. В следующих примерах показано, как работает элемент spread:

C#

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];
foreach (var element in single)
{
    Console.Write($"{element}, ");
}
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Элемент spread вычисляет каждый элемент выражения перечислений. Каждый элемент включается в выходную коллекцию.

Выражения коллекции можно использовать в любом месте, где требуется коллекция элементов. Они могут указать начальное значение для коллекции или передаваться в качестве аргументов методам, которые принимают типы коллекций. Дополнительные сведения о выражениях коллекции см. в статье справочника по языку для выражений коллекции или спецификации компонентов .

параметры `ref readonly`

C# добавил `in` параметры как способ передачи ссылок только для чтения. `in` параметры разрешают как переменные, так и значения, и могут использоваться без примечаний к аргументам.

Добавление параметров `ref readonly` обеспечивает более четкость для API, которые могут использовать параметры `ref` или параметры `in`:

- API, созданные до появления `in`, могут использовать `ref`, даже если аргумент не изменен. Эти API можно обновить с помощью `ref readonly`. Это не будет критическим изменением для вызывающих абонентов, как было бы, если параметр `ref` был изменен на `in`. Примером является System.Runtime.InteropServices.Marshal.QueryInterface.
- API, которые принимают параметр `in`, но логически требуют переменной. Выражение значения не работает. Примером является System.ReadOnlySpan<T>.ReadOnlySpan<T>(T).
- API, использующие `ref`, так как им требуется переменная, но не изменяйте эти переменные. Примером является System.Runtime.CompilerServices.Unsafe.IsNullRef.

Дополнительные сведения о параметрах `ref readonly` см. в статье о модификаторах параметров в справочнике языка, или в спецификации функции «только для чтения» параметров .

Параметры лямбда-кода по умолчанию

Теперь можно определить значения по умолчанию для параметров в лямбда-выражениях. Синтаксис и правила совпадают с добавлением значений по умолчанию для аргументов в любой метод или локальную функцию.

Дополнительные сведения о параметрах по умолчанию для лямбда-выражений см. в статье о лямбда-выражениях.

Псевдоним любого типа

Директиву псевдонима `using` можно использовать для псевдонимов любого типа, а не только именованных типов. Это означает, что можно создавать семантические псевдонимы для типов кортежей, типов массивов, типов указателей или других небезопасных типов. Дополнительную информацию см. в спецификации функции . Пример пошагового руководства по рефакторингу см. в статье Рефакторинг кода с помощью псевдонима любого типа в блоге .NET.

Встроенные массивы

Встроенные массивы используются командой среды выполнения и другими авторами библиотек для повышения производительности в ваших приложениях. Встроенные массивы позволяют разработчику создавать массив фиксированного размера в типе `struct`. Структура с встраиваемым буфером должна обеспечивать характеристики производительности, аналогичные небезопасному буферу фиксированного размера. Скорее всего, вы вряд ли будете объявлять собственные встроенные массивы, но будете использовать их прозрачно, когда они предоставляются в качестве объектов System.Span<T> или System.ReadOnlySpan<T>, предоставляемых API среды выполнения.

встроенный массив объявляется подобным образом `struct`:

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct Buffer
{
    private int _element0;
}
```

Они используются как любой другой массив:

C#

```
var buffer = new Buffer();
for (int i = 0; i < 10; i++)
{
    buffer[i] = i;
}

foreach (var i in buffer)
{
    Console.WriteLine(i);
}
```

Разница заключается в том, что компилятор может воспользоваться известными сведениями о встроенном массиве. Скорее всего, вы используете встроенные массивы так же, как и обычные массивы. Дополнительные сведения об объявлении встроенных массивов см. в справочнике по языку struct типов.

Экспериментальный атрибут

Типы, методы или сборки можно пометить System.Diagnostics.CodeAnalysis.ExperimentalAttribute, чтобы указать экспериментальную функцию. Компилятор выдает предупреждение при доступе к

методу или типу, аннотированному с помощью ExperimentalAttribute. Все типы, включенные в сборку, помеченную атрибутом `Experimental`, являются экспериментальными. Дополнительные сведения см. в статье об общих атрибутах, прочитанных компилятором, или в спецификации функций.

Перехватчики

> **⚠ Предупреждение**
>
> Перехватчики — это экспериментальная функция, доступная в режиме предварительной версии с C# 12. Эта функция может подвергаться переломным изменениям или удалению в будущей версии. Поэтому не рекомендуется для производственной среды или выпущенных приложений.
>
> Чтобы использовать перехватчики, проект пользователя должен указать свойство `<InterceptorsPreviewNamespaces>`. Это список пространств имен, которые могут содержать перехватчики.
>
> Например,
> `<InterceptorsPreviewNamespaces>$(InterceptorsPreviewNamespaces);Microsoft.AspNetCore.Http.Generated;MyLibrary.Generated</InterceptorsPreviewNamespaces>`

перехватчика — это метод, который может декларативно заменить вызов метода *перехватываемого* вызовом во время компиляции. Эта подстановка происходит путем объявления перехватчиком исходных расположений вызовов, которые он перехватывает. Перехватчики предоставляют ограниченное средство для изменения семантики существующего кода путем добавления нового кода в компиляцию, например в генератор источника.

Вы используете *перехватчик* в составе генератора исходного кода для изменения, а не добавления кода в существующую компиляцию исходного кода. Генератор исходного кода заменяет вызовы перехватываемого метода вызовом метода перехватчика .

Если вы заинтересованы в эксперименте с перехватчиками, вы можете узнать больше, прочитав спецификацию функции . Если вы используете эту функцию, не забудьте оставаться в курсе всех изменений в спецификации компонентов для этой экспериментальной функции. Если эта функция завершена, мы добавим дополнительные рекомендации на этом сайте.

См. также

- Новые возможности .NET 8

Узнайте о критических изменениях в компиляторе C#

22.05.2025

Чтобы найти критические изменения после выпуска C# 10, см. критические изменения в Roslyn после .NET 6.0.100 до .NET 7.0.100.

Команда Roslyn поддерживает список критических изменений в компиляторах C# и Visual Basic. Эти изменения можно найти по этим ссылкам в репозитории GitHub:

- Критические изменения в Roslyn в C# 10.0/.NET 6
- Критические изменения в Roslyn после .NET 5
- Критические изменения в версии 16.8 VS2019, введенные с появлением .NET 5 и C# 9.0
- Критические изменения в VS2019 с обновлением 1 и выше по сравнению с VS2019
- Критические изменения с версии VS2017 (C# 7)
- Критические изменения в Roslyn 3.0 (VS2019) из Roslyn 2.* (VS2017)
- Критические изменения в Roslyn 2.0 (VS2017) по сравнению с Roslyn 1.* (VS2015) и собственным компилятором C# (VS2013 и предыдущих).
- Изменения, нарушающие совместимость в Roslyn 1.0 (VS2015) по сравнению с исходным компилятором C# (VS2013 и предыдущие).
- Изменение версии Юникода в C# 6

История языка C#

В этой статье содержится описание основных выпусков языка C#. Команда разработчиков C# продолжает добавлять новые функции. Сведения о состоянии функции для отдельных языков, включая функции, которые будут добавлены в предстоящие выпуски, см. в репозитории dotnet/roslyn на сайте GitHub. Чтобы узнать, когда определенная функция была добавлена на язык, обратитесь к файлу журнала версий C# в репозитории dotnet/csharplang на сайте GitHub.

> **Важный**
>
> В некоторых возможностях используются типы и методы, которые в спецификации C# определены как *стандартная библиотека*. Платформа .NET предоставляет эти типы и методы в разных пакетах. например, обработка исключений. Каждая инструкция и выражение `throw` проверяется, чтобы убедиться, что вызываемый объект является производным от **Exception**. Аналогичным образом каждая инструкция `catch` проверяется, чтобы убедиться, что перехваченный тип является производным от **Exception**. В каждой версии могут добавляться новые требования. Чтобы использовать новейшие возможности языка в старой среде, может потребоваться установить определенные библиотеки. Эти зависимости описаны на странице для каждой конкретной версии. Дополнительные сведения о связи между языком и библиотекой, а также общие сведения о такой зависимости см. **здесь**.

C# версии 14

Выпущено в ноябре 2025 г.

C# 14 включает следующие новые возможности:

- Члены расширения
- Условное присваивание null
- nameof поддерживает несвязанные универсальные типы
- Более неявные преобразования для Span<T> и ReadOnlySpan<T>
- Модификаторы простых лямбда-параметров
- field поддерживаемые свойства
- partial события и конструкторы
- определяемые пользователем операторы составных назначений

C# версии 13

выпущено в ноябре 2024 г.

C# 13 включает следующие новые возможности:

- `params` коллекции: модификатор `params` не ограничивается типами массивов. Теперь можно использовать `params` с любым распознаваемым типом коллекции, включая `Span<T>` и типы интерфейсов.
- Новый тип и семантика `lock`: если целевой объект инструкции `lock` является System.Threading.Lock, компилятор создает код для использования метода Lock.EnterScope() для ввода эксклюзивной области. `ref struct`, полученный из этого, поддерживает шаблон `Dispose()` для выхода из эксклюзивной области.
- Новая последовательность экранирования — `\e`: вы можете использовать `\e` в качестве последовательности экранирования литерала символа для символа `ESCAPE`, Unicode `U+001B`.
- Небольшие оптимизации для разрешения перегрузки с участием групп методов.
- Неявный доступ индексатора в инициализаторах объектов: в выражении инициализатора объекта теперь разрешен неявный оператор индекса "с конца", `^`.
- Вы можете использовать `ref` локальные и `unsafe` контексты в итераторах и асинхронных методах.
- Для реализации интерфейсов можно использовать `ref struct` типы.
- Вы можете разрешить типы `ref struct` в качестве аргументов для параметров типа в обобщениях.
- Частичные свойства и индексаторы теперь разрешены в `partial` типах.
- Приоритет разрешения перегрузки позволяет авторам библиотек назначать одну перегрузку в качестве лучшей по сравнению с другими.

C# версии 12

Выпущено в ноябре 2023 г.

В C# 12 добавлены следующие функции:

- Первичные конструкторы — вы можете создавать первичные конструкторы в любом `class` или `struct` типе.
- Выражения коллекции — новый синтаксис для указания выражений коллекции, включая элемент spread, (`..e`), чтобы развернуть любую коллекцию.
- Встроенные массивы — встроенные массивы позволяют создавать массив фиксированного размера в типе `struct` .
- Необязательные параметры в лямбда-выражениях — можно определить значения по умолчанию для параметров в лямбда-выражениях.

- ref readonly Параметры - `ref readonly` параметров обеспечивают более четкость для API, которые могут использовать `ref` параметры или `in` параметры.
- Псевдоним любого типа . Директива `using` псевдонима можно использовать для псевдонима любого типа, а не только именованных типов.
- Экспериментальный атрибут — указывает экспериментальную функцию.

И, перехватчики - выпущены в качестве функции предварительной *версии*.

В целом C# 12 предоставляет новые функции, которые упрощают написание кода C#. Синтаксис, который вы уже знали, доступен в других местах. Другой синтаксис обеспечивает согласованность связанных понятий.

C# версии 11

Выпущено в ноябре 2022 г.

В C# 11 добавлены следующие функции:

- Необработанные строковые литералы.
- Поддержка универсальной математики.
- Универсальные атрибуты.
- Строковые литералы UTF-8.
- Новые строки в выражениях интерполяции строк.
- Шаблоны списка.
- Локальные типы файлов.
- Обязательные участники.
- Автоматические структуры по умолчанию.
- Сопоставление шаблона `Span<char>` на константу `string`.
- Расширенная `nameof` область.
- `nint` и `uint` обозначают System.IntPtr и System.UIntPtrSystem.UIntPtr, соответственно.
- ref поля и scoped ref
- Улучшено преобразование группы методов в делегат.
- Предупреждающие волны 7

C# 11 содержит *универсальные математические функции* и несколько функций, поддерживающих эту цель. Числовые алгоритмы можно записывать один раз для всех типов чисел. Существует больше возможностей для упрощения работы с `struct` типами, такими как обязательные элементы и структуры по умолчанию. Работа со строками упрощается с необработанными строковыми литералами, новой строкой в интерполяциях строк и строковых литералах UTF-8. Такие функции, как локальные типы файлов, позволяют упростить генераторы источников. Наконец, шаблоны списков добавляют дополнительную поддержку сопоставления шаблонов.

C# версии 10

Выпущено в ноябре 2021 г.

В C# 10 добавлены следующие функции и улучшения языка C#:

- Структуры записей
- Улучшения типов структуры
- Обработчики интерполированных строк
- Директивы global using
- Объявление пространства имен в пределах файла
- Расширенные шаблоны свойств
- Лямбда-выражения могут иметь естественно определяемый тип, где компилятор может вывести тип делегата из лямбда-выражения или группы методов.
- Лямбда-выражения могут объявлять тип возвращаемого значения , когда компилятор не может вывести его.
- атрибуты можно применять к лямбда-выражениям.
- В C# 10 `const` строки можно инициализировать с помощью интерполяции строк, если все заполнители являются константными строками.
- В C# 10 вы можете добавить модификатор `sealed` при переопределении `ToString` в типе записи .
- Предупреждения для определенного назначения и анализа состояния NULL более точны.
- Разрешить присваивание и объявление в одной деконструкции.
- Разрешен атрибут AsyncMethodBuilder в методах
- Атрибут CallerArgumentExpression
- C# 10 поддерживает новый формат для `#line` pragma.

C# 10 продолжает работу над темами удаления церемонии, разделения данных от алгоритмов и повышения производительности среды выполнения .NET.

Многие функции означают, что вы вводите меньше кода, чтобы выразить те же понятия. *Структуры записей* синтезируют многие из одинаковых методов, которые *записывают классы* . Структуры и анонимные типы поддерживаются *выражениями*. *Глобальные директивыи объявления* пространства имен в области файлов означают, что вы выражаете зависимости и организацию пространства имен более четко. *Лямбда-улучшения* упрощают объявление лямбда-выражений, где они используются. Новые шаблоны свойств и улучшения деконструкции создают более краткий код.

Новые интерполированные обработчики строк и `AsyncMethodBuilder` поведение могут повысить производительность. Эти языковые функции были применены в среде выполнения .NET для повышения производительности в .NET 6.

C# 10 также отмечает большую часть смены ежегодной каденции для выпусков .NET. Так как не каждая функция может быть завершена в течение ежегодного периода времени, вы можете попробовать несколько функций предварительной версии в C# 10. Можно использовать как универсальные атрибуты, так *и статические абстрактные члены в интерфейсах*, но эти предварительные версии могут измениться до их окончательного выпуска.

C# версии 9

Выпущено в ноябре 2020 г.

C# версии 9 был выпущен с .NET 5. Это версия языка используется по умолчанию для любой сборки, предназначенной для выпуска .NET 5. Она содержит следующие новые и улучшенные функции:

- Записи
- Инициализация только наборов
- Операторы верхнего уровня
- Улучшения сопоставления шаблонов: реляционные шаблоны и логические шаблоны
- Производительность и взаимодействие
  - Целые числа собственного размера
  - Указатели функций
  - Отключение флага localsinit
  - Инициализаторы модулей
  - Новые функции для частичных методов
- Функции соответствия и отделки
  - Выражения new с целевым типом
  - Анонимные функции static
  - Целевые условные выражения с типом
  - Типы возвращаемых ковариантных значений
  - Поддержка расширения GetEnumerator для циклов foreach
  - Параметры лямбда-отмены
  - Атрибуты для локальных функций

В C# 9 получили развитие три направления предыдущих выпусков: устранение формальностей, отделение данных от алгоритмов и доступность большего количества шаблонов в большем количестве мест.

Операторы верхнего уровня упрощают чтение основной программы. Сократилась потребность в формальностях: больше не нужны ни пространства имен, `Program`, ни классы, ни `static void Main()`.

Введение records содержит краткий синтаксис для ссылочных типов, которые соответствуют семантике значений для равенства. Эти типы используются для определения контейнеров данных, которые обычно определяют минимальное поведение. Методы задания только для инициализации предоставляют возможность недеструктивной мутации (`with` выражений) в записях. В C# 9 также добавлены ковариантные возвращаемые типы, благодаря чему производные записи могут переопределять виртуальные методы и возвращать тип, производный от возвращаемого типа базового метода.

Возможности сопоставления шаблонов расширяются несколькими способами. Числовые типы теперь поддерживают *шаблоны диапазонов*. Шаблоны можно комбинировать с помощью шаблонов `and`, `or` и `not`. Скобки можно добавить для уточнения более сложных шаблонов:

C# 9 включает новые улучшения сопоставления шаблонов.

- ***Шаблоны*** типов соответствуют объекту определенного типа
- ***Шаблоны в круглых скобках*** усиливают или подчеркивают приоритет сочетаний шаблонов.
- ***В шаблонах конъюнкций*** `and` требуется соответствие обоих шаблонов.
- ***В шаблонах дизъюнкций*** `or` требуется соответствие хотя бы одного из шаблонов.
- В ***шаблонах*** `not` ***с отрицанием*** требуется несоответствие данного шаблона.
- ***Реляционные шаблоны*** требуют, чтобы входные данные были меньше, чем, меньше или равно, или больше или равно заданной константе.

Эти шаблоны обогащают синтаксис шаблонов. Одним из наиболее распространенных применений нового синтаксиса является проверка значения на null.

C#

```
if (e is not null)
{
    // ...
}
```

Любой из этих шаблонов можно использовать в любом контексте, где разрешены шаблоны: выражения с шаблоном `is`, выражения `switch`, вложенные шаблоны и шаблоны метки `switch` оператора `case`.

Высокопроизводительные вычисления на C# поддерживает еще один набор функций.

- Типы `nint` и `nuint` моделируют целочисленные типы собственного размера в целевом ЦП.

- Указатели функций предоставляют функциональные возможности, подобные возможностям делегатов, но при этом не требуют выделения, необходимого для создания объекта-делегата.
- Инструкцию `localsinit` можно опустить для сохранения инструкций.

Производительность и взаимодействие

Еще один набор улучшений поддерживает сценарии, в которых *генераторы кода* добавляют функциональные возможности.

- Инициализаторы модулей — это методы, которые среда выполнения вызывает при загрузке сборки.
- Разделяемые методы поддерживают новые модификаторы доступа и возвращаемые типы, отличные от void. В таких случаях необходимо предоставить реализацию.

Функции подбора и завершения

В C# 9 добавлено множество других небольших функций, улучшающих производительность разработчиков, а также написание и чтение кода:

- Выражения `new` целевого типа
- Анонимные функции `static`
- Условные выражения целевого типа
- Поддержка расширения `GetEnumerator()` для циклов `foreach`
- Лямбда-выражения могут объявлять параметры отмены
- Атрибуты могут применяться к локальным функциям

Выпуск C# 9 — это следующий этап развития, благодаря которому C# остается современным языком программирования общего назначения. Его возможности обеспечат дальнейшую поддержку современных рабочих нагрузок и типов приложений.

C# версии 8.0

Выпущено в сентябре 2019 г.

C# 8.0 — это первый основной выпуск C#, специально предназначенный для .NET Core. Некоторые функции используют новые возможности среды CLR, другие — типы библиотек, добавленные только в .NET Core. В C# 8.0 добавлены следующие функции и улучшения языка C#:

- Члены только для чтения
- Элементы интерфейса по умолчанию

- Улучшения сопоставления шаблонов:
  - Выражения switch
  - Шаблоны свойств
  - Шаблоны кортежей
  - Позиционные шаблоны
- Объявления using.
- Статические локальные функции.
- Удаляемые ссылочные структуры.
- Ссылочные типы, допускающие значение NULL
- Асинхронные потоки.
- Индексы и диапазоны.
- Присваивание объединения со значением NULL
- Неуправляемые сконструированные типы
- Выражение stackalloc во вложенных выражениях
- Улучшение интерполированных строк verbatim

Для членов интерфейса по умолчанию требуются улучшения среды CLR. Эти функции были добавлены в среду CLR для .NET Core 3.0. Диапазоны, индексы и асинхронные потоки требуют новых типов в библиотеках .NET Core 3.0. Ссылочные типы, допускающие значение NULL, реализованные в компиляторе, гораздо удобнее использовать, когда библиотеки снабжаются заметками для предоставления семантической информации о состоянии NULL аргументов и возвращаемых значений. Эти заметки сейчас добавляются в библиотеки .NET Core.

C# версии 7.3

Выпущено в мае 2018 г.

Новые возможности в выпуске C# 7.3 можно разделить на две основные группы. Одна из них — набор функций для повышения эффективности безопасного кода до уровня небезопасного кода. Вторая — постепенные улучшения существующих функций. В этом выпуске добавлены новые параметры компилятора.

В ту группу, которая отвечает за повышение производительности безопасного кода, входят следующие новые возможности:

- доступ к полям фиксированной ширины без закрепления;
- возможность переназначать локальные переменные `ref`;
- возможность использовать инициализаторы для массивов `stackalloc`;
- возможность использовать инструкции `fixed` с любым типом, который поддерживает шаблон;

- Можно использовать более универсальные ограничения.

Для существующих функций предоставлены следующие улучшения:

- возможность проверить `==` и `!=` с типами кортежа;
- больше мест для использования переменных выражений;
- Атрибуты можно прикрепить к резервному полю автоматически реализованных свойств.
- Разрешение методов при улучшении `in` аргументов.
- стало меньше неоднозначных вариантов при разрешении перегрузок.

Новые параметры компилятора:

- -publicsign позволяет включить подписывание сборок как программного обеспечения с открытым кодом;
- -pathmap позволяет предоставить сопоставление для исходных каталогов.

C# версии 7.2

Выпущено в ноябре 2017 г.

В C# 7.2 добавлено несколько небольших языковых функций:

- возможность использовать инициализаторы для массивов `stackalloc`;
- возможность использовать инструкции `fixed` с любым типом, который поддерживает шаблон;
- доступ к полям фиксированной ширины без закрепления;
- возможность переназначать локальные переменные `ref`;
- объявление типов `readonly struct`, указывающее, что структура является неизменяемой и должна передаваться в методы члена как параметр `in`;
- добавление модификатора `in` для параметров, указывающего, что аргумент передается по ссылке, но не изменяется вызываемым методом;
- использование модификатора `ref readonly` для возвращаемого значения метода, указывающего, что метод возвращает значение по ссылке, но не допускает операции записи в соответствующий объект;
- объявление типов `ref struct`, указывающее, что тип структуры обращается напрямую к управляемой памяти и всегда должен обрабатываться с выделением стека;
- Используйте более универсальные ограничения.
- Неконечные именованные аргументы:
  - Позиционные аргументы могут следовать именованным аргументам.
- Ведущие символы подчеркивания в числовых литералах:

    - Перед любыми печатными знаками в числовых литералах теперь могут использоваться начальные знаки подчеркивания.
- private protected модификатор доступа:
    - Модификатор доступа `private protected` разрешает доступ для производных классов в одной сборке.
- Условные `ref` выражения:
    - Результат условного выражения `?:` теперь может быть ссылкой.

C# версии 7.1

Выпущено за август 2017 г.

C# начали выпускать *доработанные выпуски* с C# 7.1. В этой версии добавлен элемент конфигурации выбора версии языка, три новых возможности языка и новое поведение компилятора.

Новые языковые функции в этом выпуске

- метод asyncMain.
    - Точка входа для приложения может иметь модификатор `async`.
- Литеральные выражения default
    - Литеральные выражения по умолчанию можно использовать в выражениях значения по умолчанию, если можно вывести тип целевого объекта.
- Выводимые имена элементов кортежа
    - Имена элементов кортежа часто можно вывести из инициализации кортежа.
- Сопоставление шаблонов для параметров универсального типа
    - Выражения сопоставления шаблонов можно использовать с переменными, тип которых является параметром универсального типа.

Наконец, компилятор имеет два варианта -refout и -refonly создание эталонной сборки.

C# версии 7.0

Выпущено за март 2017 г.

C# версии 7.0 был выпущен вместе с Visual Studio 2017. Эта версия имеет некоторые эволюционные и холодные вещи в вену C# 6.0. Ниже приведены некоторые из новых функций:

- Переменные Out
- Кортежи и деконструкция
- Сопоставление шаблонов

- Локальные функции
- Расширенные элементы, воплощающие выражение
- Ссылка на локальные параметры
- Возврат ссылок

Другие возможности:

- Пустые переменные
- Двоичные литералы и цифровые разделители
- Выражения throw

Все эти функции предлагают новые возможности для разработчиков и возможность писать более чистый код, чем когда-либо. Важным аспектом является доработка объявления переменных путем использования ключевого слова `out` и поддержки нескольких возвращаемых значений благодаря кортежу. Теперь .NET Core подходит для любой операционной системы, а также уверенно движется в направлении облачных технологий и портативности. Очевидно, что разработчики активно занимаются этими новыми возможностями наравне с добавлением новых функций.

C# версии 6.0

Выпущено в июле 2015 г.

Версия 6.0, выпущенная в Visual Studio 2015, выпустила множество небольших функций, что сделало программирование C# более продуктивным. Вот некоторые из них:

- Статические импорты
- Фильтры исключений
- Инициализаторы автосвойств
- Элементы, воплощающие выражение
- Null-распространитель
- Интерполяция строк
- Оператор nameof

Другие новые возможности:

- Инициализаторы индекса
- Выражение Await в блоках Catch и Finally
- Значения по умолчанию для свойств метода получения

Если вы посмотрите на эти функции вместе, вы увидите интересный шаблон. В этой версии C# начал устранять шаблоны языка, чтобы сделать код более удобочитаемым и

доступным для чтения. Поэтому эта версия, несомненно, привлечет сторонников простого и понятного кода.

Еще одно новшество, связанное с этой версией, не является функцией языка в обычном понимании. Состоялся выпуск компилятора Roslyn в виде службы. Теперь компилятор C# написан на C#, и вы можете использовать его при программировании.

C# версии 5.0

Выпущено за август 2012 г.

Версия C# 5.0, выпущенная вместе с Visual Studio 2012, была крайне целенаправленной. Практически все нововведения в этой версии касались другой инновационной концепции: модели `async` и `await` для асинхронного программирования. Ниже приведен список основных функций:

- Асинхронные члены
- Информационные атрибуты вызывающего объекта

Информационный атрибут вызывающего объекта позволяет легко получать сведения о контексте, в котором выполняется работа, не прибегая к массивному стандартному коду отражения. Он находит обширное применение в задачах диагностики и ведения журнала.

Однако настоящими звездами этого выпуска являются `async` и `await`. После появления этих функций в 2012 г. C# удалось снова взять быка за рога, первоклассным образом внедрив асинхронность в язык.

C# версии 4.0

Выпущено за апрель 2010 г.

C# версии 4.0, выпущенной в Visual Studio 2010, представил некоторые интересные новые возможности:

- Динамическая привязка
- Именованные/дополнительные аргументы
- Универсальная ковариантность и контравариантность
- Внедренные типы взаимодействия

Внедренные типы взаимодействия облегчили создание сборок COM-взаимодействия для приложения при развертывании. Универсальная ковариантность и контравариантность расширяют возможности применения универсальных шаблонов, однако эти функции несколько академичны и высоко оценены авторами, создающими платформы и

библиотеки. Именованные и дополнительные параметры позволяют избавиться от многих перегрузок методов и сделать работу более удобной. Однако ни одна из этих функций не является кардинальным изменением.

Значимым событием стало появление ключевого слова `dynamic`. Благодаря ключевому слову `dynamic` в C# версии 4.0 появилась возможность переопределять компилятор при типизации во время компиляции. Используя это ключевое слово, вы можете создавать конструкции, характерные для языков с динамической типизацией, таких как JavaScript. Вы можете создать `dynamic x = "a string"` и добавить туда шестерку, отложив решение о дальнейших действиях во времени выполнения.

Динамическое связывание подвержено ошибкам, однако открывает великолепные возможности внутри языка.

C# версии 3.0

Выпущено в ноябре 2007 г.

C# версии 3.0 был выпущен в конце 2007 года вместе с Visual Studio 2008, однако существенно набор возможностей языка пополнится только в .NET Framework версии 3.5. Эта версия ознаменовала кардинальное изменение развития C#. Она доказала, что C# действительно является достойным языком программирования. Давайте рассмотрим некоторые основные возможности в этой версии:

- Автоматически реализуемые свойства
- Анонимные типы
- Выражения запросов
- Лямбда-выражения
- Деревья выражений
- Методы расширения
- Неявно типизированные локальные переменные
- Разделяемые методы
- Инициализаторы объектов и коллекций

Если оглянуться назад, становится ясно, что многие из этих возможностей не могли не появиться или появиться отдельно друг от друга. Все они образуют единый и стратегически значимый набор. Эта функция убийцы версии C# была выражением запроса, также известным как языковой интегрированный запрос (LINQ).

Немного углубившись, можно отметить деревья выражений, лямбда-выражения и анонимные типы как основу для создания LINQ. Однако в любом случае в C# 3.0 была

представлена революционная концепция. C# 3.0 начал создавать основы для превращения C# в гибридный объектно-ориентированный или функциональный язык.

Среди прочего, теперь вы можете создавать декларативные запросы в стиле SQL для выполнения операций с коллекциями. Вместо создания цикла `for` для вычисления среднего значения из списка целых чисел теперь вам достаточно использовать `list.Average()`. Сочетание выражений запросов и методов расширения сделало список целых чисел намного умнее.

C# версии 2.0

Выпущено в ноябре 2005 г.

Давайте рассмотрим некоторые основные возможности языка C# 2.0, выпущенного в 2005 году вместе с Visual Studio 2005:

- Универсальные шаблоны
- Разделяемые типы
- Анонимные методы
- Типы значений, допускающие значение NULL
- Итераторы
- Ковариантность и контравариантность

Другие возможности C# 2.0 расширяли существующие возможности:

- Отдельный доступ к методу получения и методу задания
- Преобразования групп методов (делегаты)
- Статические классы
- Выведение делегата

В то время как C# начался как универсальный язык объектно-ориентированного (OO), C# версии 2.0 изменился в спешке. Благодаря универсальным шаблонам вы получаете типы и методы, которые могут работать с произвольным типом, при этом обеспечивая безопасность типа. В результате, например, наличие List<T> позволяет использовать `List<string>` или `List<int>` и совершать типобезопасные операции с этими строками или целыми числами, выполняя итерацию по ним. Использование универсальных шаблонов лучше, чем создание типа, наследуемого `ListInt` от `ArrayList` `Object` каждой операции или приведения от нее.

В C# версии 2.0 появились итераторы. Если кратко, итераторы позволяют изучать все элементы в `List` (или других перечислимых типов) с помощью цикла `foreach`. Наличие

итераторов в качестве первоклассного компонента в языке значительно улучшило удобочитаемость языка и способность воспринимать код.

Версия для C# 1.2

Выпущено за апрель 2003 г.

Версия C# 1.2 входит в состав Visual Studio .NET 2003. Она содержит ряд небольших улучшений языка. Наиболее значительным изменением является то, что начиная с этой версии код, создаваемый в цикле `foreach`, вызывает Dispose в IEnumerator, если IEnumerator реализует IDisposable.

C# версии 1.0

Выпущено за январь 2002 г.

Если взглянуть на C# версии 1.0, выпущенный вместе с Visual Studio .NET 2002, станет очевидно его сходство с Java. В рамках своих указанных целей проектирования для ECMA она стремилась быть "простым, современным, объектно-ориентированным языком общего назначения". В то время, похоже, Java означает, что он достиг этих ранних целей проектирования.

Однако если взглянуть на C# 1.0 сейчас, вам может стать немного не по себе. В этой версии не было встроенных асинхронных возможностей и некоторых функций для универсальных шаблонов, к которым вы уже привыкли. Фактически, в ней вообще не было универсальных шаблонов. А LINQ? Пока недоступно. Для добавления этих возможностей потребовалось несколько лет.

По современным меркам C# версии 1.0 не предоставлял практически никаких возможностей. Вам пришлось бы писать довольно подробный код. Однако надо же было с чего-то начинать. C# версии 1.0 был достойной альтернативой Java на платформе Windows.

Основные возможности C# 1.0 включали следующие компоненты:

- Классы
- Структуры
- Интерфейсы
- События
- Свойства
- Делегаты
- Операторы и выражения

- Отчеты
- Атрибуты

Статья изначально опубликована в блоге NDepend, с разрешения Эрика Дитриха (Erik Dietrich) и Патрика Смачиа (Patrick Smacchia).

Last updated on 18.11.2025

Связи между функциями языка и типами библиотек

17.06.2025

Определение языка C# требует, чтобы стандартная библиотека имела определенные типы и определенные доступные члены у этих типов. Компилятор создает код, использующий эти необходимые типы и члены для многих различных языковых функций. По этой причине версии C# поддерживаются только для соответствующей версии .NET и более новой. Это гарантирует правильное поведение в процессе выполнения и доступность всех необходимых типов и членов.

Эта зависимость от функций стандартной библиотеки была частью языка C# с первой версии. В этой версии включены примеры:

- Exception — используется для всех исключений, созданных компилятором.
- String - синоним `string`.
- Int32 - синоним `int`.

Эта первая версия была простой: компилятор и стандартная библиотека, поставляемые вместе, и была только одна версия каждой.

В последующих версиях C# иногда добавлены новые типы или члены в зависимости. Примеры: INotifyCompletion, CallerFilePathAttribute и CallerMemberNameAttribute. C# 7.0 добавил зависимость на ValueTuple для реализации языковой функции кортежей. Для C# 8 требуются System.IndexSystem.Rangeдиапазоны и индексы, среди прочего. Каждая новая версия может добавить дополнительные требования.

Группа разработки языка работает для минимизации области поверхности типов и членов, необходимых в стандартной библиотеке. Эта цель сбалансирована с чистой структурой, в которой новые функции библиотеки легко включаются в язык. В будущих версиях C# появятся новые функции, требующие новых типов и членов в стандартной библиотеке. Средства компилятора C# теперь отделены от цикла выпуска библиотек .NET на поддерживаемых платформах.

Соображения относительно версии и обновления для разработчиков на C#

Статья • 02.03.2024

Совместимость является важной целью, так как новые функции добавляются на язык C#. В большинстве случаев существующий код сможет компилироваться в новой версии компилятора безо всяких проблем. Команда среды выполнения .NET также имеет цель обеспечить совместимость обновленных библиотек. В почти всех случаях, когда приложение запускается из обновленной среды выполнения с обновленными библиотеками, поведение точно совпадает с предыдущими версиями.

Языковая версия, используемая для компиляции приложения, обычно соответствует моникеру целевой платформы среды выполнения (TFM), на которую ссылается ваш проект. Дополнительные сведения об изменении языковой версии по умолчанию см. в статье, описанной в статье о настройке версии языка. Это поведение по умолчанию обеспечивает максимальную совместимость.

При *появлении критических изменений* они классифицируются как:

- *Двоичное критическое изменение: двоичное критическое изменение* вызывает другое поведение, в том числе, возможно, сбой в приложении или библиотеке при запуске с помощью новой среды выполнения. Чтобы включить эти изменения, необходимо перекомпилировать приложение. Существующий двоичный файл не будет работать правильно.
- *Критическое изменение источника: критическое изменение* источника изменяет смысл исходного кода. Перед компиляцией приложения с последней версией языка необходимо внести изменения в исходный код. Существующий двоичный файл будет работать правильно с новым узлом и средой выполнения. Обратите внимание, что для синтаксиса языка критическое *изменение источника также* является изменением поведения, как определено в критических изменениях среды выполнения.

Если двоичное критическое изменение влияет на приложение, необходимо перекомпилировать приложение, но вам не нужно изменять исходный код. Когда критическое изменение источника влияет на приложение, существующий двоичный файл по-прежнему работает правильно в средах с обновленной средой выполнения и библиотеками. Однако необходимо внести изменения в источник для повторной компиляции с новой языковой версией и средой выполнения. Если

изменение является критическим и двоичным, необходимо перекомпилировать приложение с последней версией и внести обновления источника.

Из-за цели избежать критических изменений командой языка C# и командой среды выполнения обновление приложения обычно является вопросом обновления TFM и перестроения приложения. Однако для библиотек, распределенных публично, следует тщательно оценить политику для поддерживаемых TFM и поддерживаемых языковых версий. Например, если вы создаете новую библиотеку с функциями из последней версии, но ее должны использовать приложения, созданные в предыдущих версиях компилятора. Или вы можете обновить существующую библиотеку и многие пользователи, возможно, еще не обновили версии.

Введение критических изменений в библиотеки

При внедрении новых языковых функций в общедоступном API библиотеки следует оценить, вводит ли функция двоичное или исходное критическое изменение для пользователей библиотеки. Все изменения внутренней реализации, которые не отображаются в `public` интерфейсах или `protected` не совместимы.

> ⓘ **Примечание**
>
> Если вы используете System.Runtime.CompilerServices.InternalsVisibleToAttribute для включения типов для просмотра внутренних элементов, внутренние члены могут вносить критические изменения.

Двоичное критическое *изменение* требует, чтобы пользователи перекомпилировать свой код, чтобы использовать новую версию. Например, рассмотрим этот общедоступный метод:

```
C#

public double CalculateSquare(double value) => value * value;
```

Если добавить `in` модификатор в метод, это двоичное критическое изменение:

```
C#
```

```
public double CalculateSquare(in double value) => value * value;
```

Пользователи должны перекомпилировать любое приложение, использующее `CalculateSquare` метод для правильной работы новой библиотеки.

Критическое *изменение* источника требует, чтобы пользователи изменили свой код перед повторной компиляцией. Например, рассмотрим этот тип:

C#

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);

    // other details omitted
}
```

В более новой версии вы хотите воспользоваться преимуществами синтезированных элементов, созданных для `record` типов. Вы вносите следующее изменение:

C#

```
public record class Person(string FirstName, string LastName);
```

Для предыдущего изменения требуются изменения любого типа, производных от `Person`. Все эти объявления должны добавить `record` модификатор в их объявления.

Влияние критических изменений

При добавлении двоичного критического *изменения* в библиотеку вы принудительно принудительно перекомпилируйте все проекты, использующие библиотеку. Однако ни один исходный код в этих проектах не должен меняться. В результате влияние критического изменения достаточно мало для каждого проекта.

При внесении критического *изменения* в библиотеку для всех проектов необходимо внести изменения в источник, чтобы использовать новую библиотеку.

Если для изменения требуются новые языковые функции, вы принудительно выполните обновление этих проектов до той же языковой версии и TFM, которую вы используете. Вам потребуется больше работы для пользователей, и, возможно, заставили их обновить.

Влияние любых критических изменений зависит от количества проектов, имеющих зависимость от библиотеки. Если библиотека используется внутри нескольких приложений, вы можете реагировать на любые критические изменения во всех затронутых проектах. Однако если библиотека загружена публично, следует оценить потенциальное влияние и рассмотреть альтернативные варианты:

- Вы можете добавить новые API, которые параллелит существующие API.
- Вы можете рассмотреть параллельные сборки для разных TFM.
- Вы можете рассмотреть возможность многонацеливания.

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Руководство: Изучение объединённых типов в C#

Тип объединения содержит одно значение, которое является одним из фиксированных типов. Эти типы обычно не связаны отношениями наследования, поэтому объединение типов позволяет группировать несвязанные типы, не вынуждая сводить их к общему базовому классу или интерфейсу. Объединения полезны, если значение концептуально "одно из них", например датчик, считывающий это число, состояние переключателя или сообщение о состоянии. Поскольку набор случаев фиксирован, компилятор может проверить, что вы обрабатываете каждый случай, когда читаете значение.

В этом руководстве вы создадите слой показаний монитора телеметрии умного дома. Вы объявляете объединённые типы для данных датчика, заменяете самодельный тип результата на объединённый тип и создаёте пользовательское объединение для производительно-критичного кода.

Изучив это руководство, вы:

- ✓ Объявите типы профсоюзов и сопоставляйте их случаи исчерпывающим образом.
- ✓ Обработка null-значений в объединении.
- ✓ Добавьте участников и используйте универсальные шаблоны с объявлениями профсоюзов.
- ✓ Создайте пользовательский тип объединения, который избегает бокса.

Необходимые условия

В этом руководстве используются функции языка предварительной версии. Вам нужен пакет SDK, поддерживающий закрытые иерархии, и версия языка, установленная на `preview`. Чтобы приступить к работе, потребуется следующее.

- Пакет SDK для .NET 11 Preview 5 или более поздней версии. Скачайте его с сайта загрузки .NET.
- Редактор, например Visual Studio или Visual Studio Code с C# Dev Kit.

> ⓘ **Важно!**

Типы объединения — это функция предварительной версии. Синтаксис и вспомогательные типы могут изменяться перед тем, как функция будет доступна. Укажите `<LangVersion>preview</LangVersion>` в своём проекте, чтобы включить эту функцию.

Создание примера решения

Вы создаете библиотеку классов, `SmartHome.Core` которая содержит типы объединения и консольное приложение, `SmartHome.App` которое выполняет их.

1. Создайте решение и проекты:

   Интерфейс командной строки.NET

   ```
   dotnet new sln -n TelemetryMonitor
   dotnet new classlib --langversion preview -n SmartHome.Core
   dotnet new console --langversion preview -n SmartHome.App
   dotnet sln add SmartHome.Core SmartHome.App
   dotnet add SmartHome.App reference SmartHome.Core
   ```

2. Предыдущие команды содержали `--langversion preview` для включения функций предварительной версии C# 15. В каждом файле проекта убедитесь, что для версии языка задано значение `preview`:

   XML

   ```
   <PropertyGroup>
     <LangVersion>preview</LangVersion>
   </PropertyGroup>
   ```

Объявление типа объединения

Начните с необработанных данных датчика. Показание поступает в виде числа, состояния переключателя или сообщения о статусе, поэтому вы моделируете его как объединение этих трёх типов.

1. В `SmartHome.Core` добавьте файл с именем `Readings.cs` и объявите объединение `Reading`. *Объявление union* перечисляет типы вариантов в скобках:

   C#

```
// A union declaration is shorthand for a struct that holds one of the listed
// case types. The 'string?' case makes the contents nullable.
public union Reading(double, bool, string?);
```

2. В том же файле добавьте метод, который считывает `Reading`, выполняя переключение по типам его вариантов:

C#

```
public static string Render(Reading reading) => reading switch
{
    // Each type pattern applies to the union's contents, not to the Reading
value.
    double measure => $"{measure:F1}",
    bool isOn => isOn ? "on" : "off",
    string text => text,
    // The contents can be null because 'string?' is a nullable case type.
    null => "no reading",
};
```

Каждый образец типа применяется к *содержимому* объединения, а не к экземпляру `Reading`, поскольку большинство образцов распаковывают объединение. Вариант `string?` может принимать значение null, поэтому компилятор рассматривает null как отдельный вариант и требует ветвь `null`. Параметр является исчерпывающим: если удалить любую руку, компилятор сообщает, что дело не рассматривается. Это гарантия является одним из преимуществ профсоюза. Если позже добавить новый вариант в объединение, каждый switch, который работает с этим объединением, укажет на отсутствующую ветку. Дополнительные сведения см. в разделе "Исчерпывающие возможности объединения " и "шаблоны союза".

Преобразование типа результата в обобщённое объединение

Если чтение с датчика может завершиться ошибкой, нужен тип, который содержит либо значение, либо ошибку. Во многих кодовых базах этот тип представляют с помощью класса, который содержит флаг успешного выполнения и два поля. Замените этот класс универсальным объединением.

1. В `SmartHome.Core` добавьте файл с именем `Result.cs` с самописным классом результата, который заменяется этой миграцией:

C#

```
// A conventional result type. Callers must remember to check 'IsSuccess' before
// they read 'Value', and nothing stops them from reading the wrong field.
public sealed class QueryResult<T>
{
    private QueryResult(bool isSuccess, T? value, Exception? error)
    {
        IsSuccess = isSuccess;
        Value = value;
        Error = error;
    }

    public bool IsSuccess { get; }
    public T? Value { get; }
    public Exception? Error { get; }

    public static QueryResult<T> Success(T value) => new(true, value, null);
    public static QueryResult<T> Failure(Exception error) => new(false, default,
error);
}
```

2. Замените этот класс универсальным объединением. `Result<T>` содержит либо `T`, либо `Exception`:

C#

```
// The same idea as a union declaration. A Result<T> holds either a T or an
Exception.
public union Result<T>(T, Exception);
```

3. Добавьте метод, который считывает результат, переключив его типы вариантов:

C#

```
public static string Describe<T>(Result<T> result) => result switch
{
    T value => $"Success: {value}",
    Exception error => $"Failure: {error.Message}",
    null => "Failure: no value",
};
```

Исходный класс позволяет вызывающему коду ошибочно считывать `Value` при сбое или `Error` при успешном выполнении. Объединение устраняет эти недопустимые состояния: `Result<T>` представляет собой либо `T`, который содержит результат успешного выполнения, либо `Exception`, который объясняет, почему операция завершилась ошибкой. Каждый случай преобразуется в объединение неявно, поэтому вы назначаете экземпляр `T` или экземпляр `Exception` непосредственно экземпляру `Result<T>`. При чтении результата

переключатель проверяет, что вы обрабатываете значение успешного выполнения, сбой и null. Дополнительные сведения см. в объявлениях и преобразованиях профсоюзов.

Повторное использование универсального объединения между числовыми типами

Датчики передают данные с разным разрешением: датчик контакта использует `byte`, датчик положения использует `short`, а счетчик использует `int`. Вместо того чтобы писать отдельное объединение для каждого типа, напишите одно универсальное объединение и инстанцируйте его для каждого числового типа.

1. В `SmartHome.Core` добавьте файл с именем `Sample.cs` и объявите объединение `Sample<T>`. Оно содержит либо показание типа `T`, находящееся в допустимом диапазоне, либо маркер `Saturated` для датчика, достигшего предела измерения. Добавьте объединению тело, которое добавляет свойство `InRange`:

   C#

   ```
   // A generic union, reusable across sensors of different numeric precision. A
   // Sample<T> holds either an in-range reading of type T or a Saturated marker
   // for a sensor that railed. The 'struct, INumber<T>' constraint keeps T a
   // non-nullable numeric value type, so a switch over the cases needs no null arm.
   public union Sample<T>(T, Saturated) where T : struct, INumber<T>
   {
       // A union can declare members. This property patterns over 'this' to report
       // whether the sample carries a usable reading.
       public bool InRange => this is T;
   }

   // A marker for a railed sensor, carrying which rail it hit.
   public readonly record struct Saturated(bool High);
   ```

2. Добавьте метод `Normalize`, который масштабирует любое значение до доли полной шкалы:

   C#

   ```
   // Scale any numeric width to a fraction of full scale. The INumber<T>
   // constraint makes the same arithmetic work for byte, short, int, and more.
   public static double Normalize<T>(Sample<T> sample, T fullScale)
       where T : struct, INumber<T> => sample switch
   {
       T value => double.CreateChecked(value) / double.CreateChecked(fullScale),
       Saturated { High: true } => 1.0,
   ```

```
    Saturated => 0.0,
};
```

Ограничение `where T : struct, INumber<T>` требует, чтобы `T` был числовым типом значения, не допускающим null, и позволяет одному методу обрабатывать числовые типы любой разрядности с помощью арифметики INumber<TSelf>. Тот же метод `Normalize` работает для `Sample<byte>`, `Sample<short>` или `Sample<int>`. Поскольку `Saturated` является типом значения, а `T` ограничен типом значения, ни один из случаев не допускает значения null, поэтому конструкция switch является исчерпывающей без ветви null. Свойство `InRange` показывает, что объявление профсоюза может содержать собственные члены и шаблон.`this` Дополнительные сведения см. в разделе Объявления объединений.

Создание пользовательского объединения, которое позволяет избежать бокса

Объявление объединения представляет собой структуру, которая хранит значение в единственном поле `object?`, поэтому вариант со значимым типом при сохранении упаковывается. В коде, критичном к производительности, реализуйте паттерн union вручную, чтобы хранить значение без упаковки.

1. В `SmartHome.Core` добавьте файл с именем `Quantity.cs`. Примените атрибут `[Union]` к структуре и определите члены доступа, которые компилятор использует для сопоставления каждого варианта:

C#

```
// A custom union type. The 'union' shorthand boxes value-type cases, so a
// performance-sensitive type can implement the union pattern by hand to store
// the value without boxing. The [Union] attribute opts the struct into union
// behaviors; the non-boxing access members (HasValue and TryGetValue) let the
// compiler match each case without boxing.
[Union]
public readonly struct Quantity : IUnion
{
    private readonly double _value;
    private readonly bool _isCount;

    public Quantity(int count) => (_value, _isCount) = (count, true);
    public Quantity(double measure) => (_value, _isCount) = (measure, false);

    public object? Value => _isCount ? (int)_value : _value;

    public bool HasValue => true;
    public bool TryGetValue(out int value)
    {
```

```
        value = (int)_value;
        return _isCount;
    }
    public bool TryGetValue(out double value)
    {
        value = _value;
        return !_isCount;
    }
}
```

2. Добавьте метод, который обрабатывает пользовательское объединение так же, как вы обрабатываете объявление объединения:

```C#
public static string Describe(Quantity quantity) => quantity switch
{
    int count => $"{count} items",
    double measure => $"{measure:F2} units",
};
```

Свойство `HasValue` и `TryGetValue` перегрузки позволяют компилятору сопоставлять каждый случай без бокса. Структура хранит значение в поле `double` и использует дискриминатор, поэтому ни вариант `int`, ни вариант `double` не требует выделения памяти. Оператор switch для `int` и `double` является исчерпывающим, и ветвь null не требуется, поскольку ни один из типов вариантов не допускает значения null. Дополнительные сведения см. в разделе "Типы пользовательских союзов".

> ⓘ **Примечание.**
>
> Другим сценарием для пользовательского объединения является добавление поддержки языка в существующий тип, который моделирует объединение. Если в вашем приложении есть типы, аналогичные объединениям, которые вы создали до появления типов `union` в C#, примените к ним атрибут `Union` и реализуйте для них интерфейс `IUnion`. Затем компилятор обеспечивает пользовательскому типу ту же языковую поддержку, что и объявлению объединения. Дополнительные сведения см. в спецификации компонентов профсоюзов.

Запустите пример

Консольное приложение управляет каждым объединением. Замените содержимое `Program.cs`:

C#

```
// Union declarations: a Reading holds a double, bool, or string.
Reading[] readings = [23.5, true, "offline"];
foreach (Reading reading in readings)
{
    Console.WriteLine($"Reading: {ReadingReporter.Render(reading)}");
}
Console.WriteLine($"Reading: {ReadingReporter.Render(default)}");

// Migrate from a conventional result type to a union.
Result<double> ok = 42.0;
Result<double> bad = new TimeoutException("sensor timed out");
Console.WriteLine(ResultReporter.Describe(ok));
Console.WriteLine(ResultReporter.Describe(bad));

// Generic union reused across numeric widths.
Sample<byte> raw = (byte)200;
Sample<short> railed = new Saturated(High: true);
Console.WriteLine($"Sample: {SampleReporter.Normalize(raw, byte.MaxValue):P0} (in
range: {raw.InRange})");
Console.WriteLine($"Sample: {SampleReporter.Normalize(railed, short.MaxValue):P0} (in
range: {railed.InRange})");

// Custom non-boxing union.
Console.WriteLine(QuantityReporter.Describe(new Quantity(3)));
Console.WriteLine(QuantityReporter.Describe(new Quantity(2.5)));
```

Запустите приложение:

Интерфейс командной строки.NET

```
dotnet run --project SmartHome.App
```

Показания, результаты, образцы и количества печатаются каждый через свой соответствующий переключатель:

Выходные данные

```
Reading: 23.5
Reading: on
Reading: offline
Reading: no reading
Success: 42
Failure: sensor timed out
Sample: 78% (in range: True)
Sample: 100% (in range: False)
3 items
2.50 units
```

Сводка

Вы создали слой показаний в мониторе телеметрии умного дома и в процессе разобрались с основными сценариями объединения. Ты:

- Объявил `Reading` объединение и обработал его с помощью исчерпывающего switch, позволяя компилятору пометить любой случай, который вы не обработали.
- Обработан случай, допускающий значение NULL, когда компилятору требуется ветка `null`.
- Самодельный тип результата заменён обобщённым `Result<T>`-объединением, которое моделирует успех или неудачу без недопустимых состояний.
- Повторно использовано универсальное объединение `Sample<T>` для нескольких числовых типов с ограничением, и в объявление объединения добавлены члены.
- Созданная настраиваемая `[Union]` структура, которая позволяет избежать бокса для кода с учетом производительности.

Связанные материалы

- Руководство по закрытым иерархиям
- Обзор сопоставления шаблонов
- Новые возможности C# 15

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 08.07.2026

Руководство. Изучение закрытых иерархий C#

Закрытая иерархия ограничивает прямые подтипы базового типа сборкой, объявляющей ее. Так как компилятор знает полный набор подтипов, он может убедиться, что `switch` выражение является исчерпывающим без руки по умолчанию. Закрытые иерархии подходят для доменов, в которых набор вариантов является стабильным, и компилятор должен пометить все места, которые необходимо изменить при добавлении дела.

В этом руководстве вы создадите модель датчика для монитора телеметрии умного дома. Вы объявляете закрытую иерархию типов датчиков, сопоставляете датчики исчерпывающим образом и решаете, какие случаи следует запечатывать и какие оставить открытыми для расширения.

Изучив это руководство, вы:

- ✓ Объявите закрытую иерархию и исчерпывающе сопоставьте все её варианты.
- ✓ Определите, какие подтипы следует запечатывать и какие оставить открытыми.
- ✓ Расширение открытого регистра из другой сборки.
- ✓ Используйте закрытую иерархию с универсальными шаблонами.

Необходимые условия

В этом руководстве используются функции языка предварительной версии. Вам нужен пакет SDK, поддерживающий закрытые иерархии, и версия языка, установленная на `preview`. Чтобы приступить к работе, потребуется следующее.

- Пакет SDK для .NET 11 Preview 5 или более поздней версии. Скачайте его с сайта загрузки .NET.
- Редактор, например Visual Studio или Visual Studio Code с C# Dev Kit.

> ⓘ **Важно!**
>
> Закрытые иерархии — это предварительная версия функции. Синтаксис и поведение могут измениться перед тем, как функция будет доступна. Укажите `<LangVersion>preview</LangVersion>` в своём проекте, чтобы включить эту функцию.

Создание примера решения

Вы создаете библиотеку классов, `SmartHome.Core` которая содержит закрытую иерархию, вторую библиотеку, `SmartHome.Extensions` которая расширяет открытый регистр и консольное приложение, `SmartHome.App` которое управляет ими.

1. Откройте терминал и выполните следующие `dotnet` команды из новой папки, созданной ранее для хранения кода для учебника:

   Интерфейс командной строки.NET

   ```
   dotnet new sln -n TelemetryMonitor
   dotnet new classlib --langversion preview -n SmartHome.Core
   dotnet new classlib --langversion preview -n SmartHome.Extensions
   dotnet new console --langversion preview -n SmartHome.App
   dotnet sln add SmartHome.Core SmartHome.Extensions SmartHome.App
   dotnet add SmartHome.Extensions reference SmartHome.Core
   dotnet add SmartHome.App reference SmartHome.Core SmartHome.Extensions
   ```

2. Предыдущие команды содержали `--langversion preview` для включения функций предварительной версии C# 15. В каждом файле проекта убедитесь, что для версии языка задано значение `preview`:

   XML

   ```
   <PropertyGroup>
     <LangVersion>preview</LangVersion>
   </PropertyGroup>
   ```

Объявление закрытой иерархии

Начните с модели датчика. Монитор поддерживает фиксированный набор типов датчиков, поэтому их можно моделировать как закрытую иерархию в одной сборке.

1. Добавьте `SmartHome.Core` файл с именем `Sensors.cs` и объявите базовый `Sensor` тип с модификатором `closed` и тремя производными типами:

   C#

   ```
   // A closed class restricts its direct subtypes to this assembly. A 'closed'
   // class is implicitly abstract.
   public closed record class Sensor;

   // Seal the cases whose shape is final.
   public sealed record class Temperature(double Celsius) : Sensor;
   ```

```
public sealed record class Humidity(double Percent) : Sensor;

// Leave a case unsealed as an "escape hatch" so other assemblies can specialize
it.
public record class Contact(bool Open) : Sensor;
```

2. В том же файле добавьте метод, который сопоставляет каждый датчик с помощью switch-выражения:

C#

```
public static string Describe(Sensor sensor) => sensor switch
{
    Temperature temperature => $"{temperature.Celsius:F1}°C",
    Humidity humidity => $"{humidity.Percent:F0}% RH",
    Contact contact => contact.Open ? "open" : "closed",
    // No default arm is needed. Sensor is closed, so these cases are exhaustive.
};
```

3. Создайте проект. .NET 11 Preview 5 поддерживает замкнутые иерархии на уровне языка, но необходимый тип добавят только в одной из следующих предварительных версий. Если возникают ошибки сборки, необходимо добавить полифил для атрибута `Closed`. Добавьте следующий код в `SmartHome.Core` проект в файле с именем `ClosedPolyfill.cs`:

C#

```
namespace System.Runtime.CompilerServices;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited =
false)]
public sealed class ClosedAttribute : Attribute { }
```

Модификатор `closed` ограничивает прямые подтипы `Sensor` сборкой, в которой он объявлен, а тип `closed` неявно является абстрактным, поэтому нельзя создать его экземпляр напрямую. Поскольку компилятор знает полный набор подтипов, оператору switch не нужна ветка default. При добавлении нового типа датчика позже компилятор сообщает, что этот параметр больше не охватывает каждый случай. Эта обратная связь является центральным преимуществом закрытой иерархии. Компилятор указывает на каждое совпадение, которое необходимо изменить. Варианты `Temperature` и `Humidity` являются закрытыми, поскольку их форма уже окончательно определена, тогда как вариант `Contact` остаётся открытым в качестве точки расширения, которую рассматривает следующий раздел. Дополнительные сведения см. в разделе closedмодификатор и Шаблоны закрытой иерархии.

Решите, что запечатывать

Подтип закрытого типа не закрыт, если вы не объявите его таким образом. Для каждого подтипа у вас есть три варианта:

- Пометьте его `closed` , чтобы продолжить иерархию: разрешены дополнительные подтипы, но только в той же сборке.
- Пометьте его `sealed` в конце иерархии: никакие дополнительные подтипы не допускаются в любом месте.
- Оставьте его незамеченным, чтобы сделать его открытым: другие сборки могут быть производными от него, и эти производные типы по-прежнему соответствуют исходному регистру.

Вы оставили `Contact` непомеченным по этой причине, поэтому теперь расширяете его из другой сборки.

1. В `SmartHome.Extensions` добавьте файл с именем `DoorContact.cs`, который наследуется от открытого случая `Contact`:

   C#

   ```
   // This type lives in a different assembly than the closed Sensor hierarchy.
   // It can't derive from Sensor directly, but it can extend the unsealed Contact
   // leaf. A DoorContact still matches the 'Contact' case, so switches over Sensor
   // stay exhaustive.
   public sealed record class DoorContact(bool Open, string Door) : Contact(Open);
   ```

2. В `SmartHome.App` добавьте код для обработки `DoorContact` через существующий оператор `Sensor`:

   C#

   ```
   // A DoorContact from another assembly still matches the Contact case.
   Sensor frontDoor = new DoorContact(Open: false, Door: "Front");
   Console.WriteLine($"Sensor: {SensorReporter.Describe(frontDoor)}");
   ```

`DoorContact` не может напрямую наследоваться от `Sensor`, так как он закрыт для других сборок. `DoorContact` вместо этого расширяет открытую `Contact` ветвь. `DoorContact` по-прежнему соответствует случаю `Contact`, поэтому исчерпывающий оператор switch по `Sensor` остаётся корректным без каких-либо изменений. При выборе того, какой вариант сделать закрытым, взвесьте компромисс: закройте вариант, чтобы зафиксировать его структуру и сохранить иерархию полностью известной, или оставьте вариант открытым,

чтобы допустить расширение ценой менее точного сопоставления, поскольку конструкция switch видит открытый базовый вариант, а не производный тип. Дополнительные сведения см. в модификаторeclosed.

Использование закрытой иерархии с универсальными шаблонами

Закрытая иерархия может быть универсальной. Отчет, поступающий от монитора, представляет собой либо отдельное значение, либо группу вложенных отчетов, поэтому его следует моделировать как обобщенную закрытую иерархию.

1. В `SmartHome.Core` добавьте файл с именем `Report.cs` и объявите закрытый базовый тип `Report<T>` с двумя его вариантами. Производный тип не может ввести параметр типа, который не имеет базового типа, но он может предоставить фиксированные аргументы для одного или нескольких параметров типа базового типа:

   C#

   ```
   // A generic closed hierarchy. Every type parameter of a derived type must appear
   // in the base type, so a single derived construction exhausts each Report<T>.
   public closed record class Report<T>;
   public sealed record class Single<T>(T Value) : Report<T>;
   public sealed record class Group<T>(Report<T> Left, Report<T> Right) : Report<T>;
   ```

2. Добавьте рекурсивный метод, который накапливает отчет, переключив его варианты:

   C#

   ```
   public static int Count<T>(Report<T> report) => report switch
   {
       Single<T> => 1,
       Group<T> group => Count(group.Left) + Count(group.Right),
       // Exhaustive: Report<T> is closed and both subtypes are handled.
   };
   ```

Переключение `Single<T>` и `Group<T>` является исчерпывающим, так как `Report<T>` закрыто, поэтому рекурсивный накопитель не нуждается в руке по умолчанию. Дополнительные сведения см. в разделе "Закрытые шаблоны иерархии".

Запустите пример

Консольное приложение создает каждый датчик и отчет, а затем выводит их с помощью исчерпывающих коммутаторов. Откройте `Program.cs` и добавьте следующий код:

C#

```
// Closed hierarchy with an exhaustive switch.
Sensor[] sensors =
[
    new Temperature(21.4),
    new Humidity(55),
    new Contact(Open: true),
];
foreach (Sensor sensor in sensors)
{
    Console.WriteLine($"Sensor: {SensorReporter.Describe(sensor)}");
}

// A DoorContact from another assembly still matches the Contact case.
Sensor frontDoor = new DoorContact(Open: false, Door: "Front");
Console.WriteLine($"Sensor: {SensorReporter.Describe(frontDoor)}");

// Generic closed hierarchy.
Report<string> report = new Group<string>(
    new Single<string>("Kitchen"),
    new Group<string>(new Single<string>("Garage"), new Single<string>("Attic")));
Console.WriteLine($"Report leaves: {ReportReporter.Count(report)}");
```

Затем запустите приложение:

Интерфейс командной строки.NET

```
dotnet run --project SmartHome.App
```

Датчики и отчет выводятся на печать через все доступные переключатели:

Выходные данные

```
Sensor: 21.4°C
Sensor: 55% RH
Sensor: open
Sensor: closed
Report leaves: 3
```

Сводка

Вы создали модель датчика монитора телеметрии smart-home и в процессе работали с сценариями закрытой иерархии. Ты:

- Объявлен базовый тип `closed Sensor`, а его подтипы сопоставлены с помощью исчерпывающего оператора switch, которому не нужна ветка default.
- Рассматривались три варианта для каждого подтипа: `closed` продолжить иерархию в той же сборке, `sealed` завершить её или оставить без пометки, чтобы сохранить точку расширения.
- Расширен открытый `Contact` регистр из отдельной сборки `DoorContact` и подтвердил, что он по-прежнему соответствует `Contact` регистру в существующем коммутаторе.
- Объявил обобщённую закрытую иерархию, `Report<T>`, и свернул её с помощью рекурсивного исчерпывающего switch-выражения.

Связанные материалы

- Руководство по типам объединения
- Обзор сопоставления шаблонов
- Новые возможности C# 15

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 08.07.2026

Руководство: изучение членов-расширений в C# 14 и C# 15

В C# 14 появились члены-расширения — развитие существующих методов расширения. Члены расширения позволяют добавлять свойства и операторы. Можно также расширять как типы, так и их экземпляры. C# 15 добавляет индексаторы расширений, поэтому существующий тип может поддерживать индексированные доступ из блока расширения.

В этом руководстве вы изучите элементы расширения, дополняя тип `System.Drawing.Point` математическими операциями, преобразованиями координат и утилитными свойствами. Затем вы добавляете доступ по индексу к типу `Path`, который хранит смещения между точками. Вы узнаете, как перенести существующие методы расширения на новый синтаксис членов расширения и в каких случаях подходит каждый из этих подходов.

Изучив это руководство, вы:

- ✓ Создайте члены расширения C# 14 со статическими свойствами и операторами.
- ✓ Реализуйте преобразования координат с помощью элементов расширения.
- ✓ Перенос традиционных методов расширения в синтаксис члена расширения.
- ✓ Добавьте индексатор расширения C# 15, который считывает и обновляет абсолютные точки пути.
- ✓ Сравнивайте члены расширения с традиционными методами расширения.

Предпосылки

- Пакет SDK для предварительной версии .NET 11. Скачайте его с сайта загрузки .NET.
- Visual Studio 2026 с включенными функциями предварительной версии. Скачайте его с страницы Visual Studio.
- В этом примере задается `<LangVersion>preview</LangVersion>`, поскольку индексаторы расширений являются возможностью предварительной версии C# 15.

Создание примера приложения

Начните с создания консольного приложения, демонстрирующего как традиционные методы расширения, так и синтаксис новых элементов расширения. Вы создаете расширения для System.Drawing.Point типа. Этот тип поставляется из `System.Drawing` пространства имен и обычно используется в приложениях Windows Forms.

1. Создайте консольное приложение.

Интерфейс командной строки.NET

```
dotnet new console -n PointExtensions
cd PointExtensions
```

2. Обновите файл проекта, чтобы пример был ориентирован на .NET 11 и использовал предварительные возможности языка:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net11.0</TargetFramework>
    <LangVersion>preview</LangVersion>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

3. Скопируйте следующий код в новый файл с именем `ExtensionMethods.cs`:

C#

```
using System.Drawing;
using System.Numerics;

namespace ExtensionMethods;

public static class PointExtensions
{
    public static Vector2 ToVector(this Point point) =>
        new Vector2(point.X, point.Y);

    public static void Translate(this ref Point point, int xDist, int yDist)
    {
        point.X += xDist;
        point.Y += yDist;
    }

    public static void Scale(this ref Point point, int xScale, int yScale)
    {
        point.X *= xScale;
        point.Y *= yScale;
    }
```

```
    public static void Rotate(this ref Point point, int angleInDegrees)
    {
        double theta = ((double)angleInDegrees * Math.PI) / 180.0;
        double sinTheta = Math.Sin(theta);
        double cosTheta = Math.Cos(theta);
        double newX = (double)point.X * cosTheta - (double)point.Y * sinTheta;
        double newY = (double)point.X * sinTheta + (double)point.Y * cosTheta;
        point.X = (int)newX;
        point.Y = (int)newY;
    }
}
```

4. Скопируйте следующий код, демонстрирующий эти методы расширения и другие способы использования System.Drawing.Point:

C#

```
using System.Drawing;
using System.Numerics;
using ExtensionMethods;

public static class ExtensionMethodsDemonstrations
{
    public static void TraditionalExtensionMethods()
    {
        OriginAsADataElement();
        ArithmeticWithPoints();
        DiscreteArithmeticWithPoints();
        ExtensionMethodsThis();
        MoreExamples();
    }

    static void OriginAsADataElement()
    {
        // Inline implementation since Point.Origin doesn't exist in
ExtensionMethods
        Point origin = Point.Empty; // Equivalent to Point.Origin
        Console.WriteLine($"Point.Origin (inline): {origin}");
        Console.WriteLine($"Same as Point.Empty: {origin == Point.Empty}");
        Console.WriteLine();
    }

    static void ArithmeticWithPoints()
    {
        Point p1 = new Point(5, 3);
        Point p2 = new Point(2, 7);

        Console.WriteLine($"Point 1: {p1}");
        Console.WriteLine($"Point 2: {p2}");

        // Inline implementation since + and - operators don't exist in
ExtensionMethods
        Point addition = new Point(p1.X + p2.X, p1.Y + p2.Y);
```

```
        Point subtraction1 = new Point(p1.X - p2.X, p1.Y - p2.Y);
        Point subtraction2 = new Point(p2.X - p1.X, p2.Y - p1.Y);

        Console.WriteLine($"Addition (p1 + p2): {addition}");
        Console.WriteLine($"Subtraction (p1 - p2): {subtraction1}");
        Console.WriteLine($"Subtraction (p2 - p1): {subtraction2}");
        Console.WriteLine();
    }

    static void DiscreteArithmeticWithPoints()
    {
        Point point = new Point(10, 8);
        int offsetX = 3;
        int offsetY = -2;
        int scaleX = 2;
        int scaleY = 3;
        int divisorX = 2;
        int divisorY = 4;

        Console.WriteLine($"Original point: {point}");
        Console.WriteLine($"Offset: ({offsetX}, {offsetY})");
        Console.WriteLine($"Scale: ({scaleX}, {scaleY})");
        Console.WriteLine($"Divisor: ({divisorX}, {divisorY})");
        Console.WriteLine();

        // Inline implementations since tuple operators don't exist in
ExtensionMethods
        Point addedOffset = new Point(point.X + offsetX, point.Y + offsetY);
        Point subtractedOffset = new Point(point.X - offsetX, point.Y - offsetY);
        Point scaledPoint = new Point(point.X * scaleX, point.Y * scaleY);
        Point dividedPoint = new Point(point.X / divisorX, point.Y / divisorY);

        Console.WriteLine($"point + offset: {addedOffset}");
        Console.WriteLine($"point - offset: {subtractedOffset}");
        Console.WriteLine($"point * scale: {scaledPoint}");
        Console.WriteLine($"point / divisor: {dividedPoint}");
        Console.WriteLine();
    }

    static void ExtensionMethodsThis()
    {
        // ToVector demonstration - using extension method
        Point vectorPoint = new Point(12, 16);
        Vector2 vector = vectorPoint.ToVector();
        Console.WriteLine($"Point {vectorPoint} as Vector2: {vector}");
        Console.WriteLine();

        // Translate demonstration - using extension method
        Point translatePoint = new Point(5, 5);
        Console.WriteLine($"Before Translate: {translatePoint}");
        translatePoint.Translate(3, -2);
        Console.WriteLine($"After Translate(3, -2): {translatePoint}");
        Console.WriteLine();

        // Scale demonstration - using extension method
```

```
        Point scalePoint = new Point(4, 6);
        Console.WriteLine($"Before Scale: {scalePoint}");
        scalePoint.Scale(2, 3);
        Console.WriteLine($"After Scale(2, 3): {scalePoint}");
        Console.WriteLine();

        // Rotate demonstration - using extension method
        Point rotatePoint1 = new Point(10, 0);
        Console.WriteLine($"Before Rotate: {rotatePoint1}");
        rotatePoint1.Rotate(90);
        Console.WriteLine($"After Rotate(90°): {rotatePoint1}");

        Point rotatePoint2 = new Point(5, 5);
        Console.WriteLine($"Before Rotate: {rotatePoint2}");
        rotatePoint2.Rotate(45);
        Console.WriteLine($"After Rotate(45°): {rotatePoint2}");

        Point rotatePoint3 = new Point(3, 4);
        Console.WriteLine($"Before Rotate: {rotatePoint3}");
        rotatePoint3.Rotate(180);
        Console.WriteLine($"After Rotate(180°): {rotatePoint3}");
        Console.WriteLine();
    }

    static void MoreExamples()
    {
        // Combining operators and methods
        Console.WriteLine("Scenario 1: Building a rectangle using inline
operators");
        Point topLeft = Point.Empty; // Inline equivalent of Point.Origin
        Point bottomRight = new Point(topLeft.X + 10, topLeft.Y + 8); // Inline
addition
        Point topRight = new Point(bottomRight.X, topLeft.Y);
        Point bottomLeft = new Point(topLeft.X, bottomRight.Y);

        Console.WriteLine($"Rectangle corners:");
        Console.WriteLine($"  Top-Left: {topLeft}");
        Console.WriteLine($"  Top-Right: {topRight}");
        Console.WriteLine($"  Bottom-Left: {bottomLeft}");
        Console.WriteLine($"  Bottom-Right: {bottomRight}");
        Console.WriteLine();

        // Transformation chain
        Console.WriteLine("Scenario 2: Transformation chain (mixed methods)");
        Point transformPoint = new Point(2, 3);
        Console.WriteLine($"Starting point: {transformPoint}");

        // Scale up - using extension method
        transformPoint.Scale(3, 2);
        Console.WriteLine($"After scaling by (3, 2): {transformPoint}");

        // Translate - using inline addition
        transformPoint = new Point(transformPoint.X + 5, transformPoint.Y +
(-3));
        Console.WriteLine($"After translating by (5, -3): {transformPoint}");
```

```
        // Rotate - using extension method
        transformPoint.Rotate(45);
        Console.WriteLine($"After rotating 45�: {transformPoint}");

        // Convert to vector - using extension method
        Vector2 finalVector = transformPoint.ToVector();
        Console.WriteLine($"Final result as Vector2: {finalVector}");
        Console.WriteLine();

        // Distance calculation using inline operators and extension methods
        Console.WriteLine("Scenario 3: Distance calculation (mixed methods)");
        Point point1 = new Point(1, 1);
        Point point2 = new Point(4, 5);
        Point difference = new Point(point2.X - point1.X, point2.Y - point1.Y);
// Inline subtraction
        Vector2 diffVector = difference.ToVector(); // Extension method
        float distance = diffVector.Length();

        Console.WriteLine($"Point 1: {point1}");
        Console.WriteLine($"Point 2: {point2}");
        Console.WriteLine($"Difference: {difference}");
        Console.WriteLine($"Distance: {distance:F2}");
        Console.WriteLine();

        Console.WriteLine("Traditional extension methods demonstration
complete!");
    }
}
```

5. Замените содержимое `Program.cs` демонстрационным кодом.

C#

```
ExtensionMethodsDemonstrations.TraditionalExtensionMethods();
```

6. Запустите пример приложения и проверьте выходные данные.

Традиционные методы расширения могут добавлять только методы экземпляров в существующие типы. Члены расширения позволяют добавлять статические свойства, что обеспечивает более естественный способ расширения типов с константами или вычисляемых значений.

Добавить статические свойства с помощью элементов расширения

Сначала изучите этот код в примере:

C#

```
// Inline implementation since Point.Origin doesn't exist in ExtensionMethods
Point origin = Point.Empty; // Equivalent to Point.Origin
Console.WriteLine($"Point.Origin (inline): {origin}");
Console.WriteLine($"Same as Point.Empty: {origin == Point.Empty}");
Console.WriteLine();
```

Многие приложения, использующие 2D-геометрию, используют концепцию , `Origin` которая совпадает с Point.Emptyзначением. Этот код использует этот факт, но некоторые разработчики могут создать `new Point(0,0)`, что приводит к дополнительной работе. В определенном домене необходимо выразить эти общие значения через статические свойства.

Создайте `NewExtensionsMembers.cs` элементы расширения, которые решают эту проблему:

C#

```
using System.Drawing;
using System.Numerics;

namespace ExtensionMembers;

public static class PointExtensions
{
    extension (Point)
    {
        public static Point Origin => Point.Empty;
    }
}
```

Приведенный выше код добавляет статическое свойство расширения к структуре. Ключевое `extension` слово вводит *блок расширения*. Этот блок расширения расширяет структуру `Point` .

Это статическое свойство можно использовать, как будто он был членом `Point` структуры.

C#

```
Console.WriteLine("1. Static Properties");
Console.WriteLine("--------------------");

Point origin = Point.Origin;
Console.WriteLine($"Point.Origin: {origin}");
Console.WriteLine($"Same as Point.Empty: {origin == Point.Empty}");
Console.WriteLine();
```

Теперь `Point.Origin` свойство отображается, как если бы оно было частью исходного `Point` типа, предоставляя более интуитивно понятный API.

Внедрить арифметические операторы

Затем изучите следующий код, выполняющий арифметику с точками:

C#

```
Point p1 = new Point(5, 3);
Point p2 = new Point(2, 7);

Console.WriteLine($"Point 1: {p1}");
Console.WriteLine($"Point 2: {p2}");

// Inline implementation since + and - operators don't exist in ExtensionMethods
Point addition = new Point(p1.X + p2.X, p1.Y + p2.Y);
Point subtraction1 = new Point(p1.X - p2.X, p1.Y - p2.Y);
Point subtraction2 = new Point(p2.X - p1.X, p2.Y - p1.Y);

Console.WriteLine($"Addition (p1 + p2): {addition}");
Console.WriteLine($"Subtraction (p1 - p2): {subtraction1}");
Console.WriteLine($"Subtraction (p2 - p1): {subtraction2}");
Console.WriteLine();
```

Традиционные методы расширения не могут добавлять операторы в существующие типы. Необходимо вручную реализовать арифметические операции, что делает код подробным и трудным для чтения. Алгоритм дублируется всякий раз, когда требуется операция, которая создает больше возможностей для небольших ошибок для ввода базы кода. Лучше разместить этот код в одном месте. Добавьте в блок `NewExtensionsMembers.cs` расширения следующие операторы:

C#

```
public static Point operator +(Point left, Point right) =>
    new Point(left.X + right.X, left.Y + right.Y);

public static Point operator -(Point left, Point right) =>
    new Point(left.X - right.X, left.Y - right.Y);
```

С помощью членов расширения можно добавлять операторы непосредственно в существующие типы. Теперь можно выполнять арифметические операции с помощью естественного синтаксиса:

C#

```
Console.WriteLine("2. Arithmetic Operators (Point + Point, Point - Point)");
Console.WriteLine("-----------------------------------------------------");

Point p1 = new Point(5, 3);
Point p2 = new Point(2, 7);

Console.WriteLine($"Point 1: {p1}");
Console.WriteLine($"Point 2: {p2}");
Console.WriteLine($"Addition (p1 + p2): {p1 + p2}");
Console.WriteLine($"Subtraction (p1 - p2): {p1 - p2}");
Console.WriteLine($"Subtraction (p2 - p1): {p2 - p1}");
Console.WriteLine();
```

Операторы расширения делают арифметику с точками такой же естественной, как работу со встроенными числовыми типами.

Добавление дополнительных операторов

Можно также добавить операторы расширения для дискретных операций, показанных в следующем примере кода:

C#

```
Point point = new Point(10, 8);
int offsetX = 3;
int offsetY = -2;
int scaleX = 2;
int scaleY = 3;
int divisorX = 2;
int divisorY = 4;

Console.WriteLine($"Original point: {point}");
Console.WriteLine($"Offset: ({offsetX}, {offsetY})");
Console.WriteLine($"Scale: ({scaleX}, {scaleY})");
Console.WriteLine($"Divisor: ({divisorX}, {divisorY})");
Console.WriteLine();

// Inline implementations since tuple operators don't exist in ExtensionMethods
Point addedOffset = new Point(point.X + offsetX, point.Y + offsetY);
Point subtractedOffset = new Point(point.X - offsetX, point.Y - offsetY);
Point scaledPoint = new Point(point.X * scaleX, point.Y * scaleY);
Point dividedPoint = new Point(point.X / divisorX, point.Y / divisorY);

Console.WriteLine($"point + offset: {addedOffset}");
Console.WriteLine($"point - offset: {subtractedOffset}");
Console.WriteLine($"point * scale: {scaledPoint}");
Console.WriteLine($"point / divisor: {dividedPoint}");
Console.WriteLine();
```

Операторы `+` и `-` являются *двоичными операторами* и требуют двух операндов, а не трех. Вместо двух дискретных целых чисел используйте *кортеж* , чтобы указать оба `X` значения и `Y` разностные значения:

C#

```
public static Point operator *(Point left, (int dx, int dy) scale) =>
    new Point(left.X * scale.dx, left.Y * scale.dy);
public static Point operator /(Point left, (int dx, int dy) scale) =>
    new Point(left.X / scale.dx, left.Y / scale.dy);
public static Point operator +(Point left, (int dx, int dy) scale) =>
    new Point(left.X + scale.dx, left.Y + scale.dy);
public static Point operator -(Point left, (int dx, int dy) scale) =>
    new Point(left.X - scale.dx, left.Y - scale.dy);
```

Предыдущий оператор включает элегантные операции на основе кортежей:

C#

```
Console.WriteLine("3. Discrete Operators using tuples (Point with (int, int))");
Console.WriteLine("-----------------------------------------");

Point point = new Point(10, 8);
var offset = (3, -2);
var scale = (2, 3);
var divisor = (2, 4);

Console.WriteLine($"Original point: {point}");
Console.WriteLine($"Offset tuple: {offset}");
Console.WriteLine($"Scale tuple: {scale}");
Console.WriteLine($"Divisor tuple: {divisor}");
Console.WriteLine();

Console.WriteLine($"point + offset: {point + offset}");
Console.WriteLine($"point - offset: {point - offset}");
Console.WriteLine($"point * scale: {point * scale}");
Console.WriteLine($"point / divisor: {point / divisor}");
Console.WriteLine();
```

Расширения могут включать несколько перегруженных операторов, при условии, что операнды различаются.

Перенос методов экземпляра в члены расширения

Члены расширения также поддерживают методы экземпляров. Вам не нужно изменять существующие методы расширения. Старые и новые формы совместимы на уровне двоичного и исходного кода. Если вы хотите сохранить все расширения в одном

контейнере, вы можете. Перенос традиционных методов расширения в новый синтаксис поддерживает те же функции.

Традиционные методы расширения

Традиционный `this` подход использует синтаксис параметра:

C#

```
public static Vector2 ToVector(this Point point) =>
    new Vector2(point.X, point.Y);

public static void Translate(this Point point, int xDist, int yDist)
{
    point.X += xDist;
    point.Y += yDist;
}

public static void Scale(this Point point, int xScale, int yScale)
{
    point.X *= xScale;
    point.Y *= yScale;
}

public static void Rotate(this Point point, int angleInDegress)
{
    double theta = ((double)angleInDegress * Math.PI) / 180.0;
    double sinTheta = Math.Sin(theta);
    double cosTheta = Math.Cos(theta);
    double newX = (double)point.X * cosTheta - (double)point.Y * sinTheta;
    double newY = (double)point.X * sinTheta + (double)point.Y * cosTheta;
    point.X = (int)newX;
    point.Y = (int)newY;
}
```

Члены расширения используют другой синтаксис, но предоставляют те же функции. Добавьте следующий код в новый класс членов расширения:

C#

```
public Vector2 ToVector() =>
    new Vector2(point.X, point.Y);

public void Translate(int xDist, int yDist)
{
    point.X += xDist;
    point.Y += yDist;
}

public void Scale(int xScale, int yScale)
```

```
{
    point.X *= xScale;
    point.Y *= yScale;
}

public void Rotate(int angleInDegrees)
{
    double theta = ((double)angleInDegrees * Math.PI) / 180.0;
    double sinTheta = Math.Sin(theta);
    double cosTheta = Math.Cos(theta);
    double newX = (double)point.X * cosTheta - (double)point.Y * sinTheta;
    double newY = (double)point.X * sinTheta + (double)point.Y * cosTheta;
    point.X = (int)newX;
    point.Y = (int)newY;
}
```

Эти методы расширяют экземпляр структуры `Point`, а не тип `Point`. Блок расширения задаёт имя параметра-получателя, чтобы тело метода могло обращаться к этой точке. В примере используется `extension(ref Point point)`, потому что `Translate`, `Scale` и `Rotate` изменяют `Point` вызывающего объекта. Без `ref` этого эти методы обновят копию структуры, и вызывающий объект не увидит изменения.

Эти новые методы экземпляра можно вызывать так же, как традиционные методы расширения.

C#

```
Console.WriteLine("4. Instance Methods");
Console.WriteLine("------------------");

// ToVector demonstration
Point vectorPoint = new Point(12, 16);
Vector2 vector = vectorPoint.ToVector();
Console.WriteLine($"Point {vectorPoint} as Vector2: {vector}");
Console.WriteLine();

// Translate demonstration
Point translatePoint = new Point(5, 5);
Console.WriteLine($"Before Translate: {translatePoint}");
translatePoint.Translate(3, -2);
Console.WriteLine($"After Translate(3, -2): {translatePoint}");
Console.WriteLine();

// Scale demonstration
Point scalePoint = new Point(4, 6);
Console.WriteLine($"Before Scale: {scalePoint}");
scalePoint.Scale(2, 3);
Console.WriteLine($"After Scale(2, 3): {scalePoint}");
Console.WriteLine();

// Rotate demonstration
```

```
Point rotatePoint1 = new Point(10, 0);
Console.WriteLine($"Before Rotate: {rotatePoint1}");
rotatePoint1.Rotate(90);
Console.WriteLine($"After Rotate(90°): {rotatePoint1}");

Point rotatePoint2 = new Point(5, 5);
Console.WriteLine($"Before Rotate: {rotatePoint2}");
rotatePoint2.Rotate(45);
Console.WriteLine($"After Rotate(45°): {rotatePoint2}");

Point rotatePoint3 = new Point(3, 4);
Console.WriteLine($"Before Rotate: {rotatePoint3}");
rotatePoint3.Rotate(180);
Console.WriteLine($"After Rotate(180°): {rotatePoint3}");
Console.WriteLine();
```

Ключевым отличием является синтаксис: члены расширения используют `extension (Type variableName)` вместо `this Type variableName`.

Добавление индексаторов расширений

C# 15 добавляет индексаторы в `extension` блоки. Индексатор не имеет имени. Код обращается к нему в объявлении с помощью `this[...]`, а в месте вызова — с использованием индексного синтаксиса.

Представьте тип пути, который сохраняет каждый шаг в качестве относительного смещения. Когда вы запрашиваете `path[i]`, вам нужно абсолютное значение точки на этом шаге. Когда вы присваиваете `path[i] = target`, нужно, чтобы тип обновил то единственное смещение, которое приводит к этому результату. Индексный доступ воспринимается как «точка на этом шаге» и сохраняет вычисления преобразования смещения в точку в одном месте.

Представьте, что `Path` поступил из библиотеки. Если вы не являетесь владельцем типа, вы не можете добавить индексатор в его исходный код. До C# 15 можно было добавлять методы, но нельзя было добавить `this[...]` индексированный доступ типу, который вы не можете изменять. В этом руководстве определяется `Path`, чтобы пример можно было запустить; считайте, что он заменяет собой тип из этой библиотеки.

В этом разделе добавьте тип `Path`. Тип хранит последовательность `(dX, dY)` смещений. Каждое смещение указывает, на какое расстояние нужно сместиться от предыдущей точки. Первое смещение начинается с `Point.Origin`, статического свойства расширения, которое вы добавили ранее.

Создайте новый файл с именем `Path.cs` в том же проекте, что и другие примеры файлов. `Path` Добавьте тип `ExtensionMembers` в пространство имен в этом файле. Сохранение `Path` в

этом пространстве имен означает, что именно `Path` будет вашим образцом типа, а не Path. В демофайле используется псевдоним `using Path = ExtensionMembers.Path;`, поэтому каждый `Path` в демофайле означает тип пути sample.

C#

```
public sealed class Path
{
    private readonly List<(int dX, int dY)> offsets = [];

    public Path(IEnumerable<(int dX, int dY)> offsets)
    {
        this.offsets.AddRange(offsets);
    }

    public int Count => offsets.Count;

    internal (int dX, int dY) GetOffset(int index) => offsets[index];

    internal void SetOffset(int index, (int dX, int dY) offset) =>
        offsets[index] = offset;
}
```

Теперь добавьте индексатор для `Path`. Поместите этот код в существующий `PointExtensions` статический класс `NewExtensionsMembers.cs`. Добавьте его как новый блок `extension(Path path)`, отдельно от блока `extension(Point)` для статических членов и операторов и отдельно от блока `extension(ref Point point)` для методов экземпляра:

C#

```
extension(Path path)
{
    public Point this[int index]
    {
        get
        {
            ValidatePathIndex(path, index);

            Point absolutePoint = Point.Origin;
            for (int current = 0; current <= index; current++)
            {
                var offset = path.GetOffset(current);
                absolutePoint += offset;
            }

            return absolutePoint;
        }
        set
        {
            ValidatePathIndex(path, index);
```

```
            Point previousPoint = Point.Origin;
            for (int current = 0; current < index; current++)
            {
                var offset = path.GetOffset(current);
                previousPoint += offset;
            }

            path.SetOffset(index, (value.X - previousPoint.X, value.Y -
previousPoint.Y));
        }
    }
}

private static void ValidatePathIndex(Path path, int index)
{
    if (index < 0 || index >= path.Count)
    {
        throw new ArgumentOutOfRangeException(nameof(index), index,
            "Index must refer to an offset in the path.");
    }
}
```

Индексаторы всегда являются членами экземпляра, поэтому в этом новом блоке расширения необходимо указать получателя: `extension(Path path)`. Блок, записанный как `extension(Path)`, не предоставит переменную `path` для тела индексатора.

`Path` — это класс, содержащий список смещений. Индексатору не нужен получатель `ref`, так как сеттер изменяет содержимое существующего объекта `Path`.

Геттер начинается с `Point.Origin`, а затем добавляет смещения, начиная с индекса `0` и до запрошенного индекса включительно. При смещениях `(2, 3)`, `(1, 1)` и `(-1, 4)` абсолютные точки равны `(2, 3)`, `(3, 4)` и `(2, 8)`.

Метод установки принимает целевую абсолютную точку. Он не затрагивает предыдущие смещения и изменяет только смещение по указанному индексу. Новое смещение — это целевая точка минус абсолютная точка предыдущего индекса. Более поздние точки сдвигаются, так как они остаются относительно измененного смещения.

Теперь используйте индексатор для чтения и записи точек вдоль пути:

C#

```
Console.WriteLine("5. Path Indexer");
Console.WriteLine("---------------");

Path path = new([(dX: 2, dY: 3), (dX: 1, dY: 1), (dX: -1, dY: 4)]);
Console.WriteLine($"First point: {path[0]}");
```

```
Console.WriteLine($"Second point: {path[1]}");
Console.WriteLine($"Third point: {path[2]}");

path[1] = new Point(10, 10);
Console.WriteLine("After setting the second point to {X=10,Y=10}:");
Console.WriteLine($"Second point: {path[1]}");
Console.WriteLine($"Third point: {path[2]}");
Console.WriteLine();
```

Оба аксессора используют общий небольшой закрытый вспомогательный метод `ValidatePathIndex` в том же классе `PointExtensions`, который проверяет, что индекс не выходит за допустимые пределы, и выбрасывает исключение ArgumentOutOfRangeException, если он не указывает на смещение в пути.

Завершенный пример

В последнем примере показаны преимущества при объединении статических свойств, операторов, методов экземпляров и индексатора для создания комплексных расширений типов.

Сравните версию элемента расширения.

C#

```
Console.WriteLine("6. Complex Scenarios");
Console.WriteLine("--------------------");

// Combining operators and methods
Console.WriteLine("Scenario 1: Building a rectangle using operators");
Point topLeft = Point.Origin;
Point bottomRight = topLeft + (10, 8);
Point topRight = new Point(bottomRight.X, topLeft.Y);
Point bottomLeft = new Point(topLeft.X, bottomRight.Y);

Console.WriteLine($"Rectangle corners:");
Console.WriteLine($"  Top-Left: {topLeft}");
Console.WriteLine($"  Top-Right: {topRight}");
Console.WriteLine($"  Bottom-Left: {bottomLeft}");
Console.WriteLine($"  Bottom-Right: {bottomRight}");
Console.WriteLine();

// Transformation chain
Console.WriteLine("Scenario 2: Transformation chain");
Point transformPoint = new Point(2, 3);
Console.WriteLine($"Starting point: {transformPoint}");

// Scale up
transformPoint.Scale(3, 2);
Console.WriteLine($"After scaling by (3, 2): {transformPoint}");
```

```
// Translate
transformPoint = transformPoint + (5, -3);
Console.WriteLine($"After translating by (5, -3): {transformPoint}");

// Rotate
transformPoint.Rotate(45);
Console.WriteLine($"After rotating 45°: {transformPoint}");

// Convert to vector
Vector2 finalVector = transformPoint.ToVector();
Console.WriteLine($"Final result as Vector2: {finalVector}");
Console.WriteLine();

// Distance calculation using operators
Console.WriteLine("Scenario 3: Distance calculation");
Point point1 = new Point(1, 1);
Point point2 = new Point(4, 5);
Point difference = point2 - point1;
Vector2 diffVector = difference.ToVector();
float distance = diffVector.Length();

Console.WriteLine($"Point 1: {point1}");
Console.WriteLine($"Point 2: {point2}");
Console.WriteLine($"Difference: {difference}");
Console.WriteLine($"Distance: {distance:F2}");
Console.WriteLine();

Console.WriteLine("Demonstration complete!");
```

С предыдущей версией:

C#

```
// Combining operators and methods
Console.WriteLine("Scenario 1: Building a rectangle using inline operators");
Point topLeft = Point.Empty; // Inline equivalent of Point.Origin
Point bottomRight = new Point(topLeft.X + 10, topLeft.Y + 8); // Inline addition
Point topRight = new Point(bottomRight.X, topLeft.Y);
Point bottomLeft = new Point(topLeft.X, bottomRight.Y);

Console.WriteLine($"Rectangle corners:");
Console.WriteLine($"  Top-Left: {topLeft}");
Console.WriteLine($"  Top-Right: {topRight}");
Console.WriteLine($"  Bottom-Left: {bottomLeft}");
Console.WriteLine($"  Bottom-Right: {bottomRight}");
Console.WriteLine();

// Transformation chain
Console.WriteLine("Scenario 2: Transformation chain (mixed methods)");
Point transformPoint = new Point(2, 3);
Console.WriteLine($"Starting point: {transformPoint}");
```

```
// Scale up - using extension method
transformPoint.Scale(3, 2);
Console.WriteLine($"After scaling by (3, 2): {transformPoint}");

// Translate - using inline addition
transformPoint = new Point(transformPoint.X + 5, transformPoint.Y + (-3));
Console.WriteLine($"After translating by (5, -3): {transformPoint}");

// Rotate - using extension method
transformPoint.Rotate(45);
Console.WriteLine($"After rotating 45�: {transformPoint}");

// Convert to vector - using extension method
Vector2 finalVector = transformPoint.ToVector();
Console.WriteLine($"Final result as Vector2: {finalVector}");
Console.WriteLine();

// Distance calculation using inline operators and extension methods
Console.WriteLine("Scenario 3: Distance calculation (mixed methods)");
Point point1 = new Point(1, 1);
Point point2 = new Point(4, 5);
Point difference = new Point(point2.X - point1.X, point2.Y - point1.Y); // Inline
subtraction
Vector2 diffVector = difference.ToVector(); // Extension method
float distance = diffVector.Length();

Console.WriteLine($"Point 1: {point1}");
Console.WriteLine($"Point 2: {point2}");
Console.WriteLine($"Difference: {difference}");
Console.WriteLine($"Distance: {distance:F2}");
Console.WriteLine();

Console.WriteLine("Traditional extension methods demonstration complete!");
```

В этом примере показано, как члены расширения создают единый API, который чувствует себя как часть исходного типа. Вы можете:

- Использование `Point.Origin` для понятной начальной точки
- Применяйте математические операторы естественным образом (`point + offset`, `point * scale`)
- Преобразования цепочки с помощью операторов и методов
- Преобразование между связанными типами (`ToVector()`)
- Считывать и обновлять абсолютные точки вдоль пути с помощью `path[index]`

Преимущества миграции

При миграции из традиционных методов расширения в члены расширения вы получите:

1. **Статические свойства**: добавление констант и вычисляемых значений в типы.

2. **Операторы**: включение естественных математических и логических операций.
3. **Индексаторы**: добавление индексированного доступа в C# 15, который может вычислять значения на основе существующего типа и обновлять его сохранённое состояние.
4. **Унифицированный синтаксис: вся логика** расширения использует одну и ту же `extension` декларацию.
5. **Расширения уровня типа**: расширение самого типа, а не только экземпляров.

Запустите полное приложение, чтобы увидеть оба подхода параллельно и наблюдать за тем, как члены расширения обеспечивают более интегрированный интерфейс разработки.

Связанный контент

- Методы расширения (руководство по программированию на C#)
- Новые возможности C# 14
- Новые возможности C# 15
- extension ключевое слово (справочник по C#)
- Спецификация функций индексаторов расширений
- Перегрузка операторов (справочник по C#)

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 21.07.2026

Руководство: Создание операторов составного присваивания

C#14 добавляет *определяемые пользователем операторы составных назначений* , которые позволяют изменять структуру данных вместо создания нового экземпляра. В предыдущих версиях C# выражение:

```
C#
a += b;
```

Расширен до следующего кода:

```
C#
// compiler-generated code prior to C# 13:
var tmp = a + b;
a = tmp;
```

В зависимости от типа `a`, это расширение приводит к чрезмерному выделению для создания новых экземпляров или копирования значений нескольких свойств для задания значений в копии. Добавление определяемого пользователем оператора `+=` указывает, что тип может более эффективно обновлять целевой объект непосредственно.

C# поддерживает существующее расширение, но использует его только в том случае, если недоступен составной определяемый пользователем оператор.

Изучив это руководство, вы:

- ✓ Запустите начальный пример.
- ✓ Определите узкие места в коде.
- ✓ Реализуйте новые операторы составного назначения.
- ✓ Анализ завершенного примера.

Предпосылки

- Пакет SDK для .NET 10. Скачайте его с сайта загрузки .NET.
- Visual Studio 2026. Скачайте его с страницы Visual Studio.

Анализ начального примера

Запустите начальное приложение. Его можно получить из dotnet/docs репозитория GitHub. Пример приложения имитирует отслеживание посещаемости концерта в театре. Модели показывают реалистичные закономерности прибытия в течение вечера, от ранних посетителей до основного наплыва перед началом шоу. Это моделирование демонстрирует выделение объектов при использовании традиционных операторов в сравнении с улучшением эффективности, возможным благодаря определяемым пользователем операторам составного присваивания.

Приложение отслеживает посещаемость через несколько театральных ворот (главный этаж и балконные зоны), когда посетители концерта прибывают. Каждый вход ведёт учёт участников с помощью `GateAttendance` записи. На протяжении всего моделирования код часто обновляет эти счетчики с помощью операций инкремента (`++`) и сложения (`+=`). В следующем коде показана часть этой имитации:

C#

```
// Gate 1 - busiest entrance (target: ~100-130 people)
gates.MainFloorGates[0] += random.Next(8, 15);     // Corporate group
++gates.MainFloorGates[0];                          // Single patron
gates.MainFloorGates[0] += random.Next(20, 30);    // Tour/large group arrival
gates.MainFloorGates[0] += random.Next(5, 12);     // Family groups
++gates.MainFloorGates[0];                          // Solo attendee

// Gate 2 - second busiest (target: ~85-115 people)
gates.MainFloorGates[1] = gates.MainFloorGates[1] + random.Next(6, 12);  // Group
booking
++gates.MainFloorGates[1];                          // Single patron
gates.MainFloorGates[1] += random.Next(18, 28);    // Large family/reunion
gates.MainFloorGates[1] += random.Next(8, 15);     // Corporate/business group
gates.MainFloorGates[1] += random.Next(4, 8);      // Couples/small groups
++gates.MainFloorGates[1];                          // Individual patron
```

Определение узких мест в коде

При использовании традиционных операторов каждая операция создает новый `GateAttendance` экземпляр, что приводит к значительному выделению памяти. Начальный `GateAttendance` компонент *неизменяемый*. Код не может изменить состояние объекта после его инициализации. Для этого решения требуется копирование объектов, если необходимо изменить состояние.

При выполнении имитации подробные выходные данные отображаются:

- Численность посещений по воротам в разные периоды прибытия.
- Общий мониторинг посещаемости через все ворота.
- Окончательный полный отчет со статистикой посещаемости.

В следующем тексте показан пример выходных данных:

```txt
Peak arrival time - all gates busy...

Peak rush period completed - all gates processed heavy traffic.

--- Gate Status After Main Rush (7:15 PM) ---
Main Floor Gates:
  Main-Floor-Gate-1: 145 attendees
  Main-Floor-Gate-2: 168 attendees
  Main-Floor-Gate-3: 149 attendees
  Main-Floor-Gate-4:  71 attendees
  Main Floor Subtotal: 533 attendees

Balcony Gates:
  Balcony-Gate-Left: 164 attendees
  Balcony-Gate-Right: 134 attendees
  Balcony Subtotal: 298 attendees

Total Current Attendance: 831 / 1000

--- Late Arrivals (7:15 PM - 7:30 PM) ---
Final patrons arriving before curtain...

Final arrivals processed - concert about to begin!
```

Изучите начальный `GateAttendance` класс записей:

```C#
public record class GateAttendance(string GateId)
{
    public int Count { get; init; }

    public static GateAttendance operator ++(GateAttendance gate)
    {
        GateAttendance updateGate = gate with { Count = gate.Count + 1 };
        return updateGate;
    }

    public static GateAttendance operator +(GateAttendance gate, int partySize)
    {
        GateAttendance updateGate = gate with { Count = gate.Count + partySize };
        return updateGate;
    }
}
```

В `InitialImplementation.GateAttendance` записи демонстрируется традиционный подход к перегрузке операторов в C#. Обратите внимание, что оператор инкремента (`++`) и

оператор сложения (`+`) создают совершенно новые экземпляры `GateAttendance` с помощью выражения `with`. Каждый раз, когда вы пишете `gate++` или `gate += partySize`, операторы выделяют новый экземпляр записи с обновленным `Count` значением, а затем возвращают этот новый экземпляр. Хотя этот подход поддерживает неизменяемость и безопасность потоков, это происходит за счёт частых выделений памяти. В сценариях с множеством операций, таких как моделирование концерта с сотнями обновлений посещаемости, эти выделения накапливаются быстро, потенциально влияя на производительность и увеличивая давление на сбор мусора.

Чтобы измерить это поведение выделения, попробуйте запустить средство отслеживания выделения объектов .NET в Visual Studio. Когда вы профилируете текущую реализацию во время моделирования концерта, вы обнаружите, что она выделяет 134 `GateAttendance` объектов для завершения относительно небольшого моделирования. Каждый вызов оператора создает новый экземпляр, демонстрируя, как быстро выделение памяти может накапливаться в реальных сценариях. Это измерение обеспечивает конкретные базовые показатели для сравнения улучшений производительности, которые вы достигаете с помощью составных операторов назначения.

Реализуйте составные операторы присваивания

C# 14 представляет операторы составного присваивания, определяемые пользователем, которые позволяют изменение на месте вместо создания новых экземпляров. Эти операторы обеспечивают более эффективную альтернативу традиционному шаблону при сохранении знакомого синтаксиса составных операторов присваивания.

Операторы составных назначений используют новый синтаксис, который объявляет `void` методы возврата с ключевым словом `operator` . Добавьте в класс следующие операторы `GateAttendance` :

C#

```
public void operator +=(int value) => this.property += value;
public void operator ++() => this.property++;
```

Основные отличия от традиционных операторов:

- **Мутация**: они изменяют текущий экземпляр напрямую с помощью `this`.
- **Нет новых экземпляров**: в отличие от традиционных операторов, возвращающих новые объекты, составные операторы изменяют существующие.
- Тип возвращаемого значения: операторы составных присваиваний возвращают , а не сам тип.

Когда компилятор сталкивается с выражениями составных назначений, например `a += b` или `++a`, он следует этому порядку разрешения:

1. **Проверьте наличие оператора составного назначения**: если тип определяет определяемый пользователем оператор составного назначения (например, `+=` или `++`), используйте его напрямую.
2. **Возврат к традиционному расширению**: если составного оператора не существует, развернуть до традиционной формы (`a = a + b`).

Это означает, что оба подхода можно реализовать одновременно. Составные операторы имеют приоритет при наличии, но традиционные операторы служат резервными вариантами для сценариев, где не подходит составное назначение.

Операторы составных назначений предоставляют несколько преимуществ:

- **Сокращение выделений**: Изменение объектов на месте вместо создания новых экземпляров.
- **Улучшенная производительность**: избегание создания временных объектов и снижение нагрузки на сборщик мусора.
- **Знакомый синтаксис**: используйте тот же `+=` синтаксис, `++` разработчики синтаксиса уже знают.
- **Обратная совместимость**: традиционные операторы продолжают работать в качестве резервных вариантов.

Новые операторы составного назначения показаны в следующем коде:

C#

```
public void operator ++() => Count++;

public void operator +=(int partySize) => Count += partySize;
```

> ⓘ **Примечание.**
>
> Разработчики, знакомые с C++ могут задаваться вопросом, почему требуется только один `++` или `--` оператор. Компилятор создает код для использования выражения до или после изменения в качестве возвращаемого значения. Созданный компилятором код выполняет присваивание, используя либо исходное, либо измененное значение, в зависимости от того, был ли применён оператор предварительного (`++x`) или постфиксного увеличения (`x++`).

Анализ готового примера

Теперь, когда вы реализовали операторы составного присваивания, пришло время оценить улучшение производительности. Чтобы измерить резкое различие в выделении памяти, снова запустите средство отслеживания выделения объектов .NET в обновленном коде.

При профилировании приложения с включенными операторами составных присваиваний наблюдается замечательное сокращение: во время всего моделирования концерта выделяется всего **10 `GateAttendance` объектов** по сравнению с предыдущими 134 выделениями. Это обновление представляет собой 92% сокращение выделения объектов!

Оставшиеся 10 выделений поступают из первоначального создания `GateAttendance` экземпляров для каждого из театральных ворот (четыре главных ворот + два балконных ворот = шесть начальных экземпляров), а также из нескольких дополнительных выделений из других частей имитации, которые не используют составные операторы.

Это сокращение распределения приводит к реальным преимуществам производительности:

- **Снижение нагрузки на память**: менее частые циклы сборки мусора.
- **Лучшее расположение кэша. Меньшее** количество создания объектов означает меньше фрагментации памяти.
- **Улучшенная пропускная способность**: экономия циклов ЦП благодаря уменьшению накладных расходов при распределении и сборке.
- **Масштабируемость**. Преимущества умножаются в сценариях с большими объемами операций.

Повышение производительности становится еще более значительным в рабочих приложениях, где аналогичные шаблоны происходят в гораздо больших масштабах—представьте, что отслеживание миллионов транзакций, обновление тысяч счетчиков или обработка потоков данных высокой частоты.

Попробуйте найти другие возможности для использования составных операторов присваивания в кодовой базе. Найдите шаблоны, в которых используются традиционные операции назначения, например `gates.MainFloorGates[1] = gates.MainFloorGates[1] + 4` и рассмотрите, могут ли они воспользоваться синтаксисом составного назначения. Хотя некоторые из этих операций уже используются `+=` в коде моделирования, принцип применяется к любому сценарию, в котором многократно изменяются объекты, а не создают новые экземпляры.

В качестве окончательного эксперимента измените тип `GateAttendance` с `record class` на `record struct`. Это другая оптимизация, и она работает в этой симуляции, так как структура имеет небольшой памятный след. Копирование `GateAttendance` структуры не является дорогой операцией. Даже поэтому вы достигаете небольших улучшений.

Связанный контент

- Новые возможности C# 14
- Перегрузка оператора — определение унарных, арифметических, операторов равенства и сравнения
- Анализ использования памяти с помощью средства выделения объектов .NET — Visual Studio

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 14.11.2025

Объявление основных конструкторов для классов и структур

Статья • 18.03.2025

В C# 12 представлены первичных конструкторов, которые предоставляют краткий синтаксис для объявления конструкторов, параметры которых доступны в любом месте текста типа.

В этой статье описывается, как объявить главный конструктор на вашем типе и определить, где хранить параметры главного конструктора. Вы можете вызывать первичные конструкторы из других конструкторов и использовать параметры первичного конструктора в членах типа.

Предпосылки

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Общие сведения о правилах для основных конструкторов

Можно добавить параметры в объявление `struct` или `class`, чтобы создать *основной конструктор*. Основные параметры конструктора находятся в области определения класса. Важно рассматривать основные параметры конструктора как *параметры*, даже если они доступны во всей области определения класса.

Несколько правил определяют, что эти конструкторы являются параметрами:

- Параметры первичного конструктора могут не храниться, если они не нужны.
- Основные параметры конструктора не являются членами класса. Например, к основному параметру конструктора с именем `param` не удается получить доступ как к `this.param`.
- Параметрам основного конструктора можно присваивать значения.
- В основных конструкторах параметры не становятся свойствами, за исключением типов записей .

Эти правила являются теми же правилами, которые уже определены для параметров любого метода, включая другие объявления конструктора.

Ниже приведены наиболее распространенные способы использования для основного параметра конструктора:

- Передать в качестве аргумента для вызова конструктора `base()`
- Инициализация поля или свойства элемента
- Ссылка на параметр конструктора в элементе экземпляра

Каждый другой конструктор для класса **должен** вызывать основной конструктор напрямую или косвенно через вызов конструктора `this()`. Это правило гарантирует, что основные параметры конструктора назначаются везде в тексте типа.

Инициализация неизменяемых свойств или полей

Следующий код инициализирует два свойства, доступные только для чтения (неизменяемые), которые вычисляются из основных параметров конструктора.

C#

```
public readonly struct Distance(double dx, double dy)
{
    public readonly double Magnitude { get; } = Math.Sqrt(dx * dx + dy *
dy);
    public readonly double Direction { get; } = Math.Atan2(dy, dx);
}
```

В этом примере используется основной конструктор для инициализации вычисляемых свойств только для чтения. Инициализаторы полей для свойств `Magnitude` и `Direction` используют основные параметры конструктора. Параметры первичного конструктора не используются нигде в структуре. Код создает структуру, как если бы она была написана следующим образом:

C#

```
public readonly struct Distance
{
    public readonly double Magnitude { get; }

    public readonly double Direction { get; }

    public Distance(double dx, double dy)
    {
        Magnitude = Math.Sqrt(dx * dx + dy * dy);
        Direction = Math.Atan2(dy, dx);
```

```
    }
}
```

Эта функция упрощает использование инициализаторов полей при необходимости аргументов для инициализации поля или свойства.

Создание изменяемого состояния

В предыдущих примерах используются параметры первичного конструктора для инициализации свойств, доступных только для чтения. Вы также можете использовать первичные конструкторы для свойств, которые не читаются.

Рассмотрим следующий код:

C#

```
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

В этом примере метод `Translate` изменяет компоненты `dx` и `dy`, что требует вычисления свойств `Magnitude` и `Direction` при их доступе. Оператор больше или равно (`=>`) обозначает аксессор с телом-выражением `get`, тогда как оператор равный (`=`) обозначает инициализатор.

Эта версия кода добавляет в структуру конструктор без параметров. Конструктор без параметров должен вызывать первичный конструктор, который гарантирует инициализацию всех параметров первичного конструктора. К основным свойствам конструктора обращаются в методе, и компилятор создает скрытые поля для представления каждого параметра.

Следующий код демонстрирует приближение того, что создает компилятор. Фактические имена полей являются допустимыми идентификаторами CIL, но недопустимыми идентификаторами C#.

C#

```
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
__unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
    }

    public Distance(double dx, double dy)
    {
        __unspeakable_dx = dx;
        __unspeakable_dy = dy;
    }
    public Distance() : this(0, 0) { }
}
```

Созданное компилятором хранилище

В первом примере этого раздела компилятору не нужно создавать поле для хранения значения параметров первичного конструктора. Однако во втором примере основной параметр конструктора используется внутри метода, поэтому компилятор должен создать хранилище для параметров.

Компилятор создает хранилище для любых основных конструкторов только тогда, когда параметр используется в теле члена вашего типа. В противном случае параметры первичного конструктора не хранятся в объекте.

Используйте внедрение зависимостей

Другим распространенным способом использования первичных конструкторов является указание параметров внедрения зависимостей. Следующий код создает простой контроллер, требующий интерфейса службы для его использования:

C#

```
public interface IService
{
```

```
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
```

Основной конструктор четко указывает параметры, необходимые для класса. Вы используете параметры первичного конструктора, как и любую другую переменную в классе.

Инициализация базового класса

Вы можете вызвать основной конструктор базового класса из основного конструктора производного класса. Этот подход является самым простым способом написания производного класса, который должен вызывать первичный конструктор в базовом классе. Рассмотрим иерархию классов, представляющих различные типы счетов в качестве банка. В следующем коде показано, что может выглядеть базовый класс:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";
}
```

Все банковские счета, независимо от типа, имеют свойства для номера счета и владельца. В завершенном приложении можно добавить другие распространенные функциональные возможности в базовый класс.

Для многих типов требуется более конкретная проверка параметров конструктора. Например, класс `BankAccount` имеет определенные требования к параметрам `owner` и `accountID`. Параметр `owner` не должен быть `null` или пробелами, а параметр `accountID` должен быть строкой, содержащей 10 цифр. Эту проверку можно добавить при назначении соответствующих свойств:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number",
nameof(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
nameof(owner))
        : owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";

    public static bool ValidAccountNumber(string accountID) =>
    accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

В этом примере показано, как проверить параметры конструктора перед их назначением свойствам. Вы можете использовать встроенные методы, такие как String.IsNullOrWhiteSpace(String) или собственный метод проверки, например `ValidAccountNumber`. В этом примере все исключения выбрасываются из конструктора при вызове инициализаторов. Если для назначения поля не используется параметр конструктора, при первом доступе к параметру конструктора возникают исключения.

Один производный класс может представлять расчетный счет:

C#

```
public class CheckingAccount(string accountID, string owner, decimal
overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
```

```
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}, Balance: {CurrentBalance}";
}
```

Производный `CheckingAccount` класс имеет основной конструктор, который принимает все параметры, необходимые в базовом классе, и другой параметр со значением по умолчанию. Основной конструктор вызывает базовый конструктор с синтаксисом `: BankAccount(accountID, owner)`. Это выражение указывает тип базового класса и аргументы основного конструктора.

Ваш производный класс не обязуется использовать основной конструктор. Конструктор можно создать в производном классе, который вызывает основной конструктор базового класса, как показано в следующем примере:

C#

```
public class LineOfCreditAccount : BankAccount
{
    private readonly decimal _creditLimit;
    public LineOfCreditAccount(string accountID, string owner, decimal
creditLimit) : base(accountID, owner)
    {
        _creditLimit = creditLimit;
    }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
```

```
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -_creditLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"{base.ToString()}, Balance:
{CurrentBalance}";
}
```

Существует одна потенциальная проблема с иерархиями классов и основными конструкторами. Можно создать несколько копий первичного параметра конструктора, так как этот параметр используется как в производных, так и в базовых классах. Следующий код создает две копии каждого из параметров `owner` и `accountID`:

C#

```
public class SavingsAccount(string accountID, string owner, decimal
interestRate) : BankAccount(accountID, owner)
{
    public SavingsAccount() : this("default", "default", 0.01m) { }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < 0)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
```

```
        }
        CurrentBalance -= amount;
    }

    public void ApplyInterest()
    {
        CurrentBalance *= 1 + interestRate;
    }

    public override string ToString() => $"Account ID: {accountID}, Owner:
{owner}, Balance: {CurrentBalance}";
}
```

Выделенная строка в этом примере показывает, что метод `ToString` использует *параметры основного конструктора* (`owner` и `accountID`) вместо *свойств базового класса* (`Owner` и `AccountID`). Результатом является то, что производный класс, `SavingsAccount`, создает хранилище для копирования параметров. Копия в производном классе отличается от свойства в базовом классе. Если свойство базового класса можно изменить, экземпляр производного класса не видит изменения. Компилятор выдает предупреждение для основных параметров конструктора, которые используются в производном классе и передаются конструктору базового класса. В этом случае исправление заключается в использовании свойств базового класса.

Связанный контент

- конструкторы без параметров (руководство по программированию на C#)
- первичные конструкторы (предложение спецификации функционала)

Учебное пособие: Исследуйте идеи, используя операторы верхнего уровня для создания кода по мере обучения

Статья • 20.12.2024

В этом руководстве описано, как:

- ✔ Изучите правила, регулирующие использование операторов верхнего уровня.
- ✔ Используйте инструкции верхнего уровня для изучения алгоритмов.
- ✔ Преобразование исследований в компоненты для повторного использования.

Необходимые условия

Необходимо настроить компьютер для запуска .NET 6 или более поздней версии. Компилятор C# доступен начиная с Visual Studio 2022 или пакета SDK для .NET.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Начало изучения

Инструкции топ-уровня позволяют избежать дополнительной обработки, которая возникает при необходимости размещения точки входа программы в статический метод в классе. Типичный начальный пункт для нового консольного приложения выглядит следующим образом:

C#

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
```

```
    }
}
```

Приведенный выше код является результатом выполнения команды `dotnet new console` и создания консольного приложения. Эти 11 строки содержат только одну строку исполняемого кода. Эту программу можно упростить с помощью новой функции инструкций верхнего уровня. Это позволяет удалить все, кроме двух строк в этой программе:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

> ⓘ **Важно!**
>
> Шаблоны C# для .NET 6 используют инструкции верхнего уровня . Приложение может не соответствовать коду в этой статье, если вы уже обновили его до .NET 6. Дополнительные сведения см. в статье о **новых шаблонах C#, которые создают инструкции на верхнем уровне**
>
> Пакет SDK для .NET 6 также добавляет набор неявных директив для проектов, использующих следующие пакеты SDK:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Эти неявные директивы `global using` включают наиболее распространенные пространства имен для типа проекта.
>
> Дополнительные сведения см. в статье о **неявных директив using**

Эта функция упрощает изучение новых идей. Инструкции верхнего уровня можно использовать для сценариев скриптов или для исследований. После работы с основами можно начать рефакторинг кода и создать методы, классы или другие сборки для созданных повторно используемых компонентов. Инструкции верхнего уровня позволяют быстро экспериментировать и учебные пособия для начинающих. Они также обеспечивают гладкий путь от экспериментирования до полных программ.

Инструкции верхнего уровня выполняются в том порядке, в котором они отображаются в файле. Инструкции верхнего уровня можно использовать только в одном исходном файле в приложении. Компилятор создает ошибку, если они используются в нескольких файлах.

Создание волшебного компьютера ответов .NET

В этом руководстве мы создадим консольное приложение, которое отвечает на вопрос "да" или "нет" с случайным ответом. Вы строите функциональность шаг за шагом. Вы можете сосредоточиться на задаче, а не на церемонии, необходимой для структуры типичной программы. Затем, когда вы удовлетворены функциональностью, вы можете отрефакторить приложение по своему усмотрению.

Хорошая отправная точка — написать вопрос обратно в консоль. Для начала можно написать следующий код:

C#

```
Console.WriteLine(args);
```

Вы не объявляете переменную `args`. Для одного исходного файла, содержащего инструкции верхнего уровня, компилятор распознает `args`, чтобы означать аргументы командной строки. Тип args — это `string[]`, как и во всех программах C#.

Чтобы протестировать код, выполните следующую команду `dotnet run`:

Интерфейс командной строки.NET

```
dotnet run -- Should I use top level statements in all my programs?
```

Аргументы после `--` в командной строке передаются программе. Вы можете увидеть тип переменной `args`, потому что эта информация выводится в консоль.

Консоль

```
System.String[]
```

Чтобы написать вопрос в консоль, необходимо перечислить аргументы и разделить их пробелом. Замените вызов `WriteLine` следующим кодом:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Теперь при запуске программы он правильно отображает вопрос в виде строки аргументов.

Ответьте случайным образом

После того как вы повторите вопрос, можно добавить код для создания случайного ответа. Начните с добавления массива возможных ответов:

C#

```
string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

Этот массив содержит 10 утвердительных ответов, пять нейтральных и пять отрицательных. Затем добавьте следующий код, чтобы создать и отобразить случайный ответ из массива:

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Вы можете снова запустить приложение, чтобы просмотреть результаты. Вы должны увидеть примерно следующее выходные данные:

Интерфейс командной строки.NET

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

Этот код отвечает на вопросы, но добавим еще одну функцию. Вы хотите, чтобы ваше приложение вопросов имитировало мышление об ответе. Это можно сделать, добавив немного анимации ASCII и делая паузы во время работы. Добавьте следующий код после строки, которая отражает вопрос:

C#

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

Кроме того, необходимо добавить директиву `using` в начало исходного файла:

C#

```
using System.Threading.Tasks;
```

Директивы `using` должны быть перед любыми другими инструкциями в файле. В противном случае это ошибка компилятора. Вы можете снова запустить программу

и увидеть анимацию. Это улучшает опыт. Поэкспериментируйте с длиной задержки, чтобы соответствовать вашему вкусу.

Предыдущий код создает набор вращающихся линий, отделенных друг от друга пробелами. Добавление ключевого слова `await` указывает компилятору создать точку входа программы в качестве метода с модификатором `async` и возвращает System.Threading.Tasks.Task. Эта программа не возвращает значение, поэтому точка входа программы возвращает `Task`. Если программа возвращает целочисленное значение, вы добавите оператор return в конец инструкций верхнего уровня. Этот оператор возврата укажет целочисленное значение, которое следует вернуть. Если в операторах верхнего уровня присутствует выражение `await`, возвращаемый тип становится System.Threading.Tasks.Task<TResult>.

Рефакторинг для будущего

Программа должна выглядеть следующим образом:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
```

```
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Приведенный выше код является разумным. Оно работает. Но это не допускает повторного использования. Теперь, когда приложение у вас работает, пришло время извлечь повторно используемые части.

Одним из кандидатов является код, отображающий анимацию ожидания. Этот фрагмент кода может стать методом:

Сначала можно создать локальную функцию в файле. Замените текущую анимацию следующим кодом:

C#

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

Предыдущий код создает локальную функцию внутри основного метода. Этот код по-прежнему не используется повторно. Извлеките этот код в класс. Создайте файл с именем *utilities.cs* и добавьте следующий код:

C#

```
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

Файл с операторами верхнего уровня также может содержать пространства имен и типы в конце файла после инструкций верхнего уровня. Но для этого руководства вы помещаете метод анимации в отдельный файл, чтобы сделать его более легко используемым повторно.

Наконец, можно очистить код анимации, чтобы удалить некоторые дублирования, используя цикл `foreach` для итерации с помощью набора элементов анимации, определенных в массиве `animations`.
Полный метод `ShowConsoleAnimation` после рефакторинга должен выглядеть следующим образом:

C#

```
public static async Task ShowConsoleAnimation()
{
    string[] animations = ["| -", "/ \\", "- |", "\\ /"];
    for (int i = 0; i < 20; i++)
```

```
    {
        foreach (string s in animations)
        {
            Console.Write(s);
            await Task.Delay(50);
            Console.Write("\b\b\b");
        }
    }
    Console.WriteLine();
}
```

Теперь у вас есть полное приложение, и вы рефакторизовали повторно используемые части для последующего использования. Вы можете вызвать новый метод служебной программы из инструкций верхнего уровня, как показано в готовой версии основной программы:

C#

```
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

В предыдущем примере добавляется вызов `Utilities.ShowConsoleAnimation` и добавляется ещё одна директива `using`.

Сводка

Инструкции верхнего уровня упрощают создание простых программ для изучения новых алгоритмов. Вы можете поэкспериментировать с алгоритмами, пробуя различные фрагменты кода. После того как вы узнали, что работает, вы можете рефакторить код, чтобы он стал более поддерживаемым.

Инструкции верхнего уровня упрощают программы, основанные на консольных приложениях. К этим приложениям относятся функции Azure, действия GitHub и другие небольшие служебные программы. Дополнительные сведения см. в инструкций верхнего уровня (руководство по программированию на C#).

Индексы и диапазоны

Статья • 14.11.2023

Диапазоны и индексы обеспечивают лаконичный синтаксис для доступа к отдельным элементам или диапазонам в последовательности.

Из этого руководства вы узнаете, как выполнять следующие задачи:

- ✓ Использовать этот синтаксис для диапазонов в последовательности.
- ✓ Неявно определяет объект Range.
- ✓ Проектировать начало и конец каждой последовательности.
- ✓ Составлять сценарии для типов Index и Range.

Поддержка языков для индексов и диапазонов

Диапазоны и индексы обеспечивают лаконичный синтаксис для доступа к отдельным элементам или диапазонам в последовательности.

Поддержка языков опирается на два новых типа и два новых оператора:

- System.Index представляет индекс в последовательности.
- Индекс из конечного оператора `^`, указывающий, что индекс относительно конца последовательности.
- System.Range представляет вложенный диапазон последовательности.
- Оператор `..`диапазона, указывающий начало и конец диапазона в качестве операндов.

Начнем с правил для использования в индексах. Рассмотрим массив `sequence`. Индекс `0` совпадает с `sequence[0]`. Индекс `^0` совпадает с `sequence[sequence.Length]`. Выражение `sequence[^0]` создает исключение так же, как и `sequence[sequence.Length]`. Для любого числа `n` индекс `^n` совпадает с `sequence.Length - n`.

C#

```
string[] words = [
                 // index from start    index from end
    "The",       // 0                   ^9
    "quick",     // 1                   ^8
    "brown",     // 2                   ^7
    "fox",       // 3                   ^6
```

```
    "jumps",     // 4                  ^5
    "over",      // 5                  ^4
    "the",       // 6                  ^3
    "lazy",      // 7                  ^2
    "dog"        // 8                  ^1
];               // 9 (or words.Length) ^0
```

Последнее слово можно получить с помощью индекса `^1`. Добавьте следующий код после инициализации:

C#

```
Console.WriteLine($"The last word is {words[^1]}");
```

Диапазон указывает *начало* и *конец* диапазона. Начало диапазона является включающим, но конец диапазона является исключающим, то есть *начало* включается в диапазон, а *конец* не включается. Диапазон `[0..^0]` представляет весь диапазон так же, как `[0..sequence.Length]` представляет весь диапазон.

Следующий код создает поддиапазон со словами "quick", "brown" и "fox". Он включает в себя элементы от `words[1]` до `words[3]`. Элемент `words[4]` в диапазон не входит.

C#

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Следующий код возвращает диапазон со словами "lazy" и "dog". Он включает элементы `words[^2]` и `words[^1]`. Конечный индекс `words[^0]` не включен. Добавьте также следующий код:

C#

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

В следующих примерах создаются диапазоны, которые должны быть открыты для начала, конца или в обоих случаях:

C#

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Диапазоны или индексы можно также объявлять как переменные. Впоследствии такую переменную можно использовать внутри символов `[` и `]`:

C#

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Следующий пример демонстрирует многие возможные причины для таких решений. Измените `x`, `y` и `z` и опробуйте различные комбинации. Поэкспериментируйте со значениями, у которых `x` меньше `y`, а `y` меньше, чем `z` для допустимых сочетаний. Добавьте в новый метод приведенный ниже код. Опробуйте различные комбинации:

C#

```
int[] numbers = [..Enumerable.Range(0, 100)];
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length -
x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and
disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},
numbers[y..z] is {y_z[0]} through {y_z[^1]}");
```

```
Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with
{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");
```

Индексы и диапазоны поддерживаются не только массивами. Можно также использовать индексы и диапазоны со строкой (Span<T> или ReadOnlySpan<T>).

Преобразования выражений операторов неявного диапазона

При использовании синтаксиса выражения оператора диапазона компилятор неявно преобразует начальные и конечные значения в и Index из них создает новый Range экземпляр. В следующем коде показан пример неявного преобразования из синтаксиса выражения оператора диапазона и соответствующего явного альтернативного варианта:

C#

```
Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//     The implicit range '3..^5' equals the explicit range '3..^5'
```

ⓘ Важно!

Неявные преобразования, отбрасываемые в **Int32Index** случае отрицательного **ArgumentOutOfRangeException** значения. Аналогичным образом `Index` конструктор создает `ArgumentOutOfRangeException` исключение при отрицательном значении `value` параметра.

Поддержка типов для индексов и диапазонов

Индексы и диапазоны предоставляют четкий и краткий синтаксис для доступа к одному элементу или диапазону элементов последовательности. Выражение индекса обычно возвращает тип элементов последовательности. Выражение диапазона обычно возвращает тот же тип последовательности, что и исходная последовательность.

Любой тип, который предоставляет индексатор с параметром Index или Range, явно поддерживает индексы или диапазоны соответственно. Индексатор, который принимает один параметр Range, может возвращать другой тип последовательности, например System.Span<T>.

ⓘ Важно!

Производительность кода, использующего оператор range, зависит от типа операнда последовательности.

Временная сложность оператора range зависит от типа последовательности. Например, если последовательность является `string` или массивом, то результатом будет копия указанного раздела входных данных, поэтому временная сложность имеет значение *O(N)* (где N — это длина диапазона). С другой стороны, если это **System.Span<T>** или **System.Memory<T>**, результат ссылается на то же резервное хранилище, что означает отсутствие копии и использование операции *O(1)*.

Помимо временной сложности, создаются дополнительные выделения и копии, влияющие на производительность. В коде, чувствительном к производительности, рекомендуется использовать `Span<T>` или `Memory<T>` в качестве типа последовательности, так как оператор range не предоставляет им выделений.

Тип является **счетным**, если у него есть свойство с именем `Length` или `Count` с доступным методом получения и типом возвращаемого значения `int`. Счетный тип, который не поддерживает индексы или диапазоны явным образом, может предоставить неявную поддержку для них. Дополнительные сведения см. в разделах о поддержке неявного индекса и неявного диапазона в примечании к предлагаемой функции. Диапазоны с поддержкой неявных диапазонов возвращают тот же тип последовательности, что и исходная.

Например, следующие типы .NET поддерживают как индексы, так и диапазоны: String, Span<T> и ReadOnlySpan<T>. List<T> поддерживает индексы, но не поддерживает диапазоны.

Array реализует более сложное поведение. Одномерные массивы поддерживают как индексы, так и диапазоны. Многомерные массивы не поддерживают индексаторы или диапазоны. Индексатор для многомерного массива имеет не один, а несколько параметров. Массивы массивов также поддерживают и диапазоны и индексаторы. В следующем примере показано выполнение итерации для прямоугольного подраздела массива массивов. В этом примере выполняется итерация раздела в центре с исключением трех первых и последних строк, а также двух первых и последних столбцов из каждой выбранной строки:

C#

```
int[][] jagged =
[
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
   [10,11,12,13,14,15,16,17,18,19],
   [20,21,22,23,24,25,26,27,28,29],
   [30,31,32,33,34,35,36,37,38,39],
   [40,41,42,43,44,45,46,47,48,49],
   [50,51,52,53,54,55,56,57,58,59],
   [60,61,62,63,64,65,66,67,68,69],
   [70,71,72,73,74,75,76,77,78,79],
   [80,81,82,83,84,85,86,87,88,89],
   [90,91,92,93,94,95,96,97,98,99],
];

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
```

```
    Console.WriteLine();
}
```

Во всех случаях оператор range для Array выделяет массив для хранения возвращаемых элементов.

Сценарии для индексов и диапазонов

Часто диапазоны и индексы используются для анализа отдельной части последовательности большего размера. В новом синтаксисе легче понять, о какой части последовательности идет речь. Локальная функция `MovingAverage` принимает Range в качестве аргумента. После этого метод перечисляет только это диапазон и вычисляет минимальное, максимальное и среднее значение. Попробуйте добавить в свой проект следующий код:

C#

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range
range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) => [..Enumerable.Range(0, count).Select(x => (int)
(Math.Sqrt(x) * 100))];
```

Примечание по индексам диапазона и массивам

При получении диапазона из массива результатом является массив, скопированный из исходного массива, а не ссылающийся. Изменение значений в результирующем массиве не изменит значения в исходном массиве.

Например:

C#

```
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] =  11; // now contains 11,2,3

Console.WriteLine(string.Join(",", firstThreeItems));
Console.WriteLine(string.Join(",", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

См. также

- Операторы и выражения доступа к членам

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Открыть проблему с документацией

Отзыв о продукте

Использование сопоставления шаблонов для создания поведения класса для улучшения кода

Статья • 19.12.2024

Функции сопоставления шаблонов в C# предоставляют синтаксис для выражения алгоритмов. Эти методы можно использовать для реализации поведения в классах. Вы можете объединить объектно-ориентированное проектирование классов с реализацией, ориентированной на данные, чтобы обеспечить краткий код при моделировании реальных объектов.

В этом руководстве описано, как:

- ✓ Экспрессируйте объектно-ориентированные классы с помощью шаблонов данных.
- ✓ Реализуйте эти шаблоны с помощью функций сопоставления шаблонов C#.
- ✓ Используйте диагностику компилятора для проверки реализации.

Необходимые условия

Необходимо настроить компьютер для запуска .NET. Скачайте Visual Studio 2022 или пакет SDK для .NET.

Создание имитации блокировки канала

В этом руководстве вы создадите класс C#, который имитирует шлюз . Вкратце, шлюз — это устройство, которое поднимает и опускает лодки, когда они перемещаются между двумя участками водоема на разных уровнях. Шлюз содержит два ворота и механизм для изменения уровня воды.

В обычном режиме лодка входит в одни из ворот, в то время как уровень воды в шлюзе соответствует уровню воды со стороны, с которой входит лодка. После блокировки уровень воды изменяется на соответствие уровню воды, где лодка покидает замок. Когда уровень воды соответствует этой стороне, ворота на стороне выхода открываются. Меры безопасности обеспечивают того, что оператор не может создать опасную ситуацию в канале. Уровень воды можно изменить только при закрытии обоих ворот. По крайней мере один ворот можно открыть. Чтобы открыть ворота, уровень воды в замке должен соответствовать уровню воды за пределами ворот, открываемых.

Класс C# можно создать для моделирования этого поведения. Класс `CanalLock` поддерживает команды для открытия или закрытия ворот. У нее будут другие команды, чтобы поднять или опустить воду. Класс также должен поддерживать свойства для чтения текущего состояния как затворов, так и уровня воды. Методы реализуют меры безопасности.

Определение класса

Вы создаете консольное приложение для тестирования класса `CanalLock`. Создайте проект консоли для .NET 5 с помощью Visual Studio или .NET CLI. Затем добавьте новый класс и назовите его `CanalLock`. Затем создайте общедоступный API, но оставьте методы не реализованы:

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
```

```
        $"The water level is {CanalLockWaterLevel}.";
}
```

Предыдущий код инициализирует объект, поэтому оба шлюза закрыты, а уровень воды низкий. Затем напишите следующий тестовый код в методе `Main`, чтобы помочь вам при создании первой реализации класса:

C#

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Затем добавьте первую реализацию каждого метода в классе `CanalLock`.

Следующий код реализует методы класса без проблем с правилами безопасности. Позже вы добавите тесты безопасности:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

Тесты, которые вы написали до сих пор, проходят. Вы реализовали основы. Теперь напишите тест для первого условия сбоя. В конце предыдущих тестов оба ворота закрыты, и уровень воды установлен на низкий. Добавьте тест, чтобы попытаться открыть верхние ворота:

C#

```
Console.WriteLine("=============================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is
low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Этот тест завершается ошибкой, так как открывается шлюз. В качестве первой реализации его можно исправить с помощью следующего кода:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
```

```
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}
```

Ваши тесты успешно пройдены. Однако с добавлением новых тестов вы добавляете больше предложений `if` и тестируете различные свойства. Вскоре эти методы будут слишком сложными при добавлении дополнительных условных условий.

Реализация команд с помощью шаблонов

Лучше использовать шаблоны , чтобы определить, находится ли объект в допустимом состоянии для выполнения команды. Можно выразить, разрешена ли команда как функцию трёх переменных: состояния шлюза, уровня воды и новой настройки.

Развернуть таблицу

Новый параметр	Состояние шлюза	Уровень воды	Результат
Закрытый	Закрытый	Высокий	Закрытый
Закрытый	Закрытый	Низкий	Закрытый
Закрытый	Откройте	Высокий	Закрытый
~~закрыто~~	~~Открыть~~	~~низкий~~	~~Закрыто~~
Откройте	Закрытый	Высокий	Открыть
Открыть	Закрытый	Низкий	Закрыто (ошибка)
Открыть	Откройте	Высокий	Открыть
~~Открыть~~	~~Открыть~~	~~низкий~~	~~закрыт (ошибка)~~

Четвертая и последняя строки в таблице содержат зачеркнутый текст, поскольку они являются недопустимыми. Добавленный код теперь должен убедиться, что ворота с высокой водой никогда не открываются, когда вода низкая. Эти состояния можно закодировать в виде одного выражения коммутатора (помните, что `false` означает "Закрыто"):

```
C#
```

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Попробуйте использовать эту версию. Ваши тесты проходят, что подтверждает правильность кода. В полной таблице показаны возможные сочетания входных данных и результатов. Это означает, что вы и другие разработчики могут быстро посмотреть на таблицу и увидеть, что вы рассмотрели все возможные входные данные. Еще проще, компилятор также может помочь. После добавления предыдущего кода можно увидеть, что компилятор создает предупреждение: *CS8524* указывает, что выражение коммутатора не охватывает все возможные входные данные. Причина этого предупреждения заключается в том, что один из входных данных является типом `enum`. Компилятор интерпретирует "все возможные входные данные" как все входные данные из базового типа, как правило, `int`. Это выражение `switch` проверяет только значения, объявленные в `enum`. Чтобы удалить предупреждение, можно добавить универсальный шаблон для последней ветви выражения. Это условие создает исключение, так как указывает на недопустимые входные данные:

C#

```
_  => throw new InvalidOperationException("Invalid internal state"),
```

Предшествующее плечо переключателя должно идти последним в вашем выражении `switch`, поскольку оно соответствует всем входным данным. Поэкспериментируйте, переместив его в более раннюю позицию в порядке. Это приводит к ошибке компилятора *CS8510* для недоступного кода в шаблоне. Естественная структура выражений коммутатора позволяет компилятору создавать ошибки и предупреждения о возможных ошибках. Компилятор, выступающий в роли "предохранительной сети", упрощает создание правильного кода в меньшем количестве итераций и предоставляет возможность объединения ветвей переключения с подстановочными знаками. Компилятор выдает ошибки, если сочетание приводит к недостижимым ветвям, которых вы не ожидали, и предупреждения, если вы удалите необходимую ветвь.

Первое изменение заключается в том, чтобы объединить все рычаги, где команда - закрыть ворота; это всегда допустимо. Добавьте следующий код в качестве первой ветви в выражении switch.

C#

```
(false, _, _) => false,
```

После добавления предыдущего переключающего рычага вы получите четыре ошибки компилятора, по одной на каждом из плеч, где команда `false`. Эти руки уже покрыты только что добавленной рукой. Вы можете безопасно удалить эти четыре строки. Вы намеревались заменить эти условия новым переключателем.

Затем можно упростить четыре руки, где команда — открыть ворота. В обоих случаях, когда уровень воды высок, ворота можно открыть. (Один из них уже открыт.) Один из случаев, когда уровень воды низкий вызывает исключение, и другой не должен произойти. Это должно быть безопасно — выбросить то же исключение, если водяная блокировка уже находится в недопустимом состоянии. Вы можете сделать следующие упрощения для этого оружия:

C#

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Снова запустите тесты, и они проходят. Ниже приведена окончательная версия метода `SetHighGate`:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
switch
    {
        (false, _,     _)               => false,
        (true, _,      WaterLevel.High) => true,
        (true, false, WaterLevel.Low)  => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _                               => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

Реализуйте шаблоны самостоятельно

Теперь, когда вы увидели технику, заполните методы `SetLowGate` и `SetWaterLevel` самостоятельно. Начните с добавления следующего кода для тестирования недопустимых операций с этими методами:

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
```

```
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Снова запустите приложение. Вы увидите, что новые тесты завершаются сбоем, и блокировка канала попадает в недопустимое состояние. Попробуйте реализовать оставшиеся методы самостоятельно. Метод установки нижних врат должен быть аналогичен методу установки верхних врат. Метод, который изменяет уровень воды, имеет различные проверки, но должен соответствовать аналогичной структуре. Возможно, полезно использовать тот же процесс для метода, который задает уровень воды. Начните со всех четырех входных данных: состояние обоих ворот, текущее состояние уровня воды и запрошенный новый уровень воды. Выражение переключателя должно начинаться с:

C#

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};
```

У вас 16 рычагов переключения, которые нужно заполнить. Затем протестируйте и упростите.

Вы создали методы примерно таким образом?

C#

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open low gate when the water is high"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
```

```
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Ваши тесты должны успешно завершиться, и канальный шлюз должен безопасно работать.

Сводка

В этом руководстве вы узнали, как использовать сопоставление шаблонов для проверки внутреннего состояния объекта перед применением изменений к этому состоянию. Можно проверить сочетания свойств. После того как вы создадите таблицы для любого из этих переходов, протестируйте код, а затем упростите его для удобочитаемости и поддерживаемости. Эти первоначальные изменения кода могут предложить дополнительные изменения кода, которые проверяют внутреннее состояние или управляют другими изменениями в API. В этом руководстве объединяются классы и объекты с более ориентированным на данные подходом на основе шаблонов для реализации этих классов.

Консольное приложение

Статья • 31.03.2025

В этом руководстве вы узнаете о нескольких функциях в .NET и языке C#. Вы узнаете:

- Основы интерфейса командной строки .NET
- Структура консольного приложения C#
- Ввод-вывод консоли
- Основы API-интерфейсов ввода-вывода файлов в .NET
- Основы асинхронного программирования на основе задач в .NET

Вы создадите приложение, которое считывает текстовый файл и отражает содержимое этого текстового файла в консоли. Вывод на консоль синхронизирован так, чтобы соответствовать ритму чтения вслух. Вы можете ускорить или замедлить темп, нажав клавиши "<" (меньше) или ">" (больше). Это приложение можно запустить в Windows, Linux, macOS или в контейнере Docker.

В этом руководстве представлено много особенностей. Давайте создадим их по одному.

Предпосылки

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Создание приложения

Первым шагом является создание нового приложения. Откройте командную строку и создайте новый каталог для приложения. Сделайте это текущим каталогом. Введите команду `dotnet new console` в командной строке. При этом создаются начальные файлы для базового приложения Hello World.

Прежде чем приступить к внесению изменений, давайте запустите простое приложение Hello World. После создания приложения введите `dotnet run` в командной строке. Эта команда запускает процесс восстановления пакета NuGet, создает исполняемый файл приложения и запускает исполняемый файл.

Простой код приложения Hello World находится в *Program.cs*. Откройте этот файл с помощью избранного текстового редактора. Замените код в *Program.cs* следующим

кодом:

```C#
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

В верхней части файла см. инструкцию `namespace`. Как и другие языки, ориентированные на объекты, которые вы могли использовать, C# использует пространства имен для упорядочивания типов. Эта программа "Hello World" ничем не отличается. Вы можете увидеть, что программа находится в пространстве имен с именем `TeleprompterConsole`.

Чтение и эхо файла

Первая функция, которую необходимо добавить, — это возможность считывания текстового файла и отображения всего этого текста в консоли. Сначала добавим текстовый файл. Скопируйте файл sampleQuotes.txt из репозитория GitHub для этого примера в каталог проекта. Это будет служить скриптом для приложения. Сведения о том, как скачать образец приложения для этого руководства, см. в инструкциях Примеры и учебники.

Затем добавьте следующий метод в класс `Program` (справа под методом `Main`):

```C#
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Этот метод представляет собой особый тип метода C#, который называется *методом итератора*. Методы итератора возвращают последовательности, которые вычисляются лениво. Это означает, что каждый элемент в последовательности создается по мере запроса кода, потребляющего последовательность. Методы итератора — это методы, содержащие один или более операторов yield return. Объект, возвращаемый методом `ReadFrom`, содержит код для создания каждого элемента в последовательности. В этом примере предполагается чтение следующей строки текста из исходного файла и возвращение этой строки. Каждый раз, когда вызывающий код запрашивает следующий элемент из последовательности, код считывает следующую строку текста из файла и возвращает его. Когда файл полностью считывается, последовательность указывает, что больше элементов нет.

Существует два элемента синтаксиса C#, которые могут быть новыми для вас. Инструкция using в этом методе управляет очисткой ресурсов. Переменная, инициализированная в инструкции `using` (`reader`, в этом примере) должна реализовать интерфейс IDisposable. Этот интерфейс определяет один метод, `Dispose`, который должен вызываться при выпуске ресурса. Компилятор создает этот вызов, когда выполнение достигает закрывающей скобки оператора `using`. Код, сгенерированный компилятором, гарантирует, что ресурс освобождается, даже если из кода внутри блока, определяемого оператором using, происходит выброс исключения.

Переменная `reader` определяется с помощью ключевого слова `var`. var определяет *неявно типизированную локальную переменную*. Это означает, что тип переменной определяется типом времени компиляции объекта, назначенного переменной. Здесь возвращается значение из метода OpenText(String), который является объектом StreamReader.

Теперь давайте заполним код для чтения файла в методе `Main`:

C#

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Запустите программу (с помощью `dotnet run`) и вы увидите каждую строку, напечатанную в консоли.

Добавление задержек и форматирование вывода

То, что у вас отображается, слишком быстро, чтобы читать вслух. Теперь необходимо добавить задержки в выходных данных. При запуске вы создадите некоторый основной код, который обеспечивает асинхронную обработку. Однако эти первые шаги будут следовать нескольким антипаттернам. Антишаблоны указываются в комментариях при добавлении кода, и код будет обновлен в последующих шагах.

В этом разделе описано два шага. Во-первых, вы обновите метод итератора, чтобы возвращать отдельные слова вместо целых строк. Это сделано с этими изменениями. Замените инструкцию `yield return line;` следующим кодом:

C#

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Затем необходимо изменить способ использования строк файла и добавить задержку после написания каждого слова. Замените инструкцию `Console.WriteLine(line)` в методе `Main` следующим блоком:

C#

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Запустите пример и проверьте выходные данные. Теперь каждое слово печатается, за которым следует задержка в 200 мс. Однако отображаемые выходные данные показывают некоторые проблемы, так как исходный текстовый файл содержит несколько строк с более чем 80 символами без разрыва строки. Это может быть трудно прочитать, пока текст прокручивается. Это легко исправить. Вы просто

следите за длиной каждой строки и создадите новую строку, когда длина строки достигает определенного порогового значения. Объявите локальную переменную после объявления `words` в методе `ReadFrom`, которая будет хранить длину строки.

```
C#

var lineLength = 0;
```

Затем добавьте следующий код после конструкции `yield return word + " ";` (перед закрывающей скобкой):

```
C#

lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Запустите пример, и вы сможете читать вслух с заранее настроенной скоростью.

Асинхронные задачи

На этом последнем шаге вы добавите код для асинхронной записи выходных данных в одной задаче, а также выполнение другой задачи для чтения входных данных от пользователя, если они хотят ускорить или замедлить отображение текста, или полностью остановить отображение текста. Это включает несколько шагов, и к концу у вас будут все необходимые обновления. Первым шагом является создание асинхронного метода Task, который возвращает код, созданный для чтения и отображения файла.

Добавьте этот метод в класс `Program` (он взят из текста метода `Main`):

```
C#

private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
```

```
        }
    }
}
```

Вы заметите два изменения. Во-первых, в тексте метода вместо вызова Wait() для синхронного ожидания завершения задачи эта версия использует ключевое слово `await`. Для этого необходимо добавить модификатор `async` в сигнатуру метода. Этот метод возвращает `Task`. Обратите внимание, что ни один оператор return не возвращает объект `Task`. Вместо этого этот объект `Task` создается кодом, который компилятор создает при использовании оператора `await`. Вы можете представить, что этот метод возвращается, когда он достигает `await`. Возвращенный `Task` указывает, что работа не завершена. Метод возобновляется, когда ожидается выполнение задачи. После полного выполнения возвращаемое значение `Task` указывает, что оно завершено. Вызывающий код может отслеживать возвращаемый `Task`, чтобы определить момент его завершения.

Добавьте ключевое слово `await` перед вызовом `ShowTeleprompter`:

C#

```
await ShowTeleprompter();
```

Для этого необходимо изменить сигнатуру метода `Main` на:

C#

```
static async Task Main(string[] args)
```

Узнайте больше о методе async Main в разделе основы.

Затем необходимо написать второй асинхронный метод для чтения из консоли и следить за клавишами '<' (меньше), '>' (больше) и 'X' или 'x'. Ниже приведен метод, который вы добавили для этой задачи:

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
```

```
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

Это создает лямбда-выражение для представления делегата Action, который считывает ключ из консоли и изменяет локальную переменную, представляющую задержку, когда пользователь нажимает клавиши "<" (меньше) или ">" (больше). Метод делегата завершается, когда пользователь нажимает клавиши X или X, что позволяет пользователю останавливать отображение текста в любое время. Этот метод использует ReadKey() для блокировки и ожидания нажатия клавиши пользователем.

Чтобы завершить эту функцию, необходимо создать новый метод возврата `async Task`, который запускает обе эти задачи (`GetInput` и `ShowTeleprompter`), а также управляет общими данными между этими двумя задачами.

Пришло время создать класс, который может обрабатывать общие данные между этими двумя задачами. Этот класс содержит два общедоступных свойства: задержку и флаг `Done`, чтобы указать, что файл полностью считывается:

C#

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
```

```
    }
}
```

Поместите этот класс в новый файл и включите этот класс в пространство имен `TeleprompterConsole`, как показано ниже. Кроме того, необходимо добавить инструкцию `using static` в верхней части файла, чтобы можно было ссылаться на методы `Min` и `Max` без вложенных имен классов или пространств имен. Инструкция using static импортирует методы из одного класса. Это отличается от инструкции `using` без `static`, которая импортирует все классы из пространства имен.

C#

```
using static System.Math;
```

Затем необходимо обновить методы `ShowTeleprompter` и `GetInput` для использования нового объекта `config`. Напишите финальный метод `Task`, который возвращает `async`, чтобы запустить обе задачи и завершить выполнение, когда первая задача завершится.

C#

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

Одним из новых методов является вызов WhenAny(Task[]). Это создает `Task`, который завершает выполнение как только любая из задач в списке его аргументов завершится.

Затем необходимо обновить методы `ShowTeleprompter` и `GetInput`, чтобы использовать объект `config` для задержки:

C#

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
```

```
            if (!string.IsNullOrWhiteSpace(word))
            {
                await Task.Delay(config.DelayInMilliseconds);
            }
        }
        config.SetDone();
    }

    private static async Task GetInput(TelePrompterConfig config)
    {
        Action work = () =>
        {
            do {
                var key = Console.ReadKey(true);
                if (key.KeyChar == '>')
                    config.UpdateDelay(-10);
                else if (key.KeyChar == '<')
                    config.UpdateDelay(10);
                else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                    config.SetDone();
            } while (!config.Done);
        };
        await Task.Run(work);
    }
```

Эта новая версия `ShowTeleprompter` вызывает новый метод в классе `TelePrompterConfig`. Теперь необходимо обновить `Main` для вызова `RunTeleprompter` вместо `ShowTeleprompter`:

C#

```
await RunTeleprompter();
```

Заключение

В этом руководстве показано несколько функций языка C# и библиотек .NET Core, связанных с работой в консольных приложениях. Вы можете опираться на эти знания, чтобы узнать больше о языке и классах, представленных здесь. Вы узнали основы ввода-вывода файлов и консоли, освоили блокированное и неблокированное использование асинхронного программирования на основе задач, получили обзор возможностей языка C#, узнали, как организованы программы на C#, и познакомились с интерфейсом командной строки .NET.

Дополнительные сведения о файловом вводе-выводе см. в File and Stream I/O. Дополнительные сведения об асинхронной модели программирования,

используемой в этом руководстве, см. в асинхронного программирования на основе задач и асинхронного программирования.

Руководство. Создание HTTP-запросов в консольном приложении .NET с помощью C#

В этом руководстве создается приложение, которое выдает HTTP-запросы к службе REST на GitHub. Приложение считывает сведения в формате JSON и преобразует JSON в объекты C#. Преобразование из JSON в объекты C# называется *десериализацией*.

В руководстве показано, как:

- ✓ Отправка HTTP-запросов.
- ✓ Десериализация ответов JSON.
- ✓ Настройте десериализацию с помощью атрибутов.

Если вы предпочитаете следовать последнему образцу этого руководства, его можно скачать. Инструкции по скачиванию смотрите в разделах Образцы и руководства.

Предпосылки

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Создание клиентского приложения

1. Откройте командную строку и создайте новый каталог для приложения. Сделайте этот каталог текущим.

2. Введите следующую команду в окне консоли:

   Интерфейс командной строки.NET

   ```
   dotnet new console --name WebAPIClient
   ```

   Эта команда создает начальные файлы для базового приложения Hello World. Имя проекта — WebAPIClient.

3. Перейдите в каталог WebAPIClient и запустите приложение.

   Интерфейс командной строки.NET

```
cd WebAPIClient
```

Интерфейс командной строки.NET

```
dotnet run
```

dotnet run автоматически запускает dotnet restore для восстановления всех зависимостей, которые требуются приложению. Он выполняется также при необходимости dotnet build. Вы увидите выходные данные `"Hello, World!"` приложения. В терминале нажмите `клавиши CTRL` + `C` , чтобы остановить приложение.

Создание HTTP-запросов

Это приложение вызывает API GitHub , чтобы получить сведения о проектах под зонтиком .NET Foundation . Конечная точка https://api.github.com/orgs/dotnet/repos. Чтобы получить сведения, он делает HTTP-запрос GET. Браузеры также делают HTTP-запросы GET, поэтому вы можете вставить этот URL-адрес в адресную строку браузера, чтобы узнать, какие данные вы будете получать и обрабатывать.

HttpClient Используйте класс для выполнения HTTP-запросов. HttpClient поддерживает только асинхронные методы для своих долговременных API. Поэтому следующие шаги создают асинхронный метод и вызывают его из метода Main.

1. Откройте файл в каталоге `Program.cs` проекта и замените его содержимое следующим образом:

   C#

   ```
   await ProcessRepositoriesAsync();

   static async Task ProcessRepositoriesAsync(HttpClient client)
   {
   }
   ```

   Этот код:

   - Заменяет инструкцию `Console.WriteLine` вызовом `ProcessRepositoriesAsync`, с использованием ключевого слова `await`.
   - Определяет пустой `ProcessRepositoriesAsync` метод.

2. `Program` В классе используйте HttpClient для обработки запросов и ответов, заменив содержимое следующим кодом C#.

C#

```
using System.Net.Http.Headers;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository
Reporter");

await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

Этот код:

- Настраивает заголовки HTTP для всех запросов:
  - Заголовок Accept для принятия ответов JSON
  - Заголовок User-Agent. Эти заголовки проверяются кодом сервера GitHub и необходимы для получения сведений из GitHub.

3. В методе `ProcessRepositoriesAsync` вызовите конечную точку GitHub, которая возвращает список всех репозиториев в организации .NET foundation:

C#

```
static async Task ProcessRepositoriesAsync(HttpClient client)
{
    var json = await client.GetStringAsync(
        "https://api.github.com/orgs/dotnet/repos");

    Console.Write(json);
}
```

Этот код:

- Ожидает задачу, возвращенную в результате вызова метода HttpClient.GetStringAsync(String). Этот метод отправляет HTTP-запрос GET в указанный универсальный код ресурса (URI). Текст ответа возвращается как объект String, который доступен при завершении задачи.
- Строка `json` ответа выводится в консоль.

4. Создайте приложение и запустите его.

Интерфейс командной строки.NET

```
dotnet run
```

Предупреждение о сборке отсутствует, так как теперь `ProcessRepositoriesAsync` содержит оператор `await`. Выходные данные — это длинное отображение текста JSON.

Десериализация результата JSON

Ниже описано, как упростить подход к выбору данных и обработке данных. Вы будете использовать GetFromJsonAsync метод расширения, который входит в пакет NuGet System.Net.Http.Json , чтобы получить и десериализировать результаты JSON в объекты.

1. Создайте файл *с именем Repository.cs* и добавьте следующий код:

   C#

   ```
   public record class Repository(string Name);
   ```

   Предыдущий код определяет класс для представления объекта JSON, возвращаемого из API GitHub. Этот класс будет использоваться для отображения списка имен репозитория.

   JSON для объекта репозитория содержит десятки свойств, но только `Name` свойство будет десериализировано. Сериализатор автоматически игнорирует свойства JSON, для которых нет совпадений в целевом классе. Эта функция упрощает создание типов, которые работают только с подмножеством полей в большом пакете JSON.

   Несмотря на `GetFromJsonAsync` то, что метод, который будет использоваться в следующей точке, имеет преимущество без учета регистра, когда дело доходит до имен свойств, соглашение C# заключается в том, чтобы заглавить первую букву имен свойств.

2. HttpClientJsonExtensions.GetFromJsonAsync Используйте метод для получения и преобразования JSON в объекты C#. Замените вызов GetStringAsync(String) в методе `ProcessRepositoriesAsync` следующими строками:

   C#

   ```
   var repositories = await client.GetFromJsonAsync<List<Repository>>
   ("https://api.github.com/orgs/dotnet/repos");
   ```

   Обновленный код заменяет GetStringAsync(String) на HttpClientJsonExtensions.GetFromJsonAsync.

Первым аргументом `GetFromJsonAsync` метода `await` является выражение. `await` выражения могут отображаться практически в любом месте кода, даже если до сих пор вы видели их только как часть инструкции назначения. Следующий параметр `requestUri` является необязательным и не должен быть указан, если он уже указан при создании `client` объекта. Вы не предоставили объекту универсальный `client` код ресурса (URI) для отправки запроса, поэтому вы указали универсальный код ресурса (URI). Последний необязательный параметр `CancellationToken` опущен в фрагменте кода.

Метод `GetFromJsonAsync` *является универсальным*, что означает, что вы предоставляете аргументы типа для того, какой тип объектов следует создать из полученного текста JSON. В этом примере вы выполняете десериализацию в объект `List<Repository>`, который является другим универсальным объектом, System.Collections.Generic.List<T>. Класс `List<T>` хранит коллекцию объектов. Аргумент типа объявляет тип объектов, хранящихся в объекте `List<T>`. Аргумент типа — это запись формата `Repository`, поскольку текст JSON представляет собой коллекцию объектов репозитория.

3. Добавьте код для отображения имени каждого репозитория. Замените строки, гласящие:

   C#

   ```
   Console.Write(json);
   ```

   указанным ниже кодом:

   C#

   ```
   foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
       Console.WriteLine(repo.Name);
   ```

4. В верхней части файла должны присутствовать следующие `using` директивы:

   C#

   ```
   using System.Net.Http.Headers;
   using System.Net.Http.Json;
   ```

5. Запустите приложение.

   Интерфейс командной строки.NET

```
dotnet run
```

Выходные данные — это список имен репозиториев, которые являются частью .NET Foundation.

Рефакторинг кода

Метод `ProcessRepositoriesAsync` может выполнять асинхронную работу и возвращать коллекцию репозиториев. Измените этот метод, чтобы он возвращал `Task<List<Repository>>`, и переместите код, который выводится в консоль, рядом с местом вызова.

1. Измените сигнатуру `ProcessRepositoriesAsync` для возврата задачи, результатом которой является список `Repository` объектов:

   C#

   ```
   static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient client)
   ```

2. Верните репозитории после обработки ответа JSON:

   C#

   ```
   var repositories = await client.GetFromJsonAsync<List<Repository>>
   ("https://api.github.com/orgs/dotnet/repos");
   return repositories ?? new();
   ```

   Компилятор создает `Task<T>` объект для возвращаемого значения, так как этот метод помечен как `async`.

3. Измените *файл Program.cs* , заменив вызов `ProcessRepositoriesAsync` следующим образом, чтобы записать результаты и записать каждое имя репозитория в консоль.

   C#

   ```
   var repositories = await ProcessRepositoriesAsync(client);

   foreach (var repo in repositories)
       Console.WriteLine(repo.Name);
   ```

4. Запустите приложение.

   Выходные данные одинаковы.

Десериализация дополнительных свойств

В следующих шагах мы расширим код для обработки дополнительных свойств полезных данных JSON, возвращаемых API GitHub. Возможно, вам не потребуется обрабатывать каждое свойство, но добавление нескольких демонстрирует дополнительные возможности C#.

1. Замените содержимое `Repository` класса следующим `record` определением. Обязательно импортируйте `System.Text.Json.Serialization` пространство имен и примените `[JsonPropertyName]` атрибут для сопоставления полей JSON с свойствами C# явным образом.

   C#

   ```
   using System.Text.Json.Serialization;

   public record class Repository(
     string Name,
     string Description,
     [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
     Uri Homepage,
     int Watchers,
     [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc
    );
   ```

   Типы Uri и `int` имеют встроенные функции для преобразования в строковое представление и обратно. Дополнительный код не требуется для десериализации из формата строки JSON в эти целевые типы. Если пакет JSON содержит данные, которые не преобразуются в целевой тип, действие сериализации создает исключение.

   JSON часто использует `lowercase` или `snake_case` в качестве имен свойств. Такие поля, как `html_url` и `pushed_at` не следуют соглашениям об именовании C# PascalCase. Использование `[JsonPropertyName]` гарантирует правильность привязки этих ключей JSON к соответствующим свойствам C#, даже если их имена отличаются в случае или содержат подчеркивания. Этот подход гарантирует прогнозируемую и стабильную десериализацию, позволяя определение имен свойств в стиле PascalCase в C#. Кроме того, `GetFromJsonAsync` метод используется `case-insensitive` при сопоставлении имен свойств, поэтому дальнейшее преобразование не требуется.

2. `foreach` Обновите цикл в файле *Program.cs*, чтобы отобразить значения свойств:

   C#

```
foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine();
}
```

3. Запустите приложение.

   Теперь список содержит дополнительные свойства.

Добавьте свойство даты

Дата последней операции передачи форматируется следующим образом в ответе JSON:

JSON

```
2016-02-08T21:27:00Z
```

Этот формат предназначен для времени по универсальному координированному времени (UTC), поэтому результат десериализации — это значение DateTime, и его свойство Kind имеет значение Utc.

Чтобы получить дату и время, представленные в часовом поясе, необходимо написать пользовательский метод преобразования.

1. В *Repository.cs* добавьте свойство для представления даты и времени в формате UTC и свойство readonly `LastPush`, которое возвращает дату, преобразованную в местное время, файл должен выглядеть следующим образом:

   C#

```
using System.Text.Json.Serialization;

public record class Repository(
    string Name,
    string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
    Uri Homepage,
    int Watchers,
    [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc
    )
{
```

```
    public DateTime LastPush => LastPushUtc.ToLocalTime();
}
```

Свойство `LastPush` определяется с помощью *члена, основанного на выражении* для `get` аксессора. Нет `set` акцессора. Опущение `set` метода доступа является одним из способов определения свойства *только для чтения* в C#. (Да, можно создать свойства *только для записи* в C#, но их значение ограничено.)

2. Добавьте еще одну инструкцию вывода в *Program.cs*: снова:

C#

```
Console.WriteLine($"Last push: {repo.LastPush}");
```

3. Полное приложение должно выглядеть следующим *Program.cs* файлом:

C#

```
using System.Net.Http.Headers;
using System.Net.Http.Json;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository
Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient client)
{
    var repositories = await client.GetFromJsonAsync<List<Repository>>
("https://api.github.com/orgs/dotnet/repos");
    return repositories ?? new List<Repository>();
}
```

4. Запустите приложение.

Выходные данные включают дату и время последнего отправки в каждый репозиторий.

Дальнейшие действия

В этом руководстве вы создали приложение, которое выполняет веб-запросы и анализирует результаты. Версия приложения должна соответствовать готовому образцу.

Подробнее о том, как настроить сериализацию JSON, читайте в разделе "Как сериализовать и десериализовать (маршалировать и демаршалировать) JSON в .NET.

Last updated on 28.10.2025

Работа с запросом Language-Integrated (LINQ)

Введение

В этом руководстве описаны функции .NET и языка C#. Вы узнаете, как:

- Создание последовательностей с помощью LINQ.
- Написание методов, которые можно легко использовать в запросах LINQ.
- Различаете жадную и ленивую оценку.

Вы узнаете эти методы, создав приложение, которое демонстрирует один из основных навыков любого мага: тасовка "фаро". Фаро перетасовка — это техника, когда вы разделяете колоду карт ровно пополам, а затем поочередно перемешиваются карты из каждой половины, чтобы восстановить исходную колоду.

Фокусники используют этот метод, потому что каждая карта находится в известном месте после каждой перетасовки, и порядок повторяется.

В этом руководстве описано, как управлять последовательности данных. Приложение создает колоду карт, выполняет последовательность перетасовок и каждый раз записывает эту последовательность. Он также сравнивает обновленный порядок с исходным порядком.

В этом руководстве описано несколько шагов. После каждого шага можно запустить приложение и просмотреть ход выполнения. Вы также можете увидеть готовый пример в репозитории dotnet/samples на GitHub. Инструкции по скачиванию смотрите в разделах Образцы и руководства.

Предпосылки

- Последняя версия .NET SDK
- редактор Visual Studio Code
- C# DevKit

Создание приложения

Создайте новое приложение. Откройте командную строку и создайте новый каталог для приложения. Сделайте это текущим каталогом. Введите команду `dotnet new console -o`

`LinqFaroShuffle` в командной строке. Эта команда создает начальные файлы для базового приложения Hello World.

Если вы еще не использовали C#, в этом руководстве объясняется структура программы C#. Вы можете прочитать это, а затем вернуться сюда, чтобы узнать больше о LINQ.

Создание набора данных

> **Кончик**
>
> В этом руководстве можно упорядочить код в пространстве имен, называемом `LinqFaroShuffle`, чтобы соответствовать примеру кода, либо использовать глобальное пространство имен по умолчанию. Если вы решили использовать пространство имен, убедитесь, что все ваши классы и методы последовательно находятся в одном пространстве имен или добавьте соответствующие `using` инструкции по мере необходимости.

Рассмотрим, что представляет собой колоду карточек. Колода игровых карт имеет четыре костюма, и каждый костюм имеет 13 значений. Как правило, можно сразу создать `Card` класс и заполнить коллекцию `Card` объектов вручную. С помощью LINQ вы можете быть более кратким, чем обычный способ создания колоды карт. Вместо создания `Card` класс создайте две последовательности для представления мастей и рангов. Создайте пару *методов итератора*, которые генерируют ранги и масти в виде строк IEnumerable<T>:

C#

```
static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
```

```
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Поместите эти методы в инструкцию `Console.WriteLine` в `Program.cs` файле. Эти два метода используют `yield return` синтаксис для продуцирования последовательности во время выполнения. Компилятор создает объект, реализующий IEnumerable<T> и создающий последовательность строк по мере их запроса.

Теперь используйте эти методы итератора для создания колоды карт. Поместите запрос LINQ в начало `Program.cs` файла. Вот как выглядит:

C#

```
var startingDeck = from s in Suits()
                   from r in Ranks()
                   select (Suit: s, Rank: r);

// Display each card that's generated and placed in startingDeck
foreach (var card in startingDeck)
{
    Console.WriteLine(card);
}
```

Когда несколько предложений `from` создают SelectMany, это формирует единую последовательность путем объединения элементов первой и второй последовательностей. Порядок важен для этого примера. Первый элемент в первой исходной последовательности (Suits) комбинируется с каждым элементом во второй последовательности (Ranks). Этот процесс создает все 13 карточек первой масти. Этот процесс повторяется с каждым элементом в первой последовательности (Suits). Конечный результат — это колода карт, упорядоченная по мастям, после чего по значениям.

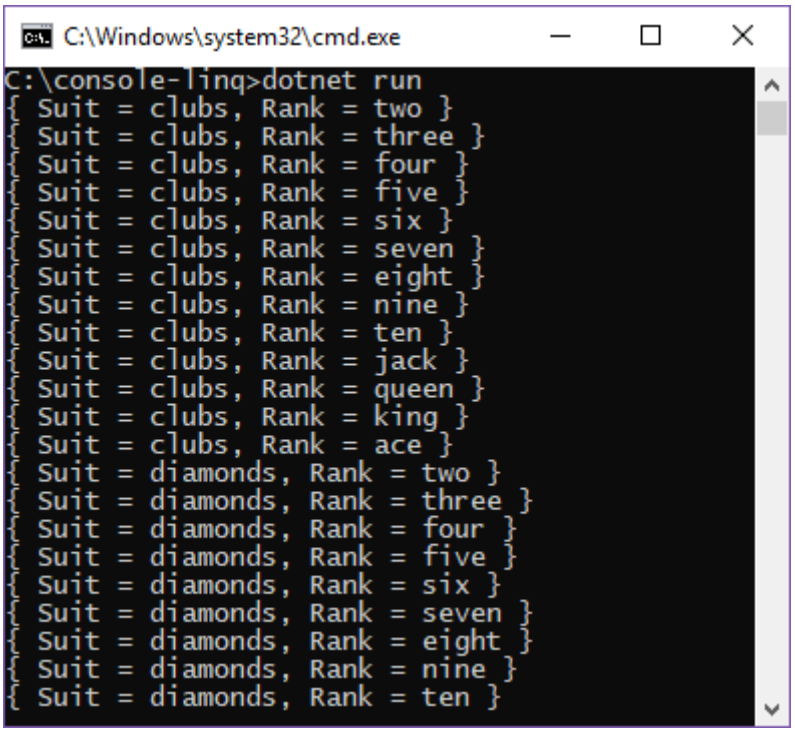
Имейте в виду, что если вы пишете LINQ в синтаксисе запроса, используемом в предыдущем примере или используйте синтаксис метода вместо этого, всегда можно перейти из одной формы синтаксиса в другую. Предыдущий запрос, написанный в синтаксисе запроса, можно записать в синтаксисе метода следующим образом:

C#

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => (Suit: suit,
Rank: rank )));
```

Компилятор преобразует инструкции LINQ, написанные с синтаксисом запроса, в эквивалентный синтаксис вызова метода. Таким образом, независимо от выбранного синтаксиса, две версии запроса создают один и тот же результат. Выберите синтаксис, который лучше всего подходит для вашей ситуации. Например, если вы работаете в команде, где некоторые члены имеют трудности с синтаксисом метода, попробуйте использовать синтаксис запроса.

Запустите пример программы, созданный на данной стадии. Он отображает все 52 карточки в колоде. Может быть полезно запустить этот пример в отладчике, чтобы наблюдать, как выполняются методы `Suits()` и `Ranks()`. Ясно видно, что каждая строка в каждой последовательности создается только по мере необходимости.



Управление порядком

Затем сосредоточьтесь на том, как вы тасуете карты в колоде. Первый шаг в любом хорошем перетасовке заключается в разбиении колоды на две части. Методы Take , Skip которые являются частью API-интерфейсов LINQ, предоставляют эту функцию. Поместите их следующим образом в цикл `foreach`:

C#

```
var top = startingDeck.Take(26);
var bottom = startingDeck.Skip(26);
```

Тем не менее, нет метода перетасовки, чтобы воспользоваться преимуществами стандартной библиотеки, поэтому вам нужно написать свой собственный. Метод

перетасовки, который вы создаете, иллюстрирует несколько техник, используемых с программами на основе LINQ, поэтому каждая часть этого процесса описана поэтапно.

Чтобы добавить функциональные возможности в взаимодействие с IEnumerable<T> результатами запросов LINQ, необходимо написать некоторые специальные методы, называемые методами расширения. Метод расширения — это *статический метод* специального назначения, который добавляет новые функциональные возможности к уже существующему типу, не изменяя исходный тип, к которому требуется добавить функциональные возможности.

Добавьте методы расширения в новое место, создав в программе новый статический класс с именем , а затем начните разработку первого метода расширения:

C#

```
public static class CardExtensions
{
    extension<T>(IEnumerable<T> sequence)
    {
        public IEnumerable<T> InterleaveSequenceWith(IEnumerable<T> second)
        {
            // Your implementation goes here
            return default;
        }
    }
}
```

> ⓘ **Заметка**
>
> Если вы используете редактор, отличный от Visual Studio (например, Visual Studio Code), вам может потребоваться добавить `using LinqFaroShuffle;` в верхнюю часть файла *Program.cs* , чтобы методы расширения были доступны. Visual Studio автоматически добавляет директиву using, но другие редакторы могут не делать этого.

Контейнер `extension` задает расширенный тип. Узел `extension` объявляет тип и имя *параметра приемника* для всех членов в контейнере `extension` . В этом примере вы расширяете `IEnumerable<T>`, а параметр называется `sequence`.

Объявления членов расширения отображаются, как будто они являются членами типа приемника:

C#

```
public IEnumerable<T> InterleaveSequenceWith(IEnumerable<T> second)
```

Метод вызывается так, как если бы он был методом-участником расширенного типа. Это объявление метода также соответствует распространенной идиоме, где типы входных и выходных данных обозначены как `IEnumerable<T>`. Эта практика позволяет объединить методы LINQ в цепочку для выполнения более сложных запросов.

Поскольку вы разделили колоду на половины, вам нужно соединить эти половины. В коде это означает, что вы перечисляете обе последовательности, которые вы получили посредством Take и Skip, одновременно, *чередуя элементы и создавая* одну последовательность: вашу теперь уже перемешанную колоду карт. Написание метода LINQ, работающего с двумя последовательности, требует понимания того, как работает IEnumerable<T>.

Интерфейс IEnumerable<T> имеет один метод: GetEnumerator. Объект, возвращаемый GetEnumerator, обладает методом для перехода к следующему элементу и свойством, которое извлекает текущий элемент из последовательности. Вы используете эти два члена для перебора коллекции и возврата элементов. Этот метод Interleave является методом итератора, поэтому вместо создания коллекции и возврата коллекции используется `yield return` синтаксис, показанный в предыдущем коде.

Вот реализация этого метода:

C#

```
public IEnumerable<T> InterleaveSequenceWith(IEnumerable<T> second)
{
    var firstIter = sequence.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Теперь, когда вы написали этот метод, вернитесь к `Main` методу и перетасуйте колоду один раз.

C#

```
var shuffledDeck = top.InterleaveSequenceWith(bottom);

foreach (var c in shuffledDeck)
{
```

```
    Console.WriteLine(c);
}
```

Сравнения

Определите, сколько перемешиваний требуется, чтобы вернуть колоду в исходный порядок. Чтобы узнать, напишите метод, определяющий, равны ли две последовательности. После того как у вас будет этот метод, поместите код, который тасует колоду, в цикл, и проверьте, когда колода возвращается в правильный порядок.

Написать метод для определения, равны ли две последовательности, должно быть несложно. Это аналогичный метод, который вы написали, чтобы перетасовать колоду. Однако на этот раз вместо использования `yield return` для каждого элемента сравниваются соответствующие элементы каждой последовательности. При перечислении всей последовательности, если каждый элемент совпадает, последовательности одинаковы:

C#

```
public bool SequenceEquals(IEnumerable<T> second)
{
    var firstIter = sequence.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Этот метод показывает второй идиом LINQ: методы терминала. Они принимают последовательность в качестве входных данных (или в данном случае две последовательности) и возвращают одно скалярное значение. При использовании методов терминала они всегда является окончательным методом в цепочке методов для запроса LINQ.

Это можно увидеть в действии, когда используется для определения, когда палуба возвращается в обратный исходный порядок. Поместите код перетасовки внутри цикла и остановите, когда последовательность восстановилась в своем исходном порядке,

применяя метод `SequenceEquals()`. Вы можете увидеть, что он всегда будет окончательным методом в любом запросе, так как он возвращает одно значение вместо последовательности:

C#

```
var startingDeck = from s in Suits()
                   from r in Ranks()
                   select (Suit: s, Rank: r);

// Display each card generated and placed in startingDeck in the console
foreach (var card in startingDeck)
{
    Console.WriteLine(card);
}

var top = startingDeck.Take(26);
var bottom = startingDeck.Skip(26);

var shuffledDeck = top.InterleaveSequenceWith(bottom);

var times = 0;
// Re-use the shuffle variable from earlier, or you can make a new one
shuffledDeck = startingDeck;
do
{
    shuffledDeck =
shuffledDeck.Take(26).InterleaveSequenceWith(shuffledDeck.Skip(26));

    foreach (var card in shuffledDeck)
    {
        Console.WriteLine(card);
    }
    Console.WriteLine();
    times++;

} while (!startingDeck.SequenceEquals(shuffledDeck));

Console.WriteLine(times);
```

Запустите код, построенный до этого момента, и обратите внимание, как колода карт перестраивается при каждом перетасовке. После 8 перетасовок (итераций цикла do-while) колода возвращается к первоначальной конфигурации, в которой она находилась при создании из исходного запроса LINQ.

Оптимизации

Созданный пример выполняет *перетасовку*, где верхние и нижние карточки остаются одинаковыми при каждом запуске. Давайте внесите одно изменение: используйте

вместо этого перетасовку , где все 52 карточки изменяют положение. Для перемешивания, когда вы чередуете карты, первая карта в нижней половине колоды становится первой картой в колоде. Это означает, что последняя карточка в верхней половине становится нижней карточкой. Для этого изменения требуется одна строка кода. Обновите текущий запрос перетасовки, переключив позиции Take и Skip. Это изменение переключает порядок верхних и нижних половин палубы:

C#

```
shuffledDeck = shuffledDeck.Skip(26).InterleaveSequenceWith(shuffledDeck.Take(26));
```

Запустите программу еще раз, и вы увидите, что для переупорядочивания колоды требуется 52 итерации. Вы также заметили некоторое серьезное снижение производительности, так как программа продолжает работать.

Существует несколько причин для снижения производительности. Вы можете решить одну из основных причин: неэффективное использование отложенной *оценки*.

Отложенное вычисление подразумевает, что оценка выражения не выполняется до тех пор, пока не потребуется его значение. Запросы LINQ — это запросы, которые оцениваются лениво. Последовательности создаются только при запросе элементов. Как правило, это основное преимущество LINQ. Однако в такой программе отложенная оценка приводит к экспоненциальному росту времени выполнения.

Не забывайте, что вы создали исходный набор карточек с помощью запроса LINQ. Каждое перемешивание осуществляется путем выполнения трех запросов LINQ на предыдущей колоде. Все эти запросы выполняются лениво. Это также означает, что они выполняются снова при каждом запросе последовательности. К тому времени, когда вы дойдете до 52-й итерации, вы воссоздаете исходную колоду много раз. Напишите журнал, чтобы продемонстрировать это поведение. После сбора данных можно повысить производительность.

`Extensions.cs` В файле введите или скопируйте метод в следующем примере кода. Этот метод расширения создает новый файл с именем `debug.log` в каталоге проекта и записывает, какой запрос выполняется в данный момент в файл журнала. Добавьте этот метод расширения к любому запросу, чтобы пометить выполнение запроса.

C#

```
public IEnumerable<T> LogQuery(string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
```

```
    }

    return sequence;
}
```

Затем добавьте лог-сообщение к определению каждого запроса.

C#

```
var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                    from r in Ranks().LogQuery("Rank Generation")
                    select (Suit: s, Rank: r)).LogQuery("Starting Deck");

foreach (var c in startingDeck)
{
    Console.WriteLine(c);
}

Console.WriteLine();
var times = 0;
var shuffle = startingDeck;

do
{
    // Out shuffle
    /*
    shuffle = shuffle.Take(26)
        .LogQuery("Top Half")
        .InterleaveSequenceWith(shuffle.Skip(26)
        .LogQuery("Bottom Half"))
        .LogQuery("Shuffle");
    */

    // In shuffle
    shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle");

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }

    times++;
    Console.WriteLine(times);
} while (!startingDeck.SequenceEquals(shuffle));

Console.WriteLine(times);
```

Обратите внимание, что каждый раз, когда вы обращаетесь к запросу, не ведётся запись в журнал. Логирование происходит только при создании исходного запроса. Программа по-прежнему занимает много времени, но теперь вы можете увидеть, почему. Если у вас

закончится терпение во время выполнения перетасовки с включенным ведением журнала, переключитесь обратно на исходную перетасовку. Вы по-прежнему видите эффекты ленивой оценки. В одном запуске выполняется 2592 запроса, включая создание значения и костюма.

Вы можете повысить производительность кода, чтобы уменьшить количество выполняемых выполнений. Простое исправление заключается в *кэшировании* результатов исходного запроса LINQ, который создает колоду карт. В настоящее время вы выполняете запросы снова и снова, когда цикл do-while проходит через итерацию, реконструируете колоду карт и перетасовываете её каждый раз. Чтобы кэшировать колоду карт, примените методы ToArray LINQ и ToList. При добавлении их к запросам они выполняют те же действия, которые вы сказали им, но теперь они хранят результаты в массиве или списке в зависимости от того, какой метод вы решили вызвать. Добавьте метод LINQ ToArray в оба запроса и снова запустите программу:

C#

```
var startingDeck = (from s in suits().LogQuery("Suit Generation")
                    from r in ranks().LogQuery("Value Generation")
                    select new { Suit = s, Rank = r })
                    .LogQuery("Starting Deck")
                    .ToArray();

foreach (var c in startingDeck)
{
    Console.WriteLine(c);
}

Console.WriteLine();

var times = 0;
var shuffle = startingDeck;

do
{
    /*
    shuffle = shuffle.Take(26)
        .LogQuery("Top Half")
        .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
        .LogQuery("Shuffle")
        .ToArray();
    */

    shuffle = shuffle.Skip(26)
        .LogQuery("Bottom Half")
        .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
        .LogQuery("Shuffle")
        .ToArray();

    foreach (var c in shuffle)
```

```
    {
        Console.WriteLine(c);
    }

    times++;
    Console.WriteLine(times);
} while (!startingDeck.SequenceEquals(shuffle));

Console.WriteLine(times);
```

Теперь количество перетасовок уменьшено до 30 запросов. Запустите снова с использованием перемешивания, и вы увидите аналогичные улучшения: теперь выполняет 162 запроса.

Этот пример **предназначен** для выделения вариантов использования, в которых отложенная оценка может вызвать трудности с производительностью. Хотя важно понять, где отложенная оценка может повлиять на производительность кода, столь же важно осознавать, что не все запросы должны выполняться сразу. Производительность снижается без использования ToArray, потому что каждое новое расположение колоды карт строится из предыдущего расположения. Использование отложенной оценки означает, что каждая новая конфигурация палубы создается на основе исходной палубы, даже выполняя код, созданный `startingDeck`. Это приводит к большому количеству дополнительной работы.

На практике некоторые алгоритмы работают хорошо с помощью жадной оценки, а другие работают хорошо с помощью отложенной оценки. Для ежедневного использования отложенная оценка обычно лучше подходит, когда источник данных является отдельным процессом, например ядром СУБД. Для баз данных отложенная оценка позволяет выполнять более сложные запросы в один цикл к процессу базы данных и обратно к остальному коду. LINQ является гибким, независимо от того, хотите ли вы использовать отложенную или немедленную оценку, поэтому измеряйте процессы и выбирайте, какая оценка даёт вам лучшую производительность.

Заключение

В этом проекте вы рассмотрели:

- Использование запросов LINQ для агрегирования данных в понятной последовательности.
- Написание методов расширения для добавления пользовательских функций в запросы LINQ.
- Поиск областей в коде, где запросы LINQ могут столкнуться с проблемами производительности, такими как снижение скорости.

- Отложенная и предварительная оценка в запросах LINQ и последствия, которые они могут иметь для производительности запросов.

Помимо LINQ, вы узнали о технике магов, используемых для карточных трюков. Маги используют перетасовку фаро, потому что могут контролировать перемещение каждой карты в колоде. Теперь, когда вы знаете, не портите это для всех остальных!

Дополнительные сведения о LINQ см. в следующем разделе:

- Общие сведения о LINQ
- Базовые операции запросов LINQ (C#)
- Преобразования данных с помощью LINQ (C#)
- Синтаксис запросов и синтаксис методов в LINQ (C#)
- Функции C#, поддерживающие LINQ

ⓘ **Заметка:** Автор создал эту статью с помощью искусственного интеллекта. Подробнее

Last updated on 25.11.2025

Интегрированный язык запросов (LINQ)

Language-Integrated Query (LINQ) — это название набора технологий, основанных на интеграции возможностей выполнения запросов непосредственно в язык C#. Традиционно запросы к данным выражаются как простые строки без проверки типа во время компиляции или поддержки IntelliSense. Кроме того, необходимо узнать другой язык запросов для каждого типа источника данных: базы данных SQL, XML-документы, различные веб-службы и т. д. При использовании LINQ запрос — это конструкция языка первого класса, как классы, методы и события.

При написании запросов наиболее видимой частью LINQ, интегрированной с языком, является выражение запроса. Выражения запросов записываются в декларативном *синтаксисе запроса*. С помощью синтаксиса запроса выполняется фильтрация, упорядочивание и группирование операций с источниками данных с минимальным количеством кода. Вы используете те же шаблоны выражений запросов для запроса и преобразования данных из любого типа источника данных.

В следующем примере показана полная операция запроса. Полная операция включает создание источника данных, определение выражения запроса и выполнение запроса в инструкции foreach .

C#

```
// Specify the data source.
int[] scores = [97, 92, 81, 60];

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (var i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Возможно, потребуется добавить директиву using , `using System.Linq;` чтобы выполнить компиляцию в предыдущем примере. Последние версии .NET используют неявные использование для добавления этой директивы в качестве глобального использования. Для более старых версий необходимо добавить его в исходный код.

Обзор выражения запроса

- Выражения запросов проверяют и преобразуют данные из любого источника данных с поддержкой LINQ. Например, один запрос может извлекать данные из базы данных SQL и создавать XML-поток в качестве выходных данных.
- Выражения запросов используют множество знакомых конструкций языка C#, что упрощает их чтение.
- Переменные в выражении запроса являются строго типизированными.
- Запрос не выполняется, пока не производится итерирование по переменной запроса, например в операторе `foreach`.
- Во время компиляции компилятор преобразует выражения запросов в стандартные вызовы метода оператора запроса в соответствии с правилами, определенными в спецификации C#. Вы можете выразить любой запрос, использующий синтаксис запроса, с помощью синтаксиса метода. В некоторых случаях синтаксис запросов является более читаемым и кратким. В других случае синтаксис метода является более читаемым. Между двумя различными формами нет никакой разницы в семантике или производительности. Дополнительные сведения см. в спецификации языка C# и обзоре стандартных операторов запросов.
- Некоторые операции запроса, такие как Count или Max, не имеют эквивалентного предложения выражения запроса и должны быть выражены как вызов метода. Синтаксис метода можно сочетать с синтаксисом запросов различными способами.
- Выражения запросов можно компилировать в деревья выражений или делегаты в зависимости от типа данных, к которому применяется запрос. Компилятор компилирует IEnumerable<T> запросы к делегатам. Компилятор компилирует запросы IQueryable и IQueryable<T> в деревья выражений. Дополнительные сведения см. в деревах выражений.

Как включить запросы LINQ к вашему источнику данных

Данные в памяти

Вы можете включить запросы LINQ к данным в памяти двумя способами. Если данные являются типом, реализующим IEnumerable<T>, запросите данные с помощью LINQ to Objects. Если не имеет смысла включать возможность перечисления путем реализации интерфейса IEnumerable<T>, определите методы стандартного оператора запроса LINQ либо в этом типе, либо как члены расширения для этого типа. Пользовательские реализации стандартных операторов запросов должны использовать отложенное выполнение для возврата результатов.

Удаленные данные

Лучший вариант включения запросов LINQ к удаленному источнику данных — реализовать IQueryable<T> интерфейс.

Поставщики IQueryable LINQ

Поставщики LINQ, которые реализуют IQueryable<T> , могут значительно отличаться в их сложности.

Менее сложный `IQueryable` поставщик может получить доступ к одному методу из веб-службы. Этот тип поставщика предназначен для источника данных, так как он ожидает определенную информацию в запросах, которые он обрабатывает. Он имеет систему закрытого типа, возможно, предоставляя один тип результата. Большая часть выполнения запроса происходит локально, например с помощью Enumerable реализаций стандартных операторов запросов. Менее сложный поставщик может изучить только одно выражение вызова метода в дереве выражений, представляющее запрос, и позволить оставшейся логике запроса обрабатываться в другом месте.

Поставщик `IQueryable` средней сложности может использовать источник данных с частично экспрессивным языком запросов. Если он предназначен для веб-службы, он может получить доступ к нескольким методам веб-службы и выбрать, какой метод вызвать, на основе информации, которую ищет запрос. Поставщик средней сложности имеет более богатую систему типов, чем простой поставщик, но она по-прежнему является системой фиксированного типа. Например, поставщик может предоставлять типы, имеющие связи "один ко многим", которые можно обойти, но не предоставляют технологии сопоставления для определяемых пользователем типов.

Сложный `IQueryable` поставщик, такой как поставщик Entity Framework Core , может перевести полные запросы LINQ на язык экспрессивных запросов, например SQL. Сложный поставщик является более общим, так как он может обрабатывать более широкий спектр вопросов в запросе. Он также имеет систему открытого типа и поэтому должен содержать обширную инфраструктуру для сопоставления определяемых пользователем типов. Для разработки сложного поставщика требуется значительное количество усилий.

ⓘ **Заметка:** Автор создал эту статью с помощью искусственного интеллекта. Подробнее

Last updated on 01.12.2025

Общие сведения о запросах LINQ в C#

Статья • 30.03.2025

запрос — это выражение, которое извлекает данные из источника данных. Различные источники данных имеют разные собственные языки запросов, например SQL для реляционных баз данных и XQuery для XML. Разработчики должны узнать новый язык запросов для каждого типа источника данных или формата данных, который они должны поддерживать. LINQ упрощает эту ситуацию, предлагая согласованную языковую модель C# для типов источников данных и форматов. В запросе LINQ вы всегда работаете с объектами C#. Для запроса и преобразования данных в XML-документах, базах данных SQL, коллекциях .NET и любом другом формате при наличии поставщика LINQ используются те же базовые шаблоны программирования.

Три части операции запроса

Все операции запроса LINQ состоят из трех отдельных действий:

1. Получите источник данных.
2. Создайте запрос.
3. Выполните запрос.

В следующем примере показано, как три части операции запроса выражаются в исходном коде. В примере для удобства используется целочисленный массив в качестве источника данных; однако те же понятия применяются и к другим источникам данных. Этот пример упоминается на протяжении всей оставшейся части этой статьи.

C#

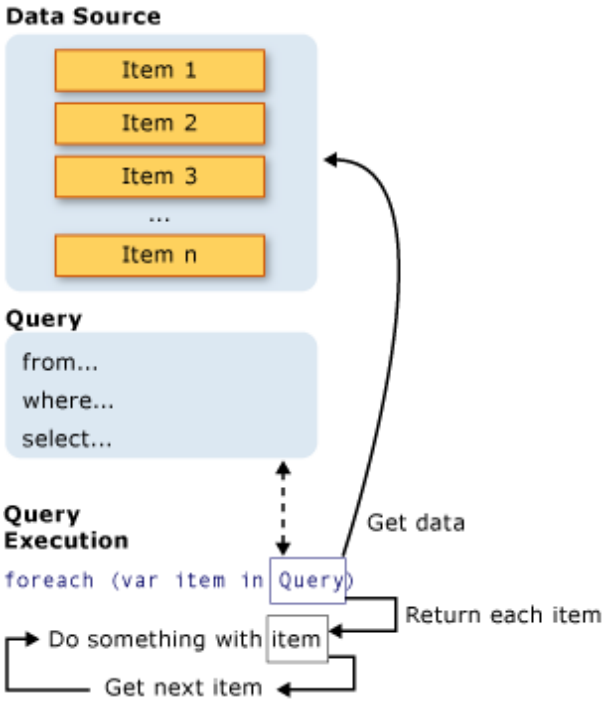
```
// The Three Parts of a LINQ Query:
// 1. Data source.
int[] numbers = [ 0, 1, 2, 3, 4, 5, 6 ];

// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery = from num in numbers
               where (num % 2) == 0
               select num;

// 3. Query execution.
foreach (int num in numQuery)
{
```

```
    Console.Write("{0,1} ", num);
}
```

На следующем рисунке показана полная операция запроса. В LINQ выполнение запроса отличается от самого запроса. Другими словами, вы не извлекаете данные, создавая переменную запроса.



Источник данных

Источник данных в предыдущем примере представляет собой массив, который поддерживает универсальный интерфейс IEnumerable<T>. Этот факт означает, что запрос может выполняться с помощью LINQ. Запрос выполняется в инструкции `foreach`, и для `foreach` требуется IEnumerable или IEnumerable<T>. Типы, поддерживающие IEnumerable<T> или производный интерфейс, такой как универсальный IQueryable<T>, называются типами для запросов .

Запрашиваемый тип не требует изменения или специального лечения для использования в качестве источника данных LINQ. Если исходные данные еще не в памяти в качестве запрашиваемого типа, поставщик LINQ должен представлять его как таковое. Например, LINQ to XML загружает XML-документ в запрашиваемый тип XElement:

C#

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

С помощью EntityFrameworkсоздается реляционное сопоставление объектов между классами C# и схемой базы данных. Вы пишете запросы по объектам, а во время выполнения Entity Framework обрабатывает взаимодействие с базой данных. В следующем примере `Customers` представляет определенную таблицу в базе данных, а тип результата запроса IQueryable<T>является производным от IEnumerable<T>.

```
C#

Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

Дополнительные сведения о создании определенных типов источников данных см. в документации для различных поставщиков LINQ. Однако базовое правило является простым: источник данных LINQ — это любой объект, поддерживающий универсальный интерфейс IEnumerable<T>, или интерфейс, наследуемый от него, обычно IQueryable<T>.

> ⓘ **Примечание**
>
> Такие типы, как ArrayList, поддерживающие не универсальный интерфейс IEnumerable, также можно использовать в качестве источника данных LINQ. Дополнительные сведения см. в статье Запрос массива с помощьюLINQ (C#).

Запрос

Запрос указывает, какие сведения следует извлечь из источника данных или источников. При необходимости запрос также указывает, как следует отсортировать, сгруппировать и сформировать данные перед возвратом. Запрос хранится в переменной запроса и инициализирован с помощью выражения запроса. Вы используете синтаксис запросов C# для написания запросов.

Запрос в предыдущем примере возвращает все четные числа из целочисленного массива. Выражение запроса содержит три предложения: `from`, `where`и `select`. (Если вы знакомы с SQL, вы заметили, что порядок предложений отменяется от порядка в SQL.) Предложение `from` указывает источник данных, предложение `where` применяет фильтр, а предложение `select` указывает тип возвращаемых элементов. Все условия запросов подробно рассматриваются в этом разделе. Сейчас важно,

чтобы в LINQ переменная запроса не принимает никаких действий и не возвращает данные. Он просто сохраняет сведения, необходимые для получения результатов при выполнении запроса в какой-то момент. Дополнительные сведения о том, как создаются запросы, см. в обзоре стандартных операторов запросов (C#) .

> ⓘ **Примечание**
>
> Запросы также можно выразить с помощью синтаксиса метода. Дополнительные сведения см. в разделе **Синтаксис запросов и синтаксис метода в LINQ**.

Классификация стандартных операторов запросов по способу выполнения

Реализация методов стандартных операторов запроса LINQ to Objects выполняется одним из двух основных способов: *немедленный* или *отложенный*. Операторы запросов, использующие отложенное выполнение, можно также разделить на две категории: *потоковая* и *непотоковая*.

Немедленно

Немедленное выполнение означает, что источник данных считывается и операция выполняется один раз. Все стандартные операторы запросов, возвращающие скалярный результат, выполняются немедленно. Примерами таких запросов являются `Count`, `Max`, `Average` и `First`. Эти методы выполняются без явной инструкции `foreach`, так как сам запрос должен использовать `foreach` для возврата результата. Эти запросы возвращают одно значение, а не коллекцию `IEnumerable`. Любой запрос *можно принудительно* выполнить немедленно с помощью методов Enumerable.ToList или Enumerable.ToArray. Немедленное выполнение позволяет повторно использовать результаты запроса, а не сам запрос. Результаты извлекаются один раз, а затем хранятся для дальнейшего использования. Следующий запрос возвращает количество четных чисел в исходном массиве:

C#

```
var evenNumQuery = from num in numbers
                   where (num % 2) == 0
                   select num;

int evenNumCount = evenNumQuery.Count();
```

Чтобы принудительно выполнить любой запрос и кэшировать результаты, можно вызвать методы ToList или ToArray.

C#

```
List<int> numQuery2 = (from num in numbers
                       where (num % 2) == 0
                       select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 = (from num in numbers
                 where (num % 2) == 0
                 select num).ToArray();
```

Вы также можете принудительно выполнить, поместив цикл `foreach` сразу после выражения запроса. Однако вызов `ToList` или `ToArray` также позволяет кэшировать все данные в единый объект коллекции.

Действие отложено

Отложенное выполнение означает, что операция не выполняется в точке кода, в котором объявлен запрос. Операция выполняется только при перечислении переменной запроса, например с помощью инструкции `foreach`. Результаты выполнения запроса зависят от содержимого источника данных при выполнении запроса, а не при определении запроса. Если переменная запроса перечисляется несколько раз, результаты могут отличаться каждый раз. Почти все стандартные операторы запросов, тип возвращаемого результата которых IEnumerable<T> или IOrderedEnumerable<TElement>, выполняются в отложенном режиме. Отложенное выполнение обеспечивает повторное использование запроса, так как запрос получает обновленные данные из источника данных при каждом итерации результатов запроса. В следующем коде показан пример отложенного выполнения:

C#

```
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

Оператор `foreach` также определяет, где извлекаются результаты запроса. Например, в предыдущем запросе переменная итерации `num` содержит каждое значение (по одному за раз) в возвращаемой последовательности.

Так как переменная запроса никогда не содержит результаты запроса, ее можно многократно выполнять для получения обновленных данных. Например, отдельное приложение может постоянно обновлять базу данных. В *приложении* можно создать один запрос, который извлекает последние данные, и его можно выполнить через интервалы, чтобы получить обновленные результаты.

Операторы запросов, использующие отложенное выполнение, можно дополнительно классифицировать как потоковые или непотоковые.

Стриминг

Операторы потоковой передачи не должны считывать все исходные данные, прежде чем они получают элементы. Во время выполнения оператор потоковой передачи выполняет свою операцию над каждым исходным элементом по мере считывания и возвращает элемент, если это уместно. Оператор потоковой передачи продолжает считывать исходные элементы до тех пор, пока не будет создан результирующий элемент. Это означает, что для получения одного результирующего элемента могут быть прочитаны несколько исходных элементов.

Нестриминговое

Непотоковые операторы должны считывать все исходные данные, чтобы получить результирующий элемент. Такие операции, как сортировка или группирование, попадают в эту категорию. Во время выполнения непотоковые операторы запросов считывают все исходные данные, помещают его в структуру данных, выполняют операцию и дают результирующий элемент.

Таблица классификации

В следующей таблице классифицируется каждый стандартный метод оператора запроса в соответствии с методом выполнения.

> **Примечание**
>
> Если оператор помечается двумя столбцами, в операции участвуют две входные последовательности, и каждая последовательность оценивается по-разному. В таких случаях это всегда первая последовательность в списке параметров, которая оценивается в отложенном, потоковом режиме.

Развернуть таблицу

Стандартный оператор запроса	Тип возвращаемого значения	Немедленное выполнение	Отложенное выполнение потоковой передачи	Отложенное выполнение задач без потоковой передачи
Aggregate	`TSource`	✓		
All	Boolean	✓		
Any	Boolean	✓		
AsEnumerable	IEnumerable<T>		✓	
Average	Одно числовое значение	✓		
Cast	IEnumerable<T>		✓	
Concat	IEnumerable<T>		✓	
Contains	Boolean	✓		
Count	Int32	✓		
DefaultIfEmpty	IEnumerable<T>		✓	
Distinct	IEnumerable<T>		✓	
ElementAt	`TSource`	✓		
ElementAtOrDefault	`TSource?`	✓		
Empty	IEnumerable<T>	✓		
Except	IEnumerable<T>		✓	✓
First	`TSource`	✓		
FirstOrDefault	`TSource?`	✓		
GroupBy	IEnumerable<T>			✓
GroupJoin	IEnumerable<T>		✓	✓
Intersect	IEnumerable<T>		✓	✓
Join	IEnumerable<T>		✓	✓
Last	`TSource`	✓		
LastOrDefault	`TSource?`	✓		
LongCount	Int64	✓		

Стандартный оператор запроса	Тип возвращаемого значения	Немедленное выполнение	Отложенное выполнение потоковой передачи	Отложенное выполнение задач без потоковой передачи
Max	Одно числовое значение, `TSource` или `TResult?`	✓		
Min	Одно числовое значение, `TSource` или `TResult?`	✓		
OfType	IEnumerable<T>		✓	
OrderBy	IOrderedEnumerable<TElement>			✓
OrderByDescending	IOrderedEnumerable<TElement>			✓
Range	IEnumerable<T>		✓	
Repeat	IEnumerable<T>		✓	
Reverse	IEnumerable<T>			✓
Select	IEnumerable<T>		✓	
SelectMany	IEnumerable<T>		✓	
SequenceEqual	Boolean	✓		
Single	`TSource`	✓		
SingleOrDefault	`TSource?`	✓		
Skip	IEnumerable<T>		✓	
SkipWhile	IEnumerable<T>		✓	
Sum	Одно числовое значение	✓		
Take	IEnumerable<T>		✓	
TakeWhile	IEnumerable<T>		✓	
ThenBy	IOrderedEnumerable<TElement>			✓
ThenByDescending	IOrderedEnumerable<TElement>			✓
ToArray	массив `TSource[]`	✓		
ToDictionary	Dictionary<TKey,TValue>	✓		
ToList	IList<T>	✓		

Стандартный оператор запроса	Тип возвращаемого значения	Немедленное выполнение	Отложенное выполнение потоковой передачи	Отложенное выполнение задач без потоковой передачи
ToLookup	ILookup<TKey,TElement>	✓		
Union	IEnumerable<T>		✓	
Where	IEnumerable<T>		✓	

Объекты LINQ

"LINQ to Objects" ссылается на использование запросов LINQ с любой коллекцией IEnumerable или IEnumerable<T> напрямую. Вы можете использовать LINQ для запроса любых перечисляемых коллекций, таких как List<T>, Arrayили Dictionary<TKey,TValue>. Коллекция может быть определяемой пользователем или типом, возвращаемым API .NET. В подходе LINQ вы пишете декларативный код, описывающий то, что требуется получить. LINQ to Objects предоставляет отличные общие сведения о программировании с помощью LINQ.

Запросы LINQ предлагают три основных преимущества по сравнению с традиционными циклами `foreach`:

- Они более краткие и читаемые, особенно при фильтрации нескольких условий.
- Они предоставляют мощные возможности фильтрации, упорядочивания и группировки с минимальным кодом приложения.
- Их можно перенести в другие источники данных без изменений.

Чем сложнее операция, которую вы хотите выполнить с данными, тем больше преимуществ вы понимаете, используя LINQ вместо традиционных методов итерации.

Хранение результатов запроса в памяти

В основном запрос — это набор инструкций по получению и упорядочению данных. Запросы выполняются лениво, так как каждый последующий элемент в результате запрашивается. При использовании `foreach` для итерации результатов элементы возвращаются по мере доступа. Чтобы оценить запрос и сохранить результаты без выполнения цикла `foreach`, просто вызовите один из следующих методов в переменной запроса:

- ToList
- ToArray
- ToDictionary
- ToLookup

Возвращаемый объект коллекции следует назначить новой переменной при хранении результатов запроса, как показано в следующем примере:

```
C#
```

```
List<int> numbers = [ 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ];

IEnumerable<int> queryFactorsOfFour = from num in numbers
                                      where num % 4 == 0
                                      select num;

// Store the results in a new variable
// without executing a foreach loop.
var factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
data.
Console.WriteLine(factorsofFourList[2]);
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

См. также

- Пошаговое руководство: написание запросов на языке C#
- foreach в
- Ключевые слова Запросов (LINQ)

Основы выражения запроса

Статья • 16.01.2025

В этой статье представлены основные понятия, связанные с выражениями запросов в C#.

Что такое запрос и для чего он нужен?

запроса — это набор инструкций, описывающих, какие данные следует извлекать из заданного источника данных (или источников), а также какие формы и организации должны иметь возвращаемые данные. Запрос отличается от результатов, которые он создает.

Как правило, исходные данные упорядочены логически как последовательность элементов одного типа. Например, таблица базы данных SQL содержит последовательность строк. В XML-файле есть "последовательность" XML-элементов (хотя XML-элементы организованы иерархически в структуре дерева). Коллекция в памяти содержит последовательность объектов.

С точки зрения приложения конкретный тип и структура исходных исходных данных не важна. Приложение всегда видит исходные данные как коллекцию IEnumerable<T> или IQueryable<T>. Например, в xml-коде LINQ to XML исходные данные отображаются как `IEnumerable`<XElement>.

С учетом этой исходной последовательности запрос может выполнять одно из трех действий:

- Получение подмножества элементов для создания новой последовательности без изменения отдельных элементов. Затем запрос может сортировать или группировать возвращаемую последовательность различными способами, как показано в следующем примере (предположим, что `scores` является `int[]`):

  C#

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Получите последовательность элементов, как в предыдущем примере, но преобразуйте их в новый тип объекта. Например, запрос может получить только имена семейств из определенных записей клиента в источнике данных. Или он может получить полную запись, а затем использовать ее для создания другого типа объекта в памяти или даже XML-данных перед созданием конечной последовательности результатов. В следующем примере показана проекция от `int` к `string`. Обратите внимание на новый тип `highScoresQuery`.

  ```
  C#

  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- Извлеките одноэлементное значение о исходных данных, например:

  - Количество элементов, соответствующих определенному условию.
  - Элемент, имеющий наибольшее или наименьшее значение.
  - Первый элемент, соответствующий условию или сумме определенных значений в указанном наборе элементов. Например, следующий запрос возвращает количество показателей, превышающее 80 из массива целых чисел `scores`:

    ```
    C#

    var highScoreCount = (
        from score in scores
        where score > 80
        select score
    ).Count();
    ```

    В предыдущем примере обратите внимание на использование круглых скобок вокруг выражения запроса перед вызовом метода Enumerable.Count. Можно также использовать новую переменную для хранения конкретного результата.

    ```
    C#

    IEnumerable<int> highScoresQuery3 =
        from score in scores
        where score > 80
        select score;
    ```

```
var scoreCount = highScoresQuery3.Count();
```

В предыдущем примере запрос выполняется в вызове `Count`, потому что `Count` должен итерировать результаты, чтобы выяснить, сколько элементов возвращает `highScoresQuery`.

Что такое выражение запроса?

Выражение запроса — это запрос, выраженный в синтаксисе запроса. Выражение запроса — это конструкция языка первого класса. Он похож на любое другое выражение и может использоваться в любом контексте, в котором допустимо выражение C#. Выражение запроса состоит из набора предложений, написанных в декларативном синтаксисе, аналогичном SQL или XQuery. Каждое предложение, в свою очередь, содержит одно или несколько выражений C#, и эти выражения могут быть либо выражением запроса, либо содержать выражение запроса.

Выражение запроса должно начинаться с из предложения и должно заканчиваться предложением выбрать или группу . Между первым предложением `from` и последним предложением `select` или `group` оно может содержать одно или несколько из этих необязательных предложений: , где, orderby, присоединиться, позволить и даже другим из предложений. Вы также можете использовать в качестве ключевого слова, чтобы результат условия `join` или `group` служил источником для других условий запроса в том же выражении запроса.

Переменная запроса

В LINQ переменная запроса — это любая переменная, которая хранит запроса вместо результатов запроса. В частности, переменная запроса всегда является перечислимым типом, который создает последовательность элементов при итерации в инструкции `foreach` или прямом вызове метода IEnumerator.MoveNext().

> **Примечание**
>
> Примеры в этой статье используют следующий источник данных и примерные данные.

```
C#
```

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

В следующем примере кода показан простое выражение запроса с одним источником данных, одним предложением фильтрации, одним предложением упорядочения и без преобразования исходных элементов. Предложение `select` заканчивает запрос.

C#

```
// Data source.
int[] scores = [90, 71, 82, 93, 75, 82];

// Query Expression.
IEnumerable<int> scoreQuery = //query variable
    from score in scores //required
    where score > 80 // optional
    orderby score descending // optional
    select score; //must end with select or group

// Execute the query to produce the results
foreach (var testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82
```

В предыдущем примере `scoreQuery` — это переменная запроса ,, которая иногда упоминается просто как *запроса*. Переменная запроса не хранит фактические данные результатов, которые создаются в цикле `foreach`. При выполнении инструкции `foreach` результаты запроса не возвращаются с помощью переменной запроса `scoreQuery`. Скорее, они возвращаются через переменную итерации `testScore`. Переменная `scoreQuery` может быть итерирована во втором цикле `foreach`. Он создает те же результаты, пока ни он, ни источник данных не были изменены.

Переменная запроса может хранить запрос, выраженный в синтаксисе запроса или синтаксисе метода, или сочетание двух. В следующих примерах `queryMajorCities` и `queryMajorCities2` являются переменными запроса:

C#

```
City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000)
];

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 30_000_000
    select city;

// Execute the query to produce the results
foreach (City city in queryMajorCities)
```

```
{
    Console.WriteLine(city);
}

// Output:
// City { Name = Tokyo, Population = 37833000 }
// City { Name = Delhi, Population = 30290000 }

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
30_000_000);
// Execute the query to produce the results
foreach (City city in queryMajorCities2)
{
    Console.WriteLine(city);
}
// Output:
// City { Name = Tokyo, Population = 37833000 }
// City { Name = Delhi, Population = 30290000 }
```

С другой стороны, в следующих двух примерах показаны переменные, которые не являются переменными запроса, даже если каждый из них инициализирован с помощью запроса. Они не являются переменными запроса, так как они хранят результаты:

C#

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
var largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
```

```
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
var largeCitiesList2 = largeCitiesQuery.ToList();
```

Явная и неявная типизация переменных запроса

Эта документация обычно предоставляет явный тип переменной запроса, чтобы отобразить связь типа между переменной запроса и предложением выбора . Однако можно также использовать ключевое слово var , чтобы указать компилятору определить тип переменной запроса (или любой другой локальной переменной) во время компиляции. Например, пример запроса, показанный ранее в этой статье, также можно выразить с помощью неявного ввода:

C#

```
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

В предыдущем примере использование var является необязательным. `queryCities` — это `IEnumerable<City>`, типизированный неявно или явно.

Запуск выражения запроса

Выражение запроса должно начинаться с предложения `from`. Он задает источник данных вместе с переменной диапазона. Переменная диапазона представляет каждый последовательный элемент в исходной последовательности по мере прохождения исходной последовательности. Переменная диапазона строго типизирована на основе типа элементов в источнике данных. В следующем примере, так как `countries` является массивом объектов `Country`, переменная диапазона также вводится как `Country`. Так как переменная диапазона строго типизирована, оператор dot можно использовать для доступа к любым доступным членам типа.

C#

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
```

```
    where country.Area > 20 //sq km
    select country;
```

Переменная диапазона остаётся в области до тех пор, пока запрос не завершится точкой с запятой или предложением продолжения.

Выражение запроса может содержать несколько условий `from`. Используйте дополнительные предложения `from`, если каждый элемент в исходной последовательности является коллекцией или содержит коллекцию. Например, предположим, что у вас есть коллекция объектов `Country`, каждая из которых содержит коллекцию объектов `City` с именем `Cities`. Чтобы запросить объекты `City` в каждом `Country`, используйте два предложения `from`, как показано ниже:

C#

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Дополнительные сведения см. в пункта.

Завершение выражения запроса

Выражение запроса должно заканчиваться предложением `group` или предложением `select`.

Групповая клаузула

Используйте предложение `group` для создания последовательности групп, упорядоченных по заданному ключу. Ключ может быть любым типом данных. Например, следующий запрос создает последовательность групп, содержащих один или несколько объектов `Country`, и ключ которого является типом `char` со значением, которое является первой буквой названий стран.

C#

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Дополнительные сведения о группировке см. в пункте группы.

Предложение select

Используйте предложение `select` для создания всех других типов последовательностей. Простое предложение `select` просто создает последовательность объектов того же типа, что и объекты, содержащиеся в источнике данных. В этом примере источник данных содержит объекты `Country`. Предложение `orderby` просто сортирует элементы в новый порядок, а предложение `select` создает последовательность переупорядоченных `Country` объектов.

C#

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

Предложение `select` можно использовать для преобразования исходных данных в последовательности новых типов. Это преобразование также называется проекцией . В следующем примере предложение `select` *проецирует* последовательность анонимных типов, которая содержит только подмножество полей в исходном элементе. Новые объекты инициализированы с помощью инициализатора объектов.

C#

```
var queryNameAndPop =
    from country in countries
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };
```

Поэтому в этом примере требуется var, так как запрос создает анонимный тип.

Дополнительные сведения обо всех способах использования предложения `select` для преобразования исходных данных см. в предложении select.

Продолжение с *до*

Ключевое слово `into` можно использовать в предложении `select` или `group` для создания временного идентификатора, включающего запрос. Используйте

предложение `into`, если необходимо выполнить дополнительные операции запроса после группировки или выборки. В следующем примере `countries` группируются в соответствии с населением в диапазоне от 10 миллионов. После создания этих групп дополнительные условия фильтруют некоторые из них, а затем упорядочивают группы в порядке возрастания. Для выполнения этих дополнительных операций требуется продолжение, представленное `countryGroup`.

C#

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int)country.Population / 1_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
    {
        Console.WriteLine(country.Name + ":" + country.Population);
    }
}
```

Для получения дополнительной информации см. раздел по теме.

Фильтрация, упорядочивание и присоединение

Между начальным предложением `from` и конечным предложением `select` или `group` все остальные предложения (`where`, `join`, `orderby`, `from`, `let`) являются необязательными. Любой из необязательных предложений может использоваться ноль раз или несколько раз в тексте запроса.

Условие WHERE

Используйте предложение `where` для фильтрации элементов из исходных данных на основе одного или нескольких выражений предиката. Предложение `where` в следующем примере содержит один предикат с двумя условиями.

C#

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population is < 15_000_000 and > 10_000_000
    select city;
```

Для получения дополнительной информации см. раздел , где упоминается условие.

Предложение orderby

Используйте предложение `orderby` для сортировки результатов в порядке возрастания или убывания. Вы также можете указать вторичные порядки сортировки. В следующем примере выполняется первичная сортировка объектов `country` с помощью свойства `Area`. Затем он выполняет вторичную сортировку с помощью свойства `Population`.

C#

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

Ключевое слово `ascending` является необязательным; Это порядок сортировки по умолчанию, если порядок сортировки не указан. Дополнительные сведения см. в предложении orderby .

Предложение join

Используйте предложение `join` для связывания и (или) объединения элементов из одного источника данных с элементами из другого источника данных на основе сравнения между указанными ключами в каждом элементе. В LINQ операции соединения выполняются на последовательностях объектов, элементы которых являются разными типами. После объединения двух последовательностей необходимо использовать инструкцию `select` или `group`, чтобы указать, какой элемент будет храниться в выходной последовательности. Можно также использовать анонимный тип для объединения свойств из каждого набора связанных элементов в новый тип для выходной последовательности. В следующем примере сопоставляются объекты `prod`, чьё свойство `Category` соответствует одной из категорий в строковом массиве `categories`. Продукты, `Category` которых не соответствуют ни одной строке в `categories`, отфильтровываются. Оператор `select` проектирует новый тип, свойства которого взяты как из `cat`, так и из `prod`.

C#

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new
    {
        Category = cat,
        Name = prod.Name
    };
```

Вы также можете выполнить присоединение к группе, сохраняя результаты операции `join` во временную переменную с помощью с ключевым словом. Для получения дополнительных сведений см. предложение соединения .

Конструкция let

Используйте предложение `let` для хранения результата выражения, например вызова метода, в новой переменной диапазона. В следующем примере переменная диапазона `firstName` сохраняет первый элемент массива строк, возвращаемых `Split`.

C#

```
string[] names = ["Svetlana Omelchenko", "Claire O'Donnell", "Sven
Mortensen", "Cesar Garcia"];
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (var s in queryFirstNames)
{
    Console.Write(s + " ");
}

//Output: Svetlana Claire Sven Cesar
```

Дополнительные сведения см. в предложении 'let' .

Подзапросы в выражении запроса

Предложение запроса может содержать выражение запроса, которое иногда называется *внутренним запросом*. Каждый вложенный запрос начинается с собственного предложения `from`, которое не обязательно указывает на тот же

источник данных в первом предложении `from`. Например, следующий запрос показывает выражение запроса, которое используется в инструкции select для получения результатов операции группировки.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
```

Дополнительную информацию см. в теме Выполнение подзапроса для операции группировки.

См. также

- ключевые слова запросов (LINQ)
- Обзор стандартных операторов запросов

Запись запросов LINQ C# для запроса данных

Большинство запросов в документации по встроенному языку (LINQ) используют синтаксис декларативного запроса LINQ. Компилятор C# преобразует синтаксис запросов в вызовы методов. Эти вызовы метода реализуют стандартные операторы запросов. У них есть такие имена, как `Where`, `Select`, `GroupBy`, `Join`, `Max` и `Average`. Их можно вызывать непосредственно с помощью синтаксиса метода вместо синтаксиса запроса.

Синтаксис запросов и синтаксис методов семантически идентичны, но синтаксис запросов часто проще и проще читать. В качестве вызовов метода необходимо выразить некоторые запросы. Например, необходимо использовать вызов метода для выражения запроса, который возвращает число элементов, соответствующих указанным критериям. Вызов метода также необходимо использовать для запроса, который получает элемент с максимальным значением в исходной последовательности. В справочной документации по стандартным операторам запросов в пространствах имен System.Linq обычно применяется синтаксис методов. Узнайте, как использовать синтаксис метода в запросах и выражениях запросов.

Стандартные методы расширения оператора запросов

В следующем примере показано простое *выражение запроса* и семантически эквивалентный ему запрос, написанный как *запрос, основанный на методе*.

C#

```
int[] numbers = [ 5, 10, 8, 3, 6, 12 ];

//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

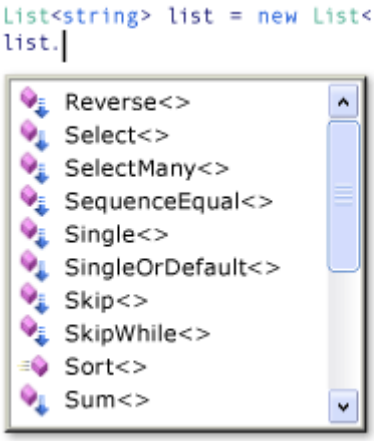
//Method syntax:
IEnumerable<int> numQuery2 = numbers
    .Where(num => num % 2 == 0)
    .OrderBy(n => n);

foreach (int i in numQuery1)
{
    Console.Write(i + " ");
```

```
}
Console.WriteLine(System.Environment.NewLine);
foreach (int i in numQuery2)
{
    Console.Write(i + " ");
}
```

Оба примера дают одинаковый результат. Тип переменной запроса одинаков в обеих формах: IEnumerable<T>.

В правой части выражения обратите внимание, что `where` предложение теперь выражается как метод экземпляра объекта `numbers` , который имеет тип `IEnumerable<int>`. Если вы знакомы с универсальным IEnumerable<T> интерфейсом `Where` , вы знаете, что у него нет метода. Однако при вызове списка завершения IntelliSense в интегрированной среде разработки Visual Studio вы увидите не только `Where` метод, но и многие другие методы, такие как `Select`, `SelectMany` `Join` и `Orderby`. Эти методы реализуют стандартные операторы запросов.



Хотя он выглядит так, как будто IEnumerable<T> включает в себя больше методов, это не так. Стандартные операторы запросов реализуются как *методы* расширения. Члены расширения "расширяют" существующий тип; их можно вызывать так, как если бы они были членами типа. Стандартные операторы запросов расширяют возможности IEnumerable<T>, поэтому вы можете написать `numbers.Where(...)`. Вы вводите расширения в область видимости с помощью директив `using` перед их вызовом.

Для получения дополнительных сведений о членах расширения смотрите в разделе "Члены расширения". Дополнительные сведения о стандартных операторах запросов см. в разделе Общие сведения о стандартных операторах запроса (C#). Некоторые поставщики LINQ, такие как Entity Framework и LINQ to XML, реализуют собственные стандартные операторы запросов и члены расширения для других типов, кроме IEnumerable<T>.

Лямбда-выражения

В предыдущем примере условное выражение (`num % 2 == 0`) передается как встроенный аргумент методу Enumerable.Where: `Where(num => num % 2 == 0).` это выражение является лямбда-выражением. Это удобный способ написания кода, который в противном случае должен быть написан в более сложной форме. `num` Слева от оператора — входная переменная, соответствующая `num` выражению запроса. Компилятор может вывести тип `num`, поскольку известно, что `numbers` является универсальным типом IEnumerable<T>. Текст лямбда-выражения совпадает с выражением в синтаксисе запроса или в любом другом выражении или операторе C#. Он может включать вызовы методов и другую сложную логику. Возвращаемое значение является результатом выражения. В синтаксисе метода можно выразить только определенные запросы, а некоторые из этих запросов требуют лямбда-выражений. Лямбда-выражения — это мощный и гибкий инструмент на панели элементов LINQ.

Возможность создания запросов

В приведенном выше примере кода метод Enumerable.OrderBy вызывается с помощью оператора dot в вызове `Where`. `Where` создает отфильтрованную последовательность, а затем `Orderby` сортирует последовательность, созданную `Where`. Поскольку запросы возвращают `IEnumerable`, объедините их в синтаксис метода, собрав вызовы методов в цепочку. Компилятор выполняет эту композицию при написании запросов с помощью синтаксиса запросов. Так как переменная запроса не хранит результаты запроса, ее можно изменить или использовать в качестве основы для нового запроса в любое время, даже после его выполнения.

В следующих примерах показаны некоторые базовые запросы LINQ с помощью каждого подхода, указанного ранее.

> **Заметка**
>
> Эти запросы работают с коллекциями в памяти; Однако синтаксис идентичен синтаксису, используемому в сущностях LINQ to Entities и LINQ to XML.

Пример — синтаксис запросов

Пишите большинство запросов с *синтаксисом запроса* для создания *выражений запроса*. В следующем примере показаны три выражения запросов. Первое выражение запроса показывает, как фильтровать или ограничивать результаты, применяя условия в

предложении `where`. Оно возвращает все элементы исходной последовательности, значения которых больше 7 или меньше 3. Второе выражение показывает, как сортировать возвращаемые результаты. Третье выражение показывает, как группировать результаты по ключу. Этот запрос возвращает две группы на основе первой буквы слова.

C#

```
List<int> numbers = [ 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 ];

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num is < 3 or > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num is < 3 or > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = ["carrots", "cabbage", "broccoli", "beans", "barley"];
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Тип запросов .IEnumerable<T> Все эти запросы можно написать, var как показано в следующем примере:

```
var query = from num in numbers...
```

В каждом предыдущем примере запросы фактически не выполняются, пока не выполняется итерацию по переменной запроса в foreach инструкции или другой инструкции.

Пример — синтаксис метода

В качестве вызова метода необходимо выразить некоторые операции запроса. Наиболее распространенными такими методами являются те методы, которые возвращают однотонные числовые значения, такие как Sum, Max, Minи Averageт. д. Вызывайте эти методы последними в любом запросе, потому что они возвращают одно значение и не могут быть использованы в качестве источника для дополнительных операций запроса. В следующем примере показан вызов метода в выражении запроса:

C#

```
List<int> numbers1 = [ 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 ];
List<int> numbers2 = [ 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 ];

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Если метод имеет System.Action или System.Func<TResult> параметры, укажите эти аргументы в виде лямбда-выражения, как показано в следующем примере:

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

В предыдущих запросах выполняется только запрос #4, так как возвращает одно значение, а не универсальную IEnumerable<T> коллекцию. Сам метод использует `foreach` или аналогичный код для вычисления значения.

Вы можете написать каждый из предыдущих запросов с помощью неявного varввода, как показано в следующем примере:

C#

```
// var is used for convenience in these queries
double average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Пример : синтаксис смешанного запроса и метода

В этом примере показано, как применять синтаксис метода к результатам предложения запроса. Заключите выражение запроса в скобки, а затем примените оператор dot и вызовите метод. В следующем примере запрос 7 возвращает количество чисел со значением от 3 до 7.

C#

```
// Query #7.

// Using a query expression with method syntax
```

```
var numCount1 = (
    from num in numbers1
    where num is > 3 and < 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num is > 3 and < 7
    select num;

var numCount2 = numbersQuery.Count();
```

Поскольку запрос 7 возвращает одно значение, а не коллекцию, он выполняется незамедлительно.

Предыдущий запрос можно написать с помощью неявного `var` ввода следующим образом:

C#

```
var numCount = (from num in numbers...
```

Его можно написать в синтаксисе метода следующим образом:

C#

```
var numCount = numbers.Count(n => n is > 3 and < 7);
```

Его можно написать с помощью явного ввода, как показано ниже.

C#

```
int numCount = numbers.Count(n => n is > 3 and < 7);
```

Динамическое определение фильтров предикатов во время выполнения

В некоторых случаях количество предикатов, которые нужно применить к исходным элементам в предложении `where`, неизвестно вплоть до времени выполнения. Одним из способов динамического определения сразу нескольких фильтров предикатов служит метод Contains, как показано в следующем примере. Запрос возвращает различные результаты в зависимости от значения `id` выполнения запроса.

C#

```
int[] ids = [ 111, 114, 112 ];

var queryNames = from student in students
                 where ids.Contains(student.ID)
                 select new
                 {
                     student.LastName,
                     student.ID
                 };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Garcia: 114
    O'Donnell: 112
    Omelchenko: 111
 */

// Change the ids.
ids = [ 122, 117, 120, 115 ];

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Adams: 120
    Feng: 117
    Garcia: 115
    Tucker: 122
 */
```

> ⓘ **Заметка**
>
> В этом примере используются следующие источник и данные:

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City> Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City", 826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro", 28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

Можно использовать инструкции потока управления, такие как `if... else` или `switch`, чтобы выбрать один из предопределенных альтернативных запросов. В следующем примере используется другое `studentQuery` предложение, `where` если значение `oddYear` времени выполнения равно `true` или `false`.

C#

```
void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery = oddYear
        ? (from student in students
           where student.Year is GradeLevel.FirstYear or GradeLevel.ThirdYear
           select student)
        : (from student in students
           where student.Year is GradeLevel.SecondYear or GradeLevel.FourthYear
           select student);
    var descr = oddYear ? "odd" : "even";
```

```
    Console.WriteLine($"The following students are at an {descr} year level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
    The following students are at an odd year level:
    Fakhouri: 116
    Feng: 117
    Garcia: 115
    Mortensen: 113
    Tucker: 119
    Tucker: 122
 */

FilterByYearType(false);

/* Output:
    The following students are at an even year level:
    Adams: 120
    Garcia: 114
    Garcia: 118
    O'Donnell: 112
    Omelchenko: 111
    Zabokritski: 121
 */
```

Обработка значений NULL в выражениях запросов

В этом примере показано, как обрабатывать возможные нулевые значения в исходных коллекциях. Коллекция объектов, например IEnumerable<T>, может содержать элементы со значением NULL. Если исходная коллекция содержит элемент, значение которого равно `null` `null`, и запрос не обрабатывает `null` значения, NullReferenceException при выполнении запроса возникает исключение.

В следующем примере используются эти типы и статические массивы данных:

C#

```
record Product(string Name, int CategoryID);
record Category(string Name, int ID);
```

C#

```
static Category?[] categories =
[
    new ("brass", 1),
    null,
    new ("winds", 2),
    default,
    new ("percussion", 3)
];

static Product?[] products =
[
    new Product("Trumpet", 1),
    new Product("Trombone", 1),
    new Product("French Horn", 1),
    null,
    new Product("Clarinet", 2),
    new Product("Flute", 2),
    null,
    new Product("Cymbal", 3),
    new Product("Drum", 3)
];
```

Чтобы избежать исключения в связи с пустой ссылкой, можно составить защитный код, как показано в следующем примере:

C#

```
var query1 = from c in categories
             where c != null
             join p in products on c.ID equals p?.CategoryID
             select new
             {
                 Category = c.Name,
                 Name = p.Name
             };
```

В предыдущем примере предложение `where` отфильтровывает все пустые элементы в последовательности категорий. Этот метод не связан с проверкой на наличие пустых значений в предложении join. Условное выражение со значением NULL в этом примере работает, поскольку `Products.CategoryID` имеет тип `int?`, что является сокращением `Nullable<int>`.

Если в предложении join только один из ключей сравнения имеет тип, допускающий значение NULL, остальные типы в выражении запроса можно привести к типу, допускающему значение NULL. В следующем примере предполагается, что `EmployeeID` — это столбец, содержащий значения типа `int?`:

C#

```
var query =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID
    select new { o.OrderID, e.FirstName };
```

В каждом из примеров в запросе используется ключевое слово `equals`. Вы также можете использовать сопоставление шаблонов, включая шаблоны для `is null` и `is not null`. Эти шаблоны не рекомендуется использовать в запросах LINQ, так как поставщики запросов могут неправильно интерпретировать новый синтаксис C#. Поставщик запросов — это библиотека, которая преобразует выражения запросов C# в собственный формат данных, например Entity Framework Core. Поставщики запросов реализуют интерфейс System.Linq.IQueryProvider, чтобы создавать источники данных, реализующие интерфейс System.Linq.IQueryable<T>.

Обработка исключений в выражениях запросов

Можно вызвать любой метод в контексте выражения запроса. Не вызывайте какой-либо метод в выражении запроса, который может создать побочный эффект, например изменение содержимого источника данных или исключение. В этом примере показано, как избежать создания исключений при вызове методов в выражениях запросов без нарушения общих рекомендаций .NET относительно обработки исключений. Эти рекомендации позволяют поймать определенное исключение, когда вы понимаете, почему оно выбрасывается в заданном контексте. Дополнительные сведения см. в разделе Лучшие методики обработки исключений.

В последнем примере показано, как обрабатывать случаи, когда необходимо создать исключение во время выполнения запроса.

В следующем примере показано, как переместить код обработки исключений за пределы выражения запроса. Этот способ можно рефакторинговать только в том случае, если метод не зависит от локальных переменных запроса. Проще справиться с исключениями за пределами выражения запроса.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception.
IEnumerable<int>? dataSource = null;
try
{
```

```
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
{
    // If we get here, it is safe to proceed.
    var query = from i in dataSource
                select i * i;

    foreach (var i in query)
    {
        Console.WriteLine(i.ToString());
    }
}
```

В блоке `catch (InvalidOperationException)` , приведенном в предыдущем примере, обработайте исключение (или не обрабатывайте) таким образом, как это подходит для приложения.

В некоторых случаях лучший ответ на исключение, которое возникает в запросе, может быть немедленное прекращение выполнения запроса. В следующем примере демонстрируется обработка исключений, которые могут быть созданы из тела запроса. Предположим, что `SomeMethodThatMightThrow` потенциально может вызвать исключение, которое требует остановки выполнения запроса.

Блок `try` заключает `foreach` цикл, а не сам запрос. Цикл `foreach` — это точка выполнения запроса. Исключения во время выполнения создаются при выполнении запроса. Таким образом, обработайте их в цикле `foreach` .

C#

```
// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        $"""C:\newFolder\{s}""";

// Data source.
string[] files = ["fileA.txt", "fileB.txt", "fileC.txt"];

// Demonstration query that throws.
var exceptionDemoQuery = from file in files
                         let n = SomeMethodThatMightThrow(file)
                         select n;

try
```

```
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

Перехватите любое исключение, которое вы можете вызвать, и выполните любую необходимую очистку в блоке `finally`.

См. также

- предложение where
- Запрос на основе состояния среды выполнения
- Nullable<T>
- Типы значений, допускающие значение NULL

ⓘ **Заметка:** Автор создал эту статью с помощью искусственного интеллекта. Подробнее

Last updated on 01.12.2025

Связи типов в операциях запросов LINQ (C#)

17.06.2025

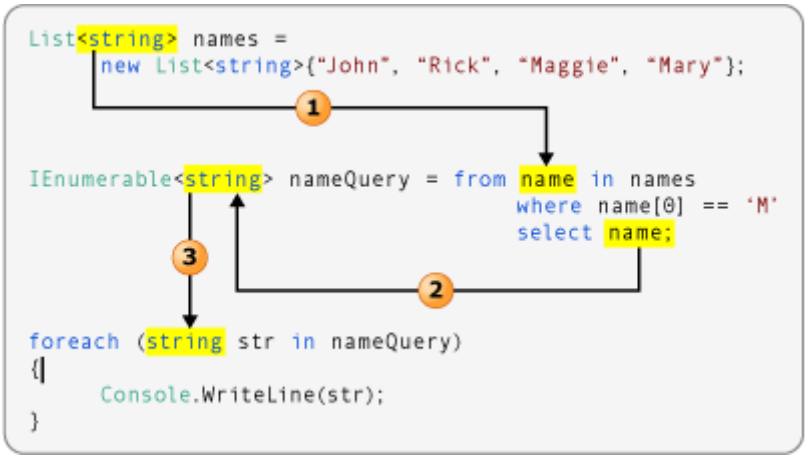
Чтобы эффективно писать запросы, следует понять, как типы переменных в полной операции запроса связаны друг с другом. Если вы понимаете эти связи, вы сможете более легко понять примеры LINQ и примеры кода в документации. Кроме того, вы поймете, что происходит при неявном типизировании переменных с помощью var.

Операции запроса LINQ строго типизированы в источнике данных, в самом запросе и в выполнении запроса. Тип переменных в запросе должен быть совместим с типом элементов в источнике данных и типом переменной итерации в инструкции foreach . Эта строгая типизация гарантирует, что ошибки типов обнаруживаются во время компиляции, когда они могут быть исправлены перед тем, как пользователи столкнутся с ними.

Для демонстрации этих связей типов большинство последующих примеров используют явное указание типов для всех переменных. В последнем примере показано, как применяются те же принципы даже при использовании неявного ввода с помощью `var`.

Запросы, не преобразующие исходные данные

На следующем рисунке показана операция запроса LINQ to Objects, которая не выполняет преобразований данных. Источник содержит последовательность строк, а выходные данные запроса — это также последовательность строк.

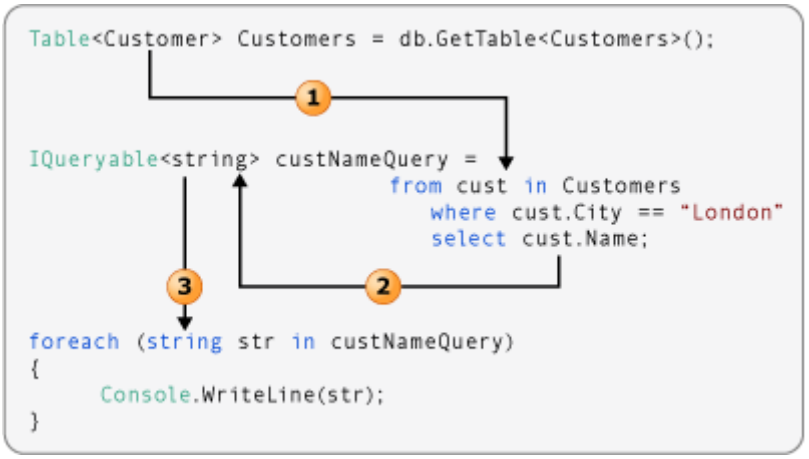


1. Аргумент типа источника данных определяет тип переменной диапазона.
2. Тип выбранного объекта определяет тип переменной запроса. `name` — это строка. Таким образом, переменная запроса является переменной `IEnumerable<string>`.

3. Переменная запроса переитерируется в инструкции `foreach` . Так как переменная запроса представляет собой последовательность строк, переменная итерации также является строкой.

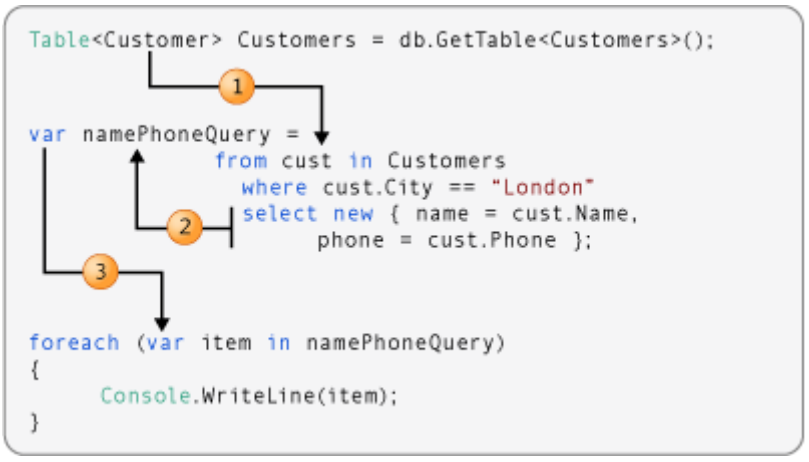
Запросы, которые преобразуют исходные данные

На следующем рисунке показана операция запроса LINQ to SQL, которая выполняет простое преобразование данных. Запрос принимает на вход последовательность объектов `Customer` и выбирает только свойство `Name` в результате. Так как `Name` это строка, запрос создает последовательность строк в виде выходных данных.



1. Аргумент типа источника данных определяет тип переменной диапазона.
2. Инструкция `select` возвращает `Name` свойство вместо полного `Customer` объекта. Так как `Name` это строка, аргумент `custNameQuery` типа имеет `string`значение , а не `Customer`.
3. Так как `custNameQuery` — это последовательность строк, переменная цикла `foreach` также должна быть строкой `string`.

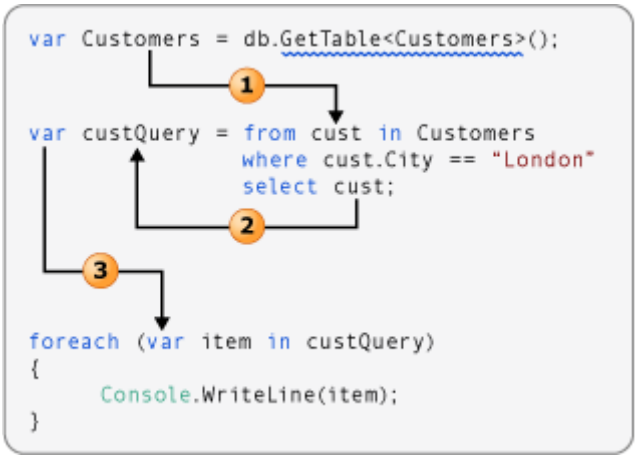
На следующем рисунке показано немного более сложное преобразование. Оператор `select` возвращает анонимный тип, который содержит только два элемента из исходного объекта `Customer`.



1. Аргумент типа источника данных всегда является типом переменной диапазона в запросе.
2. `select` Так как инструкция создает анонимный тип, переменная запроса должна быть неявно типирована с помощью `var`.
3. Поскольку тип переменной запроса неявен, переменная итерации в цикле `foreach` также должна быть неявной.

Позволяя компилятору определить информацию о типе

Хотя вы должны понимать связи типов в операции запроса, вы можете позволить компилятору выполнять всю работу. Ключевое слово var можно использовать для любой локальной переменной в операции запроса. Следующий рисунок похож на пример номер 2, который обсуждался ранее. Однако компилятор предоставляет надежный тип для каждой переменной в операции запроса.



LINQ и универсальные типы (C#)

Запросы LINQ основаны на универсальных типах. Вам не требуется глубокое знание универсальных шаблонов, прежде чем начать писать запросы. Однако вам может

потребоваться понять два основных понятия:

1. При создании экземпляра универсального класса коллекции, например List<T>, замените "T" типом объектов, которые будет храниться в списке. Например, список строк выражается как `List<string>`, а список `Customer` объектов выражается как `List<Customer>`. Универсальный список строго типизирован и предоставляет множество преимуществ по сравнению с коллекциями, которые хранят их элементы как Object. Если вы попытаетесь добавить `Customer` к `List<string>`, на этапе компиляции возникнет ошибка. Универсальные коллекции легко использовать, так как вам не нужно выполнять приведение типов во время выполнения.
2. IEnumerable<T> — интерфейс, позволяющий перечислить универсальные классы коллекции с помощью инструкции `foreach` . Универсальные классы коллекций поддерживают IEnumerable<T>, также как неуниверсальные классы коллекций, например, ArrayList, поддерживают IEnumerable.

Дополнительные сведения о универсальных типах см. в разделе Generics.

Переменные IEnumerable<T> в запросах LINQ

Переменные запроса LINQ типируются как IEnumerable<T> или производный тип, например IQueryable<T>. При отображении переменной запроса, типизированной как `IEnumerable<Customer>`, это просто означает, что запрос, когда он выполняется, будет производить последовательность из нуля или более `Customer` объектов.

C#

```
IEnumerable<Customer> customerQuery = from cust in customers
                                      where cust.City == "London"
                                      select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

Поручение компилятору обработки деклараций универсальных типов.

Если вы предпочитаете, можно избежать универсального синтаксиса с помощью ключевого слова var . Ключевое `var` слово указывает компилятору определить тип переменной запроса, глядя на источник данных, указанный в предложении `from` . В

следующем примере создается тот же скомпилированный код, что и предыдущий пример:

C#

```
var customerQuery2 = from cust in customers
                     where cust.City == "London"
                     select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

Ключевое `var` слово полезно, если тип переменной очевиден или если не так важно явно указывать вложенные универсальные типы, такие как те, которые создаются запросами группы. Как правило, рекомендуется, если вы используете `var`, понимать, что это может сделать код более сложным для других пользователей для чтения. Дополнительные сведения см. в разделе Неявно типизированные локальные переменные.

Возможности C#, поддерживающие LINQ

Выражения запросов

Выражения запросов используют декларативный синтаксис, аналогичный SQL или XQuery, для запроса по System.Collections.Generic.IEnumerable<T> коллекциям. Во время компиляции компилятор преобразует синтаксис запроса в вызовы методов к реализации стандартных методов запроса, предоставляемой поставщиком LINQ. Приложения управляют стандартными операторами запросов, имеющими область видимости, указывая соответствующее пространство имен с помощью директивы using. Следующее выражение запроса принимает массив строк, группирует их в соответствии с первым символом в строке и упорядочивает группы.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Неявно типизированные переменные (var)

Модификатор var можно использовать для указания компилятору выводить и назначать тип, как показано ниже:

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Переменные, объявленные как `var`, являются строго типизированными, так же как и переменные, тип которых указывается явно. Использование `var` позволяет создавать анонимные типы, но только для локальных переменных. Дополнительные сведения см. в разделе Неявно типизированные локальные переменные.

Инициализаторы объектов и коллекции

Инициализаторы объектов и коллекции позволяют инициализировать объекты без явного вызова конструктора для объекта. Обычно в выражениях запросов используются инициализаторы, когда они проектируют исходные данные в новый тип данных. При условии, что имеется класс с именем `Customer` и публичными свойствами `Name` и `Phone`, можно использовать инициализатор объекта, как показано в следующем коде:

C#

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Продолжая работу с классом `Customer`, предположим, что существует источник данных под названием `IncomingOrders`, и что для каждого заказа с большим значением `OrderSize` необходимо создать новый `Customer`, основанный на этом заказе. Вы можете выполнить запрос LINQ в этом источнике данных и использовать инициализацию объектов для заполнения коллекции:

C#

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

Источник данных может иметь больше свойств, определенных, чем класс `Customer`, но при инициализации объектов с `OrderSize` данные, возвращаемые из запроса, формируются в нужный тип данных. Вы выбираете только те данные, которые соответствуют вашему классу. В результате у вас теперь есть System.Collections.Generic.IEnumerable<T>, заполненный новыми `Customer`, которые вы хотели. Вы также можете написать предыдущий пример в синтаксисе метода LINQ:

C#

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y =>
new Customer { Name = y.Name, Phone = y.Phone });
```

Начиная с C# 12, можно использовать выражение коллекции для инициализации коллекции.

Дополнительные сведения можно найти здесь

- Инициализаторы объектов и коллекций
- Синтаксис выражения запроса для стандартных операторов запросов

Анонимные типы

Компилятор создает анонимный тип. Доступ к имени типа может получить только компилятор. Анонимные типы предоставляют удобный способ группировать набор свойств временно в результате запроса, не определяя отдельный именованный тип. Вы инициализируете анонимные типы с новым выражением и инициализатором объектов, как показано ниже:

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

Начиная с C# 7, можно использовать кортежи для создания неименованных типов.

Члены расширения

Член расширения является статическим членом статического класса, связанного с типом, называемым *типом приемника*. Можно вызвать член расширения, как если бы он был членом типа приемника. Эта функция позволяет добавлять новые члены в существующие типы, не изменяя их. Стандартные операторы запросов — это набор методов расширения, которые предоставляют функциональные возможности запроса LINQ для любого типа, реализующего IEnumerable<T>.

Лямбда-выражения

Лямбда-выражение — это встроенная функция, которая использует `=>` оператор для разделения входных параметров из текста функции и может быть преобразована во время компиляции в делегат или дерево выражений. При программировании LINQ при выполнении прямых вызовов методов к стандартным операторам запросов возникают лямбда-выражения.

Выражения в виде данных

Объекты запросов являются компонуемыми, то есть можно возвращать запрос из метода. Объекты, представляющие запросы, не хранят полученную коллекцию, а сохраняют этапы для получения результатов при необходимости. Преимущество возврата объектов запроса из методов заключается в том, что можно дополнительно создать или изменить их. Таким образом, любое возвращаемое значение или `out` параметр метода, возвращающего запрос, также должно иметь этот тип. Если метод материализует запрос в конкретный List<T> или Array тип, он возвращает результаты запроса вместо самого запроса. Вы по-прежнему можете создавать или изменять переменную запроса, возвращаемую из метода.

В следующем примере первый метод `QueryMethod1` возвращает запрос в качестве возвращаемого значения, а второй метод `QueryMethod2` возвращает запрос в качестве `out` параметра (`returnQ` в примере). В обоих случаях это запрос, который возвращается, а не результаты запроса.

C#

```
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ = from i in ints
              where i < 4
              select i.ToString();

int[] nums = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ];

var myQuery1 = QueryMethod1(nums);
```

`foreach` Следующий цикл выполняет запрос `myQuery1`.

C#

```
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Наведите указатель мыши на `myQuery1`, чтобы просмотреть его тип.

Вы также можете выполнить запрос, возвращенный напрямую `QueryMethod1` , без использования `myQuery1`.

C#

```
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}
```

Наведите указатель мыши на элемент `QueryMethod1`, чтобы увидеть его тип возвращаемого значения.

`QueryMethod2` возвращает запрос в качестве значения параметра `out` :

C#

```
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}
```

Запрос можно изменить с помощью композиции запросов. В этом случае предыдущий объект запроса используется для создания нового объекта запроса. Этот новый объект возвращает результаты, отличные от исходного объекта запроса.

C#

```
myQuery1 = from item in myQuery1
           orderby item descending
           select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

ⓘ **Заметка:** Автор создал эту статью с помощью искусственного интеллекта. Подробнее

Last updated on 01.12.2025

Руководство. Написание запросов в C# с помощью языкового интегрированного запроса (LINQ)

Статья • 02.04.2025

В этом руководстве вы создадите источник данных и напишите несколько запросов LINQ. Вы можете поэкспериментировать с выражениями запроса и увидеть различия в результатах. В этом пошаговом руководстве показаны функции языка C#, используемые для записи выражений запросов LINQ. Вы можете следить за процессом, создавать приложение и самостоятельно экспериментировать с запросами. В этой статье предполагается, что вы установили последний пакет SDK для .NET. В противном случае перейдите на страницу загрузки .NET и установите последнюю версию на компьютере.

Сначала создайте приложение. В консоли введите следующую команду:

Интерфейс командной строки.NET

```
dotnet new console -o WalkthroughWritingLinqQueries
```

Или, если вы предпочитаете Visual Studio, создайте консольное приложение с именем *WalkthroughWritingLinqQueries*.

Создание источника данных в памяти

Первым шагом является создание источника данных для запросов. Источник данных для запросов — это простой список `Student` записей. Каждая `Student` запись имеет имя, фамилию и массив целых чисел, который представляет их оценки тестов в классе. Добавьте новый файл *с именем students.cs* и скопируйте следующий код в этот файл:

C#

```
namespace WalkthroughWritingLinqQueries;

public record Student(string First, string Last, int ID, int[] Scores);
```

Обратите внимание на следующие характеристики:

- Запись `Student` состоит из автоматически реализованных свойств.

- Каждый студент в списке инициализируется основным конструктором.
- Последовательность оценок для каждого студента инициализируется основным конструктором.

Затем создайте последовательность записей `Student` , которые служат источником этого запроса. Откройте *Program.cs* и удалите следующий стандартный код:

```
C#

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Замените его следующим кодом, который создает последовательность записей `Student` :

```
C#

using WalkthroughWritingLinqQueries;

// Create a data source by using a collection initializer.
IEnumerable<Student> students =
[
    new Student(First: "Svetlana", Last: "Omelchenko", ID: 111, Scores: [97,
92, 81, 60]),
    new Student(First: "Claire",   Last: "O'Donnell",  ID: 112, Scores: [75,
84, 91, 39]),
    new Student(First: "Sven",     Last: "Mortensen",  ID: 113, Scores: [88,
94, 65, 91]),
    new Student(First: "Cesar",    Last: "Garcia",     ID: 114, Scores: [97,
89, 85, 82]),
    new Student(First: "Debra",    Last: "Garcia",     ID: 115, Scores: [35,
72, 91, 70]),
    new Student(First: "Fadi",     Last: "Fakhouri",   ID: 116, Scores: [99,
86, 90, 94]),
    new Student(First: "Hanying",  Last: "Feng",       ID: 117, Scores: [93,
92, 80, 87]),
    new Student(First: "Hugo",     Last: "Garcia",     ID: 118, Scores: [92,
90, 83, 78]),

    new Student("Lance",   "Tucker",      119, [68, 79, 88, 92]),
    new Student("Terry",   "Adams",       120, [99, 82, 81, 79]),
    new Student("Eugene",  "Zabokritski", 121, [96, 85, 91, 60]),
    new Student("Michael", "Tucker",      122, [94, 92, 91, 91])
];
```

- Последовательность студентов инициализируется коллекционным выражением.
- Тип `Student` записи содержит статический список всех учащихся.

- Некоторые вызовы конструктора используют именованные аргументы для уточнения того, какой аргумент соответствует параметру конструктора.

Попробуйте добавить в список студентов еще несколько учащихся с различными оценками тестов, чтобы лучше разобраться в коде.

Создание запроса

Затем создайте первый запрос. При выполнении запроса создается список всех учащихся, оценка которых на первом тесте превышает 90. Так как выбран весь `Student` объект, тип запроса .`IEnumerable<Student>` Хотя код также может использовать неявную типизацию с помощью ключевого слова var, явная типизация используется для четкой иллюстрации результатов. (Дополнительные сведения см. в `var`разделе "Неявно типизированные локальные переменные".) Добавьте следующий код в *Program.cs* после кода, создающего последовательность учащихся:

```
C#

// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

Переменная диапазона запроса `student` служит ссылкой на каждый `Student` в источнике, предоставляя доступ к членам каждого объекта.

Выполнение запроса

Теперь напишите `foreach` цикл, который приводит к выполнению запроса. Доступ к каждому элементу в возвращаемой последовательности осуществляется через переменную итерации в цикле `foreach` . Тип этой переменной — `Student`, и тип переменной запроса совместим — `IEnumerable<Student>`. После добавления следующего кода выполните сборку и запустите приложение, чтобы просмотреть результаты в окне **консоли** .

```
C#

// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
```

```
{
    Console.WriteLine($"{student.Last}, {student.First}");
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Чтобы дополнительно уточнить запрос, можно объединить несколько логических условий в предложении `where` . Следующий код добавляет условие, чтобы запрос возвращал тех учащихся, чьи первые оценки были более 90 и чьи последние оценки были менее 80. Условие `where` должно соответствовать следующему коду.

C#

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Попробуйте предыдущее `where` предложение или поэкспериментируйте с другими условиями фильтра. Для получения дополнительной информации см. раздел , где упоминается условие.

Упорядочить результаты запроса

Проще проверить результаты, если они находятся в каком-то порядке. Вы можете упорядочить возвращаемую последовательность любым доступным полем в исходных элементах. Например, следующее `orderby` предложение упорядочивает результаты в алфавитном порядке от A до Z в соответствии с именем семьи каждого учащегося. Добавьте следующее `orderby` предложение в запрос сразу после инструкции `where` и перед инструкцией `select` :

C#

```
orderby student.Last ascending
```

Теперь измените `orderby` предложение таким образом, чтобы оно упорядочивало результаты в обратном порядке по оценке на первом тесте, от самой высокой оценки к самой низкой.

C#

```
orderby student.Scores[0] descending
```

Измените `WriteLine` строку формата, чтобы увидеть оценки:

C#

```
Console.WriteLine($"{student.Last}, {student.First} {student.Scores[0]}");
```

Дополнительные сведения см. в предложении orderby .

Группирование результатов

Группирование — это мощная возможность в выражениях запросов. Запрос с оператором группировки создает последовательность групп, и каждая группа содержит `Key` и последовательность, состоящую из всех членов этой группы. В следующем новом запросе учащиеся используют первую букву имени семьи в качестве ключа.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];
```

Тип запроса изменился. Теперь он создает последовательность групп с типом `char` в качестве ключа и последовательностью `Student` объектов. Код в цикле `foreach` выполнения также должен измениться:

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
```

```
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Запустите приложение и просмотрите результаты в окне **консоли** . Дополнительные сведения см. в группы пункте.

Явное кодирование `IEnumerables` `IGroupings` может быстро стать утомительным. Создание одного и того же запроса и `foreach` цикла гораздо удобнее с помощью `var`. Ключевое слово `var` не изменяет типы ваших объектов; оно просто указывает компилятору выводить типы. Измените тип `studentQuery` и тип переменной итерации `group` на `var`, затем повторно выполните запрос. В внутреннем `foreach` цикле переменная итерации по-прежнему вводится как `Student`, и запрос работает как раньше. Измените переменную итерации `student` на `var` и снова запустите запрос. Вы видите, что вы получаете точно те же результаты.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];

foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
```

Дополнительные сведения см. в `var` разделе "Неявно типизированные локальные переменные".

Упорядочить группы по значению ключа

Группы в предыдущем запросе не в алфавитном порядке. После `group` условия можно указать `orderby` условие. Но для использования `orderby` предложения сначала требуется идентификатор, который служит ссылкой на группы, созданные предложением `group` . Вы указываете идентификатор, используя ключевое слово `into`, следующим образом:

C#

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Выполните этот запрос, и группы теперь отсортированы в алфавитном порядке.

Ключевое слово `let` можно использовать для введения идентификатора для результата любого выражения в запросе. Этот идентификатор может быть

удобным, как показано в следующем примере. Кроме того, он может повысить производительность, сохраняя результаты выражения, чтобы его не нужно вычислять несколько раз.

C#

```
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select $"{student.Last}, {student.First}";

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko, Svetlana
// O'Donnell, Claire
// Mortensen, Sven
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Дополнительные сведения см. в статье о положениилet.

Использование синтаксиса метода в выражении запроса

Как описано в синтаксисе запросов и синтаксисе метода в LINQ, некоторые операции запроса могут быть выражены только с помощью синтаксиса метода. Следующий код вычисляет общую оценку для каждой `Student` из исходных последовательностей, а затем вызывает `Average()` метод в результатах этого запроса, чтобы вычислить среднюю оценку класса.

C#

```
var studentQuery =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery.Average();
Console.WriteLine($"Class average score = {averageScore}");

// Output:
// Class average score = 334.166666666667
```

Выполните преобразование или проекцию в операторе select

Обычно запрос создает последовательность, элементы которой отличаются от элементов в исходных последовательностях. Удалите или закомментируйте предыдущий цикл запроса и выполнения и замените его следующим кодом. Запрос возвращает последовательность строк (не `Students`), и этот факт отражается в цикле `foreach` .

C#

```
IEnumerable<string> studentQuery =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

Код, приведенный ранее в этом пошаговом руководстве, указал, что средняя оценка класса составляет примерно 334. Чтобы создать последовательность `Students`, общая оценка которой выше среднего по классу, вместе с их `Student ID`, можно использовать анонимный тип в операторе `select`.

C#

```
var aboveAverageQuery =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in aboveAverageQuery)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Общие сведения о операторах запросов уровня "Стандартный"

Стандартные операторы запросов — это ключевые слова и методы, которые формируют шаблон LINQ. Язык C# определяет ключевые слова запроса LINQ , которые используются для наиболее распространенного выражения запроса. Компилятор преобразует выражения с помощью этих ключевых слов в эквивалентные вызовы метода. Две формы являются синонимами. Другие методы, которые являются частью System.Linq пространства имен, не имеют эквивалентных ключевых слов запроса. В этих случаях необходимо использовать синтаксис метода. В этом разделе рассматриваются все ключевые слова оператора запроса. Среда выполнения и другие пакеты NuGet с каждым выпуском добавляют дополнительные методы, предназначенные для работы с запросами LINQ. Наиболее распространенные методы, включая те, которые имеют эквиваленты ключевых слов запроса, рассматриваются в этом разделе. Полный список методов запросов, поддерживаемых средой выполнения .NET, см. в System.Linq.Enumerable документации по API. Помимо методов, описанных здесь, этот класс содержит методы для объединения источников данных, вычисления одного значения из источника данных, например суммы, среднего или другого значения.

> **ⓘ Важный**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider, используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

Большинство этих методов работают на последовательностях, где последовательность является объектом, тип которого реализует IEnumerable<T> интерфейс или IQueryable<T> интерфейс. Стандартные операторы запросов предоставляют возможности запроса, включая фильтрацию, проекцию, агрегирование, сортировку и многое другое. Методы, составляющие каждый набор, являются элементами расширения, определенными в Enumerable и Queryable классах соответственно. Они определяются как *члены расширения*, в которых тип приемника является либо типом `IEnumerable<T>`, либо `IQueryable<T>`, с которым они работают.

Различие между IEnumerable<T> и IQueryable<T> последовательностями определяет, как выполняется запрос во время выполнения.

Для `IEnumerable<T>` возвращаемый перечисляемый объект сохраняет аргументы, переданные методу. При перечислении этого объекта используется логика оператора запроса и возвращаются результаты запроса.

Для `IQueryable<T>`, этот запрос преобразуется в дерево выражений. Дерево выражений можно преобразовать в собственный запрос, когда источник данных может оптимизировать запрос. Библиотеки, такие как Entity Framework , преобразуют запросы LINQ в собственные SQL-запросы, которые выполняются в базе данных.

В следующем примере кода показано, как использовать стандартные операторы запросов для получения сведений о последовательности.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine($"Words of length {obj.Length}:");
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

По возможности запросы в этом разделе используют последовательность слов или чисел в качестве источника входных данных. Для запросов, где используются более сложные связи между объектами, используются следующие источники, моделиирующие школу:

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Каждый `Student` имеет класс, первичный отдел и ряд оценок. Также у `Teacher` есть свойство `City`, определяющее кампус, где учитель ведёт занятия. У `Department` есть имя и ссылка на `Teacher`, который является руководителем отдела.

Набор данных можно найти в исходном репозитории.

Типы операторов запросов

Стандартные операторы запросов различаются в зависимости от того, возвращают ли они однотонное значение или последовательность значений. Эти методы, возвращающие одноэлементное значение (например Average , и Sum) выполняются немедленно. Методы, возвращающие последовательность, откладывают выполнение запроса и возвращают перечислимый объект. Вы можете использовать выходную последовательность одного запроса в качестве входной последовательности для другого запроса. Вы объединяете методы запросов в одну последовательность, что позволяет создавать запросы произвольной сложности.

Операторы запроса

В запросе LINQ сначала необходимо указать источник данных. Предложение `from` LINQ будет первым в запросе, чтобы ввести источник данных (`students`) и переменную диапазона ().

C#

```
//queryAllStudents is an IEnumerable<Student>
var queryAllStudents = from student in students
                       select student;
```

Переменная диапазона похожа на переменную итерации в `foreach` цикле, за исключением того, что фактические итерации не выполняются в выражении запроса. При выполнении запроса переменная диапазона служит ссылкой на каждый последовательный элемент в `students` . Так как компилятор может выводить тип `student` , его не нужно указывать явным образом. В предложении `let` можно ввести дополнительные переменные диапазона. Дополнительные сведения см. в предложении 'let' .

> ⓘ **Заметка**
>
> Для не универсальных источников данных, таких как ArrayList, необходимо явно ввести переменную диапазона. Дополнительные сведения см. в статье "Запрос массива" с помощью LINQ (C#) и предложения.

После получения источника данных можно выполнить любое количество операций с этим источником данных:

- Фильтрация данных с помощью ключевого `where` слова.
- Упорядочьте данные, используя `orderby` и, при необходимости, `descending` ключевые слова.

- Группируйте данные с использованием ключевых слов `group` и, при необходимости, `into`.
- Присоедините данные с помощью ключевого слова .
- Данные проекта с помощью ключевого `select` слова.

Таблица синтаксиса выражения запроса

В следующей таблице перечислены стандартные операторы запросов, имеющие эквивалентные предложения выражения запроса.

Развернуть таблицу

Метод	Синтаксис выражения запроса C#
Cast	Используйте явно типизированные переменные диапазона: `from int i in numbers` (Дополнительные сведения см. в разделе from clause.)

GroupBy	`group … by …`
	
	-или-
	
	`group … by … into …`
	
	(Дополнительные сведения см. в предложении группы.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	`join … in … on … equals … into …` (Дополнительные сведения см. в клауза объединения.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Дополнительные сведения см. в клауза объединения.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Дополнительные сведения

Метод	Синтаксис выражения запроса C#
	см. в предложении ORDER BY.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Дополнительные сведения см. в предложении ORDER BY.)
Select	`select` (Дополнительные сведения см. в операторе select.)
SelectMany	Несколько `from` клауз. (Дополнительные сведения см. в разделе from clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Дополнительные сведения см. в предложении ORDER BY.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Дополнительные сведения см. в предложении ORDER BY.)
Where	`where` (Дополнительные сведения см. в предложении where.)

Преобразования данных с помощью LINQ

Language-Integrated LINQ касается не только извлечения данных. Это также мощный инструмент для преобразования данных. С помощью запроса LINQ можно использовать исходную последовательность в качестве входных данных и изменять ее различными способами для создания новой выходной последовательности. Вы можете сортировать и группировать, чтобы изменить саму последовательность, не изменяя сами элементы. Но, возможно, самая мощная функция запросов LINQ — это возможность создавать новые

типы. Предложение select создает выходной элемент из входного элемента. Он используется для преобразования входного элемента в выходной элемент:

- Слияние нескольких входных последовательностей в одну выходную последовательность с новым типом.
- Создайте выходные последовательности, элементы которых состоят только из одного или нескольких свойств каждого элемента в исходной последовательности.
- Создайте выходные последовательности, элементы которых состоят из результатов операций, выполняемых в исходных данных.
- Создайте выходные последовательности в другом формате. Например, можно преобразовать данные из строк SQL или текстовых файлов в XML.

Эти преобразования объединяются различными способами в одном запросе. Кроме того, вы можете использовать выходную последовательность одного запроса в качестве входной последовательности для нового запроса. В следующем примере объекты в структуре данных в памяти преобразуются в XML-элементы.

C#

```
// Create the query.
var studentsToXML = new XElement("Root",
    from student in students
    let scores = string.Join(",", student.Scores)
    select new XElement("student",
                new XElement("First", student.FirstName),
                new XElement("Last", student.LastName),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

// Execute the query.
Console.WriteLine(studentsToXML);
```

Код создает следующие выходные данные XML:

XML

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,90,73,54</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>56,78,95,95</Scores>
  </student>
```

```
  ...
  <student>
    <First>Max</First>
    <Last>Lindgren</Last>
    <Scores>86,88,96,63</Scores>
  </student>
  <student>
    <First>Arina</First>
    <Last>Ivanova</Last>
    <Scores>93,63,70,80</Scores>
  </student>
</Root>
```

Дополнительные сведения см. в статье "Создание деревьев XML в C# (LINQ to XML)".

Результаты одного запроса можно использовать в качестве источника данных для последующего запроса. В этом примере показано, как упорядочить результаты операции соединения. Этот запрос создает соединение группы, а затем сортирует группы на основе элемента `department`, который по-прежнему находится в области. Внутри инициализатора анонимного типа подзапрос упорядочивает все соответствующие элементы последовательности `students`.

C#

```
var orderedQuery = from department in departments
                   join student in students on department.ID equals
student.DepartmentID into studentGroup
                   orderby department.Name
                   select new
                   {
                       DepartmentName = department.Name,
                       Students = from student in studentGroup
                                  orderby student.LastName
                                    select student
                   };

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10} {student.FirstName,-10}");
    }
}
/* Output:
Chemistry
  Balzan     Josephine
  Fakhouri   Fadi
  Popov      Innocenty
  Seleznyova Sofiya
  Vella      Carmen
```

```
Economics
  Adams       Terry
  Adaobi      Izuchukwu
  Berggren    Jeanette
  Garcia      Cesar
  Ifeoma      Nwanneka
  Jamuike     Ifeanacho
  Larsson     Naima
  Svensson    Noel
  Ugomma      Ifunanya
Engineering
  Axelsson    Erik
  Berg        Veronika
  Engström    Nancy
  Hicks       Cassie
  Keever      Bruce
  Micallef    Nicholas
  Mortensen   Sven
  Nilsson     Erna
  Tucker      Michael
  Yermolayeva Anna
English
  Andersson   Sarah
  Feng        Hanying
  Ivanova     Arina
  Jakobsson   Jesper
  Jensen      Christiane
  Johansson   Mark
  Kolpakova   Nadezhda
  Omelchenko  Svetlana
  Urquhart    Donald
Mathematics
  Frost       Gaby
  Garcia      Hugo
  Hedlund     Anna
  Kovaleva    Katerina
  Lindgren    Max
  Maslova     Evgeniya
  Olsson      Ruth
  Sammut      Maria
  Sazonova    Anastasiya
Physics
  Åkesson     Sami
  Edwards     Amy E.
  Falzon      John
  Garcia      Debra
  Hansson     Sanna
  Mattsson    Martina
  Richardson  Don
  Zabokritski Eugene
*/
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var orderedQuery = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, studentGroup) => new
    {
        DepartmentName = department.Name,
        Students = studentGroup.OrderBy(student => student.LastName)
    })
    .OrderBy(department => department.DepartmentName);


foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10} {student.FirstName,-10}");
    }
}
```

Хотя вы можете использовать `orderby` предложение с одной или несколькими исходными последовательностями перед объединением, мы, как правило, не рекомендуем этого. Некоторые поставщики LINQ могут не сохранять порядок после соединения. Для получения дополнительных сведений см. предложение соединения .

См. также

- Enumerable
- Queryable
- Предложение select
- Члены расширения
- Ключевые слова запроса (LINQ)
- Анонимные типы

ⓘ **Заметка:** Автор создал эту статью с помощью искусственного интеллекта. Подробнее

Last updated on 01.12.2025

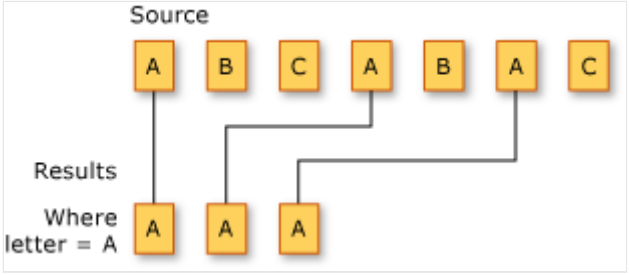
Фильтрация данных в C# с помощью LINQ

15.08.2025

Фильтрация относится к операции ограничения результирующих наборов только теми элементами, которые удовлетворяют указанному условию. Он также называется *выбором* элементов, которые соответствуют указанному условию.

> **ⓘ Важно!**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider, используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

На следующем рисунке показаны результаты фильтрации последовательности символов. Предикат для операции фильтрации указывает, что символ должен быть "A".



Стандартные методы оператора запроса, выполняющие выбор, перечислены в следующей таблице:

Развернуть таблицу

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительная информация
OfType	Выбирает значения в зависимости от их возможности приведения к указанному типу.	Неприменимо.	Enumerable.OfType Queryable.OfType
Где	Выбирает значения, основанные на функции предиката.	`where`	Enumerable.Where

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительная информация
			Queryable.Where

Используется оператор `where`, чтобы отфильтровать из массива строки, имеющие определенную длину.

> ⓘ **Примечание**
>
> Общие источники данных для этой области см. в статье "Обзор операторов стандартных запросов ".

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query =
    words.Where(word => word.Length == 3);

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
```

```
    the
    fox
*/
```

См. также

- System.Linq
- предложение where
- Как запросить метаданные сборки с помощью Reflection (LINQ) (C#)
- Как запрашивать файлы с указанным атрибутом или именем (C#)
- Как сортировать или фильтровать текстовые данные по любому слову или полю (LINQ) (C#)

Операции проекции (C#)

13.06.2025

Проекция относится к операции преобразования объекта в новую форму, которая часто состоит только из этих свойств, которые впоследствии используются. С помощью проекции можно создать новый тип, созданный из каждого объекта. Вы можете проецить свойство и выполнить математическую функцию. Вы также можете проецировать исходный объект без изменений.

> **ⓘ Важно!**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider, используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

Стандартные методы оператора запроса, выполняющие проекцию, перечислены в следующем разделе.

Методы

Развернуть таблицу

Имена методов	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
Выберите	Проекции значений, основанные на функции преобразования.	`select`	Enumerable.Select Queryable.Select
SelectMany	Проецирует последовательности значений, основанных на функции преобразования, и затем объединяет их в одну последовательность.	Используйте несколько `from` предложений	Enumerable.SelectMany Queryable.SelectMany
Zip-архив	Создает последовательность кортежей с элементами, полученными из 2-3 указанных последовательностей.	Неприменимо.	Enumerable.Zip Queryable.Zip

Select

В следующем примере условие `select` используется для проецирования первой буквы из каждой строки в списке строк.

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = words.Select(word => word.Substring(0, 1));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

В следующем примере используется несколько `from` предложений для проецирования каждого слова из каждой строки в списке.

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = phrases.SelectMany(phrase => phrase.Split(' '));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
```

```
    fox
*/
```

Метод `SelectMany` также может сформировать комбинацию сопоставления каждого элемента в первой последовательности с каждым элементом во второй последовательности:

C#

```
var query = from number in numbers
            from letter in letters
            select (number, letter);

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var method = numbers
    .SelectMany(number => letters,
    (number, letter) => (number, letter));

foreach (var item in method)
{
    Console.WriteLine(item);
}
```

Zip

Для оператора проекции `Zip` существует несколько перегрузок. Все методы работают с последовательностями двух или более возможно разнородных типов. Первые две перегрузки возвращают кортежи с соответствующим позиционным типом из заданных последовательностей.

Рассмотрим следующие коллекции:

C#

```
// An int array with 7 elements.
IEnumerable<int> numbers = [1, 2, 3, 4, 5, 6, 7];
```

```
// A char array with 6 elements.
IEnumerable<char> letters = ['A', 'B', 'C', 'D', 'E', 'F'];
```

Чтобы проецировать эти последовательности вместе, используйте Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>) оператор:

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'
```

> ⓘ **Важно!**
>
> Результирующая последовательность из операции ZIP никогда не длиннее, чем самая короткая последовательность. Длина коллекций `numbers` и `letters` отличается, и в результирующей последовательности отсутствует последний элемент из коллекции `numbers`, так как для него нет соответствующего элемента для сопоставления.

Вторая перегрузка принимает последовательность `third`. Создадим другую коллекцию, а именно `emoji`:

C#

```
// A string array with 8 elements.
IEnumerable<string> emoji = [ "🤯", "🔥", "🎉", "👀", "⭐", "💜", "✓", "💯"];
```

Чтобы проецировать эти последовательности вместе, используйте Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>) оператор:

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters, emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji: {em}");
```

```
}
// This code produces the following output:
//     Number: 1 is zipped with letter: 'A' and emoji: 😳
//     Number: 2 is zipped with letter: 'B' and emoji: 🔥
//     Number: 3 is zipped with letter: 'C' and emoji: 🥕
//     Number: 4 is zipped with letter: 'D' and emoji: 👀
//     Number: 5 is zipped with letter: 'E' and emoji: ⭐
//     Number: 6 is zipped with letter: 'F' and emoji: 💜
```

Как и предыдущая перегрузка метода `Zip`, этот метод проецирует кортеж, но в этом случае с тремя элементами.

Третья перегрузка принимает аргумент `Func<TFirst, TSecond, TResult>`, который выступает в качестве селектора результатов. Вы можете спроецировать новую результирующую последовательность из сжимаемых последовательностей.

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({(int)letter})"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//     1 = A (65)
//     2 = B (66)
//     3 = C (67)
//     4 = D (68)
//     5 = E (69)
//     6 = F (70)
```

При предыдущей `Zip` перегрузке указанная функция применяется к соответствующим элементам `number` и `letter`, создавая последовательность результатов `string`.

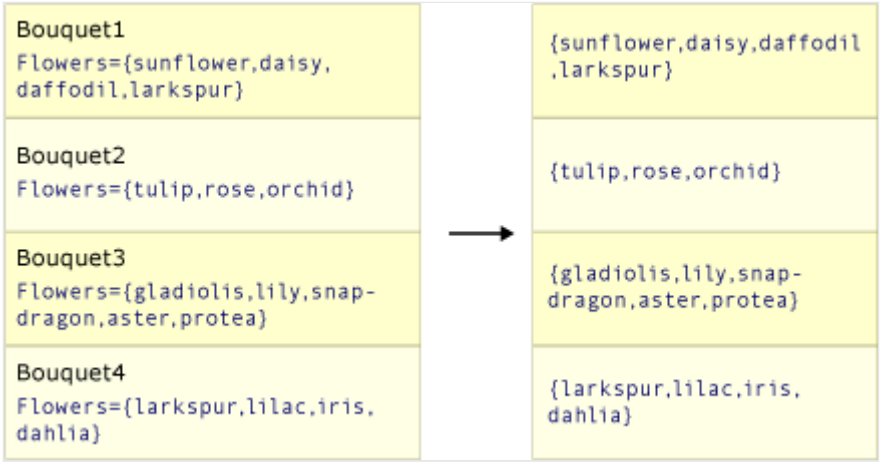
`Select` и `SelectMany`

Работа обоих `Select` и `SelectMany` заключается в создании результирующих значений (или значений) из исходных значений. `Select` создает одно значение результата для каждого исходного значения. Таким образом, общий результат представляет собой коллекцию, которая имеет то же количество элементов, что и исходная коллекция. В отличие от этого, `SelectMany` создаёт один общий результат, содержащий объединенные подколлекции из каждого исходного значения. Функция преобразования, передаваемая в качестве аргумента `SelectMany` , должна возвращать перечисленную последовательность

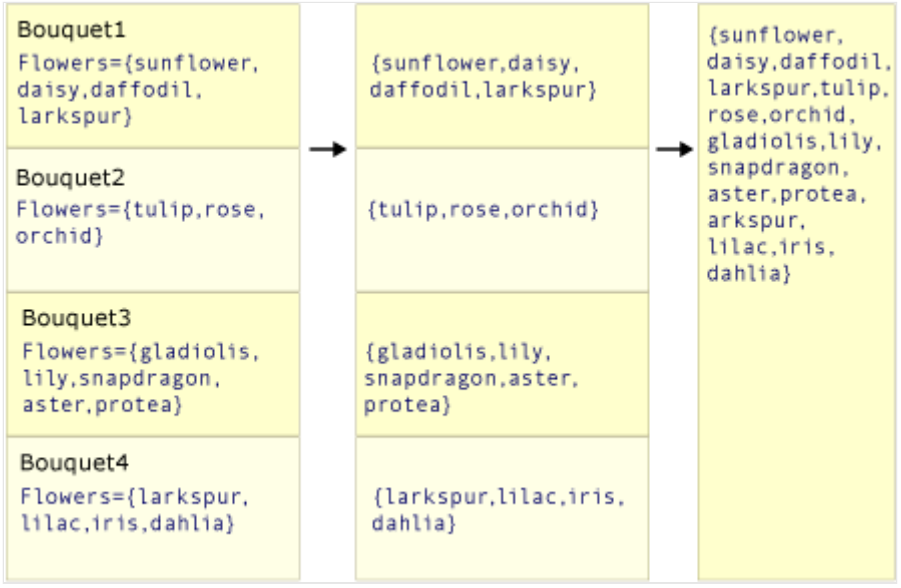
значений для каждого исходного значения. `SelectMany` объединяет эти перечисленные последовательности для создания одной большой последовательности.

На следующих двух иллюстрациях показано концептуальное различие между действиями этих двух методов. В каждом случае предположим, что функция селектора (преобразования) выбирает массив цветов из каждого исходного значения.

На этом рисунке показано, как `Select` возвращает коллекцию с таким же количеством элементов, что и исходная коллекция.



На этом рисунке показано, как `SelectMany` объединяет промежуточную последовательность массивов в одно конечное значение результата, содержащее каждое значение из каждого промежуточного массива.



Пример кода

В следующем примере сравнивается поведение `Select` и `SelectMany`. Код создает "букет" цветов, принимая элементы из каждого списка имен цветов в исходной коллекции. В следующем примере "одно значение", которое использует функция

Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>) преобразования, является коллекцией значений. В этом примере требуется дополнительный `foreach` цикл для перечисления каждой строки в каждой подпоследовательности.

C#

```
class Bouquet
{
    public required List<string> Flowers { get; init; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets =
    [
        new Bouquet { Flowers = ["sunflower", "daisy", "daffodil", "larkspur"] },
        new Bouquet { Flowers = ["tulip", "rose", "orchid"] },
        new Bouquet { Flowers = ["gladiolis", "lily", "snapdragon", "aster",
"protea"] },
        new Bouquet { Flowers = ["larkspur", "lilac", "iris", "dahlia"] }
    ];

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<string> collection in query1)
    {
        foreach (string item in collection)
        {
            Console.WriteLine(item);
        }
    }

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
    {
        Console.WriteLine(item);
    }
}
```

См. также

- System.Linq
- Предложение select
- Заполнение коллекций объектов из нескольких источников (LINQ) (C#)
- Разделение файла на множество файлов с помощью групп (LINQ) (C#)

Операции над множествами (C#)

15.08.2025

Операции задания в LINQ ссылаются на операции запроса, которые создают результирующий набор на основе присутствия или отсутствия эквивалентных элементов в одной или отдельных коллекциях.

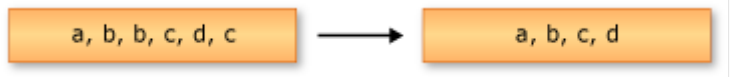
> **ⓘ Важно!**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider использовании System.Linq.IQueryable<T> источников данных и деревьев выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

Развернуть таблицу

Имена методов	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
`Distinct` или `DistinctBy`	Удаляет повторяющиеся значения из коллекции.	Неприменимо.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
`Except` или `ExceptBy`	Возвращает разницу набора, что означает, что элементы одной коллекции, которые не отображаются во второй коллекции.	Неприменимо.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
`Intersect` или `IntersectBy`	Возвращает пересечение множеств, т. е. элементы, присутствующие в каждой из двух коллекций.	Неприменимо.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
`Union` или `UnionBy`	Возвращает объединение множеств, т. е. уникальные элементы, присутствующие в одной из двух коллекций.	Неприменимо.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

`Distinct` и `DistinctBy`.

В следующем примере показано поведение метода Enumerable.Distinct применительно к последовательности строк. Возвращаемая последовательность содержит уникальные элементы из входной последовательности.



C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy",
"dog"];

IEnumerable<string> query = from word in words.Distinct()
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */
```

Вместо DistinctBy можно использовать `Distinct`, который принимает `keySelector`. `keySelector` используется как сравнительный дискриминатор для типа источника. В следующем коде слова дискриминируются на основе их `Length`, а первое слово каждой длины отображается:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy",
"dog"];

foreach (string word in words.DistinctBy(p => p.Length))
{
    Console.WriteLine(word);
}
```

```
// This code produces the following output:
//     the
//     quick
//     jumped
//     over
```

`Except` и `ExceptBy`.

В следующем примере показано поведение Enumerable.Except. Возвращаемая последовательность содержит только элементы из первой входной последовательности, которая не входит во вторую входную последовательность.



> **Примечание**
>
> В следующих примерах в этой статье используются общие источники данных для этой области.
> Каждый из них `Student` имеет уровень оценки, основной отдел и ряд показателей. У него `Teacher` также есть свойство, определяющее `City` кампус, где учитель проводит классы. У `Department` него есть имя и ссылка на `Teacher` того, кто выступает в качестве руководителя отдела.
> Пример набора данных можно найти в исходном репозитории.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }
```

```
    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

> ⓘ **Примечание**
>
> Общие источники данных для этой области см. в статье "Обзор операторов стандартных запросов ".

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Except(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * quick
 * brown
 * fox
 */
```

Вместо ExceptBy можно использовать метод `Except`, который принимает две последовательности потенциально разнородных типов и `keySelector`. Это `keySelector` тот же тип, что и тип первой коллекции. Рассмотрим следующие `Teacher` идентификаторы массива и преподавателей, чтобы исключить их. Чтобы найти преподавателей в первой

коллекции, которая не находится во второй коллекции, можно проецировать идентификатор преподавателя на вторую коллекцию:

C#

```
int[] teachersToExclude =
[
    901,    // English
    965,    // Mathematics
    932,    // Engineering
    945,    // Economics
    987,    // Physics
    901     // Chemistry
];

foreach (Teacher teacher in
    teachers.ExceptBy(
        teachersToExclude, teacher => teacher.ID))
{
    Console.WriteLine($"{teacher.First} {teacher.Last}");
}
```

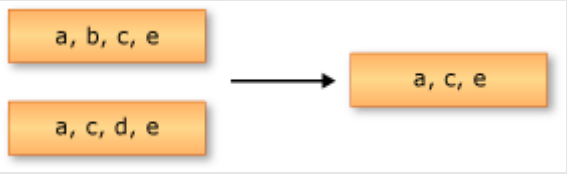
В приведенном выше коде C#:

- Массив `teachers` фильтруется только тем учителям, которые не в массиве `teachersToExclude` .
- Массив `teachersToExclude` содержит `ID` значение для всех руководителей отделов.
- Вызов приводит к `ExceptBy` новому набору значений, записанных в консоль.

Новый набор значений имеет тип, который является типом `Teacher` первой коллекции. Каждый `teacher` из массивов, не имеющих соответствующего значения идентификатора в `teachers` массиве, записывается в `teachersToExclude` консоль.

`Intersect` и `IntersectBy`.

В следующем примере показано поведение Enumerable.Intersect. Возвращаемая последовательность содержит элементы, общие для обеих входных последовательностей.



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Intersect(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 */
```

Вместо IntersectBy можно использовать метод `Intersect`, который принимает две последовательности потенциально разнородных типов и `keySelector`. `keySelector` используется как сравнительный дискриминатор для типа второй коллекции. Рассмотрим следующие массивы учащихся и преподавателей. Запрос соответствует элементам в каждой последовательности по имени, чтобы найти тех учащихся, которые также являются преподавателями:

C#

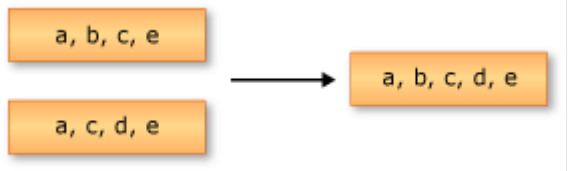
```
foreach (Student person in
    students.IntersectBy(
        teachers.Select(t => (t.First, t.Last)), s => (s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

В приведенном выше коде C#:

- Запрос создает пересечение `Teacher` имен и `Student` сравнивает их.
- В результирующей последовательности присутствуют только люди, найденные в обоих массивах.
- Полученные экземпляры `Student` записываются в консоль.

`Union` и `UnionBy`.

В следующем примере показана операция объединения двух последовательностей строк. Возвращаемая последовательность содержит уникальные элементы из обеих входных последовательностей.



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Union(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
*/
```

Вместо UnionBy можно использовать метод `Union`, который принимает две последовательности одного типа и `keySelector`. `keySelector` используется как сравнительный дискриминатор для типа источника. Следующий запрос создает список всех людей, которые являются учащимися или преподавателями. Учащиеся, которые также являются учителями, добавляются в набор профсоюзов только один раз:

C#

```
foreach (var person in
    students.Select(s => (s.FirstName, s.LastName)).UnionBy(
        teachers.Select(t => (FirstName: t.First, LastName: t.Last)), s =>
(s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

В приведенном выше коде C#:

- `teachers` Массивы `students` объединяются вместе с именами в качестве селектора ключей.
- Полученные имена записываются в консоль.

См. также

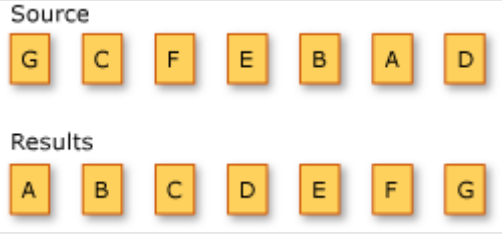
- System.Linq
- Нахождение разности множеств между двумя списками (LINQ) (C#)

Сортировка данных (C#)

Операция сортировки упорядочивает элементы последовательности на основе одного или нескольких атрибутов. Первый критерий сортировки выполняет первичную сортировку элементов. Указав второй критерий поиска, можно сортировать элементы внутри каждой группы первичной сортировки.

> **Внимание**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных на основе System.Linq.IQueryProvider используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

На следующем рисунке показаны результаты операции сортировки в алфавитном порядке в последовательности символов:



Далее перечислены методы стандартных операторов запроса, которые выполняют сортировку данных.

Методы

Развернуть таблицу

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
Упорядочить по	Сортировка значений в возрастающем порядке.	`orderby`	Enumerable.OrderBy Queryable.OrderBy
УпорядочитьПоУбыванию	Сортировка значений в	`orderby … descending`	Enumerable.OrderByDescending

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
	убывающем порядке.		Queryable.OrderByDescending
ThenBy	Выполняется дополнительная сортировка по возрастанию.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending (сортировка по убыванию)	Выполняется вторичная сортировка по убыванию.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Реверс	Изменение порядка элементов в коллекции на обратный.	Неприменимо.	Enumerable.Reverse Queryable.Reverse

> ⓘ **Примечание.**
>
> В следующих примерах в этой статье используются общие источники данных для этой области.
> Каждый из них `Student` имеет уровень оценки, основной отдел и ряд показателей. У него `Teacher` также есть свойство, указывающее на `City` кампус, где проводит занятия учитель. У `Department` есть имя и ссылка на `Teacher`, который является руководителем отдела.
> Пример набора данных можно найти в исходном репозитории.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }
```

```
    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

> ⓘ **Примечание.**
>
> Общие источники данных для этой области см. в статье "Обзор операторов стандартных запросов ".

Основная сортировка по возрастанию

В следующем примере показано, как использовать `orderby` предложение в запросе LINQ для сортировки массива учителей по имени семьи в порядке возрастания.

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Эквивалентный запрос, написанный с помощью синтаксиса метода, показан в следующем коде:

C#

```
IEnumerable<string> query = teachers
    .OrderBy(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Основная сортировка по убыванию

В следующем примере показано, как использовать `orderby descending` предложение в запросе LINQ для сортировки преподавателей по имени семьи в порядке убывания.

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last descending
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Эквивалентный запрос, написанный с помощью синтаксиса метода, показан в следующем коде:

C#

```
IEnumerable<string> query = teachers
    .OrderByDescending(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Дополнительная сортировка по возрастанию

В следующем примере показано, как использовать `orderby` предложение в запросе LINQ для выполнения первичной и вторичной сортировки. Учителя сортируются главным образом по городу, и затем по фамилии, в обоих случаях по возрастанию.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                              orderby teacher.City, teacher.Last
                              select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Эквивалентный запрос, написанный с помощью синтаксиса метода, показан в следующем коде:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenBy(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Дополнительная сортировка по убыванию

В следующем примере показано использование предложения `orderby descending` в запросе LINQ для выполнения основной сортировки по возрастанию и дополнительной сортировки по убыванию. Учителя сортируются главным образом по городу, а затем по фамилии.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                              orderby teacher.City, teacher.Last descending
                              select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Эквивалентный запрос, написанный с помощью синтаксиса метода, показан в следующем коде:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenByDescending(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

См. также

- System.Linq
- предложение orderby
- Практическое руководство. Сортировка или фильтрация текстовых данных по любому слову или полю (LINQ) (C#)

Last updated on 24.02.2026

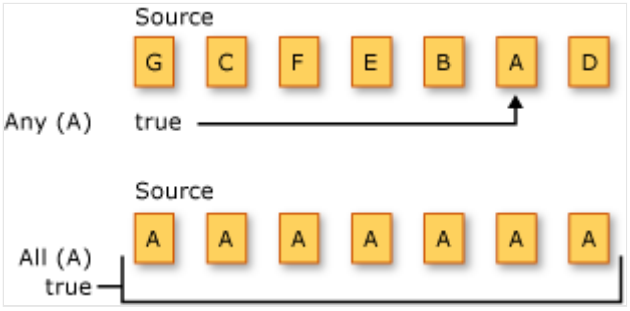
Операции квантификатора в LINQ (C#)

17.06.2025

Операции квантификатора возвращают Boolean значение, указывающее, удовлетворяют ли некоторые или все элементы последовательности определённому условию.

> **ⓘ Важно!**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider, используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

На следующем рисунке показаны две различные операции квантификатора в двух разных исходных последовательностях. Первая операция спрашивает, является ли любой из элементов символом A. Вторая операция спрашивает, являются ли все элементы символом A. Оба метода возвращают `true` в этом примере.



Развернуть таблицу

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительная информация
Все	Определяет, соответствуют ли все элементы в последовательности условию.	Неприменимо.	Enumerable.All Queryable.All
Любое	Определяет, соответствуют ли все элементы в последовательности условию.	Неприменимо.	Enumerable.Any Queryable.Any

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительная информация
Содержит	Определяет, содержит ли последовательность указанный элемент.	Неприменимо.	Enumerable.Contains Queryable.Contains

Все

В следующем примере используется `All` для поиска студентов, которые получили более 70 баллов на всех экзаменах.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.All(score => score > 70)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Cesar Garcia: 71, 86, 77, 97
// Nancy Engström: 75, 73, 78, 83
// Ifunanya Ugomma: 84, 82, 96, 80
```

Любое

В следующем примере `Any` используется для поиска студентов, которые набрали более 95 баллов на любом экзамене.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Any(score => score > 95)
                            select $"{student.FirstName} {student.LastName}:
{student.Scores.Max()}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}
```

```
// This code produces the following output:
//
// Svetlana Omelchenko: 97
// Cesar Garcia: 97
// Debra Garcia: 96
// Ifeanacho Jamuike: 98
// Ifunanya Ugomma: 96
// Michelle Caruana: 97
// Nwanneka Ifeoma: 98
// Martina Mattsson: 96
// Anastasiya Sazonova: 96
// Jesper Jakobsson: 98
// Max Lindgren: 96
```

Содержит

В следующем примере используется `Contains` для поиска студентов, которые набрали ровно 95 баллов на экзамене.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Contains(95)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Claire O'Donnell: 56, 78, 95, 95
// Donald Urquhart: 92, 90, 95, 57
```

См. также

- System.Linq
- Динамическое указание фильтров предиката во время выполнения
- Как запросить предложения, содержащие указанный набор слов (LINQ) (C#)

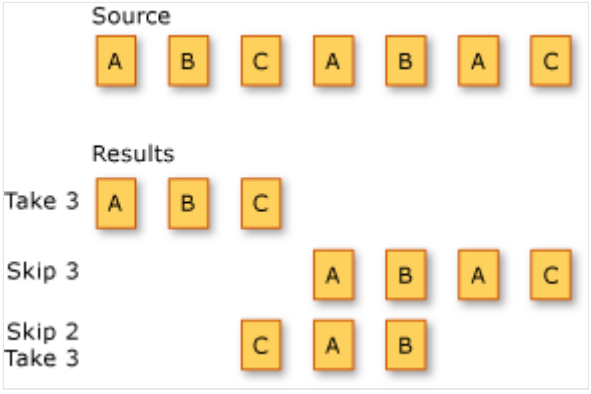
Секционирование данных (C#)

17.06.2025

Секционирование в LINQ относится к операции разделения входной последовательности на два раздела без переупорядочения элементов, а затем возврата одного из разделов.

> **ⓘ Важно!**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider, используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

На следующем рисунке показаны результаты трех разных операций секционирования в последовательности символов. Первая операция возвращает первые три элемента в последовательности. Вторая операция пропускает первые три элемента и возвращает оставшиеся элементы. Третья операция пропускает первые два элемента в последовательности и возвращает следующие три элемента.



Стандартные методы оператора запроса, которые разделяют последовательности, перечислены в следующем разделе.

Операторы

Развернуть таблицу

Имена методов	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
Пропустить	Пропускает элементы до указанной позиции в последовательности.	Неприменимо.	Enumerable.Skip Queryable.Skip
ПропуститьПока	Пропускает элементы на основе функции предиката, пока элемент не удовлетворяет условию.	Неприменимо.	Enumerable.SkipWhile Queryable.SkipWhile
Взять	Выбирает элементы до указанной позиции в последовательности.	Неприменимо.	Enumerable.Take Queryable.Take
TakeTime	Принимает элементы на основе функции предиката, пока элемент не удовлетворяет условию.	Неприменимо.	Enumerable.TakeWhile Queryable.TakeWhile
Блочное	Разбивает элементы последовательности на блоки заданного максимального размера.	Неприменимо.	Enumerable.Chunk Queryable.Chunk

Все приведенные ниже примеры используются Enumerable.Range(Int32, Int32) для создания последовательности чисел от 0 до 7.

Вы используете метод `Take` для того, чтобы взять только первые элементы в последовательности.

C#

```
foreach (int number in Enumerable.Range(0, 8).Take(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
```

Вы используете метод `Skip` чтобы пропустить первые элементы последовательности и использовать оставшиеся элементы.

C#

```
foreach (int number in Enumerable.Range(0, 8).Skip(3))
{
    Console.WriteLine(number);
```

```
}
// This code produces the following output:
// 3
// 4
// 5
// 6
// 7
```

Методы `TakeWhile` и `SkipWhile` также принимают и пропускают элементы последовательности. Однако вместо заданного числа элементов эти методы пропускают или принимают элементы на основе условия. `TakeWhile` принимает элементы последовательности, пока элемент не соответствует условию.

C#

```
foreach (int number in Enumerable.Range(0, 8).TakeWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
// 3
// 4
```

`SkipWhile` пропускает первые элементы, если условие имеет значение true. Возвращается первый элемент, не соответствующий условию, и возвращаются все последующие элементы.

C#

```
foreach (int number in Enumerable.Range(0, 8).SkipWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 5
// 6
// 7
```

Оператор `Chunk` используется для разделения элементов последовательности на основе заданного `size`.

C#

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
```

```
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

Предыдущий код C#:

- Enumerable.Range(Int32, Int32) Используется для создания последовательности чисел.
- Применяет `Chunk` оператор, разделяя последовательность на блоки с максимальным размером 3.

См. также

- System.Linq

Преобразование типов данных (C#)

Методы преобразования изменяют тип входных объектов.

> **ⓘ Внимание**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных на основе System.Linq.IQueryProvider используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

Операции преобразования в запросах LINQ полезны в различных приложениях. Ниже приводятся некоторые примеры.

- Метод Enumerable.AsEnumerable можно использовать, чтобы скрыть настраиваемую реализацию типа стандартного оператора запроса.
- Метод Enumerable.OfType позволяет использовать непараметризованные коллекции для запросов LINQ.
- Методы Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList и Enumerable.ToLookup можно использовать для принудительного немедленного выполнения запроса, не дожидаясь, пока этот запрос будет перечислен.

Методы

В следующей таблице перечислены методы стандартных операторов запросов, выполняющие преобразование типов данных.

Методы преобразования в этой таблице, имена которых начинаются с "As", изменяют статический тип исходной коллекции, но не перечисляют его. Методы, имена которых начинаются с "To", перечисляют исходную коллекцию и помещают элементы в соответствующий тип коллекции.

Развернуть таблицу

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
AsEnumerable	Возвращает входное значение, типизированное как	Неприменимо.	Enumerable.AsEnumerable

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
	IEnumerable<T>.		
AsQueryable	Преобразует IEnumerable (универсальный шаблон) в IQueryable (универсальный шаблон).	Неприменимо.	Queryable.AsQueryable
Каст	Приводит элементы коллекции к указанному типу.	Используйте явно типизированную переменную диапазона. Например: `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Фильтрует значения в зависимости от возможности их приведения к указанному типу.	Неприменимо.	Enumerable.OfType Queryable.OfType
ToArray	Преобразует коллекцию в массив. Этот метод принудительно запускает выполнение запроса.	Неприменимо.	Enumerable.ToArray
ToDictionary	Помещает элементы в Dictionary<TKey,TValue> в зависимости от функции селектора ключа. Этот метод принудительно выполняет запрос.	Неприменимо.	Enumerable.ToDictionary
ToList	Преобразует коллекцию в List<T>. Этот метод заставляет выполнить запрос.	Неприменимо.	Enumerable.ToList
ToLookup	Помещает элементы в Lookup<TKey,TElement> (словарь "один ко многим") в зависимости от функции выбора ключа. Этот метод принудительно выполняет запрос.	Неприменимо.	Enumerable.ToLookup

ⓘ Примечание.

В следующих примерах в этой статье используются общие источники данных для этой области.
Каждый из них `Student` имеет уровень оценки, основной отдел и ряд показателей. У него `Teacher` также есть свойство, указывающее на `City` кампус, где проводит занятия учитель. У `Department` есть имя и ссылка на `Teacher`, который является руководителем отдела.
Пример набора данных можно найти в исходном **репозитории**.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Примечание.

Общие источники данных для этой области см. в статье "Обзор операторов стандартных запросов ".

Пример синтаксиса выражения запроса

В следующем примере кода явным образом типизированная переменная диапазона используется для приведения типа к подтипу перед обращением к элементу, доступному только в подтипе.

C#

```
IEnumerable people = students;

var query = from Student student in people
            where student.Year == GradeLevel.ThirdYear
            select student;

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

Эквивалентный запрос можно выразить с помощью синтаксиса метода, как показано в следующем примере:

C#

```
IEnumerable people = students;

var query = people
    .Cast<Student>()
    .Where(student => student.Year == GradeLevel.ThirdYear);

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

См. также

- System.Linq
- предложение from

Last updated on 24.02.2026

Join операции в LINQ

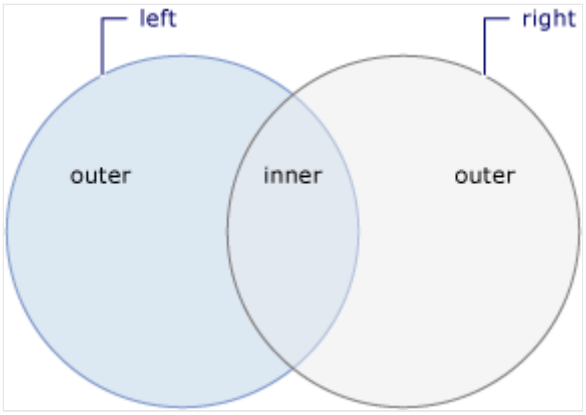
Соединение связывает объекты в одном источнике данных с объектами, которые совместно используют общий атрибут в другом источнике данных.

> **ⓘ Внимание**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных, основанные на System.Linq.IQueryProvider использовании System.Linq.IQueryable<T> источников данных и деревьев выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

Присоединение является важной операцией в запросах, направленных на источники данных, связи между которыми невозможно отследить напрямую. В объектно-ориентированном программировании присоединение может означать корреляцию между объектами, которые не моделировались, например обратное направление односторонняя связь. Пример односторонняя связь — `Student` это класс, имеющий свойство типа `Department` , представляющее основное, но `Department` класс не имеет свойства, являющегося коллекцией `Student` объектов. Если у вас есть список `Department` объектов, и вы хотите найти всех учащихся в каждом отделе, можно использовать операцию присоединения для их поиска.

Платформа LINQ предоставляет методы соединения: Join и GroupJoin. Они выполняют эквисоединения, или соединения, которые сопоставляют два источника данных на основе равенства их ключей. Для сравнения, Transact-SQL поддерживает операторы соединения, отличные от `equals`, такие как оператор `less than`. В терминах реляционной базы данных Join реализует внутреннее соединение, это тип соединения, где возвращаются только те объекты, которые имеют соответствие в другом наборе данных. Метод GroupJoin не имеет прямого эквивалента в терминах реляционных баз данных, но реализует надмножество внутренних соединений и левых внешних соединений. Левое внешнее соединение — это соединение, которое возвращает каждый элемент первого (левого) источника данных, даже если он не имеет коррелированных элементов в другом источнике данных.

На следующем рисунке показано концептуальное представление двух наборов и элементов в этих наборах, включенных в внутреннее соединение или левое внешнее соединение.



Методы

Развернуть таблицу

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
Join	Соединяет две последовательности на основании функций селектора ключа и извлекает пары значений.	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	Соединяет две последовательности на основании функций селектора ключа и группирует полученные при сопоставлении данные для каждого элемента.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin
LeftJoin	Сопоставляет элементы двух последовательностей на основе соответствующих ключей.	N/A	Enumerable.LeftJoin Queryable.LeftJoin
RightJoin	Сопоставляет элементы двух последовательностей на основе соответствующих ключей.	N/A	Enumerable.RightJoin Queryable.RightJoin

Примечание.

В следующих примерах в этой статье используются общие источники данных для этой области.
Каждый из них `Student` имеет уровень оценки, основной отдел и ряд показателей. У него `Teacher` также есть свойство, определяющее `City` кампус, где учитель проводит классы. У `Department` него есть имя и ссылка на `Teacher` того, кто выступает в качестве руководителя отдела.
Пример набора данных можно найти в исходном репозитории.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

> ⓘ **Примечание.**
>
> Общие источники данных для этой области см. в статье "Обзор операторов стандартных запросов ".

Пример ниже демонстрирует использование оператора `join … in … on … equals …` для объединения двух последовательностей на основе определенного значения.

C#

```
var query = from student in students
            join department in departments on student.DepartmentID equals
```

```
department.ID
            select new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

Предыдущий запрос можно выразить с помощью синтаксиса метода, как показано в следующем коде:

C#

```
var query = students.Join(departments,
    student => student.DepartmentID, department => department.ID,
    (student, department) => new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name });

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

В следующем примере предложение `join … in … on … equals … into …` используется для объединения двух последовательностей на основе определенного значения и группировки соответствующих совпадений для каждого элемента.

C#

```
IEnumerable<IEnumerable<Student>> studentGroups = from department in departments
                    join student in students on department.ID equals
student.DepartmentID into studentGroup
                    select studentGroup;

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

Предыдущий запрос можно выразить с помощью синтаксиса метода, как показано в следующем примере:

C#

```
// Join department and student based on DepartmentId and grouping result
IEnumerable<IEnumerable<Student>> studentGroups = departments.GroupJoin(students,
    department => department.ID, student => student.DepartmentID,
    (department, studentGroup) => studentGroup);

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

Выполнение внутренних соединений

В терминах реляционной базы данных *внутреннее соединение* создает результирующий набор, в котором каждый элемент первой коллекции отображается один раз для каждого соответствующего элемента во второй коллекции. Если элемент в первой коллекции не имеет соответствующих элементов, он не отображается в результирующем наборе. Метод Join, вызываемый условием `join` в языке C#, реализует внутреннее соединение. В следующих примерах показано, как выполнить четыре варианта внутреннего соединения:

- Простое внутреннее соединение, которое сопоставляет элементы из двух источников данных на основе простого ключа.
- Внутреннее соединение, которое сопоставляет элементы из двух источников данных на основе *составного* ключа. Составной ключ — это ключ, который состоит из нескольких значений, позволяющих сопоставлять элементы на основе сразу нескольких свойств.
- *Множественное объединение*, в котором вы последовательно добавляете операции объединения друг к другу.
- Внутреннее соединение, использующее групповой джойн.

Соединение с одним ключом

В следующем примере объекты совпадают `Teacher` с `Department` объектами, которые `TeacherId` совпадают с этим `Teacher`. Предложение `select` в C# определяет, как выглядят полученные объекты. В следующем примере результирующий объект — это анонимные типы, состоящие из имени отдела и имени преподавателя, который ведет отдел.

C#

```
var query = from department in departments
            join teacher in teachers on department.TeacherID equals teacher.ID
```

```
            select new
            {
                DepartmentName = department.Name,
                TeacherName = $"{teacher.First} {teacher.Last}"
            };

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Вы достигаете одинаковые результаты с помощью синтаксиса Join метода:

C#

```
var query = teachers
    .Join(departments, teacher => teacher.ID, department => department.TeacherID,
        (teacher, department) =>
        new { DepartmentName = department.Name, TeacherName = $"{teacher.First}
{teacher.Last}" });

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Преподаватели, которые не являются руководителями отделов, не появляются в окончательных результатах.

Составное соединение ключей

Вместо сопоставления элементов на основе только одного свойства используйте составной ключ для сравнения элементов на основе нескольких свойств. Укажите функцию селектора ключей для каждой коллекции, чтобы вернуть анонимный тип, состоящий из свойств, которые требуется сравнить. Если свойства помечаются, они должны иметь одинаковую метку в анонимном типе каждого ключа. Кроме того, свойства должны отображаться в одинаковом порядке.

В следующем примере используется список `Teacher` объектов и список `Student` объектов, чтобы определить, какие преподаватели также являются учащимися. Оба этих типа имеют свойства, представляющие имя первого и семейного имени каждого человека. Функции, создающие ключи соединения из элементов каждого списка, возвращают анонимный тип, состоящий из свойств. Операция соединения сравнивает эти составные ключи для равенства и возвращает пары объектов из обоих списков, где совпадают имя и фамилия.

C#

```
// Join the two data sources based on a composite key consisting of first and last
name,
// to determine which employees are also students.
IEnumerable<string> query =
    from teacher in teachers
    join student in students on new
    {
        FirstName = teacher.First,
        LastName = teacher.Last
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select teacher.First + " " + teacher.Last;

string result = "The following people are both teachers and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.Write(result);
```

Этот Join метод можно использовать, как показано в следующем примере:

C#

```
IEnumerable<string> query = teachers
    .Join(students,
        teacher => new { FirstName = teacher.First, LastName = teacher.Last },
        student => new { student.FirstName, student.LastName },
        (teacher, student) => $"{teacher.First} {teacher.Last}"
 );

Console.WriteLine("The following people are both teachers and students:");
foreach (string name in query)
{
    Console.WriteLine(name);
}
```

Несколько объединений

Можно добавить любое количество операций соединения для выполнения нескольких соединений. Каждое `join` предложение в C# сопоставляет указанный источник данных с результатами предыдущего соединения.

Первое `join` предложение соответствует учащимся и отделам на `Student` основе сопоставления `DepartmentID` объекта `Department ID`. Он возвращает последовательность

анонимных типов, содержащих `Student` объект и `Department` объект.

Второе `join` предложение сопоставляет анонимные типы, возвращаемые первым присоединением к `Teacher` объектам на основе идентификатора преподавателя, соответствующего идентификатору руководителя отдела. Он возвращает последовательность анонимных типов, содержащих имя учащегося, имя отдела и имя руководителя отдела. Так как эта операция является внутренним соединением, запрос возвращает только те объекты из первого источника данных, которые имеют совпадение во втором источнике данных.

C#

```
// The first join matches Department.ID and Student.DepartmentID from the list of
students and
// departments, based on a common ID. The second join matches teachers who lead
departments
// with the students studying in that department.
var query = from student in students
    join department in departments on student.DepartmentID equals department.ID
    join teacher in teachers on department.TeacherID equals teacher.ID
    select new {
        StudentName = $"{student.FirstName} {student.LastName}",
        DepartmentName = department.Name,
        TeacherName = $"{teacher.First} {teacher.Last}"
    };

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the department
run by "{obj.TeacherName}".""");
}
```

Эквивалентный запрос, использующий несколько Join методов, использует один и тот же подход с анонимным типом:

C#

```
var query = students
    .Join(departments, student => student.DepartmentID, department => department.ID,
        (student, department) => new { student, department })
    .Join(teachers, commonDepartment => commonDepartment.department.TeacherID,
teacher => teacher.ID,
        (commonDepartment, teacher) => new
        {
            StudentName = $"{commonDepartment.student.FirstName}
{commonDepartment.student.LastName}",
            DepartmentName = commonDepartment.department.Name,
            TeacherName = $"{teacher.First} {teacher.Last}"
        });
```

```
foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the department
run by "{obj.TeacherName}".""");
}
```

Внутреннее соединение с помощью сгруппированного соединения

В следующем примере показано, как реализовать внутреннее соединение с помощью соединения группы. Список `Department` объектов присоединен к списку `Student` объектов на `Department.ID` основе сопоставления `Student.DepartmentID` свойства. Соединение группы создает коллекцию промежуточных групп, где каждая группа состоит из `Department` объекта и последовательности соответствующих `Student` объектов. Второе `from` предложение объединяет (или плоские) эту последовательность последовательностей в одну длинную последовательность. Предложение `select` указывает тип элементов в конечной последовательности. Этот тип является анонимным типом, состоящим из имени учащегося и соответствующего имени отдела.

C#

```
var query1 =
    from department in departments
    join student in students on department.ID equals student.DepartmentID into gj
    from subStudent in gj
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    };
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Вы можете добиться того же результата с помощью GroupJoin метода, как показано в следующем примере:

C#

```
var queryMethod1 = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, gj) => new { department, gj })
    .SelectMany(departmentAndStudent => departmentAndStudent.gj,
        (departmentAndStudent, subStudent) => new
```

```
        {
            DepartmentName = departmentAndStudent.department.Name,
            StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
        });

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in queryMethod1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Результат эквивалентен результирующему набору, полученному с использованием предложения `join` без предложения `into` для выполнения внутреннего соединения. Следующий код демонстрирует этот эквивалентный запрос:

C#

```
var query2 = from department in departments
    join student in students on department.ID equals student.DepartmentID
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    };

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in query2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Чтобы избежать цепочки вызовов, используйте одиночный метод Join, как показано здесь.

C#

```
var queryMethod2 = departments.Join(students, departments => departments.ID, student
=> student.DepartmentID,
    (department, student) => new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    });

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in queryMethod2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Выполнение групповых соединений

Присоединение к группе полезно для создания иерархических структур данных. Каждый элемент из первой коллекции группируется с набором соответствующих элементов из второй коллекции.

> ⓘ **Примечание.**
>
> Каждый элемент первой коллекции отображается в результирующем наборе соединения группы независимо от того, найдены ли коррелированные элементы во второй коллекции. Если коррелированные элементы не найдены, последовательность коррелированных элементов для этого элемента пуста. Селектор результата имеет доступ к каждому элементу первой коллекции. Это поведение отличается от селектора результатов в негрупповом соединении, который не может получить доступ к элементам из первой коллекции, не имеющим соответствий во второй коллекции.

> ⚠ **Предупреждение**
>
> Метод Enumerable.GroupJoin не имеет прямого эквивалента в традиционных терминах реляционных баз данных. Однако он реализует расширенный набор внутренних соединений и левых внешних соединений. Обе эти операции можно записать с точки зрения сгруппированного соединения. Дополнительные сведения см. в разделе Entity Framework Core. GroupJoin

В первом примере в этой статье показано, как выполнить присоединение к группе. Во втором примере показано, как использовать объединение групп для создания XML-элементов.

Присоединение к группе

В следующем примере выполняется групповое объединение объектов типа `Department` и `Student` на основе сопоставления свойства `Department.ID` с `Student.DepartmentID`. В отличие от негруппового соединения, которое создает пару элементов для каждого совпадения, групповое соединение создает только один результирующий объект для каждого элемента первой коллекции. В этом примере первая коллекция является `Department` объектом. Соответствующие элементы из второй коллекции (в данном примере — объекты `Student`) группируются в коллекцию. Наконец, результирующая

функция селектора создает для каждого совпадения анонимный тип, состоящий из `Department.Name` и коллекции объектов `Student`.

C#

```
var query = from department in departments
    join student in students on department.ID equals student.DepartmentID into
studentGroup
    select new
    {
        DepartmentName = department.Name,
        Students = studentGroup
    };

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

В приведенном выше примере `query` переменная содержит запрос, который создает список, в котором каждый элемент является анонимным типом, содержащим имя отдела и коллекцию учащихся, изучающих этот отдел.

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var query = departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, Students) => new { DepartmentName = department.Name, Students });

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

Объединение групп для создания XML

Групповые соединения идеально подходят для создания XML с помощью LINQ to XML. Приведенный ниже пример аналогичен предыдущему за исключением того, что вместо создания анонимных типов результирующая функция селектора создает XML-элементы, представляющие соединенные объекты.

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    from department in departments
    join student in students on department.ID equals student.DepartmentID into
studentGroup
    select new XElement("Department",
        new XAttribute("Name", department.Name),
        from student in studentGroup
        select new XElement("Student",
            new XAttribute("FirstName", student.FirstName),
            new XAttribute("LastName", student.LastName)
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, Students) => new XElement("Department",
            new XAttribute("Name", department.Name),
            from student in Students
            select new XElement("Student",
                new XAttribute("FirstName", student.FirstName),
                new XAttribute("LastName", student.LastName)
            )
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

Выполнение внешних соединений

.NET 10 включает LeftJoin и RightJoin методы в System.Linq.Enumerable и System.Linq.Queryable классы. Эти методы выполняют внешнее левое *равное соединение* и внешнее правое *равное соединение* соответственно. Внешнее левостороннее равноправное соединение — это соединение, в котором каждый член первой последовательности включён в результирующую последовательность, даже если во второй последовательности отсутствует совпадение. Внешнее правое эквисоединение — это соединение, в котором каждый член второй последовательности включается в выходную последовательность, даже если первая последовательность не включает совпадение.

Эмуляция левого внешнего соединения

Перед выпуском .NET 10, используйте LINQ для выполнения левостороннего внешнего соединения, вызывая метод DefaultIfEmpty для результатов группового соединения.

В следующем примере показано, как использовать DefaultIfEmpty метод в результатах соединения группы для выполнения левого внешнего соединения.

Первым шагом в создании левого внешнего соединения двух коллекций является выполнение внутреннего соединения, используя групповое объединение. (См. раздел Выполните внутренние соединения для объяснения этого процесса.) В этом примере список `Department` объектов присоединяется к списку `Student` объектов на `Department` основе идентификатора объекта, соответствующего учащемуся `DepartmentID`.

Вторым шагом является включение каждого элемента первой (левой) коллекции в набор результатов, даже если элемент не имеет совпадений в правой коллекции. Для этого необходимо вызвать DefaultIfEmpty для каждой из последовательностей элементов, соответствующих группового соединения. В этом примере вы вызываете DefaultIfEmpty на каждой последовательности соответствующих `Student` объектов. Метод возвращает коллекцию, содержащую одно значение по умолчанию, если последовательность соответствующих `Student` объектов пуста для любого `Department` объекта, гарантируя, что каждый `Department` объект представлен в коллекции результатов.

> ⓘ **Примечание.**
>
> Значение по умолчанию для ссылочного типа — `null`, поэтому в примере проверяется пустая ссылка перед доступом к каждому элементу в каждой коллекции `Student`.

C#

```
var query =
    from student in students
    join department in departments on student.DepartmentID equals department.ID into
gj
    from subgroup in gj.DefaultIfEmpty()
    select new
    {
        student.FirstName,
        student.LastName,
        Department = subgroup?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}: {v.Department}");
}
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var query = students
    .GroupJoin(
        departments,
        student => student.DepartmentID,
        department => department.ID,
        (student, departmentList) => new { student, subgroup = departmentList })
    .SelectMany(
        joinedSet => joinedSet.subgroup.DefaultIfEmpty(),
        (student, department) => new
        {
            student.student.FirstName,
            student.student.LastName,
            Department = department?.Name ?? string.Empty
        });

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}: {v.Department}");
}
```

См. также

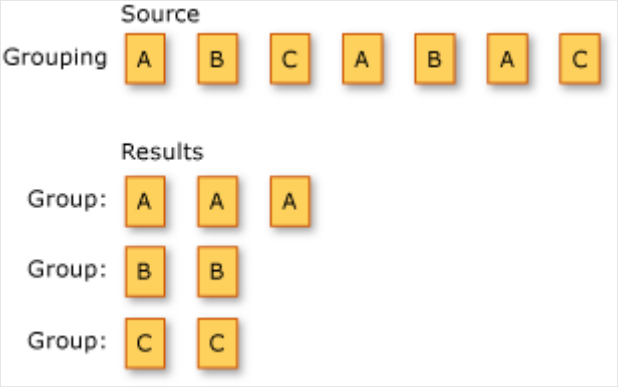
- Join
- GroupJoin
- Анонимные типы
- Формулировка запросов-объединений и запросов векторного произведения
- Условие объединения
- Предложение group

- Присоединение содержимого из разнородных файлов (LINQ) (C#)
- Заполнение коллекций объектов из нескольких источников (LINQ) (C#)

Last updated on 15.12.2025

Группирование данных (C#)

Группированием называют операцию объединения данных в группы таким образом, чтобы у элементов в каждой группе был общий атрибут. На следующем рисунке показаны результаты операции группирования последовательности символов. Ключ для каждой группы — это символ.



> **ⓘ Внимание**
>
> В этих примерах используется System.Collections.Generic.IEnumerable<T> источник данных. Источники данных на основе System.Linq.IQueryProvider используют System.Linq.IQueryable<T> источники данных и деревья выражений. Деревья выражений имеют ограничения на допустимый синтаксис C#. Кроме того, каждый `IQueryProvider` источник данных, например EF Core , может наложить больше ограничений. Ознакомьтесь с документацией по источнику данных.

Стандартные методы оператора запроса, которые группируют элементы данных, перечислены в следующей таблице.

Развернуть таблицу

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения

Группировка по	Группирует элементы с общим атрибутом. Объект представляет каждую IGrouping<TKey,TElement> группу.	`group … by …`
		
		–или–
		
		`group … by … into …`
			Enumerable.GroupBy
			
			Queryable.GroupBy

ToLookup	Вставляет элементы в Lookup<TKey,TElement> (словарь "один	Неприменимо.	Enumerable.ToLookup

Имя метода	Описание	Синтаксис выражения запроса C#	Дополнительные сведения
	ко многим") в зависимости от функции выбора ключа.		

В следующем примере кода используется `group by`, чтобы сгруппировать целые числа в списке в зависимости от того, являются ли они четными или нечетными.

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = numbers
    .GroupBy(number => number % 2);

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

> ⓘ **Примечание.**
>
> В следующих примерах в этой статье используются общие источники данных для этой области.

Каждый из них `Student` имеет уровень оценки, основной отдел и ряд показателей. У него `Teacher` также есть свойство, указывающее на `City` кампус, где проводит занятия учитель. У `Department` есть имя и ссылка на `Teacher`, который является руководителем отдела.
Пример набора данных можно найти в исходном **репозитории**.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

ⓘ **Примечание.**

Общие источники данных для этой области см. в статье "Обзор операторов стандартных запросов ".

Группировка результатов запросов

Группирование — одна из самых эффективных функций LINQ. В приведенных ниже примерах демонстрируются различные способы группирования данных:

- по отдельному свойству;
- по первой букве строкового свойства;
- по расчетному числовому диапазону;
- по логическому предикату или другому выражению;
- по составному ключу.

Кроме того, последние два запроса проецируют свои результаты в новый анонимный тип, содержащий только имя и фамилию студента. Дополнительные сведения см. в разделе групповое предложение.

Пример группировки по отдельному свойству

В этом примере показана группировка элементов источника с помощью отдельного свойства элемента в качестве ключа группы. Ключом является `enum` год обучения студента в школе. При операции группирования используется компаратор равенства, используемый по умолчанию для типа.

C#

```
var groupByYearQuery =
    from student in students
    group student by student.Year into newGroup
    orderby newGroup.Key
    select newGroup;

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Эквивалентный код с использованием синтаксиса метода показан в следующем примере:

C#

```
// Variable groupByYearQuery is an IEnumerable<IGrouping<GradeLevel,
// DataClass.Student>>.
var groupByYearQuery = students
```

```
    .GroupBy(student => student.Year)
    .OrderBy(newGroup => newGroup.Key);

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Пример группировки по значению

В следующем примере показано, как группировать элементы источника, используя в качестве ключа группы что-то, отличное от свойства объекта. В этом примере ключ — это первая буква имени семьи учащегося.

C#

```
var groupByFirstLetterQuery =
    from student in students
    let firstLetter = student.LastName[0]
    group student by firstLetter;

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Для доступа к элементам группы необходим вложенный foreach.

Эквивалентный код с использованием синтаксиса метода показан в следующем примере:

C#

```
var groupByFirstLetterQuery = students
    .GroupBy(student => student.LastName[0]);

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
```

```
    }
}
```

Пример группировки по диапазону

В этом примере показана группировка элементов источника путем использования числового диапазона в качестве ключа группы. Затем запрос преобразует результаты в анонимный тип, содержащий только имя и фамилию и диапазон процентилей, к которому относится учащийся. Анонимный тип используется, так как для отображения результатов не требуется использовать полный `Student` объект. `GetPercentile` — это вспомогательная функция, которая вычисляет процент на основе средних результатов учащегося. Метод возвращает целое число в диапазоне от 0 до 10.

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
    orderby percentGroup.Key
    select percentGroup;

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

Вложенный цикл foreach необходим для итерации по группам и элементам группы. Эквивалентный код с использованием синтаксиса метода показан в следующем примере:

C#

```
static int GetPercentile(Student s)
{
```

```
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery = students
    .Select(student => new { student, percentile = GetPercentile(student) })
    .GroupBy(student => student.percentile)
    .Select(percentGroup => new
    {
        percentGroup.Key,
        Students = percentGroup.Select(s => new { s.student.FirstName,
s.student.LastName })
    })
    .OrderBy(percentGroup => percentGroup.Key);

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup.Students)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

Пример группировки для сравнения

В этом примере показана группировка элементов источника с помощью выражения логического сравнения. В этом случае логическое выражение проверяет, превосходит ли среднее значение экзаменационного результата учащегося 75 баллов. Как и в предыдущих примерах, результаты проецируются в анонимный тип, так как полный исходный элемент не нужен. Свойства в анонимном типе становятся свойствами `Key` элемента.

C#

```
var groupByHighAverageQuery =
    from student in students
    group new
    {
        student.FirstName,
        student.LastName
    } by student.Scores.Average() > 75 into studentGroup
    select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
```

```
    }
}
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var groupByHighAverageQuery = students
    .GroupBy(student => student.Scores.Average() > 75)
    .Select(group => new
    {
        group.Key,
        Students = group.AsEnumerable().Select(s => new { s.FirstName, s.LastName
})
    });

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup.Students)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

Группировка по анонимному типу

В этом примере показано, как использовать анонимный тип для инкапсуляции ключа, содержащего несколько значений. В этом примере первое ключевое значение — это первая буква имени семьи учащегося. Второе значение ключа является логическим и указывает, набрал ли учащийся более 85 баллов на первом экзамене. Группы можно сортировать по любому из свойств в ключе.

C#

```
var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetterOfLastName
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetterOfLastName} who
```

```
scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var groupByCompoundKey = students
    .GroupBy(student => new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    })
    .OrderBy(studentGroup => studentGroup.Key.FirstLetterOfLastName);

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetterOfLastName} who
scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

Создайте вложенную группу

В следующем примере демонстрируется создание вложенных групп в выражении запроса LINQ. Каждая группа, созданная в соответствии с годом обучения или курсом учащегося, затем подразделяется на группы по именам учащихся.

C#

```
var nestedGroupsQuery =
    from student in students
    group student by student.Year into newGroup1
    from newGroup2 in
    from student in newGroup1
    group student by student.LastName
    group newGroup2 by newGroup1.Key;

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
```

```
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

Три вложенных `foreach` цикла требуются для итерации внутренних элементов вложенной группы.
(Наведите указатель мыши на переменные итерации, `outerGroup` `innerGroup` и `innerGroupElement` чтобы увидеть их фактический тип.)

Эквивалентный запрос с использованием синтаксиса метода показан в следующем коде:

C#

```
var nestedGroupsQuery =
    students
    .GroupBy(student => student.Year)
    .Select(newGroup1 => new
    {
        newGroup1.Key,
        NestedGroup = newGroup1
            .GroupBy(student => student.LastName)
    });

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup.NestedGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

Выполнение подзапроса в операции группировки

В этой статье показаны два разных способа создания запроса, упорядочивающего исходные данные в группы и затем выполняющего вложенный запрос для каждой группы

по отдельности. Основное действие в каждом примере заключается в группировании исходных элементов с помощью *продолжения* с именем `newGroup` с последующим созданием вложенного запроса для `newGroup`. Этот вложенный запрос выполняется для каждой новой группы, созданной внешним запросом. В этом примере окончательные выходные данные — это не группа, а плоская последовательность анонимных типов.

Дополнительные сведения о способах группирования см. в разделе клауза "group". Дополнительные сведения о продолжениях см. в разделе into. В приведенном ниже примере в качестве источника данных используется структура данных в памяти, но те же принципы действуют для любого типа источника данных LINQ.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.Scores.Average()
        ).Max()
    };

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

Запрос в предыдущем фрагменте также можно записать с помощью синтаксиса метода. В следующем фрагменте кода приведен семантически эквивалентный запрос, написанный с использованием синтаксиса метода.

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Max(student2 => student2.Scores.Average())
        });

var count = queryGroupMax.Count();
```

```
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

См. также

- System.Linq
- GroupBy
- IGrouping<TKey,TElement>
- Групповое условие
- Разделение файла на несколько файлов с помощью групп (LINQ) (C#)

Last updated on 24.02.2026

Практическое руководство. Использование LINQ для запроса файлов и каталогов

07.06.2025

Многие операции файловой системы по сути являются запросами и поэтому хорошо подходят для подхода LINQ. Эти запросы являются недеструктивными. Они не изменяют содержимое исходных файлов или папок. Запросы не должны вызывать побочные эффекты. Как правило, любой код (включая запросы, выполняющие операции создания и обновления и удаления), которые изменяют исходные данные, должны храниться отдельно от кода, который просто запрашивает данные.

Существует некоторая сложность создания источника данных, который точно представляет содержимое файловой системы и обрабатывает исключения корректно. В примерах этого раздела создается коллекция моментальных снимков объектов FileInfo, представляющая все файлы в указанной корневой папке и все ее вложенные папки. Фактическое состояние каждого FileInfo может измениться в период между началом и завершением выполнения запроса. Например, можно создать список FileInfo объектов для использования в качестве источника данных. Если вы пытаетесь получить доступ к `Length` свойству в запросе, FileInfo объект пытается получить доступ к файловой системе, чтобы обновить значение `Length`. Если файл больше не существует, в вашем запросе появляется FileNotFoundException, даже если вы не обращаетесь к файловой системе напрямую.

Как запрашивать файлы с указанным атрибутом или именем

В этом примере показано, как найти все файлы с указанным расширением имени файла (например,.txt") в указанном дереве каталогов. В нем также показано, как вернуть новый или старый файл в дереве на основе времени создания. Возможно, вам потребуется изменить первую строку во многих примерах, если вы запускаете этот код на Windows, Mac или Linux системе.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
```

```
var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;

// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//    Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();

Console.WriteLine($"\r\nThe newest .txt file is {newestFile.FullName}. Creation
time: {newestFile.CreationTime}");
```

Группирование файлов по расширению

В этом примере показано, как LINQ можно использовать для выполнения расширенных операций группировки и сортировки в списках файлов или папок. В нём также показано, как странично выводить данные в окне консоли с помощью методов Skip и Take.

В следующем запросе показано, как сгруппировать содержимое указанного дерева каталогов по расширению имени файла.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fileGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
```

```
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine($"\t{f.FullName.Substring(trimLength)}");
    }
    Console.WriteLine();
}
```

Выходные данные из этой программы могут быть длинными в зависимости от сведений о локальной файловой системе и заданном параметре `startFolder` . Чтобы включить просмотр всех результатов, в этом примере показано, как просмотреть результаты. Вложенный `foreach` цикл является обязательным, так как каждая группа перечисляется отдельно.

Как запросить общее количество байтов в наборе папок

В этом примере показано, как получить общее количество байтов, используемых всеми файлами в указанной папке и всеми вложенными папками. Метод Sum добавляет значения всех элементов, выбранных в предложении `select` . Этот запрос можно изменить, чтобы получить самый большой или наименьший файл в указанном дереве каталогов, вызвав метод Min или Max вместо Sum.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
                let fileLen = new FileInfo(file).Length
                where fileLen > 0
                select fileLen;

// Cache the results to avoid multiple trips to the file system.
long[] fileLengths = fileQuery.ToArray();

// Return the size of the largest file
long largestFile = fileLengths.Max();

// Return the total number of bytes in all the files under the specified folder.
```

```
long totalBytes = fileLengths.Sum();

Console.WriteLine($"There are {totalBytes} bytes in {fileList.Count()} files under
{startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

Этот пример расширяет предыдущий пример, чтобы сделать следующее:

- Как получить размер в байтах крупнейшего файла.
- Как получить размер в байтах наименьшего файла.
- Как получить самый большой или наименьший FileInfo файл объекта из одной или нескольких папок в указанной корневой папке.
- Как получить последовательность, например 10 крупнейших файлов.
- Как упорядочить файлы в группы на основе их размера в байтах, игнорируя файлы, которые меньше указанного размера.

В следующем примере содержатся пять отдельных запросов, которые показывают, как запрашивать и группировать файлы в зависимости от размера файла в байтах. Эти примеры можно изменить, чтобы создать запрос на другое свойство FileInfo объекта.

C#

```
// Return the FileInfo object for the largest file
// by sorting and selecting from beginning of list
FileInfo longestFile = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length descending
                        select fileInfo
                        ).First();

Console.WriteLine($"The largest file under {startFolder} is {longestFile.FullName}
with a length of {longestFile.Length} bytes");

//Return the FileInfo of the smallest file
FileInfo smallestFile = (from file in fileList
                         let fileInfo = new FileInfo(file)
                         where fileInfo.Length > 0
                         orderby fileInfo.Length ascending
                         select fileInfo
                        ).First();

Console.WriteLine($"The smallest file under {startFolder} is
{smallestFile.FullName} with a length of {smallestFile.Length} bytes");

//Return the FileInfos for the 10 largest files
var queryTenLargest = (from file in fileList
                       let fileInfo = new FileInfo(file)
                       let len = fileInfo.Length
                       orderby len descending
```

```
                        select fileInfo
                       ).Take(10);

Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups = from file in fileList
                      let fileInfo = new FileInfo(file)
                      let len = fileInfo.Length
                      where len > 0
                      group fileInfo by (len / 100000) into fileGroup
                      where fileGroup.Key >= 2
                      orderby fileGroup.Key descending
                      select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"\t{item.Name}: {item.Length}");
    }
}
```

Чтобы вернуть один или несколько полных FileInfo объектов, запрос сначала должен проверить каждый из них в источнике данных, а затем отсортировать их по значению свойства Length. Затем он может вернуть одну или последовательность с наибольшей длиной. Используется First для возврата первого элемента в списке. Используется Take для возврата первого n числа элементов. Укажите порядок сортировки по убыванию, чтобы поместить наименьшие элементы в начало списка.

Как выполнить поиск повторяющихся файлов в иерархии каталогов

Иногда файлы с одинаковым именем могут находиться в нескольких папках. В этом примере показано, как запрашивать такие повторяющиеся имена файлов в указанной корневой папке. Во втором примере показано, как запрашивать файлы, размер которых и время LastWrite также совпадают.

```
C#
```

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

// used in WriteLine to keep the lines shorter
int charsToSkip = startFolder.Length;

// var can be used for convenience with groups.
var queryDupNames = from file in fileList
                    group file.FullName.Substring(charsToSkip) by file.Name into
fileGroup
                    where fileGroup.Count() > 1
                    select fileGroup;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {(queryDup.Key.ToString() == string.Empty ? "
[none]" : queryDup.Key.ToString())}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"\t{fileName}");
    }
}
```

Первый запрос использует ключ для определения соответствия. Он находит файлы с одинаковым именем, но содержимое которого может отличаться. Второй запрос использует составной ключ для сопоставления с тремя свойствами FileInfo объекта. Этот запрос гораздо чаще находит файлы с одинаковым именем и аналогичным или идентичным содержимым.

C#

```
    string startFolder = """C:\Program Files\dotnet\sdk""";
    // Or
    // string startFolder = "/usr/local/share/dotnet/sdk";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    DirectoryInfo dir = new DirectoryInfo(startFolder);
    IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
```

```
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //
    var queryDupFiles = from file in fileList
                        group file.FullName.Substring(charsToSkip) by
                        (Name: file.Name, LastWriteTime: file.LastWriteTime,
Length: file.Length )
                        into fileGroup
                        where fileGroup.Count() > 1
                        select fileGroup;

    foreach (var queryDup in queryDupFiles.Take(20))
    {
        Console.WriteLine($"Filename = {(queryDup.Key.ToString() == string.Empty ?
"[none]" : queryDup.Key.ToString())}");

        foreach (var fileName in queryDup)
        {
            Console.WriteLine($"\t{fileName}");
        }
    }
}
```

Запрос содержимого текстовых файлов в папке

В этом примере показано, как запрашивать все файлы в указанном дереве каталогов, открывать каждый файл и проверять его содержимое. Этот тип метода можно использовать для создания индексов или обратных индексов содержимого дерева каталогов. В этом примере выполняется простой поиск строк. Однако более сложные типы сопоставления шаблонов можно выполнять с регулярным выражением.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

string searchTerm = "change";

var queryMatchingFiles = from file in fileList
                         where file.Extension == ".txt"
                         let fileText = File.ReadAllText(file.FullName)
                         where fileText.Contains(searchTerm)
                         select file.FullName;
```

```
// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");
foreach (string filename in queryMatchingFiles)
{
    Console.WriteLine(filename);
}
```

Сравнение содержимого двух папок

В этом примере показано три способа сравнения двух списков файлов:

- Запрашивая логическое значение, указывающее, идентичны ли два списка файлов.
- Запросив пересечение папок для получения файлов, которые находятся в обеих папках.
- Запросив разность множеств, чтобы получить файлы, которые находятся в одной папке, но не в другой.

Приведенные здесь методы можно адаптировать для сравнения последовательностей объектов любого типа.

В классе `FileComparer`, показанном здесь, демонстрируется, как использовать настраиваемый класс сравнения вместе со стандартными операторами запросов. Класс не предназначен для использования в реальных сценариях. Он просто использует имя и длину в байтах каждого файла для определения, совпадает ли содержимое папок. В реальном сценарии необходимо изменить это средство сравнения, чтобы выполнить более строгую проверку равенства.

C#

```
// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, FileInfo? f2)
    {
        return (f1?.Name == f2?.Name &&
                f1?.Length == f2?.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
    // also be equal. Because equality as defined here is a simple value equality,
not
    // reference identity, it is possible that two or more objects will produce
the same
    // hash code.
```

```
    public int GetHashCode(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}

public static void CompareDirectories()
{
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles("*.*",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles("*.*",
SearchOption.AllDirectories);

    //A custom file comparer defined below
    FileCompare myFileCompare = new FileCompare();

    // This query determines whether the two folders contain
    // identical file lists, based on the custom file comparer
    // that is defined in the FileCompare class.
    // The query executes immediately because it returns a bool.
    bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

    if (areIdentical == true)
    {
        Console.WriteLine("the two folders are the same");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine($"The following files are in both folders (total number
= {queryCommonFiles.Count()}):");
        foreach (var v in queryCommonFiles.Take(10))
        {
            Console.WriteLine(v.Name); //shows which items end up in result list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }
```

```
    // Find the set difference between the two folders.
    var queryList1Only = (from file in list1
                          select file)
                          .Except(list2, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list1 but not list2 (total
number = {queryList1Only.Count()}):");
    foreach (var v in queryList1Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }

    var queryList2Only = (from file in list2
                          select file)
                          .Except(list1, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list2 but not list1 (total
number = {queryList2Only.Count()}:");
    foreach (var v in queryList2Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }
}
```

Как изменить порядок полей файла с разделителями

Файл значений, разделённых запятыми (CSV) — это текстовый файл, который часто используется для хранения данных электронной таблицы или других табличных данных, представленных строками и столбцами. Используя Split метод для разделения полей, можно легко запрашивать и управлять CSV-файлами с помощью LINQ. На самом деле, один и тот же метод можно использовать для переупорядочения частей любой структурированной строки текста; Он не ограничен CSV-файлами.

В следующем примере предполагается, что три столбца представляют "имя семьи", "имя" и "идентификатор". Поля находятся в алфавитном порядке на основе имен семей учащихся. Запрос создает новую последовательность, в которой отображается столбец идентификатора, а затем второй столбец, который объединяет имя учащегося и имя семьи. Строки переупорядочены в соответствии с полем идентификатора. Результаты сохраняются в новом файле и исходные данные не изменяются. В следующем тексте показано содержимое файла *spreadsheet1.csv* , используемого в следующем примере:

```
txt
```

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

Следующий код считывает исходный файл и изменяет порядок столбцов в CSV-файле, чтобы изменить порядок столбцов:

C#

```
string[] lines = File.ReadAllLines("spreadsheet1.csv");

// Create the query. Put field 2 first, then
// reverse and combine fields 0 and 1 from the old field
IEnumerable<string> query = from line in lines
                            let fields = line.Split(',')
                            orderby fields[2]
                            select $"{fields[2]}, {fields[1]} {fields[0]}";

File.WriteAllLines("spreadsheet2.csv", query.ToArray());

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/
```

Разделение файла на множество файлов с помощью групп

В этом примере показано, как объединить содержимое двух файлов, а затем создать набор новых файлов, которые упорядочивают данные новым способом. Запрос использует содержимое двух файлов. В следующем тексте показано содержимое первого файла *,names1.txt*:

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

Второй файл *,names2.txt*, содержит другой набор имен, некоторые из которых являются общими для первого набора:

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Следующий код запрашивает оба файла, принимает объединение обоих файлов, а затем записывает новый файл для каждой группы, определенный первой буквой имени семейства:

C#

```
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
var mergeQuery = fileA.Union(fileB);

// Group the names by the first letter in the last name.
var groupQuery = from name in mergeQuery
                 let n = name.Split(',')[0]
```

```
                  group name by n[0] into g
                  orderby g.Key
                  select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);

    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
        Console.WriteLine($"   {item}");
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

Присоединение содержимого из разнородных файлов

В этом примере показано, как объединить данные из двух файлов с разделителями-запятыми, которые используют общее значение, используемое в качестве соответствующего ключа. Этот метод может быть полезен, если необходимо объединить данные из двух электронных таблиц или из электронной таблицы и из файла с другим форматом в новый файл. Пример можно изменить для работы с любым структурированным текстом.

В следующем тексте показано содержимое *scores.csv*. Файл представляет данные электронной таблицы. Столбец 1 — это идентификатор учащегося, а столбцы 2–5 — тестовые оценки.

```txt
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

В следующем тексте показано содержимое *names.csv*. Файл представляет электронную таблицу, содержащую имя семьи учащегося, имя и идентификатор учащегося.

```txt
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Присоединяйте содержимое из разнородных файлов, содержащих связанные сведения. *Файлnames.csv* содержит имя учащегося и номер идентификатора. *Файлscores.csv* содержит идентификатор и набор четырех показателей тестирования. Следующий запрос

объединяет оценки с именами учащихся с помощью идентификатора в качестве соответствующего ключа. Код показан в следующем примере:

C#

```
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = from name in names
                 let nameFields = name.Split(',')
                 from id in scores
                 let scoreFields = id.Split(',')
                 where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
                 select $"{nameFields[0]},{scoreFields[1]},{scoreFields[2]},
{scoreFields[3]},{scoreFields[4]}";

Console.WriteLine("\r\nMerge two spreadsheets:");
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
Console.WriteLine($"{scoreQuery.Count()} total names in list");
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

Вычисление значений столбцов в текстовом файле CSV

В этом примере показано, как выполнять статистические вычисления, такие как Sum, Average, Min и Max в столбцах файла .csv. Примеры принципов, показанных здесь, можно применить к другим типам структурированного текста.

В следующем тексте показано содержимое *scores.csv*. Предположим, что первый столбец представляет идентификатор учащегося, а последующие столбцы представляют оценки

из четырех экзаменов.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

В следующем тексте показано, как использовать Split метод для преобразования каждой строки текста в массив. Каждый элемент массива представляет столбец. Наконец, текст в каждом столбце преобразуется в его числовое представление.

C#

```
public static class SumColumns
{
    public static void ProcessColumns(string filePath, string seperator)
    {
        // Divide each exam into a group
        var exams = from line in MatrixFrom(filePath, seperator)
                    from score in line

                    // Identify the column number
                    let colNumber = Array.FindIndex(line, t =>
ReferenceEquals(score, t))

                    // The first column is the student ID, not the exam score
                    // so it needs to be excluded
                    where colNumber > 0

                    // Convert the score from string to int
                    // Group by column number, i.e. one group per exam
                    group double.Parse(score) by colNumber into g
                    select new
                    {
                        Title = $"Exam#{g.Key}",
                        Min = g.Min(),
                        Max = g.Max(),
                        Avg = Math.Round(g.Average(), 2),
                        Total = g.Sum()
                    };

        foreach (var exam in exams)
```

```
        {
            Console.WriteLine($"{exam.Title}\t"
            + $"Average:{exam.Avg,6}\t"
            + $"High Score:{exam.Max,3}\t"
            + $"Low Score:{exam.Min,3}\t"
            + $"Total:{exam.Total,5}");
        }
    }

    // Transform the file content to an IEnumerable of string arrays
    // like a matrix
    private static IEnumerable<string[]> MatrixFrom(string filePath, string
seperator)
    {
        using StreamReader reader = File.OpenText(filePath);

        for (string? line = reader.ReadLine(); line is not null; line =
reader.ReadLine())
        {
            yield return line.Split(seperator, StringSplitOptions.TrimEntries);
        }
    }
}

// Output:
// Exam#1  Average: 86.08  High Score: 99  Low Score: 35   Total: 1033
// Exam#2  Average: 86.42  High Score: 94  Low Score: 72   Total: 1037
// Exam#3  Average: 84.75  High Score: 91  Low Score: 65   Total: 1017
// Exam#4  Average: 76.92  High Score: 94  Low Score: 39   Total:  923
```

Если ваш файл разделён табуляцией, просто обновите аргумент в методе `SumColumns.ProcessColumns` на `\t`.

Практическое руководство. Использование LINQ для запроса строк

Статья • 25.04.2024

Строки хранятся в виде последовательности символов. В качестве последовательности символов их можно запрашивать с помощью LINQ. В этой статье существует несколько примеров запросов, которые запрашивают строки для разных символов или слов, фильтруют строки или смешивают запросы с регулярными выражениями.

Запрос символов в строке

В следующем примере запрашивается строка, чтобы определить количество содержащихся в ней цифр.

C#

```
string aString = "ABCDE99F-J74-12-89A";

// Select only those characters that are numbers
var stringQuery = from ch in aString
                  where Char.IsDigit(ch)
                  select ch;

// Execute the query
foreach (char c in stringQuery)
    Console.Write(c + " ");

// Call the Count method on the existing query.
int count = stringQuery.Count();
Console.WriteLine($"Count = {count}");

// Select all characters before the first '-'
var stringQuery2 = aString.TakeWhile(c => c != '-');

// Execute the second query
foreach (char c in stringQuery2)
    Console.Write(c);
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

В предыдущем запросе показано, как можно рассматривать строку как последовательность символов.

Подсчет вхождения слова в строке

В следующем примере показано, как использовать запрос LINQ для подсчета вхождения указанного слова в строке. Для выполнения подсчета сначала Split вызывается метод для создания массива слов. Есть затраты Split на производительность метода. Если единственной операцией в строке является подсчет слов, рекомендуется использовать Matches вместо этого методы или IndexOf методы.

C#

```
string text = """
    Historically, the world of data and the world of objects
    have not been well integrated. Programmers work in C# or Visual Basic
    and also in SQL or XQuery. On the one side are concepts such as classes,
    objects, fields, inheritance, and .NET APIs. On the other side
    are tables, columns, rows, nodes, and separate languages for dealing
with
    them. Data types often require translation between the two worlds; there
are
    different standard functions. Because the object world has no notion of
query, a
    query can only be represented as a string without compile-time type
checking or
    IntelliSense support in the IDE. Transferring data from SQL tables or
XML trees to
    objects in memory is often tedious and error-prone.
    """;

string searchTerm = "data";

//Convert the string into an array of words
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
string[] source = text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);

// Create the query.  Use the InvariantCultureIgnoreCase comparison to match
"data" and "Data"
var matchQuery = from word in source
                 where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                 select word;

// Count the matches, which executes the query.
int wordCount = matchQuery.Count();
Console.WriteLine($"""{wordCount} occurrences(s) of the search term "
{searchTerm}" were found.""");
```

```
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

В предыдущем запросе показано, как можно просматривать строки как последовательность слов после разделения строки на последовательность слов.

Сортировка или фильтрация текстовых данных по любому слову или полю

В следующем примере демонстрируется сортировка строк структурированного текста, таких как значения, разделенные запятыми, по любому полю в строке. Поле можно динамически указать во время выполнения. Предположим, что поля в *scores.csv* представляют идентификатор учащегося, за которым следует серия из четырех тестов:

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Следующий запрос сортирует строки на основе оценки первого экзамена, хранящегося во втором столбце:

C#

```
// Create an IEnumerable data source
string[] scores = File.ReadAllLines("scores.csv");

// Change this to any value from 0 to 4.
int sortField = 1;

Console.WriteLine($"Sorted highest to lowest by field [{sortField}]:");

// Split the string and sort on field[num]
var scoreQuery = from line in scores
                 let fields = line.Split(',')
```

```
                  orderby fields[sortField] descending
                  select line;

foreach (string str in scoreQuery)
{
    Console.WriteLine(str);
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

В предыдущем запросе показано, как управлять строками, разделяя их на поля и запрашивая отдельные поля.

Как запрашивать предложения, определенные слова

В следующем примере показано, как найти предложения в текстовом файле, который содержит совпадения для каждого из указанных наборов слов. Хотя массив терминов поиска жестко закодирован, он также может быть динамически заполнен во время выполнения. Запрос возвращает предложения, содержащие слова "Исторически", "данные" и "интегрированные".

C#

```
string text = """
Historically, the world of data and the world of objects
have not been well integrated. Programmers work in C# or Visual Basic
and also in SQL or XQuery. On the one side are concepts such as classes,
objects, fields, inheritance, and .NET APIs. On the other side
are tables, columns, rows, nodes, and separate languages for dealing with
them. Data types often require translation between the two worlds; there are
different standard functions. Because the object world has no notion of
query, a
query can only be represented as a string without compile-time type checking
or
IntelliSense support in the IDE. Transferring data from SQL tables or XML
```

```
trees to
objects in memory is often tedious and error-prone.
""";

// Split the text block into an array of sentences.
string[] sentences = text.Split(['.', '?', '!']);

// Define the search terms. This list could also be dynamically populated at
run time.
string[] wordsToMatch = [ "Historically", "data", "integrated" ];

// Find sentences that contain all the terms in the wordsToMatch array.
// Note that the number of terms to match is not specified at compile time.
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
var sentenceQuery = from sentence in sentences
                    let w =
sentence.Split(separators,StringSplitOptions.RemoveEmptyEntries)
                    where w.Distinct().Intersect(wordsToMatch).Count() ==
wordsToMatch.Count()
                    select sentence;

foreach (string str in sentenceQuery)
{
    Console.WriteLine(str);
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/
```

Запрос сначала разбивает текст на предложения, а затем разбивает каждое предложение на массив строк, в которых хранится каждое слово. Для каждого из этих массивов метод Distinct удаляет все повторяющиеся слова, после чего выполняет операцию Intersect в отношении массива слов и массива `wordsToMatch`. Если число пересечений совпадает с размером массива `wordsToMatch`, все слова были найдены и возвращается исходное предложение.

Вызов, который Split использует знаки препинания в качестве разделителей, чтобы удалить их из строки. Если вы не удалили знак препинания, например строку "Исторически", которая не будет соответствовать "Исторически" в массиве `wordsToMatch` . В зависимости от типов препинания, найденных в исходном тексте, может потребоваться использовать дополнительные разделители.

Практическое руководство. Объединение запросов LINQ с регулярными выражениями

В следующем примере показано, как использовать Regex класс для создания регулярного выражения для более сложного сопоставления в текстовых строках. Запрос LINQ упрощает фильтрацию именно тех файлов, которые требуется найти с помощью регулярного выражения, и формирование результатов.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Take a snapshot of the file system.
var fileList = from file in Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories)
               let fileInfo = new FileInfo(file)
               select fileInfo;

// Create the regular expression to find all things "Visual".
System.Text.RegularExpressions.Regex searchTerm =
    new System.Text.RegularExpressions.Regex(@"microsoft.net.
(sdk|workload)");

// Search the contents of each .htm file.
// Remove the where clause to find even more matchedValues!
// This query produces a list of files where a match
// was found, and a list of the matchedValues in that file.
// Note: Explicit typing of "Match" in select clause.
// This is required because MatchCollection is not a
// generic IEnumerable collection.
var queryMatchingFiles =
    from file in fileList
    where file.Extension == ".txt"
    let fileText = File.ReadAllText(file.FullName)
    let matches = searchTerm.Matches(fileText)
    where matches.Count > 0
    select new
    {
        name = file.FullName,
        matchedValues = from System.Text.RegularExpressions.Match match in
matches
                        select match.Value
    };

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");

foreach (var v in queryMatchingFiles)
{
    // Trim the path a bit, then write
    // the file name in which a match was found.
    string s = v.name.Substring(startFolder.Length - 1);
    Console.WriteLine(s);
```

```
    // For this file, write out all the matching strings
    foreach (var v2 in v.matchedValues)
    {
        Console.WriteLine($"  {v2}");
    }
}
```

Вы также можете запросить объект, MatchCollection возвращаемый поиском `RegEx` . В результатах создается только значение каждого совпадения. Однако также можно использовать LINQ для выполнения всех видов фильтрации, сортировки и группировки в этой коллекции. Так как MatchCollection это негенерическая IEnumerable коллекция, необходимо явно указать тип переменной диапазона в запросе.

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

LINQ и коллекции

Статья • 25.04.2024

Большинство коллекций моделит *последовательность* элементов. Для запроса любого типа коллекции можно использовать LINQ. Другие методы LINQ находят элементы в коллекции, значения вычислений из элементов коллекции или изменяют коллекцию или ее элементы. В этих примерах вы узнаете о методах LINQ и их использовании с коллекциями или другими источниками данных.

Как найти разницу между двумя списками

В этом примере показано, как использовать LINQ для сравнения двух списков строк и вывода тех строк, которые находятся в первой коллекции, но не во второй. Первая коллекция имен хранится в файле *names1.txt*:

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

Вторая коллекция имен хранится в файле *names2.txt*. Некоторые имена отображаются в обеих последовательностях.

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

В следующем коде показано, как использовать Enumerable.Except метод для поиска элементов в первом списке, которые не находятся во втором списке:

C#

```
// Create the IEnumerable data sources.
string[] names1 = File.ReadAllLines("names1.txt");
string[] names2 = File.ReadAllLines("names2.txt");

// Create the query. Note that method syntax must be used here.
var differenceQuery = names1.Except(names2);

// Execute the query.
Console.WriteLine("The following lines are in names1.txt but not
names2.txt");
foreach (string s in differenceQuery)
    Console.WriteLine(s);
/* Output:
 The following lines are in names1.txt but not names2.txt
 Potra, Cristina
 Noriega, Fabricio
 Aw, Kam Foo
 Toyoshima, Tim
 Guy, Wey Yuan
 Garcia, Debra
 */
```

Некоторые типы операций запроса, например Except, Distinctи UnionConcat, могут быть выражены только в синтаксисе на основе методов.

Объединение и сравнение коллекций строк

В этом примере показано объединение файлов, содержащих строки текста, и последующая сортировка результатов. В частности, в нем показано, как выполнять объединение, объединение и пересечение двух наборов текстовых строк. В нем используются те же два текстовых файла, которые показаны в предыдущем примере. В коде показаны примеры Enumerable.Concat, Enumerable.Unionа также Enumerable.Except.

C#

```
//Put text files in your solution folder
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

//Simple concatenation and sort. Duplicates are preserved.
var concatQuery = fileA.Concat(fileB).OrderBy(s => s);
```

```
// Pass the query variable to another function for execution.
OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are
preserved:");

// Concatenate and remove duplicate names based on
// default string comparer.
var uniqueNamesQuery = fileA.Union(fileB).OrderBy(s => s);
OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

// Find the names that occur in both files (based on
// default string comparer).
var commonNamesQuery = fileA.Intersect(fileB);
OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

// Find the matching fields in each list. Merge the two
// results by using Concat, and then
// sort using the default string comparer.
string nameMatch = "Garcia";

var tempQuery1 = from name in fileA
                 let n = name.Split(',')
                 where n[0] == nameMatch
                 select name;

var tempQuery2 = from name2 in fileB
                 let n2 = name2.Split(',')
                 where n2[0] == nameMatch
                 select name2;

var nameMatchQuery = tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"""Concat based on partial name match "
{nameMatch}":""");

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine($"{query.Count()} total names in list");
}
/* Output:
    Simple concatenate and sort. Duplicates are preserved:
    Aw, Kam Foo
    Bankov, Peter
    Bankov, Peter
    Beebe, Ann
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
```

```
    Guy, Wey Yuan
    Holm, Michael
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    20 total names in list

    Union removes duplicate names:
    Aw, Kam Foo
    Bankov, Peter
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    16 total names in list

    Merge based on intersect:
    Bankov, Peter
    Holm, Michael
    Garcia, Hugo
    Beebe, Ann
    4 total names in list

    Concat based on partial name match "Garcia":
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    3 total names in list
*/
```

Заполнение коллекций объектов из нескольких источников

В этом примере показано, как объединить данные из разных источников в последовательность новых типов.

> **ⓘ Примечание**
>
> Не пытайтесь объединить данные в памяти или данные в файловой системе с данными, которые остаются в базе данных. Такие междоменные соединения могут повлечь за собой непредвиденные результаты из-за разных способов, с помощью которых операции соединения могут определяться для запросов к базам данных и другим типам источников. Кроме того, существует риск, что такая операция может вызвать исключение нехватки памяти, если объем данных в базе данных достаточно большой. Чтобы объединить данные из базы данных с данными в памяти, сначала вызовите `ToList` или `ToArray` для запроса к базе данных, а затем выполните присоединение для возвращенной коллекции.

В этом примере используются два файла. Первый, *names.csv*, содержит имена учащихся и идентификаторы учащихся.

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Второй, *scores.csv*, содержит идентификаторы учащихся в первом столбце, за которым следуют оценки экзаменов.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
```

```
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

В следующем примере показано, как использовать именованную запись `Student` для хранения объединенных данных из двух коллекций строк в памяти, которые имитируют данные электронной таблицы в формате .csv. Идентификатор используется в качестве ключа для сопоставления учащихся с их оценками.

C#

```
// Each line of names.csv consists of a last name, a first name, and an
// ID number, separated by commas. For example, Omelchenko,Svetlana,111
string[] names = File.ReadAllLines("names.csv");

// Each line of scores.csv consists of an ID number and four test
// scores, separated by commas. For example, 111, 97, 92, 81, 60
string[] scores = File.ReadAllLines("scores.csv");

// Merge the data sources using a named type.
// var could be used instead of an explicit type. Note the dynamic
// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores = from nameLine in names
                                        let splitName = nameLine.Split(',')
                                        from scoreLine in scores
                                        let splitScoreLine =
scoreLine.Split(',')
                                        where Convert.ToInt32(splitName[2])
== Convert.ToInt32(splitScoreLine[0])
                                        select new Student
                                        (
                                            FirstName: splitName[0],
                                            LastName: splitName[1],
                                            ID:
Convert.ToInt32(splitName[2]),
                                            ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                                         select
Convert.ToInt32(scoreAsText)
                                                        ).ToArray()
                                        );

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine($"The average score of {student.FirstName}
```

```
{student.LastName} is {student.ExamScores.Average()}.");
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/
```

В предложении select каждый новый `Student` объект инициализируется из данных в двух источниках.

Если вам не нужно хранить результаты запроса, кортежи или анонимные типы могут быть более удобными, чем именованные типы. Следующий пример выполняет ту же задачу, что и предыдущий пример, но использует кортежи вместо именованных типов:

C#

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 = from nameLine in names
                        let splitName = nameLine.Split(',')
                        from scoreLine in scores
                        let splitScoreLine = scoreLine.Split(',')
                        where Convert.ToInt32(splitName[2]) ==
Convert.ToInt32(splitScoreLine[0])
                        select (FirstName: splitName[0],
                                LastName: splitName[1],
                                ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                             select
Convert.ToInt32(scoreAsText))
                                             .ToList()
                                );

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine($"The average score of {student.FirstName}
```

```
{student.LastName} is {student.ExamScores.Average()}.");
}
```

Практическое руководство. Выполнение запроса к ArrayList с помощью LINQ

При использовании LINQ для запроса негенерических IEnumerable коллекций, таких как ArrayList, необходимо явно объявить тип переменной диапазона, чтобы отразить конкретный тип объектов в коллекции. Если у вас есть ArrayList `Student` объекты, предложение from должно выглядеть следующим образом:

C#

```
var query = from Student s in arrList
//...
```

Указав тип переменной диапазона, вы приведение каждого элемента в ArrayList объекте `Student`.

Использование явным образом типизированной переменной диапазона в выражении запроса эквивалентно вызову метода Cast. Cast создает исключение, если указанный приведение не может быть выполнено. Cast и OfType являются двумя методами оператора стандартных запросов, которые работают с негенерическими IEnumerable типами. Дополнительные сведения см. в разделе Связи типов в операциях запроса LINQ. В следующем примере показан запрос по запросу ArrayList.

C#

```
ArrayList arrList = new ArrayList();
arrList.Add(
    new Student
    (
        FirstName: "Svetlana",
        LastName: "Omelchenko",
        ExamScores: new int[] { 98, 92, 81, 60 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Claire",
        LastName: "O'Donnell",
        ExamScores: new int[] { 75, 84, 91, 39 }
    ));
arrList.Add(
```

```
    new Student
    (
        FirstName: "Sven",
        LastName: "Mortensen",
        ExamScores: new int[] { 88, 94, 65, 91 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Cesar",
        LastName: "Garcia",
        ExamScores: new int[] { 97, 89, 85, 82 }
    ));

var query = from Student student in arrList
            where student.ExamScores[0] > 95
            select student;

foreach (Student s in query)
    Console.WriteLine(s.LastName + ": " + s.ExamScores[0]);
```

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Написание собственных расширений в LINQ

Все методы на основе LINQ соответствуют одному из двух аналогичных шаблонов. Они принимают перечисляемую последовательность. Они возвращают другую последовательность или одно значение. Согласованность фигуры позволяет расширить LINQ путем написания методов с аналогичной фигурой. На самом деле библиотеки .NET получили новые методы во многих выпусках .NET, так как LINQ впервые появился. В этой статье приведены примеры расширения LINQ путем написания собственных методов, которые соответствуют тому же шаблону.

Добавление пользовательских методов для запросов LINQ

Расширьте набор методов, используемых для запросов LINQ, добавив методы расширения в IEnumerable<T> интерфейс. Например, помимо стандартных средних или максимальных операций, вы создадите пользовательский агрегатный метод для вычисления одного значения из последовательности значений. Вы также создаете метод, который работает в качестве пользовательского фильтра или определенного преобразования данных для последовательности значений и возвращает новую последовательность. Примерами таких методов являются Distinct, Skipи Reverse.

При расширении IEnumerable<T> интерфейса можно применить пользовательские методы к любой перечисленной коллекции. Дополнительные сведения см. в разделе "Члены расширения".

Агрегатный метод вычисляет одно значение из набора значений. LINQ предоставляет несколько агрегатных методов, в том числе Average, Minи Max. Создайте собственный агрегатный метод, добавив метод расширения в IEnumerable<T> интерфейс.

Начиная с C# 14, можно объявить *блок расширения* для хранения нескольких членов расширения. Объявите блок расширения с ключевым словом `extension` , за которым следует параметр приемника в скобках. В следующем примере кода показано, как создать метод расширения с именем `Median` в блоке расширения. Метод вычисляет медиану для последовательности чисел типа `double`.

C#

```
extension(IEnumerable<double>? source)
{
    public double Median()
```

```
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null or
empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Можно также добавить модификатор `this` к статическому методу для объявления *метода расширения*. В следующем коде показан эквивалентный `Median` метод расширения:

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null or
empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
```

```
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Вызовите любой метод расширения для любой перечисленной коллекции так же, как и другие агрегатные методы из IEnumerable<T> интерфейса.

В следующем примере кода показано, как использовать `Median` метод для массива типов `double`.

C#

```
double[] numbers = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//     double: Median = 4.85
```

Перегрузите агрегатный метод таким образом, чтобы он принимал последовательности различных типов. Стандартный подход заключается в создании перегрузки для каждого типа. Другой подход заключается в создании перегрузки, которая принимает универсальный тип и преобразует ее в конкретный тип с помощью делегата. Вы также можете объединить оба подхода.

Создайте определенную перегрузку для каждого типа, который требуется поддерживать. В следующем примере кода показана перегрузка метода `Median` для типа `int`.

C#

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

Теперь можно вызвать `Median` перегрузки для обоих типов `integer` и `double`, как показано в следующем коде.

C#

```
double[] numbers1 = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = [1, 2, 3, 4, 5];
var query2 = numbers2.Median();
```

```
Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//     double: Median = 4.85
//     int: Median = 3
```

Вы также можете создать перегрузку, которая принимает *универсальную последовательность* объектов. Эта перегрузка принимает делегат в качестве параметра и использует его для преобразования последовательности объектов универсального типа в конкретный тип.

В следующем коде показана перегрузка метода `Median`, который принимает делегат Func<T,TResult> в качестве параметра. Этот делегат принимает объект универсального типа T и возвращает объект типа `double`.

C#

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

Теперь можно вызвать `Median` метод для последовательности объектов любого типа. Если тип не имеет собственной перегрузки метода, передайте параметр делегата. В C#для этой цели можно использовать лямбда-выражение. Кроме того, только в Visual Basic, если вы используете условие `Aggregate` или `Group By` вместо вызова метода, вы можете передать любое значение или выражение, которое находится в области действия этого условия.

В следующем примере кода показано, как вызвать `Median` метод для массива целых чисел и массив строк. Для строк вычисляется медиана длины строк в массиве. В примере показано, как передать параметр делегата Func<T,TResult> методу `Median` для каждого случая.

C#

```
int[] numbers3 = [1, 2, 3, 4, 5];

/*
    You can use the num => num lambda expression as a parameter for the Median
method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");
```

```
string[] numbers4 = ["one", "two", "three", "four", "five"];

// With the generic overload, you can also use numeric properties of objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//     int: Median = 3
//     string: Median = 4
```

IEnumerable<T> Расширьте интерфейс с помощью пользовательского метода запроса, который возвращает *последовательность значений*. В этом случае метод должен возвращать коллекцию типов IEnumerable<T>. Такие методы можно использовать для применения фильтров или преобразований данных к последовательности значений.

В следующем примере показано, как создать метод расширения с именем `AlternateElements` , который возвращает каждый другой элемент в коллекции, начиная с первого элемента.

C#

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}
```

Вызовите этот метод расширения для любой перечисленной коллекции так же, как и другие методы из IEnumerable<T> интерфейса, как показано в следующем коде:

C#

```
string[] strings = ["a", "b", "c", "d", "e"];

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
```

```
// This code produces the following output:
//     a
//     c
//     e
```

Каждый пример, показанный в этой статье, имеет другой *приемник*. Это означает, что каждый метод должен объявляться в отдельном блоке расширения, который определяет уникального приемника. В следующем примере кода показан один статический класс с тремя разными блоками расширений, каждый из которых содержит один из методов, определенных в этой статье:

C#

```
public static class EnumerableExtension
{
    extension(IEnumerable<double>? source)
    {
        public double Median()
        {
            if (source is null || !source.Any())
            {
                throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
            }

            var sortedList =
                source.OrderBy(number => number).ToList();

            int itemIndex = sortedList.Count / 2;

            if (sortedList.Count % 2 == 0)
            {
                // Even number of items.
                return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
            }
            else
            {
                // Odd number of items.
                return sortedList[itemIndex];
            }
        }
    }

    extension(IEnumerable<int> source)
    {
        public double Median() =>
            (from number in source select (double)number).Median();
    }

    extension<T>(IEnumerable<T> source)
    {
        public double Median(Func<T, double> selector) =>
```

```
            (from num in source select selector(num)).Median();

        public IEnumerable<T> AlternateElements()
        {
            int index = 0;
            foreach (T element in source)
            {
                if (index % 2 == 0)
                {
                    yield return element;
                }

                index++;
            }
        }
    }
}
```

Конечный блок расширения задает универсальный блок расширения. Параметр типа для приемника объявляется непосредственно в `extension`.

В предыдущем примере объявляется один член расширения в каждом блоке расширения. В большинстве случаев вы создаёте несколько членов расширения для одного приемника. В этих случаях объявите расширения для этих членов в одном блоке расширения.

ⓘ **Заметка:** Автор создал эту статью с помощью искусственного интеллекта. Подробнее

Last updated on 01.12.2025

Запрос на основе состояния времени выполнения

Статья • 25.04.2024

В большинстве запросов LINQ общая форма запроса устанавливается в коде. Вы можете фильтровать элементы с помощью предложения, сортировать выходную `where` коллекцию с помощью `orderby` элементов группы или выполнять некоторые вычисления. Код может предоставить параметры фильтра или ключа сортировки или другие выражения, которые являются частью запроса. Однако общая форма запроса не может измениться. Из этой статьи вы узнаете, как использовать System.Linq.IQueryable<T> интерфейс и типы, реализующие его для изменения формы запроса во время выполнения.

Эти методы используются для создания запросов во время выполнения, когда некоторые входные данные пользователя или состояние времени выполнения изменяют методы запроса, которые вы хотите использовать в рамках запроса. Вы хотите изменить запрос, добавив, удалив или изменив предложения запроса.

> ⓘ **Примечание**
>
> Убедитесь, что вы добавили `using System.Linq.Expressions;` и `using static System.Linq.Expressions.Expression;` в верхней части файла *.cs*.

Рассмотрим код, определяющий IQueryable или IQueryable<T> против источника данных:

C#

```
string[] companyNames = [
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
];

// Use an in-memory array as the data source, but the IQueryable could have
come
// from anywhere -- an ORM backed by a database, a web request, or any other
LINQ provider.
```

```
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Каждый раз, когда выполняется предыдущий код, выполняется тот же точный запрос. Давайте узнаем, как изменить запрос или изменить его. В своей основе IQueryable имеет два компонента:

- Expression— не зависящая от языка и источника данных представление компонентов текущего запроса в виде дерева выражений.
- Provider— экземпляр поставщика LINQ, который знает, как материализовать текущий запрос в значение или набор значений.

В контексте динамического запроса поставщик обычно остается неизменным; Дерево выражений запроса отличается от запроса к запросу.

Деревья выражений неизменяемы; Если требуется другое дерево выражений (и, таким образом, другой запрос), необходимо перевести существующее дерево выражений в новое. В следующих разделах описываются определенные методы запроса по-разному в ответ на состояние времени выполнения.

- Использование состояния времени выполнения из дерева выражений
- Вызов дополнительных методов LINQ
- Изменение дерева выражения, переданного в методы LINQ
- Создание дерева выражений Expression<TDelegate> с помощью методов фабрики в Expression
- Добавление узлов вызова метода в дерево выражения IQueryable.
- Создание строк и использование динамической библиотеки LINQ

Каждый из методов обеспечивает больше возможностей, но за счет повышения сложности.

Использование состояния времени выполнения из дерева выражений

Самый простой способ динамического запроса заключается в том, чтобы ссылаться на состояние времени выполнения непосредственно в запросе с помощью закрытой переменной, например `length` в следующем примере кода:

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
```

```
    .Distinct();

Console.WriteLine(string.Join(",", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(",", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

Дерево внутренних выражений (и таким образом запрос) не изменяется; Запрос возвращает разные значения, только так как значение изменено `length` .

Вызов дополнительных методов LINQ

Как правило, встроенные методы LINQ в Queryable выполняют два действия:

- Заключение текущего дерево выражения в оболочку в MethodCallExpression, который представляет вызов метода.
- Передача инкапсулированного дерева выражения в поставщик, чтобы вернуть значение через метод IQueryProvider.Execute поставщика, либо чтобы вернуть объект переведенного запроса с помощью метода IQueryProvider.CreateQuery.

Вы можете заменить исходный запрос результатом System.Linq.IQueryable<T>метода -returning, чтобы получить новый запрос. Состояние времени выполнения можно использовать, как показано в следующем примере:

C#

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

Изменение дерева выражения, переданного в методы LINQ

Вы можете передать различные выражения в методы LINQ в зависимости от состояния времени выполнения:

C#

```
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("" or null, "" or null) => x => true,
    (_, "" or null) => x => x.StartsWith(startsWith),
    ("" or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);
```

Кроме того, может потребоваться создать различные вложенные выражения с помощью другой библиотеки, такой как ПредикатBuilder LinqKit:

C#

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

Создание деревьев выражений и запросов с помощью фабричных методов

Во всех примерах до этого момента вы знаете тип элемента во время компиляции (`string` и таким образом тип запроса `IQueryable<string>`). Вы можете добавить

компоненты в запрос любого типа элемента или добавить различные компоненты в зависимости от типа элемента. Можно создавать деревья выражений с нуля, используя фабричные методы в System.Linq.Expressions.Expression и таким образом адаптировать выражение в среде выполнения к определенному типу элемента.

Создание TDelegate выражения<>

При создании выражения для передачи в один из методов LINQ вы фактически создаете экземпляр System.Linq.Expressions.Expression<TDelegate>, где `TDelegate` является некоторый тип делегата, например `Func<string, bool>` `Action`, или настраиваемый тип делегата.

System.Linq.Expressions.Expression<TDelegate> наследует от LambdaExpression, представляющее полное лямбда-выражение, как показано в следующем примере:

C#

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

LambdaExpression имеет два компонента:

1. Список параметров, `(string x)` представленный свойством Parameters .
2. Тело, `x.StartsWith("a")` представленное свойством Body .

Ниже приведены основные шаги по созданию объекта Expression<TDelegate> :

1. Определите объекты ParameterExpression для каждого из параметров (если таковые имеются) в лямбда-выражении с помощью фабричного метода Parameter.

   C#

   ```
   ParameterExpression x = Parameter(typeof(string), "x");
   ```

2. Создайте текст объектаLambdaExpression, используя ParameterExpression определенные и заводские методы.Expression Например, выражение, представляющее `x.StartsWith("a")`, может быть построено следующим образом:

   C#

   ```
   Expression body = Call(
       x,
       typeof(string).GetMethod("StartsWith", [typeof(string)])!,
   ```

```
    Constant("a")
);
```

3. Обтекайте параметры и текст в типизированном <выражении времени компиляции TDelegate>, используя соответствующую Lambda перегрузку метода фабрики:

```
C#

Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,
x);
```

В следующих разделах описывается сценарий, в котором может потребоваться создать метод Expression<TDelegate> LINQ. Он содержит полный пример того, как это сделать с помощью методов фабрики.

Создание полного запроса во время выполнения

Вы хотите написать запросы, которые работают с несколькими типами сущностей:

```
C#

record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

Для любого из этих типов сущностей необходимо отфильтровать и возвратить только те сущности, которые имеют заданный текст в одном из полей `string`. Для `Person` необходимо найти свойства `FirstName` и `LastName`:

```
C#

string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Но для `Car` требуется найти только свойство `Model`:

```
C#

string term = /* ... */;
var carsQry = new List<Car>()
```

```
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

Несмотря на то что можно написать одну настраиваемую функцию для `IQueryable<Person>` и другую для `IQueryable<Car>`, следующая функция добавляет эту фильтрацию в любой существующий запрос независимо от конкретного типа элемента.

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties = elementType
        .GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains",
[typeof(string)])!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
node like x.PropertyName.Contains("term")
            Call(                   // .Contains(...)
                Property(           // .PropertyName
                    prm,            // x
                    prp
                ),
                containsMethod,
                Constant(term)      // "term"
            )
        );

    // Combine all the resultant expression nodes using ||
    Expression body = expressions
```

```
        .Aggregate((prev, current) => Or(prev, current));

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

    // Because the lambda is compile-time-typed (albeit with a generic
parameter), we can use it with the Where method
    return source.Where(lambda);
}
```

`TextFilter` Так как функция принимает и возвращает IQueryable<T> (а не толькоIQueryable), вы можете добавить дополнительные элементы запроса во время компиляции после текстового фильтра.

C#

```
var qry = TextFilter(
        new List<Person>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
        new List<Car>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.Year == 2010);
```

Добавление узлов вызова метода в дерево выражений IQueryable<TDelegate>

Если у вас есть IQueryable вместо нее IQueryable<T>, вы не можете напрямую вызывать универсальные методы LINQ. Одним из вариантов является создание дерева внутренних выражений, как показано в предыдущем примере, и использование отражения для вызова соответствующего метода LINQ во время передачи в дерево выражений.

Можно также дублировать функциональность метода LINQ, заключив все дерево в MethodCallExpression, который представляет вызов метода LINQ:

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;
```

```
    // The logic for building the ParameterExpression and the
LambdaExpression's body is the same as in the previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
constructBody(elementType, term);
    if (body is null) { return source; }

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        [elementType],
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}
```

В этом случае у вас нет универсального заполнителя времени `T` компиляции, поэтому вы используете Lambda перегрузку, которая не требует сведений о типе компиляции, и которая создает LambdaExpression вместо него Expression<TDelegate>.

Динамическая библиотека LINQ

Создание деревьев выражений с помощью методов фабрики является относительно сложным; проще создавать строки. Динамическая библиотека LINQ предоставляет набор методов расширения для IQueryable, соответствующих стандартным методам LINQ в Queryable, и который принимает строки в специальном синтаксисе вместо деревьев выражений. Библиотека создает соответствующее дерево выражения из строки и может возвращать результирующий преобразованный IQueryable.

Например, предыдущий пример можно переписать следующим образом:

C#

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
```

```
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
    );

    return source.Where(filterExpr, term);
}
```

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Асинхронное программирование с операторами async и await

Статья • 27.03.2025

Модель асинхронного программирования задач (TAP) предоставляет уровень абстракции над типичным асинхронным кодированием. В этой модели код записывается как последовательность операторов, так же, как обычно. Разница в том, что вы можете читать код на основе задач одновременно с тем, как компилятор обрабатывает каждую инструкцию, и перед тем как он начнет обработку следующей инструкции. Для выполнения этой модели компилятор выполняет множество преобразований для выполнения каждой задачи. Некоторые инструкции могут инициировать работу и возвращать объект Task, представляющий текущую работу, и компилятор должен устранить эти преобразования. Цель асинхронного программирования задач — сделать код, который воспринимается как последовательность операторов, но исполняется в более сложной последовательности. Исполнение основано на выделении внешних ресурсов и завершении задач.

Модель асинхронного программирования задачи аналогична тому, как люди дают инструкции для процессов, включающих асинхронные задачи. В этой статье приводится пример с инструкциями по приготовлению завтрака, чтобы показать, как ключевые слова `async` и `await` упрощают понимание кода, содержащего серию асинхронных инструкций. Инструкции по созданию завтрака могут быть предоставлены в виде списка:

1. Налить чашку кофе.
2. Разогрейте кастрюлю, а затем жарите два яйца.
3. Жарить три среза бекона.
4. Поджарьте два кусочка хлеба.
5. Разложите масло и варенье на тост.
6. Налить стакан апельсинового сока.

Если у вас есть опыт приготовления пищи, вы можете выполнить эти инструкции **асинхронно**. Вы начинаете разогревать кастрюлю для яиц, а затем начать жарить бекон. Вы положили хлеб в тостер, а затем начните готовить яйца. На каждом шаге процесса вы запускаете задачу, а затем переходите к другим задачам, которые готовы к вашему внимания.

Приготовление завтрака является хорошим примером асинхронной работы, которая не параллельна. Один человек (или поток) может обрабатывать все

задачи. Один человек может готовить завтрак без задержек, начиная следующую задачу до завершения предыдущей задачи. Каждая задача приготовления пищи выполняется независимо от того, активно ли кто-то наблюдает за процессом. Как только вы начнете разогревать кастрюлю для яиц, вы можете начать жарить бекон. Как только бекон начнет готовиться, можно положить хлеб в тостер.

Для параллельного алгоритма требуется несколько людей, которые готовят (или несколько потоков). Один человек готовит яйца, другой жарит бекон, и т. д. Каждый человек фокусируется на одной конкретной задаче. Каждый, кто готовит (или каждый поток), блокируется синхронно, ожидая, пока завершится текущая задача: бекон пора переворачивать, хлеб вот-вот всплывет из тостера и т. д.



Рассмотрим тот же список синхронных инструкций, написанных в виде операторов кода C#.

```C#
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
```

```
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
```

```
        }

        private static Bacon FryBacon(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

Если вы интерпретируете эти инструкции как компьютер, завтрак занимает около 30 минут, чтобы подготовиться. Длительность — это сумма времени отдельной задачи. Компьютер блокирует каждую инструкцию до завершения всех работ, а затем переходит к следующей инструкции задачи. Этот подход может занять значительное время. В примере завтрака метод компьютера создает ненасытный завтрак. Поздние задачи в синхронном списке, такие как поджаривание хлеба, не начинаются, пока не завершатся предыдущие задачи. Некоторая еда остывает, прежде чем завтрак готов к подаче.

Если вы хотите, чтобы компьютер выполнял инструкции асинхронно, необходимо написать асинхронный код. При написании клиентских программ необходимо,

чтобы пользовательский интерфейс реагировал на входные данные пользователей. Приложение не должно заморозить все взаимодействие при скачивании данных из Интернета. При написании серверных программ не требуется блокировать потоки, которые могут обслуживать другие запросы. Использование синхронного кода, когда существуют асинхронные альтернативы, вредит вашей способности более экономно масштабироваться. Вы оплачиваете заблокированные потоки.

Для успешных современных приложений требуется асинхронный код. Без поддержки языка при написании асинхронного кода требуются обратные вызовы, события завершения или другие средства, которые скрывают исходное намерение кода. Преимущество синхронного кода — это пошаговое действие, которое упрощает сканирование и понимание. Традиционные асинхронные модели позволяют сосредоточиться на асинхронном характере кода, а не на фундаментальных действиях кода.

Не блокируйте, ожидайте вместо этого

Предыдущий код подчеркивает неудачную практику программирования: написание синхронного кода для выполнения асинхронных операций. Код блокирует текущий поток от выполнения любой другой работы. Код не прерывает поток во время выполнения задач. Результат этой модели похож на то, как вы смотрите на тостер после того, как засунули в него хлеб. Вы игнорируете любые прерывания и не запускаете другие задачи, пока не появится хлеб. Вы не берете масло и варенье из холодильника. Возможно, вы не увидите пожар, начинающийся на плите. Вы хотите одновременно и поджаривать хлеб, и заниматься другими делами. То же самое верно для твоего кода.

Для начала можно обновить код, чтобы поток не блокируется во время выполнения задач. Ключевое слово `await` предоставляет неблокирующий способ начать задачу, а затем продолжить выполнение после её завершения. Простая асинхронная версия кода завтрака выглядит так:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");
```

```
    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

Код обновляет тела методов `FryEggsAsync`, `FryBaconAsync` и `ToastBreadAsync` для возврата `Task<Egg>`, `Task<Bacon>` и `Task<Toast>` объектов соответственно. Имена методов включают суффикс Async. Метод `Main` возвращает объект `Task`, хотя у него нет выражения `return`, что предусмотрено по замыслу. Для получения дополнительной информации см. раздел Оценка асинхронной функции, не возвращающей значение.

> ⓘ **Примечание**
>
> Обновленный код еще не использует ключевые функции асинхронного программирования, что может привести к более короткому времени завершения. Код обрабатывает задачи примерно столько же времени, сколько и начальная синхронная версия. Полные реализации методов см. в окончательной версии кода , представленной далее в этой статье.

Давайте применим пример завтрака к обновленному коду. Поток не блокируется во время приготовления яиц или бекона, однако код также не запускает другие задачи до тех пор, пока текущая работа не завершится. Вы по-прежнему кладете хлеб в тостер и следите за тостером, пока хлеб не выскакивает, но теперь вы можете реагировать на отвлечения. В ресторане, где делают несколько заказов, повар может начать новый заказ, пока другой уже готовится.

В обновленном коде поток, работающий над завтраком, не блокируется во время ожидания любой запущенной задачи, которая не завершена. Для некоторых приложений это изменение необходимо. Вы можете включить ваше приложение для поддержки взаимодействия с пользователем во время загрузки данных из Интернета. В других сценариях может потребоваться запустить другие задачи, ожидая завершения предыдущей задачи.

Одновременные запуски задач

Для большинства операций необходимо немедленно запустить несколько независимых задач. По завершении каждой задачи вы инициируете другую работу, готовую к началу. При применении этой методологии к примеру завтрака вы можете быстрее подготовить завтрак. Вы также готовите все практически в одно и то же время, чтобы вы могли насладиться горячим завтраком.

Класс System.Threading.Tasks.Task и связанные типы — это классы, которые можно использовать для применения этого стиля рассуждений к задачам, которые выполняются. Этот подход позволяет писать код, который более тесно похож на то, как вы создаете завтрак в реальной жизни. Вы начинаете готовить яйца, бекон и тост одновременно. Так как каждый элемент пищи требует действий, вы обращаете внимание на эту задачу, заботитесь о действии, а затем ждете чего-то другого, требующего вашего внимания.

В коде вы запускаете задачу и удерживаете объект Task, представляющий работу. Вы используете метод `await` для задачи, чтобы отсрочить выполнение работы до тех пор, пока результат не будет готов.

Примените эти изменения к коду завтрака. Первым шагом является хранение задач для операций при запуске, а не использование выражения `await`:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Эти изменения не ускоряют подготовку вашего завтрака. Выражение `await` применяется ко всем задачам сразу после их запуска. Следующий шаг —

переместить выражения `await` для бекона и яиц в конец метода перед тем, как подать завтрак.

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

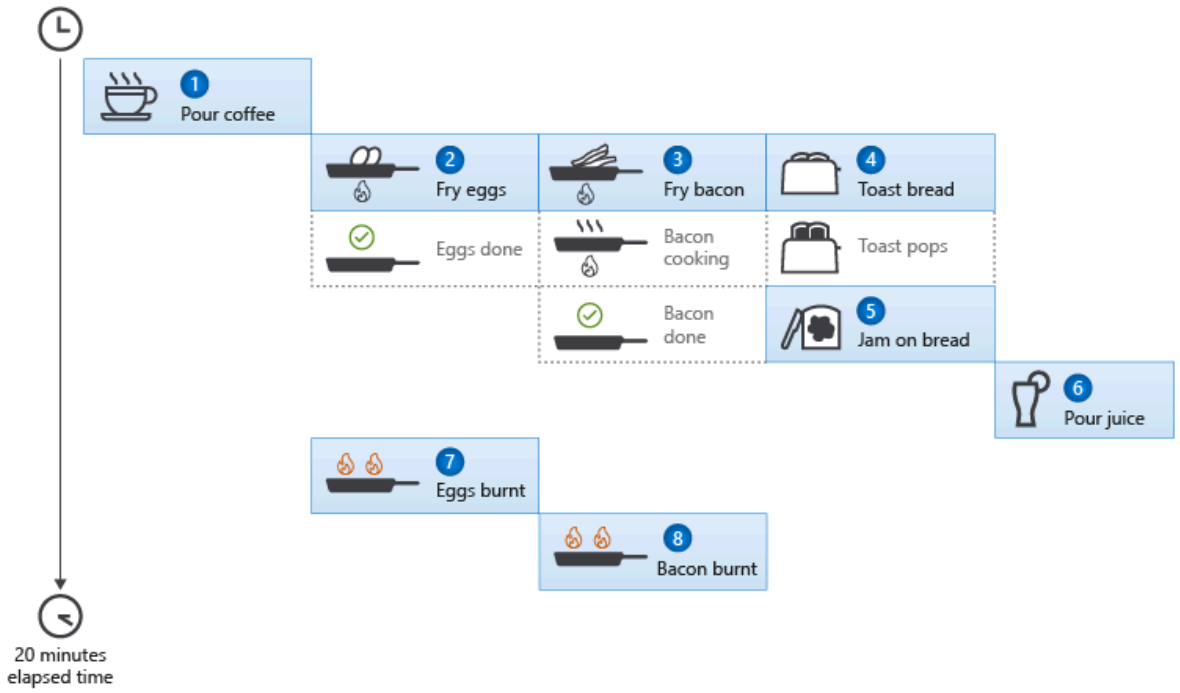
Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

Теперь у вас есть асинхронно подготовленный завтрак, который занимает около 20 минут, чтобы подготовиться. Общее время приготовления уменьшается, так как некоторые задачи выполняются одновременно.



Обновления кода улучшают процесс подготовки, уменьшая время приготовления, но они вводят регрессию путем сжигания яиц и бекона. Вы одновременно запускаете все асинхронные задачи. Вы ожидаете выполнения каждой задачи только тогда, когда вам нужны результаты. Код может быть похож на программу в веб-приложении, которая выполняет запросы к различным микрослужбам, а затем объединяет результаты на одну страницу. Вы выполняете все запросы немедленно, а затем применяете выражение `await` ко всем этим задачам и создаете веб-страницу.

Поддержка сочетания с задачами

Предыдущие редакции кода помогают подготовить всё к завтраку одновременно, за исключением тоста. Процесс приготовления тостов является *композицией* асинхронной операции (поджаривание хлеба) с синхронными операциями (намазывание масла и джема на тост). В этом примере показана важная концепция асинхронного программирования:

> **ⓘ Важно!**
>
> Композиция асинхронной операции, за которой следует синхронная работа, является асинхронной операцией. Другими словами, если любая часть операции является асинхронной, то вся операция является асинхронной.

В предыдущих обновлениях вы узнали, как использовать объекты Task или Task<TResult> для выполнения задач. Вы ожидаете каждой задачи, прежде чем использовать его результат. Следующим шагом является создание методов, представляющих сочетание других работ. Прежде чем подавать завтрак, подождите, пока закончится задача, которая подразумевает поджаривание хлеба, прежде чем вы намазываете хлеб маслом и вареньем.

Вы можете представить эту работу со следующим кодом:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

Метод `MakeToastWithButterAndJamAsync` имеет модификатор `async` в сигнатуре, который сигнализирует компилятору о том, что метод содержит выражение `await` и содержит асинхронные операции. Метод заключается в поджаривании хлеба, а затем намазывании масла и варенья. Метод возвращает объект Task<TResult>, представляющий состав трех операций.

Измененный основной блок кода теперь выглядит следующим образом:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

Это изменение кода иллюстрирует важный способ работы с асинхронным кодом. Вы создаете задачи, разделяя операции на новый метод, который возвращает задачу. Вы можете выбрать, когда ждать этой задачи. Вы можете одновременно запускать другие задачи.

Обработка асинхронных исключений

До этого момента код неявно предполагает успешное выполнение всех задач. Асинхронные методы вызывают исключения, как и их синхронные аналоги. Цели асинхронной поддержки исключений и обработки ошибок совпадают с асинхронной поддержкой в целом. Наилучшей практикой является писать код, который читается как ряд синхронных операторов. Задачи вызывают исключения,

когда они не могут завершиться успешно. Клиентский код может перехватывать эти исключения при применении выражения `await` к запущенной задаче.

В примере с завтраком предположим, что тостер загорится во время поджаривания хлеба. Эту проблему можно имитировать, изменив метод `ToastBreadAsync`, чтобы он соответствовал следующему коду:

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

> ⓘ **Примечание**
>
> При компиляции этого кода отображается предупреждение о недоступном коде. Эта ошибка сделана намеренно. После того, как тостер загорается, операции не выполняются нормально, и код возвращает ошибку.

После внесения изменений кода запустите приложение и проверьте выходные данные:

Консоль

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
```

```
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

Обратите внимание, что довольно много задач завершается между временем, когда тостер загорается и система наблюдает исключение. Когда задача, которая выполняется асинхронно, создает исключение, эта задача **неисправна**. Объект `Task` содержит исключение, возникающее в свойстве Task.Exception. Неисправные задачи вызывают исключение, если выражение `await` применяется к задаче.

Существует два важных механизма для понимания этого процесса.

- Как исключение хранится в неисправной задаче
- Как исключение распаковывается и повторно выполняется при ожидании кода (`await`) в неисправной задаче

При асинхронном выполнении кода, когда выбрасывается исключение, оно хранится в объекте `Task`. Свойство Task.Exception является объектом System.AggregateException, так как во время асинхронной работы может возникать несколько исключений. Всякое выброшенное исключение добавляется в коллекцию AggregateException.InnerExceptions. Если свойство `Exception` имеет значение NULL, создается новый объект `AggregateException`, а исключение вызывается первым элементом в коллекции.

Наиболее распространенный сценарий для неисправной задачи заключается в том, что свойство `Exception` содержит ровно одно исключение. Когда ваш код ожидает задачи с ошибкой, он повторно выбрасывает первое исключение AggregateException.InnerExceptions из коллекции. Результатом этого является причина, по которой выходные данные из примера показывают объект System.InvalidOperationException, а не объект `AggregateException`. Извлечение первого внутреннего исключения приближает работу с асинхронными методами к работе с их синхронными аналогами максимально похожей. Вы можете проверить свойство `Exception` в коде, если сценарий может создать несколько исключений.

> **Совет**
>
> Рекомендуется использовать любые исключения проверки аргументов, которые возникают *синхронно* из методов возврата задач. Для получения дополнительной информации и примеров см. раздел **Исключения в методах, возвращающих задачи**.

Прежде чем продолжить переход к следующему разделу, закомментируйте следующие две инструкции в методе `ToastBreadAsync`. Вы не хотите начать еще один огонь:

C#

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Эффективное применение выражений await к задачам

Вы можете улучшить ряд выражений `await` в конце предыдущего кода с помощью методов класса `Task`. Один API — это метод WhenAll, который возвращает объект Task, который завершается после завершения всех задач в списке аргументов. Следующий код демонстрирует этот метод:

C#

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Другой вариант — использовать метод WhenAny, который возвращает объект `Task<Task>`, который завершается после завершения любого из его аргументов. Вы можете ждать возвращаемой задачи, так как вы знаете, что задача выполнена. В следующем коде показано, как использовать метод WhenAny, чтобы ждать завершения первой задачи, а затем обработать его результат. После обработки результата из завершенной задачи удалите завершенную задачу из списка задач, переданных методу `WhenAny`.

C#

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    await finishedTask;
    breakfastTasks.Remove(finishedTask);
}
```

В конце фрагмента кода обратите внимание на выражение `await finishedTask;`. Выражение `await Task.WhenAny` не ожидает завершения задачи, а ожидает объекта `Task`, возвращаемого методом `Task.WhenAny`. Результатом метода `Task.WhenAny` является завершенная (или неисправная) задача. Лучше подождать с выполнением задачи снова, даже если вы знаете, что задача завершена. Таким образом, вы можете получить результат задачи или обеспечить выброс любого исключения, которое вызывает сбой задачи.

Просмотр окончательного кода

Вот как выглядит окончательная версия кода:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }
```

```
    class Program
    {
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                await finishedTask;
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
            ApplyJam(toast);

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");
```

```
        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static async Task<Bacon> FryBaconAsync(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            await Task.Delay(3000);
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            await Task.Delay(3000);
            Console.WriteLine("Put bacon on plate");

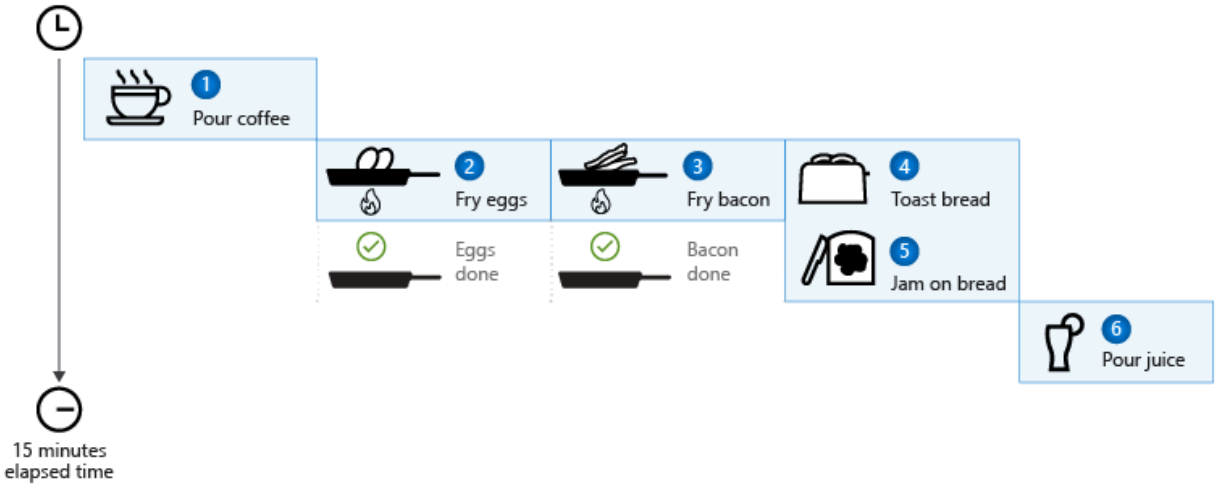
            return new Bacon();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

Код завершает асинхронные задачи завтрака за примерно 15 минут. Общее время сокращается, так как некоторые задачи выполняются одновременно. Код одновременно отслеживает несколько задач и выполняет действия только по мере необходимости.



Окончательный код является асинхронным. Это более точно отражает, как человек может готовить завтрак. Сравните окончательный код с первым примером кода в статье. Основные действия по-прежнему понятны, считывая код. Вы можете прочитать окончательный код так же, как вы читаете список инструкций по созданию завтрака, как показано в начале статьи. Функции языка для ключевых слов `async` и `await` предоставляют перевод, который каждый человек делает, чтобы следовать письменным инструкциям: начинать задачи, когда можно, и не блокировать выполнение, ожидая завершения задач.

Следующий шаг

Исследуйте реальные сценарии для асинхронных программ

Сценарии асинхронного программирования

16.08.2025

Если код реализует сценарии с привязкой ввода-вывода для поддержки запросов к сетевым данным, доступа к базе данных или операций чтения и записи файловой системы, асинхронное программирование является лучшим подходом. Вы также можете написать асинхронный код для сценариев, связанных с ЦП, таких как дорогостоящие вычисления.

C# имеет асинхронную модель программирования на уровне языка, которая позволяет легко писать асинхронный код без необходимости переключать обратные вызовы или соответствовать библиотеке, поддерживающей асинхронность. Модель следует асинхронному шаблону на основе задач, известному как (TAP).

Изучение асинхронной модели программирования

Объекты `Task` и `Task<T>` представляют ядро асинхронного программирования. Эти объекты используются для моделирования асинхронных операций, поддерживая ключевые слова `async` и `await`. В большинстве случаев модель довольно проста для сценариев с привязкой ввода-вывода и ЦП. Внутри метода `async`:

- **код с привязкой ввода-вывода** запускает операцию, представленную объектом `Task` или `Task<T>` в методе `async`.
- ЦП-зависимый код .

В обоих случаях активная `Task` представляет асинхронную операцию, которая может быть не завершена.

Именно с помощью ключевого слова `await` творится вся магия. Он передает управление вызывающему метод, содержащий выражение `await`, и в конечном счете позволяет пользовательскому интерфейсу быть отзывчивым, а службе — гибкой. Хотя существуют способы подходить к асинхронному коду помимо использования выражений `async` и `await`, в этой статье основное внимание уделяется языковым конструкциям.

> ⓘ **Примечание**

В некоторых примерах, представленных в этой статье, используется класс System.Net.Http.HttpClient для скачивания данных из веб-службы. В примере кода объект `s_httpClient` является статическим полем класса `Program` типа:

```
private static readonly HttpClient s_httpClient = new();
```

Для получения дополнительной информации, например, полный пример кода находится в конце этой статьи.

Ознакомьтесь с основными понятиями

При реализации асинхронного программирования в коде C# компилятор преобразует программу в компьютер состояния. Эта конструкция отслеживает различные операции и состояние в коде, например выполнение, когда код достигает выражения `await`, а также возобновляет выполнение после завершения фонового задания.

С точки зрения теории информатики асинхронное программирование является реализацией модели Promise асинхронности.

В модели асинхронного программирования существует несколько ключевых понятий, которые необходимо понять:

- Вы можете использовать асинхронный код для привязанного к вводу-выводу и привязанного к ЦП кода, но реализация отличается.
- Асинхронный код использует объекты `Task<T>` и `Task` в качестве конструкций для моделирования работы в фоновом режиме.
- Ключевое слово `async` объявляет метод асинхронным методом, что позволяет использовать ключевое слово `await` в тексте метода.
- При применении ключевого слова `await` код приостанавливает вызывающий метод и возвращает управление вызывающей стороне до завершения задачи.
- Выражение `await` можно использовать только в асинхронном методе.

Пример задачи, зависящей от ввода-вывода: загрузка данных из веб-сервиса

В этом примере, когда пользователь выбирает кнопку, приложение скачивает данные из веб-службы. Вы не хотите блокировать поток пользовательского интерфейса для приложения во время загрузки. Следующий код выполняет эту задачу:

```
C#
```

```
s_downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await s_httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

В этом коде намерение (скачивание данных в асинхронном режиме) выражается без запутанных операций с объектами `Task`.

Пример, привязанный к ЦП: выполнение вычисления игры

В следующем примере мобильная игра наносит урон нескольким агентам на экране в ответ на нажатие кнопки. Выполнение вычисления ущерба может быть дорогостоящим. При выполнении вычисления в потоке пользовательского интерфейса могут возникнуть проблемы с отображением и взаимодействием пользовательского интерфейса во время вычисления.

Лучший способ обрабатывать задачу — запустить фоновый поток для завершения работы с помощью метода `Task.Run`. Операция даёт результаты посредством использования выражения `await`. Операция возобновляется после завершения задачи. Такой подход позволяет пользовательскому интерфейсу работать плавно, пока работа завершается в фоновом режиме.

C#

```
static DamageResult CalculateDamageDone()
{
    return new DamageResult()
    {
        // Code omitted:
        //
        // Does an expensive calculation and returns
        // the result of that calculation.
    };
}

s_calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
```

```
    DisplayDamage(damageResult);
};
```

Код четко выражает намерение события кнопки `Clicked`. Для этого не требуется управлять фоновым потоком вручную, и задача завершается без блокировки.

Распознавание сценариев с привязкой ЦП и операций ввода-вывода

В предыдущих примерах показано, как использовать модификатор `async` и выражение `await` для ввода-вывода и процессорных задач. Пример для каждого сценария показывает, как код отличается в зависимости от того, где связана операция. Чтобы подготовиться к реализации, необходимо понять, как определить, когда операция привязана к операции ввода-вывода или ЦП. Выбор реализации может значительно повлиять на производительность кода и потенциально привести к неправильному применению конструкций.

Перед написанием любого кода необходимо задать два основных вопроса:

Развернуть таблицу

Вопрос	Сценарий	Реализация
Следует ли код ожидать результата или действия, например данных из базы данных?	**ограничение по вводу-выводу**	Используйте модификатор `async` и выражение `await` *без* метода `Task.Run`. Избегайте использования параллельной библиотеки задач.
Должен ли код выполнять дорогостоящие вычисления?	с привязкой к ЦП	Используйте модификатор `async` и выражение `await`, но отключите работу над другим потоком с помощью метода `Task.Run`. Этот подход устраняет проблемы с скоростью реагирования ЦП. Если к задаче применим параллелизм, рассмотрите возможность использования библиотеки параллельных задач.

Всегда измеряйте выполнение кода. Возможно, вы обнаружите, что работа, связанная с ЦП, недостаточно затратна по сравнению с затратами на переключения контекста при многопоточной работе. Каждый выбор имеет компромиссы. Выберите правильный компромисс для вашей ситуации.

Изучение других примеров

Примеры в этом разделе демонстрируют несколько способов написания асинхронного кода в C#. Они охватывают несколько сценариев, которые могут возникнуть.

Извлечение данных из сети

Следующий код загружает HTML из заданного URL-адреса и подсчитывает количество случаев, когда строка .NET возникает в HTML. Код использует ASP.NET для определения метода контроллера веб-API, который выполняет задачу и возвращает количество.

> ⓘ **Примечание**
>
> Если вы планируете проанализировать HTML в рабочем коде, не используйте регулярные выражения. Используйте библиотеку парсинга.

C#

```
[HttpGet, Route("DotNetCount")]
static public async Task<int> GetDotNetCountAsync(string URL)
{
    // Suspends GetDotNetCountAsync() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await s_httpClient.GetStringAsync(URL);
    return Regex.Matches(html, @"\.NET").Count;
}
```

Вы можете написать аналогичный код для универсального приложения Windows и выполнить задачу подсчета после нажатия кнопки:

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask =
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
Progress Bar.
    // It's important to do the extra work here before the "await" call,
    // so the user sees the progress bar before execution of this method is
yielded.
    NetworkProgressBar.IsEnabled = true;
```

```
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control
to its caller.
    // This action is what allows the app to be responsive and not block the UI
thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Ожидание выполнения нескольких задач

В некоторых сценариях код должен одновременно извлекать несколько фрагментов данных. API-интерфейсы `Task` предоставляют методы, которые позволяют вам писать асинхронный код для неблокирующего ожидания выполнения нескольких фоновых задач.

- метод Task.WhenAll
- метод Task.WhenAny

В следующем примере показано, как можно получить данные объекта `User` для набора объектов `userId`.

C#

```
private static async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.

    return await Task.FromResult(new User() { id = userId });
}

private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }
```

```
    return await Task.WhenAll(getUserTasks);
}
```

Этот код можно писать более кратко с помощью LINQ:

C#

```
private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

Хотя вы пишете меньше кода с помощью LINQ, следует соблюдать осторожность при сочетании LINQ с асинхронным кодом. LINQ использует отложенное (или ленивое) выполнение, что означает, что без немедленной обработки асинхронные вызовы не выполняются до тех пор, пока не будет перечислена последовательность.

Предыдущий пример является правильным и безопасным, так как он использует Enumerable.ToArray метод для немедленной оценки запроса LINQ и хранения задач в массиве. Этот подход гарантирует `id => GetUserAsync(id)`, что вызовы выполняются немедленно и все задачи запускаются одновременно, как и подход цикла `foreach`. Всегда используйте Enumerable.ToArray или Enumerable.ToList при создании задач с помощью LINQ, чтобы обеспечить немедленное выполнение и параллельное выполнение задач. Ниже приведен пример, демонстрирующий использование `ToList()` для `Task.WhenAny` обработки задач по мере их выполнения:

C#

```
private static async Task ProcessTasksAsTheyCompleteAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToList();

    while (getUserTasks.Count > 0)
    {
        Task<User> completedTask = await Task.WhenAny(getUserTasks);
        getUserTasks.Remove(completedTask);

        User user = await completedTask;
        Console.WriteLine($"Processed user {user.id}");
    }
}
```

В этом примере `ToList()` создается список, поддерживающий `Remove()` операцию, что позволяет динамически удалять завершенные задачи. Этот шаблон особенно полезен,

если вы хотите обрабатывать результаты сразу после их доступности, а не ожидая завершения всех задач.

Хотя вы пишете меньше кода с помощью LINQ, следует соблюдать осторожность при сочетании LINQ с асинхронным кодом. LINQ использует отложенное (или отложенное) выполнение. Асинхронные вызовы не происходят немедленно, так как они выполняются в `foreach` цикле, если только не принудительно выполняется итерация созданной последовательности с вызовом `.ToList()` метода или `.ToArray()` метода.

Вы можете выбрать вариант между Enumerable.ToArray сценариями и Enumerable.ToList на основе следующих вариантов:

- Используйте `ToArray()` при планировании обработки всех задач вместе, например с `Task.WhenAll`. Массивы эффективны для сценариев, когда размер коллекции фиксирован.
- Используйте `ToList()` , когда необходимо динамически управлять задачами, например с тем, где `Task.WhenAny` можно удалить завершенные задачи из коллекции по мере их завершения.

Ознакомьтесь с рекомендациями по асинхронным программированию

При асинхронном программировании следует учитывать несколько деталей, которые могут предотвратить непредвиденное поведение.

Используйте await внутри тела метода async()

При использовании модификатора `async` необходимо включить в текст метода один или несколько выражений `await`. Если компилятор не сталкивается с выражением `await`, метод не возвращает результат. Хотя компилятор создает предупреждение, код по-прежнему компилируется, а компилятор запускает метод. Компьютер состояния, созданный компилятором C# для асинхронного метода, не выполняет ничего, поэтому весь процесс является очень неэффективным.

Добавление суффикса Async в асинхронные имена методов

Соглашение о стиле .NET заключается в добавлении суффикса Async ко всем именам асинхронных методов. Этот подход помогает более легко различать синхронные и асинхронные методы. Некоторые методы, которые не вызываются явным образом в коде

(например, обработчики событий или методы веб-контроллера), не обязательно применяются в этом сценарии. Так как эти элементы не вызываются явным образом в коде, использование явного именования не так важно.

Возвращайте 'async void' только из обработчиков событий

Обработчики событий должны объявлять типы возвращаемых значений как `void` и не могут использовать или возвращать объекты `Task` и `Task<T>`, в отличие от других методов. При написании асинхронных обработчиков событий необходимо использовать модификатор `async` в методе, возвращающем `void`, для обработчиков. Другие реализации методов, возвращающих `async void`, не соответствуют модели TAP и могут создавать трудности.

- Исключения, выбрасываемые в методе `async void`, не могут быть пойманы за пределами этого метода.
- `async void` методы сложно тестировать
- методы `async void` могут вызвать негативные побочные эффекты, если вызывающий не ожидает, что они будут асинхронными.

Будьте осторожны с асинхронными лямбда-выражениями в LINQ

При внедрении асинхронных лямбда-выражений в выражениях LINQ важно проявлять осторожность. Лямбда-выражения в LINQ используют отложенное выполнение, что означает, что код может выполняться в непредвиденное время. Введение блокирующих задач в этот сценарий может легко привести к взаимоблокировке, если код не написан правильно. Кроме того, вложенность асинхронного кода также может усложнить понимание исполнения этого кода. Асинхронные операции и LINQ являются мощными, но эти методы следует использовать вместе настолько тщательно и ясно, насколько это возможно.

Выполнение задач в неблокирующем режиме

Если вашей программе нужен результат задачи, напишите код, реализующий выражение `await` в неблокирующем режиме. Блокировка текущего потока в качестве средства для синхронного ожидания завершения элемента `Task` может привести к взаимоблокировкам и заблокированным потокам контекста. Этот подход программирования может требовать более сложной обработки ошибок. В следующей таблице приведены рекомендации по доступу к результатам задач без их блокировки.

Развернуть таблицу

Сценарий задачи	Текущий код	Замените на 'await'
получение результата фоновой задачи	`Task.Wait` или `Task.Result`	`await`
Продолжить после выполнения любой задачи	`Task.WaitAny`	`await Task.WhenAny`
*Продолжить, когда **все задачи** завершены*	`Task.WaitAll`	`await Task.WhenAll`
Продолжить через некоторое время	`Thread.Sleep`	`await Task.Delay`

Рекомендуется использовать тип ValueTask

Когда асинхронный метод возвращает объект `Task`, узкие места производительности могут возникать в определенных сценариях. Так как `Task` является ссылочным типом, объект `Task` выделяется из кучи. Если метод, объявленный с помощью модификатора `async`, возвращает кэшированный результат или завершается синхронно, дополнительные выделения могут приводить к значительным временным затратам в критически важных секциях кода. Этот сценарий может стать дорогостоящим, когда выделение происходит в жестких циклах. Дополнительные сведения см. в разделе Обобщенные асинхронные типы возвращаемых значений.

Понимание того, когда следует устанавливать ConfigureAwait(false)

Разработчики часто спрашивают, когда следует использовать булевый Task.ConfigureAwait(Boolean). Этот API позволяет `Task` экземпляру настроить контекст для конечного компьютера, реализующего любое выражение `await`. Если булево значение не задано правильно, производительность может снизиться или могут произойти взаимоблокировки. Дополнительные сведения см. в разделе Вопросы и ответы о ConfigureAwait.

Пишите менее состоятельный код

Избегайте написания кода, зависящее от состояния глобальных объектов или выполнения определенных методов. Вместо этого, полагайтесь только на возвращаемые методами значения. Существует множество преимуществ написания кода, который является менее использующим состояние.

- Легче анализировать код
- Проще тестировать код

- Более простое сочетание асинхронного и синхронного кода
- Возможность избежать условий гонки в коде
- Легко координировать асинхронный код, зависящий от возвращаемых значений
- (Бонус) Хорошо совместимо с внедрением зависимостей в код.

Следует стремиться к достижению полной или почти полной ссылочной прозрачности в коде. Такой подход приводит к прогнозируемой, тестируемой и поддерживаемой базе кода.

Синхронный доступ к асинхронным операциям

В сценариях может потребоваться блокировать асинхронные операции, если `await` ключевое слово недоступно во всем стеке вызовов. Эта ситуация возникает в устаревших базах кода или при интеграции асинхронных методов в синхронные API, которые нельзя изменить.

> ⚠ **Предупреждение**
>
> Синхронная блокировка для асинхронных операций может привести к взаимоблокировкам и должна избегаться всякий раз, когда это возможно. Предпочтительное решение — использовать `async`/`await` в стеке вызовов.

Если вы должны блокировать синхронно на a `Task`, ниже приведены доступные подходы, перечисленные от большинства до наименее предпочтительных:

- Используйте GetAwaiter(). GetResult()
- Использование Task.Run для сложных сценариев
- Использование wait() и result

Используйте GetAwaiter(). GetResult()

Шаблон `GetAwaiter().GetResult()` обычно является предпочтительным подходом, если необходимо заблокировать синхронно:

C#

```
// When you cannot use await
Task<string> task = GetDataAsync();
string result = task.GetAwaiter().GetResult();
```

Такой подход:

- Сохраняет исходное исключение, не обтекая его в `AggregateException`.
- Блокирует текущий поток до завершения задачи.
- По-прежнему несет риск взаимоблокировки, если не используется тщательно.

Использование Task.Run для сложных сценариев

Для сложных сценариев, в которых необходимо изолировать асинхронную работу:

```
C#

// Offload to thread pool to avoid context deadlocks
string result = Task.Run(async () => await
GetDataAsync()).GetAwaiter().GetResult();
```

Этот шаблон:

- Выполняет асинхронный метод в потоке пула потоков.
- Может помочь избежать некоторых сценариев взаимоблокировки.
- Добавляет затраты путем планирования работы в пул потоков.

Использование wait() и result

Вы можете использовать блокирующий подход, вызывая Wait() и Result. Однако этот подход не рекомендуется, так как он упаковывает исключения в AggregateException.

```
C#

Task<string> task = GetDataAsync();
task.Wait();
string result = task.Result;
```

Проблемы с `Wait()` и `Result`:

- Исключения упаковываются в `AggregateException` оболочку, что делает обработку ошибок более сложной.
- Более высокий риск взаимоблокировки.
- Менее четкое намерение в коде.

Дополнительные рекомендации

- Предотвращение взаимоблокировки. Будьте особенно осторожны в приложениях пользовательского интерфейса или при использовании контекста синхронизации.
- Влияние на производительность. Блокировка потоков снижает масштабируемость.

- Обработка исключений: сценарии тестирования ошибок тщательно отличаются, так как поведение исключений отличается от шаблонов.

Более подробное руководство по проблемам и рекомендациям синхронных оболочек для асинхронных методов см. в статье "Следует ли предоставлять синхронные оболочки для асинхронных методов?" .

Просмотрите полный пример

Следующий код представляет полный пример, который доступен в файле примера *Program.cs*.

C#

```
using System.Text.RegularExpressions;
using System.Windows;
using Microsoft.AspNetCore.Mvc;

class Button
{
    public Func<object, object, Task>? Clicked
    {
        get;
        internal set;
    }
}

class DamageResult
{
    public int Damage
    {
        get { return 0; }
    }
}

class User
{
    public bool isEnabled
    {
        get;
        set;
    }

    public int id
    {
        get;
        set;
    }
}

public class Program
```

```
{
    private static readonly Button s_downloadButton = new();
    private static readonly Button s_calculateButton = new();

    private static readonly HttpClient s_httpClient = new();

    private static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-
app-visual-studio",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/shows/net-core-101/what-is-net",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://dotnetfoundation.org",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/maui"
    };

    private static void Calculate()
    {
        static DamageResult CalculateDamageDone()
        {
            return new DamageResult()
            {
                // Code omitted:
                //
                // Does an expensive calculation and returns
                // the result of that calculation.
            };
        }

        s_calculateButton.Clicked += async (o, e) =>
        {
            // This line will yield control to the UI while CalculateDamageDone()
            // performs its work. The UI thread is free to perform other work.
            var damageResult = await Task.Run(() => CalculateDamageDone());
            DisplayDamage(damageResult);
        };
    }

    private static void DisplayDamage(DamageResult damage)
    {
```

```
        Console.WriteLine(damage.Damage);
    }

    private static void Download(string URL)
    {
        s_downloadButton.Clicked += async (o, e) =>
        {
            // This line will yield control to the UI as the request
            // from the web service is happening.
            //
            // The UI thread is now free to perform other work.
            var stringData = await s_httpClient.GetStringAsync(URL);
            DoSomethingWithData(stringData);
        };
    }

    private static void DoSomethingWithData(object stringData)
    {
        Console.WriteLine($"Displaying data: {stringData}");
    }

    private static async Task<User> GetUserAsync(int userId)
    {
        // Code omitted:
        //
        // Given a user Id {userId}, retrieves a User object corresponding
        // to the entry in the database with {userId} as its Id.

        return await Task.FromResult(new User() { id = userId });
    }

    private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
    {
        var getUserTasks = new List<Task<User>>();
        foreach (int userId in userIds)
        {
            getUserTasks.Add(GetUserAsync(userId));
        }

        return await Task.WhenAll(getUserTasks);
    }

    private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int>
userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
        return await Task.WhenAll(getUserTasks);
    }

    private static async Task ProcessTasksAsTheyCompleteAsync(IEnumerable<int>
userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToList();
```

```
        while (getUserTasks.Count > 0)
        {
            Task<User> completedTask = await Task.WhenAny(getUserTasks);
            getUserTasks.Remove(completedTask);

            User user = await completedTask;
            Console.WriteLine($"Processed user {user.id}");
        }
    }

    [HttpGet, Route("DotNetCount")]
    static public async Task<int> GetDotNetCountAsync(string URL)
    {
        // Suspends GetDotNetCountAsync() to allow the caller (the web server)
        // to accept another request, rather than blocking on this one.
        var html = await s_httpClient.GetStringAsync(URL);
        return Regex.Matches(html, @"\.NET").Count;
    }

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        Console.WriteLine("Counting '.NET' phrase in websites...");
        int total = 0;
        foreach (string url in s_urlList)
        {
            var result = await GetDotNetCountAsync(url);
            Console.WriteLine($"{url}: {result}");
            total += result;
        }
        Console.WriteLine("Total: " + total);

        Console.WriteLine("Retrieving User objects with list of IDs...");
        IEnumerable<int> ids = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
        var users = await GetUsersAsync(ids);
        foreach (User? user in users)
        {
            Console.WriteLine($"{user.id}: isEnabled={user.isEnabled}");
        }

        Console.WriteLine("Processing tasks as they complete...");
        await ProcessTasksAsTheyCompleteAsync(ids);

        Console.WriteLine("Application ending.");
    }
}

// Example output:
//
// Application started.
// Counting '.NET' phrase in websites...
// https://learn.microsoft.com: 0
// https://learn.microsoft.com/aspnet/core: 57
// https://learn.microsoft.com/azure: 1
```

```
// https://learn.microsoft.com/azure/devops: 2
// https://learn.microsoft.com/dotnet: 83
// https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-
studio: 31
// https://learn.microsoft.com/education: 0
// https://learn.microsoft.com/shows/net-core-101/what-is-net: 42
// https://learn.microsoft.com/enterprise-mobility-security: 0
// https://learn.microsoft.com/gaming: 0
// https://learn.microsoft.com/graph: 0
// https://learn.microsoft.com/microsoft-365: 0
// https://learn.microsoft.com/office: 0
// https://learn.microsoft.com/powershell: 0
// https://learn.microsoft.com/sql: 0
// https://learn.microsoft.com/surface: 0
// https://dotnetfoundation.org: 16
// https://learn.microsoft.com/visualstudio: 0
// https://learn.microsoft.com/windows: 0
// https://learn.microsoft.com/maui: 6
// Total: 238
// Retrieving User objects with list of IDs...
// 1: isEnabled= False
// 2: isEnabled= False
// 3: isEnabled= False
// 4: isEnabled= False
// 5: isEnabled= False
// 6: isEnabled= False
// 7: isEnabled= False
// 8: isEnabled= False
// 9: isEnabled= False
// 0: isEnabled= False
// Application ending.
```

Связанные ссылки

- асинхронная модель программирования задач (C#)

Асинхронная модель программирования задач

13.10.2025

Вы можете избежать узких мест производительности и повысить общую скорость реагирования приложения с помощью асинхронного программирования. Однако традиционные методы написания асинхронных приложений могут быть сложными, что затрудняет их запись, отладку и обслуживание.

C# поддерживает упрощенный подход, асинхронное программирование, которое использует асинхронную поддержку в среде выполнения .NET. Компилятор выполняет сложную работу, которую разработчик использовал для выполнения, и приложение сохраняет логическую структуру, которая напоминает синхронный код. В результате вы получаете все преимущества асинхронного программирования с частью усилий.

Эта статья содержит общие сведения о том, когда и как использовать асинхронное программирование и содержит ссылки на другие статьи, содержащие подробные сведения и примеры.

Асинхрон улучшает скорость реагирования

Асинхронность важна для действий, которые могут блокировать, например, доступ к веб-ресурсам. Доступ к веб-ресурсу иногда замедляется или задерживается. Если такое действие блокируется в синхронном процессе, все приложение должно ждать. В асинхронном процессе приложение может продолжать работу, которая не зависит от веб-ресурса, пока потенциально блокирующая задача не завершится.

В следующей таблице показаны типичные области, в которых асинхронное программирование повышает скорость реагирования. Перечисленные API из .NET и среды выполнения Windows содержат методы, поддерживающие асинхронное программирование.

Развернуть таблицу

Область приложения	Типы .NET с асинхронными методами	Типы среды выполнения Windows с асинхронными методами
Веб-доступ	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Работа с файлами	JsonSerializer StreamReader	StorageFile

Область приложения	Типы .NET с асинхронными методами	Типы среды выполнения Windows с асинхронными методами
	StreamWriter XmlReader XmlWriter	
Работа с изображениями		MediaCapture BitmapEncoder BitmapDecoder
Программирование WCF	Синхронные и асинхронные операции	

Асинхронность является особенно ценной для приложений, которые обращаются к потоку пользовательского интерфейса, так как все UI-активности обычно используют один поток. Если любой процесс заблокирован в синхронном приложении, все блокируются. Приложение перестает отвечать, и вы можете предположить, что оно не работает, хотя на самом деле оно просто ждет.

При использовании асинхронных методов приложение продолжает реагировать на пользовательский интерфейс. Можно изменить размер или свернуть окно, например, или закрыть приложение, если вы не хотите ждать завершения.

Асинхронный подход добавляет эквивалент автоматической передачи в список параметров, которые можно выбрать при разработке асинхронных операций. То есть вы получаете все преимущества традиционного асинхронного программирования, но с гораздо меньшими усилиями от разработчика.

Асинхронные методы легко записывать

Ключевые слова async и await в C# являются сердцем асинхронного программирования. С помощью этих двух ключевых слов можно использовать ресурсы в .NET Framework, .NET Core или среде выполнения Windows для создания асинхронного метода почти так же легко, как и при создании синхронного метода. Асинхронные методы, определяемые с помощью ключевого `async` слова, называются *асинхронными методами*.

В следующем примере показан асинхронный метод. Почти все в коде должно выглядеть знакомо.

Вы можете найти полный пример Windows Presentation Foundation (WPF), доступный для скачивания из Асинхронное программирование с использованием async и await в C#.

```
C#
```

```
public async Task<int> GetUrlContentLengthAsync()
{
    using var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Вы можете научиться нескольким практикам из предыдущего примера. Начните с сигнатуры метода. Он включает модификатор `async` . Тип возвращаемого значения — `Task<int>` (см. раздел "Возвращаемые типы" для получения дополнительных параметров). Имя метода заканчивается `Async`. В тексте метода `GetStringAsync` возвращается `Task<string>`. Это означает, что когда вы `await`, вы получаете `string` (`contents`). Перед ожиданием задачи можно выполнить работу, которая не зависит от `string` `GetStringAsync`.

Обратите внимание на `await` оператора. Приостанавливает `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` не может продолжаться, пока `getStringTask` не завершится.
- Между тем контроль возвращается вызывающей стороне `GetUrlContentLengthAsync`.
- Управление возобновляется здесь после завершения `getStringTask`.
- Затем `await` оператор извлекает `string` результат из `getStringTask`.

Оператор return устанавливает целочисленный результат. Все методы, ожидающие `GetUrlContentLengthAsync` получения значения длины.

Если `GetUrlContentLengthAsync` у него нет никаких действий, которые он может сделать между вызовом `GetStringAsync` и ожиданием его завершения, можно упростить код, вызвав и ожидая в следующей одной инструкции.

C#

```
string contents = await
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

Следующие характеристики обобщают то, что делает предыдущий пример асинхронным методом:

- Сигнатура `async` метода включает модификатор.
- Имя асинхронного метода по соглашению заканчивается суффиксом Async.
- Возвращаемый тип является одним из следующих типов:
  - Task<TResult> Если метод имеет оператор return, в котором операнд имеет тип `TResult`.
  - Task Если метод не имеет инструкции return или имеет оператор return без операнда.
  - `void` Если вы пишете асинхронный обработчик событий.
  - Любой другой тип, имеющий метод `GetAwaiter`.

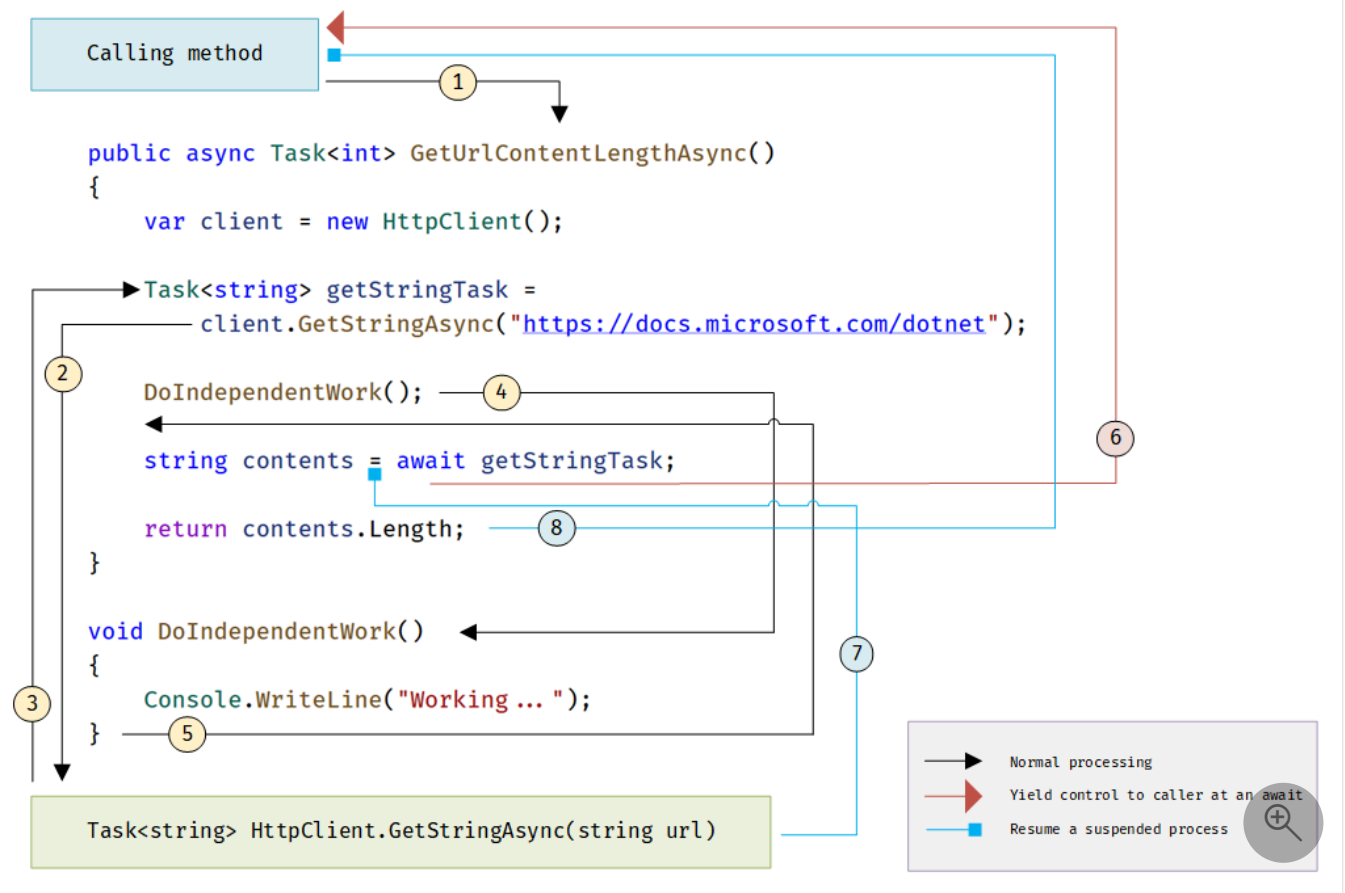
  Дополнительные сведения см. в разделе "Типы возврата" и "Параметры ".
- Обычно этот метод включает по крайней мере одно `await` выражение, которое помечает точку, в которой метод не может продолжаться до завершения ожидаемой асинхронной операции. В то же время метод приостановлен, а управление возвращается вызвавшему его. В следующем разделе этой статьи показано, что происходит в точке приостановки.

В асинхронных методах используйте указанные ключевые слова и типы, чтобы указать, что нужно сделать, и компилятор выполняет остальные действия, включая отслеживание того, что должно произойти при возврате элемента управления в точку ожидания в приостановленном методе. Некоторые обычные процессы, такие как циклы и обработка исключений, могут быть трудно обрабатывать в традиционном асинхронном коде. В асинхронном методе эти элементы записываются так же, как и в синхронном решении, и проблема решена.

Дополнительные сведения об асинхронности в предыдущих версиях .NET Framework см. в статье TPL и традиционное асинхронное программирование .NET Framework.

Что происходит в асинхронном методе

Самое главное, чтобы понять в асинхронном программировании, заключается в том, как поток управления перемещается из метода в метод. Диаграмма ниже проведет вас через процесс:



Числа на схеме соответствуют следующим шагам, которые начинаются, когда вызывающий метод вызывает асинхронный метод.

1. Вызывающий метод вызывает и ожидает асинхронный `GetUrlContentLengthAsync` метод.

2. `GetUrlContentLengthAsync` создаёт экземпляр HttpClient и вызывает асинхронный метод GetStringAsync для загрузки содержимого веб-сайта в виде строки.

3. Что-то происходит в `GetStringAsync`, что приостанавливает его продвижение. Возможно, он должен ждать загрузки веб-сайта или другого блокирующего действия. Чтобы избежать блокировки ресурсов, `GetStringAsync` возвращает управление вызывающим объекту `GetUrlContentLengthAsync`.

   `GetStringAsync` Task<TResult>возвращает значение , где `TResult` является строка и `GetUrlContentLengthAsync` назначает задачу переменной`getStringTask`. Задача представляет текущий процесс вызова `GetStringAsync` с обязательством создать фактическое строковое значение, когда работа будет завершена.

4. Поскольку `getStringTask` еще не завершен, `GetUrlContentLengthAsync` может выполнять другие задачи, которые не зависят от конечного результата из `GetStringAsync`. Эта работа представлена вызовом синхронного метода `DoIndependentWork`.

5. `DoIndependentWork` — синхронный метод, который выполняет свою работу и возвращает управление вызывающему.

6. `GetUrlContentLengthAsync` заканчивает всю работу, которую может выполнять без результата от `getStringTask`. `GetUrlContentLengthAsync` далее требуется вычислить и вернуть длину скачаемой строки, но метод не может вычислить это значение, пока метод не будет содержать строку.

   Поэтому `GetUrlContentLengthAsync` использует оператор await для приостановки выполнения и передачи управления методу, который вызвал `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` возвращает `Task<int>` вызывающему. Задача представляет обещание создать целочисленный результат, который является длиной скачаемой строки.

   > ⓘ **Примечание**
   >
   > Если `GetStringAsync` (и поэтому `getStringTask`) завершается до того, как `GetUrlContentLengthAsync` его ожидает, управление остается в `GetUrlContentLengthAsync`. Расходы на приостановку и последующее возвращение к `GetUrlContentLengthAsync` окажутся напрасными, если вызываемый асинхронный процесс `getStringTask` завершен и `GetUrlContentLengthAsync` не должен ждать окончательного результата.

   Обработка продолжается внутри вызывающего метода. Вызывающий может выполнить другие действия, не зависящие от результата `GetUrlContentLengthAsync`, прежде чем ожидать этот результат, или может ожидать немедленно. Вызывающий метод ожидает `GetUrlContentLengthAsync`, а `GetUrlContentLengthAsync` ожидает `GetStringAsync`.

7. `GetStringAsync` завершает и создает строковый результат. Строковый результат не возвращается вызовом `GetStringAsync` так, как вы могли бы ожидать. (Помните, что метод уже вернул задачу на шаге 3.) Вместо этого строковый результат хранится в задаче, представляющей завершение метода `getStringTask`. Оператор await извлекает результат из `getStringTask`. Инструкция назначения назначает полученный результат `contents`.

8. При `GetUrlContentLengthAsync` получении результата строки метод может вычислить длину строки. Затем работа `GetUrlContentLengthAsync` также завершена, а обработчик события ожидания может возобновить свою работу. В полном примере, представленном в конце статьи, можно убедиться, что обработчик событий

извлекает и распечатывает значение результата длины. Если вы не знакомы с асинхронным программированием, обратите внимание на разницу между синхронным и асинхронным поведением. Синхронный метод возвращается после завершения работы (шаг 5), но асинхронный метод возвращает значение задачи при приостановке работы (шаги 3 и 6). Когда асинхронный метод в конечном итоге завершит свою работу, задача помечается как завершенная, а результат, если он есть, хранится в задаче.

Асинхронные методы API

Вы, возможно, задумываетесь, где можно найти такие методы, как `GetStringAsync`, которые поддерживают асинхронное программирование. .NET Framework 4.5 или более поздней версии и .NET Core содержат множество элементов, которые работают с `async` и `await`. Их можно распознать по суффиксу Async, добавленному к имени члена, и по их типу возвращаемого значения, как Task или Task<TResult>. Например, `System.IO.Stream` класс содержит такие методы, как CopyToAsync, ReadAsyncи WriteAsync наряду с синхронными методами CopyTo, Readи Write.

Среда выполнения Windows также содержит множество методов, которые можно использовать с `async` и `await` в приложениях Windows. Дополнительные сведения см. в статьях "Потоки и асинхронное программирование" для разработки UWP, "Асинхронное программирование (приложения Магазина Windows)" и "Краткое руководство: Вызов асинхронных API в C# или Visual Basic" при использовании более ранних версий среды выполнения Windows.

Потоки

Асинхронные методы предназначены для неблокирующих операций. Выражение `await` в асинхронном методе не блокирует текущий поток во время выполнения ожидаемой задачи. Вместо этого выражение регистрирует остальную часть метода как продолжение и возвращает управление вызывающему асинхронный метод.

Ключевые слова `async` и `await` не приводят к созданию дополнительных потоков. Для асинхронных методов не требуется многопоточность, так как асинхронный метод не выполняется в собственном потоке. Метод выполняется в текущем контексте синхронизации и использует время в потоке, только если метод активен. Вы можете использовать Task.Run для перемещения работы, привязанной к ЦП, в фоновый поток, но фоновый поток не помогает в процессе, который просто ожидает получения результатов.

Асинхронный подход к асинхронным программированию предпочтительнее существующих подходов практически в каждом случае. В частности, этот подход лучше, чем BackgroundWorker класс для операций ввода-вывода, так как код проще, и вам не нужно защищаться от условий гонки. В сочетании с методом Task.Run, асинхронное программирование лучше, чем BackgroundWorker, для операций, связанных с загрузкой ЦП, поскольку оно отделяет координацию выполнения кода от задач, которые `Task.Run` передает в пул потоков.

Асинхронная и ожидание

Если указать, что метод является асинхронным методом с помощью асинхронного модификатора, включите следующие две возможности.

- Помеченный асинхронный метод может использовать ожидание для назначения точек приостановки. Оператор `await` сообщает компилятору, что асинхронный метод не может продолжаться до завершения ожидаемого асинхронного процесса. В это время управление возвращается вызывающему асинхронный метод.

  Приостановка асинхронного метода в `await` выражении не представляет собой выход из метода, и `finally` блоки не выполняются.

- Помеченный асинхронный метод может ожидаться методами, вызывающими его.

Асинхронный метод обычно содержит одно или несколько вхождений `await` оператора, но отсутствие `await` выражений не приводит к ошибке компилятора. Если асинхронный метод не использует `await` оператор для маркировки точки приостановки, метод выполняется как синхронный метод, несмотря на `async` модификатор. Компилятор выдает предупреждение для таких методов.

`async` и `await` являются контекстными ключевыми словами. Дополнительные сведения и примеры см. в следующих статьях:

- асинхронные
- ожидайте

Возвращаемые типы и параметры

Асинхронный метод обычно возвращает Task или Task<TResult>. В асинхронном методе оператор `await` применяется к задаче, возвращаемой из вызова другого асинхронного метода.

Тип Task<TResult> указывается как возвращаемое значение, если метод содержит оператор return, указывающий операнд типа `TResult`.

Вы используете Task в качестве типа возвращаемого значения, если метод не имеет инструкции return или имеет оператор return, который не возвращает операнду.

Можно также указать любой другой возвращаемый тип, если тип содержит метод `GetAwaiter`. ValueTask<TResult> пример такого типа. Он доступен в пакете NuGet System.Threading.Tasks.Extension .

В следующем примере показано, как объявлять и вызывать метод, который возвращает Task<TResult> или Task.

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Каждая возвращаемая задача представляет текущую работу. Задача инкапсулирует сведения о состоянии асинхронного процесса и в конечном счёте либо конечный результат этого процесса, либо исключение, которое вызывает процесс, если он терпит неудачу.

Асинхронный метод также может иметь тип возвращаемого `void` значения. Этот тип возвращаемого значения используется в основном для определения обработчиков

событий, где требуется возвращаемый тип `void`. Асинхронные обработчики событий часто служат отправной точкой для асинхронных программ.

Асинхронный метод с возвращаемым типом `void` не может быть ожидаем, и вызывающий метод с возвращаемым типом void не может перехватывать исключения, которые он может выбрасывать.

Асинхронный метод не может объявлять параметры с модификаторами in, ref или out, но может вызывать методы, в которых такие параметры используются. Аналогичным образом асинхронный метод не может возвращать значение по ссылке, хотя он может вызывать методы с возвращаемыми значениями ссылок.

Дополнительные сведения и примеры см. в статье async return types (C#).

Асинхронные API в программировании среды выполнения Windows имеют один из следующих типов возвращаемых значений, аналогичных задачам:

- IAsyncOperation<TResult>, соответствующий Task<TResult>
- IAsyncAction, соответствующий Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

Соглашение об именовании

По соглашению методы, возвращающие часто ожидаемые типы (например, `Task`, `Task<T>`, , `ValueTask`, `ValueTask<T>`) должны иметь имена, заканчивающиеся "Async". Методы, которые запускают асинхронную операцию, но не возвращают ожидаемый тип, не должны иметь имена, заканчивающиеся "Async", но могут начинаться с "Begin", "Start" или другой командой, чтобы предложить этот метод не возвращать или вызывать результат операции.

Вы можете игнорировать соглашение, в котором событие, базовый класс или контракт интерфейса предлагает другое имя. Например, не следует переименовать распространенные обработчики событий, например `OnButtonClick`.

Связанные статьи (Visual Studio)

Развернуть таблицу

Заголовок	Описание
Как выполнять несколько веб-запросов параллельно с использованием async и await (C#)	Демонстрирует, как одновременно запускать несколько задач.
Асинхронные возвращаемые типы (C#)	Иллюстрирует типы, которые асинхронные методы могут возвращать, и объясняет, когда каждый тип подходит.
Отменяйте задачи, используя токен отмены в качестве механизма сигнализации.	Показывает, как добавить следующие функции в асинхронное решение: - Отмена списка задач (C#) - Отмена задач после периода времени (C#) - Обработка асинхронных задач по мере их завершения (C#)
Использование асинхронного доступа к файлам (C#)	Перечисляет и демонстрирует преимущества использования async и await для доступа к файлам.
Асинхронный шаблон на основе задач (TAP)	Описывает асинхронный шаблон. Шаблон основан на типах Task и Task<TResult>.
Асинхронные видео на канале 9	Содержит ссылки на различные видеоролики об асинхронном программировании.

См. также

- Асинхронное программирование с async и await
- асинхронные
- ожидайте

Асинхронные возвращаемые типы (C#)

Статья • 20.12.2024

Асинхронные методы могут иметь следующие типы возвращаемых данных:

- Taskдля асинхронного метода, выполняющего операцию, но не возвращает значения.
- Task<TResult>для асинхронного метода, возвращающего значение.
- `void`для обработчика событий.
- Любой тип, имеющий доступный метод `GetAwaiter`. Объект, возвращаемый методом `GetAwaiter`, должен реализовать интерфейс System.Runtime.CompilerServices.ICriticalNotifyCompletion.
- IAsyncEnumerable<T>для асинхронного метода, возвращающего *асинхронный поток*.

Дополнительные сведения об асинхронных методах см. в асинхронном программировании с использованием async и await.

Существуют и другие типы, относящиеся к рабочим нагрузкам Windows:

- DispatcherOperationдля асинхронных операций, ограниченных Windows.
- IAsyncActionдля асинхронных действий в приложениях универсальной платформы Windows (UWP), которые не возвращают значение.
- IAsyncActionWithProgress<TProgress>для асинхронных действий в приложениях UWP, которые сообщают о ходе выполнения, но не возвращают значение.
- IAsyncOperation<TResult>для асинхронных операций в приложениях UWP, возвращающих значение.
- IAsyncOperationWithProgress<TResult,TProgress>для асинхронных операций в приложениях UWP, которые сообщают о ходе выполнения и возвращают значение.

Тип возвращаемого значения задачи

Асинхронные методы, не содержащие инструкцию `return` или содержащие инструкцию `return`, которая не возвращает операнду, обычно имеет тип возвращаемого значения Task. Такие методы возвращают `void`, если они выполняются синхронно. Если для асинхронного метода используется тип возврата

Task, вызывающий метод может использовать оператор `await` для приостановки завершения вызывающего метода до завершения вызываемого асинхронного метода.

В следующем примере метод `WaitAndApologizeAsync` не содержит инструкцию `return`, поэтому метод возвращает объект Task. Возврат `Task` позволяет ожидать `WaitAndApologizeAsync`. Тип Task не включает свойство `Result`, так как оно не имеет возвращаемого значения.

C#

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

`WaitAndApologizeAsync` ожидается с использованием инструкции await, а не выражения await, аналогично вызову синхронного метода, возвращающего void. Приложение оператора await в данном случае не создает значение. Если правый операнд `await` является Task<TResult>, выражение `await` создает результат `T`. Когда правый операнд `await` является Task, `await` и его операнд являются выражением.

Вызов `WaitAndApologizeAsync` можно отделить от использования оператора await, например, в следующем коде. Однако помните, что `Task` не имеет свойства `Result`, и что значение не создается при применении оператора await к `Task`.

Следующий код отделяет вызов метода `WaitAndApologizeAsync` от ожидания задачи, возвращаемой методом.

C#

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

Тип возврата<задачи TResult>

Тип Task<TResult> используется для асинхронного метода, который содержит оператор , возвращающий, в котором операнд — `TResult`.

В следующем примере метод `GetLeisureHoursAsync` содержит оператор `return`, возвращающий целое число. Объявление метода должно указывать тип возвращаемого значения `Task<int>`. Асинхронный метод FromResult — это заполнитель для операции, возвращающей DayOfWeek.

C#

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

При вызове `GetLeisureHoursAsync` из выражения await в методе `ShowTodaysInfo` выражение await извлекает целочисленное значение (значение `leisureHours`), хранящееся в задаче, возвращаемой методом `GetLeisureHours`. Дополнительные сведения о выражениях await см. в разделе await.

Вы можете лучше понять, как `await` получает результат из `Task<T>`, отделив вызов к `GetLeisureHoursAsync` от применения `await`, как это показано в следующем коде. Вызов метода `GetLeisureHoursAsync`, который не ожидается сразу же, возвращает `Task<int>`, как ожидается из объявления метода. Задача назначается переменной `getLeisureHoursTask` в примере. Так как `getLeisureHoursTask` является Task<TResult>, он содержит свойство Result типа `TResult`. В этом случае `TResult` представляет целочисленный тип. Когда `await` применяется к `getLeisureHoursTask`, выражение await вычисляется в значение свойства Result объекта `getLeisureHoursTask`. Значение назначается переменной `ret`.

> ⓘ **Важно!**
>
> Свойство **Result** является блокирующим свойством. Если вы попытаетесь получить доступ к нему до завершения задачи, поток, который сейчас активен, будет заблокирован до окончания задачи и пока значение не станет доступным. В большинстве случаев необходимо получить доступ к значению с помощью `await` вместо прямого доступа к свойству.
>
> В предыдущем примере было получено значение свойства **Result** для блокировки основного потока, чтобы метод `Main` мог распечатать `message` в консоль до завершения приложения.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Тип возвращаемого значения Void

Тип возврата `void` используется в асинхронных обработчиках событий, для которых требуется тип возврата `void`. Для методов, отличных от обработчиков событий, которые не возвращают значение, следует вернуть Task вместо этого, так как асинхронный метод, возвращающий `void`, не может быть ожидаем. Любой вызывающий метод должен продолжать выполняться, не ожидая завершения вызываемого асинхронного метода. Вызывающий объект должен быть независимым от любых значений или исключений, создаваемых асинхронным методом.

Вызывающий метод async, возвращающий void, не может перехватывать исключения, вызванные методом. Такие необработанные исключения могут привести к сбою приложения. Если метод, возвращающий Task или Task<TResult>, создает исключение, исключение сохраняется в возвращаемой задаче. Исключение бросается снова, когда ожидается завершение задачи. Убедитесь, что любой из асинхронных методов, который может вызвать исключение, имеет тип возвращаемого значения Task или Task<TResult> и что вызовы этого метода ожидаются.

В следующем примере показано поведение асинхронного обработчика событий. В примере кода асинхронный обработчик событий должен сообщить основному потоку о завершении. Затем основной поток может ожидать завершения асинхронного обработчика событий перед выходом из программы.

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;
```

```
        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

Обобщенные асинхронные типы возвращаемых данных и ValueTask<TResult>

Асинхронный метод может возвращать любой тип, который имеет доступный метод `GetAwaiter` и возвращает экземпляр типа ожидателя . Кроме того, тип, возвращаемый методом `GetAwaiter`, должен иметь атрибут System.Runtime.CompilerServices.AsyncMethodBuilderAttribute. Дополнительные сведения см. в статье о атрибутах , прочитанных компилятором или спецификацией C# для шаблона построителя типов задач .

Эта функция является дополнением к ожидаемым выражениям, которые описывают требования для операнда `await`. Обобщенные асинхронные возвращаемые типы позволяют компилятору создавать методы `async`, возвращающие разные типы. Общие асинхронные типы возвращаемых значений позволили улучшить производительность в библиотеках .NET. Так как Task и Task<TResult> являются ссылочными типами, выделение памяти на производительно-значимых участках, особенно если выделение происходит в плотных циклах, может отрицательно повлиять на производительность. Поддержка обобщенных типов возвращаемых значений означает, что вместо ссылочного типа можно возвращать упрощенный тип значения, чтобы избежать выделения памяти.

.NET предоставляет структуру System.Threading.Tasks.ValueTask<TResult> как облегчённую реализацию обобщенного возвращаемого значения задач. В следующем примере используется структура ValueTask<TResult> для получения значений двух бросков кубиков.

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
```

```
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
// Example output:
//    Shaking dice...
//    You rolled 8
```

Написание обобщённого асинхронного возвращаемого типа является сложным сценарием и предназначено для использования в специализированных средах. Вместо этого рекомендуется использовать `Task`, `Task<T>` и типы `ValueTask<T>`, которые охватывают большинство сценариев асинхронного кода.

Атрибут `AsyncMethodBuilder` можно применить к асинхронным методу (вместо объявления асинхронного возвращаемого типа), чтобы переопределить построитель для этого типа. Обычно этот атрибут применяется для использования другого построителя, предоставленного в среде выполнения .NET.

Асинхронные потоки с IAsyncEnumerable<T>

Асинхронный метод может возвращать *асинхронный поток*, представленный IAsyncEnumerable<T>. Асинхронный поток предоставляет способ перечисления элементов, считываемых из потока, когда элементы создаются частями с помощью повторяющихся асинхронных вызовов. В следующем примере показан асинхронный метод, который создает асинхронный поток:

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
              Here is the second line of text.
              And there is one more for good measure.
              Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
```

```
        }

        line = await readStream.ReadLineAsync();
    }
}
```

В предыдущем примере строки считываются из текста асинхронно. После чтения каждой строки код перечисляет каждое слово в строке. Вызывающие будут перечислять каждое слово с помощью инструкции `await foreach`. Метод ожидает, когда необходимо асинхронно считывать следующую строку из исходной строки.

См. также

- FromResult
- обрабатывать асинхронные задачи по мере их завершения
- Асинхронное программирование с использованием async и await (C#)
- асинхронные
- ожидайте

Обработка асинхронных задач по мере их выполнения (C#)

С помощью Task.WhenAnyможно одновременно запускать несколько задач и обрабатывать их по одному, а не обрабатывать их в том порядке, в котором они запущены.

В следующем примере используется запрос для создания коллекции задач. Каждая задача скачивает содержимое указанного веб-сайта. В каждой итерации цикла во время ожидания вызов WhenAny возвращает задачу в коллекции задач, завершающих загрузку первым. Эта задача удаляется из коллекции и обрабатывается. Цикл повторяется, пока коллекция не содержит больше задач.

Предпосылки

Вы можете следовать этому руководству, используя один из следующих вариантов:

- Visual Studio 2022 с установленным компонентом **разработки десктопных приложений .NET**. Пакет SDK для .NET устанавливается автоматически при выборе этой рабочей нагрузки.
- Пакет SDK для .NET с выбранным редактором кода, например Visual Studio Code.

Создание примера приложения

Создайте консольное приложение .NET Core. Ее можно создать с помощью команды dotnet new console или из Visual Studio.

Откройте *файл Program.cs* в редакторе кода и замените существующий код следующим кодом:

C#

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Добавление полей

В определении `Program` класса добавьте следующие два поля:

C#

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/maui"
};
```

`HttpClient` открывает возможность отправлять HTTP-запросы и получать HTTP-ответы. Элемент `s_urlList` содержит все URL-адреса, которые приложение планирует обрабатывать.

Обновление точки входа приложения

Основная точка входа в консольное приложение — метод `Main`. Замените существующий метод следующим:

C#

```
static Task Main() => SumPageSizesAsync();
```

Обновленный `Main` метод теперь считается async main, что позволяет использовать асинхронную точку входа в исполняемый файл. Это означает вызов к `SumPageSizesAsync`.

Создайте асинхронный метод суммирования размеров страниц

Ниже метода `Main` добавьте метод `SumPageSizesAsync`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

Цикл `while` удаляет одну из задач в каждой итерации. После завершения каждой задачи цикл завершается. Метод начинается с создания экземпляра и запуска Stopwatch. Затем он включает запрос, который при выполнении создает коллекцию задач. Каждый вызов `ProcessUrlAsync` в следующем коде возвращает Task<TResult>, где `TResult` — целое число.

C#

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

Из-за отложенного выполнения в LINQ, вызывайте Enumerable.ToList, чтобы запустить каждую задачу.

```C#
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

Цикл `while` выполняет следующие действия для каждой задачи в коллекции:

1. Ожидает вызова `WhenAny` для определения первой задачи в коллекции, которая завершила загрузку.

   ```C#
   Task<int> finishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Удаляет эту задачу из коллекции.

   ```C#
   downloadTasks.Remove(finishedTask);
   ```

3. Ожидает `finishedTask`, который возвращается вызовом `ProcessUrlAsync`. Переменная `finishedTask` — это Task<TResult>, где `TResult` — целое число. Задача уже завершена, но вы должны подождать, чтобы получить длину загруженного веб-сайта, как показано в следующем примере. Если задача неисправна, `await` вызовет первое дочернее исключение, хранящееся в `AggregateException`свойстве, в отличие от чтения Task<TResult>.Result свойства, которое вызовет исключение `AggregateException`.

   ```C#
   total += await finishedTask;
   ```

Добавление метода процесса

Добавьте следующий `ProcessUrlAsync` метод под методом `SumPageSizesAsync` :

```C#
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
```

```
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Для любого данного URL-адреса метод будет использовать предоставленный `client` экземпляр, чтобы получить ответ в виде `byte[]`. Длина возвращается после записи URL-адреса и длины в консоль.

Запустите программу несколько раз, чтобы убедиться, что скачанные длины не всегда отображаются в одном порядке.

> ⊗ **Внимание**
>
> Вы можете использовать в цикле `WhenAny` как показано в примере, для решения проблем, связанных с небольшим количеством задач. Однако другие подходы более эффективны при наличии большого количества задач для обработки. Дополнительные сведения и примеры см. в разделе "Обработка задач по мере их выполнения".

Упростите подход с помощью `Task.WhenEach`

Цикл `while`, реализованный в `SumPageSizesAsync` методе, можно упростить, используя новый метод Task.WhenEach, представленный в .NET 9, вызвав его в цикле `await foreach`. Замените ранее реализованный `while` цикл:

C#

```
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }
```

с упрощенным `await foreach`:

C#

```
    await foreach (Task<int> t in Task.WhenEach(downloadTasks))
    {
        total += await t;
    }
```

Этот новый подход позволяет не вызывать `Task.WhenAny` задачу вручную многократно и удалять ее по завершении, так как `Task.WhenEach` выполняет итерацию по задачам в порядке их выполнения.

Полный пример

Следующий код представляет собой полный текст файла *Program.cs* для примера.

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/maui"
};

await SumPageSizesAsync();

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

```
    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:    {total:#,#}");
    Console.WriteLine($"Elapsed time:              {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                   132,517
// https://learn.microsoft.com/powershell                         57,375
// https://learn.microsoft.com/gaming                             33,549
// https://learn.microsoft.com/aspnet/core                        88,714
// https://learn.microsoft.com/surface                            39,840
// https://learn.microsoft.com/enterprise-mobility-security       30,903
// https://learn.microsoft.com/microsoft-365                      67,867
// https://learn.microsoft.com/windows                            26,816
// https://learn.microsoft.com/maui                               57,958
// https://learn.microsoft.com/dotnet                             78,706
// https://learn.microsoft.com/graph                              48,277
// https://learn.microsoft.com/dynamics365                        49,042
// https://learn.microsoft.com/office                             67,867
// https://learn.microsoft.com/system-center                      42,887
// https://learn.microsoft.com/education                          38,636
// https://learn.microsoft.com/azure                             421,663
// https://learn.microsoft.com/visualstudio                       30,925
// https://learn.microsoft.com/sql                                54,608
// https://learn.microsoft.com/azure/devops                       86,034

// Total bytes returned:    1,454,184
// Elapsed time:            00:00:01.1290403
```

См. также

- WhenAny
- WhenEach
- Асинхронное программирование с async и await (C#)

Last updated on 20.03.2026

Асинхронный доступ к файлам (C#)

Используя асинхронную функцию для доступа к файлам, можно вызывать асинхронные методы без использования обратных вызовов или разделения кода по нескольким методам или лямбда-выражениям. Чтобы сделать синхронный код асинхронным, вызовите асинхронный метод вместо синхронного метода и добавьте в код несколько ключевых слов.

Рассмотрите возможность добавления асинхронности в вызовы доступа к файлам по следующим причинам:

- ✓ Asynchrony повышает скорость реагирования приложений пользовательского интерфейса, так как поток пользовательского интерфейса, запускающий операцию, может выполнять другую работу. Если поток пользовательского интерфейса должен выполнять код, который занимает много времени (например, более 50 миллисекунд), пользовательский интерфейс может заморозить до завершения ввода-вывода, а поток пользовательского интерфейса может снова обрабатывать ввод клавиатуры и мыши и другие события.
- ✓ Асинхронность повышает масштабируемость ASP.NET и других серверных приложений, уменьшая потребность в потоках. Если приложение использует выделенный поток для каждого ответа и тысячи запросов обрабатываются одновременно, требуются тысячи потоков. Асинхронные операции часто не требуют использования потока в период ожидания. Они используют существующий поток завершения операций ввода-вывода ненадолго в конце.
- ✓ Задержка операции доступа к файлам может быть очень низкой в текущих условиях, но задержка может значительно увеличиться в будущем. Например, файл может быть перемещен на сервер, который находится во всем мире.
- ✓ Дополнительные затраты на использование функции Async невелики.
- ✓ Несколько асинхронных операций ввода-вывода могут выполняться без блокировки вызывающего потока.

Используйте соответствующие классы

Простые примеры в этом разделе демонстрируют File.WriteAllTextAsync и File.ReadAllTextAsync. Для точного управления операциями ввода-вывода файлов используйте FileStream класс, который имеет параметр, позволяющий выполнять асинхронные операции ввода-вывода на уровне операционной системы. С помощью этого

параметра можно избежать блокировки потока из пула во многих ситуациях. Чтобы включить этот параметр, укажите аргумент `useAsync=true` или `options=FileOptions.Asynchronous` в вызове конструктора.

Этот параметр нельзя использовать вместе с StreamReader и StreamWriter, если открываете их напрямую, указывая путь к файлу. Однако этот параметр можно использовать, если предоставить им Stream, который был открыт классом FileStream. Асинхронные вызовы выполняются быстрее в приложениях пользовательского интерфейса, даже если поток из пула рабочих потоков заблокирован, поскольку поток пользовательского интерфейса не блокируется во время ожидания.

Написание текста

В следующих примерах записывается текст в файл. На каждой инструкции await метод немедленно прекращает выполнение. После завершения ввода-вывода файла метод возобновляется в инструкции, следующей за инструкцией await. Асинхронный модификатор находится в определении методов, использующих инструкцию await.

Простой пример

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Пример конечного элемента управления

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);
```

```
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

В исходном примере имеется выражение `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, которое является сокращением следующих двух операторов:

C#

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

Первая инструкция возвращает задачу и приводит к началу обработки файлов. Вторая инструкция с ожиданием приводит к немедленному выходу метода и возврату другой задачи. После завершения обработки файлов выполнение возвращается в инструкцию, которая следует за ожиданием.

Чтение текста

В следующих примерах считывается текст из файла.

Простой пример

C#

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Пример конечного элемента управления

Текст буферичен и, в данном случае, помещается в объект StringBuilder. В отличие от предыдущего примера, оценка await создает значение. Метод ReadAsync возвращает

значениеTask<Int32>, поэтому оценка await создает `Int32` значение `numRead` после завершения операции. Дополнительные сведения см. в разделе Async Return Types (C#).

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

Несколько асинхронных операций ввода-вывода

В следующих примерах запускается несколько асинхронных операций записи. Среда выполнения ставит в очередь эти операции, и базовая реализация может использовать

асинхронный ввод-вывод операционной системы (ОС) или потоки пула потоков в зависимости от платформы и конфигурации, поэтому фактическая конкуренция зависит от ОС и оборудования.

Простой пример

C#

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

Пример конечного элемента управления

Для каждого файла WriteAsync метод возвращает задачу, добавленную в список задач. Оператор `await Task.WhenAll(tasks);` завершает метод и возобновляет работу в методе при завершении обработки файлов для всех задач.

Пример закрывает все FileStream экземпляры в блоке `finally` после завершения задач. Если `FileStream` была вместо этого создана в инструкции `using`, `FileStream` может быть удален до завершения задачи.

Асинхронный подход позволяет избежать блокировки вызывающего потока во время ожидания ввода-вывода. Во многих случаях улучшения пропускной способности зависят от ОС, оборудования и на некоторых платформах, поведения среды выполнения .NET, таких как ограничения пула потоков и планирование.

C#

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();
```

```
    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

При использовании методов WriteAsync и ReadAsync можно указать CancellationToken, чтобы отменить операцию в середине потока. Дополнительные сведения см. в разделе "Отмена" в управляемых потоках.

См. также

- Асинхронное программирование с использованием async и await (C#)
- Асинхронные возвращаемые типы (C#)

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.04.2026

Отмена списка задач

Статья • 25.03.2025

Вы можете отменить асинхронное консольное приложение, если вы не хотите ждать завершения. Следуя примеру в этом разделе, вы можете добавить отмену в приложение, которое скачивает содержимое списка веб-сайтов. Можно отменить многие задачи, связав экземпляр CancellationTokenSource с каждой задачей. Если выбрать клавишу , вы отмените все задачи, которые еще не завершены.

Темы, рассматриваемые в этом руководстве:

- ✓ Создание консольного приложения .NET
- ✓ Написание асинхронного приложения, которое поддерживает отмену
- ✓ Демонстрация отмены сигнала

Предпосылки

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Создание примера приложения

Создайте консольное приложение .NET Core. Его можно создать с помощью команды dotnet new console или в Visual Studio. Откройте файл *Program.cs* в избранном редакторе кода.

Заменить директивы using

Замените существующие директивы `using` следующими объявлениями:

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Добавьте поля

В определении класса `Program` добавьте следующие три поля:

C#

```
static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/maui"
};
```

CancellationTokenSource используется для сигнала о запрошенной отмене CancellationToken. `HttpClient` предоставляет возможность отправлять HTTP-запросы и получать HTTP-ответы. `s_urlList` содержит все URL-адреса, которые приложение планирует обрабатывать.

Обновление точки входа приложения

Основной точкой входа в консольное приложение является метод `Main`. Замените существующий метод следующим:

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}
```

Обновленный метод `Main` теперь считается асинхронным главным , что позволяет создать асинхронную точку входа в исполняемый файл. Он выводит несколько инструкций в консоль, а затем объявляет экземпляр Task с именем `cancelTask`, считывающий нажатия клавиш консоли. Если нажатие клавиши ввод `ввод` вызывается CancellationTokenSource.Cancel(). Это сигнализирует об отмене. Затем переменная `sumPageSizesTask` назначается из метода `SumPageSizesAsync`. Затем обе задачи передаются в Task.WhenAny(Task[]), который продолжит выполнение, когда любая из двух задач завершится.

Следующий блок кода гарантирует, что приложение не завершает работу до тех пор, пока отмена не будет обработана. Если первой задачей будет выполнение

`cancelTask`, тогда ожидается `sumPageSizeTask`. Если он был отменен, когда ждали, он выдает System.Threading.Tasks.TaskCanceledException. Блок перехватывает это исключение и выводит сообщение.

Создать асинхронный метод суммирования размеров страниц

Под методом `Main` добавьте метод `SumPageSizesAsync`:

```
C#
```

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

Метод начинается с создания и запуска экземпляра Stopwatch. Затем он циклит по каждому URL-адресу в `s_urlList` и вызывает `ProcessUrlAsync`. При каждой итерации `s_cts.Token` передается в метод `ProcessUrlAsync`, а код возвращает Task<TResult>, где `TResult` является целым числом:

```
C#
```

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Добавление метода процесса

Добавьте следующий метод `ProcessUrlAsync` ниже метода `SumPageSizesAsync`:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Для любого URL-адреса метод будет использовать предоставленный экземпляр `client` для получения ответа как `byte[]`. Экземпляр CancellationToken передается в методы HttpClient.GetAsync(String, CancellationToken) и HttpContent.ReadAsByteArrayAsync(). `token` используется для регистрации запрошенной отмены. Длина возвращается после того, как в консоль записаны URL-адрес и длина.

Пример выходных данных приложения

Консоль

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com                                   37,357
https://learn.microsoft.com/aspnet/core                       85,589
https://learn.microsoft.com/azure                            398,939
https://learn.microsoft.com/azure/devops                      73,663
https://learn.microsoft.com/dotnet                            67,452
https://learn.microsoft.com/dynamics365                       48,582
https://learn.microsoft.com/education                         22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Полный пример

Следующий код представляет собой полный текст файла *Program.cs* для примера.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/maui"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }

            Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();
```

```
        Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
        if (finishedTask == cancelTask)
        {
            // wait for the cancellation to take place:
            try
            {
                await sumPageSizesTask;
                Console.WriteLine("Download task completed before cancel
request was processed.");
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Download task has been cancelled.");
            }
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

См. также

- CancellationToken

- CancellationTokenSource
- Асинхронное программирование с использованием async и await (C#)

Дальнейшие действия

Отмена асинхронных задач после периода времени (C#)

Отмена асинхронных задач через период времени

Статья • 09.09.2023

Если не нужно дожидаться, пока завершится выполнение асинхронной операции, ее можно отменить по истечении определенного периода времени с помощью метода CancellationTokenSource.CancelAfter. Этот метод планирует отмену всех связанных задач, не завершенных в течение времени, установленного выражением `CancelAfter`.

В этом примере добавляется код, составленный в разделе Отмена списка задач (C#), для загрузки списка веб-сайтов и отображения длины содержимого каждого из них.

Темы, рассматриваемые в этом руководстве:

- ✓ Обновление существующего консольного приложения .NET
- ✓ Планирование отмены

Необходимые компоненты

Для работы с данным учебником требуется следующее:

- Предполагается, что вы создали приложение в учебнике Отмена списка задач (C#).
- Пакет SDK для .NET 5 или более поздней версии
- Интегрированная среда разработки (IDE)
  - Мы рекомендуем Visual Studio или Visual Studio Code

Обновление точки входа приложения

Замените существующий метод `Main` следующим кодом.

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);
```

```
        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

Обновленный метод `Main` записывает в консоль несколько инструкций. try-catchВ рамках вызова по расписанию CancellationTokenSource.CancelAfter(Int32) отмены. Эта операция будет сообщать об отмене по истечении определенного периода времени.

Затем ожидается метод `SumPageSizesAsync`. Если обработка всех URL-адресов выполняется быстрее запланированной отмены, приложение завершается. Однако если запланированная отмена запускается до обработки всех URL-адресов, создается OperationCanceledException.

Пример выходных данных приложения

Консоль

```
Application started.

https://learn.microsoft.com                                37,357
https://learn.microsoft.com/aspnet/core                    85,589
https://learn.microsoft.com/azure                         398,939
https://learn.microsoft.com/azure/devops                   73,663

Tasks cancelled: timed out.

Application ending.
```

Полный пример

Приведенный ниже код — это полный текст файла *Program.cs* для примера.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }
```

```
        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

См. также

- CancellationToken
- CancellationTokenSource
- Асинхронное программирование с использованием ключевых слов Async и Await (C#)
- Отмена списка задач (C#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Руководство. Создание и использование асинхронных потоков с помощью C# и .NET

Статья • 25.03.2023

Асинхронные потоки моделируют источник потоковой передачи данных. Потоки данных часто извлекают или создают элементы асинхронно. Они предоставляют естественную модель программирования для асинхронных потоковых источников данных.

В этом руководстве вы узнаете, как:

- ✓ Создать источник данных, который формирует последовательность элементов данных асинхронно.
- ✓ Использовать этот источник данных асинхронно.
- ✓ Поддержка отмены и перехваченных контекстов для асинхронных потоков.
- ✓ Распознавать, когда новый интерфейс и источник данных предпочтительнее для более ранних синхронных последовательностей данных.

Предварительные требования

Вам потребуется настроить компьютер для запуска .NET, включая компилятор C#. Компилятор C# доступен в Visual Studio 2022 или пакете SDK для .NET.

Чтобы вы могли получить доступ к конечной точке GraphQL GitHub, необходимо создать маркер доступа GitHub. Выберите следующие разрешения для маркеров доступа GitHub.

- repo:status
- public_repo

Храните маркер доступа в надежном месте, чтобы вы могли использовать его для получения доступа к конечной точке API GitHub.

> ⚠ **Предупреждение**
>
> Храните свой личный маркер доступа в безопасном месте. Любое программное обеспечение с вашим личным маркером доступа может выполнять вызовы API GitHub с помощью ваших прав доступа.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Запуск начального приложения

Код для начального приложения, используемого в этом руководстве, можно получить из репозитория dotnet/docs в папке асинхронного программирования и фрагментов кода .

Начальное приложение представляет собой консольное приложение, которое использует интерфейс GraphQL GitHub для получения последних проблем, написанных в репозитории dotnet/docs. Начнем с просмотра следующего кода для метода `Main` начального приложения.

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-
for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub
access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment
variable",
    "");

    var client = new GitHubClient(new
Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new
CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery,
"docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
```

```
            Console.WriteLine(issue);
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Work has been cancelled");
        }
    }
```

Вы можете задать переменную среды `GitHubKey` личному маркеру доступа или заменить последний аргумент в вызове на `GetEnvVariable` с помощью личного маркера доступа. Не размещайте код доступа в исходном коде, если будете предоставлять общий доступ к источнику другим пользователям. Никогда не отправляйте коды доступа в репозиторий с общим исходным кодом.

После создания клиента GitHub код в `Main` создает объект отчета о ходе выполнения и маркер отмены. После создания этих объектов `Main` вызывает `RunPagedQueryAsync`, чтобы получить более 250 недавно созданных проблем. Результаты отобразятся после выполнения этой задачи.

При запуске начального приложения вы можете обнаружить некоторые важные замечания о том, как будет выполняться приложение. Вы увидите ход выполнения, передаваемый каждой странице, возвращенной с GitHub. Прежде чем GitHub вернет каждую новую страницу проблем, возникает заметная пауза. Наконец, проблемы отображаются только после того, как получены все 10 страниц с GitHub.

Изучение реализации

Реализация показывает, почему возникло поведение, обсуждавшееся в предыдущем разделе. Изучите код для `RunPagedQueryAsync`.

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,
string queryText, string repoName, CancellationToken cancel, IProgress<int>
progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;
```

```
    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();
        finalResults.Merge(issues(results)["nodes"]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

Давайте сконцентрируемся на алгоритме разбивки по страницам и асинхронной структуре предыдущего кода. (Дополнительные сведения об API GraphQL GitHub см. в этой документации.) Метод `RunPagedQueryAsync` перечисляет проблемы от самых последних до самых старых. Чтобы продолжить с предыдущей страницы, он запрашивает по 25 выпусков на страницу и проверяет структуру ответа `pageInfo`. Это следует за стандартной поддержкой страниц GraphQL для многостраничных ответов. Ответ включает в себя объект `pageInfo`, который содержит значение `hasPreviousPages` и `startCursor`, используемые для запроса предыдущей страницы. Проблемы в массиве `nodes`. Метод `RunPagedQueryAsync` добавляет эти узлы в массив, который содержит результаты со всех страниц.

После получения и восстановления страницы результатов `RunPagedQueryAsync` сообщает о ходе выполнения и проверяет наличие отмены. Если есть запрос на отмену, `RunPagedQueryAsync` выдает OperationCanceledException.

Существует несколько элементов в этом коде, которые можно улучшить. Самое главное, `RunPagedQueryAsync` должен выделить хранилище для всех возвращенных проблем. Этот пример останавливается после нахождения 250 проблем, так как для извлечения всех открытых проблем потребуется гораздо больше памяти на их

хранение. Протоколы для поддержки отчетов о ходе выполнения и отмены делают алгоритм более сложным для понимания при первом чтении. Задействовано больше типов и API. Вы должны отслеживать передачу данных с помощью CancellationTokenSource и связанного с ним CancellationToken, чтобы понять, где запрашивается отмена и где она предоставляется.

Предоставление лучшего способа асинхронных потоков

Асинхронные потоки и связанная языковая поддержка обращаются ко всем этим вопросам. Код, который формирует последовательность, теперь может использовать `yield return` для возврата элементов в методе, который был объявлен с помощью модификатора `async`. Вы можете применить асинхронный поток, используя цикл `await foreach`, аналогично любой последовательности с помощью цикла `foreach`.

Эти новые языковые функции зависят от трех новых интерфейсов, добавленных в .NET Standard 2.1 и реализованных в .NET Core 3.0.

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

Большинство разработчиков C# должны знать об этих трех интерфейсах. Они ведут себя подобно своим синхронным аналогам.

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

Один тип, который может быть незнаком, — System.Threading.Tasks.ValueTask. Структура `ValueTask` предоставляет API, аналогичный классу System.Threading.Tasks.Task. `ValueTask` используется в этих интерфейсах по причинам производительности.

Преобразование в асинхронные потоки

Затем для создания асинхронного потока преобразуйте метод `RunPagedQueryAsync`. Сначала измените подпись `RunPagedQueryAsync`, чтобы вернуть `IAsyncEnumerable<JToken>`, затем удалите маркер отмены и объекты хода выполнения из списка параметров, как показано в следующем коде.

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

В следующем коде показано, как начальный код обрабатывает каждую страницу для извлечения.

C#

```
finalResults.Merge(issues(results)["nodes"]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Замените эти три строки следующим кодом.

C#

```
foreach (JObject issue in issues(results)["nodes"]!)
    yield return issue;
```

Вы также можете удалить объявление `finalResults` ранее в этом методе и оператор `return`, следующий за измененным циклом.

Вы завершили изменения для создания асинхронного потока. Готовый метод должен иметь вид, аналогичный приведенному ниже коду:

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
```

```
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

Затем измените код, который использует коллекцию, для асинхронного потока. Найдите следующий код в `Main`, который обрабатывает коллекцию проблем.

C#

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Замените код следующим циклом `await foreach`.

C#

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

Новый интерфейс IAsyncEnumerator<T> является производным от IAsyncDisposable. Это означает, что предыдущий цикл будет асинхронно удалять поток по завершении цикла. Цикл похож на следующий код:

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Элементы потока по умолчанию обрабатываются в захваченном контексте. Чтобы отключить захват контекста, используйте метод расширения TaskAsyncEnumerableExtensions.ConfigureAwait. Дополнительные сведения о контекстах синхронизации и захвате текущего контекста см. в статье, посвященной использованию асинхронной модели на основе задач.

Асинхронные потоки поддерживают отмену, используя тот же протокол, что и другие методы `async`. Для поддержки отмены можно изменить сигнатуру для метода асинхронного итератора следующим образом:

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
```

```
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

Атрибут System.Runtime.CompilerServices.EnumeratorCancellationAttribute заставляет компилятор создать код для IAsyncEnumerator<T>, который делает токен, передаваемый `GetAsyncEnumerator`, видимым в тексте асинхронного итератора в виде аргумента. Внутри `runQueryAsync` можно проверить состояние маркера и отменить дальнейшую работу при необходимости.

Используйте другой метод расширения, WithCancellation, чтобы передать токен отмены асинхронному потоку. Измените цикл, перечисляя проблемы следующим образом:

C#

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
```

```
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Код для готового руководства можно получить из репозитория dotnet/docs в папке асинхронного программирования и фрагментов кода .

Запуск готового приложения

Снова запустите приложение. Сравните его поведение с поведением начального приложения. Первая страница результатов перечисляется, как только она становится доступной. Поскольку каждую новую страницу запрашивают и извлекают, результаты следующей страницы быстро перечисляются, возникает пауза. Блок `try` / `catch` не требует обработки отмены. Вызывающий может прекратить перечисление коллекции. Отчет о ходе выполнения четко сформирован, так как асинхронный поток формирует результаты скачивания каждой страницы. Состояние каждой возвращенной проблемы включается в цикл `await foreach`. Для отслеживания хода выполнения объект обратного вызова не требуется.

Изучив код, вы увидите улучшения в использовании памяти. Вам больше не нужно выделять коллекцию для хранения всех результатов до их перечисления. Вызывающий может определить, как использовать результаты и нужен ли набор хранилищ.

Запустите начальное и готовое приложение, и вы увидите различия между реализациями самостоятельно. Вы можете удалить маркер доступа GitHub, созданный при начале работы с этим руководством, после завершения изучения. Если злоумышленник получил доступ к этому маркеру, ему удастся получить доступ к API GitHub с помощью ваших учетных данных.

В этом руководстве вы использовали асинхронные потоки для чтения отдельных элементов из сетевого API, который возвращает страницы данных. Асинхронные потоки также могут считывать из "бесконечных потоков", таких как тиккер акций или устройство датчика. Вызов возвращает `MoveNextAsync` следующий элемент, как только он будет доступен.

Collaborate with us on GitHub

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.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Методы в C#

Статья • 17.04.2025

Метод — это блок кода, содержащий ряд инструкций. Программа инициирует выполнение инструкций, вызывая метод и указывая все аргументы, необходимые для этого метода. В C# все инструкции выполняются в контексте метода.

> ⓘ **Примечание**
>
> В этой статье рассматриваются именованные методы. Дополнительные сведения об анонимных функциях см. в статье Лямбда-выражения.

Сигнатуры методов

Методы объявляются в `class`, `record` или `struct` с указанием следующих данных.

- Уровень доступа (необязательно), например `public` или `private`. Значение по умолчанию — `private`.
- Необязательные модификаторы, например `abstract` или `sealed`.
- Возвращаемое значение или `void`, если у метода его нет.
- Имя метода.
- Любые параметры методов. Параметры метода заключаются в скобки и разделяются запятыми. Пустые скобки указывают, что параметры методу не требуются.

Вместе все эти части формируют сигнатуру метода.

> ⓘ **Важно!**
>
> Возвращаемый тип метода не является частью сигнатуры метода в целях перегрузки метода. Однако она является частью сигнатуры метода при определении совместимости между делегатом и методом, на который он указывает.

В следующем примере определяется класс с именем `Motorcycle`, содержащий пять методов:

C#

```
namespace MotorCycleExample
{
    abstract class Motorcycle
```

```
    {
        // Anyone can call this.
        public void StartEngine() {/* Method statements here */ }

        // Only derived classes can call this.
        protected void AddGas(int gallons) { /* Method statements here */ }

        // Derived classes can override the base class implementation.
        public virtual int Drive(int miles, int speed) { /* Method statements here
*/ return 1; }

        // Derived classes can override the base class implementation.
        public virtual int Drive(TimeSpan time, int speed) { /* Method statements
here */ return 0; }

        // Derived classes must implement this.
        public abstract double GetTopSpeed();
    }
```

Класс `Motorcycle` включает перегруженный метод. `Drive` Два метода имеют одинаковое имя, но их списки параметров различают их.

Вызов метода

Можно использовать метод *instance* или *static*. Необходимо создать экземпляр объекта для вызова метода экземпляра в этом экземпляре; Метод экземпляра работает с этим экземпляром и его данными. Вы вызываете статический метод, ссылаясь на имя типа, к которому принадлежит метод; статические методы не работают с данными экземпляра. При попытке вызвать статический метод с помощью экземпляра объекта возникает ошибка компилятора.

Вызов метода аналогичен доступу к полю. После имени объекта (при вызове метода экземпляра) или имени типа (если вы вызываете `static` метод), добавьте период, имя метода и круглые скобки. Аргументы перечисляются в этих скобках и разделяются запятыми.

Определение метода задает имена и типы всех необходимых параметров. Когда вызывающий код вызывает метод, он предоставляет конкретные значения, называемые аргументами, для каждого параметра. Аргументы должны быть совместимы с типом параметра, но имя аргумента, если он используется в вызывающем коде, не должен совпадать с параметром, указанным в методе. В следующем примере метод `Square` имеет один параметр типа `int` с именем *i*. Первый вызов метода передает методу `Square` переменную типа `int` с именем *num*; второй — числовую константу, а третий — выражение.

```
```

C#

```
public static class SquareExample
{
    public static void Main()
    {
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}
```

В наиболее распространенной форме вызова методов используются позиционные аргументы; они передаются в том же порядке, что и параметры метода. Таким образом, методы класса `Motorcycle` могут вызываться, как показано в следующем примере. Например, вызов метода `Drive` включает два аргумента, которые соответствуют двум параметрам в синтаксисе метода. Первый становится значением `miles` параметра. Второй становится значением `speed` параметра.

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        _ = moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine($"My top speed is {speed}");
    }
}
```

При вызове метода вместо позиционных аргументов можно также использовать *именованные аргументы*. При использовании именованных аргументов необходимо указать имя параметра, двоеточие (":"), а затем аргумент. Аргументы для метода могут отображаться в любом порядке, при условии, что все обязательные аргументы присутствуют. В следующем примере для вызова метода `TestMotorcycle.Drive` используются именованные аргументы. В этом примере именованные аргументы передаются из списка параметров метода в обратном порядке.

C#

```
namespace NamedMotorCycle;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed) =>
        (int)Math.Round((double)miles / speed, 0);

    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        int travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine($"Travel time: approx. {travelTime} hours");
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

Метод можно вызывать, используя и позиционные, и именованные аргументы. Однако позиционные аргументы могут следовать за именованными аргументами, только если именованные аргументы находятся в правильных позициях. В следующем примере метод `TestMotorcycle.Drive` из предыдущего примера вызывается с использованием одного позиционного и одного именованного аргумента.

C#

```
int travelTime = moto.Drive(170, speed: 55);
```

Унаследованные и переопределенные методы

Помимо членов, определенных в нем явно, тип наследует члены, определенные в его базовых классах. Так как все типы в системе управляемых типов напрямую или косвенно

наследуются из класса Object, все типы наследуют его члены, такие как Equals(Object), GetType() и ToString(). В следующем примере определяется класс `Person`, который создает экземпляры двух объектов `Person` и вызывает метод `Person.Equals`, чтобы определить, равны ли эти объекты. Однако `Equals` метод не определен в `Person` классе; он наследуется от Object.

C#

```
public class Person
{
    public string FirstName = default!;
}

public static class ClassTypeExample
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
}
// The example displays the following output:
//       p1 = p2: False
```

Типы могут переопределять унаследованные члены, используя ключевое слово `override` и реализуя переопределенный метод. Сигнатура метода должна совпадать с сигнатурой переопределяемого метода. Следующий пример аналогичен предыдущему за тем исключением, что переопределяет метод Equals(Object). (Он также переопределяет метод GetHashCode(), поскольку оба эти метода предназначены для получения согласованных результатов.)

C#

```
namespace methods;

public class Person
{
    public string FirstName = default!;

    public override bool Equals(object? obj) =>
        obj is Person p2 &&
        FirstName.Equals(p2.FirstName);

    public override int GetHashCode() => FirstName.GetHashCode();
}

public static class Example
{
```

```
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
}
// The example displays the following output:
//       p1 = p2: True
```

Передача параметров

Типы в C# делятся на *типы значений* и *ссылочные типы*. Список встроенных типов значений см. в разделе Типы. По умолчанию типы значений и ссылочные типы передаются по значению в метод.

Передача параметров по значению

Когда значимый тип передается в метод по значению, вместо самого объекта передается его копия. Это значит, что изменения объекта в вызываемом методе не отражаются на исходном объекте, когда управление возвращается вызывающему объекту.

В следующем примере значимый тип передаётся в метод по значению, и вызываемый метод пытается изменить его значение. Он определяет переменную типа `int`, который является типом значения, присваивает ему значение 20 и передает его в метод с именем `ModifyValue`, который изменяет значение переменной на 30. Однако, когда метод возвращается, значение переменной остается неизменным.

C#

```
public static class ByValueExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
```

```
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Если объект ссылочного типа передается в метод по значению, ссылка на этот объект передается по значению. Это значит, что метод получает не сам объект, а аргумент, который указывает расположение объекта. Если с помощью этой ссылки в член объекта вносится изменение, это изменение отражается в объекте, когда управление возвращается вызывающему методу. Однако замена объекта, переданного в метод, не оказывает никакого влияния на оригинальный объект, когда управление возвращается вызывающему объекту.

В следующем примере определяется класс (ссылочного типа) с именем `SampleRefType`. Он создает экземпляр объекта `SampleRefType`, задает в его поле `value` значение 44 и передает объект в метод `ModifyObject`. Этот пример фактически выполняет ту же задачу, что и предыдущий пример — он передает аргумент по значению методу. Однако поскольку здесь используется ссылочный тип, результат будет другим. В данном случае в методе `ModifyObject` изменено поле `obj.value`, при этом поле `value` аргумента `rt` в методе `Main` также изменяется на 33, как видно из результатов в предыдущем примере.

C#

```
public class SampleRefType
{
    public int value;
}

public static class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType { value = 44 };
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj) => obj.value = 33;
}
```

Передача параметров по ссылке

Параметр передается по ссылке, когда нужно изменить значение аргумента в методе и отразить это изменение после того, как управление вернется к вызывающему методу. Для передачи параметра по ссылке используйте ключевое слово ref или out. Можно также

передать значение по ссылке, чтобы предотвратить копирование, и при этом запретить внесение изменений с помощью ключевого слова in.

Следующий пример идентичен предыдущему, за исключением того, что значение передается ссылкой на `ModifyValue` метод. Если значение параметра изменить в методе `ModifyValue`, то это изменение отразится, когда управление вернётся к вызывающему объекту.

C#

```
public static class ByRefExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    private static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

Общий шаблон, при котором параметры передаются по ссылке, предполагает обмен значениями переменных. Вы передаёте две переменные в метод по ссылке, и метод меняет их содержимое местами. В следующем примере происходит перестановка целочисленных значений.

C#

```


public static class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        Console.WriteLine($"i = {i}  j = {j}");

        Swap(ref i, ref j);

        Console.WriteLine($"i = {i}  j = {j}");
```

```
    }

    static void Swap(ref int x, ref int y) =>
        (y, x) = (x, y);
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Передача параметров ссылочного типа позволяет изменить значение самой ссылки, а не отдельных ее элементов или полей.

Коллекции параметров

В некоторых случаях требование об указании точного числа аргументов для метода является строгим. Используя ключевое `params` слово, указывающее, что параметр является коллекцией параметров, можно вызывать метод с переменным числом аргументов. Параметр, `params` помеченный ключевым словом, должен быть типом коллекции, и он должен быть последним параметром в списке параметров метода.

Вызывающий объект может вызвать метод в любом из четырех способов для `params` параметра:

- Путем передачи коллекции соответствующего типа, содержащей требуемое количество элементов. В примере используется выражение коллекции, поэтому компилятор создает соответствующий тип коллекции.
- передавая в метод список отдельных аргументов соответствующего типа, разделенный запятыми; Компилятор создает соответствующий тип коллекции.
- Путем передачи `null`.
- Не предоставляя аргумент для коллекции параметров.

В следующем примере определяется метод с именем `GetVowels` , который возвращает все гласные из коллекции параметров. Метод `Main` демонстрирует все четыре способа вызова метода. Вызывающие не обязаны предоставлять аргументы для параметров с модификатором `params`. В этом случае параметр является пустой коллекцией.

C#

```
static class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(["apple", "banana", "pear"]);
        Console.WriteLine($"Vowels from collection expression: '{fromArray}'");
```

```
        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params IEnumerable<string>? input)
    {
        if (input == null || !input.Any())
        {
            return string.Empty;
        }

        char[] vowels = ['A', 'E', 'I', 'O', 'U'];
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Перед C# 13 `params` модификатор можно использовать только с одниммерным массивом.

Необязательные параметры и аргументы

Определение метода может указать, что его параметры являются обязательными или являются необязательными. По умолчанию параметры обязательны. Для определения необязательных параметров значения параметра по умолчанию включаются в определение метода. Если при вызове метода никакие аргументы для необязательного параметры не указываются, вместо них используется значение по умолчанию.

Значение по умолчанию параметра назначается одним из следующих типов выражений:

- Константа, например, строковый литерал или число.
- Выражение, имеющее вид `default(SomeType)`, где `SomeType` может быть либо типом значения, либо ссылочным типом. Если это ссылочный тип, это фактически то же

самое, что указание `null`. Вы можете использовать литерал `default`, так как компилятор может определить тип из объявления параметра.

- Выражение в форме `new ValType()`, где `ValType` — это тип значения. Это выражение вызывает неявный конструктор типа значения без параметров, который не является фактическим членом типа.

  > ⓘ **Примечание**
  >
  > Если выражение формы `new ValType()` вызывает явно определенный конструктор без параметров типа значения, компилятор создает ошибку, так как значение параметра по умолчанию должно быть константой времени компиляции. Используйте выражение `default(ValType)` или литерал `default` для предоставления значения параметра по умолчанию. Дополнительные сведения о конструкторах без параметров см. в **разделе "Инициализация структуры и значения** по умолчанию" статьи "Типы структур".

Если метод содержит как обязательные, так и необязательные параметры, необязательные параметры определяются в конце списка параметров после всех обязательных параметров.

В следующем примере определяется метод `ExampleMethod`, который имеет один обязательный и два необязательных параметра.

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} = {required
+ optionalInt}";
        Console.WriteLine(msg);
    }
}
```

Вызывающий объект должен указать аргумент для всех необязательных параметров до последнего необязательного параметра, для которого предоставляется аргумент. В методе `ExampleMethod`, например, если вызывающая сторона предоставляет аргумент для параметра `description`, она также должна указать аргумент для параметра `optionalInt`. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` — допустимый вызов метода;

`opt.ExampleMethod(2, , "Addition of 2 and 0");` вызывает ошибку компилятора "Аргумент отсутствует".

Если метод вызывается с помощью именованных аргументов или комбинации позиционных и именованных аргументов, вызывающий объект может опустить любые аргументы, следующие за последним позиционным аргументом в вызове метода.

В следующем примере метод `ExampleMethod` вызывается трижды. В первых двух вызовах метода используются позиционные аргументы. В первом пропускаются оба необязательных аргумента, а во втором — последний. Третий вызов метода предоставляет позиционный аргумент для обязательного параметра, но использует именованный аргумент для передачи значения в параметр `description`, в то время как аргумент `optionalInt` опускается.

C#

```
public static class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

Использование необязательных параметров влияет на *разрешение* перегрузки или способ, которым компилятор C# определяет, какая перегрузка вызывается для вызова метода, как показано ниже.

- Член является кандидатом на выполнение, если каждый из его параметров соответствует по имени или по позиции к одному аргументу. Кроме того, этот аргумент можно преобразовать в тип параметра.
- Если найдено более одного кандидата, правила разрешения перегрузки для предпочтительных преобразований применяются к аргументам, указанным явно. Опущенные аргументы для необязательных параметров игнорируются.
- Если два кандидата оцениваются как одинаково хорошие, предпочтение отдается кандидату, вызов которого не содержит опущенных аргументов для необязательных параметров.

Возвращаемые значения

Методы могут возвращать значение вызывающей стороне. Если возвращаемый тип (тип, указанный перед именем метода) не является `void`, метод может возвращать значение, используя ключевое слово `return`. Инструкция с ключевым словом `return`, за которым следует переменная, константа или выражение, соответствующее типу возвращаемого значения, возвращает это значение вызывающему методу. Методы с ненулевым типом возвращаемого значения должны использовать ключевое слово `return` для возврата значения. Ключевое слово `return` также останавливает выполнение метода.

Если тип возврата — `void`, инструкцию `return` без значения по-прежнему можно использовать для завершения выполнения метода. `return` Без ключевого слова метод перестает выполняться, когда он достигает конца блока кода.

Например, в следующих двух методах ключевое слово `return` используется для возврата целочисленных значений.

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2) =>
        number1 + number2;

    public int SquareANumber(int number) =>
        number * number;
}
```

В предыдущих примерах приведены элементы, входящие в выражение. Члены выражения возвращают значение, возвращаемое выражением.

Вы также можете определить методы с текстом инструкции и оператором `return` :

C#

```
class SimpleMathExtension
{
    public int DivideTwoNumbers(int number1, int number2)
    {
        return number1 / number2;
    }
}
```

Чтобы использовать значение, возвращаемое из метода, можно назначить возвращаемое значение переменной:

C#

```
int result = obj.DivideTwoNumbers(6,2);
// The result is 3.
Console.WriteLine(result);
```

Вызывающий метод может также использовать сам вызов метода в любом месте, где достаточно значения того же типа. Например, следующие два примера кода выполняют одну и ту же задачу:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

В некоторых случаях нужно, чтобы метод возвращал больше одного значения. Для возврата нескольких значений используются *кортежные типы* и *кортежные литералы*. Тип кортежа определяет типы данных для элементов кортежа. Литералы кортежей определяют фактические значения возвращаемого кортежа. В следующем примере `(string, string, string, int)` определяет тип кортежа, возвращаемый методом `GetPersonalInfo`. Выражение `(per.FirstName, per.MiddleName, per.LastName, per.Age)` представляет собой литерал кортежа. Метод возвращает первое, среднее и семейное имя, а также возраст `PersonInfo` объекта.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

Вызывающий объект затем может использовать возвращенный кортеж с помощью следующего кода:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Имена также могут быть присвоены элементам кортежа при определении его типа. В следующих примерах демонстрируется альтернативная версия метода `GetPersonalInfo`, в котором используются именованные элементы:

C#

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string
id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

После этого предыдущий вызов метода `GetPersonalInfo` можно изменить следующим образом:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Если метод принимает массив в качестве параметра и изменяет значение отдельных элементов, нет необходимости, чтобы метод возвращал массив. C# передает все ссылочные типы по значению, а значение ссылки на массив — указатель на массив. В следующем примере изменения в содержимом массива `values`, сделанные в методе `DoubleValues`, может отслеживать любой код, имеющий ссылку на этот массив.

C#

```


public static class ArrayValueExample
{
    static void Main()
    {
        int[] values = [2, 4, 6, 8];
        DoubleValues(values);
        foreach (var value in values)
        {
            Console.Write("{0}  ", value);
        }
    }

    public static void DoubleValues(int[] arr)
```

```
    {
        for (var ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
        {
            arr[ctr] *= 2;
        }
    }
}
// The example displays the following output:
//       4  8  12  16
```

Методы расширения

Как правило, добавлять методы в существующий тип можно двумя способами:

- Измените исходный код этого типа. Изменение источника создает разрушающее изменение, если вы также добавляете закрытые поля данных для поддержки метода.
- Определение нового метода в производном классе. Метод нельзя добавить таким образом с помощью наследования для других типов, таких как структуры и перечисления. Кроме того, оно не позволяет "добавить" метод в запечатанный класс.

Члены расширения позволяют добавлять элементы в существующий тип, не изменяя сам тип или реализуя новый метод в наследуемом типе. Член расширения также не должен находиться в той же сборке, что и тип, который он расширяет. Вызовите метод расширения, как будто он является определенным членом типа.

Дополнительные сведения см. в разделе "Члены расширения".

Асинхронные методы

С помощью функции async можно вызывать асинхронные методы, не прибегая к использованию явных обратных вызовов или ручному разделению кода между несколькими методами или лямбда-выражениями.

Если пометить метод с помощью модификатора async, можно использовать в этом методе инструкцию await. Когда элемент управления достигает `await` выражения в асинхронном методе, элемент управления возвращается вызывающему объекту, если ожидаемая задача не завершена, и ход выполнения метода с `await` ключевым словом приостановлен до завершения ожидаемой задачи. После завершения задачи выполнение в методе может быть возобновлено.

> ⓘ **Примечание**

Асинхронный метод возвращается в вызывающий объект, когда он встречает первый ожидаемый объект, выполнение которого еще не завершено, или когда выполнение асинхронного метода доходит до конца — в зависимости от того, что происходит раньше.

Асинхронный метод обычно имеет тип возвращаемого значения Task<TResult>, Task, IAsyncEnumerable<T> или `void`. Тип возвращаемого значения `void` в основном используется для определения обработчиков событий, где требуется возвращать тип `void`. Асинхронный метод, который возвращает тип `void`, не может быть ожидающим. Вызывающий объект метода, возвращающего значение типа void, не может перехватывать исключения, которые выдает этот метод. Асинхронный метод может иметь любой тип возвращаемого значения, аналогичный задаче.

В следующем примере `DelayAsync` представляет собой асинхронный метод с оператором return, который возвращает целое число. Так как это асинхронный метод, его объявление метода должно иметь возвращаемый тип `Task<int>`. Поскольку тип возврата — `Task<int>`, вычисление выражения `await` в `DoSomethingAsync` создает целое число, как показывает следующий оператор: `int result = await delayTask`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Асинхронный метод не может объявлять параметры in, ref или out, но может вызывать методы, имеющие такие параметры.

Дополнительные сведения об асинхронных методах см. в разделах Асинхронное программирование с ключевыми словами async и await (C#) и Типы возвращаемых значений асинхронных методов.

Элементы, воплощающие выражение

Обычно используются определения методов, которые сразу возвращают результат выражения, либо имеют единственное выражение в качестве тела метода. Есть синтаксическая форма для определения таких методов с помощью `=>`.

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Если метод возвращает `void` или является асинхронным, текст этого метода должен быть выражением оператора (как и при использовании лямбда-выражений). Свойства и индексаторы должны быть только для чтения, и ключевое слово `get` accessor не используется.

Итераторы

Итератор выполняет настраиваемую итерацию по коллекции, например по списку или массиву. Итератор использует инструкцию yield return для возврата всех элементов по одному. По достижении утверждения `yield return`, текущее положение сохраняется в памяти, чтобы вызывающий мог запросить следующий элемент в последовательности.

Тип возвращаемого итератора может быть IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumeratorили IEnumerator<T>.

Дополнительные сведения см. в разделе Итераторы.

См. также

- Модификаторы доступа

- Статические классы и члены статических классов
- Наследование
- Абстрактные и закрытые классы и члены классов
- params
- вне
- ref;
- в
- Передача параметров

Итераторы

17.06.2025

Почти в каждой написанной вами программе возникнет необходимость в итерации по коллекции. Вы напишете код, который проверяет каждый элемент в коллекции.

Вы также создадите метод итератора, который создаёт *итератор* для элементов этого класса. *Итератор* — это объект, который проходит через контейнер, особенно списки. Итераторы можно использовать для:

- Выполнение действия для каждого элемента в коллекции.
- Перечисление коллекции, определенной пользователем.
- Расширение LINQ или других библиотек.
- Создание конвейера данных, в котором данные эффективно передаются с помощью методов итератора.

Язык C# предоставляет функции для создания и использования последовательностей. Эти последовательности можно производить и использовать синхронно или асинхронно. В этой статье представлен обзор этих функций.

Итерация с использованием foreach

Перечисление коллекции - это просто: ключевое слово `foreach` перечисляет коллекцию, выполняя внедренную инструкцию один раз для каждого элемента коллекции.

C#

```
foreach (var item in collection)
{
    Console.WriteLine(item?.ToString());
}
```

Это всё. Чтобы выполнить итерацию по всем элементам коллекции, достаточно `foreach` инструкции. Это не магия `foreach` , хотя. Он использует два универсальных интерфейса, определенных в библиотеке .NET Core, чтобы создать код, необходимый для итерации коллекции: `IEnumerable<T>` и `IEnumerator<T>`. Этот механизм более подробно описан ниже.

Оба этих интерфейса также имеют не универсальные аналоги: `IEnumerable` и `IEnumerator`. Универсальные версии предпочтительнее для современного кода.

При асинхронной генерации последовательности можно использовать `await foreach` инструкцию для асинхронного потребления последовательности.

C#

```
await foreach (var item in asyncSequence)
{
Console.WriteLine(item?.ToString());
}
```

Если последовательность является последовательностью System.Collections.Generic.IEnumerable<T>, используется `foreach`. Если последовательность является последовательностью System.Collections.Generic.IAsyncEnumerable<T>, используется `await foreach`. В последнем случае последовательность создается асинхронно.

Источники перечисления с методами итератора

Еще одна отличная функция языка C# позволяет создавать методы, создающие источник перечисления. Эти *методы называются методами итератора*. Метод итератора определяет, как создавать объекты в последовательности при запросе. Вы используете `yield return` контекстные ключевые слова для определения метода итератора.

Этот метод можно написать для создания последовательности целых чисел от 0 до 9:

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

В приведенном выше коде показаны отдельные `yield return` инструкции, чтобы выделить тот факт, что в методе итератора можно использовать несколько дискретных `yield return` инструкций. Вы можете (и часто) использовать другие конструкции языка, чтобы упростить код метода итератора. Определение метода, приведенное ниже, создает ту же последовательность чисел:

C#

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

Вам не нужно решать одну или другую. Вы можете иметь столько `yield return` вставок, сколько необходимо для удовлетворения потребностей вашего метода:

C#

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Все эти предыдущие примеры имеют асинхронный аналог. В каждом случае вы замените тип возвращаемого `IEnumerable<T>` на `IAsyncEnumerable<T>`. Например, предыдущий пример будет иметь следующую асинхронную версию:

C#

```
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Это синтаксис синхронных и асинхронных итераторов. Рассмотрим реальный пример. Представьте, что вы находитесь в проекте Интернета вещей, и датчики устройств создают очень большой поток данных. Чтобы получить представление о данных, можно написать метод, который берёт каждый N-й элемент данных. Этот небольшой метод итератора решает задачу.

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence, int
interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Если чтение с устройства Интернета вещей создаст асинхронную последовательность, измените метод, как показано ниже.

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Существует одно важное ограничение для методов итератора: вы не можете использовать в одном методе одновременно оператор `return` и оператор `yield return`. Следующий код не компилируется:

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;
```

```
    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}
```

Обычно это ограничение не является проблемой. У вас есть выбор: либо использовать `yield return` во всем методе, либо разделить исходный метод на несколько методов, некоторые из которых используют `return`, а другие — `yield return`.

Вы можете немного изменить последний метод, чтобы использовать `yield return` везде:

C#

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

Иногда правильный ответ — разделить метод итератора на два разных метода. Одна использует `return`, а вторая использует `yield return`. Рассмотрим ситуацию, когда требуется вернуть пустую коллекцию или первые пять нечетных чисел, в зависимости от значения логического аргумента. Вы могли бы записать это как два метода:

C#

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
```

```
        index++;
    }
}
```

Ознакомьтесь с приведенными выше методами. Первый использует стандартную `return` инструкцию для возврата пустой коллекции или итератора, созданного вторым методом. Второй метод использует инструкцию `yield return` для создания запрошенной последовательности.

Более глубокое погружение в `foreach`

Инструкция `foreach` разворачивается в стандартную идиому, которая использует интерфейсы `IEnumerable<T>` и `IEnumerator<T>` для итерации по всем элементам коллекции. Она также сводит к минимуму ошибки разработчиков, не управляя ресурсами должным образом.

Компилятор преобразует цикл, показанный `foreach` в первом примере, в что-то похожее на эту конструкцию:

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

Точный код, созданный компилятором, более сложен и обрабатывает ситуации, когда объект, возвращаемый `GetEnumerator()`, реализует интерфейс `IDisposable`. Полное расширение генерирует код, похожий на следующий пример.

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
```

```
        // dispose of enumerator.
    }
}
```

Компилятор преобразует первый асинхронный пример в что-то похожее на эту конструкцию:

C#

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

Способ удаления перечислителя зависит от характеристик типа `enumerator`. В общем синхронном случае `finally` пункт расширяется следующим образом:

C#

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

Общий асинхронный случай расширяется до:

C#

```
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

Однако если тип `enumerator` является запечатанным, и нет неявного преобразования из типа `enumerator` в `IDisposable` или `IAsyncDisposable`, то предложение `finally`

расширяется до пустого блока.

C#

```
finally
{
}
```

Если существует неявное преобразование из типа `enumerator` в `IDisposable`, и `enumerator` является типом значения, недопускающим null, формулировка `finally` расширяется так:

C#

```
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

К счастью, вам не нужно помнить все эти детали. Оператор `foreach` обрабатывает все эти нюансы для вас. Компилятор создаст правильный код для любой из этих конструкций.

Общие сведения о делегатах и событиях в C#

17.06.2025

Делегаты предоставляют *механизм поздней привязки* в .NET. Поздняя привязка означает, что создается алгоритм, в котором вызывающий объект также предоставляет по крайней мере один метод, реализующий часть алгоритма.

Например, рассмотрите возможность сортировки списка звезд в астрономическом приложении. Вы можете сортировать эти звезды по их расстоянию от земли, или величину звезды, или их воспринимаемую яркость.

Во всех этих случаях метод Sort() выполняет по существу одно и то же: упорядочивает элементы в списке на основе некоторого сравнения. Код, сравнивающий две звезды, отличается для каждого порядка сортировки.

Эти типы решений использовались в программном обеспечении в течение полувека. Концепция делегатов в языке C# обеспечивает поддержку языка на высшем уровне и безопасность типов в рамках концепции.

Как вы увидите далее в этой серии, код C#, который вы пишете для алгоритмов, таких как этот тип, является безопасным. Компилятор гарантирует, что типы соответствуют аргументам и возвращаемым типам.

указатели функций поддерживают аналогичные сценарии, где требуется больше контроля над соглашением о вызовах. Код, связанный с делегатом, вызывается с помощью виртуального метода, добавленного в тип делегата. С помощью указателей функций можно указать различные соглашения.

Цели проектирования языка для делегатов

Конструкторы языков перечислили несколько целей для функции, которая в конечном итоге стала делегатами.

Команда хотела создать общую языковую конструкцию, которая может использоваться для любых поздних алгоритмов привязки. Делегаты позволяют разработчикам изучать одну концепцию и использовать ту же концепцию во многих различных проблемах программного обеспечения.

Во-вторых, команда хотела поддерживать вызовы методов единичной передачи и многоадресной передачи. (Делегаты многоадресной рассылки — это делегаты, которые

объединяют несколько вызовов методов. Ниже вы увидите примеры в этой серии.)

Команда хотела, чтобы делегаты поддерживали ту же безопасность типов, которую разработчики ожидают от всех конструкций C#.

Наконец, команда признала, что шаблон событий является одним из конкретных шаблонов, где делегаты или любой алгоритм поздней привязки полезны. Команда хотела убедиться, что код делегатов может служить основой для шаблона событий .NET.

Результатом всей этой работы стала поддержка делегатов и событий в C# и .NET.

Остальные статьи в этой серии будут охватывать функции языка, поддержку библиотек и распространенные идиомы, используемые при работе с делегатами и событиями. Вы узнаете:

- Ключевое `delegate` слово и код, который он создает.
- Функции в `System.Delegate` классе и способы использования этих функций.
- Как создать безопасные для типов делегаты.
- Как создать методы, которые можно вызывать с помощью делегатов.
- Как работать с делегатами и событиями с помощью лямбда-выражений.
- Как делегаты становятся одним из строительных блоков в LINQ.
- Как делегаты являются основой для шаблона событий .NET и как они отличаются.

Давайте начнем.

Далее

Ključevoe slovo `delegate` i System.Delegate

16.07.2025

Предыдущий

В этой статье рассматриваются классы в .NET, которые поддерживают делегатов и как они сопоставляются с ключевым словом `delegate` .

Что такое делегаты?

Рассмотрим делегат как способ хранения ссылки на метод, аналогичный тому, как можно сохранить ссылку на объект. Так же, как можно передать объекты в методы, можно передать ссылки на методы с помощью делегатов. Это полезно, если вы хотите написать гибкий код, где различные методы могут быть "подключены" для обеспечения различных действий.

Например, представьте, что у вас есть калькулятор, который может выполнять операции с двумя числами. Вместо того чтобы жестко кодировать сложение, вычитание, умножение и деление в отдельные методы, можно использовать делегаты для представления любой операции, принимающей два числа и возвращающей результат.

Определение типов делегатов

Теперь давайте посмотрим, как создать типы делегатов с помощью ключевого `delegate` слова. При определении типа делегата вы, по сути, создаете шаблон, описывающий тип методов, которые могут храниться в этом делегате.

Тип делегата определяется с помощью синтаксиса, который выглядит как сигнатура метода, но с `delegate` ключевым словом в начале:

```
C#
```

```
// Define a simple delegate that can point to methods taking two integers and
returning an integer
public delegate int Calculator(int x, int y);
```

Этот `Calculator` делегат может содержать ссылки на любой метод, который принимает два `int` параметра и возвращает объект `int`.

Рассмотрим более практический пример. Если вы хотите сортировать список, необходимо сообщить алгоритму сортировки, как сравнить элементы. Давайте

посмотрим, как делегаты помогают с методом `List.Sort()` . Первым шагом является создание типа делегата для операции сравнения:

```
C#
```

```
// From the .NET Core library
public delegate int Comparison<in T>(T left, T right);
```

Этот `Comparison<T>` делегат может содержать ссылки на любой метод, который:

- Принимает два параметра типа `T`
- `int` Возвращает значение (обычно значение -1, 0 или 1, указывающее "меньше", "равно" или "больше")

При определении типа делегата, подобного этому, компилятор автоматически создает класс, производный от `System.Delegate`, который соответствует вашей сигнатуре. Этот класс берет на себя все сложности, связанные с хранением и вызовом ссылок на методы.

Тип `Comparison` делегата является универсальным типом, что означает, что он может работать с любым типом `T`. Дополнительные сведения о универсальных классах и методах см. в разделе "Универсальные классы и методы".

Обратите внимание, что несмотря на то, что синтаксис выглядит аналогично объявлению переменной, вы фактически объявляете новый *mun*. Можно определить типы делегатов внутри классов, непосредственно внутри пространств имен или даже в глобальном пространстве имен.

> ⓘ **Примечание**
>
> Объявление типов делегатов (или других типов) непосредственно в глобальном пространстве имен не рекомендуется.

Компилятор также создает обработчики для этого нового типа, чтобы клиенты этого класса могли добавлять и удалять методы из списка вызовов экземпляра. Компилятор принудительно применяет подпись добавляемого или удаленного метода, совпадающую с сигнатурой, используемой при объявлении типа делегата.

Объявление экземпляров для делегатов

После определения типа делегата можно создать экземпляры (переменные) этого типа. Подумайте об этом как о создании "слота", где можно сохранить ссылку на метод.

Как и для всех переменных в C#, нельзя объявлять экземпляры делегатов непосредственно в пространстве имен или в глобальном пространстве имен.

```
C#

// Inside a class definition:
public Comparison<T> comparator;
```

Тип этой переменной (`Comparison<T>` тип делегата, определенный ранее), а имя переменной — `comparator`. На этом этапе `comparator` пока не указывает на какой-либо метод— это как пустой слот, ожидающий заполнения.

Можно также объявить переменные делегата как локальные переменные или параметры метода, как и любой другой тип переменной.

Вызов делегатов

Получив экземпляр делегата, указывающий на метод, можно вызвать (вызвать) этот метод с помощью делегата. Вы вызываете методы, которые находятся в списке вызовов делегата, обращаясь к этому делегату так, как если бы он был методом.

Вот как `Sort()` метод использует делегат сравнения для определения порядка объектов:

```
C#

int result = comparator(left, right);
```

В этой строке код *вызывает* метод, подключенный к делегату. Переменная делегата обрабатывается как имя метода и вызывается с помощью синтаксиса вызова обычного метода.

Однако эта строка кода делает небезопасное предположение: предполагается, что целевой метод был добавлен делегату. Если методы не были присоединены, строка выше вызовет выброс ошибки `NullReferenceException`. Шаблоны, используемые для решения этой проблемы, более сложны, чем простая проверка null, и рассматриваются далее в этой серии.

Назначение, добавление и удаление целевых объектов вызова

Теперь вы знаете, как определять типы делегатов, объявлять экземпляры делегатов и вызывать делегатов. Как именно вы связываете метод с делегатом? Именно в этом случае происходит назначение делегата.

Чтобы использовать делегат, требуется присвоить ему метод. Назначенный метод должен иметь ту же сигнатуру (те же параметры и тип возвращаемого значения), что и тип делегата.

Рассмотрим практический пример. Предположим, вы хотите отсортировать список строк по их длине. Необходимо создать метод сравнения, соответствующий сигнатуре делегата `Comparison<string>` :

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

Этот метод принимает две строки и возвращает целое число, указывающее, какая строка "больше" (длиннее в данном случае). Метод объявлен как приватный, что совершенно нормально. Вам не нужно, чтобы метод был частью общедоступного интерфейса, чтобы использовать его с делегатом.

Теперь этот метод можно передать в метод `List.Sort()` :

C#

```
phrases.Sort(CompareLength);
```

Обратите внимание, что вы используете имя метода без скобок. Это сообщает компилятору преобразовать ссылку на метод в делегат, который можно вызвать позже. Метод `Sort()` вызывает `CompareLength` метод всякий раз, когда требуется сравнить две строки.

Кроме того, можно быть более явным, объявив переменную делегата и назначив метод для него:

C#

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Оба подхода выполняют то же самое. Первый подход является более кратким, а второй делает назначение делегата более явным.

Для простых методов обычно используются лямбда-выражения вместо определения отдельного метода:

```
C#
```

```
Comparison<string> comparer = (left, right) =>
left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Лямбда-выражения предоставляют компактный способ определения простых методов. Использование лямбда-выражений для целевых объектов делегатов подробно рассматривается в следующем разделе.

В примерах до сих пор показаны делегаты с одним целевым методом. Однако объекты делегатов могут поддерживать списки вызовов с несколькими целевыми методами, подключенными к одному объекту делегата. Эта возможность особенно полезна для сценариев обработки событий.

Классы Делегат и MulticastDelegate

За кулисами функции делегата, которые вы использовали, основаны на двух ключевых классах в платформе .NET: Delegate и MulticastDelegate. Обычно вы не работаете с этими классами напрямую, но они предоставляют основу, которая позволяет делегатам функционировать.

Класс `System.Delegate` и его прямой подкласс `System.MulticastDelegate` обеспечивают поддержку платформы для создания делегатов, регистрации методов в качестве целевых объектов делегата и вызова всех методов, зарегистрированных в делегате.

Вот интересная деталь дизайна: `System.Delegate` и `System.MulticastDelegate` не являются сами типами делегатов, которые можно использовать. Вместо этого они служат базовыми классами для всех создаваемых типов делегатов. Язык C# не позволяет напрямую наследовать от этих классов. Вместо этого необходимо использовать ключевое `delegate` слово.

При использовании `delegate` ключевого слова для объявления типа делегата компилятор C# автоматически создает класс, производный от `MulticastDelegate` с вашей конкретной сигнатурой.

Почему этот дизайн?

Эта конструкция имеет свои корни в первом выпуске C# и .NET. Команда разработчиков имела несколько целей:

1. **Безопасность типов**: команда хотела убедиться, что при использовании делегатов язык применяет безопасность типов. Это требует, чтобы делегаты вызывались с правильным типом и количеством аргументов, а возвращаемые типы правильно проверялись на этапе компиляции.

2. **Производительность**. Путем создания конкретных классов делегатов, представляющих определенные подписи методов, среда выполнения может оптимизировать вызовы делегатов.

3. **Простота**. Делегаты были включены в выпуск .NET версии 1.0, который был до появления универсальных шаблонов. Дизайн должен был работать в рамках временных ограничений.

Решением было создание компилятором конкретных классов делегатов, которые соответствуют подписям метода, обеспечивая безопасность типов при скрытии сложности от вас.

Работа с методами делегата

Несмотря на то, что вы не можете создавать производные классы напрямую, иногда используйте методы, определенные для `Delegate` и `MulticastDelegate` классов. Ниже приведены наиболее важные из них, о которых нужно знать:

Каждый делегат, с которым вы работаете, является производным от `MulticastDelegate`. Делегат типа "многократный" означает, что при вызове через делегат можно вызвать более одного целевого метода. В первоначальном проекте рассматривается различие между делегатами, которые могут вызывать только один метод и делегаты, которые могут вызывать несколько методов. На практике это различие оказалось менее полезным, чем первоначально думалось, поэтому все делегаты в .NET поддерживают несколько целевых методов.

Наиболее часто используемые методы при работе с делегатами:

- `Invoke()`: вызывает все методы, подключенные к делегату
- `BeginInvoke()` / `EndInvoke()`: используется для шаблонов асинхронного вызова (хотя `async`/`await` теперь предпочтительнее)

В большинстве случаев вы не будете вызывать эти методы напрямую. Вместо этого вы будете использовать синтаксис вызова метода в переменной делегата, как показано в

приведенных выше примерах. Однако, как вы увидите далее в этой серии, существуют шаблоны, которые работают непосредственно с этими методами.

Сводка

Теперь, когда вы узнали, как синтаксис языка C# сопоставляется с базовыми классами .NET, вы можете изучить, как строго типизированные делегаты используются, создаются и вызываются в более сложных сценариях.

Далее

Строго типизированные делегаты

17.06.2025

Предыдущий

В предыдущей статье вы увидели, что создаете определенные типы делегатов с помощью ключевого слова `delegate`.

Абстрактный класс Делегат предоставляет инфраструктуру для свободного связывания и вызова. Конкретные типы делегатов становятся гораздо более полезными, используя и обеспечивая строгую типизацию для методов, добавленных в список вызовов объекта делегата. При использовании ключевого `delegate` слова и определении конкретного типа делегата компилятор создает эти методы.

На практике это приведет к созданию новых типов делегатов всякий раз, когда требуется другая подпись метода. Эта работа может стать утомительной со временем. Для каждой новой функции требуются новые типы делегатов.

К счастью, это не обязательно. Платформа .NET Core содержит несколько типов, которые можно использовать повторно при необходимости типов делегатов. Это общие определения, которые позволяют объявить настройки, когда необходимо новое объявление методов.

Первым из этих типов является Action тип и несколько вариантов:

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

Модификатор `in` для аргумента универсального типа рассматривается в статье по ковариации.

Существуют варианты делегата `Action` , содержащего до 16 аргументов, таких как Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. Важно, чтобы эти определения использовали разные универсальные аргументы для каждого из аргументов делегата: это обеспечивает максимальную гибкость. Аргументы метода могут быть разного типа, но также могут быть и одинакового типа.

Используйте один из `Action` типов для любого типа делегата, имеющего тип возвращаемого значения void.

Платформа также включает несколько универсальных типов делегатов, которые можно использовать для типов делегатов, возвращающих значения:

C#

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

Модификатор `out` для аргумента результирующих универсальных типов рассматривается в статье по ковариации.

Существуют варианты делегата `Func` с до 16 входных аргументов, таких как Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. Тип результата всегда указан последним среди параметров типа во всех `Func` объявлениях в соответствии с принятым соглашением.

Используйте один из `Func` типов для любого типа делегата, возвращающего значение.

Существует также специализированный Predicate<T> тип делегата, который возвращает тест по одному значению:

C#

```
public delegate bool Predicate<in T>(T obj);
```

Вы можете заметить, что для любого `Predicate` типа существует структурны эквивалентный `Func` тип, например:

C#

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Вы можете подумать, что эти два типа эквивалентны. Они нет. Эти две переменные нельзя использовать взаимозаменяемо. Переменная одного типа не может быть назначена другому типу. Система типов C# использует имена определенных типов, а не структуру.

Все эти определения типов делегатов в библиотеке .NET Core должны означать, что вам не нужно определять новый тип делегата для любой новой функции, которая требует делегатов. Эти универсальные определения должны предоставлять все необходимые типы делегатов в большинстве случаев. Можно просто создать экземпляр одного из этих

типов с необходимыми параметрами типа. В случае алгоритмов, которые можно сделать универсальными, эти делегаты можно использовать в качестве универсальных типов.

Это должно сэкономить время и свести к минимуму количество новых типов, которые необходимо создать для работы с делегатами.

В следующей статье вы увидите несколько распространенных шаблонов для работы с делегатами на практике.

Далее

Общие шаблоны делегатов

17.06.2025

Предыдущий

Делегаты предоставляют механизм, обеспечивающий разработку программного обеспечения с минимальным взаимодействием между компонентами.

Одним из отличных примеров для такого вида дизайна является LINQ. Шаблон выражения запроса LINQ зависит от делегатов для всех его функций. Рассмотрим этот простой пример:

```
C#

var smallNumbers = numbers.Where(n => n < 10);
```

Это фильтрует последовательность чисел только тем, которые меньше значения 10. Метод `Where` использует делегат, определяющий, какие элементы последовательности передают фильтр. При создании запроса LINQ вы предоставляете реализацию делегата, специально предназначенного для этой цели.

Прототип метода Where:

```
C#

public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource>
source, Func<TSource, bool> predicate);
```

Этот пример повторяется со всеми методами, которые входят в состав LINQ. Все они полагаются на делегатов для кода, управляющего определённым запросом. Этот шаблон разработки API — мощный шаблон, который стоит изучить и понять.

В этом простом примере показано, как делегаты требуют очень мало взаимодействия между компонентами. Вам не нужно создавать класс, производный от определенного базового класса. Вам не нужно реализовать определенный интерфейс. Единственным требованием является предоставление реализации одного метода, который является фундаментальным для задачи.

Постройте свои собственные компоненты с помощью делегатов

Давайте продолжим развитие этого примера, создав компонент, используя дизайн, который зависит от делегатов.

Определим компонент, который можно использовать для сообщений журнала в большой системе. Компоненты библиотеки можно использовать во многих разных средах на нескольких разных платформах. В компоненте, который управляет журналами, существует множество распространенных функций. Он должен принимать сообщения из любого компонента в системе. Эти сообщения будут иметь разные приоритеты, которым может управлять основной компонент. Сообщения должны иметь метки времени в окончательной архивной форме. Для более сложных сценариев можно фильтровать сообщения по исходному компоненту.

Существует один аспект функции, которая часто меняется: где записываются сообщения. В некоторых средах они могут быть записаны в консоль ошибок. В других - файл. Другие возможности включают хранилище баз данных, журналы событий ОС или другое хранилище документов.

Существуют также сочетания выходных данных, которые могут использоваться в разных сценариях. Может потребоваться записать сообщения в консоль и в файл.

Проектирование, основанное на делегатах, обеспечит значительную гибкость и облегчит поддержку механизмов хранения, которые могут быть добавлены в будущем.

В этой структуре основной компонент журнала может быть не виртуальным, даже закрытым классом. Вы можете подключить любой набор делегатов для записи сообщений в различные средства хранения. Встроенная поддержка мультикастовых делегатов облегчает создание сценариев, в которых сообщения должны выводиться в несколько точек (например, в файл и консоль).

Первая реализация

Давайте начнем с малого: начальная реализация будет принимать новые сообщения и записывать их с помощью любого присоединенного делегата. Вы можете начать с одного делегата, который записывает сообщения в консоль.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            WriteMessage(msg);
```

```
    }
}
```

Приведенный выше статический класс — это самая простая вещь, которая может работать. Нам нужно написать единую реализацию для метода, который записывает сообщения в консоль:

C#

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Наконец, необходимо подключить делегат к делегату WriteMessage, объявленному в логгере.

C#

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Практики

Наш пример до сих пор довольно прост, но он по-прежнему демонстрирует некоторые важные рекомендации по проектированию с участием делегатов.

Использование типов делегатов, определенных в основной платформе, упрощает работу пользователей с делегатами. Вам не нужно определять новые типы, а разработчикам, использующим библиотеку, не нужно изучать новые специализированные типы делегатов.

Используемые интерфейсы являются максимально минимальными и гибкими: чтобы создать новый средство ведения журнала выходных данных, необходимо создать один метод. Этот метод может быть статическим методом или методом экземпляра. У него может быть любой уровень доступа.

Формат выходных данных

Давайте сделаем эту первую версию более надежной, а затем начнем создавать другие механизмы ведения журнала.

Далее давайте добавим несколько аргументов в `LogMessage()` метод, чтобы класс журнала создал более структурированные сообщения:

C#

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Затем давайте используем этот `Severity` аргумент для фильтрации сообщений, отправляемых в выходные данные журнала.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
```

```
    }
}
```

Практики

Вы добавили новые функции в инфраструктуру логирования. Так как компонент средства ведения журнала очень слабо связан с любым механизмом вывода, эти новые функции можно добавлять без влияния на любой код, реализующего делегат средства ведения журнала.

По мере продолжения разработки вы увидите больше примеров того, как это свободное связывание обеспечивает большую гибкость при обновлении частей сайта без каких-либо изменений в других местах. В самом деле, в более крупном приложении классы вывода логгера могут находиться в другой сборке, и даже не потребуется повторная сборка.

Создание второго выходного движка

Компонент журнала успешно развивается. Давайте добавим еще один модуль вывода, который записывает сообщения в файл. Это будет немного более сложная выходная система. Это будет класс, который инкапсулирует операции с файлами и гарантирует, что файл всегда закрыт после каждой записи. Это гарантирует, что все данные сбрасываются на диск после создания каждого сообщения.

Вот средство ведения журнала на основе файлов:

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
```

```
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

После создания этого класса вы можете создать его экземпляр и привязать его метод LogMessage к компоненту Logger.

C#

```
var file = new FileLogger("log.txt");
```

Эти два не являются взаимоисключающими. Можно подключить как методы журнала, так и создать сообщения в консоль и файл:

C#

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the
static class we utilized earlier
```

Позже, даже в том же приложении, вы можете удалить один из делегатов без каких-либо других проблем в системе:

C#

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Практики

Теперь вы добавили второй обработчик выходных данных для подсистемы ведения журнала. Для правильной поддержки файловой системы требуется немного больше инфраструктуры. Делегат — это метод экземпляра. Это также частный метод. Нет необходимости в повышенной доступности, так как инфраструктура делегатов может обеспечивать связь между делегатами.

Во-вторых, конструктор на основе делегатов позволяет использовать несколько методов вывода без дополнительного кода. Вам не нужно создавать дополнительную инфраструктуру для поддержки нескольких методов вывода. Они просто становятся очередным методом в списке вызовов.

Обратите особое внимание на код в методе вывода журнала файлов. Кодируется, чтобы гарантировать, что он не создает никаких исключений. Хотя это не всегда строго необходимо, это часто хорошая практика. Если любой из методов делегата создает исключение, остальные делегаты, которые находятся в вызове, не будут вызваны.

В качестве последней заметки средство ведения журнала файлов должно управлять своими ресурсами, открывая и закрывая файл в каждом сообщении журнала. Вы можете сохранить файл открытым и реализовать `IDisposable` , чтобы закрыть файл после завершения. Любой метод имеет свои преимущества и недостатки. Оба создают немного больше связей между классами.

Для поддержки любого сценария не потребуется обновить код в `Logger` классе.

Обработка делегатов со значением null

Наконец, давайте обновим метод LogMessage таким образом, чтобы он был надежным для тех случаев, когда выходной механизм не выбран. Текущая реализация вызывает `NullReferenceException`, когда делегат `WriteMessage` не имеет присоединенного списка вызовов. Вы можете предпочесть дизайн, который незаметно продолжается, если методы не были подключены. Это легко использовать условный оператор NULL, в сочетании с методом `Delegate.Invoke()` :

C#

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

Оператор условного нуля (`?.`) выполняет короткое замыкание, когда левый операнд (`WriteMessage` в данном случае) имеет значение NULL, что означает, что попытка записать сообщение не предпринимается.

Вы не найдете метод `Invoke()`, перечисленный в документации для `System.Delegate` или `System.MulticastDelegate`. Компилятор создает безопасный `Invoke` метод типа для любого объявленного типа делегата. В этом примере это означает, что `Invoke` принимает один аргумент `string` и имеет тип возвращаемого значения void.

Сводка методик

Вы видели начало компонента журнала, который можно расширить с помощью других средств записи и других функций. Используя делегаты при разработке, эти различные компоненты слабо связаны между собой. Это дает несколько преимуществ. Легко создать новые механизмы вывода и подключить их к системе. Эти другие механизмы требуют только одного метода: метод, который записывает сообщение журнала. Это дизайн, который устойчив при добавлении новых функций. Требуемый контракт для любого автора заключается в реализации одного метода. Этот метод может быть статическим или экземплярным методом. Это может быть общедоступный, частный или любой другой юридический доступ.

Класс Logger может вносить любое количество улучшений или изменений без внесения критических изменений. Как и любой класс программирования, нельзя изменять общедоступный API без риска изменений, которые могут привести к поломкам. Но, так как связь между средством ведения журнала и любыми обработчиками выходных данных осуществляется только через делегат, другие типы (например, интерфейсы или базовые классы) не участвуют. Муфта настолько мала, насколько это возможно.

Далее

Общие сведения о событиях

Статья • 21.03.2025

Предыдущий

События, как и делегаты, представляют собой механизм *поздней привязки*. На самом деле события основаны на поддержке языка делегатов.

События — это способ уведомления объекта всем заинтересованным компонентам в системе о том, что что-то произошло. Любой другой компонент может подписаться на событие и получать уведомления при возникновении события.

Возможно, вы использовали события в некоторых программах. Многие графические системы имеют модель событий для отчета о взаимодействии с пользователем. Эти события будут сообщать о перемещении мыши, нажатия кнопки и аналогичных взаимодействиях. Это один из наиболее распространенных, но не единственный сценарий, в котором используются события.

Вы можете определить события, которые должны быть вызваны для ваших классов. Одним из важных соображений при работе с событиями является то, что для определенного события может не быть зарегистрирован какой-либо объект. Необходимо написать код, чтобы он не вызывал события, если прослушиватели не настроены.

Подписка на событие также создает связь между двумя объектами (источником событий и приемником событий). Необходимо убедиться, что приемник событий отменяет подписку с источника событий, когда в событиях больше нет заинтересованности.

Цели разработки для поддержки событий

Разработка языка для событий нацелена на следующие цели:

- Включите минимальное взаимодействие между источником событий и приемником событий. Эти два компонента могут быть написаны различными организациями и даже могут быть обновлены по разным расписаниям.
- Подписаться на событие и отписаться от него должно быть просто.
- Источники событий должны поддерживать несколько подписчиков событий. Оно также должно поддерживать наличие отсутствия подключенных подписчиков событий.

Вы можете увидеть, что цели у мероприятий аналогичны целям делегатов. Поэтому поддержка языка событий основана на поддержке языка делегатов.

Поддержка языков для событий

Синтаксис для определения событий и подписки или отмены подписки на события является расширением синтаксиса для делегатов.

Ключевое слово `event` используется для определения события:

```
C#

public event EventHandler<FileFoundArgs>? FileFound;
```

Тип события (`EventHandler<FileListArgs>` в этом примере) должен быть делегатным типом. Существуют соглашения, которые следует следовать при объявлении события. Как правило, тип делегата события имеет возвращаемое значение void. Названия событий должны быть глаголом или глагольной фразой. Используйте прошлое время, когда событие сообщает о том, что произошло. Используйте глагол в настоящем времени (например, `Closing`), чтобы сообщить о том, что вот-вот произойдет. Часто использование настоящего времени указывает на то, что ваш класс поддерживает определенное поведение настройки. Одним из наиболее распространенных сценариев является поддержка отмены. Например, событие `Closing` может включать аргумент, указывающий, следует ли продолжить операцию закрытия или нет. Некоторые сценарии позволяют вызывающим объектам изменять поведение путем обновления свойств аргументов события. Вы можете вызвать событие, чтобы указать предлагаемое следующее действие, которое будет предпринять алгоритм. Обработчик событий может потребовать другого действия, изменив свойства аргумента события.

Если вы хотите вызвать событие, вызовите обработчики событий с помощью синтаксиса вызова делегата:

```
C#

FileFound?.Invoke(this, new FileFoundArgs(file));
```

Как обсуждалось в разделе о делегатах , оператор `?.` упрощает обеспечение того, что вы не попытаетесь вызвать событие, когда нет подписчиков на это событие.

Вы подписываетесь на событие, используя оператор `+=`:

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

Метод обработчика обычно имеет префикс On, за которым следует имя события, как показано в предыдущем коде.

Отмена подписки с помощью оператора `-=`:

C#

```
fileLister.FileFound -= onFileFound;
```

Важно объявить локальную переменную для выражения, представляющего обработчик событий. Это гарантирует, что отмена подписки приводит к удалению обработчика. Если вместо этого вы использовали тело лямбда-выражения, вы пытаетесь удалить обработчик, который никогда не был присоединен, поэтому это не оказывает никакого эффекта.

В следующей статье вы узнаете больше о типичных шаблонах событий и различных вариантах в этом примере.

следующее

Стандартные шаблоны событий .NET

Статья • 25.03.2025

Предыдущий

События .NET обычно следуют нескольким известным шаблонам. Стандартизация этих шаблонов означает, что разработчики могут применять знания об этих стандартных шаблонах, которые можно применить к любой программе событий .NET.

Давайте рассмотрим эти стандартные шаблоны, чтобы у вас были все знания, необходимые для создания стандартных источников событий, а также подписки и обработки стандартных событий в коде.

Подписи делегатов событий

Стандартной сигнатурой делегата события .NET является:

```
C#

void EventRaised(object sender, EventArgs args);
```

Эта стандартная сигнатура предоставляет представление о том, когда используются события:

- ***Тип возвращаемого значения — void***. События могут иметь от нуля до многих слушателей. Вызов события уведомляет всех прослушивателей. Как правило, слушатели не предоставляют значения в ответ на события.
- ***События указывают отправителя***: подпись события включает объект, вызвавшее событие. Это обеспечивает любой прослушиватель механизмом для взаимодействия с отправителем. Тип времени компиляции `sender` — `System.Object`, хотя вы, вероятно, знаете более специфичный тип, который всегда будет правильным. По соглашению используйте `object`.
- ***пакет дополнительных сведений в одной структуре***: параметр `args` является типом, производным от System.EventArgs, который содержит дополнительные необходимые сведения. (Вы увидите в следующем разделе , что это соглашение больше не применяется.) Если тип вашего события не требует дополнительных аргументов, все равно необходимо предоставить оба аргумента. Существует специальное значение, EventArgs.Empty, которое

следует использовать для обозначения того, что событие не содержит дополнительных сведений.

Создадим класс, который перечисляет соответствующие шаблону файлы в каталоге или любом из его подкаталогов. Этот компонент создает событие для каждого найденного файла, который соответствует шаблону.

Использование модели событий обеспечивает некоторые преимущества разработки. Можно создать несколько прослушивателей событий, которые выполняют разные действия при нахождении искомого файла. Сочетание разных прослушивателей позволяет создавать более надежные алгоритмы.

Ниже приведено начальное объявление аргумента события для поиска запрошенного файла:

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Несмотря на то, что этот тип выглядит как небольшой тип, содержащий только данные, вы должны выполнить соглашение и назначить его ссылочным типом (`class`). Это означает, что объект аргумента передается по ссылке, и все обновления данных просматриваются всеми подписчиками. Первая версия является неизменяемым объектом. Рекомендуется сделать свойства в типе аргумента события неизменяемыми. Таким образом, один подписчик не может изменить значения, прежде чем другой подписчик увидит их. (Существуют исключения для этой практики, как вы видите позже.)

В следующем шаге нужно создать объявление события в классе FileSearcher. Использование типа System.EventHandler<TEventArgs> означает, что вам не нужно создавать еще одно определение типа. Вы просто используете универсальную специализацию.

Заполним класс FileSearcher для поиска файлов, соответствующих шаблону, и вызова правильного события при обнаружении совпадения.

C#

```
public class FileSearcher
{
```

```
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Определение и вызов событий, таких как поле

Самый простой способ добавить событие в класс — объявить это событие как открытое поле, как показано в предыдущем примере.

C#

```
public event EventHandler<FileFoundArgs>? FileFound;
```

Это похоже на объявление общедоступного поля, которое, как представляется, является плохой объектно-ориентированной практикой. поскольку необходимо обеспечить защиту доступа к данным с помощью свойств и методов. Хотя этот код может выглядеть как плохая практика, код, созданный компилятором, создает оболочки, чтобы объекты событий могли быть доступны только безопасным способом. Единственной операцией, доступной для события, подобного полю, является *добавление обработчика*.

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

И удалить обработчик:

```
```

```
C#
```

```
fileLister.FileFound -= onFileFound;
```

Для обработчика существует локальная переменная. Если вы использовали тело лямбда, удаление не сработает правильно. Будет существовать другой экземпляр делегата, не выполняющий никаких действий.

Код за пределами класса не может вызвать событие и не может выполнять другие операции.

Возврат значений от подписчиков событий

Простая версия работает нормально. Давайте добавим еще одну возможность — отмену.

При вызове события *Found* прослушиватели должны прекратить дальнейшую обработку, если этот файл является последним искомым.

Обработчики событий не возвращают значение, поэтому необходимо сообщить об этом другим способом. Стандартный шаблон событий `EventArgs` использует объект для включения полей, которые подписчики событий могут использовать для передачи информации об отмене.

Можно использовать два разных шаблона на основе семантики договора об отмене. В обоих случаях вы добавляете логическое поле в EventArguments для события обнаруженного файла.

Один шаблон позволяет любому одному подписчику отменить операцию. Для этого шаблона новое поле инициализируется значением `false`. Любой подписчик может изменить это на `true`. После вызова события для всех подписчиков компонент FileSearcher проверяет логическое значение и принимает меры.

Второй шаблон отменит операцию только в том случае, если все подписчики хотели отменить операцию. В этом шаблоне новое поле инициализируется для указания того, что операцию следует отменить, и любой подписчик может изменить его, чтобы указать, что следует продолжить операцию. После того как все подписчики обработали вызванное событие, компонент FileSearcher проверяет логическое значение и принимает меры. В этом шаблоне есть еще один дополнительный шаг: компонент должен знать, ответил ли любой подписчик на событие. Если подписчики отсутствуют, поле неверно сообщит об отмене.

Реализуем первую версию для этого примера. Добавьте логическое поле с именем `CancelRequested` в тип `FileFoundArgs`:

```C#
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Это новое поле автоматически инициализируется в `false`, чтобы не отменить по ошибке. Единственное другое изменение компонента заключается в проверке флага после вызова события, чтобы узнать, запрашивает ли любой из подписчиков отмену:

```C#
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

Одно из преимуществ этого шаблона заключается в том, что это не является критическим изменением. Никто из подписчиков не запрашивал отмену раньше, и они до сих пор этого не сделали. Ни один из кодов подписчика не требует обновлений, если они не хотят поддерживать новый протокол отмены.

Изменим подписчика, чтобы он инициировал отмену, как только обнаружит первый исполняемый файл.

```C#
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Добавление другой декларации события

Добавим еще одну функцию и продемонстрируем языковые идиомы для описания событий. Добавим перегрузку метода `Search`, который проходит через все подкаталоги в поиске файлов.

Этот метод может быть длительной операцией в каталоге с множеством подкаталогов. Добавим событие, которое вызывается в начале каждого нового поиска в каталоге. Это событие позволяет подписчикам отслеживать ход выполнения и информировать пользователя о ходе выполнения. Все созданные вами примеры являются общедоступными. Давайте сделаем это событие внутренним событием. Это означает, что также можно сделать типы аргументов внутренними.

Начните с создания нового производного класса EventArgs для создания отчетов о новом каталоге и ходе выполнения.

C#

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Опять же, вы можете следовать рекомендациям по созданию неизменяемого ссылочного типа для аргументов событий.

Теперь определим событие. На этот раз используется другой синтаксис. Помимо использования синтаксиса поля, вы можете явно создать свойство события с помощью обработчиков добавления и удаления. В этом примере не требуется дополнительный код в этих обработчиках, но в этом примере показано, как создать их.

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

Во многих отношениях код, который вы пишете здесь, отражает код, который компилятор создает для определений событий поля, которые вы видели ранее. Событие создается с помощью синтаксиса, аналогичного свойствам . Обратите внимание, что обработчики имеют разные имена: `add` и `remove`. Эти акцессоры используются для подписки на событие или отмены подписки на событие. Учтите, что вы также должны объявить закрытое резервное поле для хранения переменной событий. Эта переменная инициализирована в null.

Теперь добавим перегрузку метода `Search`, который обходит подкаталоги и вызывает оба события. Самый простой способ — использовать аргумент по умолчанию, чтобы указать, что вы хотите искать все каталоги:

C#

```
public void Search(string directory, string searchPattern, bool
searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*",
SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            _directoryChanged?.Invoke(this, new (dir, totalDirs,
completedDirs++));
            // Search 'dir' and its subdirectories for files that match the
search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        _directoryChanged?.Invoke(this, new (directory, totalDirs,
completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}
```

```
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

На этом этапе можно запустить приложение, вызывающее перегрузку для поиска всех подкаталогов. Нет подписчиков на новом событии `DirectoryChanged`, но использование идиомы `?.Invoke()` гарантирует правильную работу.

Добавим обработчик для написания строки, показывающей ход выполнения в окне консоли.

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
};
```

Вы видели шаблоны, которые используются во всей экосистеме .NET. Изучая эти шаблоны и соглашения, вы быстро пишете идиоматические C# и .NET.

См. также

- Общие сведения о событиях
- Проектирование событий
- Обработка и генерация событий

Далее вы увидите некоторые изменения в этих шаблонах в последнем выпуске .NET.

Далее

Обновленный шаблон событий .NET Core

Статья • 25.03.2025

предыдущий

В предыдущей статье рассматриваются наиболее распространенные шаблоны событий. .NET Core имеет более расслабленный шаблон. В этой версии определение `EventHandler<TEventArgs>` больше не имеет ограничения, что `TEventArgs` должен быть классом, производным от `System.EventArgs`.

Это повышает гибкость для вас и обеспечивает обратную совместимость. Начнем с гибкости. Реализация для System.EventArgs использует метод, определенный в System.Object: MemberwiseClone(), который создает поверхностную копию объекта. Этот метод должен использовать отражение для реализации функций любого класса, производного от `EventArgs`. Эта функция проще создавать в определенном производном классе. Это фактически означает, что наследование от System.EventArgs является ограничением, которое сужает возможности ваших разработок, но при этом не дает никаких дополнительных преимуществ. На самом деле можно изменить определения `FileFoundArgs` и `SearchDirectoryArgs`, чтобы они не были производными от `EventArgs`. Программа работает точно так же.

Вы также можете изменить `SearchDirectoryArgs` на структуру, если внести еще одно изменение:

C#

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Дополнительное изменение заключается в вызове конструктора без параметров перед вводом конструктора, который инициализирует все поля. Без этого дополнения правила C# указывают на то, что обращение к свойствам происходит до их присвоения.

Не следует изменять `FileFoundArgs` из класса (ссылочного типа) на структуру (тип значения). Протокол для обработки отмены требует передачи аргументов событий по ссылке. Если вы внесли то же изменение, класс поиска файлов никогда не мог наблюдать никаких изменений, внесенных любым из подписчиков событий. Новая копия структуры будет использоваться для каждого подписчика, и эта копия будет отличаться от копии, которую видел объект поиска файлов.

Далее рассмотрим, как это изменение может быть обратно совместимым. Удаление ограничения не влияет на существующий код. Все существующие типы аргументов событий по-прежнему являются производными от `System.EventArgs`. Обратная совместимость является одной из основных причин, почему они продолжают использовать `System.EventArgs`. Все существующие подписчики событий являются подписчиками на событие, которое следует классическому образцу.

После аналогичной логики любой тип аргумента события, созданный сейчас, не будет иметь подписчиков в существующих базах кода. Новые типы событий, не производные от `System.EventArgs`, не нарушают эти базы кода.

События с подписчиками Async

Последний шаблон, который вам нужно изучить: как правильно писать подписчиков событий, которые вызывают асинхронный код. Задача описана в статье об асинхронном асинхронном и ожидании. Асинхронные методы могут иметь тип возвращаемого значения void, но этого следует избегать. Если код подписчика события вызывает асинхронный метод, у вас нет выбора, кроме создания метода `async void`. Требуется подпись обработчика событий.

Вам нужно примирить эти противоречивые указания. Как-то необходимо создать безопасный метод `async void`. Основы шаблона, который необходимо реализовать, показаны в следующем коде:

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
```

```
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

Сначала обратите внимание, что обработчик помечается как асинхронный обработчик. Поскольку он назначается делегату типа обработчика событий, он имеет возвращаемое значение типа void. Это означает, что необходимо выполнить шаблон, показанный в обработчике, и не разрешать исключение из контекста асинхронного обработчика. Так как он не возвращает задачу, нет задачи, которая может сообщить об ошибке, войдя в состояние сбоя. Так как метод является асинхронным, метод не может вызвать исключение. (Вызывающий метод продолжает выполнение, так как он `async`.) Фактическое поведение среды выполнения определяется по-разному для разных сред. Он может завершить поток или процесс, принадлежащий потоку, или оставить процесс в неопределенном состоянии. Все эти потенциальные результаты являются весьма нежелательными.

Необходимо упаковать выражение `await` для асинхронной задачи в собственном блоке try. Если это приведет к сбою задачи, можно записать ошибку. Если это ошибка, из которой приложение не может восстановиться, вы можете быстро и корректно выйти из программы.

В этой статье описаны основные обновления шаблона событий .NET. Вы можете увидеть множество примеров предыдущих версий в библиотеках, с которыми вы работаете. Тем не менее, вы должны понять, какие сейчас последние тенденции. Готовый код можно просмотреть в нашем репозитории примеров на сайте GitHub.

В следующей статье этой серии вы узнаете, как использовать `delegates` и `events` в проектах. Они похожие понятия, и эта статья помогает вам принять лучшее решение для ваших программ.

следующее

Различение делегатов и событий

Статья • 22.03.2025

предыдущий

Разработчики, которые являются новыми для платформы .NET, часто борются при выборе между проектом на основе `delegates` и дизайна на основе `events`. Выбор делегатов или событий часто затруднен, так как два языковых компонента похожи. События даже создаются с помощью поддержки языка для делегатов. Объявление обработчика событий объявляет тип делегата.

Оба предлагают сценарий последней привязки: они позволяют использовать сценарии, в которых компонент взаимодействует, вызывая метод, известный только во время выполнения. Они поддерживают как одиночные, так и множественные методы подписки. Эти термины могут называться поддержкой одноадресной рассылки и многоадресной рассылки. Они оба поддерживают аналогичный синтаксис для добавления и удаления обработчиков. Наконец, вызов события и вызов делегата используют точно тот же синтаксис вызова метода. Они даже поддерживают один и тот же синтаксис метода `Invoke()` для использования с оператором `?.`.

При всех этих сходствах легко запутаться в том, когда какое из них использовать.

Прослушивание событий является необязательным

Самым важным соображением при выборе языковой функции является необходимость наличия подключенного подписчика. Если код должен вызывать код, предоставленный подписчиком, следует использовать дизайн на основе делегатов, когда необходимо реализовать обратный вызов. Если код может завершить всю работу без вызова подписчиков, следует использовать дизайн на основе событий.

Рассмотрим примеры, созданные в этом разделе. Код, созданный с помощью `List.Sort()`, должен быть предоставлен функцию сравнения для правильной сортировки элементов. Запросы LINQ должны предоставляться делегатам, чтобы определить, какие элементы следует возвращать. Оба использовали дизайн, созданный с помощью делегатов.

Рассмотрим событие `Progress`. Он сообщает о ходе выполнения задачи. Задача продолжается независимо от того, есть ли слушатели. `FileSearcher` является еще одним примером. Он всё равно будет искать и находить все файлы, которые были запрошены, даже без подключённых подписчиков событий. Элементы управления UX по-прежнему работают правильно, даже если подписчики не прослушивают события. Они оба используют дизайны на основе событий.

Возвращаемые значения требуют делегатов

Другим фактором является прототип метода, который требуется для метода делегата. Как вы видели, делегаты, используемые для событий, имеют тип возвращаемых данных void. Существуют идиомы для создания обработчиков событий, которые передают информацию обратно в источники событий путем изменения свойств объекта аргумента события. Хотя эти идиомы работают, они не так естественны, как возврат значения из метода.

Обратите внимание, что эти два эвристика часто могут присутствовать: если метод делегата возвращает значение, он влияет на алгоритм каким-то образом.

События имеют приватный вызов

Классы, кроме того класса, в котором находится событие, могут только добавлять и удалять прослушиватели событий; только класс, содержащий событие, может его вызвать. События обычно являются элементами общедоступного класса. В отличие от этого, делегаты часто передаются в качестве параметров и сохраняются как закрытые члены класса, если они вообще хранятся.

Слушатели событий часто имеют более длительный срок существования

Более длительное время существования прослушивателей событий является немного слабым обоснованием. Однако вам может показаться, что разработки на основе событий более естественны, когда источник событий генерирует события в течение длительного периода времени. Примеры проектирования на основе событий для элементов управления пользовательским интерфейсом можно увидеть во многих системах. После подписки на событие источник событий может вызывать события в течение всего времени существования программы. (Вы можете отменить подписку на события, если они больше не нужны.)

В отличие от многих проектов, основанных на делегатах, где делегат используется как аргумент метода, но перестает использоваться после его завершения.

Тщательно оцените

Приведенные выше рекомендации не являются сложными и быстрыми правилами. Вместо этого они представляют рекомендации, которые помогут вам решить, какой выбор лучше всего подходит для конкретного использования. Поскольку они похожи, вы можете даже создать прототипы для обоих и подумать, с каким из них будет более естественно работать. Оба они хорошо обрабатывают сценарии позднего связывания. Используйте тот, который лучше всего передает вашу дизайнерскую идею.

Управление версиями в C#

09.07.2025

В этом учебнике вы получите представление об управлении версиями в .NET. Кроме того, вы узнаете, какие факторы следует учитывать при управлении версиями библиотеки и обновлении до ее новой версии.

Версия языка

Компилятор C# является частью пакета SDK для .NET. По умолчанию компилятор выбирает версию языка C#, которая соответствует выбранному TFM для проекта. Если версия пакета SDK больше выбранной платформы, компилятор может использовать более высокую языковую версию. Вы можете изменить значение по умолчанию, задав `LangVersion` элемент в проекте. Вы можете узнать, как описано в нашей статье о параметрах компилятора.

> ⚠ **Предупреждение**
>
> `LangVersion` Настройка элемента `latest` не рекомендуется. Параметр означает, `latest` что установленный компилятор использует последнюю версию. Это может измениться с компьютера на компьютер, что делает сборки ненадежными. Кроме того, он включает функции языка, для которых могут потребоваться функции среды выполнения или библиотеки, не включенные в текущий пакет SDK.

Разработка библиотек

Если вы занимались разработкой библиотек .NET для открытого использования, то, скорее всего, вам приходилось развертывать обновления. Способ выполнения этой процедуры важен, так как необходимо обеспечить плавный перевод существующего кода на новую версию библиотеки. Ниже описывается ряд моментов, которые следует учитывать при создании выпуска.

Семантическое управление версиями

Семантическое версионирование (SemVer) — это соглашение об именовании, применяемое к версиям библиотеки для обозначения ключевых вех. В идеале сведения о версии, включаемые в библиотеку, должны помочь разработчикам определить

совместимость с проектами, в которых используются предыдущие версии этой библиотеки.

Базовым вариантом SemVer является трехкомпонентный формат `MAJOR.MINOR.PATCH`, где:

- `MAJOR` увеличивается при внесении изменений, приводящих к несовместимости API;
- `MINOR` увеличивается при добавлении функциональных возможностей с обеспечением обратной совместимости;
- `PATCH` увеличивается при исправлении ошибок с обеспечением обратной совместимости.

Общие сведения о добавочных версиях с примерами

Чтобы уточнить, когда следует увеличить каждый номер версии, ниже приведены конкретные примеры:

Увеличение ОСНОВНЫХ версий (несовместимые изменения API)

Эти изменения требуют от пользователей изменения кода для работы с новой версией:

- Удаление общедоступного метода или свойства:

```
C#

// Version 1.0.0
public class Calculator
{
    public int Add(int a, int b) => a + b;
    public int Subtract(int a, int b) => a - b; // This method exists
}

// Version 2.0.0 - MAJOR increment required
public class Calculator
{
    public int Add(int a, int b) => a + b;
    // Subtract method removed - breaking change!
}
```

- Изменение сигнатур методов:

```
C#

// Version 1.0.0
public void SaveFile(string filename) { }
```

```
// Version 2.0.0 - MAJOR increment required
public void SaveFile(string filename, bool overwrite) { } // Added required
parameter
```

- Изменение поведения существующих методов таким образом, чтобы сломать ожидания:

```
C#

// Version 1.0.0 - returns null when file not found
public string ReadFile(string path) => File.Exists(path) ?
File.ReadAllText(path) : null;

// Version 2.0.0 - MAJOR increment required
public string ReadFile(string path) => File.ReadAllText(path); // Now throws
exception when file not found
```

Увеличение минорных версий (обратная совместимость функциональности)

Эти изменения добавляют новые функции без нарушения существующего кода:

- Добавление новых общедоступных методов или свойств:

```
C#

// Version 1.0.0
public class Calculator
{
    public int Add(int a, int b) => a + b;
}

// Version 1.1.0 - MINOR increment
public class Calculator
{
    public int Add(int a, int b) => a + b;
    public int Multiply(int a, int b) => a * b; // New method added
}
```

- Добавление новых перегрузок:

```
C#

// Version 1.0.0
public void Log(string message) { }

// Version 1.1.0 - MINOR increment
```

```
public void Log(string message) { } // Original method unchanged
public void Log(string message, LogLevel level) { } // New overload added
```

- Добавление необязательных параметров в существующие методы:

```
C#

// Version 1.0.0
public void SaveFile(string filename) { }

// Version 1.1.0 - MINOR increment
public void SaveFile(string filename, bool overwrite = false) { } // Optional
parameter
```

> ⓘ **Примечание**
>
> Это *исходное совместимое изменение*, но *двоичное критическое изменение*. Пользователи этой библиотеки должны перекомпилировать ее для правильной работы. Многие библиотеки рассматривают это только при *основных* изменениях версии, а не при изменениях *незначительных* версий.

Увеличение версии патча (исправления ошибок с поддержанием обратной совместимости)

Эти изменения устраняют проблемы без добавления новых функций или нарушения существующих функциональных возможностей:

- Исправление ошибки в реализации существующего метода:

```
C#

// Version 1.0.0 - has a bug
public int Divide(int a, int b)
{
    return a / b; // Bug: doesn't handle division by zero
}

// Version 1.0.1 - PATCH increment
public int Divide(int a, int b)
{
    if (b == 0) throw new ArgumentException("Cannot divide by zero");
    return a / b; // Bug fixed, behavior improved but API unchanged
}
```

- Улучшения производительности, которые не изменяют API:

C#

```
// Version 1.0.0
public List<int> SortNumbers(List<int> numbers)
{
    return numbers.OrderBy(x => x).ToList(); // Slower implementation
}

// Version 1.0.1 - PATCH increment
public List<int> SortNumbers(List<int> numbers)
{
    var result = new List<int>(numbers);
    result.Sort(); // Faster implementation, same API
    return result;
}
```

Основным принципом является: если существующий код может использовать новую версию без каких-либо изменений, это обновление MINOR или PATCH. Если существующий код необходимо изменить для работы с новой версией, это основное обновление.

Существуют также способы указания других сценариев, например предварительной версии, при применении сведений о версиях к библиотеке .NET.

Обратная совместимость

При выпуске новых версий библиотеки обеспечение обратной совместимости с предыдущими версиями, скорее всего, будет одной из основных задач. Новая версия библиотеки является совместимой с предыдущей на уровне исходного кода, если код, который зависит от предыдущей версии, после перекомпиляции может работать с новой версией. Новая версия библиотеки является совместимой на уровне двоичного кода, если приложение, которое зависело от предыдущей версии, может работать с новой версией без перекомпиляции.

Ниже описывается ряд моментов, которые следует учитывать при обеспечении обратной совместимости с предыдущими версиями библиотеки.

- Виртуальные методы: когда вы делаете виртуальный метод невиртуальным в новой версии, это означает, что проекты, в которых этот метод переопределяется, необходимо будет обновить. Это очень существенное изменение, которое настоятельно не рекомендуется производить.
- Сигнатуры методов. При обновлении поведения метода необходимо также изменить ее сигнатуру, вместо этого следует создать перегрузку, чтобы вызов кода в этот метод по-прежнему работал. Прежнюю сигнатуру метода всегда можно

настроить так, чтобы она вызывала новую сигнатуру, обеспечив тем самым согласованность реализаций.

- Атрибут obsolete: с помощью этого атрибута можно указывать в коде классы или члены классов, которые являются нерекомендуемыми и, вероятно, будут удалены в будущих версиях. Благодаря этому разработчики, использующие вашу библиотеку, будут лучше подготовлены к существенным изменениям.
- Необязательные аргументы метода: если вы делаете необязательные аргументы метода обязательными или меняете их значения по умолчанию, то код, который не задает эти аргументы, потребуется обновить.

> ⓘ **Примечание**
>
> Если обязательные аргументы становятся необязательными, то, как правило, это не должно иметь особых последствий, особенно если при этом поведение метода не меняется.

Чем проще пользователям будет произвести обновление до новой версии библиотеки, тем скорее они это сделают.

Файл конфигурации приложения

Как разработчик .NET вы, весьма вероятно, встречали файл app.config в большинстве типов проектов. Этот простой файл конфигурации может очень упростить развертывание обновлений. В основном библиотеки следует проектировать так, чтобы сведения, которые, скорее всего, будут меняться регулярно, хранились в файле `app.config`. Благодаря этому при изменении таких сведений достаточно будет заменить файл конфигурации предыдущей версии на новый, не перекомпилируя библиотеку.

Использование библиотек

Если при разработке вы используете библиотеки .NET, созданные другими разработчиками, то, скорее всего, знаете, что новая версия библиотеки может быть не полностью совместима с вашим проектом. В таком случае приходится обновлять код в соответствии с изменениями в библиотеке.

К счастью, экосистема C# и .NET предоставляет возможности и методы, которые позволяют обновлять приложения для работы с новыми версиями библиотек, в которые были внесены существенные изменения.

Перенаправление привязки сборок

С помощью файла *app.config* можно изменить версию библиотеки, используемую приложением. Добавив так называемое *перенаправление привязки*, вы можете использовать новую версию библиотеки, не перекомпилируя приложение. В приведенном ниже примере показано, как можно изменить файл приложения *app.config* так, чтобы оно использовало версию исправления `1.0.1` библиотеки `ReferencedLibrary` вместо версии `1.0.0`, с которой оно изначально компилировалось.

XML

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1"
culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **Примечание**
>
> Такой подход будет работать только в том случае, если новая версия библиотеки `ReferencedLibrary` совместима с приложением на уровне двоичного кода. Сведения об изменениях, на которые следует обращать внимание при определении совместимости, см. выше в подразделе Обратная совместимость.

Создать...

Модификатор `new` служит для скрытия унаследованных членов базового класса. Это один из способов, которым производные классы могут реагировать на изменения в базовом классе.

Возьмем следующий пример:

C#

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
```

```
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Выходные данные

Консоль

```
A base method
A derived method
```

В приведенном выше примере можно видеть, как в классе `DerivedClass` скрывается метод `MyMethod`, имеющийся в классе `BaseClass`. Таким образом, если в базовом классе в новой версии библиотеки появляется член, который уже имеется в вашем производном классе, можно просто использовать модификатор `new` для члена производного класса, чтобы скрыть соответствующий член базового класса.

Если модификатор `new` не указан, в производном классе будут по умолчанию скрываться конфликтующие члены базового класса. При этом будет выводиться предупреждение компилятора, однако код будет компилироваться. Это означает, что добавление новых членов в существующий класс делает новую версию библиотеки совместимой с кодом, в котором она используется, на уровне как исходного, так и двоичного кода.

переопределение

Модификатор `override` означает, что производная реализация расширяет реализацию члена базового класса, а не скрывает его. К члену базового класса должен быть применен модификатор `virtual`.

C#

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}
```

```
public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine($"Base Method One: {b.MethodOne()}");
    Console.WriteLine($"Derived Method One: {d.MethodOne()}");
}
```

Выходные данные

Консоль

```
Base Method One: Method One
Derived Method One: Derived Method One
```

Модификатор `override` оценивается во время компиляции, и компилятор выдаст ошибку, если ему не удастся найти виртуальный член для переопределения.

Знание рассмотренных приемов и ситуаций, в которых их следует применять, поможет вам значительно упростить переход с одной версии библиотеки на другую.

Практическое руководство (C#)

17.06.2025

В разделе "Практическое руководство по C#" можно найти краткие ответы на распространенные вопросы. В некоторых случаях статьи могут быть перечислены в нескольких разделах. Мы хотели сделать их легко доступными для различных путей поиска.

Общие понятия C#

Существует несколько советов и рекомендаций, которые являются распространенными методиками разработчика C#:

- Инициализировать объекты с помощью инициализатора объектов.
- Используйте перегрузку оператора.
- Реализуйте и вызывайте пользовательский метод расширения.
- Создайте новый метод для enum типа с помощью методов расширения.

Члены классов, записей и структур

Вы создаете классы, записи и структуры для реализации программы. Эти методы часто используются при написании классов, записей или структур.

- Объявляйте автоматически реализованные свойства.
- Объявляйте и используйте свойства чтения и записи.
- Определите константы.
- Переопределите ToString метод для получения строкового вывода.
- Определите абстрактные свойства.
- Используйте функции xml-документации для документирования кода.
- Явно реализуйте члены интерфейса, чтобы ваш публичный интерфейс оставался кратким.
- Явно реализуйте члены двух интерфейсов.

Работа с коллекциями

Эти статьи помогают работать с коллекциями данных.

- Инициализация словаря с помощью инициализатора коллекции.

Работа со строками

Строки — это базовый тип данных, используемый для отображения или управления текстом. В этих статьях демонстрируются распространенные методики с строками.

- Сравнение строк.
- Измените содержимое строки.
- Определите, представляет ли строка число.
- Используется String.Split для разделения строк.
- Объединение нескольких строк в одну.
- Найдите текст в строке.

Преобразование между типами

Может потребоваться преобразовать объект в другой тип.

- Определите, представляет ли строка число.
- Преобразование строковых значений, представляющих шестнадцатеричные числа, в числа и обратно.
- Преобразуйте строку в .DateTime
- Преобразование массива байтов в int.
- Преобразуйте строку в число.
- Используйте сопоставление шаблонов и операторы as и is для безопасного приведения к другому типу.
- Определение преобразований пользовательских типов.
- Определите, является ли тип значением, допускаемым значением NULL.
- Преобразование между nullable и non-nullable типами значений

Сравнения на равенство и упорядоченность

Вы можете создавать типы, определяющие собственные правила равенства или определяющие естественное упорядочение между объектами этого типа.

- Проверка на равенство на основе ссылок.
- Определите равенство на основе значений для типа.

Обработка исключений

Программы .NET сообщают, что методы не завершили свою работу, выбрасывая исключения. В этих статьях вы узнаете, как работать с исключениями.

- Обработка исключений с помощью try и catch.
- Очистка ресурсов посредством finally условий.

- Восстановление после исключений, не относящихся к CLS (спецификации общего языка).

Делегаты и события

Делегаты и события предоставляют возможность стратегий, включающих слабо связанные блоки кода.

- Объявите, создайте экземпляр и используйте делегаты.
- Объедините делегаты многоадресной рассылки.

События предоставляют механизм публикации или подписки на уведомления.

- Подписка и отмена подписки на события.
- Реализуйте события, объявленные в интерфейсах.
- Следуйте рекомендациям .NET, когда код публикует события.
- Вызов событий, определенных в базовых классах из производных классов.
- Реализуйте пользовательские методы доступа к событиям.

Методики LINQ

LINQ позволяет создавать код для запроса любого источника данных, поддерживающего шаблон выражения запроса LINQ. Эти статьи помогают понять шаблон и работать с различными источниками данных.

- Запрос коллекции.
- Используйте var в выражениях запросов.
- Возвращает подмножества свойств элемента из запроса.
- Написание запросов с помощью сложной фильтрации.
- Сортировка элементов источника данных.
- Сортировка элементов по нескольким ключам.
- Управление типом проекции.
- Подсчет количества вхождений значения в исходной последовательности.
- Вычислите промежуточные значения.
- Отладка пустых результатов запроса.
- Добавьте настраиваемые методы в запросы LINQ.

Несколько потоков и асинхронная обработка

Современные программы часто используют асинхронные операции. В этих статьях вы узнаете, как использовать эти методы.

- Повышение асинхронной производительности с помощью System.Threading.Tasks.Task.WhenAll.
- Выполнение нескольких веб-запросов параллельно с использованием async и await.
- Используйте пул потоков.

Аргументы командной строки для вашей программы

Как правило, программы C# имеют аргументы командной строки. В этих статьях вы узнаете, как получить доступ к этим аргументам командной строки и обработать их.

- Извлеките все аргументы командной строки с помощью for.

Как перехватить исключение, не соответствующее CLS

Некоторые языки .NET, включая C++/CLI, позволяют объектам выбрасывать исключения, которые не являются производными от Exception. Такие исключения называются *исключениями, отличными от CLS* , или *исключениями, не являющихся исключениями*. В C# нельзя создавать исключения, отличные от CLS, но их можно поймать двумя способами:

- `catch (RuntimeWrappedException e)` внутри блока.

  По умолчанию сборка Visual C# перехватывает исключения, отличные от CLS, в виде упакованных исключений. Используйте этот метод, если вам нужен доступ к исходному исключению, к которому можно получить доступ через RuntimeWrappedException.WrappedException свойство. Далее в этой статье описывается, как перехватывать исключения таким образом.

- В общем блоке catch (блок catch без указания типа исключения), который помещается после всех остальных `catch` блоков.

  Используйте этот метод, если вы хотите выполнить некоторые действия (например, запись в файл журнала) в ответ на исключения, отличные от CLS, и вам не нужен доступ к сведениям об исключении. По умолчанию среда CLR упаковывает все исключения. Чтобы отключить это поведение, добавьте этот атрибут уровня сборки в код, как правило, в файле AssemblyInfo.cs: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`

Перехват исключения, не соответствующего CLS

В блоке `catch(RuntimeWrappedException e)` получите доступ к исходному исключению через свойство RuntimeWrappedException.WrappedException.

Пример

В следующем примере показано, как перехватывать исключение, отличное от CLS, которое было создано из библиотеки классов, написанной в C++/CLI. Обратите внимание, что в этом примере клиентский код C# заранее знает, что выбрасывается исключение типа System.String. Свойство можно преобразовать обратно в его исходный тип RuntimeWrappedException.WrappedException, если этот тип доступен для вашего кода.

```
```

C#

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

См. также

- RuntimeWrappedException
- Исключения и обработка исключений

Last updated on 31.03.2026

Стратегии миграции, допускающие значение NULL

> **Совет**
>
> **Запуск нового проекта?** Новые проекты, созданные из .NET 6 или более поздних шаблонов, уже имеют набор `<Nullable>enable</Nullable>`. Вам не нужна стратегия миграции: перейдите к разделу "Разрешить предупреждения, допускающие значение NULL".
>
> **Сохранение существующей базы кода?** Сначала прочтите раздел Ссылочные типы, допускающие значение null, чтобы понять контексты, аннотации и состояние null. В этой статье предполагается, что вы знакомы с этими понятиями и готовы к планированию развертывания.

При включении ссылочных типов, допускающих значение NULL, в большом проекте, запущенном до появления ссылочных типов, допускающих значение NULL, компилятор одновременно создает множество предупреждений. Миграция — это *поэтапная организация* работы: выбор контекста по умолчанию, показ предупреждений по одному файлу или по одному разделу и постепенный переход к `<Nullable>enable</Nullable>` во всём проекте. Правильная последовательность зависит от того, насколько активна база кода и сколько рисков вы можете принять в одном проходе.

Конечное состояние во всех случаях одинаково: проект задаёт `<Nullable>enable</Nullable>` и не содержит директив препроцессора `#nullable`.

Выбор контекста по умолчанию

Контекст, допускающий значение NULL, имеет два независимых флага: *заметки* (объявляется ли `?` ссылочный тип, допускающий значение NULL), и *предупреждения* (выдает ли компилятор диагностику). Задайте их вместе как одно `<Nullable>` значение:

Развернуть таблицу

Значение по умолчанию	Annotations	Предупреждения	лучше всего подходит для
`disable` *(неявное)*	off	off	Стабильные библиотеки, для которых на этом этапе не планируется работа над новыми функциями.
`enable`	on	on	Активные базы кода с частыми новыми файлами. Новый код включён по умолчанию.
`warnings`	off	on	Двухэтапная миграция: сначала устраните предупреждения, затем добавьте аннотации.
`annotations`	on	off	Аннотируйте публичный API перед исправлением внутренних предупреждений.

Выберите стратегию, которая лучше всего соответствует целям миграции проекта:

- **Отключить как значение по умолчанию.** Задайте `<Nullable>disable</Nullable>` и добавьте `#nullable enable` в верхней части каждого файла при его переносе. Существующие файлы остаются вне контекста nullable, пока вы их не затронете. Этот вариант имеет наименьшее трение для стабильных библиотек, так как новая функция работает редко.
- **Включите в качестве значения по умолчанию.** Установите `<Nullable>enable</Nullable>` и добавьте `#nullable disable` в начало каждого файла, который вы еще не перенесли. Каждый новый файл изначально создаётся с учётом nullable-аннотаций, поэтому объём незавершённой миграции может только сокращаться. Этот выбор лучше, если разработка активна.
- **Предупреждения по умолчанию.** Задайте `<Nullable>warnings</Nullable>`. Выберите этот параметр по умолчанию для двухэтапной миграции: предупреждения об адресе, в то время как каждый ссылочный тип по-прежнему рассматривается как невнимательный, а затем включите заметки. Разделение между двумя этапами позволяет сосредоточиться на каждом шаге.
- **Аннотации по умолчанию.** Задайте `<Nullable>annotations</Nullable>`. Начните с аннотирования общедоступного API (`?` для членов, которые допускают `null`), прежде чем разбираться с предупреждениями. Компилятор пока не выдает предупреждения, поэтому можно урегулировать поверхность API без отвлекающих факторов.

Файл проекта определяет глобальное значение по умолчанию. `#nullable` Директивы препроцессора переопределяют значение по умолчанию для региона кода:

XML

```
<PropertyGroup>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

В файлах исходного кода директива включает или отключает для области действие параметра проекта, определяющего допустимость значений NULL:

C#

```
#nullable disable
public static class LegacyHelper
{
    // This file is nullable-oblivious. Reference types use the legacy rules.
    public static string GetGreeting(string name) =>
        name == null ? "hello" : $"hello {name}";
}
#nullable restore

#nullable enable
public static class MigratedHelper
{
    // This file is fully migrated. Reference types are non-nullable by default.
    public static string GetGreeting(string? name) =>
        name is null ? "hello" : $"hello {name}";
}
#nullable restore
```

Перенос файла по файлу

Самый предсказуемый способ миграции большого проекта — включать предупреждения или аннотации для каждого файла отдельно. Шаблон одинаков независимо от выбранного по умолчанию:

1. Выберите файл. Начните с наиболее глубоких листовых типов в графе зависимостей, а затем переходите к внешним уровням. Аннотирование типа вызывает новые предупреждения в вызывающем коде, поэтому работа снизу вверх сводит к минимуму объём переделок.
2. Добавьте директиву `#nullable`, которая подключает файл к новому поведению. Используйте `#nullable enable`, если нужны оба флага. Используется `#nullable enable warnings` только для предупреждения.
3. Устраните предупреждения в файле с помощью методов, описанных в разделе Устранение предупреждений о допустимости значений NULL.
4. Повторите для следующего файла.

5. Если директива задана в каждом файле проекта, удалите эти директивы и установите `<Nullable>enable</Nullable>` на уровне проекта.

Если у вашей базы кода уже есть `<Nullable>enable</Nullable>`, вы движетесь *в противоположном* направлении. Отключайте предупреждения в неигрированных файлах до тех пор, пока не будете готовы. Используйте `#nullable disable`, чтобы исключить файлы, затем удаляйте подавления по одному.

Миграция на двух этапах

Двухэтапная миграция отделяет два типа работы, к которым относятся ссылочные типы, допускающие значение NULL. Вы можете расположить фазы в любом порядке, в зависимости от того, какая форма стабильности для вас важнее.

Предупреждения сначала, а затем заметки

Начинайте с предупреждений, если приоритетом является исправление скрытых System.NullReferenceException ошибок:

1. **Этап 1: Устранение предупреждений.** Задайте для проекта значение по умолчанию `warnings`. Ссылочные типы по-прежнему игнорируют допускаемость NULL, поэтому система типов пока не меняется. Компилятор выдает предупреждения везде, где существующий код уже может вызвать System.NullReferenceException. Добавьте проверки на null, перестройте логику выполнения или примените атрибуты, пока в проекте не останется предупреждений. Каждое исправление делает рабочий код более устойчивым даже до появления заметок.
2. **Этап 2. Добавление заметок.** Переключить значение проекта по умолчанию на `enable`. Ссылочные типы теперь по умолчанию не допускают значения null, а локальные переменные `var` становятся допускающими значение null. Новые предупреждения отражают объявления, которые не соответствуют использованию переменных. Добавьте `?` в типы, которые должны разрешать `null`. Ужесточьте API, которые должны требовать аргументы, отличные от null.

Сначала заметки, а затем предупреждения

Используйте аннотации, если приоритетом является стабилизация публичного API. Эта последовательность хорошо подходит для библиотек: вы можете поставлять аннотированные сигнатуры, чтобы пользователи видели корректные контракты, а затем устранить внутренние предупреждения в удобные для вас сроки.

1. **Этап 1. Добавление заметок.** Задайте для проекта значение по умолчанию `annotations`. Ссылочные типы по умолчанию считаются не допускающими значения null, но компилятор не выводит предупреждения, поэтому лишний шум не мешает вам. Просмотрите общедоступный API и добавьте `?` к каждому элементу, который может возвращать или принимать `null`. Ужесточите требования к подписям там, где этого делать не следует. Поскольку предупреждения отключены, вы можете зафиксировать структуру API в отдельных целевых коммитах, не пытаясь одновременно разбираться с реализацией.
2. **Этап 2: Устраните предупреждения.** Переключите проект по умолчанию на `enable`. Аннотации, которые вы добавили на этапе 1, теперь используются при анализе состояний допустимости NULL, поэтому предупреждения, выдаваемые компилятором, с самого начала становятся точнее: каждое из них указывает на участок кода, поведение которого не соответствует контракту, который вы уже опубликовали. Устраните их с помощью методов, описанных в предупреждениях, допускающих значение NULL.

Выбор между упорядочениями

Каждое упорядочивание разделяет этапы на более мелкие, более проверяемые диффы. Один этап изменяет только поведение, а другие изменения изменяют только типы. Недостатком является то, что вы посещаете каждый файл дважды. Для зрелого, стабильного кода, где каждое изменение связано с риском, два прохода обычно оправданы. Сначала выберите *предупреждения* , когда требуется ужесточить выполнение кода. Выбирайте *аннотации* в первую очередь, если вам важнее опубликовать стабильный контракт.

Созданный код исключен

Компилятор обрабатывает файлы, помеченные как созданные, как если бы контекст, допускающий значение NULL, был отключен независимо от параметра проекта. Файл считается сгенерированным, если выполняется любое из следующих условий:

- Правило `.editorconfig` задаёт `generated_code = true` для файла.
- Первый комментарий в файле содержит `<auto-generated>` или `<auto-generated/>`.
- Имя файла начинается с `TemporaryGeneratedFile_`.
- Имя файла заканчивается на `.designer.cs`, `.generated.cs`, `.g.cs` или `.g.i.cs`.

Генераторы, создающие вывод с учетом nullable-аннотаций, могут снова включить эту возможность, добавив `#nullable enable` в начало сгенерированного файла.

Когда вы закончите

После того как каждый файл участвует в проекте по умолчанию, и `<Nullable>enable</Nullable>` элемент задается:

- Удалите каждую `#nullable` директиву в источнике.
- Удалите инициализаторы `null!` и `default!`, которые вы добавили только для того, чтобы подавить предупреждения при миграции. Замените их корректной инициализацией или сделайте тип ссылочным типом, допускающим значение NULL.
- Выборочно проверьте общедоступный API. Каждый член, который возвращает или принимает `null`, должен быть аннотирован с помощью `?`. Заметки являются частью контракта после отправки пакета.

Теперь у вас та же ситуация, что и в новых проектах: ссылочные типы, допускающие значение null, являются частью системы типов, и любые новые предупреждения отражают реальное несоответствие между объявлениями и кодом. Используйте устранение предупреждений о допустимости значений NULL, чтобы устранять их по мере появления.

Связанный контент

- Ссылочные типы, допускающие значение NULL
- Разрешение предупреждений, допускающих значение NULL
- Атрибуты статического анализа, допускающие значение NULL
- Работа со ссылочными типами, допускающими значение NULL, в EF Core

ⓘ **Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 26.05.2026

Атрибуты

Статья • 14.03.2025

Атрибуты предоставляют эффективный способ связывания метаданных или декларативной информации с кодом (сборки, типы, методы, свойства и т. д.). После связывания атрибута с сущностью программы можно запросить атрибут во время выполнения с помощью метода, называемого *отражением*.

Атрибуты имеют следующие свойства:

- Атрибуты добавляют метаданные в программу. *метаданные* — это сведения о типах, определенных в программе. Все сборки .NET содержат указанный набор метаданных, описывающий типы и члены типов, определенные в сборке. Вы можете добавить настраиваемые атрибуты, чтобы указать любые другие необходимые сведения.
- Атрибуты можно применять ко всем сборкам, модулям или небольшим элементам программы, таким как классы и свойства.
- Атрибуты могут принимать аргументы так же, как методы и свойства.
- Атрибуты позволяют программе проверять собственные метаданные или метаданные в других программах с помощью отражения.

Работа с отражением

API отражения, предоставляемые Type, описывают сборки, модули и типы. Можно использовать отражение для динамического создания экземпляра типа, привязки типа к существующему объекту или получения типа из существующего объекта и вызова методов или доступа к его полям и свойствам. При использовании атрибутов в коде отражение позволяет получить к ним доступ. Дополнительные сведения см. в разделе Атрибуты.

Вот простой пример использования метода GetType() для отражения. Все типы из `Object` базового класса наследуют этот метод, который используется для получения типа переменной:

> **ⓘ Примечание**
>
> Убедитесь, что вы добавили операторы `using System;` и `using System.Reflection;` в верхней части файла кода C# (*.cs*).

```
C#
```

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

Выходные данные показывают тип:

Выходные данные

```
System.Int32
```

В следующем примере используется отражение для получения полного имени загруженной сборки.

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

Результат аналогичен следующему примеру:

Выходные данные

```
System.Private.CoreLib, Version=7.0.0.0, Culture=neutral,
PublicKeyToken=7cec85d7bea7798e
```

Различия ключевых слов для IL

Ключевые слова `protected` C# и `internal` не имеют значения в промежуточном языке (IL) и не используются в API отражения. Соответствующие термины в IL — это *семейства* и *сборки*. Ниже приведены некоторые способы использования следующих терминов:

- Чтобы определить `internal` метод с помощью отражения, используйте IsAssembly свойство.
- Чтобы определить метод `protected internal`, используйте IsFamilyOrAssembly.

Работа с атрибутами

Атрибуты можно поместить почти в любое объявление, хотя определенный атрибут может ограничить типы объявлений, для которых он действителен. В

C#можно указать атрибут, поместив имя атрибута, заключенного в квадратные скобки (`[]`) над объявлением сущности, к которой она применяется.

В этом примере вы используете атрибут SerializableAttribute, чтобы применить определенную характеристику к классу:

```
C#

[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

Можно объявить метод с атрибутом DllImportAttribute :

```
C#

[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

В объявлении можно поместить несколько атрибутов:

```
C#

void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Некоторые атрибуты можно указать несколько раз для определенной сущности. В следующем примере показано многопользовательское использование атрибута ConditionalAttribute :

```
C#

[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> ⓘ **Примечание**
>
> По соглашению все имена атрибутов заканчиваются суффиксом "Атрибут", чтобы отличить их от других типов в библиотеках .NET. Однако при

использовании атрибутов в коде не требуется указывать суффикс атрибутов. Например, `[DllImport]` объявление эквивалентно `[DllImportAttribute]` объявлению, но `DllImportAttribute` является фактическим именем класса в библиотеке классов .NET.

Параметры атрибута

Многие атрибуты имеют параметры, которые могут быть *позициональными*, *неназванными* или *именованными*. В следующей таблице описывается, как работать с именованными и позиционных атрибутами:

Позиционные параметры

Параметры конструктора атрибутов:

Именованные параметры

Свойства или поля атрибута:

- Необходимо указать, нельзя пропустить
- Всегда указывайте первое
- Укажите в **определенном** порядке
- Всегда необязательный, опущен при значении false
- Укажите после позиционных параметров
- Укажите в любом порядке

Например, в следующем коде показаны три эквивалентных `DllImport` атрибута:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

Первый параметр, имя библиотеки DLL, является позиционным и всегда идет первым. Другие случаи — это именованные параметры. В этом сценарии оба именованных параметра по умолчанию используют значение false, поэтому они могут быть опущены. Сведения о значениях параметров по умолчанию см. в документации по отдельным атрибутам. Дополнительные сведения о разрешенных типах параметров см. в разделе Атрибуты спецификации языка C#.

Цели атрибутов

целевой атрибута — это сущность, к которой применяется атрибут. Например, атрибут может применяться к классу, методу или сборке. По умолчанию атрибут применяется к элементу, который следует за ним. Но вы также можете явно определить элемент для связывания, например метод, параметр или возвращаемое значение.

Чтобы явно определить целевой объект атрибута, используйте следующий синтаксис:

```
C#
```

```
[target : attribute-list]
```

В следующей таблице показан список возможных `target` значений.

Развернуть таблицу

Целевое значение	Применимо к
`assembly`	Вся сборка
`module`	Текущий модуль сборки
`field`	Поле в классе или структуре
`event`	Событие
`method`	Метод или методы доступа к свойствам `get` и `set`
`param`	Параметры метода или параметры аксессоров свойств `set`
`property`	Свойство
`return`	Возвращаемое значение метода, индексатора свойств или метода доступа к свойствам `get`
`type`	Структура, класс, интерфейс, перечисление или делегат

Целевое `field` значение можно указать для применения атрибута к резервному полю, созданному для автоматически реализованного свойства.

В следующем примере показано, как применять атрибуты к сборкам и модулям. Дополнительные сведения см. в разделе "Общие атрибуты" (C#).

```
C#
```

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

В следующем примере показано, как применять атрибуты к методам, параметрам метода и возвращаемым значениям метода в C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

> ⓘ **Примечание**
>
> Независимо от целевых объектов, для которых атрибут `ValidatedContract` считается допустимым, целевой объект `return` должен быть указан, даже если атрибут `ValidatedContract` определен только для применения к возвращаемым значениям. Другими словами, компилятор не использует `AttributeUsage` сведения для разрешения неоднозначных целевых объектов атрибутов. Дополнительные сведения см. в разделе AttributeUsage.

Просмотр способов использования атрибутов

Ниже приведены некоторые распространенные способы использования атрибутов в коде:

- Пометьте методы контроллера, которые отвечают на сообщения POST с помощью атрибута `HttpPost` . Дополнительные сведения см. в

HttpPostAttribute классе.
- Описание способов маршалирования параметров метода при взаимодействии с машинным кодом. Дополнительные сведения см. в MarshalAsAttribute классе.
- Описание свойств объектной модели компонента (COM) для классов, методов и интерфейсов.
- Вызов неуправляемого кода с помощью DllImportAttribute класса.
- Опишите сборку с точки зрения названия, версии, описания или торговой марки.
- Определите, какие элементы класса следует сериализовать для обеспечения сохранности данных.
- Опишите, как сопоставить члены класса и узлы XML для сериализации XML.
- Описание требований безопасности для методов.
- Укажите характеристики, используемые для обеспечения безопасности.
- Управляйте оптимизациями с помощью JIT-компилятора, чтобы код оставался легким для отладки.
- Получение сведений о вызывающем методе.

Просмотр сценариев отражения

Отражение полезно в следующих сценариях:

- Доступ к атрибутам в метаданных программы. Дополнительные сведения см. в разделе "Получение сведений", хранящихся в атрибутах.
- Проверьте и инициализируйте типы в сборке.
- Создайте новые типы во время выполнения с помощью классов в System.Reflection.Emit пространстве имен.
- Выполните методы поздней привязки и доступа к типам, созданным во время выполнения. Дополнительные сведения см. в разделе динамической загрузки и использования типов.

Связанные ссылки

- Общие атрибуты (C#)
- Сведения о вызывающем объекте (C#)
- Атрибуты
- Отражение
- Просмотр сведений о типе
- Отражение и универсальные типы
- System.Reflection.Emit

- Получение сведений, хранящихся в атрибутах

Создание настраиваемых атрибутов

17.06.2025

Вы можете создавать собственные настраиваемые атрибуты, определяя класс атрибутов, класс, производный напрямую или косвенно от Attribute, что позволяет быстро и легко идентифицировать определения атрибутов в метаданных. Предположим, вы хотите пометить типы с именем программиста, который написал тип. Можно определить пользовательский `Author` класс атрибутов:

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;
        Version = 1.0;
    }
}
```

Имя класса — это имя атрибута `AuthorAttribute`, плюс суффикс `Author`. Он является производным от `System.Attribute`, поэтому это класс настраиваемых атрибутов. Параметры конструктора — это позиционные параметры настраиваемого атрибута. В этом примере `name` используется позиционный параметр. Все открытые поля или свойства для чтения и записи являются именованными параметрами. В этом случае `version` является единственным именованным параметром. Обратите внимание на использование атрибута `AttributeUsage`, чтобы атрибут `Author` был допустим только в объявлениях классов и `struct`.

Этот новый атрибут можно использовать следующим образом:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` имеет именованный параметр, `AllowMultiple`, с помощью которого можно сделать атрибут настраиваемым на одноразовое или многоразовое использование. В следующем примере кода создается многопользовательский атрибут.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}
```

В следующем примере кода к классу применяются несколько атрибутов одного типа.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}
```

См. также

- System.Reflection
- Написание настраиваемых атрибутов
- AttributeUsage (C#)

Доступ к атрибутам с помощью отражения

17.06.2025

Тот факт, что вы можете определить настраиваемые атрибуты и поместить их в исходный код, не будет иметь небольшого значения без какого-то способа получения этой информации и действия над ним. С помощью отражения можно получить сведения, определенные с помощью пользовательских атрибутов. Ключевым методом является `GetCustomAttributes`, который возвращает массив объектов, которые являются эквивалентами времени выполнения атрибутов исходного кода. Этот метод имеет множество перегруженных версий. Дополнительные сведения см. в разделе Attribute.

Спецификация атрибута, например:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

концептуально эквивалентен следующему коду:

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

Однако код не выполняется, пока `SampleClass` не будет запрашивать атрибуты. Вызов `GetCustomAttributes` на `SampleClass` приводит к созданию и инициализации объекта `Author`. Если класс имеет другие атрибуты, другие объекты атрибутов создаются аналогичным образом. `GetCustomAttributes` затем возвращает `Author` объект и любые другие объекты атрибутов в массиве. Затем можно выполнить итерацию по этому массиву, определить, какие атрибуты были применены на основе типа каждого элемента массива, а также извлечь сведения из объектов атрибутов.

Ниже приведен полный пример. Настраиваемый атрибут определяется, применяется к нескольким сущностям и извлекается с помощью отражения.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    public static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine($"Author information for {t}");
```

```
        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  //
Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is AuthorAttribute a)
            {
                System.Console.WriteLine($"   {a.GetName()}, version
{a.Version:f}");
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

См. также

- System.Reflection
- Attribute
- Извлечение сведений, хранящихся в атрибутах

Создание объединения C/C++ с помощью атрибутов в C#

Статья • 05.03.2024

С помощью атрибутов можно настраивать расположение структур в памяти. Например, можно создать так называемое объединение в C/C++ с помощью атрибутов `StructLayout(LayoutKind.Explicit)` и `FieldOffset`.

В этом сегменте кода все поля объединения `TestUnion` начинаются с одного адреса в памяти.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

Следующий код является другим примером, в котором поля начинаются с разных явно заданных расположений.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
```

```
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

Два целочисленных поля `i1` и объединенные используют те же расположения памяти, что `lg` и `i2` . Использует `lg` первые 8 байтов или `i1` использует первые 4 байта и `i2` использует следующие 4 байта. Такое управление расположением структуры полезно при использовании вызова неуправляемого кода.

См. также

- System.Reflection
- Attribute
- Атрибуты

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Универсальные шаблоны и атрибуты

Атрибуты можно применять к универсальным типам так же, как и негенерические типы. Однако атрибуты можно применять только к *открытым универсальным типам и закрытым универсальным типам, а не к частично созданным универсальным типам*. Открытый обобщённый тип — это тот, где ни один из аргументов типа не указан, например `Dictionary<TKey, TValue>`. Закрытый построенный обобщённый тип указывает все аргументы типа, такие как `Dictionary<string, object>`. Частично *созданный универсальный тип* указывает некоторые, но не все аргументы типа. Например, `Dictionary<string, TValue>`. Несвязанный универсальный *тип* является одним из случаев, когда аргументы типа опущены, например `Dictionary<,>`.

Такой настраиваемый атрибут используется в следующих примерах:

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

Атрибут может ссылаться на несвязанный универсальный тип:

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Укажите несколько параметров типа, используя соответствующее количество запятых. В этом примере `GenericClass2` имеет два параметра типа:

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

Атрибут может ссылаться на закрытый сконструированный универсальный тип:

C#

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Атрибут, ссылающийся на параметр универсального типа, вызывает ошибку во время компиляции:

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error CS0416
class ClassD<T> { }
```

Универсальный тип может наследоваться от Attribute:

C#

```
public class CustomGenericAttribute<T> : Attribute { }
```

Для получения сведений об универсальном типе или параметре типа во время выполнения можно использовать методы из System.Reflection. Дополнительные сведения см. в разделе "Обобщения и рефлексия".

См. также

- Универсальные шаблоны
- Атрибуты

Last updated on 24.02.2026

Практическое руководство. Выполнение запроса к метаданным сборки при помощи отражения (LINQ)

Статья • 02.03.2024

Api отражения .NET используются для проверки метаданных в сборке .NET и создания коллекций типов, элементов типа и параметров, которые находятся в этой сборке. Поскольку эти коллекции поддерживают универсальный интерфейс IEnumerable<T>, их можно запрашивать с помощью LINQ.

В следующем примере показано использование LINQ с отражением для извлечения определенных метаданных о методах, соответствующих условиям поиска. В этом случае запрос находит имена всех методов в сборке, возвращающих перечислимые типы, такие как массивы.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                        || (method.ReturnType.GetInterface(

typeof(System.Collections.Generic.IEnumerable<>).FullName!) != null
                        && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

В примере используется метод Assembly.GetTypes для возвращения массива типов в указанной сборке. Чтобы вернуть только открытые типы, применяется фильтр where. Для каждого открытого типа создается вложенный запрос с использованием массива MethodInfo, который возвращается из вызова Type.GetMethods. Эти

результаты фильтруются для возвращения только тех методов, возвращаемый тип которых является массивом или типом, который реализует IEnumerable<T>. Наконец, эти результаты группируются с помощью имени типа в качестве ключа.

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Обобщения и рефлексия

17.06.2025

Так как среда CLR имеет доступ к данным универсального типа во время выполнения, можно использовать отражение для получения сведений о универсальных типах таким же образом, как и для негенерических типов. Дополнительные сведения см. в разделе «Универсальные шаблоны в среде выполнения».

Пространство System.Reflection.Emit имен также содержит новые члены, поддерживающие дженерики. Смотрите Практическое руководство по определению универсального типа с использованием Reflection.Emit.

Список инвариантных условий терминов, используемых в универсальном отражении, см. в IsGenericType примечаниях свойств:

- IsGenericType: возвращает значение true, если тип является универсальным.
- GetGenericArguments: возвращает массив `Type` объектов, представляющих аргументы типа, предоставленные для созданного типа, или параметры типа определения универсального типа.
- GetGenericTypeDefinition: возвращает базовое определение универсального типа для текущего созданного типа.
- GetGenericParameterConstraints: возвращает массив `Type` объектов, представляющих ограничения текущего параметра универсального типа.
- ContainsGenericParameters: возвращает значение true, если тип или любой из его вложенных типов или методов содержит параметры типа, для которых определенные типы не были предоставлены.
- GenericParameterAttributes: получает сочетание `GenericParameterAttributes` флагов, описывающих специальные ограничения текущего параметра универсального типа.
- GenericParameterPosition: Для объекта `Type`, представляющего параметр типа, получает индекс этого параметра типа в списке параметров универсального типа или универсального метода, в котором он объявлен.
- IsGenericParameter: возвращает значение, указывающее, представляет ли текущий `Type` параметр типа универсального типа или определения метода.
- IsGenericTypeDefinition: получает значение, указывающее, представляет ли текущий Type определение универсального типа, из которого можно создать другие универсальные типы. Возвращает значение true, если тип представляет определение универсального типа.
- DeclaringMethod: возвращает универсальный метод, определяющий текущий параметр универсального типа, или значение NULL, если параметр типа не определен универсальным методом.

- MakeGenericType: заменяет элементы массива типов для параметров типа текущего определения универсального типа и возвращает Type объект, представляющий результирующий созданный тип.

Кроме того, члены MethodInfo класса позволяют включить сведения о времени выполнения для универсальных методов. См. IsGenericMethod примечания свойства для списка инвариантных условий терминов, используемых для отражения универсальных методов:

- IsGenericMethod: возвращает значение true, если метод является универсальным.
- GetGenericArguments: возвращает массив объектов Type, представляющих аргументы типа созданного универсального метода или параметры типа определения универсального метода.
- GetGenericMethodDefinition: возвращает определение базового универсального метода для текущего созданного метода.
- ContainsGenericParameters: возвращает значение true, если метод или любой из его вложенных типов содержит параметры типа, для которых определенные типы не были предоставлены.
- IsGenericMethodDefinition: возвращает значение true, если текущий MethodInfo представляет определение универсального метода.
- MakeGenericMethod: заменяет элементы массива типов для параметров типа текущего определения универсального метода и возвращает MethodInfo объект, представляющий результирующий созданный метод.

См. также

- Обобщения
- Рефлексия и универсальные типы
- Обобщения

Определение и чтение настраиваемых атрибутов

17.06.2025

Атрибуты предоставляют способ связывания информации с кодом декларативным способом. Они также могут предоставлять многократно используемый элемент, который можно применять к различным целевым объектам. Рассмотрим ObsoleteAttribute. Его можно применять к классам, структурым, методам, конструкторам и т. д. Он *объявляет* , что элемент устарел. Задача компилятора C# заключается в том, чтобы найти этот атрибут и выполнить соответствующие действия в ответ.

В этом руководстве вы узнаете, как добавлять атрибуты в код, создавать и использовать собственные атрибуты, а также использовать некоторые атрибуты, встроенные в .NET.

Предпосылки

Необходимо настроить компьютер для запуска .NET. Инструкции по установке можно найти на странице загрузки .NET . Это приложение можно запустить в Windows, Ubuntu Linux, macOS или в контейнере Docker. Вам нужно установить любимый редактор кода. В следующих описаниях используется Visual Studio Code, который является кроссплатформенным редактором с открытым кодом. Однако вы можете использовать все инструменты, с которыми вы комфортно.

Создание приложения

Теперь, когда вы установили все средства, создайте консольное приложение .NET. Чтобы использовать генератор командной строки, выполните следующую команду в любимой оболочке:

Интерфейс командной строки.NET

```
dotnet new console
```

Эта команда создает минимальные файлы проекта .NET. Запустите `dotnet restore` чтобы восстановить зависимости, необходимые для компиляции этого проекта.

Вам не нужно выполнять команду dotnet restore, так как она выполняется неявно всеми командами, которые требуют восстановления, например `dotnet new`, `dotnet build`, `dotnet`

`run`, `dotnet test`, `dotnet publish` и `dotnet pack`. Чтобы отключить неявное восстановление, используйте параметр `--no-restore`.

Команду `dotnet restore` по-прежнему удобно использовать в некоторых сценариях, где необходимо явное восстановление, например в сборках с использованием непрерывной интеграции в Azure DevOps Services или системах сборки, где требуется явно контролировать время восстановления.

В документации по dotnet restore приведены сведения об управлении каналами NuGet.

Чтобы выполнить программу, используйте `dotnet run`. Вы должны увидеть вывод "Hello, World" на консоль.

Добавление атрибутов в код

В C#атрибуты — это классы, наследуемые от `Attribute` базового класса. Любой класс, наследующий от `Attribute`, может использоваться как своего рода "тег" на других элементах кода. Например, есть атрибут, называемый `ObsoleteAttribute`. Этот атрибут сигнализирует о том, что код устарел и больше не должен использоваться. Этот атрибут помещается в класс, например, с помощью квадратных скобок.

C#

```
[Obsolete]
public class MyClass
{
}
```

Хотя класс называется `ObsoleteAttribute`, в коде необходимо использовать только `[Obsolete]`. Большинство кода C# следует этому соглашению. При выборе можно использовать полное имя `[ObsoleteAttribute]` .

При маркировке устаревшего класса рекомендуется предоставить некоторые сведения о *том, почему* это устарело, и (или) *что* следует использовать. Для предоставления этого объяснения вы включаете строковый параметр в устаревший атрибут.

C#

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

Строка передается в качестве аргумента конструктору `ObsoleteAttribute` , как если бы вы писали `var attr = new ObsoleteAttribute("some string")`.

Параметры конструктора атрибутов ограничены простыми типами и литералами: `bool, int, double, string, Type, enums, etc` и массивами этих типов. Нельзя использовать выражение или переменную. Вы можете использовать позиционные или именованные параметры.

Создание собственного атрибута

Вы создаете атрибут, определяя новый класс, наследуемый от `Attribute` базового класса.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

В приведенном выше коде можно использовать `[MySpecial]` (или `[MySpecialAttribute]`) в качестве атрибута в другом месте базы кода.

C#

```
[MySpecial]
public class SomeOtherClass
{
}
```

Атрибуты в библиотеке базовых классов .NET, такие как `ObsoleteAttribute`, вызывают определенные действия в компиляторе. Однако любой атрибут, создаваемый вами, действует только как метаданные и не приводит к выполнению кода в классе атрибутов. Вам решать, как использовать эти метаданные в другом месте кода.

Здесь есть подвох, будьте внимательны. Как упоминалось ранее, при использовании атрибутов можно передавать только определенные типы. Однако при создании типа атрибута компилятор C# не останавливает создание этих параметров. В следующем примере вы создали атрибут с конструктором, который компилируется правильно.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
```

```
    }
}
```

Однако вы не можете использовать этот конструктор с синтаксисом атрибутов.

C#

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

Предыдущий код вызывает ошибку компилятора, например `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

Ограничение использования атрибутов

Атрибуты можно использовать в следующих "целевых объектах". Приведенные выше примеры показывают их на классах, но их также можно использовать для:

- Собрание
- Класс
- Конструктор
- Делегировать
- Перечисление
- Событие
- Поле
- Обобщенный параметр
- Интерфейс
- Метод
- Модуль
- Параметр
- Недвижимость
- ReturnValue
- Структура

При создании класса атрибутов по умолчанию C# позволяет использовать этот атрибут для любого из возможных целевых объектов атрибутов. Если вы хотите ограничить атрибут определенными целевыми объектами, это можно сделать с помощью `AttributeUsageAttribute` класса атрибутов. Вот именно, атрибут на атрибуте!

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Если вы попытаетесь поместить приведенный выше атрибут на то, что не является классом или структурой, вы получите ошибку компилятора, например `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

C#

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

Использование атрибутов, присоединенных к элементу кода

Атрибуты действуют как метаданные. Без какой-либо внешней силы, они на самом деле ничего не делают.

Для поиска и работы с атрибутами требуется рефлексия. Отражение позволяет писать код в C#, который проверяет другой код. Например, можно использовать отражение для получения сведений о классе (добавьте `using System.Reflection;` в голову кода):

C#

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

Это выводит что-то вроде: `The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`

После получения объекта `TypeInfo` (или объекта `MemberInfo`, `FieldInfo` или другого объекта), можно использовать метод `GetCustomAttributes`. Этот метод возвращает

коллекцию `Attribute` объектов. Вы также можете использовать `GetCustomAttribute` и указать тип атрибута.

Вот пример применения `GetCustomAttributes` на `MemberInfo` для `MyClass` (который мы видели ранее с атрибутом `[Obsolete]`).

C#

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Это выводится в консоль: `Attribute on MyClass: ObsoleteAttribute` Попробуйте добавить другие атрибуты к `MyClass`.

Важно отметить, что эти `Attribute` объекты создаются лениво. То есть они не создаются, пока вы не используете `GetCustomAttribute` или `GetCustomAttributes`. Они также создаются каждый раз. Вызов `GetCustomAttributes` дважды в строке возвращает два разных экземпляра `ObsoleteAttribute`.

Общие атрибуты во время выполнения

Атрибуты используются многими инструментами и платформами. NUnit использует такие `[Test]` атрибуты, как и `[TestFixture]` используемые тестовой командой NUnit. ASP.NET MVC использует такие атрибуты, как `[Authorize]` и предоставляет платформу фильтров действий для выполнения перекрестных задач по действиям MVC. PostSharp использует синтаксис атрибута, чтобы разрешить программирование на основе аспектов в C#.

Ниже приведены несколько важных атрибутов, встроенных в библиотеки базовых классов .NET Core:

- `[Obsolete]`. Этот объект использовался в приведенных выше примерах, и он находится в `System` пространстве имен. Полезно предоставить декларативную документацию по изменяющейся базе кода. Сообщение можно указать в виде строки, а другой логический параметр можно использовать для эскалации предупреждения компилятора до ошибки компилятора.
- `[Conditional]`. Этот атрибут находится в `System.Diagnostics` пространстве имен. Этот атрибут можно применять к методам (или классам атрибутов). Необходимо передать строку конструктору. Если эта строка не соответствует директиве `#define` , компилятор C# удаляет вызовы этого метода (но не сам метод). Обычно этот метод используется для отладки (диагностики).

- `[CallerMemberName]`. Этот атрибут можно использовать для параметров и является частью пространства имен `System.Runtime.CompilerServices`. `CallerMemberName` — это атрибут, используемый для внедрения имени метода, вызывающего другой метод. Это способ исключить "волшебные строки" при реализации INotifyPropertyChanged в различных платформах пользовательского интерфейса. В качестве примера:

C#

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName =
default!)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string? _name;
    public string? Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not needed here
explicitly
            }
        }
    }
}
```

В приведенном выше коде не требуется иметь литеральную `"Name"` строку. Использование `CallerMemberName` предотвращает ошибки, связанные с опечатками, а также обеспечивает более плавное рефакторингом и переименование. Атрибуты приносят декларативную силу в C#, но они являются формой метаданных кода и не действуют сами по себе.

Руководство. Обновление интерфейсов с помощью методов интерфейса по умолчанию

17.06.2025

Вы можете определить реализацию, когда объявляете члена интерфейса. Наиболее распространенный сценарий заключается в безопасном добавлении членов в интерфейс, который уже выпущен и используется ненумеримыми клиентами.

В этом руководстве описано, как:

- ✓ Безопасное расширение интерфейсов путем добавления методов с помощью реализаций.
- ✓ Создайте параметризованные реализации, чтобы обеспечить большую гибкость.
- ✓ Разрешите разработчикам предоставить более конкретную реализацию в виде переопределения.

Предпосылки

Необходимо настроить компьютер для запуска .NET, включая компилятор C#. Компилятор C# доступен с Visual Studio 2022 или пакета SDK для .NET.

Обзор сценария

В этом руководстве рассматривается версия 1 библиотеки управления отношениями с клиентами. Вы можете получить начальное приложение в репозитории примеров на сайте GitHub. Компания, которая создала эту библиотеку, намеревалась, чтобы клиенты с существующими приложениями приняли её библиотеку. Они предоставили минимальные определения интерфейса для пользователей своей библиотеки для реализации. Ниже приведено определение интерфейса для клиента:

C#

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
```

```
    IDictionary<DateTime, string> Reminders { get; }
}
```

Они определили второй интерфейс, представляющий заказ:

```
C#

public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

Из этих интерфейсов команда может создать библиотеку для своих пользователей, чтобы создать лучший интерфейс для своих клиентов. Их целью было создание более глубоких отношений с существующими клиентами и улучшение отношений с новыми клиентами.

Теперь пришло время обновить библиотеку для следующего выпуска. Одна из запрошенных функций обеспечивает скидку на лояльность для клиентов с большим количеством заказов. Эта новая скидка на лояльность применяется всякий раз, когда клиент делает заказ. Конкретная скидка — это свойство каждого отдельного клиента. Каждая реализация `ICustomer` может задавать разные правила для скидки на лояльность.

Самый естественный способ добавить эту функциональность — улучшить `ICustomer` интерфейс с помощью метода, чтобы применить любую скидку на лояльность. Это предложение по проектированию вызвало озабоченность среди опытных разработчиков: "Интерфейсы неизменяемы после того, как они были выпущены! Не делай критическое изменение!" Для обновления интерфейсов следует использовать реализации интерфейса по умолчанию. Авторы библиотеки могут добавлять новых членов в интерфейс и предоставлять реализацию по умолчанию для этих членов.

Реализации интерфейса по умолчанию позволяют разработчикам обновлять интерфейс, при этом предоставляя возможность всем тем, кто его реализует, переопределить данную реализацию. Пользователи библиотеки могут принять реализацию по умолчанию в качестве неисключаемого изменения. Если их бизнес-правила отличаются, они могут их изменить.

Обновление с помощью методов интерфейса по умолчанию

Команда согласилась с наиболее вероятной реализацией по умолчанию: скидка на лояльность для клиентов.

Обновление должно предоставить функциональные возможности для задания двух свойств: количество заказов, необходимых для получения скидки, и процент скидки. Эти функции делают его идеальным сценарием для методов интерфейса по умолчанию. Вы можете добавить метод в `ICustomer` интерфейс и предоставить наиболее вероятные реализации. Все существующие и любые новые реализации могут использовать реализацию по умолчанию или предоставить собственную.

Сначала добавьте новый метод в интерфейс, включая текст метода:

C#

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

Автор библиотеки написал первый тест, чтобы проверить реализацию:

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Обратите внимание на следующую часть теста:

C#

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Приведение от `SampleCustomer` к `ICustomer` необходимо. Классу `SampleCustomer` не нужно предоставлять реализацию для `ComputeLoyaltyDiscount`; это предоставляется интерфейсом `ICustomer` . `SampleCustomer` Однако класс не наследует элементы из его интерфейсов. Это правило не изменилось. Чтобы вызвать любой метод, объявленный и реализованный в интерфейсе, переменная должна быть типом интерфейса в `ICustomer` этом примере.

Укажите параметризацию

Реализация по умолчанию слишком ограничена. Многие потребители этой системы могут выбирать разные пороговые значения для количества покупок, другой длины членства или другой процентной скидки. Вы можете обеспечить более широкий интерфейс обновления для других клиентов, предоставив возможность задать эти параметры. Давайте добавим статический метод, который задает эти три параметра, управляющие реализацией по умолчанию:

C#

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

В этом фрагменте кода отображается множество новых возможностей языка. Теперь интерфейсы могут включать статические элементы, включая поля и методы. Также включены различные модификаторы доступа. Другие поля являются закрытыми, новый метод является открытым. Любой из модификаторов разрешен в членах интерфейса.

Приложения, использующие общую формулу для вычисления скидки на лояльность, но разные параметры, не должны предоставлять пользовательскую реализацию; они могут задать аргументы через статический метод. Например, следующий код задает "оценку клиентов", которая вознаграждает любого клиента с более чем одним месяцем членства:

```
C#

ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Расширение реализации по умолчанию

Добавленный до сих пор код предоставил удобную реализацию для тех сценариев, когда пользователи хотят что-то вроде реализации по умолчанию или предоставить несвязанный набор правил. В качестве завершающей возможности давайте немного рефакторим код, чтобы дать пользователям возможность разрабатывать сценарии на основе стандартной реализации.

Рассмотрим стартап, который хочет привлечь новых клиентов. Они предлагают 50% скидку от первого заказа нового клиента. В противном случае существующие клиенты получают стандартную скидку. Автор библиотеки должен переместить реализацию по умолчанию в `protected static` метод, чтобы любой класс, реализующий этот интерфейс, может повторно использовать код в их реализации. Реализация элемента интерфейса по умолчанию вызывает этот общий метод, а также:

```
C#

public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

В реализации класса, реализующего этот интерфейс, переопределение может вызвать статический вспомогательный метод и расширить эту логику, чтобы предоставить скидку "новый клиент":

C#

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

Полный код можно просмотреть в репозитории примеров на сайте GitHub. Вы можете получить начальное приложение в репозитории примеров на сайте GitHub.

Эти новые функции означают, что интерфейсы можно безопасно обновлять при наличии разумной реализации по умолчанию для этих новых членов. Тщательно проектируйте интерфейсы для выражения отдельных функциональных идей, реализованных несколькими классами. Это упрощает обновление определений этих интерфейсов при обнаружении новых требований для той же функциональной идеи.

Учебное пособие: Смешивание функциональных возможностей при создании классов с использованием интерфейсов с методами интерфейса по умолчанию

Вы можете определить реализацию, когда объявляете члена интерфейса. Эта функция предоставляет новые возможности, в которых можно определить реализации по умолчанию для функций, объявленных в интерфейсах. Классы могут выбирать, когда следует переопределять функциональные возможности, когда использовать функции по умолчанию и когда не объявлять поддержку дискретных функций.

В этом руководстве вы узнаете, как:

- ✓ Создайте интерфейсы с реализациями, описывающими дискретные функции.
- ✓ Создайте классы, использующие реализации по умолчанию.
- ✓ Создайте классы, которые переопределяют некоторые или все реализации по умолчанию.

Предпосылки

Необходимо настроить компьютер для запуска .NET, включая компилятор C#. Компилятор C# доступен с Visual Studio 2022 или пакета SDK для .NET.

Ограничения расширений

Одним из способов реализации поведения, которое отображается в интерфейсе, является определение элементов расширения , которые обеспечивают поведение по умолчанию. Интерфейсы объявляют минимальный набор элементов, предоставляя большую область поверхности для любого класса, реализующего этот интерфейс. Например, члены расширения в Enumerable предоставляют реализацию, позволяющую любой последовательности быть источником запроса LINQ.

Члены расширения разрешаются во время компиляции, используя объявленный тип переменной. Классы, реализующие интерфейс, могут обеспечить лучшую реализацию для любого члена расширения. Объявления переменных должны соответствовать типу реализации, чтобы компилятор мог выбрать реализацию. Когда тип времени компиляции соответствует интерфейсу, вызовы метода разрешаются члену расширения. Еще одна

проблема с членами расширения заключается в том, что они доступны везде, где доступен класс, содержащий их. Классы не могут объявлять, следует или не следует им предоставлять возможности, объявленные в членах расширения.

Реализации по умолчанию можно объявить как методы интерфейса. Затем каждый класс автоматически использует реализацию по умолчанию. Любой класс, который может обеспечить лучшую реализацию, может переопределить определение метода интерфейса с помощью лучшего алгоритма. В определенном смысле этот прием похож на то, как можно использовать члены расширения.

Из этой статьи вы узнаете, как реализации интерфейса по умолчанию позволяют создавать новые сценарии.

Проектирование приложения

Рассмотрим приложение домашней автоматизации. У вас, вероятно, есть много различных типов света и индикаторов, которые могут использоваться во всем доме. Каждый свет должен поддерживать API, чтобы включить и отключить их, а также сообщить о текущем состоянии. Некоторые световые индикаторы и сигнальные лампочки могут поддерживать другие функции, такие как:

- Включите свет, а затем отключите его после таймера.
- Мигать светом в течение определенного периода времени.

Некоторые из этих расширенных возможностей можно эмулировать на устройствах, поддерживающих минимальный набор. Это означает предоставление реализации по умолчанию. Для тех устройств, которые имеют больше возможностей, программное обеспечение устройства будет использовать собственные возможности. Для других осветительных приборов они могут выбрать внедрение интерфейса и воспользоваться реализацией по умолчанию.

Элементы интерфейса по умолчанию предоставляют лучшее решение для этого сценария, чем члены расширения. Авторы классов могут контролировать, какие интерфейсы они выбирают для реализации. Эти интерфейсы, которые они выбирают, доступны в качестве методов. Кроме того, поскольку методы интерфейса по умолчанию являются виртуальными по умолчанию, диспетчер метода всегда выбирает реализацию в классе.

Давайте создадим код для демонстрации этих различий.

Создание интерфейсов

Сначала создайте интерфейс, определяющий поведение для всех световых индикаторов:

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Базовый накладный светильник может реализовать этот интерфейс, как показано в следующем коде:

C#

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

В этом руководстве код не использует устройства Интернета вещей, но эмулирует эти действия, записывая сообщения в консоль. Вы можете изучить код без автоматизации вашего дома.

Далее давайте определим интерфейс для света, который может автоматически отключиться после истечения времени ожидания:

C#

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Вы можете добавить базовую реализацию в потолочный светильник, но лучшее решение — изменить это определение интерфейса, чтобы обеспечить `virtual` реализацию по умолчанию.

C#

```
public interface ITimerLight : ILight
{
```

```
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the
ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

Класс `OverheadLight` может реализовать функцию таймера, объявив поддержку интерфейса:

C#

```
public class OverheadLight : ITimerLight { }
```

Другой тип света может поддерживать более сложный протокол. Он может предоставить собственную реализацию для `TurnOnFor`, как показано в следующем коде:

C#

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

В отличие от переопределения методов виртуального `TurnOnFor` класса, объявление в `HalogenLight` классе не использует ключевое `override` слово.

Комбинирование и согласование возможностей

Преимущества методов интерфейса по умолчанию становятся более понятными при внедрении более сложных возможностей. Использование интерфейсов позволяет смешивать и сопоставлять возможности. Он также позволяет каждому автору класса выбирать между реализацией по умолчанию и пользовательской реализацией. Давайте добавим интерфейс с реализацией по умолчанию для мигающего света:

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
    }
}
```

Реализация по умолчанию позволяет любому свету мигать. Потолочный светильник может иметь функции таймера и мигания с помощью реализации по умолчанию.

C#

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Новый тип света, `LEDLight`, напрямую поддерживает как функцию таймера, так и функцию мигания. Этот светлый стиль реализует как интерфейсы `ITimerLight`, так и `IBlinkingLight`, и переопределяет метод `Blink`.

C#

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

`ExtraFancyLight` может напрямую поддерживать как функции мигания, так и таймера.

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Ранее созданный `HalogenLight` не поддерживает мигание. Поэтому не добавляйте этот элемент `IBlinkingLight` в список поддерживаемых интерфейсов.

Обнаружение типов света с помощью сопоставления шаблонов

Далее давайте напишем тестовый код. Вы можете использовать функцию сопоставления шаблонов C#, чтобы определить возможности света, проверив, какие интерфейсы она поддерживает. Следующий метод реализует поддерживаемые возможности каждого света:

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ? "on" :
"off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ? "on" :
"off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

Код в вашем методе `Main` создает каждый тип света по очереди и проверяет этот свет.

C#

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
```

```
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

Как компилятор определяет лучшую реализацию

В этом сценарии показан базовый интерфейс без каких-либо реализаций. Добавление метода в `ILight` интерфейс представляет новые сложности. Языковые правила, регулирующие методы интерфейса по умолчанию, минимизируют влияние на конкретные классы, реализующие несколько производных интерфейсов. Давайте расширим исходный интерфейс с помощью нового метода, чтобы показать, как это изменит его использование. Каждый индикатор может сообщать о состоянии питания в виде перечисленного значения:

C#

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

Реализация по умолчанию не предполагает никакой мощности:

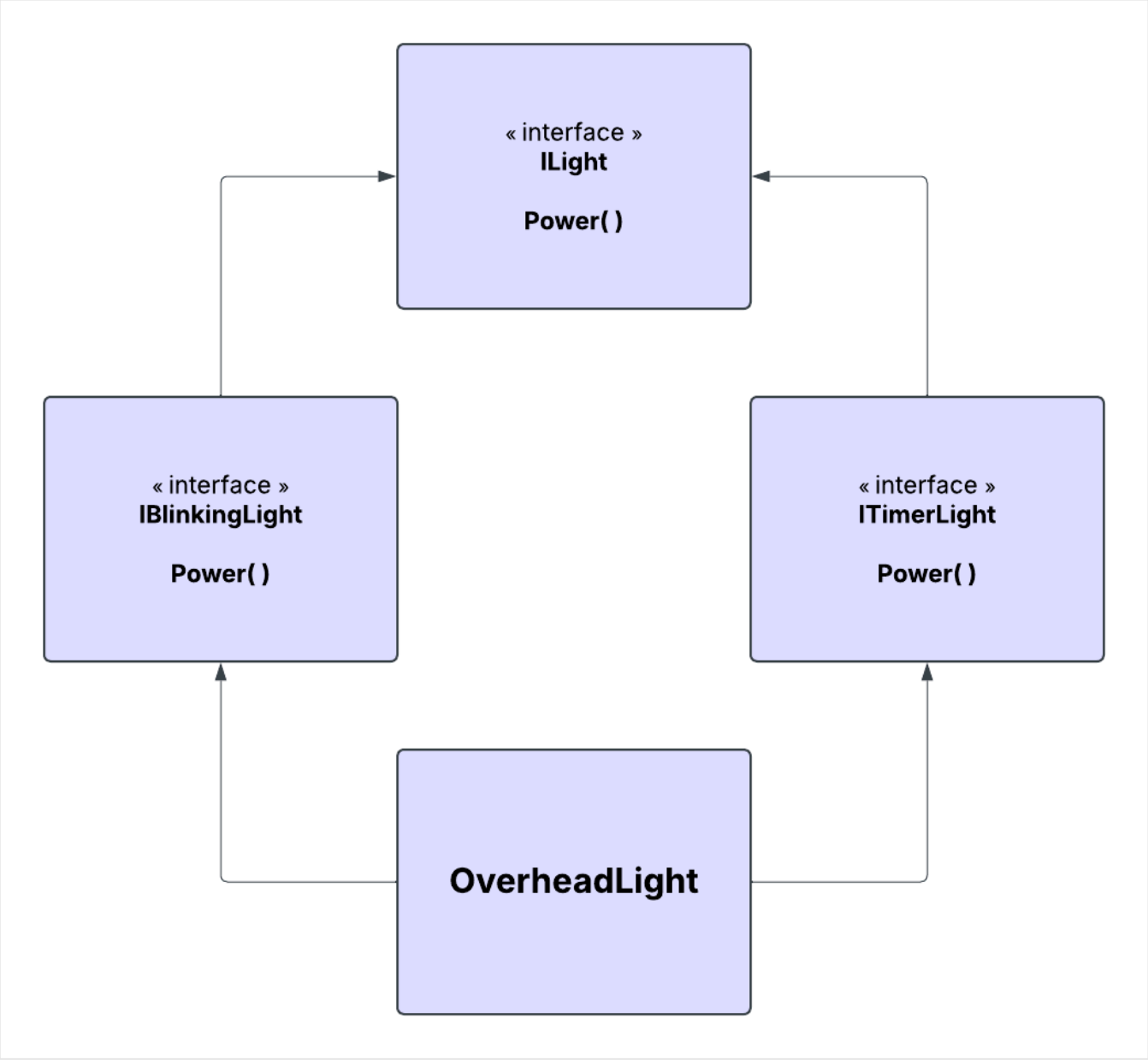
C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
```

```
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Эти изменения компилируются чисто, даже если `ExtraFancyLight` объявляет поддержку интерфейсу `ILight` и обоим производным интерфейсам, `ITimerLight` и `IBlinkingLight`. В интерфейсе `ILight` объявлена только одна "ближайшая" реализация. Любой класс, который задекларирует переопределение, станет "ближайшей" реализацией. Вы видели примеры в предыдущих классах, которые переопределяли элементы других интерфейсов, производных от базовых.

Избегайте переопределения одного метода в нескольких производных интерфейсах. Это создает неоднозначный вызов метода всякий раз, когда класс реализует оба производных интерфейса. Компилятор не может выбрать лучший метод, и поэтому выдает ошибку. Если `IBlinkingLight` и `ITimerLight` реализуют переопределение `Power()`, `OverheadLight` нужно предоставить более конкретное переопределение. В противном случае компилятор не может выбрать между реализацией в двух производных интерфейсах. Эта ситуация показана на следующей схеме:



На предыдущей схеме показана неоднозначность. `OverheadLight` не предоставляет реализацию `ILight.Power()`. И `IBlinkingLight`, и `ITimerLight` предоставляют переопределения, которые более конкретны. Вызов `ILight.Power()` к экземпляру `OverheadLight` является неоднозначным. Чтобы устранить неоднозначность, необходимо добавить новое переопределение в `OverheadLight`.

Как правило, эту ситуацию можно избежать, сохраняя определения интерфейса небольшими и ориентированными на одну функцию. В этом сценарии каждая функция света имеет свой отдельный интерфейс; только классы могут наследовать несколько интерфейсов.

В этом примере показан один сценарий, в котором можно определить дискретные функции, которые можно смешать в классы. Вы объявляете любой набор поддерживаемых функций, объявляя, какие интерфейсы поддерживают класс. Использование методов интерфейса по умолчанию позволяет классам использовать или определять другую реализацию для любого или всех методов интерфейса. Эта

возможность языка предоставляет новые способы моделирования реальных систем, которые вы создаете. Методы интерфейса по умолчанию предоставляют более четкий способ выражения связанных классов, которые могут смешивать и сопоставлять различные функции с помощью виртуальных реализаций этих возможностей.

Last updated on 24.11.2025

Руководство. Изучение статических виртуальных элементов в интерфейсах

Статические виртуальные элементы интерфейса позволяют определять интерфейсы, включающие перегруженные операторы или другие статические элементы. Определив интерфейсы со статическими элементами, эти интерфейсы можно использовать в качестве ограничений для создания универсальных типов, использующих операторы или другие статические методы. Даже если вы не создаете интерфейсы с перегруженными операторами, вы, скорее всего, используете эту функцию и универсальные математические классы, включенные обновлением языка.

В этом руководстве вы узнаете, как:

- ✓ Определите интерфейсы со статическими абстрактными и статическими виртуальными элементами.
- ✓ Используйте интерфейсы для определения классов, реализующих интерфейсы с определенными операторами.
- ✓ Предоставьте реализации по умолчанию, используя виртуальные статические методы.
- ✓ Создайте универсальные алгоритмы, использующие статические методы интерфейса.

Предпосылки

- Последняя версия .NET SDK
- Visual Studio Code редактор
- C# DevKit

Статические абстрактные методы интерфейса

Начнем с примера. Следующий метод возвращает середину двух `double` чисел:

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

Та же логика будет работать для любого числового типа: `int`, `short`, `long` или `float` `decimal` любого типа, представляющего число. Необходимо иметь способ использовать `+` операторы и `/` операторы, а также определить значение для `2`. Вы

можете использовать интерфейс System.Numerics.INumber<TSelf> для написания ранее упомянутого метода как следующего универсального метода:

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2);  // note: the
addition of left and right may overflow here; it's just for demonstration purposes
```

Любой тип, реализующий INumber<TSelf> интерфейс, должен включать определение для `operator +` и для `operator /`. Знаменатель определяется `T.CreateChecked(2)` для создания значения `2` для любого числового типа, что заставляет знаменатель совпадать с двумя параметрами. INumberBase<TSelf>.CreateChecked<TOther>(TOther) создает экземпляр типа из указанного значения и выбрасывает OverflowException, если значение выходит за пределы представляющего диапазона. (Эта реализация может привести к переполнению, если `left` и `right` оба являются достаточно большими значениями. Существуют альтернативные алгоритмы, которые могут избежать этой потенциальной проблемы.)

Вы определяете статические абстрактные члены в интерфейсе, используя знакомый синтаксис: добавьте модификатор `static` и `abstract` к любому статическому элементу, который не предоставляет реализацию. В следующем примере определяется интерфейс `IGetNext<T>`, который можно применить к любому типу, который переопределяет `operator ++`.

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

Ограничение, которое реализует аргумент типа, `T` гарантирует `IGetNext<T>` , что сигнатура для оператора включает содержащий тип или его аргумент типа. Многие операторы требуют, чтобы их параметры соответствовали типу или были параметрами типа, ограниченными реализацией содержащего типа. Без этого ограничения оператор `++` не может быть определен в интерфейсе `IGetNext<T>` .

Вы можете создать структуру, которая создает строку символов "A", где каждый шаг добавляет другой символ в строку с помощью следующего кода:

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
```

```
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

В целом можно создать любой алгоритм, в котором может потребоваться определить `++` значение "создать следующее значение этого типа". Использование этого интерфейса создает четкий код и результаты:

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

В предыдущем примере создаются следующие выходные данные:

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAAA
AAAAAAAAA
AAAAAAAAAA
```

В этом небольшом примере демонстрируется мотивация этой функции. Вы можете использовать естественный синтаксис для операторов, константных значений и других статических операций. Эти методы можно изучить при создании нескольких типов, использующих статические члены, включая перегруженные операторы. Определите интерфейсы, соответствующие возможностям типов, а затем объявите поддержку этих типов для нового интерфейса.

Методы статического виртуального интерфейса

В предыдущем примере `static abstract` использовался для объявления оператора `++`, и каждый тип реализации *должен* предоставлять собственную реализацию. Вы также можете объявить `static virtual` элементы, предоставляющие реализацию по умолчанию. Реализация типов может использовать значение по умолчанию или переопределить его. Это различие полезно, если существует разумное значение по умолчанию, которое работает для большинства типов, но некоторые типы нуждаются в специализированном поведении.

В следующем примере определяется интерфейс с свойством `IDescribable<T>`, `static abstract`, и методом `TypeName`, `static virtual`. Свойство `TypeName` абстрактно, так как каждый тип должен указать собственное имя. Метод `Describe()` является виртуальным, так как возврат имени типа является разумным выбором по умолчанию для большинства типов.

C#

```
public interface IDescribable<T> where T : IDescribable<T>
{
    static abstract string TypeName { get; }
    static virtual string Describe() => T.TypeName;
}
```

Тип, реализующий `IDescribable<T>`, должен предоставлять `TypeName`. Он может выбрать, следует ли переопределить `Describe()`. Тип `Widget` зависит от значения по умолчанию:

C#

```
public struct Widget : IDescribable<Widget>
{
    public static string TypeName => "Widget";

    // Uses the default Describe(): returns "Widget"
}
```

Тип `Gadget` переопределяет `Describe()` для добавления дополнительных сведений.

C#

```
public struct Gadget : IDescribable<Gadget>
{
    public static string TypeName => "Gadget";

    // Overrides the default Describe() with a custom message
    public static string Describe() => $"{TypeName} (version 2.0)";
}
```

Вы вызываете `static virtual` члены с помощью параметра типа в универсальном методе аналогично тому, как `static abstract` члены.

C#

```
static void PrintDescription<T>() where T : IDescribable<T>
{
    Console.WriteLine(T.Describe());
}

PrintDescription<Widget>();
PrintDescription<Gadget>();
```

В предыдущем примере создаются следующие выходные данные:

PowerShell

```
Widget
Gadget (version 2.0)
```

`Widget` использует реализацию `Describe()` по умолчанию, которая возвращает `TypeName`. `Gadget` переопределяет `Describe()` своей версией. Компилятор определяет правильную реализацию во время компиляции на основе аргумента типа. Таким образом, используйте `static abstract`, когда каждый тип должен иметь свою реализацию. Используйте `static virtual`, если вы можете предоставить полезное значение по умолчанию, которое будут принимать большинство типов.

Общая математика

Мотивирующий сценарий для разрешения статических методов, включая операторы, в интерфейсах — поддержка универсальных математических алгоритмов. Библиотека базовых классов .NET 7 содержит определения интерфейса для многих арифметических операторов и производные интерфейсы, которые объединяют многие арифметические операторы в интерфейсе `INumber<T>` . Давайте применим эти типы для создания записи, которая может использовать любой `Point<T>` числовой тип для `T`. Точку можно перемещать, используя оператор `XOffset`, по некоторым `YOffset` и `+`.

Начните с создания консольного приложения с помощью `dotnet new` или Visual Studio.

Общедоступный интерфейс для `Translation<T>` и `Point<T>` должен выглядеть следующим образом:

C#

```
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

Тип `record` используется для обоих типов `Translation<T>`: оба типа `Point<T>` хранят два значения, и они представляют собой хранилище данных, а не сложное поведение. Реализация `operator +` будет выглядеть следующим образом:

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

Чтобы предыдущий код `T` скомпилировался, необходимо объявить, что `IAdditionOperators<TSelf, TOther, TResult>` поддерживает интерфейс. Этот интерфейс включает статический `operator +` метод. Он объявляет три параметра типа: один для левого операнда, один для правого операнда и один для результата. Некоторые типы реализуют `+`, чтобы поддерживать различные типы операндов и результатов. Добавьте объявление, что аргумент типа `T` реализует `IAdditionOperators<T, T, T>`.

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

После добавления этого ограничения класс `Point<T>` может использовать `+` свой оператор сложения. Добавьте такое же ограничение к объявлению `Translation<T>`:

C#

```
public record Translation<T>(T XOffset, T YOffset) where T : IAdditionOperators<T,
T, T>;
```

Ограничение `IAdditionOperators<T, T, T>` запрещает разработчику вашего класса создавать `Translation`, использующий тип, который не соответствует ограничению для добавления к точке. Вы добавили необходимые ограничения в параметр типа `Translation<T>`, чтобы этот код `Point<T>` работал. Вы можете протестировать, добавив код, подобный следующему, над объявлениями `Translation` и `Point` в файле *Program.cs* :

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

Этот код можно сделать более повторно используемым, заявив, что эти типы реализуют соответствующие арифметические интерфейсы. Первое изменение, которое нужно сделать, это объявить, что `Point<T, T>` реализует интерфейс `IAdditionOperators<Point<T>, Translation<T>, Point<T>>`. Тип `Point` использует различные типы для операндов и результатов. Тип `Point` уже реализует `operator +` с этой сигнатурой, поэтому достаточно добавить интерфейс в объявление.

C#

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>,
Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Наконец, при выполнении сложения полезно иметь свойство, определяющее значение аддитивной единицы для этого типа. Существует новый интерфейс для этой функции: IAdditiveIdentity<TSelf,TResult> Перевод `{0, 0}` является аддитивной идентификацией: результирующая точка совпадает с левым операндом. Интерфейс `IAdditiveIdentity<TSelf, TResult>` определяет одно свойство только для чтения, `AdditiveIdentity`, которое возвращает идентификаторное значение. Для реализации этого интерфейса требуется внести несколько изменений в `Translation<T>`.

C#

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
}
```

Здесь есть несколько изменений, поэтому давайте рассмотрим их по одному. Сначала вы объявляете, что `Translation` тип реализует `IAdditiveIdentity` интерфейс:

C#

```
public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
```

Далее можно попробовать реализовать элемент интерфейса, как показано в следующем коде:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

Предыдущий код не компилируется, так как `0` зависит от типа. Ответ: используйте `IAdditiveIdentity<T>.AdditiveIdentity` для `0`. Это изменение означает, что ваши ограничения теперь должны предусматривать, что `T` реализует `IAdditiveIdentity<T>`. Это приводит к следующей реализации:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset: T.AdditiveIdentity);
```

Теперь, когда вы добавили это ограничение, необходимо добавить то же ограничение `Translation<T>` в `Point<T>`:

C#

```
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>,
Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right) =>
        left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
}
```

В этом примере вы узнаете, как составляются интерфейсы для генерической математики. Вы узнали, как:

- ✓ Напишите метод, зависящий от `INumber<T>` интерфейса, чтобы этот метод мог использоваться с любым числовым типом.
- ✓ Создайте тип, основанный на интерфейсах сложения для реализации типа, поддерживающего только одну математические операции. Этот тип объявляет поддержку тех же интерфейсов, чтобы его можно было компоновать другими способами. Алгоритмы записываются с помощью самого естественного синтаксиса математических операторов.

Поэкспериментируйте с этими функциями и зарегистрируйте отзывы. Вы можете использовать пункт меню "*Отправить отзыв*" в Visual Studio или создать новую проблему в репозитории roslyn на сайте GitHub. Создание универсальных алгоритмов, работающих с любым числовым типом. Создание алгоритмов с использованием этих интерфейсов, где аргумент типа реализует только подмножество числовых возможностей. Даже если вы не создаете новые интерфейсы, использующие эти возможности, вы можете экспериментировать с ними в алгоритмах.

См. также

- Универсальная математика

Last updated on 14.03.2026

Деревья выражений

13.10.2025

Деревья выражений представляют код в структуре данных, например в виде дерева, где каждый узел является выражением, например вызов метода или двоичная операция, например `x < y`.

Если вы использовали LINQ, у вас есть опыт работы с богатой библиотекой, в которой `Func` типы являются частью набора API. (Если вы не знакомы с LINQ, вероятно, хотите прочитать руководство по LINQ и статью о лямбда-выражениях перед этим. Деревья выражений обеспечивают более расширенное взаимодействие с аргументами, которые являются функциями.

При написании аргументов функций вы обычно используете лямбда-выражения при создании запросов LINQ. В типичном запросе LINQ эти аргументы функции преобразуются в делегат, который создает компилятор.

Вы уже пишете код, использующий деревья выражений. API LINQ Entity Framework принимают деревья выражений в качестве аргументов для шаблона выражения запроса LINQ. Это позволяет Entity Framework переводить запрос, написанный на C# в SQL, который выполняется в ядре СУБД. Другим примером является Moq, который является популярной платформой макетирования для .NET.

Если требуется более расширенное взаимодействие, необходимо использовать *деревья выражений*. Деревья выражений представляют код как структуру, которую вы проверяете, изменяете или выполняете. Эти средства позволяют управлять кодом во время выполнения. Вы пишете код, который проверяет выполнение алгоритмов или внедряет новые возможности. В более сложных сценариях можно изменять выполняемые алгоритмы и даже переводить выражения C# в другую форму для выполнения в другой среде.

Вы компилируете и запускаете код, представленный деревьями выражений. Создание и выполнение деревьев выражений обеспечивает динамическое изменение исполняемого кода, выполнение запросов LINQ в различных базах данных и создание динамических запросов. Дополнительные сведения о деревьях выражений в LINQ см. в статье "Использование деревьев выражений для создания динамических запросов".

Деревья выражений также используются в среде исполнения динамических языков (DLR) для обеспечения взаимодействия между динамическими языками и .NET, а также чтобы компиляторы могли генерировать деревья выражений вместо промежуточного языка Microsoft (CIL). Дополнительные сведения о DLR см. в разделе "Обзор динамической языковой среды выполнения".

Вы можете поручить компилятору C# или Visual Basic создать для вас дерево выражений на основе анонимного лямбда-выражения или создать деревья выражений вручную с помощью пространства имен System.Linq.Expressions.

Когда лямбда-выражение назначается переменной типа Expression<TDelegate>, компилятор выдает код для создания дерева выражений, представляющего лямбда-выражение.

В следующих примерах кода показано, как создать дерево выражений C#, представляющее лямбда-выражение `num => num < 5`.

```
C#

Expression<Func<int, bool>> lambda = num => num < 5;
```

В коде создаются деревья выражений. Дерево создается путем создания каждого узла и присоединения узлов к структуре дерева. Вы узнаете, как создавать выражения в статье о создании деревьев выражений.

Деревья выражений неизменяемы. Если вы хотите изменить дерево выражений, необходимо создать новое дерево выражений путем копирования существующего и замены узлов в нем. Вы используете посетителя дерева выражений для обхода существующего дерева выражений. Дополнительные сведения см. в статье о переводе деревьев выражений.

После создания дерева выражений выполните код, представленный деревом выражений.

Ограничения

Компилятор C# создает деревья выражений только из выражений-лямбда и однострочных лямбда-функций. Невозможно разобрать лямбда-выражения (или многострочные лямбда-выражения). Дополнительные сведения о лямбда-выражениях в C#см. в лямбда-выражениях.

Существуют некоторые новые элементы языка C#, которые не преобразуются в деревья выражений. Деревья выражений не могут содержать `await` выражения или `async` лямбда-выражения. Многие функции, добавленные в C# 6 и более поздних версий, не отображаются точно так же, как и в деревьях выражений. Вместо этого новые функции предоставляются в деревах выражений в эквивалентном, более раннем синтаксисе, где это возможно. Остальные конструкции недоступны. Это означает, что код, который интерпретирует деревья выражений, работает так же, как при появлении новых возможностей языка. Однако даже с этими ограничениями деревья выражений

позволяют создавать динамические алгоритмы, основанные на интерпретации и изменении кода, представленного как структура данных. Она позволяет расширенным библиотекам, таким как Entity Framework, выполнять то, что они делают.

Деревья выражений не поддерживают новые типы узлов выражений. Это было бы критическое изменение для всех библиотек, интерпретирующих деревья выражений, чтобы ввести новые типы узлов. В следующем списке содержатся большинство элементов языка C#, которые нельзя использовать:

- Условные методы удалены из выходных данных
- base доступ
- Выражения групп методов, включая *адрес*& () группу методов и анонимные выражения методов
- Ссылки на локальные функции
- Операторы, включая присвоение (`=`) и выражения телесного выражения оператора
- Частичные методы только с объявлением определения
- Небезопасные операции указателя
- dynamic операции
- Операторы объединения с null или default буквальной левой частью, оператор присваивания объединения null и оператор распространения null (?.)
- Инициализаторы многомерного массива, индексированные свойства и инициализаторы словаря
- коллекционные выражения
- throw Выражения
- Доступ к членам static virtual или abstract интерфейса
- Лямбда-выражения с атрибутами
- Интерполированные строки
- Преобразования строк UTF-8 или строковые литералы UTF-8
- Вызовы методов с помощью аргументов переменной, именованных аргументов или необязательных аргументов
- Выражения, использующие System.Index или System.Range, оператор индекса "с конца" (^) или выражения диапазона (..)
- async лямбда-выражения или await выражения, включая await foreach и await using
- Кортежные литералы, преобразования кортежей, кортеж == или !=, или with выражения
- Отбрасывание (_), деконструктивное присваивание, оператор сопоставления по is шаблону или выражение сопоставления по switch шаблону
- ВЫЗОВ COM с `ref` опущенными аргументами
- ref, in или out параметры, возвращаемые значения, `ref` `out` аргументы или любые значения ref struct типа

Деревья выражений — данные, определяющие код

17.06.2025

Дерево выражений — это структура данных, которая определяет код. Деревья выражений основаны на той же структуре, что компилятор использует для анализа кода и создания скомпилированных выходных данных. При чтении этой статьи вы заметили довольно много сходства между деревьями выражений и типами, используемыми в API Roslyn для создания анализаторов и CodeFixes. (Анализаторы и CodeFixes — это пакеты NuGet, которые выполняют статический анализ кода и предлагают потенциальные исправления для разработчика.) Основные понятия аналогичны, и конечный результат — это структура данных, которая позволяет проверять исходный код в понятном виде. Однако деревья выражений основаны на другом наборе классов и API, отличных от API Roslyn. Ниже приведена строка кода:

```C#
var sum = 1 + 2;
```

При анализе предыдущего кода в виде дерева выражений дерево содержит несколько узлов. Самый внешний узел — это оператор объявления переменной с назначением (`var sum = 1 + 2;`) Тот внешний узел содержит несколько дочерних узлов: объявление переменной, оператор назначения и выражение, представляющее правую сторону знака равенства. Это выражение также подразделяется на выражения, представляющие операцию добавления, а также левые и правые операнды добавления.

Давайте рассмотрим немного больше выражений, составляющих правую сторону знака равенства. Выражение — `1 + 2`двоичное выражение. В частности, это двоичное выражение сложения. Двоичное выражение сложения содержит два дочерних элемента, представляющих левые и правые узлы выражения сложения. Здесь оба узла являются константными выражениями: левый операнд является значением `1`, а правый операнд — значение `2`.

Визуально все утверждение является деревом: вы можете начать с корневого узла и перейти к каждому узлу в этом дереве, чтобы увидеть код, составляющий утверждение.

- Оператор объявления переменной с присвоением (`var sum = 1 + 2;`)
  - Объявление типа неявной переменной (`var sum`)
    - Неявное ключевое слово var (`var`)
    - Объявление имени переменной (`sum`)

- Оператор назначения (`=`)
- Двоичное выражение сложения (`1 + 2`)
  - Левый операнд (`1`)
  - Оператор сложения (`+`)
  - Правый операнд (`2`)

Предыдущее дерево может выглядеть сложно, но это очень мощно. Следуя тому же процессу, вы разложите намного более сложные выражения. Рассмотрим это выражение:

```
C#
```

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

Предыдущее выражение также является объявлением переменной с назначением. В этом случае правая часть присваивания является гораздо более сложным деревом. Вы не собираетесь разложить это выражение, но рассмотрите, какие узлы могут быть разными. Существуют вызовы методов, где текущий объект используется в качестве приемника: один имеет явный `this` приемник, а другой - нет. Существуют вызовы методов с помощью других объектов-получателей, существуют постоянные аргументы различных типов. И, наконец, есть двоичный оператор сложения. В зависимости от типа возвращаемого значения `SecretSauceFunction()` или `MoreSecretSauce()`, этот оператор двоичного сложения может быть вызовом метода переопределенного оператора сложения, который затем разрешается в статический вызов оператора двоичного сложения, определенного для класса.

Несмотря на эту воспринимаемую сложность, предыдущее выражение создает структуру дерева, перемещаемую так же легко, как в первом примере. Вы продолжаете проходить по дочерним узлам, чтобы находить листовые узлы в выражении. Узлы-родители имеют ссылки на дочерние узлы, и каждый узел имеет свойство, описывающее тип узла.

Структура дерева выражений очень согласована. После того как вы узнали основы, вы понимаете даже самый сложный код, когда он представлен как дерево выражений. Элегантность в структуре данных объясняет, как компилятор C# анализирует самые сложные программы C# и создает правильные выходные данные из этого сложного исходного кода.

Когда вы ознакомитесь со структурой деревьев выражений, вы обнаружите, что полученные знания быстро позволяют работать с множеством более сложных сценариев. Деревья выражений обладают невероятной мощью.

Помимо преобразования алгоритмов для выполнения в других средах, деревья выражений упрощают написание алгоритмов, которые проверяют код перед выполнением кода. Вы пишете метод, аргументы которого являются выражениями, и перед выполнением кода изучаете эти выражения. Дерево выражений представляет собой полное представление кода: вы видите значения любого подэкспрессии. Вы видите имена методов и свойств. Отображается значение любых константных выражений. Вы преобразуете дерево выражений в исполняемый делегат и выполняете код.

API-интерфейсы для деревьев выражений позволяют создавать деревья, которые представляют практически любую допустимую конструкцию кода. Тем не менее, чтобы сохранить вещи как можно проще, некоторые идиомы C# не могут быть созданы в дереве выражений. Одним из примеров являются асинхронные выражения (с помощью ключевых слов `async` и `await`). Если вашим потребностям требуются асинхронные алгоритмы, необходимо напрямую управлять `Task` объектами, а не полагаться на поддержку компилятора. Другим является создание циклов. Как правило, вы создаёте эти циклы, используя циклы `for`, `foreach`, `while` или `do`. Как показано далее в этой серии, API-интерфейсы для деревьев выражений поддерживают одно циклическое выражение и `break` `continue` выражения, которые управляют повторяющимся циклом.

Единственное, что нельзя сделать, — изменить дерево выражений. Деревья выражений — это неизменяемые структуры данных. Если вы хотите изменить (изменить) дерево выражений, необходимо создать новое дерево, которое является копией исходного, но с нужными изменениями.

Поддержка среды выполнения .NET для деревьев выражений

17.06.2025

Существует большой список классов в среде выполнения .NET, которые работают с деревьями выражений. Полный список можно увидеть на System.Linq.Expressions. Вместо перечисления полного списка давайте посмотрим, как были разработаны классы среды выполнения.

В структуре языка выражение — это текст кода, который вычисляет и возвращает значение. Выражения могут быть простыми: константное выражение `1` возвращает константное значение 1. Они могут быть более сложными: выражение `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` возвращает один корень для квадратного уравнения (в случае, если уравнение имеет решение).

System.Linq.Expression и производные типы

Одной из сложностей работы с деревьями выражений является то, что многие различные виды выражений допустимы во многих местах в программах. Рассмотрим выражение назначения. Правая сторона операции присваивания может быть константным выражением, переменной, выражением вызова метода или другими выражениями. Эта гибкость языка означает, что при переходе по дереву выражений можно столкнуться с множеством различных типов выражений. Поэтому при работе с типом базового выражения это самый простой способ работы. Однако иногда вам нужно знать больше. Базовый `NodeType` класс Expression содержит свойство для этой цели. Возвращает значение `ExpressionType`, которое представляет собой перечисление возможных типов выражений. После того как вы знаете тип узла, вы приведете его к такому типу и выполните определенные действия, зная тип узла выражения. Вы можете искать определенные типы узлов, а затем работать с определенными свойствами этого типа выражения.

Например, этот код выводит имя переменной для выражения доступа к переменной. В следующем коде показана практика проверки типа узла, а затем приведение к выражению доступа к переменной, а затем проверка свойств конкретного типа выражения:

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;
```

```
if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0];  -- first

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Создание деревьев выражений

Класс `System.Linq.Expression` также содержит множество статических методов для создания выражений. Эти методы создают узел выражения с помощью аргументов, предоставленных для дочерних элементов. Таким образом, вы создаете выражение из его листовых узлов. Например, этот код создает выражение Add:

C#

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

В этом простом примере можно увидеть, что многие типы участвуют в создании и работе с деревьями выражений. Эта сложность необходима для предоставления возможностей богатого словаря, предоставляемого языком C#.

Навигация по API

Существуют типы узлов expression, которые сопоставляются почти со всеми элементами синтаксиса языка C#. Каждый тип имеет определенные методы для этого типа элемента языка. Это много, чтобы держать в голове сразу. Вместо того чтобы попытаться запомнить все, вот методы, которые вы используете для работы с деревьями выражений:

1. Просмотрите элементы перечисления `ExpressionType` , чтобы определить возможные узлы, которые следует проверить. Этот список помогает, когда вы хотите обойти и понять дерево выражений.
2. Просмотрите статические члены `Expression` класса, чтобы создать выражение. Эти методы могут создавать любой тип выражения из набора дочерних узлов.
3. Просмотрите `ExpressionVisitor` класс для создания измененного дерева выражений.

Вы найдете больше, как вы посмотрите на каждую из этих трех областей. Неизменно, вы найдете то, что вам нужно, когда вы начинаете с одного из этих трех шагов.

Выполнение деревьев выражений

17.06.2025

Дерево выражений — это структура данных, представляющая некоторый код. Это не компилированный и исполнимый код. Если вы хотите выполнить код .NET, представленный деревом выражений, необходимо преобразовать его в исполняемые инструкции IL. Выполнение дерева выражений может возвращать значение или просто выполнять действие, например вызов метода.

Можно выполнять только деревья выражений, представляющие лямбда-выражения. Деревья выражений, представляющие лямбда-выражения, имеют тип LambdaExpression или Expression<TDelegate>. Чтобы выполнить эти деревья выражений, вызовите метод Compile для создания исполняемого делегата, а затем выполните вызов делегата.

> **Примечание**
>
> Если тип делегата не известен, то есть лямбда-выражение имеет тип LambdaExpression, а не Expression<TDelegate>, вызовите метод DynamicInvoke на делегате, вместо того чтобы вызывать его напрямую.

Если дерево выражений не представляет лямбда-выражение, можно создать новое лямбда-выражение, которое имеет исходное дерево выражений в качестве его тела, вызвав Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>) метод. Затем можно выполнить лямбда-выражение, как описано ранее в этом разделе.

Лямбда-выражения для функций

Вы можете преобразовать любое LambdaExpression или любой тип, производный от LambdaExpression, в выполнимый IL. Другие типы выражений нельзя напрямую преобразовать в код. Это ограничение на практике мало влияет. Лямбда-выражения — это единственные типы выражений, которые требуется выполнить путем преобразования в исполняемый промежуточный язык (IL). (Подумайте о том, что означало бы прямое выполнение System.Linq.Expressions.ConstantExpression. Было бы это чем-то полезным?) Любое дерево выражений, которое является System.Linq.Expressions.LambdaExpression или типом, производным от `LambdaExpression`, может быть преобразовано в IL. Тип System.Linq.Expressions.Expression<TDelegate> выражения является единственным конкретным примером в библиотеках .NET Core. Он используется для представления выражения, которое сопоставляется с любым типом делегата. Так как этот тип сопоставляется с типом делегата, .NET может проверить выражение и создать IL для соответствующего делегата, который соответствует подписи лямбда-выражения. Тип

делегата основан на типе выражения. Необходимо знать тип возвращаемого значения и список аргументов, если требуется использовать объект делегата строго типизированным образом. Метод `LambdaExpression.Compile()` возвращает `Delegate` тип. Для проверки списка аргументов или типа возврата при помощи средств контроля времени компиляции необходимо привести его к правильному типу делегата.

В большинстве случаев существует простое сопоставление между выражением и соответствующим делегатом. Например, дерево выражений, представленное `Expression<Func<int>>`, преобразуется в делегат типа `Func<int>`. Для лямбда-выражения с любым возвращаемым типом и списком аргументов существует тип делегата, который является целевым типом исполняемого кода, представленного этим лямбда-выражением.

Тип System.Linq.Expressions.LambdaExpression содержит LambdaExpression.Compile и LambdaExpression.CompileToMethod элементы, которые будут использоваться для преобразования дерева выражений в исполняемый код. Метод `Compile` создает делегат. Метод `CompileToMethod` обновляет объект System.Reflection.Emit.MethodBuilder с помощью IL, который представляет собой скомпилированный результат дерева выражений.

> ⓘ **Важно!**
>
> `CompileToMethod` доступен только в .NET Framework, а не в .NET Core или .NET 5 и более поздних версиях.

При желании вы также можете указать System.Runtime.CompilerServices.DebugInfoGenerator, который получает сведения об отладке символов для созданного объекта делегата. Элемент `DebugInfoGenerator` предоставляет полную информацию об отладке созданного делегата.

Выражение преобразуется в делегат с помощью следующего кода:

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

В следующем примере кода показаны конкретные типы, используемые при компиляции и выполнении дерева выражений.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

В следующем примере кода показано, как выполнить дерево выражений, представляющее возведение числа в степень, создав лямбда-выражение и выполнив его. Отображается результат, представляющий число, возведенное в степень.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Выполнение и время существования

Код выполняется путем вызова делегата, созданного при вызове `LambdaExpression.Compile()`. Предыдущий код `add.Compile()` возвращает делегат. Этот делегат вызывается путем вызова `func()`, который выполняет код.

Этот делегат представляет код в дереве выражений. Вы можете сохранить хэндл для этого делегата и вызвать его в будущем. Не нужно компилировать дерево выражений при каждом выполнении кода, который он представляет. (Помните, что деревья выражений неизменяемы, и компиляция того же дерева выражений позже создает делегат, который выполняет тот же код.)

> ⊗ **Внимание!**
>
> Не создавайте более сложные механизмы кэширования для повышения производительности, избегая ненужных вызовов компиляции. Сравнение двух произвольных деревьев выражений для определения того, являются ли они одинаковым алгоритмом операцией, требующей много времени. Время вычислений, которое вы экономите, избегая дополнительных вызовов `LambdaExpression.Compile()`, скорее всего, больше, чем время выполнения кода, которое определяет, приводит ли два разных дерева выражений к одному исполняемому коду.

Предупреждения

Компиляция лямбда-выражения в делегат и вызов этого делегата являются одними из самых простых операций, которые можно выполнить в дереве выражений. Однако даже с этой простой операцией есть предостережение, о которых вы должны знать.

Лямбда-выражения создают закрытия для любых локальных переменных, на которые ссылается выражение. Необходимо гарантировать, что все переменные, которые будут частью делегата, доступны в среде, где вы вызываете `Compile`, а также при выполнении полученного делегата. Компилятор гарантирует, что переменные находятся в области видимости. Однако если выражение обращается к переменной, реализующей `IDisposable`, возможно, что код может удалить объект, пока он все еще находится в дереве выражений.

Например, этот код работает хорошо, так как `int` не реализует `IDisposable`:

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

Делегат захватил ссылку на локальную переменную `constant`. Эта переменная используется в любое время позже, когда исполнится возвращаемая функция `CreateBoundFunc`.

Однако рассмотрим следующий довольно надуманный класс, который реализует System.IDisposable:

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

Если вы используете его в выражении, как показано в следующем коде, вы получите System.ObjectDisposedException, когда выполните код, на который ссылается свойство `Resource.Argument`.

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

Делегат, возвращаемый этим методом, закрыл `constant` объект, из которого был удален. (Это было удалено, потому что оно было объявлено в `using` заявлении.)

Теперь при выполнении делегата, возвращаемого из этого метода, возникает `ObjectDisposedException` исключение в точке выполнения.

Кажется странным, что ошибка во время выполнения представляет собой конструкцию времени компиляции, но это мир, в который вы вступаете при работе с деревьями выражений.

Существует множество вариаций этой проблемы, поэтому трудно предложить общее руководство, как их избежать. Будьте внимательны к доступу к локальным переменным при определении выражений и будьте осторожны с доступом к состоянию в текущем объекте (представленном `this`) при создании дерева выражений, возвращаемого через общедоступный API.

Код в выражении может ссылать на методы или свойства в других сборках. Эта сборка должна быть доступна при определении выражения, при компиляции и при вызове результирующего делегата. Вы будете сталкиваться с `ReferencedAssemblyNotFoundException`, если он отсутствует в системе.

Сводка

Деревья выражений, представляющие лямбда-выражения, можно скомпилировать для создания делегата, который можно выполнить. Деревья выражений предоставляют один механизм для выполнения кода, представленного деревом выражений.

Дерево выражений представляет код, который будет выполняться для любой создаваемой конструкции. Если среда, в которой выполняется компиляция и выполнение кода, соответствует среде, в которой создается выражение, все работает должным образом. Когда этого не происходит, ошибки предсказуемы, и они перехватываются в первых тестах любого кода, использующего деревья выражений.

Интерпретация выражений

Статья • 08.04.2025

В следующем примере кода показано, как дерево выражений, представляющее лямбда-выражение `num => num < 5` , может быть разложено на его части.

C#

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine($"Decomposed expression: {param.Name} => {left.Name}
{operation.NodeType} {right.Value}");

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Теперь давайте напишем код, чтобы изучить структуру *дерева выражений*. Каждый узел в дереве выражений — это объект класса, производный от `Expression`.

Эта конструкция делает посещение всех узлов в дереве выражений относительно простой рекурсивной операцией. Общая стратегия — начать с корневого узла и определить тип узла.

Если у типа узла есть дочерние элементы, рекурсивно обойдите каждый дочерний элемент. На каждом дочернем узле повторите процесс, используемый на корневом узле: определите тип, а если тип имеет дочерних элементов, посетите каждый из дочерних элементов.

Проверка выражения без дочерних элементов

Начнем с посещения каждого узла в простом дереве выражений. Ниже приведен код, который создает константное выражение, а затем проверяет его свойства:

C#

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

Предыдущий код выводит следующие выходные данные:

Выходные данные

```
This is a/an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Теперь давайте напишем код, который будет изучать это выражение и записывать в него некоторые важные свойства.

Выражение сложения

Давайте начнем с примера добавления из введения к этому разделу.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

> ⓘ **Примечание**
>
> Не используйте `var` для объявления этого дерева выражений, так как естественный тип делегата `Func<int>`, а не `Expression<Func<int>>`.

Корневой узел — это `LambdaExpression`. Чтобы получить интересный код в правой части `=>` оператора, необходимо найти один из дочерних элементов `LambdaExpression`. Это можно сделать со всеми выражениями в этом разделе. Родительский узел действительно помогает найти тип возвращаемого значения `LambdaExpression`.

Чтобы проверить каждый узел в этом выражении, необходимо рекурсивно посетить множество узлов. Ниже приведена простая первая реализация:

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;
```

```
Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "<null>"
: addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They
are:");
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type: {argumentExpression.Type.ToString()},
Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}");
```

Этот пример выводит следующие выходные данные:

Выходные данные

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

Вы заметили много повторений в предыдущем примере кода. Давайте приведем в порядок и создадим посетителя узлов выражений общего назначения. Это потребует от нас написания рекурсивного алгоритма. Любой узел может быть типом с возможностью иметь дочерние элементы. Любой узел с дочерними элементами требует, чтобы мы посетили эти элементы и идентифицировали, каким является этот узел. Вот упрощённая версия, которая использует рекурсию для обработки операций сложения.

C#

```
using System.Linq.Expressions;
```

```
namespace Visitors;
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression
type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name ==
null) ? "<null>" : node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count}
argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
```

```
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node) => this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType}
expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name}, ByRef:
{node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
    }
}
```

Этот алгоритм служит основой для посещения любого произвольного `LambdaExpression`. Созданный код ищет только небольшой пример возможных наборов узлов дерева

выражений, с которыми он может столкнуться. Тем не менее, вы по-прежнему можете узнать довольно много от того, что он производит. (По умолчанию в `Visitor.CreateFromExpression` методе выводится сообщение в консоль ошибок при обнаружении нового типа узла. Таким образом, вы знаете, чтобы добавить новый тип выражения.)

При запуске этого объекта посещения в ранее упомянутом выражении сложения вы получите следующий результат.

Выходные данные

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Теперь, когда вы создали более общую реализацию посетителей, вы можете посетить и обработать гораздо больше различных типов выражений.

Выражение сложения с большим количеством операндов

Давайте рассмотрим более сложный пример, но по-прежнему ограничиваем типы узлов только для добавления:

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Перед тем как запустить эти примеры на алгоритме посетителя, попробуйте представить себе, каким может быть результат. Помните, что `+` оператор является *двоичным оператором*: он должен иметь два дочерних элемента, представляющих левые и правые

операнды. Существует несколько возможных способов создания дерева, которое может быть правильным:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Вы можете увидеть разделение на два возможных ответа, чтобы выделить наиболее перспективные. Первый представляет *правильные ассоциативные* выражения. Второй представляет *левые ассоциативные* выражения. Преимущество обоих этих двух форматов заключается в том, что формат масштабируется до любого произвольного числа выражений сложения.

Если вы выполняете это выражение через посетителя, вы увидите эти выходные данные, убедившись, что простое выражение сложения *остается ассоциативным*.

Чтобы запустить этот пример, и просмотреть дерево полного выражения, необходимо внести одно изменение в дерево исходного выражения. Если дерево выражений содержит все константы, результирующее дерево просто содержит постоянное значение `10`. Компилятор выполняет все добавление и уменьшает выражение до самой простой формы. Простое добавление одной переменной в выражение достаточно, чтобы увидеть исходное дерево:

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Создайте посетителя для этой суммы и запустите посетителя, чтобы увидеть следующий вывод:

Выходные данные

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
```

```
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

Вы можете запустить любой из других примеров с помощью кода визитора и посмотреть, какое дерево оно представляет. Ниже приведен пример предыдущего `sum3` выражения (с дополнительным параметром, чтобы предотвратить вычисление константы компилятором):

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Ниже приведены выходные данные посетителя:

Выходные данные

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
```

```
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Обратите внимание, что скобки не являются частью выходных данных. В дереве выражений нет узлов, представляющих скобки в входном выражении. Структура дерева выражений содержит всю информацию, необходимую для передачи порядка выполнения операций.

Расширение этого примера

В этом примере рассматриваются только самые рудиментарные деревья выражений. Код, который вы видели в этом разделе, обрабатывает только целые числа констант и двоичный `+` оператор. В качестве последнего примера давайте изменим способ обработки посетителем более сложного выражения. Давайте применим его к следующему факториальному выражению:

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Этот код представляет собой одну из возможных реализаций математической *факториальной* функции. Способ написания этого кода выделяет два ограничения для создания деревьев выражений путем назначения лямбда-выражений выражениям. Во-первых, лямбда-выражения инструкции не допускаются. Это означает, что вы не можете использовать циклы, блоки, операторы if / else и другие структуры управления, распространенные в C#. Вы ограничены использованием выражений. Во-вторых, вы не можете рекурсивно вызывать то же выражение. Можно было бы, если бы он уже был делегатом, но вы не можете вызвать его в виде дерева выражений. В разделе о создании деревьев выражений вы узнаете, как преодолеть эти ограничения.

В этом выражении встречаются узлы всех этих типов.

1. Равно (двоичное выражение)
2. Умножение (двоичное выражение)

3. Условное выражение `? :`
4. Выражение вызова метода (вызовы `Range()` и `Aggregate()`)

Одним из способов изменения алгоритма посетителя является сохранение его выполнения и запись типа узла при каждом достижении `default` предложения. После нескольких итераций вы увидите каждый из потенциальных узлов. Тогда у вас есть все, что вам нужно. Результат будет примерно таким:

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant    => new
ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda      => new LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter   => new
ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add         => new BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal       => new BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply    => new BinaryVisitor((BinaryExpression) node),
        ExpressionType.Conditional => new
ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call        => new MethodCallVisitor((MethodCallExpression)
node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };
```

Процесс `ConditionalVisitor` и `MethodCallVisitor` обрабатывают эти два узла.

C#

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
```

```
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.
{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

И выходные данные для дерева выражений будут следующими:

Выходные данные

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
```

```
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef: False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef: False
                        The expression body is:
                                This binary expression is a Multiply expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product, ByRef:
False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor, ByRef:
False
```

Расширение библиотеки примеров

В примерах в этом разделе показаны основные методы посещения и проверки узлов в дереве выражений. Это упрощает типы узлов, с которыми вы столкнетесь, чтобы сосредоточиться на основных задачах посещения и доступа к узлам в дереве выражений.

Во-первых, посетители обрабатывают только константы, которые являются целыми числами. Константные значения могут быть любым другим числовым типом, а язык C# поддерживает преобразования и повышение между этими типами. Более надежная версия этого кода отражает все эти возможности.

Даже в последнем примере распознается подмножество возможных типов узлов. Вы все еще можете привести множество выражений, которые вызывают его сбой. Полная реализация включена в .NET Standard под именем ExpressionVisitor и может обрабатывать все возможные типы узлов.

Наконец, библиотека, используемая в этой статье, была создана для демонстрации и обучения. Он не оптимизирован. Это позволяет сделать структуры понятными и подчеркнуть методы, используемые для посещения узлов и анализа их содержимого.

Даже с этими ограничениями вы должны быть хорошо на пути к написанию алгоритмов, которые считывают и понимают деревья выражений.

Построение деревьев выражений

17.06.2025

Компилятор C# создал все деревья выражений, которые вы видели до сих пор. Вы создали лямбда-выражение, присвоенное переменной, тип которой является `Expression<Func<T>>` или аналогичным типом. Во многих сценариях вы создаете выражение в памяти во время выполнения.

Деревья выражений неизменяемы. Будучи неизменяемым означает, что вы должны построить дерево из листьев до корня. API, используемые для создания деревьев выражений, отражают этот факт: методы, используемые для создания узла, принимают все его дочерние элементы в качестве аргументов. Давайте рассмотрим несколько примеров, чтобы показать вам методы.

Создание узлов

Вы начинаете с выражения сложения, с которым вы работали в течение этих разделов.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

Чтобы построить это дерево выражений, сначала создайте узлы-листья. Конечные узлы являются константами. Constant Используйте метод для создания узлов:

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Затем создайте выражение сложения:

C#

```
var addition = Expression.Add(one, two);
```

Создав выражение добавления, создайте лямбда-выражение:

C#

```
var lambda = Expression.Lambda(addition);
```

Это лямбда-выражение не содержит аргументов. Далее в этом разделе показано, как сопоставить аргументы с параметрами и создать более сложные выражения.

Для таких выражений можно объединить все вызовы в одну инструкцию:

```C#
var lambda2 = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Создание дерева

В предыдущем разделе показаны основы создания дерева выражений в памяти. Более сложные деревья обычно означают больше типов узлов и больше узлов в дереве. Давайте рассмотрим еще один пример и покажем два других типа узлов, которые обычно создаются при создании деревьев выражений: узлов аргументов и узлов вызова метода. Давайте создадим дерево выражений для создания этого выражения:

```C#
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

Начните с создания выражений параметров для `x` и `y`:

```C#
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

Создание выражений умножения и добавления следует шаблону, который вы уже видели:

```C#
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Затем необходимо создать выражение вызова метода для обращения к `Math.Sqrt`.

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ?? throw
new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

Вызов `GetMethod` может вернуть `null`, если метод не найден. Скорее всего, это связано с ошибкой имени метода. В противном случае это может означать, что требуемая сборка не загружается. Наконец, вы помещаете вызов метода в лямбда-выражение и определите аргументы для лямбда-выражения:

C#

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

В этом более сложном примере вы увидите несколько дополнительных методов, которые часто требуются для создания деревьев выражений.

Сначала необходимо создать объекты, представляющие параметры или локальные переменные, прежде чем использовать их. После создания этих объектов их можно использовать в дереве выражений везде, где вам нужно.

Во-вторых, необходимо использовать подмножество API отражения для создания System.Reflection.MethodInfo объекта, чтобы создать дерево выражений для доступа к такому методу. Необходимо ограничиться подмножеством API отражения, которые доступны на платформе .NET Core. Опять же, эти методы расширяются на другие деревья выражений.

Создавайте код углубленно

Вы не ограничены тем, что можно создать с помощью этих API. Тем не менее, чем сложнее дерево выражений, которое вы хотите создать, тем труднее управлять и читать код.

Давайте создадим дерево выражений, эквивалентное этому коду:

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
```

```
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Предыдущий код не построил дерево выражений, а просто создал делегат. `Expression` Используя класс, нельзя создавать лямбда-инструкции. Ниже приведен код, необходимый для создания той же функциональности. Не существует API для создания `while` цикла, необходимо создать цикл, содержащий условный тест и метку для выхода из цикла.

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

Код для создания дерева выражений для факториальной функции довольно длинный, более сложный, и он переполнен метками, операторами останова и другими элементами, которые мы хотим избегать в наших повседневных задачах программирования.

В этом разделе вы написали код, чтобы посетить каждый узел в этом дереве выражений и записать сведения о узлах, созданных в этом примере. Вы можете просмотреть или скачать пример кода в репозитории dotnet/docs GitHub. Поэкспериментируйте самостоятельно, создав и выполнив примеры.

Сопоставление конструкций кода с выражениями

В следующем примере кода показано дерево выражений, представляющее лямбда-выражение `num => num < 5` с помощью API.

C#

```
// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

API деревьев выражений также поддерживает операции присваивания и выражения управления потоком, такие как циклы, условные блоки и `try-catch` блоки. С помощью API можно создавать деревья выражений, которые являются более сложными, чем те, которые можно создать из лямбда-выражений компилятором C#. В следующем примере показано, как создать дерево выражений, вычисляющее факториал числа.

C#

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
        // Adding a loop.
```

```
        Expression.Loop(
           // Adding a conditional block into the loop.
           Expression.IfThenElse(
               // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
               // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
               // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
       // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

Дополнительные сведения см. в статье "Создание динамических методов с помощью деревьев выражений" в Visual Studio 2010, которое также относится к более поздним версиям Visual Studio.

Перевод деревьев выражений

17.06.2025

В этой статье вы узнаете, как посетить каждый узел в дереве выражений при создании измененной копии этого дерева выражений. Вы переводите деревья выражений, чтобы понять алгоритмы, чтобы их можно было перенести в другую среду. Вы изменяете созданный алгоритм. Вы можете добавлять логирование, перехватывать вызовы методов и отслеживать их для различных целей.

Код, который вы создаете для перевода дерева выражений, является расширением того, что вы уже видели, чтобы посетить все узлы в дереве. При переводе дерева выражений вы посещаете все узлы и, посещая их, строите новое дерево. Новое дерево может содержать ссылки на исходные узлы или новые узлы, помещенные в дерево.

Давайте посетим дерево выражений и создадим новое дерево с некоторыми заменёнными узлами. В этом примере давайте заменим любую константу константой, которая составляет 10 раз больше. В противном случае дерево выражений остается нетронутым. Вместо того, чтобы считывать значение константы и заменять его новой константой, замените этот узел константой новым узлом, выполняющим умножение.

После поиска постоянного узла создается новый узел умножения, дочерние элементы которого являются исходной константой и константой `10`:

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Создайте новое дерево, заменив исходный узел заменой. Вы проверяете изменения путем компиляции и выполнения замененного дерева.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Создание нового дерева — это сочетание посещения узлов в существующем дереве и создания новых узлов и их вставки в дерево. В предыдущем примере показано важность неизменяемых деревьев выражений. Обратите внимание, что новое дерево, созданное в предыдущем коде, содержит смесь только что созданных узлов и узлов из существующего дерева. Узлы можно использовать в обоих деревьях, так как узлы в существующем дереве не могут быть изменены. Повторное использование узлов приводит к значительной эффективности памяти. Одни и те же узлы можно использовать на протяжении всего дерева или в нескольких деревьях выражений. Так как узлы не могут быть изменены, один и тот же узел можно повторно использовать при необходимости.

Обход и выполнение добавления

Давайте проверим новое дерево операций сложения, создав второй объект, который проходит по дереву узлов сложения и вычисляет результат. Сделайте пару изменений для посетителя, который вы видели до сих пор. В этой новой версии посетитель возвращает частичную сумму операции добавления до этой точки. Для константного выражения это просто значение константного выражения. Для выражения сложения результатом является сумма левых и правых операндов, после того как эти деревья были пройдены.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
```

```
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) +
aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Здесь довольно много кода, но основные понятия легко усваиваются. Этот код посещает дочерние элементы в глубине первого поиска. При обнаружении постоянного узла посетитель возвращает значение константы. После того, как посетитель посетил обоих детей, эти дети вычислили сумму, вычисленную для этой поддеревки. Теперь узел сложения может вычислить сумму. После посещения всех узлов в дереве выражений сумма вычисляется. Чтобы отслеживать выполнение, запустите пример в отладчике и следите за процессом выполнения.

Упростим отслеживание того, как узлы дерева анализируются и как вычисляется сумма путем его обхода. Вот обновленная версия метода Aggregate, которая содержит много информации для трассировки.

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
```

```
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Выполнение выражения `sum` дает следующие выходные данные:

Выходные данные

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Проанализируйте результаты и следуйте ходу выполнения в предыдущем коде. Вы должны иметь возможность разобраться, как код посещает каждый узел и вычисляет сумму в процессе обхода дерева.

Теперь давайте рассмотрим другой запуск с выражением, заданным `sum1`:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Ниже приведены выходные данные для изучения этого выражения:

Выходные данные

```
Found Addition Expression
Computing Left node
```

```
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Хотя окончательный ответ один и тот же, порядок обхода дерева отличается. Узлы перемещаются в другом порядке, так как дерево было создано с различными операциями, происходящими в первую очередь.

Создание измененной копии

Создайте проект **консольного приложения** . Добавьте директиву `using` в файл для пространства имен `System.Linq.Expressions`. Добавьте класс `AndAlsoModifier` в ваш проект.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
```

```
b.IsLiftedToNull, b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

Этот класс наследует ExpressionVisitor класс и специализирован для изменения выражений, представляющих условные `AND` операции. Он изменяет эти операции с условного `AND` на условный `OR`. Класс переопределяет VisitBinary метод базового типа, так как условные `AND` выражения представлены как двоичные выражения. В методе `VisitBinary`, если выражение, переданное в него, представляет собой условную операцию `AND`, код строит новое выражение, которое содержит условный оператор `OR` вместо условного оператора `AND`. Если переданное выражение `VisitBinary` не представляет условную `AND` операцию, метод передаёт выполнение реализации базового класса. Методы базового класса создают узлы, похожие на переданные деревья выражений, но их поддеревья заменяются деревьями выражений, которые рекурсивно создаёт посетитель.

Добавьте директиву `using` в файл для пространства имен `System.Linq.Expressions`. Добавьте код в `Main` метод в файле Program.cs, чтобы создать дерево выражений и передать его в метод, который изменяет его.

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

Код создает выражение, содержащее условную `AND` операцию. Затем он создает экземпляр `AndAlsoModifier` класса и передает выражение `Modify` методу этого класса. И исходные, и измененные деревья выражений выводятся для отображения изменения. Скомпилируйте и запустите приложение.

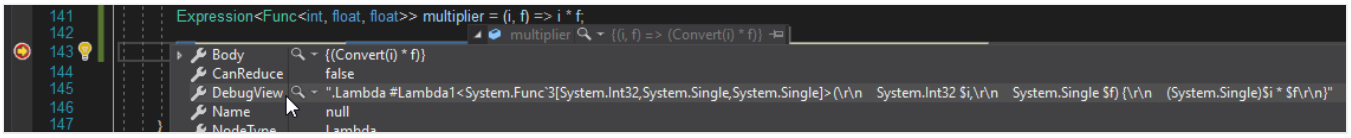
Подробнее

В этом примере показано небольшое подмножество кода, которое будет построено для обхода и интерпретации алгоритмов, представленных деревом выражений. Прочитайте эту серию Мэтта Уоррена, чтобы узнать о создании библиотеки общего назначения, которая переводит деревья выражений на другой язык. Он подробно описывает, как перевести любой код, который можно найти в дереве выражений.

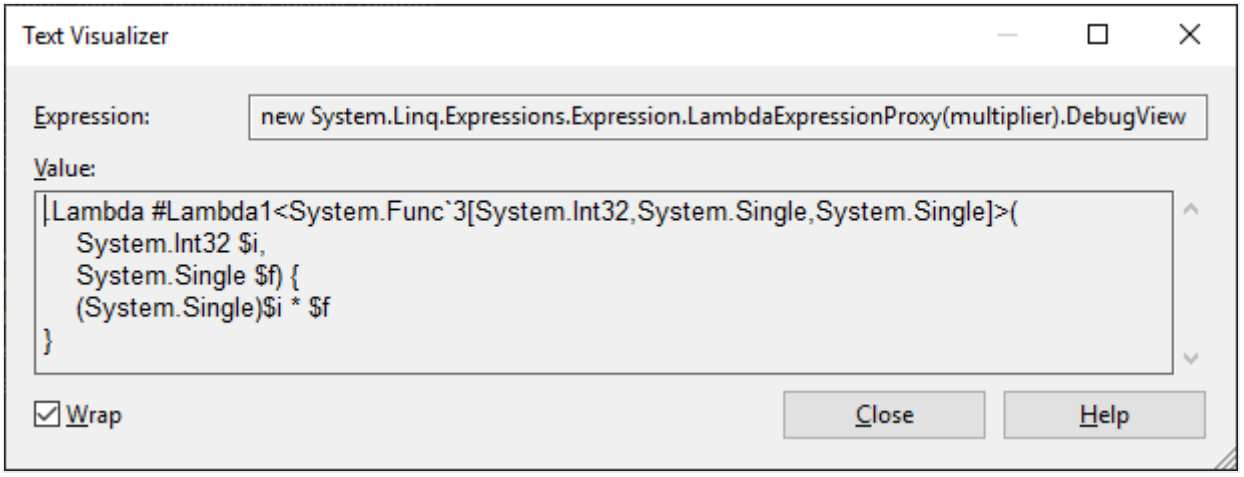
Теперь вы увидели истинную мощь деревьев выражений. Вы проверяете набор кода, вносите изменения в этот код и выполняете измененную версию. Так как деревья выражений неизменяемы, вы создаете новые деревья с помощью компонентов существующих деревьев. Повторное использование узлов сводит к минимуму объем памяти, необходимой для создания измененных деревьев выражений.

Отладка деревьев выражений в Visual Studio

При отладке приложений можно анализировать структуру и содержимое деревьев выражений. Чтобы получить краткий обзор структуры дерева выражений, можно использовать `DebugView` свойство, представляющее деревья выражений с помощью специального синтаксиса. `DebugView` доступен только в режиме отладки.

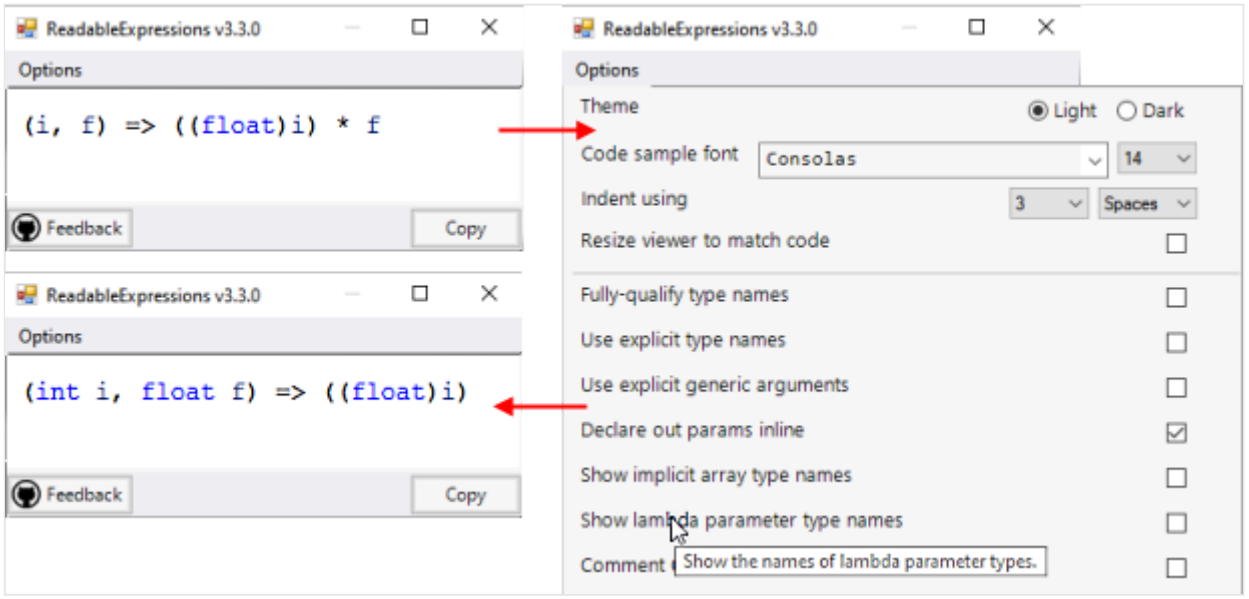


Так как `DebugView` это строка, вы можете использовать встроенный визуализатор текста для просмотра его в нескольких строках, выбрав **визуализатор текста** из значка с увеличением стекла рядом с `DebugView` меткой.

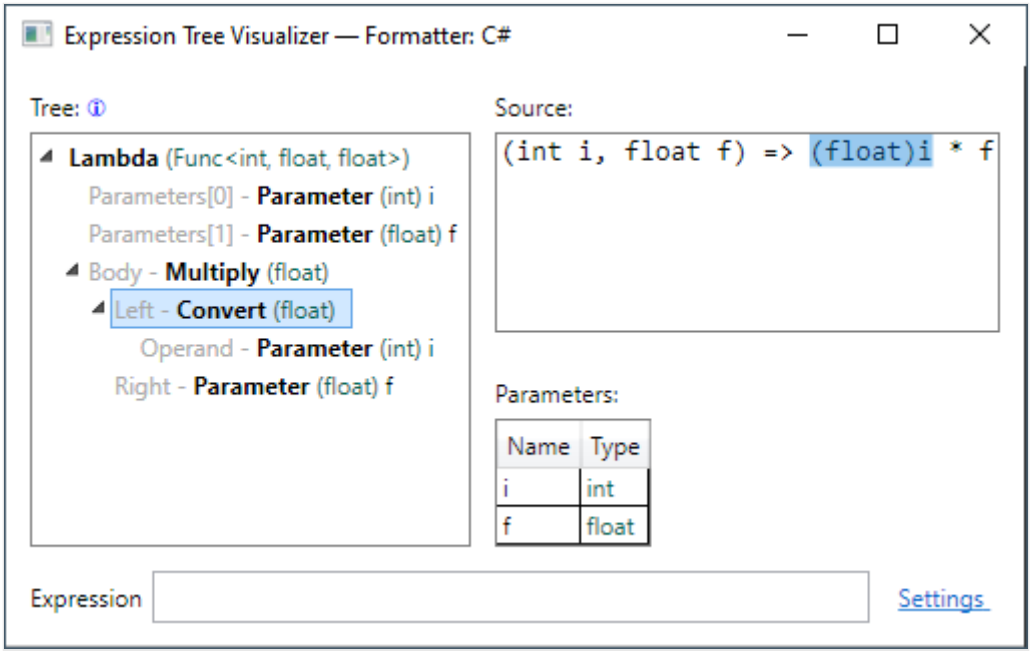


Кроме того, можно установить и использовать настраиваемый визуализатор для деревьев выражений, например:

- Доступные для чтения выражения (лицензия MIT, доступная в Visual Studio Marketplace), отображают дерево выражений в виде кода C# с различными параметрами отрисовки:

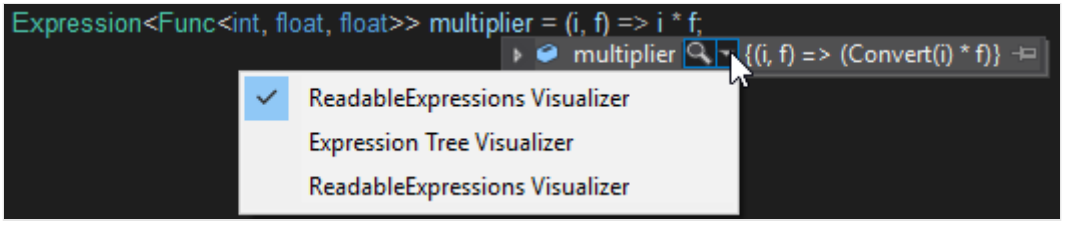


- Визуализатор дерева выражений (лицензия MIT) предоставляет представление дерева выражений и отдельных узлов:



Открытие визуализатора для дерева выражений

Выберите значок лупы, который отображается рядом с деревом выражений в **подсказках данных**, окне **Наблюдение**, окне **Авто** или окне **Локальные**. Отображается список доступных визуализаторов:



Выберите визуализатор, который вы хотите использовать.

См. также

- Отладка в Visual Studio
- Создание пользовательских визуализаторов
- DebugView синтаксиса

Last updated on 24.02.2026

Синтаксис DebugView

17.06.2025

Свойство **DebugView** (доступно только при отладке) предоставляет строковую отрисовку деревьев выражений. Большая часть синтаксиса довольно простая для понимания; Специальные случаи описаны в следующих разделах.

За каждым примером следует комментарий блока, содержащий **DebugView**.

ParameterExpression

ParameterExpression Имена переменных отображаются с символом `$` в начале.

Если параметр не имеет имени, он назначается автоматически созданное имя, например `$var1` или `$var2`.

C#

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

Для ConstantExpression объектов, представляющих целые значения, строки и `null`, отображается значение константы.

Для числовых типов, имеющих стандартные суффиксы в виде литералов C#, суффикс добавляется в значение. В следующей таблице показаны суффиксы, связанные с различными числовыми типами.

Развернуть таблицу

Тип	Ключевое слово	Суффикс
System.UInt32	uint	У

Тип	Ключевое слово	Суффикс
System.Int64	длинный	L
System.UInt64	ulong	УЛ
System.Double	двойной	Д
System.Single	плавать	Ф
System.Decimal	десятичное	М

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

Если тип BlockExpression объекта отличается от типа последнего выражения в блоке, тип отображается в угловых скобках (`<` и `>`). В противном случае тип BlockExpression объекта не отображается.

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

Выражение лямбда

LambdaExpression Объекты отображаются вместе с их типами делегатов.

Если лямбда-выражение не имеет имени, оно назначается автоматически созданное имя, например `#Lambda1` или `#Lambda2`.

C#

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1),
"SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

Если указать значение по умолчанию для LabelExpression объекта, это значение отображается перед LabelTarget объектом.

Маркер `.Label` указывает начало метки. Маркер `.LabelTarget` указывает место назначения, к которому нужно перейти.

Если метка не имеет имени, она назначается автоматически созданное имя, например `#Label1` или `#Label2`.

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
```

```
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Проверенные операторы

Проверенные операторы отображаются символом `#`, расположенным перед оператором. Например, проверяемый оператор добавления отображается как `#+`.

C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),
Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),
typeof(int));
/*
    #(System.Int32)10D
*/
```

Обзор взаимодействия

17.06.2025

Взаимодействие позволяет сохранять и использовать существующие инвестиции в неуправляемый код. Код, который выполняется под контролем среды CLR, является *управляемым кодом*, а код, который выполняется вне среды CLR, *неуправляемый код*. КОМПОНЕНТЫ COM, COM+, C++, компоненты ActiveX и API Microsoft Windows являются примерами неуправляемого кода.

.NET обеспечивает взаимодействие с неуправляемым кодом через службы вызова платформ, пространство имен System.Runtime.InteropServices, взаимодействие C++ и взаимодействие с COM (COM-interop).

Вызов платформенных функций

Вызов платформы — это служба, которая позволяет управляемому коду вызывать неуправляемые функции, реализованные в библиотеках динамических ссылок (DLL), таких как API Microsoft Windows. Он находит и вызывает экспортированную функцию и маршалирует свои аргументы (целые числа, строки, массивы, структуры и т. д.) через границу взаимодействия по мере необходимости.

Дополнительные сведения см. в разделе "Использование неуправляемых функций DLL " и "Как использовать вызов платформы для воспроизведения WAV-файла".

> ⓘ **Примечание**
>
> Среда **Common Language Runtime** (CLR) управляет доступом к системным ресурсам. Вызов неуправляемого кода, который находится за пределами среды CLR, обходит этот механизм безопасности и, следовательно, представляет угрозу безопасности. Например, неуправляемый код может напрямую вызывать ресурсы в неуправляемом коде, обходя механизмы безопасности CLR. Дополнительные сведения см. **в разделе "Безопасность" в .NET**.

Интероперабельность C++

Для обертки класса на родном C++ можно использовать C++ interop, также известный как It Just Works (IJW). Взаимодействие C++ позволяет создавать код на языке C# или другом языке .NET для доступа к нему. Вы пишете код на C++, чтобы обернуть собственный DLL или COM компонент. В отличие от других языков .NET, Visual C++

поддерживает взаимодействие, которое включает управляемый и неуправляемый код в одном приложении и даже в том же файле. Затем вы создаете код C++ с помощью переключателя **компилятора /clr** для создания управляемой сборки. Наконец, вы добавите ссылку на сборку в проекте C# и используйте упакованные объекты так же, как и другие управляемые классы.

Доступ к COM-компонентам в C#

Компонент COM можно использовать в проекте C#. Общий порядок действий.

1. Найдите com-компонент для использования и зарегистрируйте его. Используйте regsvr32.exe для регистрации или отмены регистрации библиотеки DLL COM.
2. Добавьте в проект ссылку на com-компонент или библиотеку типов. При добавлении ссылки Visual Studio использует Tlbimp.exe (Импортёр библиотеки типов), который принимает библиотеку типов в качестве входных данных и создаёт сборку взаимодействия .NET. Сборка, также называемая вызываемой оболочкой среды выполнения (RCW), содержит управляемые классы и интерфейсы, которые оборачиваются вокруг классов и интерфейсов COM, находящихся в библиотеке типов. Visual Studio добавляет в проект ссылку на созданную сборку.
3. Создайте экземпляр класса, определенного в RCW. Создание экземпляра этого класса создает экземпляр COM-объекта.
4. Используйте объект так же, как и другие управляемые объекты. Когда объект удаляется сборщиком мусора, экземпляр COM-объекта также освобождается в памяти.

Дополнительные сведения см. в статье о предоставлении COM-компонентов платформе .NET Framework.

Предоставление доступа C# к COM

Клиенты COM могут использовать типы C#, которые были правильно предоставлены. Ниже приведены основные шаги для предоставления типов C#:

1. Добавьте атрибуты взаимодействия в проект C#. Вы можете сделать сборку COM видимой, изменив свойства проекта C#. Дополнительные сведения см. в диалоговом окне "Сведения о сборке".
2. Создайте библиотеку типов COM и зарегистрируйте ее для использования COM. Свойства проекта C# можно изменить, чтобы автоматически зарегистрировать сборку C# для взаимодействия с COM. Visual Studio использует Regasm.exe (средство регистрации сборок) с помощью `/tlb` переключателя командной строки, который принимает управляемую сборку в качестве входных данных для создания

библиотеки типов. Эта библиотека типов описывает `public` типы в сборке и добавляет записи реестра, чтобы клиенты COM могли создавать управляемые классы.

Для получения дополнительной информации см. предоставление компонентов .NET Framework COM и пример COM-класса.

См. также

- Повышение производительности взаимодействия
- Общие сведения о взаимодействии между COM и .NET
- Введение в COM-взаимодействие в Visual Basic
- Маршалинг между управляемым и неуправляемным кодом
- Взаимодействие с неуправляемым кодом
- Регистрация сборок с помощью COM

Пример COM-класса

Статья • 02.03.2024

Следующий код является примером класса, который будет предоставляться в виде COM-объекта. После размещения этого кода в .cs-файле, добавленном в проект, задайте **для свойства Register for COM Interop** значение **True**. Дополнительные сведения см. в разделе Практическое руководство. Регистрация компонента для COM-взаимодействия.

Предоставление объектов C# com требует объявления интерфейса класса, интерфейса событий при необходимости и самого класса. Члены класса должны соответствовать указанным ниже правилам, чтобы стать доступными для COM.

- Класс должен быть открытым.
- Свойства, методы и события должны быть открытыми.
- Свойства и методы должны быть объявлены в интерфейсе класса.
- События должны быть объявлены в интерфейсе событий.

Другие общедоступные члены класса, которые не объявляются в этих интерфейсах, не видны COM, но они видны другим объектам .NET. Чтобы предоставить свойства и методы COM, их необходимо объявить в интерфейсе класса, пометить атрибутом `DispId` и реализовать в классе. Порядок объявления элементов в интерфейсе — это порядок, используемый для vtable COM. Чтобы предоставить события из класса, их необходимо объявить в интерфейсе событий и пометить атрибутом `DispId`. Класс не должен реализовывать этот интерфейс.

Класс реализует интерфейс класса; он может реализовать несколько интерфейсов, но первая реализация — это интерфейс класса по умолчанию. Реализуйте методы и свойства, доступные модели COM здесь. Они должны быть общедоступными и должны соответствовать объявлениям в интерфейсе класса. Кроме того, объявите здесь события, инициируемые классом. Они должны быть общедоступными и должны соответствовать объявлениям в интерфейсе событий.

Пример

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
```

```
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

См. также

- Совместимость
- Страница "Сборка" в конструкторе проектов (C#)

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Использование вызова платформы для воспроизведения WAV-файла

В следующем примере кода C# показано, как использовать службы вызова платформы для воспроизведения звукового файла WAV в операционной системе Windows.

Example

В этом примере кода используется DllImportAttribute для импорта точки входа метода `PlaySound` из `winmm.dll` как `Form1 PlaySound()`. В примере есть простая Windows Форма с кнопкой. При нажатии кнопки открывается стандартное диалоговое окно Windows OpenFileDialog, в котором можно выбрать файл для воспроизведения. При выборе WAV-файла он воспроизводится с использованием метода `PlaySound()` библиотеки *winmm.dll*. Дополнительные сведения об этом методе см. в разделе "Использование функции PlaySound с файлами Waveform-Audio". Найдите и выберите файл с расширением .wav, а затем выберите **Открыть**, чтобы воспроизвести WAV-файл с помощью P/Invoke. Текстовое поле показывает полный путь к выбранному файлу.

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1()  // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true, CharSet =
CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
```

```
        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a default
method
        // with this name. We add it here to prevent confusion for those using the
samples.
    }
}
```

Диалоговое окно "**Открыть файлы**" фильтруется для отображения только файлов с расширением .wav с помощью параметров фильтра.

Компиляция программы

Создайте проект приложения Windows Forms C# в Visual Studio и назовите его **WinSound**. Скопируйте предыдущий код и вставьте его в содержимое файла *Form1.cs* . Скопируйте следующий код и вставьте его в файл *Form1.Designer.cs*, в метод `InitializeComponent()`, после любого существующего кода.

C#

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

Скомпилируйте и запустите код.

См. также

- Подробнее о Platform Invoke
- Маршалинг данных с помощью вызова платформы

Last updated on 12.07.2026

Практическое руководство. Использование индексированных свойств в программировании COM-взаимодействия

Статья • 04.10.2023

Индексированные свойства работают вместе с другими функциями в C#, такими как именованные и необязательные аргументы, новый тип (динамический) и внедренные сведения о типах для улучшения программирования Microsoft Office.

> **ⓘ Важно!**
>
> **VSTO (набор средств Visual Studio для Office)** зависит от **платформа .NET Framework**. Надстройки COM также можно записывать с помощью платформа .NET Framework. Надстройки Office нельзя создавать с помощью **.NET Core и .NET 5+**, последних версий .NET. Это связано с тем, что .NET Core/.NET 5+ не может работать вместе с платформа .NET Framework в том же процессе и может привести к сбоям загрузки надстроек. Вы можете продолжать использовать платформа .NET Framework для записи надстроек VSTO и COM для Office. Корпорация Майкрософт не будет обновлять VSTO или платформу надстройки COM для использования .NET Core или .NET 5+. Вы можете воспользоваться преимуществами .NET Core и .NET 5+, включая ASP.NET Core, чтобы создать серверную часть **надстроек** Office Web.

В более ранних версиях C# методы доступны как свойства только при условии, что у метода `get` нет параметров, а у метода `set` есть только один параметр значения. Однако не все свойства COM удовлетворяют этим ограничениям. Например, свойство Range[] Excel имеет метод доступа `get`, которому требуется параметр для имени диапазона. В прошлом, так как вы не могли получить доступ к `Range` свойству напрямую, вам пришлось использовать `get_Range` метод, как показано в следующем примере.

{language}

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Индексированные свойства позволяют вместо этого использовать следующий код:

{language}

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

В предыдущем примере также используются необязательные аргументы, которые позволяют опустить `Type.Missing`.

Индексированные свойства позволяют писать следующий код.

{language}

```
// Visual C# 2010.
targetRange.Value = "Name";
```

Не удается создать индексированные свойства собственных. Эта возможность поддерживает только использование имеющихся индексированных свойств.

Пример

Ниже приведен полный пример кода. См. сведения о создании проекта, обращающегося к Office API, в руководстве по получению доступа к объектам взаимодействия Office с помощью функций C#.

{language}

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
```

```
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

См. также

- Именованные и необязательные аргументы
- Динамический
- Использование типа dynamic

Collaborate with us on GitHub

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.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Как получить доступ к объектам взаимодействия Office

Статья • 04.10.2023

В C# предусмотрены функции, которые упрощают доступ к объектам API Office. К новым функциям относятся именованные и необязательные аргументы, новый тип `dynamic`, а также возможность передавать аргументы ссылочным параметрам в методах COM, как если бы они были параметрами значений.

В этой статье вы используете новые функции для написания кода, создающего и отображающего лист Microsoft Office Excel. Код создается для добавления документа Office Word, содержащего значок, связанный с листом Excel.

Для выполнения данного пошагового руководства на компьютере должны быть установлены Microsoft Office Excel 2007 и Microsoft Office Word 2007 или более поздние версии продуктов.

> ⓘ **Примечание**
>
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе **Персонализация среды IDE**.

> ⓘ **Важно!**
>
> **VSTO (набор средств Visual Studio для Office)** зависит от **платформа .NET Framework**. Надстройки COM также можно записывать с помощью платформа .NET Framework. Надстройки Office нельзя создавать с помощью **.NET Core и .NET 5+**, последних версий .NET. Это связано с тем, что .NET Core/.NET 5+ не может работать вместе с платформа .NET Framework в том же процессе и может привести к сбоям загрузки надстроек. Вы можете продолжать использовать платформа .NET Framework для записи надстроек VSTO и COM для Office. Корпорация Майкрософт не будет обновлять VSTO или платформу надстройки COM для использования .NET Core или .NET 5+. Вы можете воспользоваться преимуществами .NET Core и .NET 5+, включая ASP.NET Core, чтобы создать серверную часть **надстроек** Office Web.

Создание нового проекта консольного приложения

1. Запустите среду Visual Studio.
2. В меню **Файл** последовательно выберите команды **Создать**и **Проект**. Откроется диалоговое окно **Создание проекта** .
3. **В области установленных шаблонов** разверните **C#**, а затем выберите **Windows**.
4. В верхней части диалогового **окна "Новый проект"**, **чтобы выбрать платформа .NET Framework 4** (или более поздней версии) в качестве целевой платформы.
5. **В области "Шаблоны"** выберите **консольное приложение**.
6. Введите имя проекта в поле **Имя**.
7. Нажмите **ОК**.

В **обозревателе решений** появится новый проект.

Добавление ссылок

1. В **Обозреватель решений** щелкните правой кнопкой мыши имя проекта и нажмите кнопку **"Добавить ссылку"**. Откроется диалоговое окно **Добавление ссылки**.
2. На странице **Сборки** в списке **Имя компонента** выберите **Microsoft.Office.Interop.Word**, а затем, удерживая нажатой клавишу CTRL, выберите **Microsoft.Office.Interop.Excel**. Если сборки не отображаются, их может потребоваться установить. Узнайте , как установить основные сборки взаимодействия Office.
3. Нажмите **ОК**.

Добавление необходимых директив using

В **Обозреватель решений** щелкните файл Program.cs *правой* кнопкой мыши и выберите **команду View Code**. В начало файла кода добавьте следующие директивы `using`:

```
C#
```

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Создание списка банковских счетов

Вставьте следующее определение класса в файл **Program.cs** в класс `Program`.

C#

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Чтобы создать список `bankAccounts`, содержащий два счета, добавьте в метод `Main` следующий код.

C#

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

Объявление метода, экспортирующего сведения о счетах в Excel

1. Чтобы настроить лист Excel, добавьте в класс `Program` следующий метод. У метода Add есть необязательный параметр для указания конкретного шаблона. Необязательные параметры позволяют опустить аргумент для этого параметра, если вы хотите использовать значение по умолчанию параметра. Так как аргумент не задан, `Add` использует шаблон по умолчанию и создает новую книгу. В эквивалентном операторе в более ранних версиях C# необходимо было использовать аргумент-местозаполнитель `ExcelApp.Workbooks.Add(Type.Missing)`.

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
    // Because no argument is sent in this example, Add creates a new
workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

Добавьте в конец метода `DisplayInExcel` следующий код. Этот код вставляет значения в первые два столбца первой строки листа.

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

Добавьте в конец метода `DisplayInExcel` следующий код. Цикл `foreach` помещает сведения из списка счетов в первые два столбца последовательных строк листа.

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

Добавьте в конец метода `DisplayInExcel` следующий код, чтобы ширина столбца изменялась в соответствии с содержимым.

C#

```
workSheet.Columns[1].AutoFit();
```

```
workSheet.Columns[2].AutoFit();
```

В более ранних версиях C# требовалось явное приведение типов для этих операций, поскольку `ExcelApp.Columns[1]` возвращает тип `Object`, а метод `AutoFit` является методом Excel Range. Приведение показано в следующих строках.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# преобразует возвращаемое `Object` `dynamic` значение автоматически, если сборка ссылается **на параметр компилятора EmbedInteropTypes** или, аналогично, если **свойство Excel Embed Interop Types** имеет значение true. True является значением по умолчанию для этого свойства.

Запуск проекта

Добавьте в конец метода `Main` следующую строку.

C#

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

Нажмите клавиши CTRL+F5. Появится книга Excel, содержащая данные о двух счетах.

Добавление документа Word

Следующий код открывает приложение Word и создает значок, который ссылается на лист Excel. Вставьте метод `CreateIconInWordDoc`, указанный далее в этом шаге, в класс `Program`. `CreateIconInWordDoc` использует именованные и необязательные аргументы, чтобы упростить вызовы методов Add и PasteSpecial. Эти вызовы включают две другие функции, которые упрощают вызовы к COM-методам, имеющим ссылочные параметры. Во-первых, аргументы можно передать в ссылочные параметры, как если бы они были параметрами значений. Это значит, что значения можно передавать напрямую без создания переменной для каждого ссылочного параметра. Компилятор создает временные переменные для хранения значений аргументов и уничтожает эти переменные после завершения

вызываемого метода. Во-вторых, ключевое слово `ref` в списке аргументов можно опустить.

У метода `Add` имеется четыре ссылочных параметра, все из которых являются необязательными. Аргументы для любого или всех параметров можно опустить, если вы хотите использовать их значения по умолчанию.

Метод `PasteSpecial` вставляет содержимое буфера обмена. У метода имеется семь ссылочных параметров, все из которых являются необязательными. Следующий код задает аргументы для двух из них: `Link` для создания ссылки на исходное содержимое буфера и `DisplayAsIcon` для отображения ссылки в виде значка. Для этих двух аргументов можно использовать именованные аргументы и опустить другие. Хотя эти аргументы являются ссылочными параметрами, вам не нужно использовать `ref` ключевое слово или создавать переменные для отправки в качестве аргументов. Значения можно передать напрямую.

C#

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

Добавьте в конец метода `Main` следующую инструкцию.

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

Добавьте в конец метода `DisplayInExcel` следующую инструкцию. Метод `Copy` добавляет лист в буфер обмена.

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

Нажмите клавиши CTRL+F5. Появится документ Word, содержащий значок. Дважды щелкните значок, чтобы отобразить лист на переднем плане.

Задание свойства "Внедрить типы взаимодействия"

Дополнительные улучшения возможны при вызове типа COM, который не требует основной сборки взаимодействия (PIA) во время выполнения. Избавление от зависимостей от PIA приводит к независимости версий и делает развертывание более простым. Дополнительные сведения о преимуществах программирования без основных сборок взаимодействия см. в разделе Пошаговое руководство. Внедрение данных о типах из управляемых сборок.

Кроме того, программирование проще, так как `dynamic` тип представляет необходимые и возвращаемые типы, объявленные в методах COM. Переменные, имеющие тип `dynamic` , не оцениваются до времени выполнения, что устраняет необходимость явного приведения. Дополнительные сведения см. в разделе Использование типа dynamic.

Внедрение сведений о типе вместо использования ЛИЧНЫХ данных — это поведение по умолчанию. Из-за этого по умолчанию несколько предыдущих примеров упрощены. Вам не требуется явное приведение. Например, объявление `worksheet` в методе `DisplayInExcel` записывается как `Excel._Worksheet workSheet = excelApp.ActiveSheet`, а не как `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. Для вызовов `AutoFit` в рамках одного метода также требовалось бы явное приведение без поведения по умолчанию, поскольку `ExcelApp.Columns[1]` возвращает тип `Object`, а `AutoFit` является методом Excel. Приведение показано в следующем примере.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

Чтобы изменить значения по умолчанию и использовать пиА вместо внедрения сведений о типе, разверните узел "Ссылки" в Обозреватель решений, а затем выберите **Microsoft.Office.Interop.Excel** или **Microsoft.Office.Interop.Word**. Если окно свойств не отображается, **нажмите клавишу F4.** В списке свойств найдите свойство **Внедрить типы взаимодействия** и измените его значение на **False**. Также можно выполнить компиляцию с помощью командной строки с использованием параметра компилятора **References** вместо **EmbedInteropTypes**.

Дополнительное форматирование таблицы

Замените два вызова `AutoFit` в методе `DisplayInExcel` следующей инструкцией.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

У метода AutoFormat имеется семь параметров значений, и все они являются необязательными. Именованные и необязательные аргументы позволяют задать аргументы для всех параметров, их части или ни для одного параметра. В предыдущем операторе приведен аргумент только для одного из параметров `Format`. Так как `Format` это первый параметр в списке параметров, вам не нужно указать имя параметра. Однако оператор может быть проще понять, если вы включаете имя параметра, как показано в следующем коде.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

Нажмите сочетание клавиш CTRL + F5, чтобы увидеть результат. Другие форматы можно найти в перечислении XlRangeAutoFormat .

Пример

Ниже приведен полный пример кода.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgrammingWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection
returned
            // by property Workbooks. The new workbook becomes the active
workbook.
            // Add has an optional parameter for specifying a particular
template.
            // Because no argument is sent in this example, Add creates a
new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet =
(Excel.Worksheet)excelApp.ActiveSheet;
```

```
            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy
method has one
            // optional parameter for specifying a destination. Because no
argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which
are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to
create
            // variables to send in as arguments. You can send the values
directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
```

```
    }
}
```

См. также

- Type.Missing
- Динамический
- Именованные и необязательные аргументы
- Использование именованных и необязательных аргументов в программировании приложений Office

Collaborate with us on GitHub

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.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Использование именованных и необязательных аргументов в программировании Office

03.07.2025

Именованные аргументы и необязательные аргументы повышают удобство, гибкость и удобочитаемость в программировании на C#. Кроме того, эти функции значительно упрощают доступ к COM-интерфейсам, таким как API автоматизации Microsoft Office.

> ⓘ **Важно!**
>
> VSTO (Средства Visual Studio для Office) использует платформу .NET Framework. Надстройки COM также могут быть написаны с помощью .NET Framework. Надстройки Office нельзя создавать с помощью .NET Core и .NET 5+, последних версий .NET. Это связано с тем, что .NET Core/.NET 5+ не может работать вместе с .NET Framework в том же процессе и может привести к сбоям загрузки надстроек. Вы можете продолжать использовать .NET Framework для записи надстроек VSTO и COM для Office. Корпорация Майкрософт не будет обновлять VSTO или платформу надстройки COM для использования .NET Core или .NET 5+. Вы можете воспользоваться преимуществами .NET Core и .NET 5+, включая ASP.NET Core, для создания серверной части надстроек Office Web.

В следующем примере метод ConvertToTable имеет 16 параметров, представляющих характеристики таблицы, такие как количество столбцов и строк, форматирование, границы, шрифты и цвета. Все 16 параметров являются необязательными, так как в большинстве случаев вы не хотите указывать определенные значения для всех этих параметров. Однако, если не используются именованные и необязательные аргументы, вы должны указать значение или использовать значение-заполнитель. При использовании именованных и необязательных аргументов значения указываются только для параметров, необходимых для проекта.

Для выполнения этих процедур на компьютере необходимо установить Microsoft Office Word.

> ⓘ **Примечание**
>
> На компьютере могут отображаться различные имена или расположения для некоторых элементов пользовательского интерфейса Visual Studio в следующих инструкциях. Выпуск Visual Studio, который у вас есть, и параметры, которые вы

используете, определяют эти элементы. Дополнительные сведения см. в разделе Персонализация интегрированной среды разработки.

Создание нового консольного приложения

Запустите Visual Studio. В меню **Файл** выберите пункт **Создать**, а затем выберите **Проект**. В области **"Категории шаблонов** " разверните **C#**, а затем выберите **Windows**. В верхней части области **шаблонов** убедитесь, что **платформа .NET Framework 4** отображается в поле **"Целевая платформа** ". В области **"Шаблоны** " выберите **консольное приложение**. Введите имя проекта в поле **"Имя** ". Нажмите **OK**. Новый проект появится в **обозревателе решений**.

Добавление ссылки

В **обозревателе решений** щелкните правой кнопкой мыши имя проекта и выберите **команду Добавить ссылку**. Диалоговое окно **Добавление ссылки** появляется. На странице **.NET** выберите **Microsoft.Office.Interop.Word** в списке **имен компонентов** . Нажмите **OK**.

Добавьте необходимые директивы using

В **обозревателе решений** щелкните правой кнопкой мыши файл *Program.cs* и выберите **"Просмотреть код**". Добавьте следующие `using` директивы в начало файла кода:

```
C#

using Word = Microsoft.Office.Interop.Word;
```

Отображение текста в документе Word

`Program` В классе в *Program.cs* добавьте следующий метод для создания приложения Word и документа Word. Метод Add имеет четыре необязательных параметра. В этом примере используются значения по умолчанию. Поэтому в операторе вызова не требуется никаких аргументов.

> ⓘ **Примечание**

Чтобы избежать проблем с потоком COM и временем, которые могут привести к исключениям, таким как "Фильтр сообщений указывает, что приложение занято" (HRESULT 0x8001010A), приложение Word остается невидимым во время операций и становится видимым только после завершения всех операций.

C#

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    // Keep Word invisible during operations to avoid COM threading issues
    wordApp.Visible = false;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();

    // Make Word visible after operations are complete
    wordApp.Visible = true;
}
```

Добавьте следующий код в конце метода, чтобы определить место отображения текста в документе и отображаемый текст:

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Запуск приложения

Добавьте следующую инструкцию в Main:

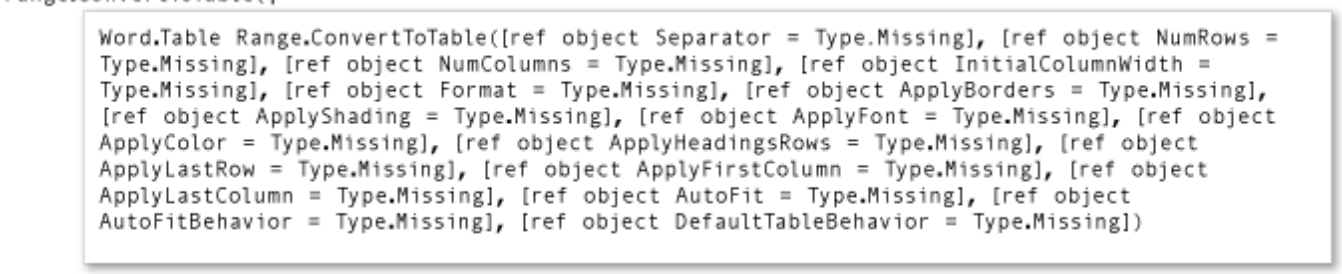
C#

```
DisplayInWord();
```

Нажмите клавиши CTRL + F5 , чтобы запустить проект. Появится документ Word, содержащий указанный текст.

Изменить текст на таблицу

Используйте метод `ConvertToTable`, чтобы заключить текст в таблицу. Метод имеет 16 необязательных параметров. IntelliSense заключает необязательные параметры в квадратные скобки, как показано на следующем рисунке. Значения `Type.Missing` по умолчанию — это простое имя для `System.Type.Missing`.



Именованные и необязательные аргументы позволяют указывать значения только для параметров, которые требуется изменить. Добавьте следующий код в конец метода `DisplayInWord` для создания таблицы. Аргумент указывает, что запятые в текстовой строке в `range` разделяют ячейки таблицы.

C#

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

Нажмите клавиши CTRL + F5 , чтобы запустить проект.

Эксперимент с другими параметрами

Измените таблицу так, чтобы она содержала один столбец и три строки, замените последнюю строку в `DisplayInWord` на следующую инструкцию и нажмите CTRL + F5.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

Укажите предопределенный формат таблицы, замените последнюю строку в `DisplayInWord` следующей инструкции и введите CTRL + F5. Формат может быть любым из

константа WdTableFormat .

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Пример

Следующий код содержит полный пример:

C#

```
using System;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            // Keep Word invisible during operations to avoid COM threading issues
            wordApp.Visible = false;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
```

```
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);

            // Make Word visible after all operations are complete
            wordApp.Visible = true;
        }
    }
}
```

Использование динамического типа

17.06.2025

Тип `dynamic` является статическим типом, но объект типа `dynamic` проходит проверку статического типа. В большинстве случаев он работает как тип `object`. Компилятор предполагает, что `dynamic` элемент поддерживает любую операцию. Поэтому не нужно определить, получает ли объект значение из COM-API из динамического языка, например IronPython, из модели объектов HTML-документа (DOM), из отражения или из другого места в программе. Однако если код недействителен, возникают ошибки во время выполнения.

Например, если метод `exampleMethod1` экземпляра в следующем коде имеет только один параметр, компилятор распознает, что первый вызов метода недействителен, `ec.exampleMethod1(10, 4)` так как он содержит два аргумента. Этот вызов приводит к ошибке компилятора. Компилятор не проверяет второй вызов метода, `dynamic_ec.exampleMethod1(10, 4)` так как тип `dynamic_ec` имеет значение `dynamic`. В результате ошибка компилятора не возникает. Однако ошибка не выходит заметок на неопределенный срок. Он отображается во время выполнения и вызывает исключение во время выполнения.

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
```

```
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

Роль компилятора в этих примерах заключается в том, чтобы упаковать сведения о том, что каждая инструкция предлагает сделать для `dynamic` объекта или выражения. Среда выполнения проверяет хранимую информацию и любую инструкцию, которая не является допустимой, вызывает исключение во время выполнения.

Результатом большинства динамических операций является сам `dynamic`. Например, при наведении указателя мыши на `testSum` в следующем примере IntelliSense отображает тип **(локальная переменная) dynamic testSum**.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Операции, в которых результат не `dynamic` включается:

- Преобразования из `dynamic` в другой тип.
- Вызовы конструктора, которые включают аргументы типа `dynamic`.

Например, `testInstance` в следующих объявлениях имеет тип `ExampleClass`, а не `dynamic`:

C#

```
var testInstance = new ExampleClass(d);
```

Преобразования

Преобразования динамических объектов в другие типы выполнять очень просто. Преобразования позволяют разработчику переключаться между динамическим и нединамических поведением.

Вы можете преобразовать любой объект `dynamic` в неявно, как показано в следующих примерах.

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

И наоборот, можно динамически применять любое неявное преобразование к любому выражению типа `dynamic`.

C#

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Разрешение перегрузки с аргументами типа dynamic

Разрешение перегрузки возникают во время выполнения, а не компиляции, если один или несколько аргументов в вызове метода имеют тип `dynamic` либо получатель вызова метода имеет тип `dynamic`. В следующем примере, если единственный доступный `exampleMethod2` метод принимает строковый аргумент, отправка `d1` как аргумент не приводит к ошибке компилятора, но вызывает исключение во время выполнения. Разрешение перегрузки завершается ошибкой во время выполнения, поскольку `d1` имеет тип времени выполнения `int`, а `exampleMethod2` требуется строка.

C#

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Среда DLR

Среда выполнения динамического языка (DLR) предоставляет инфраструктуру, которая поддерживает `dynamic` тип в C#, а также реализацию динамических языков программирования, таких как IronPython и IronRuby. Дополнительные сведения о DLR см. в разделе Общие сведения о среде DLR.

взаимодействие COM

Многие методы COM допускают вариативность в типах аргументов и возвращают тип, назначая типы как `object`. Взаимодействие COM требует явного приведения значений для координации с строго типизированными переменными в C#. При компиляции с использованием параметра **EmbedInteropTypes** (параметры компилятора C#) введение типа `dynamic` позволяет обрабатывать вхождения `object` в сигнатурах COM, как если бы они имели тип `dynamic`, и таким образом избежать большей части приведения. Дополнительные сведения об использовании `dynamic` типа с COM-объектами см. в статье о доступе к объектам взаимодействия Office с помощью функций C#.

Связанные статьи

Развернуть таблицу

Заголовок	Описание
Динамический	Описывает применение ключевого слова `dynamic`.
Общие сведения о среде DLR	Содержит общие сведения о среде DLR — среде выполнения, которая добавляет в общую среду языковой среды исполнения (CLR) набор служб для динамических языков.
Пошаговое руководство. Создание и использование динамических объектов	Содержит пошаговые инструкции по созданию пользовательских динамических объектов и проекта, который обращается к библиотеке `IronPython`.

Пошаговое руководство. Создание и использование динамических объектов в C#

17.06.2025

Динамические объекты предоставляют такие элементы, как свойства и методы во время выполнения, а не во время компиляции. Динамические объекты позволяют создавать объекты для работы со структурами, которые не соответствуют статичным типам или формату. Например, динамический объект можно использовать для ссылки на объектную модель HTML-документа (DOM), которая может содержать любое сочетание допустимых элементов разметки HTML и атрибутов. Так как каждый HTML-документ является уникальным, элементы для определенного HTML-документа определяются во время выполнения. Распространенный метод ссылки на атрибут html-элемента — передать имя атрибута `GetProperty` методу элемента. Чтобы ссылаться на `id` атрибут HTML-элемента `<div id="Div1">`, сначала получите ссылку на элемент `<div>`, а затем используйте `divElement.GetProperty("id")`. При использовании динамического объекта можно ссылаться на `id` атрибут как `divElement.id`.

Динамические объекты также обеспечивают удобный доступ к динамическим языкам, таким как IronPython и IronRuby. Динамический объект можно использовать для ссылки на динамический скрипт, интерпретируемый во время выполнения.

Вы обращаетесь к динамическому объекту, используя позднее связывание. Тип объекта с поздней привязкой указывается как `dynamic`. Дополнительные сведения см. в типе dynamic.

Вы можете создавать пользовательские динамические объекты, используя классы в пространстве имен System.Dynamic. Например, можно создать ExpandoObject и указать члены этого объекта во время выполнения. Вы также можете создать собственный тип, наследующий DynamicObject класс. Затем можно переопределить члены DynamicObject класса, чтобы обеспечить динамическую функциональность во время выполнения.

В этой статье содержатся два независимых пошагового руководства.

- Создайте пользовательский объект, динамически предоставляющий содержимое текстового файла в качестве свойств объекта.
- Создайте проект, использующий библиотеку `IronPython` .

Предпосылки

- Visual Studio 2022 версии 17.3 или более поздней версии с установленной рабочей нагрузкой **разработка приложений для настольных ПК .NET**. Пакет SDK для .NET 7 включается при выборе этой рабочей нагрузки.

> ⓘ **Примечание**
>
> На компьютере могут отображаться различные имена или расположения для некоторых элементов пользовательского интерфейса Visual Studio в следующих инструкциях. Выпуск Visual Studio, который у вас есть, и параметры, которые вы используете, определяют эти элементы. Дополнительные сведения см. в разделе **Персонализация интегрированной среды разработки**.

- Во втором пошаговом руководстве установите IronPython для .NET. Перейдите на страницу скачивания , чтобы получить последнюю версию.

Создание настраиваемого динамического объекта

В первом пошаговом руководстве определяется пользовательский динамический объект, который выполняет поиск содержимого текстового файла. Динамическое свойство определяет текст, который нужно искать. Например, если вызывающий код указывает `dynamicFile.Sample`, динамический класс возвращает обобщенный список строк, содержащий все строки из файла, которые начинаются с "Sample". Поиск не учитывает регистр. Динамический класс также поддерживает два необязательных аргумента. Первый аргумент — это значение перечисления параметра поиска, указывающее, что динамический класс должен искать совпадения в начале строки, конце строки или в любом месте строки. Второй аргумент указывает, что динамический класс должен обрезать начальные и конечные пробелы от каждой строки перед поиском. Например, если вызывающий код задает `dynamicFile.Sample(StringSearchOption.Contains)`, динамический класс ищет "Sample" в любом месте строки. Если вызывающий код указывает `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, динамический класс выполняет поиск "Sample" в начале каждой строки и не удаляет начальные и конечные пробелы. Поведение динамического класса по умолчанию заключается в поиске совпадения в начале каждой строки и удалении начальных и конечных пространств.

Создание настраиваемого динамического класса

Запустите Visual Studio. Выберите **Создать новый проект**. В диалоговом окне **"Создание проекта** " выберите C#, выберите **консольное приложение** и нажмите кнопку **"Далее**". В

диалоговом окне "**Настройка нового проекта** " введите `DynamicSample` **имя проекта** и нажмите кнопку "**Далее**". В диалоговом окне "**Дополнительные сведения**" **выберите .NET 7.0 (Current)** для **целевой платформы** и нажмите кнопку "**Создать**". В **обозревателе решений** щелкните правой кнопкой мыши проект DynamicSample и выберите "**Добавить**>**класс**". В поле "**Имя**" введите `ReadOnlyFile` и нажмите кнопку "**Добавить**". В верхней части *файла ReadOnlyFile.cs или ReadOnlyFile.vb* добавьте следующий код для импорта System.IO и System.Dynamic пространств имен.

C#

```
using System.IO;
using System.Dynamic;
```

Настраиваемый динамический объект использует перечисление для определения критериев поиска. Перед оператором класса добавьте следующее определение перечисления.

C#

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

Обновите определение класса, чтобы наследовать класс `DynamicObject`, как показано в примере кода ниже.

C#

```
class ReadOnlyFile : DynamicObject
```

Добавьте следующий код в `ReadOnlyFile` класс, чтобы определить частное поле для пути к файлу и конструктор для `ReadOnlyFile` класса.

C#

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
```

```
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

1. Добавьте следующий `GetPropertyValue` метод в `ReadOnlyFile` класс. Метод `GetPropertyValue` принимает в качестве входных данных, критерии поиска и возвращает строки из текстового файла, соответствующего этим критериям поиска. Динамические методы, `ReadOnlyFile` предоставляемые классом, вызывают `GetPropertyValue` метод для получения соответствующих результатов.

C#

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search
options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) {
```

```
results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

Добавьте следующий код после метода `GetPropertyValue`, чтобы переопределить метод TryGetMember класса DynamicObject. Метод TryGetMember вызывается при запросе члена динамического класса, и аргументы не указаны. Аргумент `binder` содержит сведения о указанном элементе, а `result` аргумент ссылается на результат, возвращаемый для указанного элемента. Метод TryGetMember возвращает логическое значение, возвращающее `true` , если запрошенный член существует; в противном случае возвращается `false`.

C#

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member
calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

Добавьте следующий код после метода `TryGetMember`, чтобы переопределить метод TryInvokeMember класса DynamicObject. Метод TryInvokeMember вызывается, когда член динамического класса запрашивается с аргументами. Аргумент `binder` содержит сведения о указанном элементе, а `result` аргумент ссылается на результат, возвращаемый для указанного элемента. Аргумент `args` содержит массив аргументов, передаваемых члену. Метод TryInvokeMember возвращает логическое значение, возвращающее `true` , если запрошенный член существует; в противном случае возвращается `false`.

Пользовательская версия метода ожидает, что первый аргумент будет значением из перечисления `TryInvokeMember`, который вы определили на предыдущем шаге. Метод `TryInvokeMember` ожидает, что второй аргумент будет булевым значением. Если один или оба аргумента являются допустимыми значениями, они передаются методу `GetPropertyValue` для получения результатов.

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                      object[] args,
                                      out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

Сохраните и закройте файл.

Создание примера текстового файла

В **обозревателе решений** щелкните правой кнопкой мыши проект DynamicSample и выберите "**Добавить**>**новый элемент**". В области "**Установленные шаблоны** " выберите "**Общие**" и выберите шаблон "**Текстовый файл** ". Оставьте имя по умолчанию *TextFile1.txt*

в поле "**Имя** ", а затем нажмите **кнопку** "**Добавить**". Скопируйте следующий текст в файл *TextFile1.txt* .

```text
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

Сохраните и закройте файл.

Создание примера приложения, использующего пользовательский динамический объект

В **обозревателе решений** дважды щелкните файл *Program.cs* . Добавьте следующий код в `Main` процедуру для создания экземпляра `ReadOnlyFile` класса для файла *TextFile1.txt* . Код использует позднюю привязку для вызова динамических элементов и получения строк текста, содержащих строку Customer.

```C#
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

Сохраните файл и нажмите клавиши `CTRL` + `F5` для сборки и запуска приложения.

Вызов динамической языковой библиотеки

В следующем пошаговом руководстве создается проект, который обращается к библиотеке, написанной на динамическом языке IronPython.

Создание настраиваемого динамического класса

В Visual Studio выберите **файл**>**нового**>**проекта**. В диалоговом окне "**Создание проекта** " выберите C#, выберите **консольное приложение** и нажмите кнопку "**Далее**". В диалоговом окне "**Настройка нового проекта** " введите `DynamicIronPythonSample`**имя проекта** и нажмите кнопку "**Далее**". В диалоговом окне "**Дополнительные сведения**" **выберите .NET 7.0 (Current)** для **целевой платформы** и нажмите кнопку "**Создать**". Установите пакет NuGet IronPython . Измените *файл Program.cs* . В верхней части файла добавьте следующий код, чтобы импортировать пространства имен `Microsoft.Scripting.Hosting` и `IronPython.Hosting` из библиотек IronPython, а также пространство имен `System.Linq`.

C#

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

В методе Main добавьте следующий код, чтобы создать новый `Microsoft.Scripting.Hosting.ScriptRuntime` объект для размещения библиотек IronPython. Объект `ScriptRuntime` загружает модуль библиотеки IronPython random.py.

C#

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

После загрузки модуля random.py добавьте следующий код, чтобы создать массив целых чисел. Массив передается методу `shuffle` модуля random.py, который случайно сортирует значения в массиве.

C#

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("--------------------");
}
```

Сохраните файл и нажмите клавиши `CTRL` + `F5` для сборки и запуска приложения.

См. также

- System.Dynamic
- System.Dynamic.DynamicObject
- Использование динамического типа
- динамические
- Реализация динамических интерфейсов (скачиваемый PDF-файл из Microsoft TechNet)

Сокращение выделения памяти с помощью новых функций C#

17.06.2025

> **ⓘ Важно!**
>
> Описанные в этом разделе методы повышают производительность при *применении к горячим путям* в коде. *Горячие пути* — это те разделы базы кода, которые выполняются часто и многократно в обычных операциях. Применение этих методов к коду, который не часто выполняется, будет иметь минимальное влияние. Перед внесением изменений для улучшения производительности важно измерить исходный уровень. Затем проанализируйте базовые показатели, чтобы определить, где происходят узкие места памяти. Вы можете узнать о многих кроссплатформенных средствах для измерения производительности приложения в разделе "Диагностика и инструментирование". Вы можете практиковать сеанс профилирования в руководстве по измерению использования памяти в документации По Visual Studio.
>
> После измерения использования памяти и определения того, что можно уменьшить объем выделений, используйте методы, описанные в этом разделе, чтобы сократить объем выделений. После каждого последовательного изменения снова измеряйте использование памяти. Убедитесь, что каждое изменение оказывает положительное влияние на использование памяти в приложении.

Производительность работы в .NET часто означает удаление выделений из кода. Каждый блок памяти, который вы выделяете, в конечном итоге должен быть освобожден. Уменьшение количества выделений сокращает время, потраченное на сбор мусора. Это позволяет более предсказуемо контролировать время выполнения, удалив сборщики мусора из определенных участков кода.

Распространенная тактика сокращения выделения заключается в изменении критически важных структур данных с `class` типов на `struct` типы. Это изменение влияет на семантику использования этих типов. Параметры и возвращаемые значения теперь передаются по значению, вместо ссылки. Стоимость копирования значения незначительна, если объем типов мал — три слова или меньше (учитывая, что размер одного слова соответствует размеру одного целого числа). Это измеряемое значение, которое может реально повлиять на производительность крупных типов данных. Чтобы бороться с эффектом копирования, разработчики могут передать эти типы через `ref`, чтобы сохранить их предполагаемую семантику.

Функции C# `ref` позволяют выразить нужную семантику для `struct` типов, не влияя на общую удобство использования. До этих улучшений разработчикам необходимо использовать `unsafe` конструкции с указателями и необработанной памятью, чтобы добиться того же влияния на производительность. Компилятор создает *проверенный безопасный код* для новых `ref` связанных функций. *Проверенный безопасный код* означает, что компилятор обнаруживает возможные переполнения буфера или доступ к нераспределенной или освобожденной памяти. Компилятор обнаруживает и предотвращает некоторые ошибки.

Передача и возврат по ссылке

Переменные в C# хранят *значения*. В `struct` типах значение — это содержимое экземпляра типа. В `class` типах значение является ссылкой на блок памяти, в который хранится экземпляр типа. `ref` Добавление модификатора означает, что переменная сохраняет *ссылку* на значение. В `struct` типах ссылка указывает на хранилище, содержащее значение. В `class` типах ссылка указывает на хранилище, содержащее ссылку на блок памяти.

В C#параметры методов *передаются по значению*, а *возвращаемые значения возвращаются по значению*. *Значение* аргумента передается методу. *Значение* возвращаемого аргумента — возвращаемое значение.

Модификатор `ref`, `in`, `ref readonly`, или `out` указывает, что аргумент *передается по ссылке*. *Ссылка на* расположение хранилища передается методу. Добавление `ref` к сигнатуре метода означает, что возвращаемое значение *возвращается по ссылке*. *Ссылка на* расположение хранилища — это возвращаемое значение.

Вы также можете использовать *назначение ссылок*, чтобы переменная ссылалась на другую переменную. Обычное присваивание копирует *значение* правой стороны в переменную на левой стороне присваивания. *Присваивание ссылки* копирует адрес памяти переменной справа в переменную слева. Теперь `ref` ссылается на исходную переменную:

C#

```
int anInteger = 42; // assignment.
ref int location = ref anInteger; // ref assignment.
ref int sameLocation = ref location; // ref assignment

Console.WriteLine(location); // output: 42

sameLocation = 19; // assignment
```

```
Console.WriteLine(anInteger); // output: 19
```

При *назначении* переменной измените его *значение*. При назначении ссылки переменной, вы изменяете то, на что она ссылается.

Вы можете работать непосредственно с хранилищем для значений, используя `ref` переменные, передачу параметров по ссылке и присвоение с помощью ref. Правила области, применяемые компилятором, обеспечивают безопасность при работе непосредственно с хранилищем.

`ref readonly` И `in` модификаторы указывают, что аргумент должен передаваться по ссылке и не может быть переназначирован в методе. Разница заключается в том, что `ref readonly` метод использует параметр в качестве переменной. Метод может использовать параметр или вернуть его через ссылку только для чтения. В этих случаях следует использовать `ref readonly` модификатор. `in` В противном случае модификатор обеспечивает большую гибкость. Не нужно добавлять модификатор `in` к аргументу для параметра `in`, так что вы можете безопасно обновлять существующие сигнатуры API, используя модификатор `in`. Компилятор выдает предупреждение, если вы не добавляете `ref` или `in` модификатор в аргумент параметра `ref readonly` .

Безопасный контекст ссылки

C# содержит правила для `ref` выражений, чтобы `ref` выражение не могло быть доступным, когда хранилище, на которое оно ссылается, больше не является действительным. Рассмотрим следующий пример:

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe context is the body of CantEscape
}
```

Компилятор сообщает об ошибке, так как невозможно вернуть ссылку на локальную переменную из метода. Вызывающий объект не может получить доступ к указанному хранилищу. Безопасный *контекст ссылки* определяет область, в которой `ref` выражение безопасно для доступа или изменения. В следующей таблице перечислены *безопасные контексты ссылок* для типов переменных. `ref` поля не могут быть объявлены в `class` или в `struct` без ссылки, поэтому эти строки не указаны в таблице.

Развернуть таблицу

Декларация	***Безопасный контекст ссылок***
локальный без ссылки	блок, где объявлена локальная переменная
Параметр, отличный от ссылок	текущий метод
`ref`, `ref readonly`, `in` параметр	метод вызова
`out` параметр	текущий метод
Поле `class`	метод вызова
Поле, не являющееся ссылкой `struct`	текущий метод
`ref` поле `ref struct`	метод вызова

Переменная может быть `ref` возвращена, если её *безопасный контекст ссылки* является вызывающим методом. Если его *безопасный контекст ссылки* является текущим методом или блоком, возвращение запрещено. В следующем фрагменте кода показаны два примера. Поле-член можно получить из области вызова метода, поэтому *безопасный контекст поля ссылок* класса или структуры является вызывающим методом. *Контекст безопасной ссылки* для параметра с `ref` или `in` модификаторами — это весь метод. Оба могут быть `ref` возвращены из метода-члена:

C#

```
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

> ⓘ **Примечание**
>
> Если модификатор `ref readonly` или `in` применяется к параметру, этот параметр может быть возвращен через `ref readonly`, а не `ref`.

Компилятор гарантирует, что ссылка не может избежать *безопасного контекста ссылки*. Вы можете безопасно использовать `ref` параметры, `ref return`, и `ref` локальные переменные, так как компилятор обнаруживает, если вы случайно написали код, в котором можно получить доступ к выражению `ref`, когда его хранилище недействительно.

Безопасные структуры контекста и ссылок

`ref struct` типы требуют больше правил, чтобы обеспечить их безопасное использование. Тип `ref struct` может включать `ref` поля. Для этого требуется введение *безопасного контекста*. Для большинства типов *безопасный контекст* является вызывающим методом. Другими словами, значение, которое не `ref struct` всегда может быть возвращено из метода.

Неформальный *безопасный контекст* для `ref struct` определяется областью, в которой ко всем его `ref` полям можно получить доступ. Другими словами, это пересечение *безопасного контекста ссылок* всех его `ref` полей. Следующий метод возвращает `ReadOnlySpan<char>` к члену поля, так что его *безопасный контекст* — это сам метод.

C#

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

В отличие от этого, следующий код выдает ошибку, так как `ref field` член `Span<int>` ссылается на массив целых чисел, выделенный в стеке. Он не может избежать метода

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Объединение типов памяти

Введение System.Span<T> и System.Memory<T> предоставляют единую модель для работы с памятью. System.ReadOnlySpan<T> и System.ReadOnlyMemory<T> предоставьте доступные для чтения версии для доступа к памяти. Все они предоставляют абстракцию по блоку памяти, в котором хранится массив похожих элементов. Разница заключается в том, что `Span<T>` и `ReadOnlySpan<T>` являются типами `ref struct`, тогда как `Memory<T>` и `ReadOnlyMemory<T>` являются типами `struct`. Диапазоны содержат `ref field`. Поэтому экземпляры цепочки не могут покинуть свой безопасный контекст. Безопасный *контекст* объекта `ref struct` — это *безопасный контекст ссылки* его `ref field`. Реализация `Memory<T>` и `ReadOnlyMemory<T>` устраняет это ограничение. Эти типы используются для прямого доступа к буферам памяти.

Повышение производительности с помощью безопасности указателей

Использование этих функций для повышения производительности включает следующие задачи:

- *Избегайте распределения ресурсов*: При изменении типа данных с `class` на `struct`, вы изменяете способ его хранения. Локальные переменные хранятся в стеке. Члены хранятся встроенно, когда выделяется объект контейнера. Это изменение означает меньше выделений и уменьшает работу сборщика мусора. Это также может снизить давление памяти, поэтому сборщик мусора работает реже.
- *Сохранение ссылочной семантики*: изменение типа из `class` в `struct` изменяет семантику передачи переменной в метод. Код, изменяющий состояние параметров, нуждается в изменении. Теперь, когда параметр является параметром `struct`, метод изменяет копию исходного объекта. Вы можете восстановить исходную семантику, передав этот параметр в `ref` качестве параметра. После этого изменения метод изменяет исходный файл `struct` еще раз.
- *Избегайте копирования данных*: копирование больших `struct` типов может повлиять на производительность в некоторых путях кода. Вы также можете добавить модификатор `ref`, чтобы передавать более крупные структуры данных методам по ссылке, а не по значению.
- *Ограничение изменений*: когда `struct` тип передается по ссылке, вызываемый метод может изменить состояние структуры. Вы можете заменить модификатор `ref` на модификаторы `ref readonly` или `in`, чтобы указать, что аргумент нельзя изменить. Предпочитайте `ref readonly`, когда метод захватывает параметр или возвращает его по ссылке только для чтения. Вы также можете создавать `readonly struct` типы или

`struct` типы с `readonly` элементами, чтобы обеспечить более широкий контроль над тем, какие элементы `struct` можно изменить.

- *Непосредственное управление памятью*: некоторые алгоритмы наиболее эффективны при обработке структур данных в виде блока памяти, содержащего последовательность элементов. Типы `Span` и `Memory` обеспечивают безопасный доступ к блокам памяти.

Ни один из этих методов не требует кода `unsafe` . При грамотном использовании вы можете получить характеристики производительности из безопасного кода, которые ранее были возможны только с помощью небезопасных методов. Вы можете попробовать методы, описанные в руководстве по сокращению выделения памяти.

Руководство по сокращению выделения памяти с помощью `ref` защищённости

17.06.2025

Часто настройка производительности для приложения .NET включает два метода. Во-первых, уменьшите количество и размер выделений памяти в куче. Во-вторых, уменьшите частоту копирования данных. Visual Studio предоставляет отличные средства , которые помогают анализировать использование памяти приложения. Когда вы определите, где приложение делает ненужные выделения, вы вносите изменения, чтобы свести к минимуму эти выделения. Вы преобразуете типы `class` в типы `struct`. Вы используете `ref` функции безопасности для сохранения семантики и минимизации дополнительных копий.

Используйте Visual Studio 17.5 для получения наилучшего опыта с этим учебником. Средство выделения объектов .NET, используемое для анализа использования памяти, является частью Visual Studio. Вы можете использовать Visual Studio Code и командную строку для запуска приложения и внесения всех изменений. Однако вы не сможете просмотреть результаты анализа изменений.

Приложение, которое вы будете использовать, — это симуляция IoT-приложения, которое отслеживает несколько датчиков, чтобы определить, вошел ли злоумышленник в секретную галерею с ценными экспонатами. Датчики Интернета вещей постоянно отправляют данные, которые измеряют сочетание кислорода (O2) и углекислого газа (CO2) в воздухе. Они также сообщают о температуре и относительной влажности. Каждое из этих значений изменяется немного все время. Однако, когда человек входит в комнату, изменение немного больше, и всегда в том же направлении: кислород уменьшается, углекислый газ увеличивается, температура увеличивается, как и относительная влажность. Когда показания датчиков увеличиваются, срабатывает сигнализация о вторжении.

В этом руководстве вы запустите приложение, выполните измерения по выделению памяти, а затем улучшите производительность, уменьшая количество выделений. Исходный код доступен в браузере примеров.

Изучение начального приложения

Скачайте приложение и запустите начальный пример. Начальное приложение работает правильно, но так как оно выделяет множество небольших объектов с каждым циклом измерения, его производительность медленно снижается по мере выполнения с течением времени.

Консоль

```
Press <return> to start simulation

Debounced measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906
Average measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906

Debounced measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
Average measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
```

Многие строки удалены.

Консоль

```
Debounced measurements:
    Temp:      67.597
    Humidity:  46.543%
    Oxygen:    19.021%
    CO2 (ppm): 429.149
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High

Debounced measurements:
    Temp:      67.602
    Humidity:  46.835%
    Oxygen:    19.003%
    CO2 (ppm): 429.393
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
```

```
Current intruders: 3
Calculated intruder risk: High
```

Вы можете изучить код, чтобы узнать, как работает приложение. Основная программа выполняет имитацию. После нажатия `<Enter>` клавиши он создает комнату и собирает некоторые начальные базовые данные:

C#

```
Console.WriteLine("Press <return> to start simulation");
Console.ReadLine();
var room = new Room("gallery");
var r = new Random();

int counter = 0;

room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        Console.WriteLine();
        counter++;
        return counter < 20000;
    });
```

После установки базовых данных он запускает имитацию в комнате, где генератор случайных чисел определяет, вошел ли злоумышленник в комнату:

C#

```
counter = 0;
room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        room.Intruders += (room.Intruders, r.Next(5)) switch
        {
            ( > 0, 0) => -1,
            ( < 3, 1) => 1,
            _ => 0
        };

        Console.WriteLine($"Current intruders: {room.Intruders}");
        Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
        Console.WriteLine();
        counter++;
        return counter < 200000;
    });
```

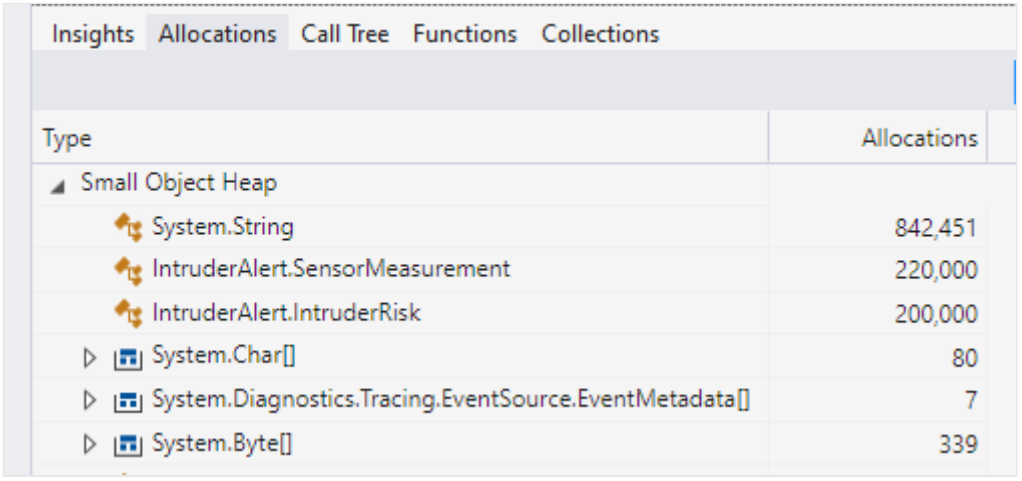
Другие типы содержат измерения, устраненное дребезжание, среднее значение последних 50 измерений, а также среднее значение всех произведенных измерений.

Затем запустите приложение с помощью средства выделения объектов .NET. Убедитесь, что вы используете сборку `Release` , а не сборку `Debug` . В меню *отладки* откройте *профилировщик производительности*. Проверьте параметр *отслеживания выделения объектов .NET* , но ничего другого. Запустите приложение до завершения. Профилировщик измеряет выделение объектов и сообщает о выделениях и цикле сборки мусора. Вы увидите граф, аналогичный следующему изображению:



На предыдущем графике показано, что работа с сокращением распределения обеспечит преимущества производительности. В графике активных объектов отображается шаблон пилы. Это говорит о том, что создаются многочисленные объекты, которые быстро становятся мусором. Позже их собирают, как показано на графике дельта объекта. Вниз красные полосы указывают цикл сборки мусора.

Затем перейдите на вкладку "*Выделения*" под графами. В этой таблице показано, какие типы выделяются чаще всего:



Тип System.String учитывает большинство выделений. Наиболее важной задачей является минимизация частоты выделения строк. Это приложение постоянно печатает многочисленные отформатированные выходные данные в консоли. Для этой симуляции важно сохранить сообщения, поэтому мы сосредоточим внимание на следующих двух строках: тип `SensorMeasurement` и тип `IntruderRisk`.

Дважды щелкните `SensorMeasurement` строку. Вы можете увидеть, что все выделения осуществляются в методе `static` `SensorMeasurement.TakeMeasurement`. Этот метод можно увидеть в следующем фрагменте кода:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 * generator.NextDouble() -
10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Каждое измерение выделяет новый `SensorMeasurement` объект, который является типом `class` . Каждое `SensorMeasurement` созданное вызывает выделение кучи.

Замена классов на структуры

В следующем коде показано начальное объявление `SensorMeasurement`:

C#

```
public class SensorMeasurement
{
    private static readonly Random generator = new Random();

    public static SensorMeasurement TakeMeasurement(string room, int intruders)
    {
        return new SensorMeasurement
        {
            CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
            O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
            Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
            Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
            Room = room,
```

```
            TimeRecorded = DateTime.Now
        };
    }

    private const double CO2Concentration = 409.8; // increases with people.
    private const double O2Concentration = 0.2100; // decreases
    private const double TemperatureSetting = 67.5; // increases
    private const double HumiditySetting = 0.4500; // increases

    public required double CO2 { get; init; }
    public required double O2 { get; init; }
    public required double Temperature { get; init; }
    public required double Humidity { get; init; }
    public required string Room { get; init; }
    public required DateTime TimeRecorded { get; init; }

    public override string ToString() => $"""
            Room: {Room} at {TimeRecorded}:
                Temp:      {Temperature:F3}
                Humidity:  {Humidity:P3}
                Oxygen:    {O2:P3}
                CO2 (ppm): {CO2:F3}
            """;
}
```

Тип изначально был создан как объект `class` , так как он содержит многочисленные `double` измерения. Это больше, чем вы хотите скопировать в горячие пути. Однако это решение означает большое количество выделений. Измените тип с `class` на `struct`.

Переход с `class` на `struct` приводит к появлению нескольких ошибок компилятора, так как в оригинальном коде в нескольких местах использовались проверки ссылок `null`. Первый — в `DebounceMeasurement` классе, в методе `AddMeasurement` :

C#

```
public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i] is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
```

```
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
}
```

Тип `DebounceMeasurement` содержит массив из 50 измерений. Значения для датчика отображаются в среднем за последние 50 измерений. Это снижает шум в чтениях. Прежде чем были произведены полные 50 измерений, эти значения `null`. Код проверяет наличие `null` ссылки для вывода правильное среднее значение при запуске системы. После изменения `SensorMeasurement` типа в структуру необходимо использовать другой тест. Тип `SensorMeasurement` содержит `string`, который служит идентификатором комнаты, поэтому вы можете использовать этот тест вместо:

C#

```
if (recentMeasurements[i].Room is not null)
```

Остальные три ошибки компилятора находятся в методе, который неоднократно принимает измерения в комнате:

C#

```
public void TakeMeasurements(Func<SensorMeasurement, bool> MeasurementHandler)
{
    SensorMeasurement? measure = default;
    do {
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    } while (MeasurementHandler(measure));
}
```

В начальном методе локальная переменная для этой `SensorMeasurement` переменной является *ссылкой, допускающей значение NULL*:

C#

```
SensorMeasurement? measure = default;
```

Теперь, когда `SensorMeasurement` является `struct` вместо `class`, nullable является *типом значений, допускающим null*. Вы можете изменить объявление на тип значения, чтобы исправить оставшиеся ошибки компилятора:

```
C#

SensorMeasurement measure = default;
```

Теперь, когда были устранены ошибки компилятора, необходимо проверить код, чтобы убедиться, что семантика не изменилась. Так как `struct` типы передаются по значению, изменения, внесенные в параметры метода, не отображаются после возврата метода.

> **ⓘ Важно!**
>
> Изменение типа с `class` на `struct` может изменить семантику вашей программы. `class` При передаче типа в метод все изменения, внесенные в метод, вносятся в аргумент. Когда `struct` тип передается в метод, изменения, сделанные в методе, применяются к *копии* аргумента. Это означает, что любой метод, который по задумке изменяет свои аргументы, следует обновить для использования модификатора `ref` для любого типа аргумента, который вы изменили с `class` на `struct`.

Тип `SensorMeasurement` не включает методы, которые изменяют состояние, поэтому это не проблема в этом примере. Вы можете доказать это, добавив модификатор `readonly` к структуре `SensorMeasurement`.

```
C#

public readonly struct SensorMeasurement
```

Компилятор обеспечивает `readonly` свойства `SensorMeasurement` структуры. Если проверка кода пропустила какой-то метод, который изменил состояние, компилятор сообщит вам. Ваше приложение по-прежнему выполняет сборку без ошибок, поэтому этот тип является `readonly`. Добавление модификатора `readonly` при изменении типа с `class` на `struct` поможет вам найти элементы, которые изменяют состояние типа `struct`.

Избегайте копирования

Вы удалили большое количество ненужных выделений из приложения. Тип `SensorMeasurement` не отображается в таблице нигде.

Теперь это делает дополнительную работу, копируя `SensorMeasurement` структуру каждый раз, когда она используется в качестве параметра или возвращаемого значения. Структуру `SensorMeasurement` содержит четыре двойника, а DateTime и а `string`. Эта структура значительно больше, чем эталон. Добавим модификаторы `ref` или `in` в места, где используется тип `SensorMeasurement`.

Следующий шаг — найти методы, возвращающие измерение, или принять измерение в качестве аргумента, и использовать ссылки, где это возможно. Начните в структуре `SensorMeasurement`. Статический `TakeMeasurement` метод создает и возвращает новое `SensorMeasurement`:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 * generator.NextDouble() -
10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Мы оставим это как есть и будем возвращать по значению. Если вы попытаетесь выполнить возвращение с помощью `ref`, вы получите ошибку компилятора. Невозможно вернуть `ref` в новую структуру, созданную локально в методе. Конструкция неизменяемой структуры означает, что можно задать только значения измерения при построении. Этот метод должен создать новую структуру измерения.

Давайте еще раз рассмотрим `DebounceMeasurement.AddMeasurement`. Необходимо добавить модификатор `in` к параметру `measurement`.

C#

```
public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
```

```
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
}
```

Это сохраняет одну операцию копирования. Параметр `in` является ссылкой на копию, уже созданную вызывающим оператором. Вы также можете сохранить копию с помощью метода `TakeMeasurement` в типе `Room`. Этот метод иллюстрирует, как компилятор обеспечивает типовую безопасность при передаче аргументов с помощью `ref`. Начальный `TakeMeasurement` метод в типе `Room` принимает аргумент `Func<SensorMeasurement, bool>`. Если вы попытаетесь добавить модификатор `in` или `ref` к этому объявлению, компилятор сообщит об ошибке. Нельзя передавать аргумент типа `ref` в лямбда-выражение. Компилятор не может гарантировать, что вызываемое выражение не копирует ссылку. Если лямбда-выражение *захватывает* ссылку, ссылка может иметь более продолжительное время существования, чем значение, на которое она ссылается. Доступ к нему за пределами *безопасного контекста ссылки* приведет к повреждению памяти. Правила `ref` безопасности этого не разрешают. Дополнительные сведения см. в обзоре функций безопасности ссылок.

Сохранение семантики

Окончательные наборы изменений не влияют на производительность этого приложения, так как типы не создаются в горячих путях. Эти изменения иллюстрируют некоторые другие методы, которые вы использовали бы в настройке производительности. Рассмотрим начальный `Room` класс:

```
C#
```

```
public class Room
{
    public AverageMeasurement Average { get; } = new ();
    public DebounceMeasurement Debounce { get; } = new ();
    public string Name { get; }

    public IntruderRisk RiskStatus
    {
        get
        {
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 / 4;
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 / 4.0;
            var TempVariance = (Debounce.Temperature - Average.Temperature) > 0.05
/ 4.0;
            var HumidityVariance = (Debounce.Humidity - Average.Humidity) > 0.20 /
4;
            IntruderRisk risk = IntruderRisk.None;
            if (CO2Variance) { risk++; }
            if (O2Variance) { risk++; }
            if (TempVariance) { risk++; }
            if (HumidityVariance) { risk++; }
            return risk;
        }
    }

    public int Intruders { get; set; }


    public Room(string name)
    {
        Name = name;
    }

    public void TakeMeasurements(Func<SensorMeasurement, bool> MeasurementHandler)
    {
        SensorMeasurement? measure = default;
        do {
            measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
            Average.AddMeasurement(measure);
            Debounce.AddMeasurement(measure);
        } while (MeasurementHandler(measure));
    }
}
```

Этот тип содержит несколько свойств. Некоторые из них являются `class` типами. Создание объекта `Room` включает несколько выделений. Один для самого `Room`, и один для каждого члена типа `class`, который он содержит. Вы можете преобразовать два из этих свойств: из типов `class` в типы `struct`, а именно типы `DebounceMeasurement` и `AverageMeasurement`. Давайте рассмотрим это преобразование с обоими типами.

Измените тип `DebounceMeasurement` с `class` на `struct`. Это представляет ошибку `CS8983: A 'struct' with field initializers must include an explicitly declared constructor` компилятора. Это можно исправить, добавив пустой конструктор без параметров:

```
C#

public DebounceMeasurement() { }
```

Дополнительные сведения об этом требовании см. в справочной статье по языку о структурам.

Переопределение Object.ToString() не изменяет ни одно из значений структуры. Можно добавить модификатор `readonly` в данное объявление метода. Тип `DebounceMeasurement` является *изменяемым*, поэтому необходимо позаботиться о том, что изменения не влияют на копии, которые удаляются. Метод `AddMeasurement` изменяет состояние объекта. Он вызывается из `Room` класса в методе `TakeMeasurements` . Вы хотите, чтобы эти изменения сохранялись после вызова метода. Свойство `Room.Debounce` можно изменить, чтобы оно возвращало *ссылку* на один экземпляр типа `DebounceMeasurement`.

```
C#

private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref debounce; } }
```

В предыдущем примере есть несколько изменений. Во-первых, *свойство* является свойством только для чтения и возвращает ссылку только для чтения на экземпляр, принадлежащий этой комнате. Теперь это поддерживается объявленным полем, которое инициализируется при создании экземпляра объекта `Room`. После внесения этих изменений вы обновите реализацию `AddMeasurement` метода. Он использует приватное резервное поле `debounce`, а не свойство `Debounce` только для чтения. Таким образом, изменения происходят на одном экземпляре, созданном во время инициализации.

Тот же метод работает со свойством `Average` . Во-первых, вы измените `AverageMeasurement` тип из `class` на `struct` и добавьте `readonly` модификатор к `ToString` метод.

```
C#

namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
```

```
    private double sumO2 = 0;
    private double sumTemperature = 0;
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
    public readonly double Temperature => sumTemperature / totalMeasurements;
    public readonly double Humidity => sumHumidity / totalMeasurements;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
        Average measurements:
            Temp:      {Temperature:F3}
            Humidity:  {Humidity:P3}
            Oxygen:    {O2:P3}
            CO2 (ppm): {CO2:F3}
        """;
}
```

Затем вы измените `Room` класс, следуя тому же методу, который использовался для `Debounce` свойства. Свойство `Average` возвращает `readonly ref` частное поле для среднего измерения. Метод `AddMeasurement` изменяет внутренние поля.

C#

```
private AverageMeasurement average = new();
public  ref readonly AverageMeasurement Average { get { return ref average; } }
```

Избегайте бокса

Существует одно окончательное изменение для повышения производительности. Основная программа выводит статистику по комнате, включая оценку рисков.

C#

```
Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
```

Вызов созданных `ToString` полей задает значение перечисления. Это можно избежать, написав переопределение в `Room` классе, который форматирует строку на основе значения предполагаемого риска:

C#

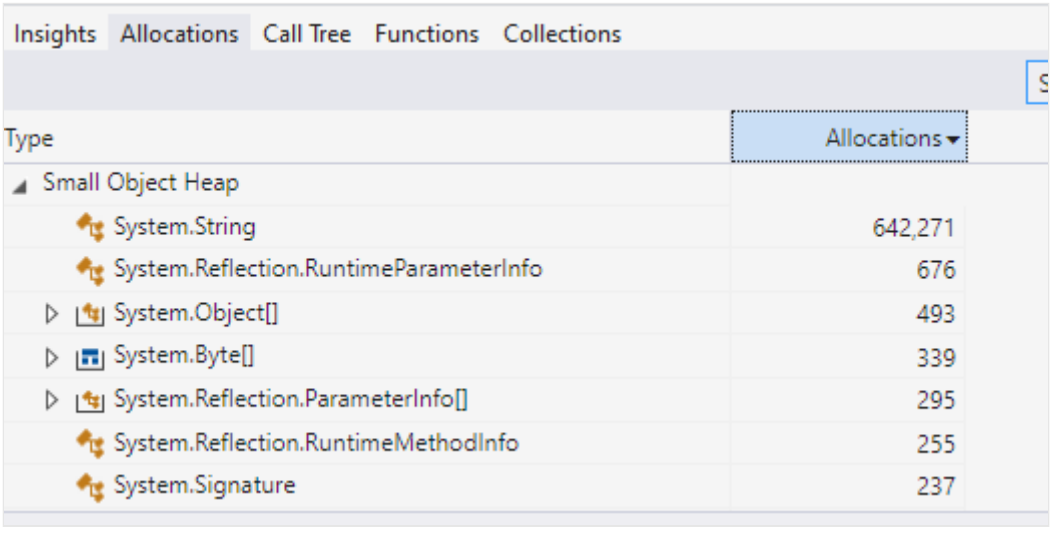
```
public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";
```

Затем измените код в главной программе, чтобы вызвать этот новый `ToString` метод:

C#

```
Console.WriteLine(room.ToString());
```

Запустите приложение с помощью профилировщика и просмотрите обновленную таблицу для выделений.



Вы удалили многочисленные выделения и предоставили приложению повышение производительности.

Использование безопасного обращения с ссылками в приложении

Эти методы являются низкоуровневой настройкой производительности. Они могут повысить производительность вашего приложения, если применены к критическим участкам, и при измерении влияния этих изменений до и после их внедрения. В большинстве случаев цикл, которого вы будете придерживаться:

- *Распределение мер*: определите, какие типы выделяются чаще всего, и когда можно уменьшить выделение кучи.
- *Преобразование класса в структуру*: во многих случаях типы можно преобразовать из типа `class` в структуру `struct`. Ваше приложение использует пространство стека вместо выделения памяти в куче.
- *Сохранение семантики*: преобразование `class` в `struct` может повлиять на семантику параметров и возвращаемых значений. Теперь любой метод, изменяющий его параметры, должен пометить эти параметры модификатором `ref` . Это гарантирует, что изменения вносятся в правильный объект. Аналогичным образом, если значение возвращаемого свойства или метода должно быть изменено вызывающей стороной, то возвращаемое значение должно быть отмечено модификатором `ref` .
- *Избегайте копирования*. При передаче большой структуры в качестве параметра можно пометить параметр модификатором `in` . Можно передать ссылку в меньшем количестве байтов и убедиться, что метод не изменяет исходное значение. Можно также возвращать значения, `readonly ref` возвращая ссылку, которая не может быть изменена.

С помощью этих методов можно повысить производительность в горячих путях кода.

Руководство. Создание пользовательского обработчика интерполяции строк

В этом руководстве описано, как:

- ✓ Реализуйте шаблон обработчика интерполяции строк.
- ✓ Взаимодействие с приемником в операциях интерполяции строк.
- ✓ Добавьте аргументы в обработчик интерполяции строк.
- ✓ Ознакомьтесь с новыми функциями библиотеки для интерполяции строк.

Необходимые условия

Настройте компьютер для запуска .NET. Компилятор C# доступен через Visual Studio и пакет SDK для .NET.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или Visual Studio Code и C# DevKit.

Можно написать настраиваемый *интерполированный обработчик строк*. Интерполированный обработчик строк — это тип, обрабатывающий выражение заполнителя в интерполированной строке. Без пользовательского обработчика система обрабатывает заполнители наподобие String.Format. Каждый заполнитель форматируется как текст, а затем компоненты объединяются для формирования результирующей строки.

Обработчик можно написать для любого сценария, где используются сведения о результирующей строке. Рассмотрим такие вопросы: используется ли она? Какие ограничения относятся к формату? Ниже приведены некоторые примеры:

- Может потребоваться, чтобы ни одна из результирующей строки не превышала некоторые ограничения, например 80 символов. Можно обработать интерполированные строки для заполнения буфера фиксированной длины и остановить обработку после достижения этой длины буфера.
- У вас может быть табличный формат, и каждый местозаполнитель должен иметь фиксированную длину. Настраиваемый обработчик может применить это ограничение, а не принудительно соблюдать весь клиентский код.

В этом руководстве вы создадите обработчик интерполяции строк для одного из основных сценариев производительности: библиотеки ведения журнала. В зависимости от установленного уровня ведения журнала, работу по созданию сообщения журнала можно не выполнять. Если ведение журнала отключено, работа по созданию строки из

интерполированного строкового выражения не требуется. Сообщение никогда не печатается, поэтому любое объединение строк можно пропустить. Кроме того, любые выражения, используемые в заполнителях, включая создание трассировок стека, не требуется выполнять.

Интерполированный обработчик строк может определить, используется ли форматированная строка, и при необходимости выполнить только необходимую работу.

Начальная реализация

Начните с базового `Logger` класса, поддерживающего различные уровни:

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

Эта `Logger` поддерживает шесть различных уровней. Если сообщение не проходит фильтр уровня логирования, логгер не создает выход. Общедоступный API для средства ведения журнала принимает полностью отформатированную строку в качестве сообщения. Вызывающий объект выполняет все действия для создания строки.

Реализуйте шаблон обработчика

На этом шаге вы создадите *интерполированный обработчик строки* , который воссоздает текущее поведение. Интерполированный обработчик строк — это тип, который должен иметь следующие характеристики:

- System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute был применён к типу.
- Конструктор с двумя параметрами `int`, `literalLength` и `formattedCount`. (Разрешено больше параметров).
- Открытый метод `AppendLiteral` с сигнатурой: `public void AppendLiteral(string s)`.
- Универсальный открытый `AppendFormatted` метод с сигнатурой: `public void AppendFormatted<T>(T t)`.

Внутри построителя создается форматированная строка и предоставляется метод, который позволяет клиенту получить эту строку. В следующем коде показан тип `LogInterpolatedStringHandler`, соответствующий этим требованиям:

C#

```
[InterpolatedStringHandler]
public struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }

    public override string ToString() => builder.ToString();
}
```

ⓘ **Примечание.**

Если интерполированное строковое выражение является константой времени компиляции (т. е. не имеет заполнителей), компилятор использует целевой тип `string` вместо вызова пользовательского интерполированного обработчика строки. Это поведение означает, что интерполированные строки полностью обходят пользовательские обработчики.

Теперь можно добавить перегрузку для `LogMessage` в классе `Logger`, чтобы попробовать новый обработчик интерполированной строки.

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.ToString());
}
```

Вам не нужно удалять исходный `LogMessage` метод. Если аргумент является интерполированным строковым выражением, компилятор предпочитает метод с параметром интерполированного обработчика над методом с параметром `string` .

Вы можете убедиться, что новый обработчик вызывается с помощью следующего кода в качестве основной программы:

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This is an
error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This won't be
printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a string, not
an interpolated string expression.");
```

Запуск приложения создает выходные данные, аналогичные следующему тексту:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
```

```
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be
printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string expression.
```

В процессе трассировки результата можно увидеть, как компилятор добавляет код для вызова обработчика и создания строки.

- Компилятор добавляет вызов для создания обработчика, передав общую длину литерального текста в строке формата и количество заполнителей.
- Компилятор добавляет вызовы функций `AppendLiteral` и `AppendFormatted` для каждого раздела строки литерала и для каждого заполнителя.
- Компилятор вызывает метод `LogMessage` с помощью `CoreInterpolatedStringHandler` в качестве аргумента.

Наконец, обратите внимание, что последнее предупреждение не вызывает интерполированный обработчик строк. Аргумент представляет собой `string`, поэтому вызов вызывает другую перегрузку строковым параметром.

> **ⓘ Внимание**
>
> Используйте `ref struct` для интерполированных обработчиков строк, только если это необходимо. `ref struct` типы имеют ограничения, поскольку они должны сохраняться в стеке. Например, они не работают, если интерполированное строковое отверстие содержит `await` выражение, так как компилятор должен хранить обработчик в реализации, созданной `IAsyncStateMachine` компилятором.

Расширьте возможности обработчика

Предыдущая версия интерполированного обработчика строк реализует шаблон. Чтобы избежать обработки каждого выражения замещающего элемента, вам нужна дополнительная информация в обработчике. В этом разделе вы улучшаете обработчик, чтобы он выполнял меньше работы, если созданная строка не записывается в журнал. Вы используете System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute, чтобы указать сопоставление параметров с общедоступным API и параметрами

конструктора обработчика. Это сопоставление предоставляет обработчику информацию, необходимую для определения необходимости вычисления интерполированной строки.

Начните с внесения изменений в обработчик. Сначала добавьте поле для отслеживания, включен ли обработчик. Добавьте два параметра в конструктор: один, чтобы указать уровень журнала для этого сообщения, а другой — ссылку на объект журнала:

C#

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger
logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

Затем используйте поле, чтобы обработчик добавлял только литералы или отформатированные объекты при использовании последней строки:

C#

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type {typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

Затем обновите `LogMessage` объявление, чтобы компилятор передавал дополнительные параметры конструктору обработчика. Обработайте этот шаг с помощью аргумента System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute обработчика:

C#

```
public void LogMessage(LogLevel level, [InterpolatedStringHandlerArgument("",
"level")] LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.ToString());
}
```

Этот атрибут задает список аргументов для `LogMessage`, которые сопоставлены с параметрами, следующими после обязательных параметров `literalLength` и `formattedCount`. Пустая строка (`""`), указывает приемник. Компилятор подставляет значение объекта `Logger`, который представлен как `this`, для следующего аргумента конструктора обработчика. Компилятор заменяет значение `level` для следующего аргумента. Вы можете указать любое количество аргументов для любого обработчика, который вы пишете. Добавляемые аргументы являются строковыми аргументами.

> ⓘ **Примечание.**
>
> InterpolatedStringHandlerArgumentAttribute Если список аргументов конструктора пуст, поведение совпадает с тем, что атрибут был опущен полностью.

Эту версию можно запустить с помощью того же тестового кода. На этот раз вы увидите следующие результаты:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be
printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string expression.
```

Вы видите, что вызываются методы `AppendLiteral` и `AppendFormat`, но они не выполняют никакой работы. Обработчик определил, что окончательная строка не нужна, поэтому обработчик не создает его. Есть еще несколько улучшений, чтобы сделать.

Сначала можно добавить перегрузку `AppendFormatted`, которая ограничивает аргумент типом, реализующим System.IFormattable. Эта перегрузка позволяет пользователям добавлять форматные строки в заполнители. При внесении этого изменения также измените тип возврата других `AppendFormatted` и `AppendLiteral` методов на `void` `bool`. Если любой из этих методов имеет разные типы возвращаемых значений, возникает ошибка компиляции. Это изменение позволяет выполнять *короткое замыкание*. Методы возвращают `false`, чтобы указать, что обработка интерполированного строкового выражения должна быть остановлена. Возврат значения `true` указывает, что процесс должен продолжаться. В этом примере вы используете его для остановки обработки, когда результирующая строка не нужна. Короткое замыкание поддерживает более детализированные действия. Вы можете остановить обработку выражения после достижения определенной длины, чтобы поддерживать буферы фиксированной длины. Или какое-то условие может указать, что остальные элементы не нужны.

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t} with
format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
}

public void AppendFormatted<T>(T t, int alignment, string format) where T :
IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t} with
alignment {alignment} and format {{{format}}} is of type {typeof(T)},");
    var formatString =$"{alignment}:{format}";
    builder.Append(string.Format($"{{0,{formatString}}}", t));
    Console.WriteLine($"\tAppended the formatted object");
}
```

С помощью этого дополнения можно указать строки формата в интерполированном строковом выражении:

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The time
doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This is an
error. It will be printed.");
```

```
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This won't
be printed.");
```

`:t` в первом сообщении указывает "короткий формат времени" для текущего времени. В предыдущем примере показана одна из перегрузок метода `AppendFormatted`, который можно создать для обработчика. Не нужно указывать универсальный аргумент для отформатированного объекта. Возможно, у вас есть более эффективные способы преобразования типов, создаваемых в строку. Вы можете создавать перегрузки функций `AppendFormatted`, которые принимают эти типы вместо универсального аргумента. Компилятор выбирает лучшую перегрузку. Среда выполнения использует этот метод для преобразования System.Span<T> в строковые выходные данные. Можно добавить целочисленный параметр, чтобы указать выравнивание выходных данных, с IFormattableили без IFormattable. System.Runtime.CompilerServices.DefaultInterpolatedStringHandler, который поставляется в комплекте с .NET 6, содержит девять перегрузок AppendFormatted для различных видов использования. Его можно использовать в качестве ссылки при создании обработчика для ваших целей.

Запустите пример, и вы увидите, что для сообщения `Trace` вызывается только первая `AppendLiteral`:

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29 PM with
format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string expression.
```

Вы можете внести одно окончательное обновление в конструктор обработчика, который повышает эффективность. Обработчик может добавить окончательный параметр `out`

`bool`. Установка этого параметра на `false` указывает, что обработчик не должен вызываться вообще для обработки интерполированного строкового выражения:

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger
logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

Это изменение означает, что вы можете удалить поле `enabled`. Затем можно изменить тип возвращаемого значения `AppendLiteral` и `AppendFormatted` на `void`. Теперь при запуске примера вы увидите следующие выходные данные:

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10 PM with
format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string expression.
```

Единственным выходным результатом, когда был указан `LogLevel.Trace`, является выходные данные конструктора. Обработчик указал, что он не включен, поэтому ни один из `Append` методов не вызывается.

В этом примере показана важная точка интерполированных обработчиков строк, особенно при использовании библиотек ведения журнала. Любые побочные эффекты в заполнителях могут не возникать. Добавьте следующий код в основную программу и просмотрите это поведение в действии:

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index {index++}");
    numberOfIncrements++;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:
{numberOfIncrements}");
```

Вы можете видеть, что переменная `index` увеличивается в каждой итерации цикла. Поскольку заполнители вычисляются только для уровней `Critical`, `Error` и `Warning`, а не для `Information` и `Trace`, итоговое значение `index` не соответствует ожиданиям.

PowerShell

```
Critical
Critical: Increment index 0
Error
Error: Increment index 1
Warning
Warning: Increment index 2
Information
Trace
Value of index 3, value of numberOfIncrements: 5
```

Интерполированные обработчики строк обеспечивают больший контроль над преобразованием интерполированного строкового выражения в строку. Команда среды выполнения .NET использовала эту функцию для повышения производительности в нескольких областях. Вы можете использовать ту же возможность в собственных библиотеках. Дополнительные сведения см. в System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. Это обеспечивает более полную реализацию, чем вы создали здесь. Вы видите много дополнительных перегрузок, которые возможны для методов `Append`.

Last updated on 05.02.2026

Пакет SDK для платформы компилятора .NET

17.06.2025

Компиляторы создают подробную модель кода приложения, так как проверяют синтаксис и семантику этого кода. Они используют эту модель для создания исполняемых выходных данных из исходного кода. Пакет SDK для платформы компилятора .NET предоставляет доступ к этой модели. Все чаще мы опираемся на интегрированные функции среды разработки (IDE), такие как IntelliSense, рефакторинг, интеллектуальное переименование, "Поиск всех ссылок" и "Перейти к определению", чтобы повысить производительность. Мы опираемся на средства анализа кода для улучшения качества кода и генераторов кода для помощи в создании приложений. По мере того как эти средства становятся более умными, им нужен доступ к все большему объему модели, который создают только компиляторы при обработке кода приложения. Это основная миссия API Roslyn: открытие закрытых систем и предоставление инструментам и конечным пользователям возможности делиться богатым объемом информации, имеющейся у компиляторов о нашем коде. Вместо того чтобы быть непрозрачными переводчиками исходного кода в объектный код, с помощью Roslyn компиляторы становятся платформами — интерфейсы API, которые можно использовать для задач, связанных с кодом, в инструментах и приложениях.

Основные понятия пакета SDK для платформы компилятора .NET

Пакет SDK платформы компилятора .NET значительно снижает барьер для создания ориентированных на код средств и приложений. Он создает множество возможностей для инноваций в таких областях, как метапрограммирование, создание кода и преобразование, интерактивное использование языков C# и Visual Basic, а также внедрение C# и Visual Basic на языках, относящихся к домену.

Пакет SDK для платформы компилятора .NET позволяет создавать ***анализаторы*** и ***исправления кода*** , которые находят и исправляют ошибки программирования. ***Анализаторы*** понимают синтаксис (структуру кода) и семантику для обнаружения рекомендаций, которые следует исправить. ***Исправления кода*** предоставляют одно или несколько предлагаемых исправлений для решения ошибок кодирования, обнаруженных анализаторами или диагностикой компилятора. Как правило, анализатор и связанные исправления кода упаковываются в один проект.

Анализаторы и исправления кода используют статический анализ для понимания кода. Они не запускают код или предоставляют другие преимущества тестирования. Однако они могут указывать на практики, которые часто приводят к ошибкам, неуправляемому коду или стандартному нарушению руководства.

Помимо анализаторов и исправлений кода, пакет SDK платформы компилятора .NET также позволяет создавать ***рефакторинг кода***. Он также предоставляет один набор API, который позволяет изучать и понимать базу кода C# или Visual Basic. Так как вы можете использовать эту единую базу кода, вы можете легко создавать анализаторы и исправления кода, используя api синтаксического и семантического анализа, предоставляемые пакетом SDK платформы компилятора .NET. Освобожденная от большой задачи репликации анализа, выполняемого компилятором, можно сосредоточиться на более ориентированной задаче поиска и устранения распространенных ошибок программирования для проекта или библиотеки.

Меньшее преимущество заключается в том, что анализаторы и исправления кода меньше и используют гораздо меньше памяти при загрузке в Visual Studio, чем если вы написали собственную базу кода, чтобы понять код в проекте. Используя те же классы, которые используются компилятором и Visual Studio, можно создать собственные статические средства анализа. Это означает, что ваша команда может использовать анализаторы и исправления кода без заметного влияния на производительность интегрированной среды разработки.

Существует три основных сценария написания анализаторов и исправлений кода:

1. *Применение стандартов кодирования команды*
2. *Дайте рекомендации по пакетам библиотеки*
3. *Предоставление общих рекомендаций*

Применение стандартов кодирования команды

Многие команды имеют стандарты кодирования, которые применяются через проверки кода с другими участниками команды. Анализаторы и исправления кода могут сделать этот процесс гораздо более эффективным. Проверки кода происходят, когда разработчик делится своей работой с другими пользователями в команде. Разработчик будет инвестировать все время, необходимое для завершения новой функции, прежде чем получать комментарии. Недели могут пойти, пока разработчик усиливает привычки, которые не соответствуют практики команды.

Анализаторы выполняются, пока разработчик пишет код. Разработчик получает немедленную обратную связь, которая рекомендует немедленно следовать рекомендациям. Разработчик создает привычки писать соответствующий код сразу после

начала прототипирования. Когда функция готова к просмотру людьми, все стандартные рекомендации были строго соблюдены.

Команды могут создавать анализаторы кода и исправления, которые выявляют наиболее распространенные практики, нарушающие стандарты кодирования команды. Их можно установить на каждом компьютере разработчика, чтобы обеспечить соблюдение стандартов.

> **Совет**
>
> Прежде чем создавать собственный анализатор, ознакомьтесь с уже встроенными анализаторами. Дополнительные сведения см. в разделе **"Правила стиля кода**".

Предоставление руководства по библиотечным пакетам

Существует множество библиотек, доступных для разработчиков .NET в NuGet. Некоторые из них приходят от Корпорации Майкрософт, некоторые из сторонних компаний, а также другие из членов сообщества и добровольцев. Эти библиотеки получают больше внедрения и более высокие отзывы, когда разработчики могут успешно использовать эти библиотеки.

Помимо предоставления документации, вы можете предоставлять анализаторы и исправления кода, которые находят и исправляют распространенные неправильные способы использования библиотеки. Эти немедленные исправления помогут разработчикам быстрее добиться успеха.

Анализаторы и исправления кода можно упаковыть с помощью библиотеки в NuGet. В этом сценарии каждый разработчик, который устанавливает пакет NuGet, также установит пакет анализатора. Все разработчики, использующие вашу библиотеку, немедленно получат рекомендации от вашей команды в виде немедленной обратной связи по ошибкам и предлагаемым исправлениям.

Предоставление общих рекомендаций

Сообщество разработчиков .NET обнаружило, используя опыт, шаблоны, которые хорошо работают и шаблоны, которые лучше всего избежать. Несколько членов сообщества создали анализаторы, которые применяют эти рекомендуемые шаблоны. По мере того как мы узнаем больше, всегда есть место для новых идей.

Эти анализаторы можно отправить в Visual Studio Marketplace и скачать разработчиками с помощью Visual Studio. Новички, изучающие язык и платформу, быстро изучают принятые методики и становятся продуктивными на начальных этапах освоения .NET. По мере того как они становятся более широко используемыми, сообщество принимает эти методики.

Генераторы источников

Генераторы источников стремятся включить *метапрограммирование времени компиляции*, то есть код, который можно создать во время компиляции и добавить в компиляцию. Генераторы источников могут считывать содержимое компиляции перед выполнением, а также получать доступ к *любым дополнительным файлам*. Эта возможность позволяет им интроспектировать как пользовательские файлы кода C#, так и файлы генератора. Вы можете узнать, как создавать инкрементные генераторы исходного кода с помощью руководства по генераторам исходного кода.

Дальнейшие шаги

Пакет SDK платформы компилятора .NET включает последние языковые объектные модели для создания кода, анализа и рефакторинга. В этом разделе представлен обзор пакета SDK для платформы компилятора .NET. Дополнительные сведения см. в руководствах по быстрому старту, примерах и учебных пособиях.

Дополнительные сведения о концепциях пакета SDK для платформы компилятора .NET см. в следующих пяти разделах:

- Изучение кода с помощью визуализатора синтаксиса
- Общие сведения о модели API компилятора
- Работа с синтаксисом
- Работа с семантикой
- Работа с рабочей средой

Чтобы приступить к работе, необходимо установить **пакет SDK платформы компилятора .NET**:

Инструкции по установке — Visual Studio Installer

Существует два разных способа найти **пакет SDK платформы компилятора .NET** в **установщике Visual Studio**:

Установка с помощью установщика Visual Studio — представление рабочих нагрузок

Пакет SDK платформы компилятора .NET не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как необязательный компонент.

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Проверьте рабочую нагрузку **разработки расширений Visual Studio** .
4. Откройте узел **разработки расширений Visual Studio** в дереве сводки.
5. Установите флажок **.NET Compiler Platform SDK**. Он будет находиться в последнем разделе необязательных компонентов.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Откройте узел **отдельных компонентов** в дереве сводки.
2. Установите флажок для **редактора DGML**

Установка с помощью установщика Visual Studio — вкладка "Отдельные компоненты"

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Выберите вкладку **"Отдельные компоненты"**
4. Установите флажок **.NET Compiler Platform SDK**. Его можно найти в верхней части раздела **"Компиляторы", "Средства сборки" и "Среды выполнения** ".

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Установите флажок **для редактора DGML**. Его можно найти в разделе **"Инструменты кода** ".

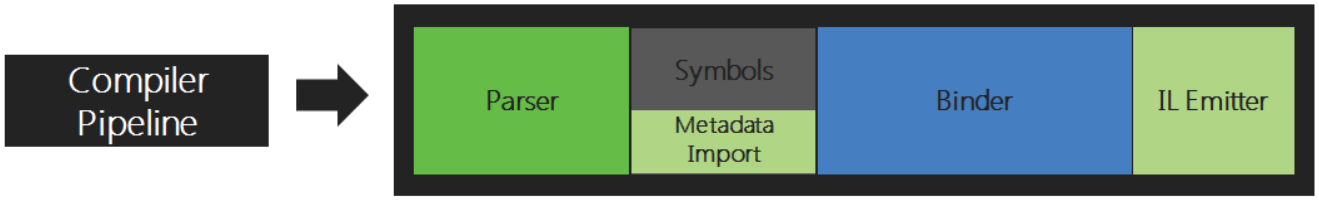
Общие сведения о модели пакета SDK платформы компилятора .NET

17.06.2025

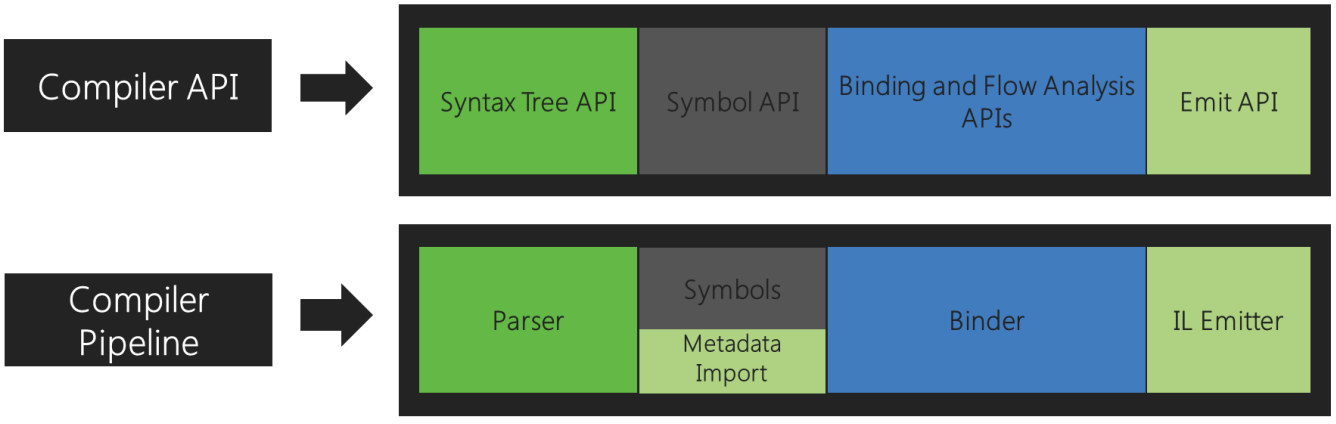
Компиляторы обрабатывают код, который вы пишете по структурированным правилам, которые часто отличаются от того, как люди осмысливают и читают код. Базовое понимание модели, используемой компиляторами, важно для понимания API, используемых при создании средств на основе Roslyn.

Функциональные области конвейера компилятора

Пакет SDK платформы компилятора .NET предоставляет вам доступ к анализу кода компиляторов C# и Visual Basic путем предоставления слоя API, который имитирует традиционный поток компилятора.

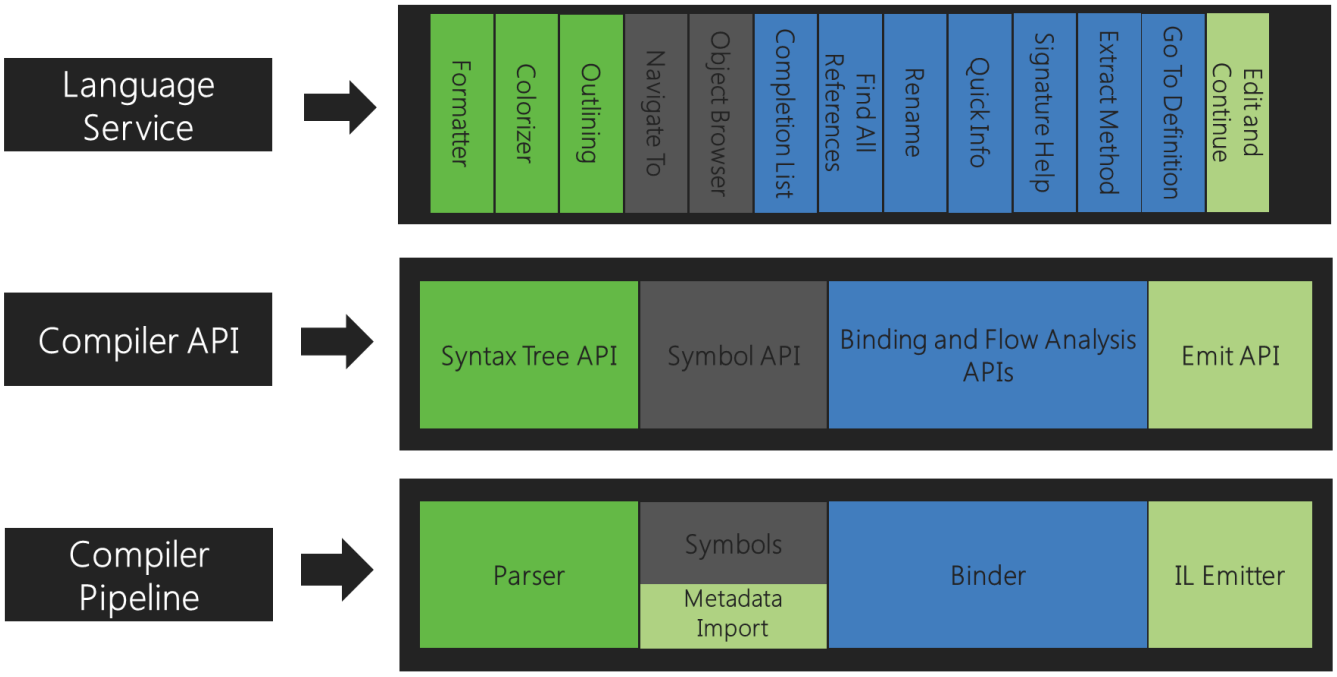


Каждый этап этого конвейера является отдельным компонентом. Во-первых, на этапе анализа исходный текст разбивается на токены и синтаксис, который соответствует грамматике языка. Во-вторых, этап объявления анализирует исходные и импортированные метаданные для формирования именованных символов. Затем этап привязки сопоставляет идентификаторы в коде с символами. Наконец, этап выдачи выдает сборку со всеми сведениями, созданными компилятором.



В соответствии с каждым из этих этапов пакет SDK платформы компилятора .NET предоставляет объектную модель, которая позволяет получить доступ к данным на этом

этапе. Этап синтаксического анализа предоставляет дерево синтаксиса, этап объявления предоставляет иерархическую таблицу символов, этап привязки предоставляет результат семантического анализа компилятора, а этап выдачи — ЭТО API, который создает коды байтов IL.



Каждый компилятор объединяет эти компоненты как единое комплексное целое.

Эти API являются теми же, которые используются Visual Studio. Например, функции структурирования кода и форматирования используют синтаксические деревья, **браузер объектов** и функции навигации, использующие таблицу символов, рефакторинг и переход **к определению** , используют семантику модели, а также **"Изменить и продолжить** " используют все эти элементы, включая API отправки.

Уровни API

Пакет SDK компилятора .NET состоит из нескольких уровней API: API компилятора, диагностические API, API скриптов и API рабочих областей.

API компилятора

Слой компилятора содержит объектные модели, соответствующие сведениям, предоставляемым на каждом этапе конвейера компилятора, как синтаксическим, так и семантичным. Слой компилятора также содержит неизменяемый моментальный снимок одного вызова компилятора, включая ссылки на сборки, параметры компилятора и файлы исходного кода. Существует два разных API, представляющих язык C# и язык Visual Basic. Два API похожи в форме, но адаптированы для высокой точности для каждого отдельного языка. Этот слой не имеет зависимостей от компонентов Visual Studio.

API диагностики

В рамках анализа компилятор может создать набор диагностики, охватывающий все от синтаксиса, семантики и определенных ошибок назначения до различных предупреждений и информационной диагностики. Уровень API компилятора предоставляет диагностику через расширяемый API, который позволяет пользовательским анализаторам подключаться к процессу компиляции. Он позволяет создавать пользовательские диагностические данные, создаваемые такими инструментами, как StyleCop, наряду с диагностикой, определяемой компилятором. Создание диагностики таким образом имеет преимущество естественной интеграции с такими инструментами, как MSBuild и Visual Studio, которые зависят от диагностики для возможностей, таких как остановка сборки на основе политики, отображение живых ошибок в редакторе и предложение исправлений кода.

API-интерфейсы сценариев

API хостинга и скриптинга создаются на базе слоя компилятора. Сценарные API можно использовать для выполнения фрагментов кода и накопления контекста выполнения. Интерактивный rePL C# (цикл чтенияEvaluate-Print) использует эти API. REPL позволяет использовать C# в качестве языка сценариев, выполняя код в интерактивном режиме при написании.

API рабочих областей

Уровень рабочих областей содержит API рабочей области, который является отправной точкой для анализа кода и рефакторинга во всех решениях. Он помогает упорядочить все сведения о проектах в решении в одной объектной модели, предоставляя прямой доступ к объектным моделям уровня компилятора, не требуя анализа файлов, настройки параметров или управления зависимостями проекта в проект.

Кроме того, слой рабочих областей предоставляет набор API, используемых при реализации инструментов анализа и рефакторинга кода, которые функционируют в хост-окружении, например, в интегрированной среде разработки (IDE) Visual Studio. Примерами являются API поиска всех ссылок, форматирования и создания кода.

Этот слой не имеет зависимостей от компонентов Visual Studio.

Работа с синтаксисом

17.06.2025

Дерево синтаксиса — это фундаментальная неизменяемая структура данных, представленная API компилятора. Эти деревья представляют лексическую и синтаксическую структуру исходного кода. Они служат двумя важными целями:

- Чтобы разрешить таким средствам, как интегрированная среда разработки, надстройки, средства анализа кода и рефакторинг, просматривать и обрабатывать синтаксическую структуру исходного кода в проекте пользователя.
- Чтобы включить такие средства, как рефакторинг и интегрированная среда разработки, для создания, изменения и перестановки исходного кода естественным образом без использования прямого редактирования текста. Создавая и управляя деревьями, средства могут легко создавать и изменять исходный код.

Деревья синтаксиса

Деревья синтаксиса — это основная структура, используемая для компиляции, анализа кода, привязки, рефакторинга, функций интегрированной среды разработки и создания кода. Ни одна часть исходного кода не понимается без первого определения и классификации на один из многих известных структурных элементов языка.

> ⓘ **Примечание**
>
> RoslynQuoter — это средство с открытым исходным кодом, которое показывает вызовы API фабрики синтаксиса, используемые для создания дерева синтаксиса программы. Чтобы попробовать его в прямом эфире, см. http://roslynquoter.azurewebsites.net.

Деревья синтаксиса имеют три ключевых атрибута:

- Они содержат всю исходную информацию в полной точности. Полная точность означает, что дерево синтаксиса содержит каждую часть информации, найденную в исходном тексте, каждую грамматическую конструкцию, каждый лексический маркер и все остальное между ними, включая пробелы, комментарии и директивы препроцессора. Например, каждый литерал, упомянутый в источнике, представлен точно так же, как он был введен. Деревья синтаксиса также фиксируют ошибки в исходном коде, если программа неполная или некорректная, представляя пропущенные маркеры.

- Они могут создать точный текст, из которого они были проанализированы. Из любого синтаксического узла можно получить текстовое представление поддерева, корнем которого является этот узел. Эта возможность означает, что деревья синтаксиса можно использовать как способ создания и редактирования исходного текста. Создавая дерево, вы, тем самым, создаете эквивалентный текст, а превращая изменения существующего дерева в новое дерево, вы, таким образом, эффективно редактируете текст.
- Они неизменяемы и потокобезопасны. После получения дерева это моментальный снимок текущего состояния кода и никогда не изменяется. Это позволяет нескольким пользователям взаимодействовать с одинаковым деревом синтаксиса одновременно в разных потоках без блокировки или дублирования. Так как деревья неизменяемы и их нельзя изменять непосредственно, фабричные методы помогают создавать и модифицировать синтаксические деревья, создавая дополнительные моментальные снимки. Деревья эффективны в том, как они повторно используют базовые узлы, поэтому новая версия может быть перестроена быстро и с небольшим объемом дополнительной памяти.

Дерево синтаксиса — это буквально структура данных дерева, в которой неконтерминальные структурные элементы являются родительскими другими элементами. Каждое дерево синтаксиса состоит из узлов, маркеров и тривии.

Узлы синтаксиса

Узлы синтаксиса являются одним из основных элементов деревьев синтаксиса. Эти узлы представляют собой синтаксические конструкции, такие как объявления, операторы, предложения и выражения. Каждая категория синтаксического узла представлена отдельным классом, производным от Microsoft.CodeAnalysis.SyntaxNode. Набор классов узлов не расширяем.

Все узлы синтаксиса — это нетерминальные узлы в дереве синтаксиса, что означает, что они всегда имеют другие узлы и токены в качестве дочерних. В качестве дочернего элемента другого узла каждый узел имеет родительский узел, к которому можно получить доступ через SyntaxNode.Parent свойство. Так как узлы и деревья неизменяемы, родительский элемент узла никогда не изменяется. Корень дерева имеет нулевого родителя.

Каждый узел имеет SyntaxNode.ChildNodes() метод, который возвращает список дочерних узлов в последовательном порядке на основе их положения в исходном тексте. Этот список не содержит токены. Каждый узел также имеет методы для проверки потомков, таких как DescendantNodes, DescendantTokens, или DescendantTrivia - которые

представляют список всех узлов, токенов или служебной информации, которые существуют в поддереве, корнем которого является этот узел.

Кроме того, каждый подкласс узла синтаксиса открывает доступ ко всем тем же дочерним элементам через строго типизированные свойства. Например, BinaryExpressionSyntax класс узла имеет три дополнительных свойства, относящиеся к двоичным операторам: Left, OperatorTokenи Right. Тип Left и Right есть ExpressionSyntax, а тип OperatorToken — SyntaxToken.

Некоторые узлы синтаксиса имеют необязательные дочерние узлы. Например, у IfStatementSyntax есть необязательный ElseClauseSyntax. Если дочерний элемент отсутствует, свойство возвращает значение NULL.

Маркеры синтаксиса

Маркеры синтаксиса — это терминалы грамматики языка, представляющие наименьшие синтаксические фрагменты кода. Они никогда не являются родителями других узлов или маркеров. Маркеры синтаксиса состоят из ключевых слов, идентификаторов, литералы и препинания.

В целях эффективности тип SyntaxToken является типом значения CLR. Таким образом, в отличие от узлов синтаксиса, существует только одна структура для всех видов маркеров с сочетанием свойств, которые имеют значение в зависимости от типа маркера, представленного.

Например, целочисленный литеральный токен представляет числовое значение. Помимо необработанного исходного текста, охватываемого токенами, литеральный токен имеет свойство Value, которое указывает точное декодированное целочисленное значение. Это свойство указано как Object, поскольку оно может быть одним из многих примитивных типов.

Свойство ValueText предоставляет ту же информацию, что и свойство Value, однако это свойство всегда имеет тип String. Идентификатор в исходном тексте C# может содержать символы экранирования Юникода, но синтаксис самой экранирующей последовательности не считается частью имени идентификатора. Поэтому, хотя необработанный текст, охватываемый токеном, включает управляющую последовательность, свойство ValueText этого не делает. Вместо этого он включает символы Юникода, определяемые escape-файлом. Например, если исходный текст содержит идентификатор, записанный как `\u03C0`, то это свойство маркера ValueText вернет `π`.

Тривиа синтаксиса

Тривиа синтаксиса представляют части исходного текста, которые в значительной степени незначительны для нормального понимания кода, таких как пробелы, комментарии и директивы препроцессора. Как и маркеры синтаксиса, тривия — это типы значений. Один Microsoft.CodeAnalysis.SyntaxTrivia тип используется для описания всех видов тривии.

Так как тривия не являются частью обычного синтаксиса языка и могут отображаться в любом месте между двумя маркерами, они не включаются в дерево синтаксиса в качестве дочернего элемента узла. Тем не менее, поскольку они важны при реализации такой функции, как рефакторинг и для обеспечения полной точности с исходным текстом, они существуют в составе дерева синтаксиса.

Вы можете получить доступ к данным о структуре, исследуя коллекции SyntaxToken.LeadingTrivia или SyntaxToken.TrailingTrivia токена. При анализе исходного текста последовательности тривия связаны с маркерами. Как правило, маркер владеет любой тривией после него в той же строке до следующего маркера. Любая тривия после этой строки связана со следующим токеном. Первый токен в исходном файле включает все начальные служебные элементы, а последняя последовательность таких элементов в файле присоединяется к токену конца файла, который иначе не имеет ширины.

В отличие от узлов синтаксиса и маркеров, синтаксическая вспомогательная информация не имеет родительских элементов. Тем не менее, поскольку они являются частью дерева и каждый связан с одним маркером, вы можете использовать свойство SyntaxTrivia.Token для доступа к связанному с ним маркеру.

Диапазоны

Каждый узел, токен или тривиа знает свою позицию в исходном тексте и количество символов, из которых он состоит. Позиция текста представлена как 32-разрядное целое число, индекс которого начинается с нуля и обозначен как `char`. TextSpan Объект — это начальная позиция и количество символов, представленных как целые числа. Если TextSpan имеет нулевую длину, это обозначает расположение между двумя символами.

Каждый узел имеет два TextSpan свойства: Span и FullSpan.

Свойство Span — это диапазон текста от начала первого маркера в поддереве узла до конца последнего маркера. Этот диапазон не включает никаких начальных или конечных тривии.

Это FullSpan текстовый диапазон, включающий обычный диапазон узла, а также диапазон любой начальной или конечной тривии.

Рассмотрим пример.

```
C#
```

```
      if (x > 3)
      {
||        // this is bad
          |throw new Exception("Not right.");|  // better exception?||
      }
```

Узел инструкции внутри блока имеет интервал, указанный одной вертикальной полосой (|). Он содержит символы `throw new Exception("Not right.");`. Полный диапазон обозначается двойными вертикальными полосами (||). Он включает те же символы, что и диапазон, и символы, связанные с ведущим и конечным тривией.

Виды

Каждый узел, токен или тривия имеет SyntaxNode.RawKind свойство типа System.Int32, определяющее точный элемент синтаксиса, представленный. Это значение можно преобразовать в языко-зависимое перечисление. Каждый язык, C# или Visual Basic, имеет одно `SyntaxKind` перечисление (Microsoft.CodeAnalysis.CSharp.SyntaxKind и Microsoft.CodeAnalysis.VisualBasic.SyntaxKind соответственно), которое перечисляет все возможные узлы, маркеры и лишние символы в грамматике. Это преобразование можно сделать автоматически, получив доступ к методам CSharpExtensions.Kind или методам расширения VisualBasicExtensions.Kind.

Свойство RawKind позволяет легко диамбигуировать типы синтаксического узла, которые используют один и тот же класс узлов. Для маркеров и тривии это свойство является единственным способом отличия одного типа элемента от другого.

Например, у одного BinaryExpressionSyntax класса есть Leftи OperatorTokenRight дочерние элементы. Свойство Kind определяет, является ли это узлом синтаксиса типа AddExpression, SubtractExpression или MultiplyExpression.

> **Совет**
>
> Рекомендуется проверить типы с помощью **IsKind** методов расширения (для C#) или **IsKind** (для VB).

Ошибки

Даже если исходный текст содержит синтаксические ошибки, выстраивается полное синтаксическое дерево, которое можно преобразовать обратно в исходный текст. Когда средство синтаксического анализа обнаруживает код, который не соответствует определенному синтаксису языка, он использует один из двух методов для создания дерева синтаксиса:

- Если средство синтаксического анализа ожидает определённый тип токена, но не находит его, оно может вставить отсутствующий токен в дерево синтаксиса в том месте, где ожидался токен. Отсутствующий токен представляет собой ожидаемый токен, но он имеет пустой диапазон, и его свойство возвращает SyntaxNode.IsMissing `true`.

- Средство синтаксического анализа может пропускать токены до тех пор, пока не найдет тот, на котором можно продолжить синтаксический анализ. В этом случае пропущенные маркеры присоединяются как узел trivia с типом SkippedTokensTrivia.

Работа с семантикой

17.06.2025

Деревья синтаксиса представляют лексическую и синтаксическую структуру исходного кода. Хотя эти сведения достаточно, чтобы описать все объявления и логику в источнике, недостаточно информации, чтобы определить, на что ссылается ссылка. Имя может представлять:

- тип
- поле;
- метод
- локальная переменная

Хотя каждое из них уникально различно, для определения того, на какой идентификатор на самом деле ссылаются, часто требуется глубокое понимание правил языка.

Существуют элементы программы, представленные в исходном коде, и программы также могут ссылаться на ранее скомпилированные библиотеки, упакованные в файлы сборок. Хотя для сборок недоступен исходный код, а значит, нет синтаксических узлов или деревьев, программы по-прежнему могут ссылаться на элементы внутри них.

Для этих задач требуется **семантическая модель**.

Помимо синтаксической модели исходного кода, семантическая модель инкапсулирует правила языка, что позволяет легко сопоставить идентификаторы с правильным элементом программы, на который ссылается.

Компиляция

Компиляция — это представление всех необходимых для компиляции программы C# или Visual Basic, которая включает все ссылки на сборки, параметры компилятора и исходные файлы.

Так как все эти сведения хранятся в одном месте, элементы, содержащиеся в исходном коде, можно более подробно описать. Компиляция представляет каждый объявленный тип, член или переменную в виде символа. Компиляция содержит различные методы, которые помогают найти и связать символы, объявленные в исходном коде или импортированные в виде метаданных из сборки.

Как и в деревьях синтаксиса, компиляции неизменяемы. После создания компиляции его нельзя изменить вами или кем-либо другим пользователем, с которым вы можете предоставить общий доступ. Однако можно создать новую компиляцию из

существующей компиляции, указав изменение при этом. Например, можно создать компиляцию, которая одинакова во всех отношениях с существующей компиляцией, за исключением того, что она может содержать дополнительную исходную ссылку или ссылку на сборку.

Символы

Символ представляет отдельный элемент, объявленный исходным кодом или импортированный из сборки в виде метаданных. Каждое пространство имен, тип, метод, свойство, поле, событие, параметр или локальная переменная представлена символом.

Различные методы и свойства Compilation типа помогают находить символы. Например, можно найти символ объявленного типа по общему имени метаданных. Вы также можете получить доступ ко всей таблице символов в виде дерева символов, с корнем в глобальном пространстве имен.

Символы также содержат дополнительные сведения, которые компилятор определяет из источника или метаданных, например других ссылочных символов. Каждый тип символа представлен отдельным интерфейсом, производным от ISymbol, каждый из них имеет свои методы и свойства, подробно описывающие информацию, собранную компилятором. Многие из этих свойств напрямую ссылались на другие символы. Например, свойство IMethodSymbol.ReturnType сообщает вам фактический символ типа, который возвращает метод.

Символы представляют собой общее представление пространств имен, типов и элементов между исходным кодом и метаданными. Например, метод, объявленный в исходном коде, и метод, импортированный из метаданных, представлены одинаковыми IMethodSymbol свойствами.

Символы похожи на систему типов CLR, представленную System.Reflection API, но они более богаты в том, что они моделиируют больше, чем просто типы. Пространства имен, локальные переменные и метки — это все символы. Кроме того, символы являются представлением концепций языка, а не концепций CLR. В этом есть много пересечений, но также существуют значимые различия. Например, метод итератора в C# или Visual Basic является одним символом. Однако при переводе метода итератора в метаданные CLR он становится типом и включает несколько методов.

Семантическая модель

Семантическая модель представляет все семантические сведения для одного исходного файла. Его можно использовать для обнаружения следующих элементов:

- Символы, на которые ссылается определенное расположение в источнике.
- Результирующий тип любого выражения.
- Все диагностические сведения, включающие ошибки и предупреждения.
- Как переменные втекают и выходят из регионов источника.
- Ответы на более спекулятивные вопросы.

Работа с рабочей областью

17.06.2025

Уровень **рабочих областей** является отправной точкой для анализа кода и рефакторинга во всех решениях. В этом слое API рабочей области помогает упорядочивать все сведения о проектах в решении в одной объектной модели, предоставляя прямой доступ к объектным моделям уровня компилятора, таким как исходный текст, деревья синтаксиса, семантические модели и компиляции, не требуя анализа файлов, настройки параметров или управления зависимостями между проектами.

Среды выполнения, такие как интегрированная среда разработки (IDE), предоставляют рабочую область для вас, соответствующую открытому решению. Кроме того, эту модель можно использовать за пределами интегрированной среды разработки, просто загрузив файл решения.

Рабочая область

Рабочая область — это активное представление решения в виде коллекции проектов, каждая из которых содержит коллекцию документов. Рабочая область обычно связана с хост-средой, которая постоянно меняется, когда пользователь вводит текст и управляет свойствами.

Предоставляет Workspace доступ к текущей модели решения. При изменении среды узла рабочая область запускает соответствующие события, а Workspace.CurrentSolution свойство обновляется. Например, когда пользователь вводит в текстовый редактор, соответствующий одному из исходных документов, рабочая область использует событие для сигнала о том, что общая модель решения изменилась и какой документ был изменен. Затем вы можете реагировать на эти изменения, анализируя новую модель для правильности, выделения областей важности или предложения по изменению кода.

Вы также можете создавать автономные рабочие области, которые отключены от среды узла или используются в приложении, в котором нет среды узла.

Решения, проекты и документы

Хотя рабочая область может изменяться каждый раз при нажатии клавиши, вы можете работать с моделью решения в изоляции.

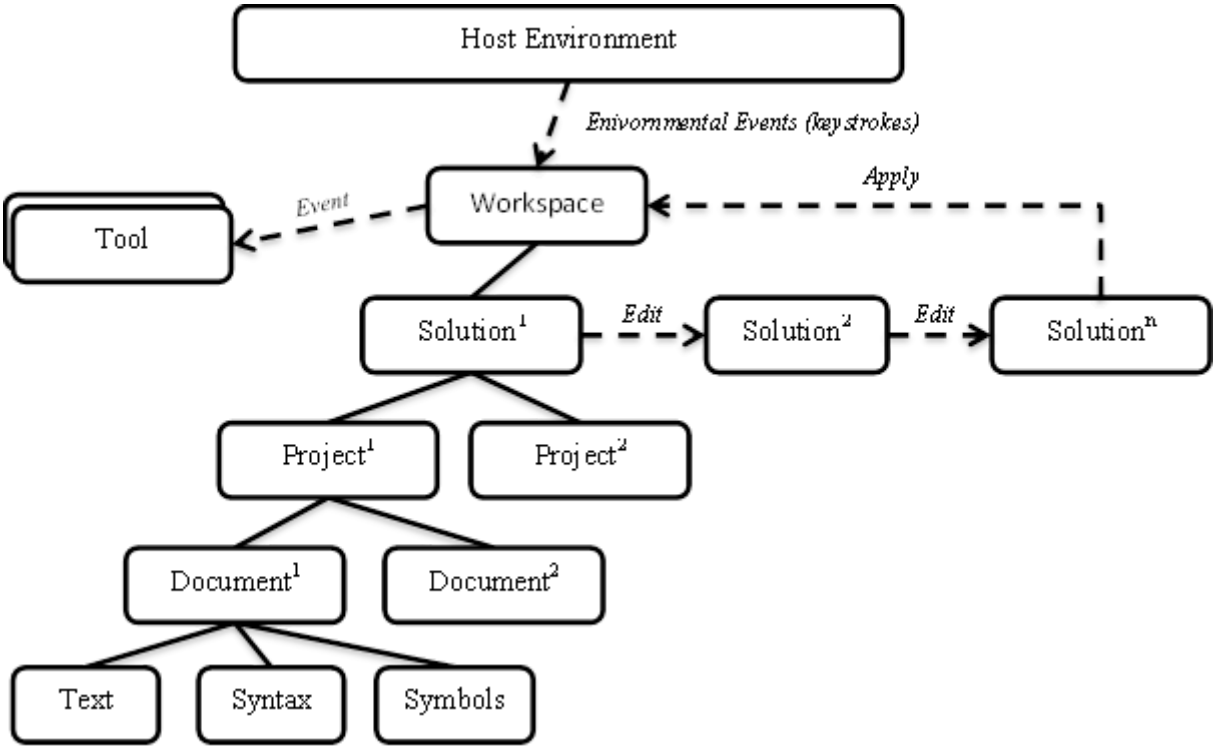
Решение — это неизменяемая модель проектов и документов. Это означает, что модель может быть предоставлена без блокировки или дублирования. После того как вы получите экземпляр решения из свойства Workspace.CurrentSolution, этот экземпляр

никогда не изменится. Однако, как и в деревах синтаксиса и компиляциях, можно изменять решения, создавая новые экземпляры на основе существующих решений и конкретных изменений. Чтобы изменения отразились в рабочей области, необходимо в явной форме применить измененное решение.

Проект является частью общей неизменяемой модели решения. Он представляет все документы исходного кода, параметры синтаксического анализа и компиляции, а также ссылки на сборки и проекты. Из проекта можно получить доступ к соответствующей компиляции, не требуя определения зависимостей проекта или синтаксического анализа исходных файлов.

Документ также является частью общей неизменяемой модели решения. Документ представляет один исходный файл, из которого можно получить доступ к тексту файла, дерева синтаксиса и семантической модели.

На следующей схеме представлено представление того, как рабочая область связана с средой узла, инструментами и способом изменения.



Сводка

Roslyn предоставляет набор API-интерфейсов компилятора и API рабочих областей, которые предоставляют широкие сведения о исходном коде и имеют полную точность с языками C# и Visual Basic. Пакет SDK платформы компилятора .NET значительно снижает барьер для создания средств и приложений, ориентированных на код. Он создает множество возможностей для инноваций в таких областях, как метапрограммирование,

создание кода и преобразование, интерактивное использование языков C# и Visual Basic, а также внедрение C# и Visual Basic на языках, относящихся к домену.

Изучение кода с помощью визуализатора синтаксиса Roslyn в Visual Studio

09.07.2025

В этой статье представлен обзор средства визуализатора синтаксиса, который поставляется в составе пакета SDK для компилятора .NET (Roslyn). Визуализатор синтаксиса — это окно инструментов, которое помогает проверять и изучать деревья синтаксиса. Это важное средство для понимания моделей кода, который требуется проанализировать. Это также помощь в отладке при разработке собственных приложений с помощью пакета SDK для компилятора .NET ("Roslyn"). Откройте это средство при создании первых анализаторов. Визуализатор помогает понять модели, используемые API. Вы также можете использовать такие инструменты, как SharpLab или LINQPad , для проверки кода и понимания деревьев синтаксиса.

Инструкции по установке — Visual Studio Installer

Существует два разных способа найти **пакет SDK платформы компилятора .NET** в **установщике Visual Studio**:

Установка с помощью установщика Visual Studio — представление рабочих нагрузок

Пакет SDK платформы компилятора .NET не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как необязательный компонент.

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Проверьте рабочую нагрузку **разработки расширений Visual Studio** .
4. Откройте узел **разработки расширений Visual Studio** в дереве сводки.
5. Установите флажок **.NET Compiler Platform SDK**. Он будет находиться в последнем разделе необязательных компонентов.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Откройте узел **отдельных компонентов** в дереве сводки.

2. Установите флажок для **редактора DGML**

Установка с помощью установщика Visual Studio — вкладка "Отдельные компоненты"

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Выберите вкладку **"Отдельные компоненты"**
4. Установите флажок **.NET Compiler Platform SDK**. Его можно найти в верхней части раздела **"Компиляторы", "Средства сборки" и "Среды выполнения** ".

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

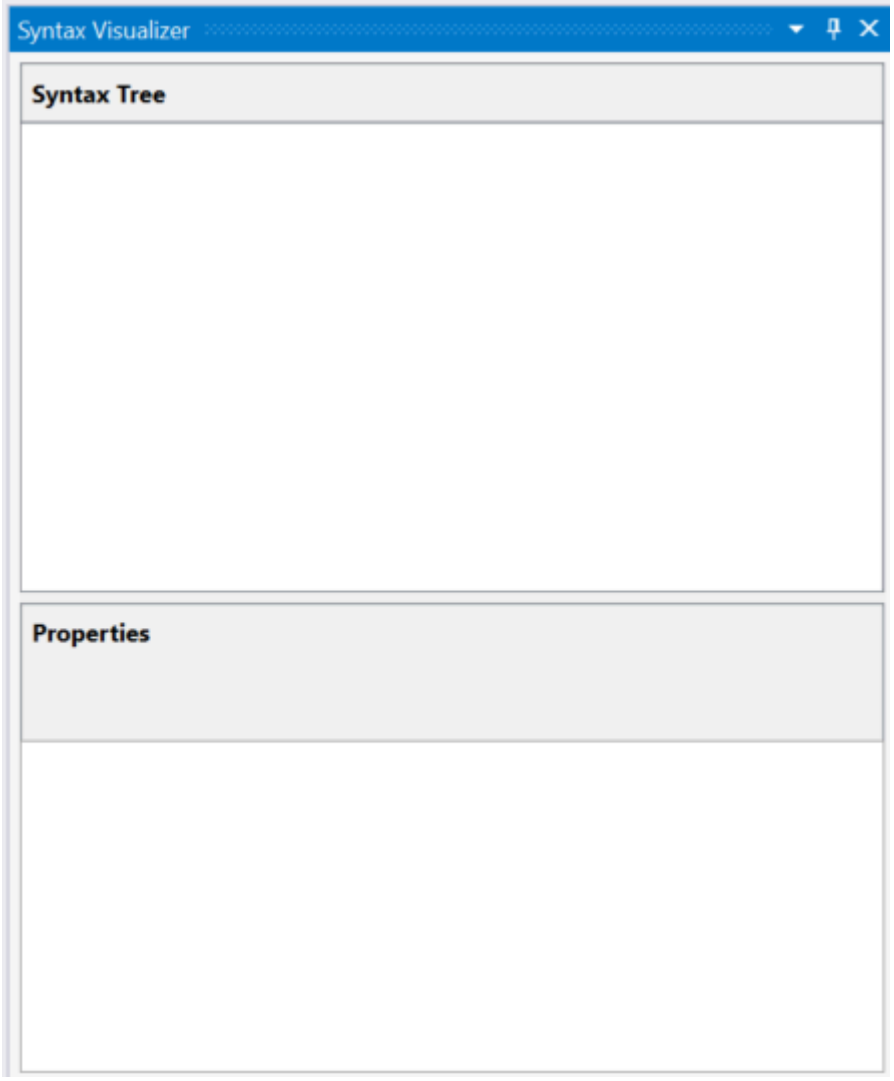
1. Установите флажок **для редактора DGML**. Его можно найти в разделе **"Инструменты кода** ".

Ознакомьтесь с понятиями, используемыми в платформе SDK компилятора .NET, прочитав статью Обзор. В ней приведены общие сведения о деревах синтаксиса, узлах, маркерах и тривии.

Визуализатор синтаксиса

Визуализатор синтаксиса позволяет проверить дерево синтаксиса для файла кода C# или Visual Basic в текущем окне активного редактора в интегрированной среде разработки Visual Studio. Визуализатор можно запустить, щелкнув по **View**>**Other Windows**>**Syntax Visualizer**. Панель **быстрого запуска** также можно использовать в правом верхнем углу. Введите "синтаксис", и команда, чтобы открыть **визуализатор синтаксиса** , должна появиться.

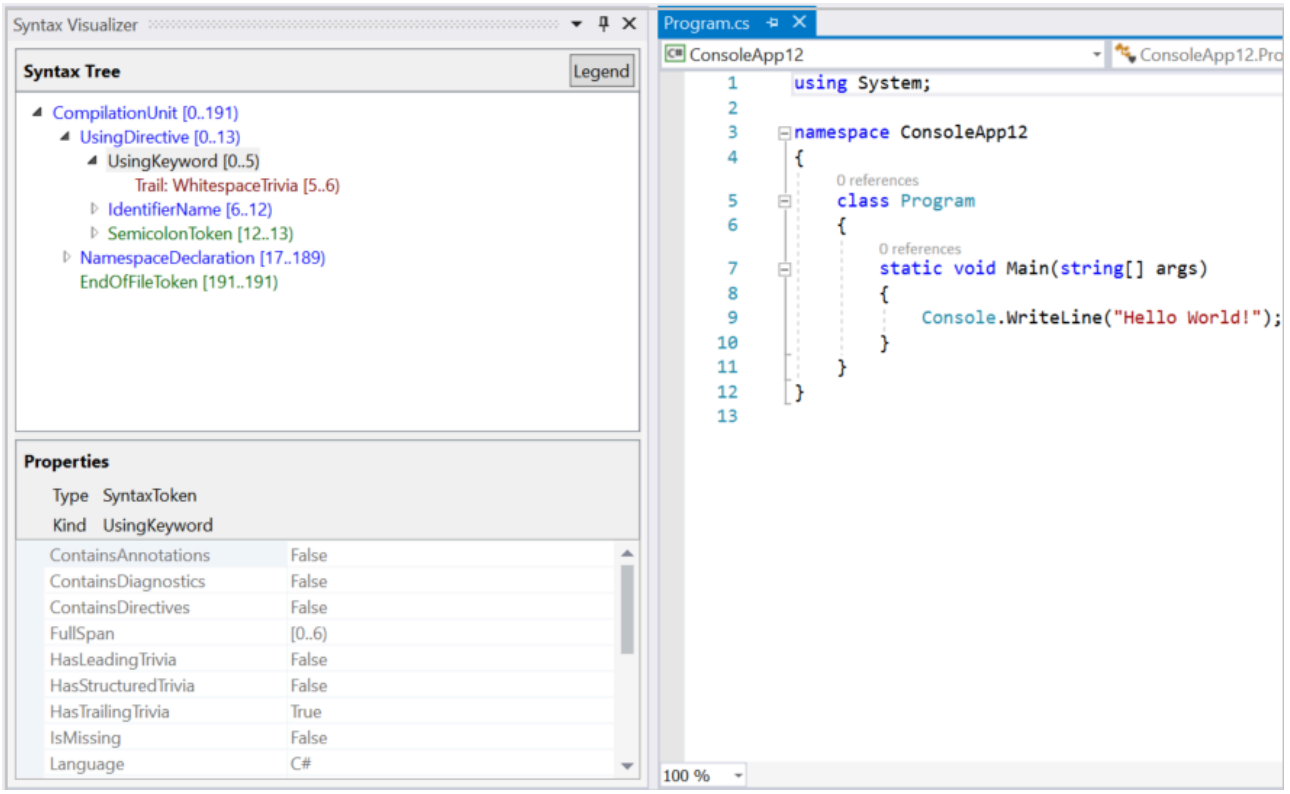
Эта команда открывает визуализатор синтаксиса в виде плавающего окна инструмента. Если у вас нет открытого окна редактора кода, отображается пустой экран, как показано на следующем рисунке.



Закрепите окно инструментов в удобном месте в Visual Studio, например, слева. Визуализатор отображает сведения о текущем файле кода.

Создайте проект с помощью команды "**Файл**>**нового проекта** ". Вы можете создать проект Visual Basic или C#. Когда Visual Studio открывает основной файл кода для этого проекта, визуализатор отображает дерево синтаксиса для него. В этом экземпляре Visual Studio можно открыть любой существующий файл C# или Visual Basic, а визуализатор отображает дерево синтаксиса этого файла. Если в Visual Studio открыто несколько файлов кода, визуализатор отображает дерево синтаксиса для текущего активного файла кода (файл кода с фокусом клавиатуры).)

C#



Как показано на предыдущих изображениях, окно инструментов визуализатора отображает дерево синтаксиса в верхней части и сетку свойств внизу. Сетка свойств отображает свойства элемента, выбранного в настоящее время в дереве, включая *mun* .NET и *mun* (СинтаксисKind) элемента.

Деревья синтаксиса состоят из трех типов элементов — *узлов*, *маркеров* и *тривии*. Дополнительные сведения об этих типах см. в статье " Работа с синтаксисом ". Элементы каждого типа представлены с помощью другого цвета. Нажмите кнопку "Легенда" для обзора используемых цветов.

Каждый элемент в дереве также отображает свой **собственный интервал**. **Диапазон** — это индексы (начальная и конечная позиция) этого узла в текстовом файле. В приведенном выше примере C# выбранный маркер UsingKeyword [0..5) имеет **диапазон** шириной пять символов, [0..5). Нотация "[.)" означает, что начальный индекс является частью диапазона, но конечный индекс не является.

Существует два способа навигации по дереву:

- Разверните или щелкните элементы в дереве. Визуализатор автоматически выбирает текст, соответствующий диапазону этого элемента в редакторе кода.
- Щелкните или выделите текст в редакторе кода. В предыдущем примере Visual Basic при выборе строки, содержащей модуль Module1 в редакторе кода, визуализатор автоматически переходит к соответствующему узлу ModuleStatement в дереве.

Визуализатор выделяет элемент в дереве, диапазон которого лучше всего соответствует диапазону текста, выбранного в редакторе.

Визуализатор обновляет дерево в соответствии с изменениями в активном файле кода. Добавьте вызов `Console.WriteLine()` внутрь `Main()`. При вводе визуализатор обновляет дерево.

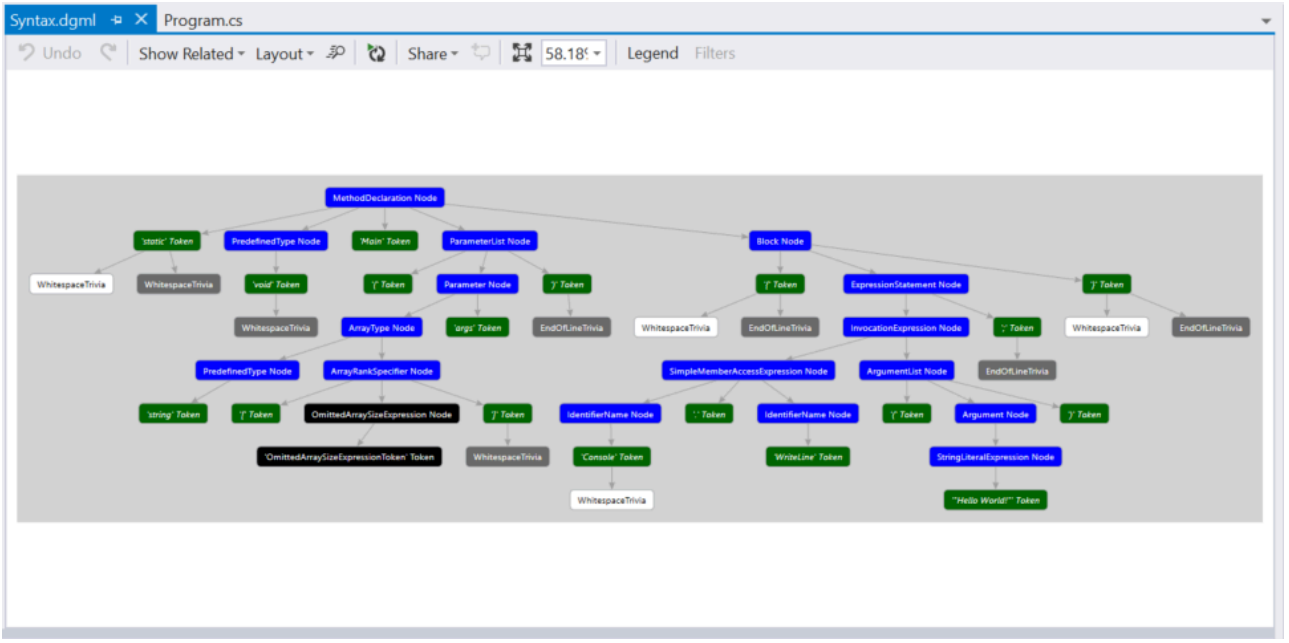
Поставьте ввод на паузу, как только вы введёте `Console.`. Дерево имеет некоторые элементы розового цвета. На этом этапе в написанном коде возникают ошибки (также называемые "Диагностика"). Эти ошибки присоединяются к узлам, маркерам и тривии в дереве синтаксиса. Визуализатор показывает, какие элементы имеют ошибки, прикрепленные к ним, подчеркивая фон в розовом цвете. Ошибки можно проверить на любом элементе розового цвета, наведя указатель мыши на элемент. Визуализатор отображает только синтаксические ошибки (эти ошибки, связанные с синтаксисом типизированного кода); Он не отображает семантические ошибки.

Графы синтаксиса

Щелкните правой кнопкой мыши на любом элементе в дереве и выберите «**Просмотр направленного синтаксического графа**».

C#

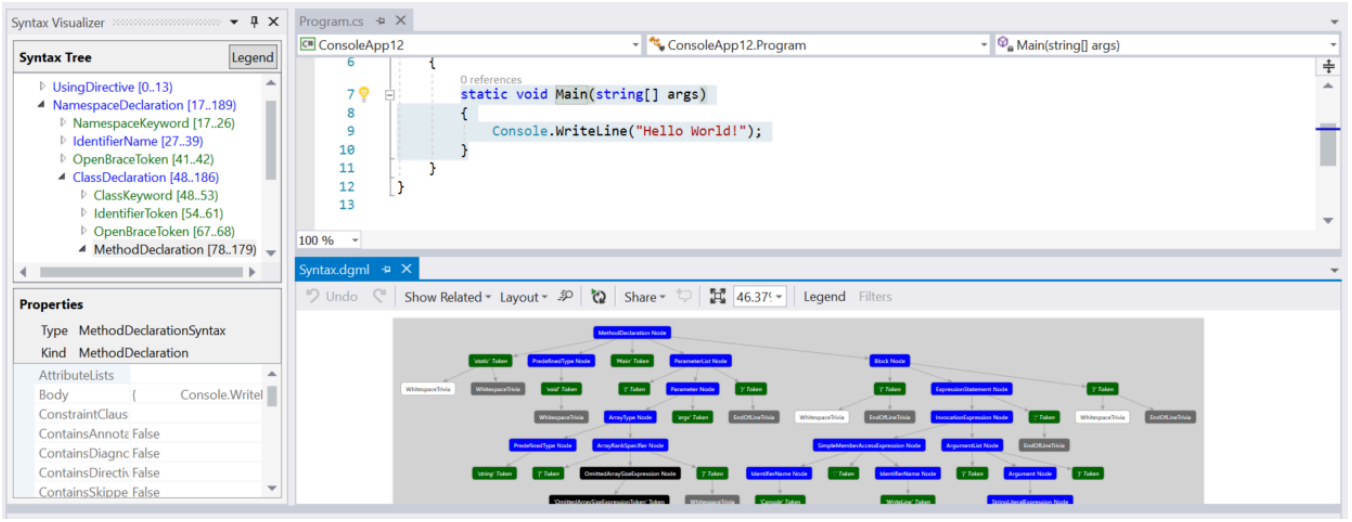
Визуализатор отображает графическое представление поддерева, корнем которого является выбранный элемент. Выполните следующие действия для узла **MethodDeclaration** , соответствующего `Main()` методу в примере C#. Визуализатор отображает граф синтаксиса, который выглядит следующим образом:



Средство просмотра синтаксического графа имеет возможность отображения условных обозначений для его цветовой схемы. Вы также можете навести указатель мыши на отдельные элементы в графе синтаксиса, чтобы просмотреть свойства, соответствующие этому элементу.

Вы можете просматривать графы синтаксиса для разных элементов в дереве многократно, а графы всегда будут отображаться в одном окне в Visual Studio. Это окно можно закрепить в удобном расположении в Visual Studio, чтобы вам не нужно переключаться между вкладками для просмотра нового графа синтаксиса. На нижней части, ниже окон редактора кода, часто удобно.

Ниже приведен макет док-станции для использования с окном средства визуализатора и окном графа синтаксиса:

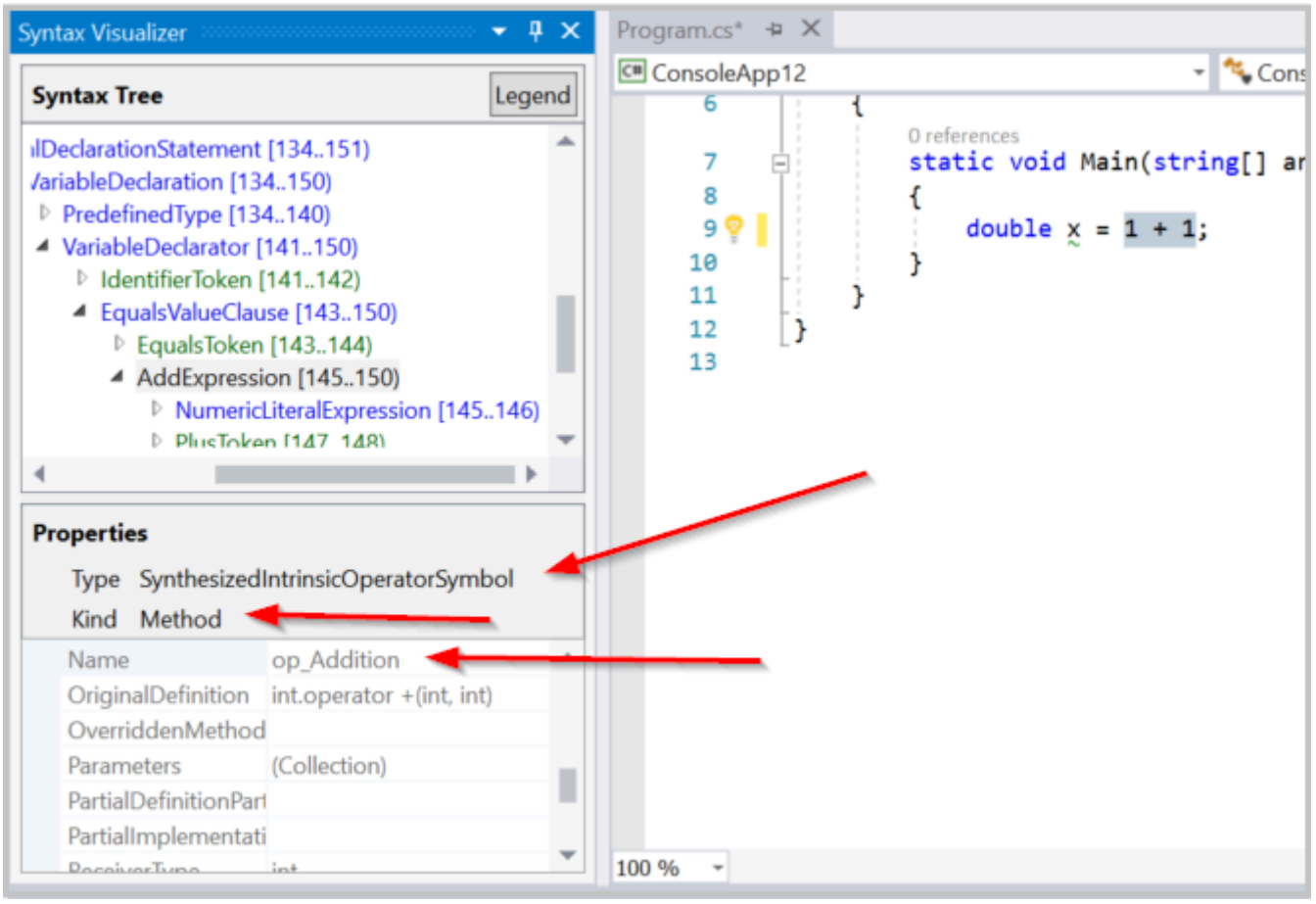


Другим вариантом является размещение окна графа синтаксиса во втором мониторе в настройке двойного монитора.

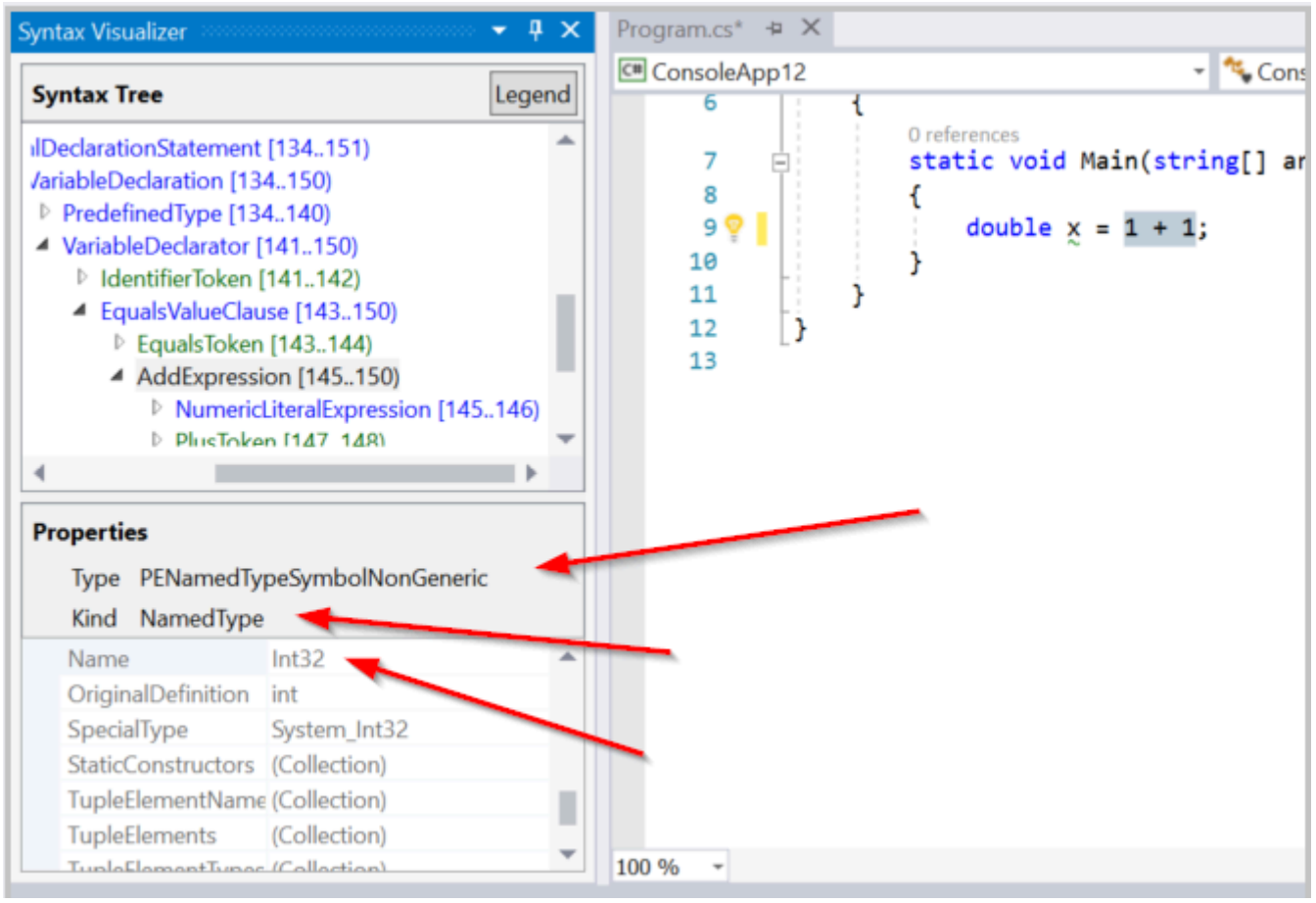
Проверка семантики

Визуализатор синтаксиса позволяет проверять символы и семантические сведения. Введите `double x = 1 + 1;` внутри Main() в примере C#. Затем выберите выражение `1 + 1` в окне редактора кода. Визуализатор выделяет узел **AddExpression** в визуализаторе. Щелкните правой кнопкой мыши на **AddExpression** и выберите **Показать символ (если есть)**. Обратите внимание, что большинство элементов меню имеют условный элемент "если есть". Визуализатор синтаксиса проверяет свойства узла, включая свойства, которые могут не присутствовать для всех узлов.

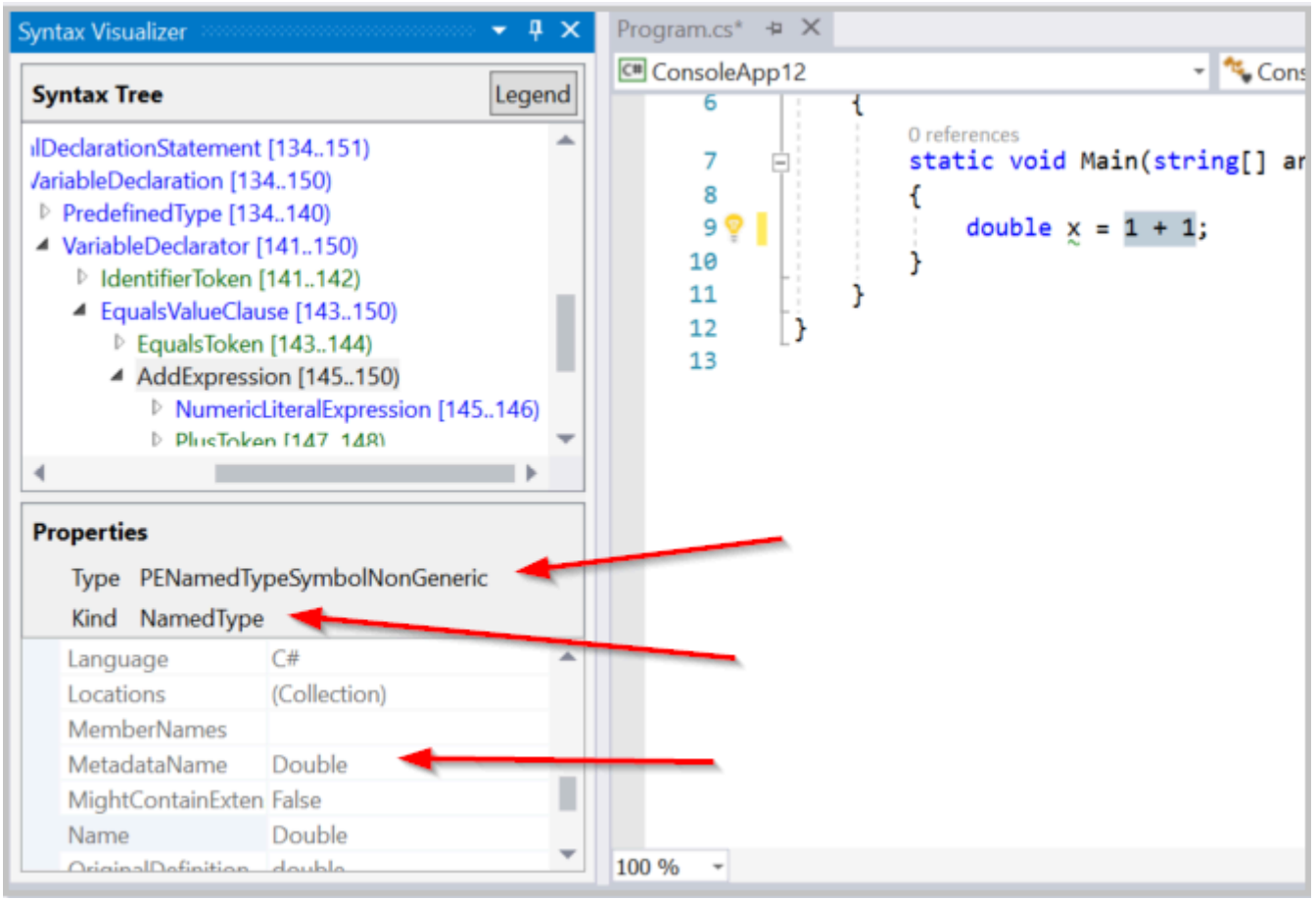
Сетка свойств в визуализаторе обновляется, как показано на следующем рисунке: символ выражения — **SynthesizedIntrinsicOperatorSymbol** с атрибутом **Kind = Method**.



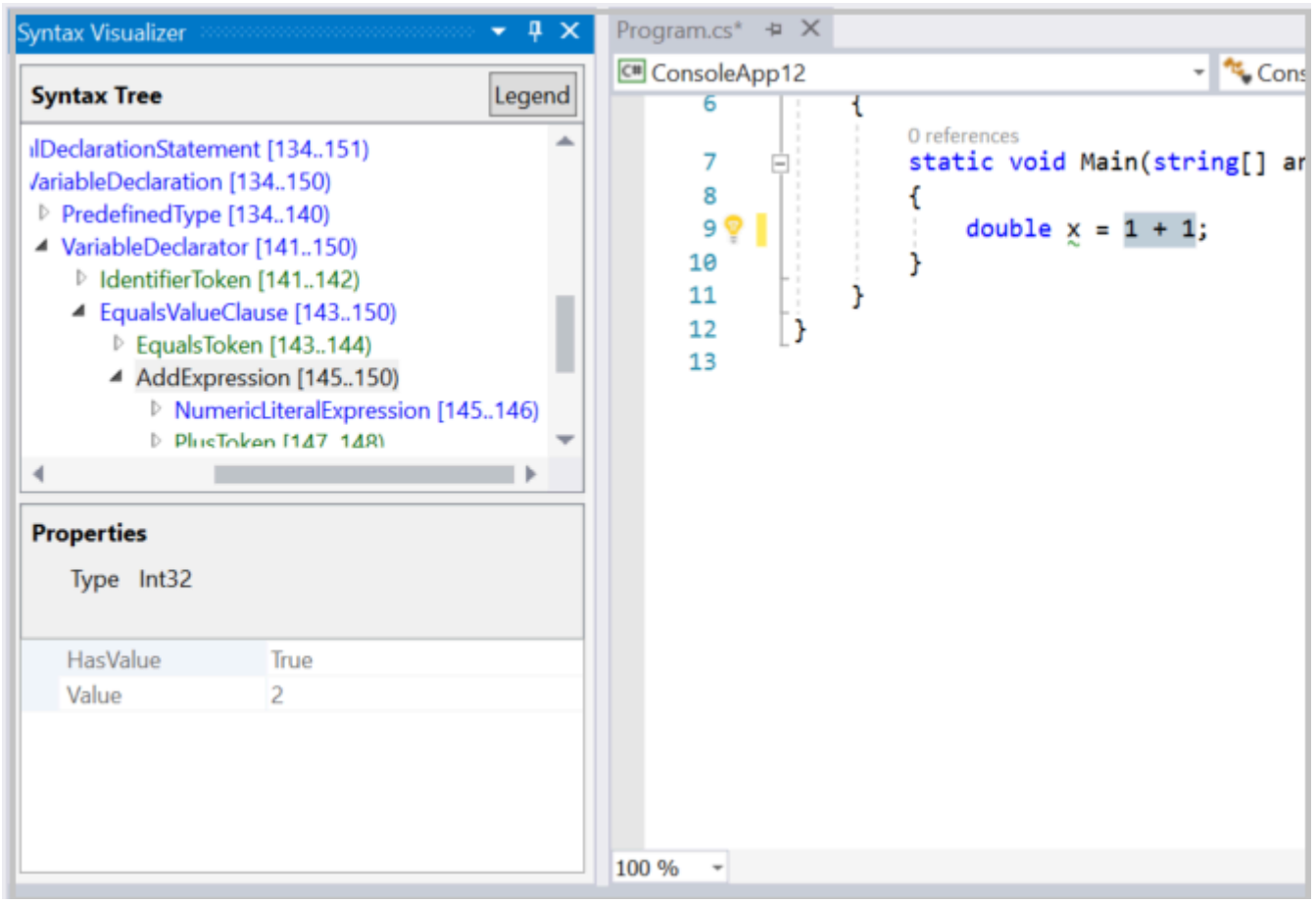
Щелкните правой кнопкой мыши тот же узел **AddExpression** и выберите **View TypeSymbol (если таковые есть)**. Сетка свойств в визуализаторе обновляется, как показано на следующем рисунке, указывая на тип выбранного выражения `Int32`.



Щелкните правой кнопкой мыши тот же узел **AddExpression** и выберите **Просмотр преобразованного TypeSymbol (если есть)**. Сетка свойств обновляется и указывает, что хотя тип выражения — `Int32`, преобразованный тип выражения — `Double`, как показано на следующем рисунке. Этот узел содержит информацию о преобразованных символах типа, потому что `Int32` выражение возникает в контексте, который требует его преобразования в `Double`. Это преобразование удовлетворяет типу `Double` , указанному для переменной `x` слева от оператора назначения.



Наконец, щелкните правой кнопкой мыши тот же узел **AddExpression**, и выберите **Просмотреть значение константы (если таковое имеется)**. Сетка свойств показывает, что значение выражения является константой времени компиляции со значением `2`.



Предыдущий пример также можно реплицировать в Visual Basic. Введите `Dim x As Double = 1 + 1` в файл Visual Basic. Выберите выражение `1 + 1` в окне редактора кода. Визуализатор выделяет соответствующий узел **AddExpression** в визуализаторе. Повторите предыдущие шаги для этого **AddExpression** , и вы увидите идентичные результаты.

Ознакомьтесь с дополнительным кодом в Visual Basic. Обновите основной файл Visual Basic следующим кодом:

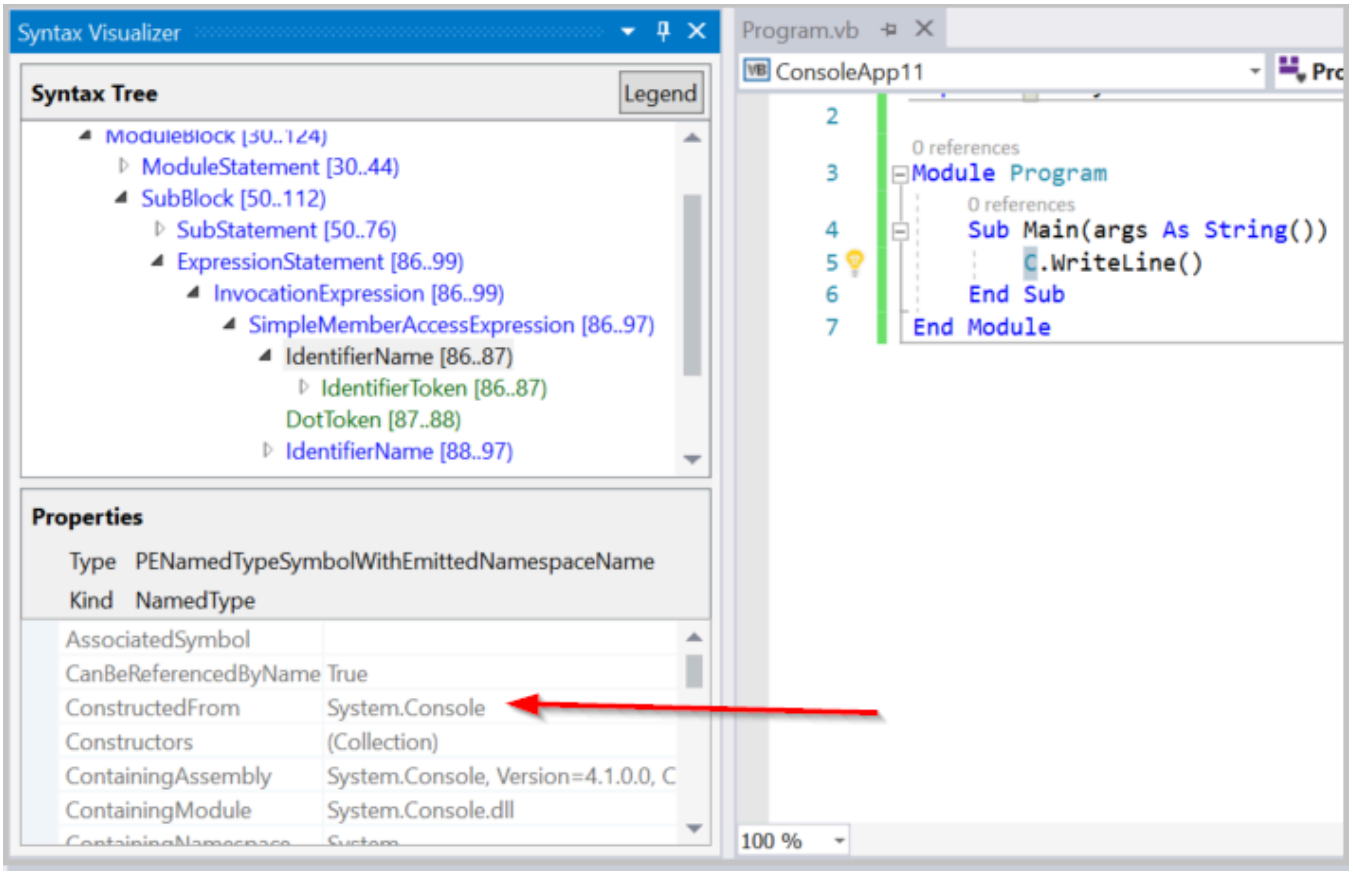
VB

```
Imports C = System.Console

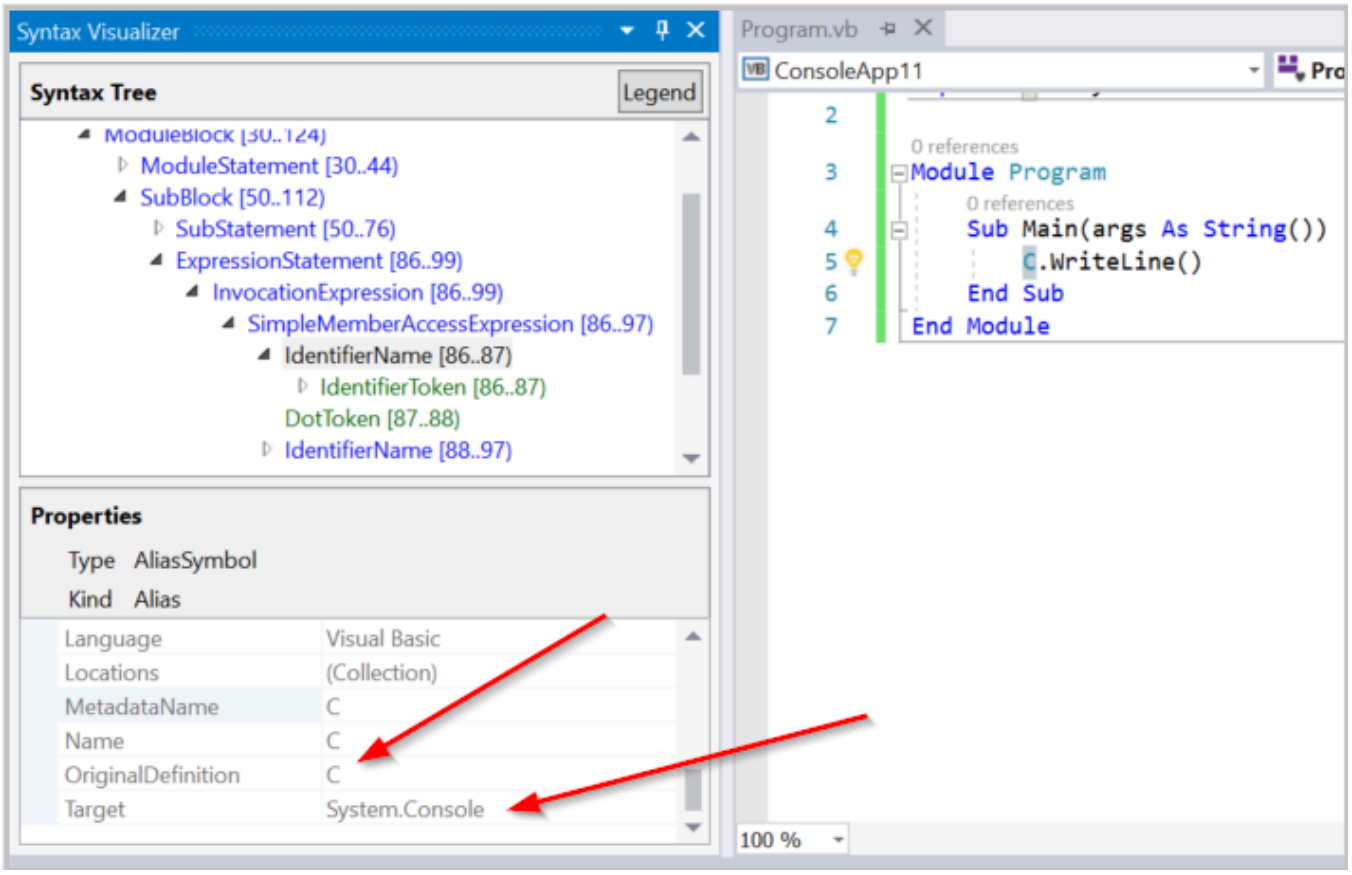
Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

Этот код представляет псевдоним `C` , который сопоставляется с типом `System.Console` в верхней части файла и использует этот псевдоним внутри `Main()`. Выберите использование этого псевдонима `C` в `C.WriteLine()`, в методе `Main()`. Визуализатор выбирает соответствующий узел **IdentifierName** в визуализаторе. Щелкните этот узел правой кнопкой мыши и нажмите **смотреть символ (если он есть)**. Сетка свойств

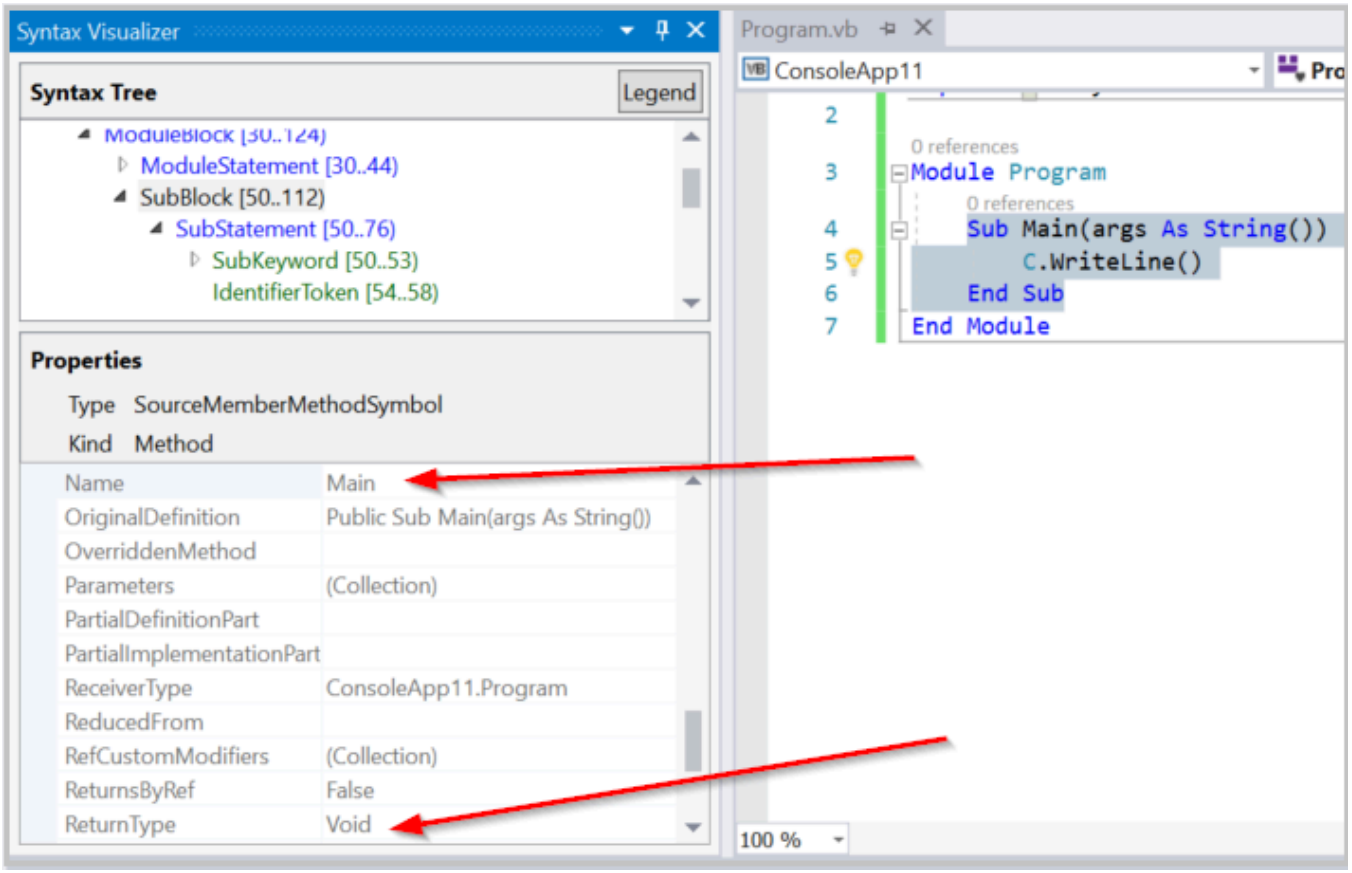
указывает, что этот идентификатор привязан к типу `System.Console` , как показано на следующем рисунке:



Щелкните правой кнопкой мыши тот же узел **IdentifierName** и выберите **View AliasSymbol (если таковые есть)**. Сетка свойств указывает, что идентификатор является псевдонимом с именем `C` , привязанным к целевому объекту `System.Console` . Другими словами, сетка свойств предоставляет сведения об **ПсевдонимеSymbol** , соответствующем идентификатору `C`.



Проверьте символ, соответствующий любому объявленному типу, методу, свойству. Выберите соответствующий узел в визуализаторе и щелкните правой кнопкой мыши, чтобы получить доступ к **символу представления (если таковой имеется)**. Выберите метод `Sub Main()`, включая текст метода. Щелкните правой кнопкой мыши соответствующий узел **SubBlock** в визуализаторе и выберите **значок представления (при наличии).** В сетке свойств отображается **MethodSymbol** для этого **SubBlock** с именем `Main` и типом возвращаемого значения `Void`.



Приведенные выше примеры Visual Basic можно легко реплицировать в C#. Введите `using C = System.Console;` вместо `Imports C = System.Console` для псевдонима. Предыдущие шаги в C# дают идентичные результаты в окне визуализатора.

Операции семантической проверки доступны только на узлах. Они недоступны для токенов или тривии. Не все узлы имеют интересную семантику для проверки. Если узел не содержит интересных семантических сведений, щелкните правой кнопкой мыши и выберите "**Просмотр * Символ (если доступен)**", чтобы увидеть пустую сетку свойств.

Дополнительные сведения об API-интерфейсах для выполнения семантического анализа см. в документе " Работа с семантикой ".

Закрытие визуализатора синтаксиса

Окно визуализатора можно закрыть, если он не используется для изучения исходного кода. Визуализатор синтаксиса обновляет его отображение при переходе по коду, редактированию и изменению источника. Это может отвлекать, когда вы не используете его.

Выбор идентификаторов диагностики

17.06.2025

Идентификатор диагностики — это строка, связанная с данной диагностикой, например ошибка компилятора или диагностика, созданная анализатором.

Идентификаторы извлекаются из различных API, таких как:

- DiagnosticDescriptor.Id
- ObsoleteAttribute.DiagnosticId
- ExperimentalAttribute.DiagnosticId

Идентификаторы диагностики также используются в качестве идентификаторов в источнике, например, из #pragma warning disable или .editorconfig файлов.

Соображения

- Идентификаторы диагностики должны быть уникальными
- Идентификаторы диагностики должны быть допустимыми идентификаторами в C#
- Идентификаторы диагностики должны содержать менее 15 символов
- Идентификаторы диагностики должны иметь форму `<PREFIX><number>`
  - Префикс зависит от проекта
  - Число представляет конкретную диагностику

> ⓘ **Примечание**
>
> Изменение диагностических идентификаторов приводит к разрушительным последствиям для исходного кода, поскольку существующие подавления будут игнорироваться, если идентификатор изменится.

Не ограничивайте префикс двумя символами (например `CSXXX`, и `CAXXXX`). Вместо этого используйте более длинный префикс, чтобы избежать конфликтов. Например, диагностики `System.*` используют `SYSLIB` в качестве префикса.

Начало работы с анализом синтаксиса

В этом руководстве вы изучите **API синтаксиса**. API синтаксиса предоставляет доступ к структурам данных, описывающим программу C# или Visual Basic. Эти структуры данных имеют достаточно подробных сведений, что они могут полностью представлять любую программу любого размера. Эти структуры могут описать полные программы, которые компилируются и выполняются правильно. Они также могут описать неполные программы, как вы их пишете, в редакторе.

Чтобы включить это расширенное выражение, структуры данных и API, составляющие API синтаксиса, обязательно являются сложными. Начнем с того, как выглядит структура данных для типичной программы Hello World:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Просмотрите текст предыдущей программы. Вы распознаете знакомые элементы. Весь текст представляет один исходный файл или **единицу компиляции**. Первые три строки исходного файла **используют директивы**. Оставшийся источник содержится в **объявлении пространства имен**. Объявление пространства имен содержит объявление дочернего **класса**. Объявление класса содержит одно **объявление метода**.

API синтаксиса создает структуру дерева с корнем, представляющим единицу компиляции. Узлы в дереве представляют `using` директивы, объявление пространства имен и все остальные элементы программы. Структура дерева продолжается до самых низких уровней: строка "Hello World!" является **строковым литеральным маркером** , который является потомком **аргумента**. API синтаксиса предоставляет доступ к структуре программы. Вы можете запросить конкретные методики кода, ходить по всему дереву, чтобы понять код и создать новые деревья, изменив существующее дерево.

Это краткое описание содержит обзор типа информации, доступной с помощью API синтаксиса. API синтаксиса — это не более формального API, описывающего знакомые конструкции кода, которые вы знаете из C#. Полные возможности включают сведения о форматировании кода, включая разрывы строк, пробелы и отступы. Используя эти сведения, вы можете полностью представить код как написанный и читаемый программистами или компилятором. Эта структура позволяет взаимодействовать с исходным кодом на глубоко понятном уровне. Это больше не текстовые строки, но данные, представляющие структуру программы C#.

Чтобы приступить к работе, необходимо установить **пакет SDK платформы компилятора .NET**:

Инструкции по установке — Visual Studio Installer

Существует два разных способа найти **пакет SDK платформы компилятора .NET** в **установщике Visual Studio**:

Установка с помощью установщика Visual Studio — представление рабочих нагрузок

Пакет SDK платформы компилятора .NET не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как необязательный компонент.

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Проверьте рабочую нагрузку **разработки расширений Visual Studio** .
4. Откройте узел **разработки расширений Visual Studio** в дереве сводки.
5. Убедитесь, что установлен флажок **для пакета SDK для платформы компилятора .NET** .
6. Нажмите кнопку **Изменить**.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Откройте узел **отдельных компонентов** в дереве сводки.
2. Установите флажок для **редактора DGML**

Установка с помощью установщика Visual Studio — вкладка "Отдельные компоненты"

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Выберите вкладку **"Отдельные компоненты"**
4. Установите флажок для **пакета SDK платформы компилятора .NET**. Его можно найти в верхней части раздела **"Компиляторы", "Средства сборки" и "Среды выполнения** ".
5. Нажмите кнопку **Изменить**.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Установите флажок **для редактора DGML**. Его можно найти в разделе **"Инструменты кода** ".

Общие сведения о деревах синтаксиса

Для анализа структуры кода C# используется API синтаксиса. **API синтаксиса** предоставляет средства синтаксического анализа, деревья синтаксиса и служебные программы для анализа и создания деревьев синтаксиса. Это способ поиска в коде определенных элементов синтаксиса или чтения кода программы.

Дерево синтаксиса — это структура данных, используемая компиляторами C# и Visual Basic для понимания программ C# и Visual Basic. Деревья синтаксиса создаются тем же средством синтаксического анализа, который выполняется при построении проекта или нажатии разработчика F5. Деревья синтаксиса имеют полное соответствие с языком; каждый бит информации в файле кода представлен в дереве. Написание дерева синтаксиса в текст воспроизводит точный исходный текст, который был проанализирован. Деревья синтаксиса также **неизменяемы**; После создания дерева синтаксиса никогда не может быть изменено. Потребители деревьев могут анализировать деревья на нескольких потоках без блокировки или других мер параллелизма, зная, что данные не меняются. API можно использовать для создания новых деревьев, которые являются результатом изменения существующего дерева.

Четыре основных блока деревьев синтаксиса:

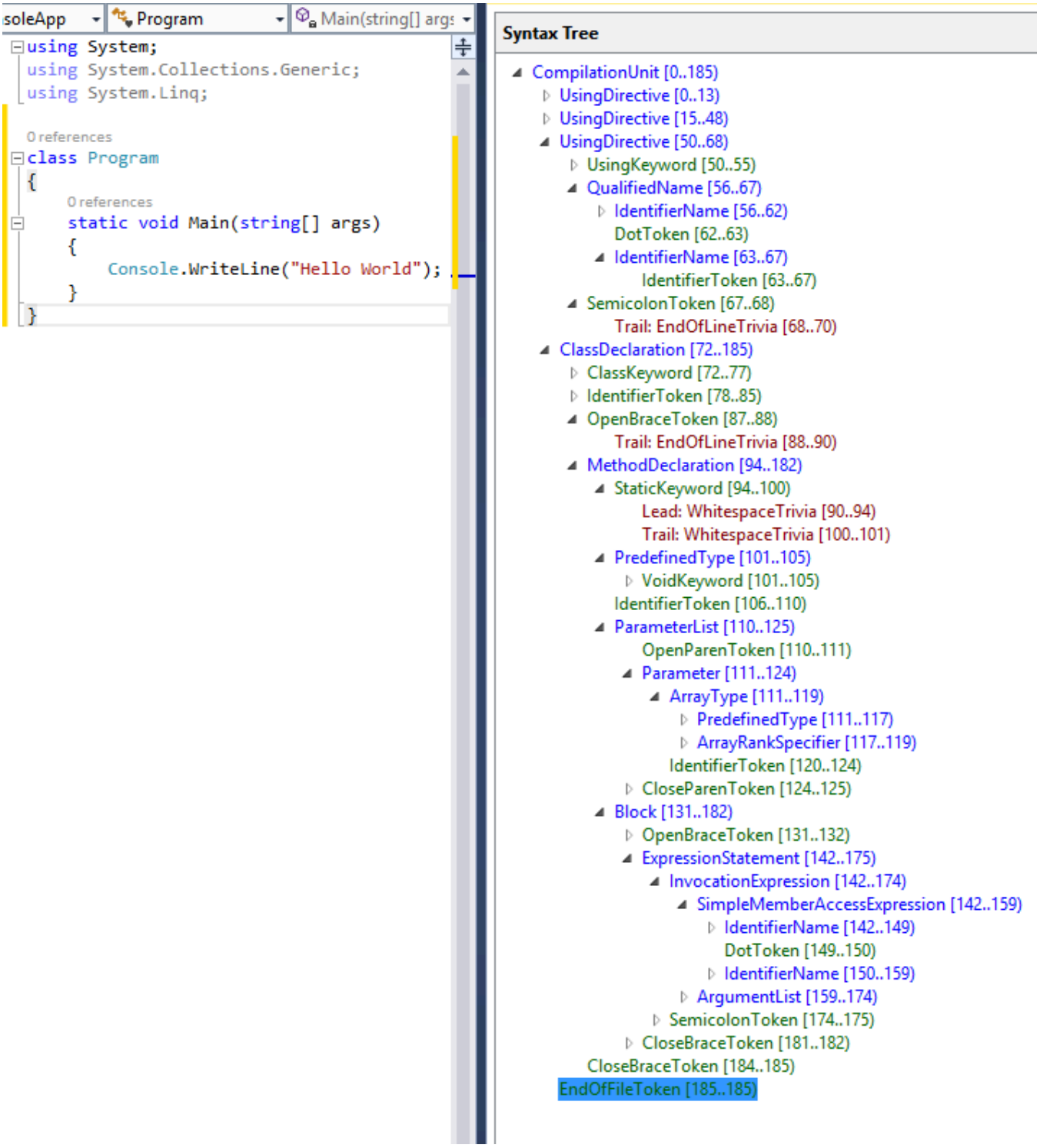
- Класс Microsoft.CodeAnalysis.SyntaxTree , экземпляр которого представляет целое дерево синтаксического анализа. SyntaxTree — абстрактный класс, имеющий производные от языка. Методы Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree синтаксического анализа класса (или Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree) используются для анализа текста в C# (или Visual Basic).
- Класс Microsoft.CodeAnalysis.SyntaxNode , экземпляры которых представляют синтаксические конструкции, такие как объявления, операторы, предложения и

выражения.

- Структура Microsoft.CodeAnalysis.SyntaxToken , представляющая отдельное ключевое слово, идентификатор, оператор или знак препинания.
- И наконец Microsoft.CodeAnalysis.SyntaxTrivia , структура, представляющая синтаксически незначительные биты информации, такие как пробел между токенами, директивами предварительной обработки и комментариями.

Тривия, маркеры и узлы иерархически структурированы для формирования дерева, полностью представляющего все в фрагменте кода Visual Basic или C#. Эта структура отображается с помощью окна **визуализатора синтаксиса** . В Visual Studio выберите **Вид**>**Другие окна**>**Визуализатор синтаксиса**. Например, предыдущий исходный файл C#, рассмотренный с помощью **визуализатора синтаксиса** , выглядит следующим образом:

SyntaxNode: Синий | **SyntaxToken**: Зеленый | **SyntaxTrivia**: Красный



Перейдя к этой структуре дерева, вы можете найти любую инструкцию, выражение, маркер или бит пробела в файле кода.

Хотя вы можете найти что-либо в файле кода с помощью API синтаксиса, большинство сценариев включают изучение небольших фрагментов кода или поиск конкретных инструкций или фрагментов. В двух примерах, приведенных ниже, показаны типичные способы просмотра структуры кода или поиска отдельных инструкций.

Обход деревьев

Узлы можно проверить в дереве синтаксиса двумя способами. Вы можете пройти по дереву для проверки каждого узла или запросить определенные элементы или узлы.

Обход вручную

Готовый код для этого примера можно увидеть в нашем репозитории GitHub.

> **Примечание.**
>
> Типы дерева синтаксиса используют наследование для описания различных элементов синтаксиса, допустимых в разных расположениях программы. Использование этих API часто означает приведение свойств или членов коллекции к определенным производным типам. В следующих примерах присваивание и приведение данных выполняются как отдельные инструкции с использованием явно типизированных переменных. Вы можете прочитать код, чтобы просмотреть возвращаемые типы API и тип среды выполнения возвращаемых объектов. На практике чаще используются неявно типизированные переменные и используются имена API для описания типа проверяемых объектов.

Создайте новый проект C# для **автономного средства анализа кода**.

- В Visual Studio выберите "**Файл**>**нового проекта**">, чтобы отобразить диалоговое окно "Новый**проект**".
- В **разделе "Расширяемость Visual C#"**> выберите **самостоятельное средство анализа кода**.
- Присвойте проекту имя "**SyntaxTreeManualTraversal**" и нажмите кнопку "OK".

Вы собираетесь проанализировать базовую программу "Hello World!", показанную ранее. Добавьте текст для программы Hello World в качестве константы в `Program` классе:

C#

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
```

```
        }
    }
}";
```

Затем добавьте следующий код, чтобы создать **дерево синтаксиса** для текста кода в `programText` константе. Добавьте следующую строку в `Main` метод:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Эти две строки создают дерево и извлекают корневой узел этого дерева. Теперь вы можете проверить узлы в дереве. Добавьте эти строки в `Main` метод, чтобы отобразить некоторые свойства корневого узла в дереве:

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using directives. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Запустите приложение, чтобы узнать, что ваш код обнаружил о корневом узле в этом дереве.

Как правило, вы проходите по дереву, чтобы изучить код. В этом примере вы анализируете код, который вы знаете, чтобы изучить API. Добавьте следующий код для проверки первого члена `root` узла:

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Этот элемент является Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. Он описывает всё в области объявления `namespace HelloWorld`. Добавьте следующий код, чтобы определить, какие узлы объявлены внутри пространства имен `HelloWorld`.

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared in
this namespace.");
```

```
WriteLine($"The first member is a {helloWorldDeclaration.Members[0].Kind()}.");
```

Запустите программу, чтобы узнать, что вы узнали.

Теперь, когда вы знаете, что объявление относится к типу Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, объявите новую переменную этого типа, чтобы изучить объявление класса. Этот класс содержит только один член: `Main` метод. Добавьте следующий код, чтобы найти метод `Main` и привести его к Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

C#

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in the
{programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

Узел объявления метода содержит все синтаксические сведения о методе. Давайте отобразим возвращаемый тип `Main` метода, число и типы аргументов и текст текста метода. Добавьте следующий код:

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is
{mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}
parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is {item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method follows:");
WriteLine(mainDeclaration.Body?.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Запустите программу, чтобы просмотреть все обнаруженные сведения об этой программе:

text

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using directives. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
```

```
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Методы запроса

Помимо обхода деревьев, вы также можете изучить дерево синтаксиса с помощью методов запроса, определенных в Microsoft.CodeAnalysis.SyntaxNode. Эти методы должны быть немедленно знакомы всем, кто знаком с XPath. Эти методы можно использовать с LINQ, чтобы быстро найти вещи в дереве. У SyntaxNode имеются методы запроса, такие как DescendantNodes, AncestorsAndSelf и ChildNodes.

Эти методы запроса можно использовать для поиска аргумента `Main` метода в качестве альтернативы навигации по дереву. Добавьте следующий код в конец метода `Main`:

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

Первая инструкция использует выражение LINQ и метод для поиска того же параметра, что и DescendantNodes в предыдущем примере.

Запустите программу, и вы увидите, что выражение LINQ обнаружило тот же параметр, что и навигация по дереву вручную.

В примере используются операторы `WriteLine` для отображения информации о деревах синтаксиса при их обходе. Вы также можете узнать больше, выполнив завершенную программу под отладчиком. Вы можете изучить больше свойств и методов, которые являются частью дерева синтаксиса, созданного для программы hello world.

Обходчики синтаксиса

Часто требуется найти все узлы определенного типа в дереве синтаксиса, например каждое объявление свойств в файле. Расширяя класс и переопределяя Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalkerVisitPropertyDeclaration(PropertyDeclarationSyntax) метод, вы обрабатываете каждое объявление свойств в дереве синтаксиса, не зная его структуру заранее. CSharpSyntaxWalker — это конкретный вид CSharpSyntaxVisitor , который рекурсивно посещает узел и каждый из его дочерних элементов.

В этом примере реализуется CSharpSyntaxWalker, который анализирует дерево синтаксиса. Он собирает `using` директивы, которые он находит и которые не импортируют пространство имен `System`.

Создайте новый проект на C# **Standalone-средства анализа кода**; присвойте ему имя "**SyntaxWalker**".

Готовый код для этого примера можно увидеть в нашем репозитории GitHub. Пример на GitHub содержит оба проекта, описанные в этом руководстве.

Как и в предыдущем примере, можно определить строковую константу для хранения текста программы, которую вы собираетесь проанализировать:

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;
```

```
        class Bar { }
    }
}";
```

Этот исходный текст содержит `using` директивы, разбросанные между четырьмя разными расположениями: на уровне файла, в пространстве имен верхнего уровня и в двух вложенных пространствах имен. В этом примере выделен основной сценарий для использования CSharpSyntaxWalker класса для запроса кода. Было бы сложно посетить каждый узел в корневом дереве синтаксиса, чтобы найти операторы using. Вместо этого создается производный класс и переопределяется метод, который вызывается только в том случае, если текущий узел в дереве является директивой `using` . Ваш посетитель не работает ни с какими другими типами узлов. Этот метод проверяет каждую из директив `using` и создает коллекцию пространств имен, которые не входят в пространство имен `System`. Вы создаете сборку CSharpSyntaxWalker , которая проверяет все `using` директивы, но только `using` директивы.

Теперь, когда вы определили текст программы, необходимо создать `SyntaxTree` и получить корень этого дерева:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Затем создайте новый класс. В Visual Studio выберите "**Добавить новый элемент**>". В диалоговом окне **Добавление нового элемента** введите в качестве имени файла *UsingCollector.cs*.

Вы реализуете `using` функции посетителя в `UsingCollector` классе. Начните с того, чтобы класс `UsingCollector` наследовался от CSharpSyntaxWalker.

C#

```
class UsingCollector : CSharpSyntaxWalker
```

Требуется хранилище для удержания собираемых узлов пространства имен. Объявите общедоступное свойство, доступное только для чтения, в классе `UsingCollector`, чтобы использовать эту переменную для хранения найденных узлов UsingDirectiveSyntax.

C#

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new
```

```
List<UsingDirectiveSyntax>();
```

Базовый класс CSharpSyntaxWalker реализует логику для посещения каждого узла в дереве синтаксиса. Производный класс переопределяет методы, вызываемые для определенных узлов, которые вам нужны. В этом случае вы заинтересованы в любой `using` директиве. Это означает, что необходимо переопределить VisitUsingDirective(UsingDirectiveSyntax) метод. Одним из аргументов этого метода является Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax объект. Это важное преимущество использования паттерна Посетитель: они вызывают переопределенные методы с аргументами, уже приведёнными к конкретному типу узла. Класс Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax имеет Name свойство, которое сохраняет имя импортируемого пространства имен. Это Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Добавьте следующий код в VisitUsingDirective(UsingDirectiveSyntax) переопределение:

C#

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

Как и в предыдущем примере, вы добавили различные `WriteLine` инструкции для понимания этого метода. Вы можете видеть, когда он вызывается, и какие аргументы передаются в него каждый раз.

Наконец, необходимо добавить две строки кода, чтобы создать `UsingCollector`, который обходит корневой узел и собирает все директивы `using`. Затем добавьте цикл `foreach`, чтобы отобразить все директивы `using`, найденные вашим сборщиком:

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Скомпилируйте и запустите программу. Вы увидите следующие выходные данные:

Консоль

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Поздравляем! **Вы использовали API синтаксиса** для поиска определенных типов директив и объявлений в исходном коде C#.

Last updated on 26.02.2026

Начало работы с семантическим анализом

В этом руководстве предполагается, что вы знакомы с API синтаксиса. В статье о начале работы с синтаксическим анализом содержится достаточное введение.

В этом руководстве вы изучите API символов и привязки. Эти API предоставляют сведения о *семантическом значении* программы. Они позволяют задавать и отвечать на вопросы о типах, представленных любым символом в программе.

Необходимо установить **пакет SDK платформы компилятора .NET**:

Инструкции по установке — Visual Studio Installer

Существует два разных способа найти **пакет SDK платформы компилятора .NET** в **установщике Visual Studio**:

Установка с помощью установщика Visual Studio — представление рабочих нагрузок

Пакет SDK платформы компилятора .NET не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как необязательный компонент.

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Проверьте рабочую нагрузку **разработки расширений Visual Studio** .
4. Откройте узел **разработки расширений Visual Studio** в дереве сводки.
5. Убедитесь, что установлен флажок **для пакета SDK для платформы компилятора .NET** .
6. Нажмите кнопку **Изменить**.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Откройте узел **отдельных компонентов** в дереве сводки.
2. Установите флажок для **редактора DGML**

Установка с помощью установщика Visual Studio — вкладка "Отдельные компоненты"

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Выберите вкладку **"Отдельные компоненты"**
4. Установите флажок для **пакета SDK платформы компилятора .NET**. Его можно найти в верхней части раздела **"Компиляторы", "Средства сборки" и "Среды выполнения "**.
5. Нажмите кнопку **Изменить**.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Установите флажок **для редактора DGML**. Его можно найти в разделе **"Инструменты кода "**.

Общие сведения о компиляциях и символах

При работе с пакетом SDK компилятора .NET вы знакомы с различиями между API синтаксиса и семантической API. **API синтаксиса** позволяет просмотреть *структуру* программы. Однако часто требуется более подробная информация о семантике или *значении* программы. Хотя свободный файл кода или фрагмент кода Visual Basic или C# можно синтаксически анализировать в изоляции, не имеет смысла задавать такие вопросы, как "тип этой переменной" в вакууме. Значение имени типа может зависеть от ссылок на сборки, импорта пространства имен или других файлов кода. Эти вопросы отвечают с помощью **семантического API**, в частности Microsoft.CodeAnalysis.Compilation класса.

Экземпляр Compilation аналогичен одному проекту, как показано компилятором и представляет все необходимое для компиляции программы Visual Basic или C#. **Компиляция** включает набор исходных файлов для компиляции, ссылок на сборки и параметров компилятора. Вы можете подумать о значении кода, используя все остальные сведения в этом контексте. А Compilation позволяет находить сущности, такие как **символы**, включая типы, пространства имен, члены и переменные, к которым относятся имена и другие выражения. Процесс связывания имен и выражений с **символами** называется **Привязкой**.

Например Microsoft.CodeAnalysis.SyntaxTree, Compilation является абстрактным классом с производными от языка. При создании экземпляра компиляции необходимо вызвать метод фабрики класса Microsoft.CodeAnalysis.CSharp.CSharpCompilation (или Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation).

Запрос символов

В этом руководстве вы снова посмотрите на программу Hello World. На этот раз вы запрашиваете символы в программе, чтобы понять, какие типы символы представляют. Вы запрашиваете типы в пространстве имен и учитесь находить доступные методы в типе.

Готовый код для этого примера можно увидеть в нашем репозитории GitHub.

> **Примечание.**
>
> Типы дерева синтаксиса используют наследование для описания различных элементов синтаксиса, допустимых в разных расположениях программы. Использование этих API часто означает приведение свойств или членов коллекции к определенным производным типам. В следующих примерах присваивание и приведение данных выполняются как отдельные инструкции с использованием явно типизированных переменных. Вы можете прочитать код, чтобы просмотреть возвращаемые типы API и тип среды выполнения возвращаемых объектов. На практике чаще используются неявно типизированные переменные и используются имена API для описания типа проверяемых объектов.

Создайте новый проект C# для **автономного средства анализа кода**.

- В Visual Studio выберите "**Файл**>**нового проекта**">, чтобы отобразить диалоговое окно "Новый**проект**".
- В **разделе "Расширяемость Visual C#"**> выберите **самостоятельное средство анализа кода**.
- Присвойте проекту имя "**SemanticQuickStart**" и нажмите кнопку "OK".

Вы собираетесь проанализировать базовую программу "Hello World!", показанную ранее. Добавьте текст для программы Hello World в качестве константы в `Program` классе:

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
```

```
    }
}";
```

Затем добавьте следующий код, чтобы создать дерево синтаксиса для текста кода в `programText` константе. Добавьте следующую строку в `Main` метод:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Затем создайте CSharpCompilation из дерева, которое вы уже создали. Пример «Hello World» основан на типах String и Console. Необходимо ссылаться на сборку, которая объявляет эти два типа в процессе компиляции. Добавьте следующую строку в `Main` метод для создания компиляции дерева синтаксиса, включая ссылку на соответствующую сборку:

C#

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

Метод CSharpCompilation.AddReferences добавляет ссылки на компиляцию. Метод MetadataReference.CreateFromFile загружает сборку в качестве ссылки.

Запрос семантической модели

После того как у вас будет Compilation, вы можете запросить у него SemanticModel для любого SyntaxTree, содержащегося в указанном Compilation. Вы можете рассматривать семантику модели как источник для всех сведений, которые вы обычно получаете от intellisense. Выражение SemanticModel может отвечать на вопросы, такие как "Какие имена находятся в области видимости в этой точке?", "Какие элементы доступны из этого метода?", "Какие переменные используются в этом блоке текста?" и "На что ссылается это имя или выражение?" Добавьте это выражение для создания семантической модели.

C#

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Привязка имени

Compilation создает объект SemanticModel из SyntaxTree. После создания модели его можно запросить, чтобы найти первую `using` директиву и получить сведения о символах для `System` пространства имен. Добавьте эти две строки в `Main` метод для создания семантической модели и получения символа для первой `using` директивы:

C#

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

В приведенном выше коде показано, как связать имя в первой директиве `using`, чтобы получить элемент Microsoft.CodeAnalysis.SymbolInfo для пространства имен `System`. Приведенный выше код также иллюстрирует использование **модели синтаксиса** для поиска структуры кода; вы используете **семантику модели** для понимания его смысла. Модель **синтаксиса** находит строку `System` в директиве `using` . **Семантическая модель** содержит все сведения о типах, определенных в `System` пространстве имен.

Из объекта SymbolInfo можно получить Microsoft.CodeAnalysis.ISymbol с помощью свойства SymbolInfo.Symbol. Это свойство возвращает символ, на который ссылается это выражение. Для выражений, которые не ссылаются ни на что (например, числовые литералы) это свойство `null`. SymbolInfo.Symbol Если значение не равно NULL, ISymbol.Kind он обозначает тип символа. В этом примере свойство ISymbol.Kind является SymbolKind.Namespace. Добавьте следующий код в `Main` метод. Он извлекает символ для `System` пространства имен, а затем отображает все дочерние пространства имен, объявленные в `System` пространстве имен:

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;
if (systemSymbol?.GetNamespaceMembers() is not null)
{
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
    {
        Console.WriteLine(ns);
    }
}
```

Запустите программу и вы увидите следующие выходные данные:

Выходные данные

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

> ⓘ **Примечание.**
>
> Выходные данные не включают все пространства имен, которые являются дочерними для пространства имен `System`. Отображаются все пространства имен, присутствующие в данной компиляции, которая ссылается только на сборку, где объявляется `System.String`. Все пространства имен, объявленные в других сборках, не известны этой компиляции

Привязка выражения

В приведенном выше коде показано, как найти символ путем привязки к имени. В программе C# существуют другие выражения, которые могут быть привязаны, и они не обязательно являются именами. Чтобы продемонстрировать эту возможность, давайте перейдем к привязке к простому строковому литералу.

Программа Hello World содержит Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntaxстроку "Hello, World!", отображаемую в консоли.

Вы найдете строку "Hello, World!", найдя единственный строковый литерал в программе. Затем после того, как вы нашли узел синтаксиса, получите сведения о типе для этого узла из семантической модели. Добавьте следующий код в `Main` метод:

C#

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
```

```
.OfType<LiteralExpressionSyntax>()
.Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

Структура Microsoft.CodeAnalysis.TypeInfo включает в себя свойство TypeInfo.Type, которое позволяет получить доступ к семантической информации о типе литерала. В этом примере тип — это `string`. Добавьте объявление, которое назначает это свойство локальной переменной:

C#

```
var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

Чтобы завершить работу с этим руководством, давайте создадим запрос LINQ, который создает последовательность всех открытых методов, объявленных в типе `string`, которые возвращают `string`. Данный запрос становится сложным, поэтому давайте построим его построчно, а затем соберем в виде одного полного запроса. Источником этого запроса является последовательность всех членов, объявленных в типе `string` :

C#

```
var allMembers = stringTypeSymbol?.GetMembers();
```

Эта исходная последовательность содержит все элементы, включая свойства и поля, поэтому фильтруйте его с помощью ImmutableArray<T>.OfType метода для поиска элементов, являющихся Microsoft.CodeAnalysis.IMethodSymbol объектами:

C#

```
var methods = allMembers?.OfType<IMethodSymbol>();
```

Затем добавьте еще один фильтр, чтобы вернуть только те методы, которые являются публичными и возвращают `string`:

C#

```
var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Выберите только свойство name и только отдельные имена, удаляя все перегрузки:

C#

```
var distinctMethods = publicStringReturningMethods?.Select(m => m.Name).Distinct();
```

Вы также можете создать полный запрос с помощью синтаксиса запроса LINQ, а затем отобразить все имена методов в консоли:

C#

```
foreach (string name in (from method in stringTypeSymbol?
                         .GetMembers().OfType<IMethodSymbol>()
                         where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol) &&
                         method.DeclaredAccessibility == Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Создайте и запустите программу. Вы увидите следующие выходные данные:

Выходные данные

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

Вы использовали семантический API для поиска и отображения сведений о символах, входящих в эту программу.

Last updated on 26.02.2026

Начало работы с преобразованием синтаксиса

В этом руководстве описаны основные понятия и методы, рассмотренные в руководстве по началу работы с синтаксическим анализом и краткими руководствами по началу работы с семантическим анализом . Если вы еще не сделали этого, перед началом работы с этим руководством необходимо выполнить эти краткие руководства.

В этом кратком руководстве вы изучите методы создания и преобразования деревьев синтаксиса. В сочетании с методами, которые вы узнали в предыдущих кратких руководствах, вы создадите первый рефакторинг для командной строки!

Инструкции по установке — Visual Studio Installer

Существует два разных способа найти **пакет SDK платформы компилятора .NET** в **установщике Visual Studio**:

Установка с помощью установщика Visual Studio — представление рабочих нагрузок

Пакет SDK платформы компилятора .NET не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как необязательный компонент.

1. Запуск **установщика Visual Studio**
2. Выберите **"Изменить"**
3. Проверьте рабочую нагрузку **разработки расширений Visual Studio** .
4. Откройте узел **разработки расширений Visual Studio** в дереве сводки.
5. Установите флажок для **пакета SDK платформы компилятора .NET**. Он будет находиться в последнем разделе необязательных компонентов.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Откройте узел **отдельных компонентов** в дереве сводки.
2. Установите флажок для **редактора DGML**

Установка с помощью установщика Visual Studio — вкладка "Отдельные компоненты"

1. Запуск **установщика Visual Studio**
2. Выберите **"Изменить"**
3. Выберите вкладку **"Отдельные компоненты"**
4. Установите флажок **.NET Compiler Platform SDK**. Его можно найти в верхней части раздела **"Компиляторы", "Средства сборки" и "Среды выполнения** ".

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Установите флажок **для редактора DGML**. Его можно найти в разделе **"Инструменты кода** ".

Неизменяемость и платформа компилятора .NET

Неизменяемость — это фундаментальный принцип платформы компилятора .NET. Неизменяемые структуры данных нельзя изменить после их создания. Неизменяемые структуры данных можно безопасно совместно использовать и анализировать несколькими потребителями одновременно. Нет никакой опасности, что один потребитель влияет на другого непредсказуемо. Анализатор не нуждается в блокировках или других мерах параллелизма. Это правило применяется к деревьям синтаксиса, компиляциям, символам, семантической модели и каждой другой структуре данных, с которой вы столкнулись. Вместо изменения существующих структур API создают новые объекты на основе указанных различий в старых. Это понятие применяется к деревьям синтаксиса для создания новых деревьев с помощью преобразований.

Создание и преобразование деревьев

Вы выбираете одну из двух стратегий для преобразований синтаксиса. **Фабричные методы** лучше всего использовать, когда вы ищете определенные узлы для замены или определенные места, в которые требуется вставить новый код. **Перезаписи** лучше всего использовать при сканировании всего проекта для шаблонов кода, которые необходимо заменить.

Создавайте узлы с помощью методов фабрики

Первое преобразование синтаксиса демонстрирует методы фабрики. Вы собираетесь заменить инструкцию `using System.Collections;` на инструкцию `using System.Collections.Generic;`. В этом примере показано, как создавать объекты Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode с помощью фабричных методов Microsoft.CodeAnalysis.CSharp.SyntaxFactory. Для каждого типа **узла**, **маркера** или **тривиа**

существует фабричный метод, который создает экземпляр этого типа. Вы создаете деревья синтаксиса, иерархически составляя узлы снизу вверх. Затем вы преобразуете существующую программу, заменив существующие узлы новым деревом, которое вы создали.

Запустите Visual Studio и создайте новый проект автономного средства анализа кода на C#. В Visual Studio выберите **"Файл>нового проекта"**>, чтобы отобразить диалоговое окно "Новый**проект**". В разделе "Расширяемость Visual C#"> выберите автономное средство анализа кода. В этом кратком руководстве есть два примера проектов, поэтому назовите решение **SyntaxTransformationQuickStart** и назовите проект **ConstructionCS**. Нажмите кнопку **OK**.

Этот проект использует методы класса Microsoft.CodeAnalysis.CSharp.SyntaxFactory для создания пространства имен Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax для `System.Collections.Generic`.

Добавьте следующую `using` директиву в верхнюю часть `Program.cs`.

C#

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

Вы создадите **узлы синтаксиса имен**, чтобы построить дерево, представляющее оператор `using System.Collections.Generic;`. NameSyntax — базовый класс для четырех типов имен, которые отображаются в C#. Вы создаете эти четыре типа имен вместе, чтобы создать любое имя, которое может появиться на языке C#:

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, представляющий простые имена отдельных идентификаторов, например `System` и `Microsoft`.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax, представляющий универсальный тип или имя метода, например `List<int>`.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax, представляющий полное имя формы `<left-name>.<right-identifier-or-generic-name>` , например `System.IO`.
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax, представляющий имя с использованием внешнего псевдонима сборки, например `LibraryV2::Foo`.

Вы используете метод IdentifierName(String) для создания узла NameSyntax. Добавьте следующий код в ваш метод `Main` в `Program.cs`.

C#

```
NameSyntax name = IdentifierName("System");
```

```
WriteLine($"\tCreated the identifier {name}");
```

Предыдущий IdentifierNameSyntax код создает объект и назначает его переменной `name`. Многие API Roslyn возвращают базовые классы, чтобы упростить работу с связанными типами. Переменную `name`, как NameSyntax, можно использовать повторно в процессе построения QualifiedNameSyntax. Не используйте вывод типов при сборке примера. Этот шаг будет автоматизироваться в этом проекте.

Вы создали имя. Теперь пришло время добавить больше узлов в структуру дерева, создав QualifiedNameSyntax. Новое дерево использует `name` слева от имени и новую IdentifierNameSyntax для пространства имен `Collections`, расположенного справа от QualifiedNameSyntax. Добавьте следующий код в `program.cs`:

C#

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Запустите код еще раз и просмотрите результаты. Вы создаете дерево узлов, представляющих код. Вы будете продолжать этот шаблон для создания пространства имен QualifiedNameSyntax для `System.Collections.Generic`. Добавьте следующий код в `Program.cs`:

C#

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Запустите программу еще раз, чтобы увидеть, что вы создали дерево для добавления кода.

Создание измененного дерева

Вы создали небольшое дерево синтаксиса, содержащее одну инструкцию. API для создания новых узлов являются правильным выбором для создания отдельных операторов или других небольших блоков кода. Однако для создания больших блоков кода следует использовать методы, заменяющие узлы или вставляющие узлы в существующее дерево. Помните, что деревья синтаксиса неизменяемы. **API синтаксиса** не предоставляет никакого механизма изменения существующего дерева синтаксиса после построения. Вместо этого он предоставляет методы, которые создают новые деревья на основе изменений существующих. `With*` методы определяются в конкретных классах, производных от SyntaxNode членов расширения, объявленных в SyntaxNodeExtensions

классе. Эти методы создают новый узел, применяя изменения к дочерним свойствам существующего узла. Кроме того, член расширения ReplaceNode можно использовать для замены узла-потомка в поддереве. Этот метод также обновляет родителя, чтобы указать на новый дочерний элемент, и повторяет этот процесс вверх по всему дереву - процесс, известный как перестроение дерева.

Следующим шагом является создание дерева, представляющего всю (небольшую) программу, а затем изменить ее. Добавьте следующий код в начало `Program` класса:

C#

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

> ⓘ **Заметка**
>
> В примере кода используется `System.Collections` пространство имен, а не `System.Collections.Generic` пространство имен.

Затем добавьте следующий код в нижней части `Main` метода, чтобы проанализировать текст и создать дерево:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

В этом примере используется метод WithName(NameSyntax) для замены имени в узле UsingDirectiveSyntax на то, что было сконструировано в предыдущем коде.

Создайте новый узел UsingDirectiveSyntax, используя метод WithName(NameSyntax), чтобы обновить имя `System.Collections` на имя, созданное в предыдущем коде. Добавьте

следующий код в нижней `Main` части метода:

```C#
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Запустите программу и внимательно изучите выходные данные. Он `newUsing` не был помещен в корневое дерево. Исходное дерево не было изменено.

Добавьте следующий код с помощью ReplaceNode метода расширения для создания нового дерева. Новое дерево является результатом замены существующего импорта обновленным `newUsing` узлом. Вы назначаете это новое дерево существующей `root` переменной:

```C#
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Снова запустите программу. На этот раз дерево правильно импортирует `System.Collections.Generic` пространство имен.

Преобразование деревьев с помощью `SyntaxRewriters`

Методы `With*` и ReplaceNode предоставляют удобные средства для преобразования отдельных ветвей дерева синтаксиса. Класс Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter выполняет несколько преобразований в дереве синтаксиса. Класс Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter является подклассом Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. Преобразование CSharpSyntaxRewriter применяется к определенному типу SyntaxNode. Преобразования можно применять к нескольким типам объектов, где бы они ни находились в дереве синтаксиса SyntaxNode . Второй проект в этом кратком руководстве создает рефакторинг командной строки, который удаляет явные типы в объявлениях локальных переменных в любом месте, где можно использовать вывод типов.

Создайте новый проект **автономного средства анализа кода** на C#. В Visual Studio щелкните правой кнопкой мыши по узлу решения `SyntaxTransformationQuickStart`. Нажмите кнопку "**Добавить**>**новый проект** ", чтобы отобразить **диалоговое окно "Новый проект**". В **разделе "Расширяемость Visual C#"**> выберите **самостоятельное средство анализа кода**. Присвойте проекту `TransformationCS` имя и нажмите кнопку "OK".

Первым шагом является создание класса, который наследуется от CSharpSyntaxRewriter для выполнения ваших преобразований. Добавьте новый файл класса в проект. В Visual Studio выберите **Проект**>**Добавить класс...**. В диалоговом окне **Добавление нового элемента** введите `TypeInferenceRewriter.cs` в качестве имени файла.

Добавьте в `TypeInferenceRewriter.cs` файл следующие директивы `using`:

C#

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Затем сделайте `TypeInferenceRewriter` класс расширением CSharpSyntaxRewriter класса:

C#

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Добавьте следующий код, чтобы объявить частное поле только для чтения для хранения SemanticModel и инициализации его в конструкторе. Это поле потребуется позже, чтобы определить, где можно использовать вывод типов:

C#

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel =
semanticModel;
```

Переопределите метод VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax).

C#

```
public override SyntaxNode
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> ⓘ **Заметка**
>
> Многие API Roslyn объявляют возвращаемые типы, которые являются базовыми классами фактических типов среды выполнения, возвращаемых. Во многих сценариях один тип узла может быть заменен другим типом узла полностью или

даже удален. В этом примере VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) метод возвращает SyntaxNodeвместо производного LocalDeclarationStatementSyntaxтипа. Этот переписыватель возвращает новый LocalDeclarationStatementSyntax узел на основе уже существующего.

В этом кратком руководстве обрабатываются объявления локальных переменных. Его можно расширить на другие объявления, такие как `foreach` циклы, `for` циклы, выражения LINQ и лямбда-выражения. Кроме того, этот переписчик преобразует только объявления простейшей формы:

C#

```
Type variable = expression;
```

Если вы хотите самостоятельно изучить, рассмотрите возможность расширения готового примера для следующих типов объявлений переменных:

C#

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Добавьте следующий код в тело метода `VisitLocalDeclarationStatement`, чтобы избежать перезаписи этих форм объявлений:

C#

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

Метод указывает, что перезапись не выполняется путем возврата `node` параметра, не измененного. Если ни одно из этих `if` выражений не истинно, узел представляет возможное объявление с инициализацией. Добавьте эти инструкции, чтобы извлечь имя типа, указанное в объявлении, и привязать его с помощью SemanticModel поля для получения символа типа:

C#

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Теперь добавьте эту инструкцию для привязки выражения инициализатора:

C#

```
var initializerInfo = SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Наконец, добавьте следующую `if` инструкцию, чтобы заменить существующее имя `var` типа ключевым словом, если тип выражения инициализатора соответствует указанному типу:

C#

```
if (SymbolEqualityComparer.Default.Equals(variableType, initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

Условие является обязательным, так как объявление может привести выражение инициализатора к базовому классу или интерфейсу. Если это необходимо, типы в левой и правой части назначения не совпадают. Удаление явного типа в этих случаях изменит семантику программы. `var` указывается в качестве идентификатора, а не ключевого слова, так как `var` это контекстное ключевое слово. Начальные и конечные пробелы переносятся из старого имени типа в ключевое слово `var`, чтобы сохранить вертикальные пробелы и отступы. Проще использовать `ReplaceNode`, чем `With*` для преобразования LocalDeclarationStatementSyntax, поскольку имя типа на самом деле является потомком инструкции объявления.

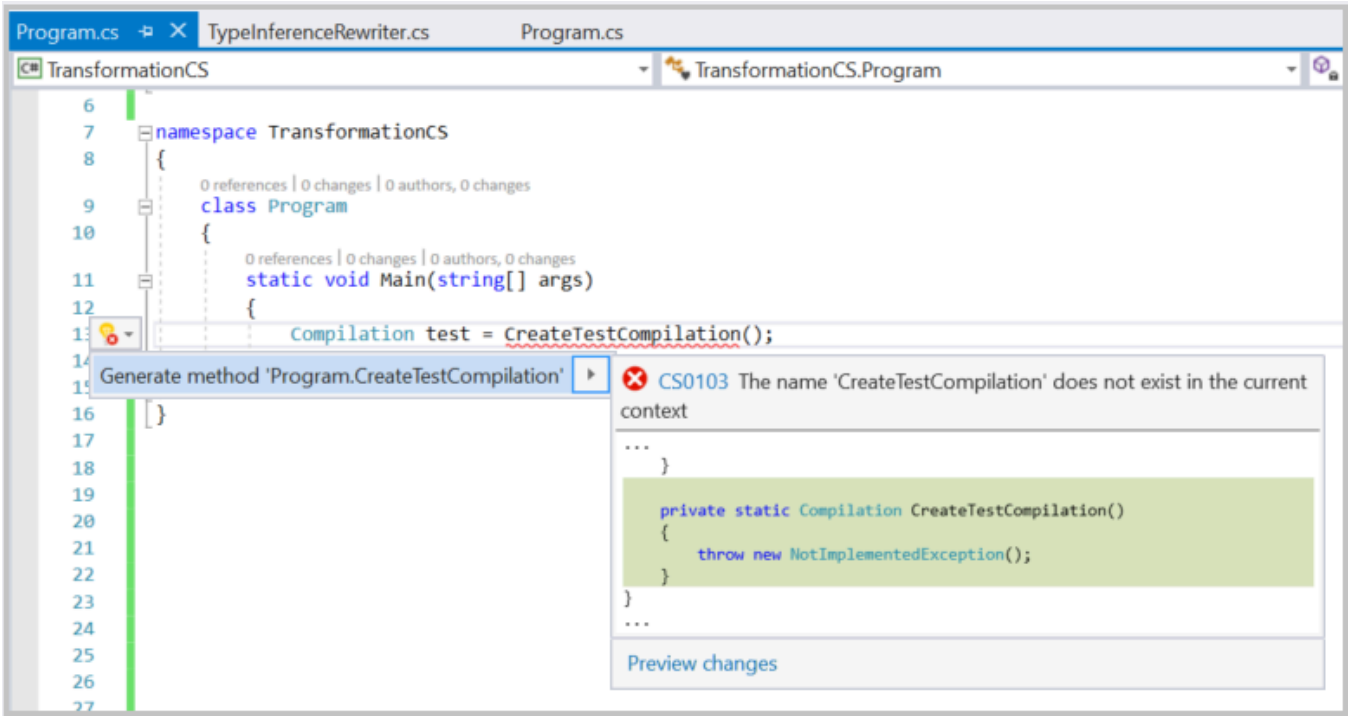
Вы закончили `TypeInferenceRewriter`. Теперь вернитесь к `Program.cs` файлу, чтобы завершить пример. Создайте тест Compilation и получите из него SemanticModel.

Используйте это SemanticModel, чтобы попробовать ваш `TypeInferenceRewriter`. Выполните этот шаг в последнюю очередь. В то же время объявите переменную заполнителя, представляющую тестовую компиляцию:

C#

```
Compilation test = CreateTestCompilation();
```

После паузы на мгновение вы должны увидеть волнистую линию ошибки, сообщающую, что метод `CreateTestCompilation` не существует. Нажмите **клавиши CTRL+Period** , чтобы открыть лампочку, а затем нажмите клавишу ВВОД, чтобы вызвать команду "**Создать заглушку метода** ". Эта команда создаст заглушку для метода `CreateTestCompilation` в классе `Program`. Вы вернетесь, чтобы заполнить этот метод позже:



Напишите следующий код, чтобы выполнить итерацию по каждому SyntaxTree в тесте Compilation. Для каждого из них инициализируйте новый `TypeInferenceRewriter` с SemanticModel, относящийся к этому дереву:

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
```

```
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

В созданной `foreach` инструкции добавьте следующий код, чтобы выполнить преобразование в каждом исходном дереве. Этот код условно записывает новое преобразованное дерево, если были сделаны какие-либо изменения. Переписыватель должен изменять дерево только в том случае, если он встречает одно или несколько объявлений локальных переменных, которые могут быть упрощены посредством вывода типов.

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Вы должны увидеть волнистые линии под кодом `File.WriteAllText`. Выберите лампочку и добавьте необходимую `using System.IO;` инструкцию.

Вы почти закончили! Остался один шаг: создание теста Compilation. Так как вы совсем не использовали вывод типов во время этого краткого руководства, это было бы идеально для теста. К сожалению, создание компиляции из файла проекта C# выходит за рамки этого пошагового руководства. Но, к счастью, если вы внимательно следили за инструкциями, есть надежда. Замените содержимое `CreateTestCompilation` метода следующим кодом. Он создает тестовую компиляцию, которая совпадает с проектом, описанным в этом кратком руководстве:

C#

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                               .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                               .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };
```

```
MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
        MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =

MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Перекрестите пальцы и запустите проект. В Visual Studio выберите ">". Visual Studio должно уведомить вас, что файлы в вашем проекте изменились. Нажмите кнопку "**Да для всех**", чтобы перезагрузить измененные файлы. Рассмотрите их, чтобы ощутить свою невероятность. Обратите внимание, насколько чище выглядит код без всех этих явных и избыточных спецификаторов типов.

Поздравляю! **Вы использовали API компилятора** для записи собственного рефакторинга, который выполняет поиск всех файлов в проекте C# для определенных синтаксических шаблонов, анализирует семантику исходного кода, соответствующего этим шаблонам, и преобразует его. Теперь вы официально автор рефакторинга!

Last updated on 26.11.2025

Руководство: Как написать ваш первый анализатор и исправление кода

Пакет SDK для платформы компилятора .NET предоставляет средства, необходимые для создания пользовательских диагностических средств (анализаторов), исправлений кода, рефакторинг кода и диагностических подавителей, предназначенных для кода C# или Visual Basic. **Анализатор** содержит код, который распознает нарушения правила. **Исправление кода** содержит код, который исправляет нарушение. Правила, которые вы реализуете, могут быть все, от структуры кода до стиля программирования до соглашений об именовании и многое другое. Платформа компилятора .NET предоставляет основу для проведения анализа в реальном времени, пока разработчики пишут код, и все функции пользовательского интерфейса Visual Studio для исправления кода: отображение зигзагов в редакторе, заполнение списка ошибок в Visual Studio, создание предложений с "лампочкой" и показ детального предварительного просмотра предлагаемых исправлений.

В этом руководстве вы узнаете о создании **анализатора** и сопутствующем **исправлении кода** с помощью API Roslyn. Анализатор — это способ выполнения анализа исходного кода и сообщения о проблеме пользователю. При необходимости можно связать исправление кода с анализатором, чтобы представить изменение исходного кода пользователя. В этом руководстве создается анализатор, который находит объявления локальных переменных, которые могли бы быть объявлены с использованием модификатора `const`, но не используются. Исправление сопутствующего кода изменяет эти объявления для добавления `const` модификатора.

Предпосылки

- Visual Studio

Необходимо установить **пакет SDK платформы компилятора .NET** с помощью установщика Visual Studio:

Инструкции по установке — Visual Studio Installer

Существует два разных способа найти **пакет SDK платформы компилятора .NET** в **установщике Visual Studio**:

Установка с помощью установщика Visual Studio — представление рабочих нагрузок

Пакет SDK платформы компилятора .NET не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как необязательный компонент.

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Проверьте рабочую нагрузку **разработки расширений Visual Studio** .
4. Откройте узел **разработки расширений Visual Studio** в дереве сводки.
5. Убедитесь, что установлен флажок **для пакета SDK для платформы компилятора .NET** .
6. Нажмите кнопку **Изменить**.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Откройте узел **отдельных компонентов** в дереве сводки.
2. Установите флажок для **редактора DGML**

Установка с помощью установщика Visual Studio — вкладка "Отдельные компоненты"

1. Запуск **установщика Visual Studio**
2. Выберите **Изменить.**
3. Выберите вкладку **"Отдельные компоненты"**
4. Установите флажок **.NET Compiler Platform SDK**. Его можно найти в верхней части раздела **"Компиляторы", "Средства сборки" и "Среды выполнения** ".
5. Нажмите кнопку **Изменить**.

Кроме того, вы также хотите, чтобы **редактор DGML** отображал графы в визуализаторе:

1. Установите флажок **для редактора DGML**. Его можно найти в разделе **"Инструменты кода** ".

Создание и проверка анализатора включает несколько шагов.

1. Создайте решение.
2. Зарегистрируйте имя и описание анализатора.
3. Предупреждения и рекомендации анализатора отчетов.
4. Реализуйте исправление кода для принятия рекомендаций.
5. Улучшение анализа с помощью модульных тестов.

Создание решения

1. В Visual Studio выберите **файл>нового>проекта или решения** , чтобы открыть диалоговое окно "Новый проект".
2. В поле поиска введите **Analyzer**, чтобы найти шаблон **Анализатор с исправлением кода (.NET Standard)**.
3. Нажмите кнопку **Далее**.
4. Присвойте проекту имя **MakeConst** и нажмите кнопку **"Создать"**.

> ⓘ **Примечание.**
>
> Может возникнуть ошибка компиляции (*MSB4062: не удалось загрузить задачу CompareBuildTaskVersion*). Чтобы устранить эту ошибку, обновите пакеты NuGet в решении с помощью диспетчера пакетов NuGet или используйте `Update-Package` в окне консоли диспетчера пакетов.

Исследуйте шаблон анализатора

Анализатор с шаблоном исправления кода создает пять проектов:

- **MakeConst**, содержащий анализатор.
- **MakeConst.CodeFixes**, содержащий исправление кода.
- **MakeConst.Package**, который используется для создания пакета NuGet для анализатора и исправления кода.
- **MakeConst.Test**, который является проектом модульного теста.
- **MakeConst.Vsix** — проект запуска по умолчанию, который открывает второй экземпляр Visual Studio с загруженным новым анализатором. Нажмите `клавишу F5` , чтобы запустить проект VSIX.

> ⓘ **Примечание.**
>
> Анализаторы должны использовать .NET Standard 2.0, так как они могут работать в среде .NET Core (сборки командной строки) и среде .NET Framework (Visual Studio).

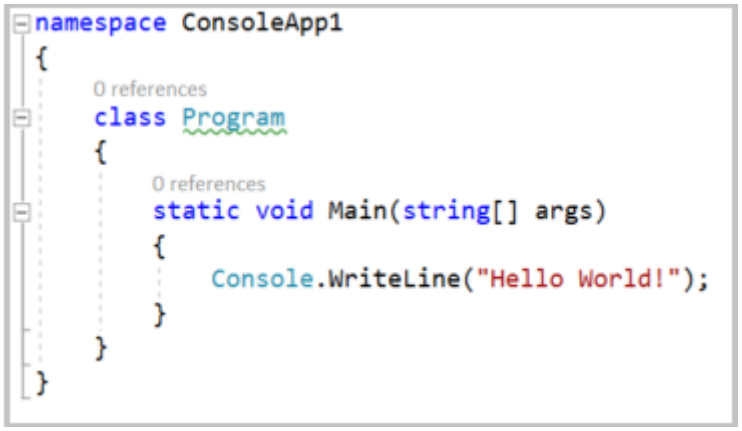
> 💡 **Совет**
>
> При запуске анализатора запускается вторая копия Visual Studio. Вторая копия использует другой куст реестра для хранения параметров. Это различие позволяет различать визуальные параметры в двух копиях Visual Studio. Вы можете выбрать

другую тему для экспериментального запуска Visual Studio. Кроме того, не перемещайте параметры или не войдите в учетную запись Visual Studio с помощью экспериментального запуска Visual Studio. Это сохраняет параметры различными.

Hive включает не только анализатор разработки, но и все анализаторы, открытые ранее. Чтобы сбросить куст Roslyn, необходимо вручную удалить его из *%LocalAppData%\Microsoft\VisualStudio*. Имя папки Roslyn hive заканчивается `Roslyn`, например `16.0_9ae182f9Roslyn`. Возможно, потребуется очистить решение и перестроить его после удаления улья.

Во втором экземпляре Visual Studio, который вы только что запустили, создайте проект консольного приложения C# (любая целевая платформа будет работать — анализаторы работают на исходном уровне).) Наведите указатель мыши на маркер с подчеркиванием волнистым цветом и отображается текст предупреждения, предоставленный анализатором.

Шаблон создает анализатор, который сообщает предупреждение о каждом объявлении типа, в котором имя типа содержит строчные буквы, как показано на следующем рисунке:



Шаблон также предоставляет исправление кода, которое изменяет любое имя типа, содержащее символы нижнего регистра на все верхние регистры. Вы можете выбрать лампочку, отображаемую с предупреждением, чтобы увидеть предложенные изменения. Принятие предлагаемых изменений обновляет имя типа и все ссылки на этот тип в решении. Теперь, когда вы видите первичный анализатор в действии, закройте второй экземпляр Visual Studio и вернитесь к вашему проекту анализатора.

Вам не нужно запускать вторую копию Visual Studio и создавать новый код для проверки каждого изменения анализатора. Шаблон также создает для вас проект для модульного тестирования. Этот проект содержит два теста. `TestMethod1` показывает типичный формат теста, который анализирует код без активации диагностики. `TestMethod2` показывает формат теста, который активирует диагностику, а затем применяет предлагаемое

исправление кода. При создании анализатора и исправления кода вы напишете тесты для различных структур кода, чтобы проверить работу. Модульные тесты для анализаторов гораздо быстрее, чем тестировать их в интерактивном режиме с помощью Visual Studio.

> **Совет**
>
> Модульные тесты анализатора — это отличный инструмент, когда вы знаете, какие конструкции кода должны и не должны активировать анализатор. Загрузка анализатора в другой копии Visual Studio — это отличный инструмент для изучения и поиска конструкций, о которых вы еще не думали.

В этом руководстве вы напишете анализатор, который сообщает пользователю любые объявления локальных переменных, которые можно преобразовать в локальные константы. Например, рассмотрим следующий код:

```C#
int x = 0;
Console.WriteLine(x);
```

В предыдущем коде `x` присваивается постоянное значение и никогда не изменяется. Его можно объявить с помощью `const` модификатора:

```C#
const int x = 0;
Console.WriteLine(x);
```

Анализ, определяющий, может ли переменная быть сделана константой, требует синтаксического анализа, анализа постоянства выражения инициализатора и анализа потока данных, чтобы убедиться, что в переменную никогда не записываются данные. Платформа компилятора .NET предоставляет API-интерфейсы, упрощающие выполнение этого анализа.

Создание регистраций анализаторов

Шаблон создает начальный `DiagnosticAnalyzer` класс в *файле MakeConstAnalyzer.cs* . В этом исходном анализаторе показаны два важных свойства каждого анализатора.

- Каждый диагностический `[DiagnosticAnalyzer]` анализатор должен предоставить атрибут, описывающий язык, на котором он работает.

- Каждый диагностический анализатор должен быть производным (напрямую или косвенно) от DiagnosticAnalyzer класса.

В шаблоне также показаны основные функции, которые являются частью любого анализатора:

1. Запись действий. Действия представляют изменения кода, которые должны активировать анализатор для проверки кода на наличие нарушений. Когда Visual Studio обнаруживает изменения кода, соответствующие зарегистрированным действиям, он вызывает зарегистрированный метод анализатора.
2. Создайте диагностику. При обнаружении нарушения анализатор создает диагностический объект, который Visual Studio использует для уведомления пользователя о нарушении.

Вы регистрируете действия в вашем переопределении метода DiagnosticAnalyzer.Initialize(AnalysisContext). В этом руководстве вы будете просматривать **узлы синтаксиса**, ища локальные объявления, и выясните, какие из них имеют константные значения. Если объявление может быть константным, ваш анализатор создает и выдает диагностическое сообщение.

Первым шагом является обновление констант регистрации и метода `Initialize`, чтобы эти константы обозначали анализатор "Make Const". Большинство строковых констант определяются в файле строковых ресурсов. Следуйте этой практике для упрощения локализации. Откройте файл *Resources.resx* для проекта анализатора **MakeConst** . Это действие отображает редактор ресурсов. Обновите строковые ресурсы следующим образом:

- Перейдите `AnalyzerDescription` на "Variables that aren't modified should be made constants.".
- Перейдите `AnalyzerMessageFormat` на "Variable '{0}' can be made constant".
- Перейдите `AnalyzerTitle` на "Variable can be made constant".

По завершении редактор ресурсов должен отображаться, как показано на следующем рисунке:

	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

Внесите оставшиеся изменения в файл анализатора. Откройте *MakeConstAnalyzer.cs* в Visual Studio. Измените зарегистрированное действие с одного, который действует на символы, на тот, который действует на синтаксисе. В методе

`MakeConstAnalyzerAnalyzer.Initialize` найдите строку, которая регистрирует действие для символов:

C#

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Замените его следующей строкой:

C#

```
context.RegisterSyntaxNodeAction(AnalyzeNode,
SyntaxKind.LocalDeclarationStatement);
```

После этого изменения можно удалить `AnalyzeSymbol` метод. Этот анализатор проверяет SyntaxKind.LocalDeclarationStatement, а не SymbolKind.NamedType выражения. Обратите внимание, что под `AnalyzeNode` находятся красные волнистые линии. Добавленный вами код ссылается на метод `AnalyzeNode`, который не объявлен. Объявите этот метод с помощью следующего кода:

C#

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Измените `Category` на "Usage" в *MakeConstAnalyzer.cs*, как показано в следующем коде.

C#

```
private const string Category = "Usage";
```

Определите локальные определения, которые могут быть константными

Теперь вы готовы написать первую версию `AnalyzeNode` метода. Следует искать одно локальное объявление, которое может быть `const`, но не является таким, как в следующем коде.

C#

```
int x = 0;
Console.WriteLine(x);
```

Первым шагом является поиск локальных объявлений. Добавьте следующий код `AnalyzeNode` в *MakeConstAnalyzer.cs*:

C#

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Этот каст всегда выполняется успешно, так как анализатор учитывает изменения локальных объявлений и только локальные объявления. Ни один другой тип узла не инициирует вызов метода `AnalyzeNode`. Затем проверьте, есть ли в объявлении какие-либо `const` модификаторы. Если их найти, вернитесь немедленно. Следующий код ищет модификаторы `const` в локальном объявлении:

C#

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Наконец, необходимо проверить, может ли переменная быть `const`. Эта проверка гарантирует, что переменная никогда не назначается после инициализации.

Выполните семантический анализ с помощью SyntaxNodeAnalysisContext. Используйте аргумент `context`, чтобы определить, можно ли объявление локальной переменной сделать `const`. A Microsoft.CodeAnalysis.SemanticModel представляет все семантические сведения в одном исходном файле. Дополнительные сведения см. в статье, которая охватывает семантические модели. Используйте Microsoft.CodeAnalysis.SemanticModel для выполнения анализа потока данных в локальном операторе объявления. Затем используйте результаты анализа потока данных, чтобы убедиться, что локальная переменная не записывается с новым значением в другом месте. GetDeclaredSymbol Вызовите член расширения, чтобы получить значение ILocalSymbol для переменной и убедитесь, что оно не содержится в коллекции DataFlowAnalysis.WrittenOutside анализа потока данных. Добавьте следующий код в конец `AnalyzeNode` метода:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
```

```
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

Добавленный код гарантирует, что переменная не изменена и поэтому может быть сделана `const`. Пришло время улучшить диагностику. Добавьте следующий код в качестве последней строки:`AnalyzeNode`

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

Вы можете проверить ход выполнения, нажав `клавишу F5` , чтобы запустить анализатор. Консольное приложение, созданное ранее, можно загрузить, а затем добавить следующий тестовый код:

C#

```
int x = 0;
Console.WriteLine(x);
```

Должна появиться лампочка, и анализатор должен сообщить о диагностике. Однако в зависимости от версии Visual Studio вы увидите следующее:

- Лампочка, которая по-прежнему использует исправление кода, сгенерированное по шаблону, сообщит вам, что код можно перевести в верхний регистр.
- Сообщение баннера в верхней части редактора, в котором говорится, что "MakeConstCodeFixProvider" обнаружил ошибку и был отключен. Эта ошибка возникает потому, что поставщик исправлений кода еще не был изменен и вместо элементов `TypeDeclarationSyntax` по-прежнему ожидает элементов `LocalDeclarationStatementSyntax`.

В следующем разделе объясняется, как написать исправление кода.

Напишите исправление кода

Анализатор может предоставлять одно или несколько исправлений кода. Исправление кода определяет изменение, которое устраняет сообщаемую проблему. Для созданного анализатора можно указать исправление кода, которое вставляет ключевое слово const:

diff

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

Пользователь выбирает его из пользовательского интерфейса лампочки в редакторе и Visual Studio изменяет код.

Откройте файл *CodeFixResources.resx* и перейдите `CodeFixTitle` на "Make constant".

Откройте файл *MakeConstCodeFixProvider.cs* , добавленный шаблоном. Это исправление кода уже подключено к идентификатору диагностики, созданному анализатором диагностики, но пока не реализует правильное преобразование кода.

Затем удалите `MakeUppercaseAsync` метод. Он больше не применяется.

Все поставщики решений для исправления кода происходят от CodeFixProvider. Все они переопределяют CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext), чтобы сообщить о доступных исправлениях кода. В `RegisterCodeFixesAsync` измените тип узла-предка, который вы ищете, на LocalDeclarationStatementSyntax, чтобы он соответствовал диагностике.

C#

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarat
ionStatementSyntax>().First();
```

Затем измените последнюю строку, чтобы зарегистрировать исправление кода. Исправление создает новый документ, который приводит к добавлению `const` модификатора в существующее объявление:

C#

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration,
c),
```

```
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

Вы заметите красные волнистые линии в коде, который вы только что добавили на символ `MakeConstAsync`. Добавьте объявление для `MakeConstAsync`, как в следующем коде:

C#

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

Ваш новый метод `MakeConstAsync` преобразует Document, представляющий исходный файл пользователя, в новый Document, который теперь содержит объявление `const`.

Вы создаете новый `const` маркер ключевого слова для вставки в начале инструкции объявления. Будьте осторожны, чтобы сначала удалить любые ведущие тривии из первого маркера инструкции объявления и прикрепить его к маркеру `const` . Добавьте следующий код в метод `MakeConstAsync`:

C#

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword, SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Далее добавьте токен `const` к объявлению с помощью следующего кода:

C#

```
// Insert the const token into the modifiers list, creating a new modifiers list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Затем отформатируйте новое объявление в соответствии с правилами форматирования C#. Форматирование изменений в соответствии с существующим кодом создает лучший

интерфейс. Добавьте следующую инструкцию сразу после существующего кода:

C#

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Для этого кода требуется новое пространство имен. Добавьте следующую `using` директиву в начало файла:

C#

```
using Microsoft.CodeAnalysis.Formatting;
```

Последний шаг — сделать редактирование. Для этого процесса необходимо выполнить три шага.

1. Получите дескриптор существующего документа.
2. Создайте новый документ, заменив существующее объявление новым объявлением.
3. Верните новый документ.

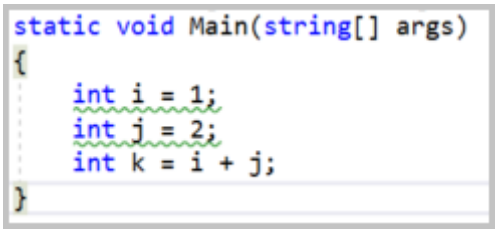
Добавьте следующий код в конец `MakeConstAsync` метода:

C#

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Исправление кода готово к тестированию. Нажмите `клавишу F5` , чтобы запустить проект анализатора во втором экземпляре Visual Studio. Во втором экземпляре Visual Studio создайте новый проект консольного приложения C# и добавьте в метод Main несколько объявлений локальных переменных, инициализированных константными значениями. Вы увидите, что они отображаются как предупреждения, как показано на следующем рисунке.



```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

Вы добились большого прогресса. Под объявлениями, которые можно сделать `const`, имеются волнистые линии. Но все еще предстоит сделать. Это работает хорошо, если вы добавляете `const` в объявления, начиная с `i`, затем `j`, и, наконец, `k`. Но если вы добавляете модификатор `const` в другом порядке, начиная с `k`, ваш анализатор создает ошибки: `k` не может быть объявлен как `const`, если `i` и `j` оба уже не являются `const`. Необходимо выполнить дополнительный анализ, чтобы обеспечить обработку различных способов объявления и инициализации переменных.

Создание модульных тестов

Ваш анализатор и исправление ошибок в коде работают над простым случаем одного объявления, которое можно объявить как константу. Многочисленные возможные операторы объявления существуют, когда эта реализация делает ошибки. В этих случаях вы будете работать с библиотекой модульных тестов, предоставленной шаблоном. Это гораздо быстрее, чем многократное открытие второй копии Visual Studio.

Откройте *файл MakeConstUnitTests.cs* в проекте модульного теста. Шаблон создал два теста, которые соответствуют двум общим шаблонам анализатора и модульного теста исправления кода. `TestMethod1` показывает шаблон теста, который гарантирует, что анализатор не сообщает о диагностике, когда она не должна. `TestMethod2` показывает шаблон для создания отчетов о диагностике и выполнении исправления кода.

Шаблон использует пакеты Microsoft.CodeAnalysis.Testing для модульного тестирования.

> **Совет**
>
> Библиотека тестирования поддерживает специальный синтаксис разметки, включая следующие:
>
> - `[|text|]`: указывается диагностическое сообщение для `text`. По умолчанию эта форма может использоваться только для тестирования анализаторов с точно одним `DiagnosticDescriptor`, предоставленным `DiagnosticAnalyzer.SupportedDiagnostics`.

- `{|ExpectedDiagnosticId:text|}`: указывает, что диагностика с параметрами Id `ExpectedDiagnosticId` сообщается для `text`.

Замените тесты шаблона в `MakeConstUnitTest` классе следующим методом теста:

C#

```
        [TestMethod]
        public async Task LocalIntCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

Выполните этот тест, чтобы убедиться, что он проходит. В Visual Studio откройте **обозреватель тестов**, выбрав "**Проверить**>**обозреватель тестов**>". Затем нажмите **кнопку "Выполнить все**".

Создайте тесты для проверки допустимых объявлений

Как правило, анализаторы должны выйти как можно быстрее, выполняя минимальную работу. Visual Studio вызывает зарегистрированные анализаторы, так как пользователь изменяет код. Скорость реагирования является ключевым требованием. Для кода существует несколько тестовых вариантов, которые не должны вызывать диагностику.

Анализатор уже обрабатывает несколько этих тестов. Добавьте следующие методы тестирования для представления этих случаев:

C#

```
        [TestMethod]
        public async Task VariableIsAssigned_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
        }
```

C#

```
        [TestMethod]
        public async Task VariableIsAlreadyConst_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

C#

```
        [TestMethod]
        public async Task NoInitializer_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
```

```
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

Эти тесты проходят, так как анализатор уже обрабатывает следующие условия:

- Переменные, назначенные после инициализации, обнаруживаются анализом потока данных.
- Объявления, которые уже `const`, отфильтровываются на основе проверки наличия ключевого слова `const`.
- Объявления без инициализатора обрабатываются анализом потока данных, который обнаруживает назначения за пределами объявления.

Затем добавьте методы тестирования для условий, которые еще не обработаны:

- Объявления, в которых инициализатор не является константой, так как они не могут быть константами во время компиляции:

  C#

```
        [TestMethod]
        public async Task InitializerIsNotConstant_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
        }
```

Это может быть еще сложнее, так как C# разрешает несколько объявлений в виде одной инструкции. Рассмотрим следующую строковую константу тестового случая:

C#

```
        [TestMethod]
        public async Task MultipleInitializers_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
        }
```

Переменная `i` может быть константой, но переменная `j` не может. Поэтому это выражение не может быть объявлением константы.

Снова запустите тесты, и вы увидите, что последние два тестовых случая завершаются сбоем.

Обновите анализатор, чтобы пропускать правильные объявления

Необходимо улучшить метод анализатора `AnalyzeNode` , чтобы отфильтровать код, соответствующий этим условиям. Эти условия связаны, поэтому аналогичные изменения исправляют все эти условия. Внесите следующие изменения в `AnalyzeNode`:

- Проверка семантики анализировала одно объявление переменной. Этот код должен находиться в цикле `foreach` , который проверяет все переменные, объявленные в одной инструкции.
- Каждая объявленная переменная должна иметь инициализатор.
- Инициализатор каждой объявленной переменной должен быть константой во время компиляции.

`AnalyzeNode` В методе замените исходный семантический анализ:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);
```

```
// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

с помощью следующего фрагмента кода:

C#

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis region.
    ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

Первый `foreach` цикл проверяет каждое объявление переменной с помощью синтаксического анализа. Первая проверка гарантирует, что переменная имеет инициализатор. Вторая проверка гарантирует, что инициализатор является константой. Второй цикл имеет исходный семантический анализ. Семантические проверки находятся в отдельном цикле, так как они оказывают большее влияние на производительность. Снова запустите тесты, и вы увидите, что они все пройдут.

Добавьте финальные штрихи

Осталось совсем немного. Существует несколько дополнительных условий для обработки анализатора. Visual Studio вызывает анализаторы во время написания кода пользователем. Часто бывает, что анализатор вызывается для кода, который не компилируется. Метод диагностического анализатора `AnalyzeNode` не проверяет, преобразуется ли константное значение в тип переменной. Таким образом, текущая реализация без проблем трансформирует ошибочное объявление, например, `int i = "abc"`, в локальную константу. Добавьте метод теста для этого случая:

C#

```
        [TestMethod]
        public async Task DeclarationIsInvalid_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
}
");
        }
```

Кроме того, ссылочные типы не обрабатываются должным образом. Единственное значение константы, допустимое для ссылочного типа `null`, за исключением случаев System.String, в котором разрешены строковые литералы. Иными словами, `const string s = "abc"` является законным, но `const object s = "abc"` не является. Этот фрагмент кода проверяет условие.

C#

```
        [TestMethod]
        public async Task DeclarationIsNotString_NoDiagnostic()
```

```
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
        }
```

Чтобы быть тщательным, необходимо добавить еще один тест, чтобы убедиться, что можно создать константу для строки. Следующий фрагмент кода определяет код, который вызывает диагностику, и код после применения исправления:

C#

```
        [TestMethod]
        public async Task StringCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
        }
```

Наконец, если переменная объявлена с `var` ключевым словом, исправление кода делает не такую вещь и создает `const var` объявление, которое язык C# не поддерживает. Чтобы устранить эту ошибку, исправление кода должно заменить `var` ключевое слово именем выводимого типа:

C#

```
        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
        }

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");
        }
```

К счастью, вы можете устранить все перечисленные выше ошибки, используя те же методы, которые вы только что узнали.

Чтобы устранить первую ошибку, откройте *MakeConstAnalyzer.cs* и найдите `foreach` цикл, где проверяется каждый из инициализаторов локального объявления, чтобы убедиться, что они назначены константным значениям. Непосредственно *перед* первым `foreach` циклом вызовите метод `context.SemanticModel.GetTypeInfo()` для получения подробных сведений о типе локального объявления:

C#

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType = context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;
```

Затем проверяйте каждый инициализатор в цикле `foreach`, чтобы удостовериться, что он преобразуется в тип переменной. Добавьте следующую проверку после того, как убедитесь, что инициализатор является константой:

C#

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion = context.SemanticModel.ClassifyConversion(initializer.Value,
variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

Следующее изменение основывается на последнем. Перед закрывающей фигурной скобкой первого `foreach` цикла добавьте следующий код, чтобы проверить тип локальной переменной, если значение константы – строка или null.

C#

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
```

```
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

Чтобы заменить ключевое слово `var` на правильное имя типа, необходимо немного доработать код в вашей функции исправления кода. Вернитесь к *MakeConstCodeFixProvider.cs*. Добавленный код выполняет следующие действия.

- Проверьте, является ли объявление объявлением `var` , и если это:
- Создайте новый тип для выведенного типа.
- Убедитесь, что объявление типа не является псевдотипом. Если да, должным образом декларируйте `const var`.
- Убедитесь, что `var` это не имя типа в этой программе. (Если да, `const var` является законным).
- Упростите полное имя типа

Это звучит как много кода. Это не так. Замените строку, которая объявляет и инициализирует `newLocal` следующим кодом. Он идет сразу после инициализации `newModifiers`:

C#

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration = localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName =
```

```
SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal.WithModifiers(newModifiers)
                            .WithDeclaration(variableDeclaration);
```

Чтобы использовать `using` тип, необходимо добавить одну Simplifier директиву:

C#

```
using Microsoft.CodeAnalysis.Simplification;
```

Запустите тесты, и все они должны пройти. Поздравьте себя, запустив готовый анализатор. Нажмите `клавиши CTRL` + `F5` , чтобы запустить проект анализатора во втором экземпляре Visual Studio с загруженным расширением Roslyn Preview.

- Во втором экземпляре Visual Studio создайте проект консольного приложения C# и добавьте `int x = "abc";` его в метод Main. Благодаря первому исправлению ошибки, предупреждение не отображается для этого объявления локальной переменной (хотя ожидаемая ошибка компилятора всё ещё присутствует).
- Затем добавьте `object s = "abc";` в метод Main. Из-за второго исправления ошибок предупреждение не сообщается.
- Наконец, добавьте другую локальную переменную, которая использует ключевое `var` слово. Вы видите, что сообщается предупреждение, а рекомендация появляется снизу слева.
- Переместите курсор редактора на волнистое подчеркивание и нажмите клавишу `Ctrl` + `.` . для отображения предлагаемого исправления кода. После выбора исправления для кода обратите внимание, что теперь ключевое слово `var` обрабатывается правильно.

Наконец, добавьте следующий код:

C#

```
int i = 2;
int j = 32;
int k = i + j;
```

После этих изменений красные волнистые линии отображаются только в первых двух переменных. Добавьте `const` в `i` и `j`, и вы получите новое предупреждение на `k`, так как теперь оно может быть `const`.

Поздравляю! Вы создали ваше первое расширение платформы компилятора .NET, которое выполняет анализ кода в реальном времени для обнаружения проблемы и предоставляет быстрое решение. На этом этапе вы узнали многие API кода, которые являются частью пакета SDK платформы компилятора .NET (API Roslyn). Вы можете проверить работу над завершенным примером в репозитории примеров GitHub.

Другие ресурсы

- Начало работы с анализом синтаксиса
- Начало работы с семантическим анализом

Last updated on 17.02.2026

Основные понятия программирования (C#)

10.10.2025

В этом разделе описываются основные понятия программирования на языке C#.

В этом разделе

Развернуть таблицу

Название	Описание
Ковариантность и контравариантность (C#)	Показывает, как включить неявное преобразование параметров универсального типа в интерфейсах и делегатах.
Итераторы (C#)	Описывает итераторы, которые используются для пошагового выполнения коллекций и возврата элементов одновременно.

Связанные разделы

- Советы по производительности

  Описывает несколько основных правил, которые помогут повысить производительность приложения.

Ковариантность и контравариантность (C#)

17.06.2025

В C#ковариация и контравариантность обеспечивают неявное преобразование ссылок для типов массивов, типов делегатов и аргументов универсального типа. Ковариация сохраняет совместимость назначений и контравариантность изменяет ее.

В следующем коде показано различие между совместимостью назначений, ковариантностью и контравариантностью.

C#

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived
type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Ковариация для массивов обеспечивает неявное преобразование массива более производного типа в массив менее производного типа. Но эта операция не является безопасной, как показано в следующем примере кода.

C#

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

Поддержка ковариации и контравариации для групп методов позволяет сопоставлять сигнатуры методов с типами делегатов. Это позволяет назначать делегатам не только методы, имеющие соответствующие сигнатуры, но и методы, возвращающие более производные типы (ковариантность) или принимающие параметры, имеющие менее производные типы (контравариантность), чем указанные типом делегата. Дополнительные сведения см. в разделах "Вариативность в делегатах (C#)" и "Использование вариативности в делегатах (C#)".

В следующем примере кода показана поддержка ковариации и контравариации для групп методов.

C#

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

В .NET Framework 4 и более поздних версиях C# поддерживает ковариацию и контравариацию в универсальных интерфейсах и делегатах и позволяет неявное преобразование параметров универсального типа. Для получения дополнительной информации см. раздел "Вариативность в универсальных интерфейсах (C#)" и "Вариативность в делегатах (C#)".

В следующем примере кода показано неявное преобразование ссылок для универсальных интерфейсов.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Универсальный интерфейс или делегат называется *вариантом* , если его универсальные параметры объявлены ковариантными или контравариантными. C# позволяет создавать

собственные вариантные интерфейсы и делегаты. Для получения дополнительной информации см. раздел Создание вариантных универсальных интерфейсов (C#) и Вариантность в делегатах (C#).

Связанные разделы

Развернуть таблицу

Название	Описание
Вариативность в универсальных интерфейсах (C#)	Обсуждает ковариантность и контравариантность в универсальных интерфейсах и предоставляет список вариантов универсальных интерфейсов в .NET.
Создание вариантных универсальных интерфейсов (C#)	Показывает, как создавать интерфейсы пользовательских вариантов.
Использование дисперсии в интерфейсах для универсальных коллекций (C#)	Показывает, как поддержка ковариации и контравариации в IEnumerable<T>IComparable<T> интерфейсах помогает повторно использовать код.
Вариативность в делегатах (C#)	Обсуждает ковариантность и контравариантность в универсальных и не универсальных делегатах и предоставляет список вариантов универсальных делегатов в .NET.
Использование дисперсии в делегатах (C#)	Показывает, как использовать поддержку ковариации и контравариации в необобщённых делегатах для соответствия подписей методов типам делегатов.
Использование вариативности для func и универсальных делегатов действий (C#)	Показывает, как поддержка ковариации и контравариации в `Func` и `Action` делегатах может помочь повторно использовать код.

Вариативность в универсальных интерфейсах (C#)

17.06.2025

Платформа .NET Framework 4 представила поддержку дисперсии для нескольких существующих универсальных интерфейсов. Поддержка ковариантности обеспечивает автоматическое преобразование классов, реализующих эти интерфейсы.

Начиная с .NET Framework 4, следующие интерфейсы являются вариантами:

- IEnumerable<T> (T является ковариантным)
- IEnumerator<T> (T является ковариантным)
- IQueryable<T> (T является ковариантным)
- IGrouping<TKey,TElement> (`TKey` и `TElement` ковариантные)
- IComparer<T> (T является контравариантным)
- IEqualityComparer<T> (T является контравариантным)
- IComparable<T> (T является контравариантным)

Начиная с .NET Framework 4.5, следующие интерфейсы являются вариантами:

- IReadOnlyList<T> (T является ковариантным)
- IReadOnlyCollection<T> (T является ковариантным)

Ковариация позволяет методу иметь более производный тип возвращаемого значения, чем определенный параметром универсального типа интерфейса. Чтобы проиллюстрировать функцию ковариации, рассмотрим следующие универсальные интерфейсы: `IEnumerable<Object>` и `IEnumerable<String>`. Интерфейс `IEnumerable<String>` не наследует `IEnumerable<Object>` интерфейс. Однако тип `String` наследует `Object` тип, и в некоторых случаях может потребоваться назначить объекты этих интерфейсов друг другу. Это показано в следующем примере кода.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

В более ранних версиях .NET Framework этот код вызывает ошибку компиляции в C# и, если `Option Strict` он включен, в Visual Basic. Но теперь можно использовать `strings` вместо `objects`, как показано в предыдущем примере, потому что интерфейс IEnumerable<T> является ковариантным.

Контравариантность позволяет методу иметь типы аргументов, которые являются менее производными, чем указанные универсальным параметром интерфейса. Чтобы проиллюстрировать контравариантность, предположим, что вы создали `BaseComparer` класс для сравнения экземпляров `BaseClass` класса. Класс `BaseComparer` реализует интерфейс `IEqualityComparer<BaseClass>`. IEqualityComparer<T> Так как интерфейс теперь является контравариантным, можно использовать `BaseComparer` для сравнения экземпляров классов, наследующих `BaseClass` класс. Это показано в следующем примере кода.

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Дополнительные примеры см. в разделе "Использование дисперсии в интерфейсах для универсальных коллекций (C#)".

Ковариантность в универсальных интерфейсах поддерживается только для ссылочных типов. Типы значений не поддерживают дисперсию. Например, `IEnumerable<int>` нельзя неявно преобразовать `IEnumerable<object>` в , так как целые числа представлены типом значения.

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

Также важно помнить, что классы, реализующие интерфейсы вариантов, по-прежнему инвариантны. Например, хотя List<T> реализует ковариантный интерфейс IEnumerable<T>, нельзя неявно преобразовать `List<String>` в `List<Object>`. Это показано в следующем примере кода.

C#

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

См. также

- Использование дисперсии в интерфейсах для универсальных коллекций (C#)
- Создание вариантных универсальных интерфейсов (C#)
- Универсальные интерфейсы
- Вариативность в делегатах (C#)

Создание вариативных универсальных интерфейсов (C#)

17.06.2025

Параметры универсального типа можно объявить в интерфейсах как ковариантный или контравариантный. *Ковариация* позволяет методам интерфейса иметь более производные типы возвращаемых значений, чем определенные параметрами универсального типа. *Контравариантность* позволяет методам интерфейса иметь типы аргументов, которые являются менее производными, чем указанные универсальными параметрами. Универсальный интерфейс с ковариантными или контравариантными параметрами универсального типа называется *вариантом*.

> ⓘ **Примечание**
>
> Платформа .NET Framework 4 представила поддержку дисперсии для нескольких существующих универсальных интерфейсов. Список вариантов интерфейсов в .NET см. в разделе "Вариативность" в универсальных интерфейсах (C#).

Объявление универсальных интерфейсов variant

Вы можете объявлять ковариантные и контравариантные универсальные интерфейсы с помощью ключевых слов `in` и `out` для параметров универсального типа.

> ⓘ **Важно!**
>
> `ref`, `in` и `out` параметры в C# не могут быть вариантами. Типы значений также не поддерживают дисперсию.

Параметр универсального типа можно объявить ковариантным с помощью ключевого `out` слова. Ковариантный тип должен соответствовать следующим условиям:

- Тип используется только в качестве возвращаемого типа методов интерфейса и не используется в качестве типа аргументов метода. Это показано в следующем примере, в котором тип `R` объявлен ковариантным.

  ```
  C#
  ```

```
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
    // void SetSomething(R sampleArg);

}
```

Существует одно исключение из этого правила. Если у вас есть контравариантный универсальный делегат в качестве параметра метода, вы можете использовать этот тип в качестве параметра универсального типа для делегата. Это иллюстрируется типом `R` в следующем примере. Дополнительные сведения см. в разделе "Вариативность в делегатах (C#)" и "Использование вариативности для Func и Action в обобщенных делегатах (C#)".

C#

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- Тип не используется в качестве универсального ограничения для методов интерфейса. Это показано в следующем коде.

C#

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Параметр универсального типа можно объявить контравариантным с помощью ключевого `in` слова. Контравариантный тип можно использовать только в качестве типа аргументов метода, а не в качестве возвращаемого типа методов интерфейса. Контравариантный тип также можно использовать для универсальных ограничений. В следующем коде показано, как объявить контрвариантный интерфейс и использовать универсальное ограничение для одного из его методов.

C#

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

Кроме того, можно поддерживать ковариацию и контравариантность в одном интерфейсе, но для различных параметров типа, как показано в следующем примере кода.

C#

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

реализация универсальных интерфейсов с вариантными параметрами

Вы реализуете вариантные универсальные интерфейсы в классах с помощью того же синтаксиса, который используется для инвариантных интерфейсов. В следующем примере кода показано, как реализовать ковариантный интерфейс в универсальном классе.

C#

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Классы, реализующие вариантные интерфейсы, являются инвариантными. Например, рассмотрим следующий код.

C#

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Расширение универсальных интерфейсов с вариативностями

При расширении универсального интерфейса необходимо использовать `in` ключевые слова и `out` ключевые слова, чтобы явно указать, поддерживает ли производный интерфейс дисперсию. Компилятор не выводит дисперсию из расширенного интерфейса. Например, рассмотрим следующие интерфейсы.

C#

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

В интерфейсе `IInvariant<T>` параметр `T` универсального типа является инвариантным, в то время как в `IExtCovariant<out T>` параметре типа является ковариантным, хотя оба интерфейса расширяют один и тот же интерфейс. Это же правило применяется к параметрам контравариантного универсального типа.

Можно создать интерфейс, который расширяет и интерфейс, в котором параметр универсального типа `T` является ковариантным, и интерфейс, где он является контравариантным, если в интерфейсе расширения параметр универсального типа `T` является инвариантным. Это показано в следующем примере кода.

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

Однако если параметр `T` универсального типа объявлен ковариантным в одном интерфейсе, его нельзя объявить контравариантным в интерфейсе расширения или наоборот. Это показано в следующем примере кода.

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Избегание неоднозначности

При реализации вариантных универсальных интерфейсов вариативность иногда может привести к неоднозначности. Следует избежать такой неоднозначности.

Например, если вы явно реализуете один и тот же вариант универсального интерфейса с разными параметрами универсального типа в одном классе, это может создать неоднозначность. Компилятор не создает ошибку в этом случае, но не указывается, какая реализация интерфейса будет выбрана во время выполнения. Эта неоднозначность может привести к тонким ошибкам в коде. Рассмотрим следующий пример кода.

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
```

```
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

В этом примере не указано, как метод `pets.GetEnumerator` делает выбор между `Cat` и `Dog`. Это может привести к проблемам в коде.

См. также

- Вариативность в универсальных интерфейсах (C#)
- Использование вариативности для func и универсальных делегатов действий (C#)

Использование ковариантности в интерфейсах для универсальных коллекций (C#)

17.06.2025

Ковариантный интерфейс позволяет своим методам возвращать более производные типы, чем указанные в интерфейсе. Контравариантный интерфейс позволяет своим методам принимать параметры менее производных типов, чем указанные в интерфейсе.

В .NET Framework 4 несколько существующих интерфейсов стали ковариантными и контравариантными. В их числе IEnumerable<T> и IComparable<T>. Это позволяет повторно использовать методы, работающие с универсальными коллекциями базовых типов для коллекций производных типов.

Список вариантов интерфейсов в .NET см. в разделе "Вариативность" в универсальных интерфейсах (C#).

Преобразование универсальных коллекций

В следующем примере показаны преимущества поддержки ковариации в интерфейсе IEnumerable<T> . Метод `PrintFullName` принимает коллекцию `IEnumerable<Person>` типа в качестве параметра. Однако его можно повторно использовать для коллекции `IEnumerable<Employee>` типа, так как `Employee` наследуется `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
```

```
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Сравнение универсальных коллекций

В следующем примере показаны преимущества поддержки контравариантности в интерфейсе IEqualityComparer<T> . Класс `PersonComparer` реализует интерфейс `IEqualityComparer<Person>`. Однако этот класс можно повторно использовать для сравнения последовательности объектов `Employee` типа, так как `Employee` наследуется `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
```

```
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName = "Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

См. также

- Вариативность в универсальных интерфейсах (C#)

Вариативность делегатов (C#)

17.06.2025

Платформа .NET Framework 3.5 представила поддержку дисперсии для сопоставления подписей методов с типами делегатов во всех делегатах в C#. Это означает, что можно назначить делегатам не только методы, имеющие соответствующие сигнатуры, но и методы, возвращающие более производные типы (ковариантность) или принимающие параметры, имеющие менее производные типы (контравариантность), чем указанные типом делегата. Это включает как универсальные, так и не универсальные делегаты.

Например, рассмотрим следующий код, имеющий два класса и два делегата: универсальный и не универсальный.

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

При создании делегатов `SampleDelegate` или `SampleGenericDelegate<A, R>` типов можно назначить один из следующих методов этим делегатам.

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

В следующем примере кода показано неявное преобразование между сигнатурой метода и типом делегата.

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Дополнительные примеры см. в разделе "Использование вариативности в делегатах (C#)" и "Использование вариативности для Func и Action Generic Delegates (C#)".

Дисперсия в параметрах универсального типа

В .NET Framework 4 или более поздней версии можно включить неявное преобразование между делегатами. Универсальные делегаты, имеющие различные типы, указанные универсальными параметрами типа, могут быть назначены друг другу, если типы наследуются друг от друга, как требуется для вариативности.

Чтобы включить неявное преобразование, необходимо явно объявить универсальные параметры в делегате как ковариантные или контравариантные с помощью ключевого слова `in` или `out`.

В следующем примере кода показано, как создать делегат с ковариантным параметром универсального типа.

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

Если вы используете только поддержку дисперсии для сопоставления подписей методов с типами делегатов и не используете ключевые слова `in` и `out`, то можете обнаружить, что иногда возможно создать экземпляры делегатов с идентичными лямбда-выражениями или методами, но нельзя присвоить один делегат другому.

В следующем примере кода `SampleGenericDelegate<String>` невозможно явно преобразовать в `SampleGenericDelegate<Object>`, хотя `String` наследует `Object`. Эту проблему можно устранить, пометив универсальный параметр `T` ключевым словом `out` .

C#

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Универсальные делегаты с параметрами типа вариантов в .NET

Платформа .NET Framework 4 ввела поддержку для вариативности параметров обобщённого типа в нескольких уже существующих обобщённых делегатах.

- `Action` делегаты из пространства имен System, например, Action<T> и Action<T1,T2>
- `Func` делегаты из пространства имен System, например, Func<TResult> и Func<T,TResult>
- Делегат Predicate<T>
- Делегат Comparison<T>
- Делегат Converter<TInput,TOutput>

Дополнительные сведения и примеры см. в разделе "Использование вариативности" для func и action generic делегатов (C#).

Объявление параметров типа ковариантных и контрвариантных в обобщённых делегатах

Если универсальный делегат имеет ковариантные или контравариантные параметры универсального типа, он может называться *вариантным универсальным делегатом*.

Параметр универсального типа можно объявить ковариантным в универсальном делегате с помощью ключевого `out` слова. Ковариантный тип можно использовать только в качестве типа возвращаемого метода, а не в качестве типа аргументов метода. В следующем примере кода показано, как объявить ковариантный универсальный делегат.

C#

```
public delegate R DCovariant<out R>();
```

Параметр универсального типа можно объявить контравариантным в универсальном делегате с помощью ключевого слова `in`. Контравариантный тип можно использовать только в качестве типа аргументов метода, а не в качестве возвращаемого типа метода. В следующем примере кода показано, как объявить контравариантный универсальный делегат.

C#

```
public delegate void DContravariant<in A>(A a);
```

> ⓘ **Важно!**
>
> `ref`, `in` и `out` параметры в C# не могут быть помечены как вариант.

Кроме того, можно поддерживать дисперсию и ковариацию в одном делегате, но для разных параметров типов. Это показано в следующем примере.

C#

```
public delegate R DVariant<in A, out R>(A a);
```

Создание экземпляров и вызов вариантных универсальных делегатов.

Вы можете создавать экземпляры вариантных делегатов и вызывать их точно так же, как создавать и вызывать экземпляры инвариантных делегатов. В следующем примере делегат создается при помощи лямбда-выражения.

C#

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Объединение обобщённых делегатов с вариативностью

Не объединяйте делегатов с вариантностью. Метод Combine не поддерживает преобразование вариантов делегатов и ожидает, что делегаты будут иметь точно тот же тип. Это может привести к исключению во время выполнения при объединении делегатов с помощью Combine метода или с помощью `+` оператора, как показано в следующем примере кода.

C#

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Дисперсия в параметрах универсального типа для типов значений и ссылочных типов

Вариативность параметров универсального типа поддерживается только для ссылочных типов. Например, `DVariant<int>` нельзя неявно преобразовать в `DVariant<Object>` или `DVariant<long>`, так как целое число является типом значения.

В следующем примере показано, что дисперсии в параметрах универсального типа не поддерживаются для типов значений.

C#

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

См. также

- Обобщения
- Использование вариативности для func и универсальных делегатов действий (C#)
- Практическое руководство. Объединение делегатов (многоадресные делегаты)

Использование дисперсии в делегатах (C#)

03.07.2025

При назначении метода делегату *ковариация* и *контравариантность* обеспечивают гибкость сопоставления типа делегата с сигнатурой метода. Ковариация позволяет методу иметь тип возвращаемого значения, который является более производным, чем определенный в делегате. Контравариантность допускает использование метода, параметры которого имеют типы, менее производные, чем типы в делегате.

Пример 1. Ковариация

Описание

В этом примере показано, как делегаты могут использоваться с методами, имеющими типы возвращаемых данных, производные от возвращаемого типа в сигнатуре делегата. Тип данных, возвращаемый `DogsHandler`, имеет тип `Dogs`, который является производным от типа `Mammals`, определенного в делегате.

Код

C#

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;
```

```
        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Пример 2. Контравариантность

Описание

В этом примере показано, как делегаты могут использоваться с методами, имеющими параметры, типы которых являются базовыми типами типа параметра подписи делегата. С помощью контравариантности можно использовать один обработчик событий вместо отдельных обработчиков. В следующем примере используется два делегата:

- Настраиваемый `KeyEventHandler` делегат, определяющий сигнатуру ключевого события. Его подпись:

  C#

  ```
  public delegate void KeyEventHandler(object sender, KeyEventArgs e)
  ```

- Настраиваемый `MouseEventHandler` делегат, определяющий сигнатуру события мыши. Его подпись:

  C#

  ```
  public delegate void MouseEventHandler(object sender, MouseEventArgs e)
  ```

В примере определяется обработчик событий с параметром EventArgs и используется для обработки событий ключа и мыши. Это работает, так как EventArgs это базовый тип пользовательских `KeyEventArgs` и `MouseEventArgs` классов, определенных в примере. Контравариантность позволяет методу, который принимает параметр базового типа, быть использованным для событий, предоставляющих параметры производного типа.

Как работает контравариантность в этом примере

При подписке на событие компилятор проверяет, совместим ли метод обработчика событий с подписью делегата события. С контравариантностью:

1. Событие `KeyDown` ожидает метод, который принимает `KeyEventArgs`.

2. Событие `MouseClick` ожидает метод, который принимает `MouseEventArgs`.
3. Метод `MultiHandler` принимает базовый тип `EventArgs`.
4. Поскольку `KeyEventArgs` и `MouseEventArgs` оба наследуют от `EventArgs`, их можно безопасно передать в метод, ожидающий `EventArgs`.
5. Компилятор разрешает это назначение, так как оно безопасно — `MultiHandler` может работать с любым `EventArgs` экземпляром.

Это контравариантность в действии: можно использовать метод с параметром "менее конкретный" (базовый тип), где ожидается параметр "более конкретный" (производный тип).

Код

C#

```
// Custom EventArgs classes to demonstrate the hierarchy
public class KeyEventArgs(string keyCode) : EventArgs
{
    public string KeyCode { get; set; } = keyCode;
}

public class MouseEventArgs(int x, int y) : EventArgs
{
    public int X { get; set; } = x;
    public int Y { get; set; } = y;
}

// Define delegate types that match the Windows Forms pattern
public delegate void KeyEventHandler(object sender, KeyEventArgs e);
public delegate void MouseEventHandler(object sender, MouseEventArgs e);

// A simple class that demonstrates contravariance with events
public class Button
{
    // Events that expect specific EventArgs-derived types
    public event KeyEventHandler? KeyDown;
    public event MouseEventHandler? MouseClick;

    // Method to simulate key press
    public void SimulateKeyPress(string key)
    {
        Console.WriteLine($"Simulating key press: {key}");
        KeyDown?.Invoke(this, new KeyEventArgs(key));
    }

    // Method to simulate mouse click
    public void SimulateMouseClick(int x, int y)
    {
        Console.WriteLine($"Simulating mouse click at ({x}, {y})");
```

```
        MouseClick?.Invoke(this, new MouseEventArgs(x, y));
    }
}

public class Form1
{
    private Button button1;

    public Form1()
    {
        button1 = new Button();

        // Event handler that accepts a parameter of the base EventArgs type.
        // This method can handle events that expect more specific EventArgs-
derived types
        // due to contravariance in delegate parameters.

        // You can use a method that has an EventArgs parameter,
        // although the KeyDown event expects the KeyEventArgs parameter.
        button1.KeyDown += MultiHandler;

        // You can use the same method for an event that expects
        // the MouseEventArgs parameter.
        button1.MouseClick += MultiHandler;
    }

    // Event handler that accepts a parameter of the base EventArgs type.
    // This works for both KeyDown and MouseClick events because:
    // - KeyDown expects KeyEventHandler(object sender, KeyEventArgs e)
    // - MouseClick expects MouseEventHandler(object sender, MouseEventArgs e)
    // - Both KeyEventArgs and MouseEventArgs derive from EventArgs
    // - Contravariance allows a method with a base type parameter (EventArgs)
    //   to be used where a derived type parameter is expected
    private void MultiHandler(object sender, EventArgs e)
    {
        Console.WriteLine($"MultiHandler called at: {DateTime.Now:HH:mm:ss.fff}");

        // You can check the actual type of the event args if needed
        switch (e)
        {
            case KeyEventArgs keyArgs:
                Console.WriteLine($"  - Key event: {keyArgs.KeyCode}");
                break;
            case MouseEventArgs mouseArgs:
                Console.WriteLine($"  - Mouse event: ({mouseArgs.X},
{mouseArgs.Y})");
                break;
            default:
                Console.WriteLine($"  - Generic event: {e.GetType().Name}");
                break;
        }
    }

    public void DemonstrateEvents()
    {
```

```
        Console.WriteLine("Demonstrating contravariance in event handlers:");
        Console.WriteLine("Same MultiHandler method handles both events!\n");

        button1.SimulateKeyPress("Enter");
        button1.SimulateMouseClick(100, 200);
        button1.SimulateKeyPress("Escape");
        button1.SimulateMouseClick(50, 75);
    }
}
```

Ключевые моменты о контравариантности

C#

```
// Demonstration of how contravariance works with delegates:
//
// 1. KeyDown event signature: KeyEventHandler(object sender, KeyEventArgs e)
//    where KeyEventArgs derives from EventArgs
//
// 2. MouseClick event signature: MouseEventHandler(object sender, MouseEventArgs
e)
//    where MouseEventArgs derives from EventArgs
//
// 3. Our MultiHandler method signature: MultiHandler(object sender, EventArgs e)
//
// 4. Contravariance allows us to use MultiHandler (which expects EventArgs)
//    for events that provide more specific types (KeyEventArgs, MouseEventArgs)
//    because the more specific types can be safely treated as their base type.
//
// This is safe because:
// - The MultiHandler only uses members available on the base EventArgs type
// - KeyEventArgs and MouseEventArgs can be implicitly converted to EventArgs
// - The compiler knows that any EventArgs-derived type can be passed safely
```

При запуске этого примера вы увидите, что один и тот же `MultiHandler` метод успешно обрабатывает события ключа и мыши, демонстрируя, как контравариантность обеспечивает более гибкий и многократно используемый код обработки событий.

См. также

- Вариативность в делегатах (C#)
- Использование вариативности для func и универсальных делегатов действий (C#)

Использование вариативности для универсальных делегатов Func и Action (C#)

07.08.2025

В этих примерах показано, как использовать ковариацию и контравариацию в `Func` и `Action` универсальных делегатах, чтобы включить повторное использование методов и обеспечить большую гибкость в коде.

Дополнительные сведения о ковариации и контравариации см. в статье "Вариативность делегатов (C#)."

Использование делегатов с ковариантными параметрами типа

В следующем примере показаны преимущества поддержки ковариации в обобщённых `Func` делегатах. Метод `FindByTitle` принимает параметр `String` типа и возвращает объект `Employee` типа. Однако этот метод можно назначить делегату `Func<String, Person>` , так как `Employee` наследует `Person`.

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
```

```
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Использование делегатов с контравариантными параметрами типа

В следующем примере показаны преимущества поддержки контравариантности в универсальных `Action` делегатах. Метод `AddToContacts` принимает параметр `Person` типа. Однако этот метод можно назначить делегату `Action<Employee>` , так как `Employee` наследует `Person`.

C#

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

Контравариантность и анонимные функции

При работе с анонимными функциями (лямбда-выражениями) может возникнуть контринтуитивное поведение, связанное с контравариантностью. Рассмотрим следующий пример:

C#

```
public class Person
{
    public virtual void ReadContact() { /*...*/ }
}

public class Employee : Person
{
    public override void ReadContact() { /*...*/ }
}

class Program
{
    private static void Main()
    {
        var personReadContact = (Person p) => p.ReadContact();

        // This works - contravariance allows assignment.
        Action<Employee> employeeReadContact = personReadContact;

        // This causes a compile error: CS1661.
        // Action<Employee> employeeReadContact2 = (Person p) => p.ReadContact();
    }
}
```

Это поведение кажется противоречивым: если контравариантность позволяет назначать делегат, принимающий базовый тип (`Person`), на переменную делегата, ожидающую производный тип (`Employee`), то почему прямое назначение лямбда-выражения не удается?

Ключевое различие — **вывод типов**. В первом случае лямбда-выражение сначала присваивается переменной с типом `var`, что заставляет компилятор определить тип лямбды как `Action<Person>`. Последующее назначение `Action<Employee>` успешно выполнено из-за правил контравариантности для делегатов.

Во втором случае компилятор не может напрямую определить, что лямбда-выражение `(Person p) => p.ReadContact()` должно иметь тип `Action<Person>` при присвоении `Action<Employee>`. Правила вывода типов для анонимных функций автоматически не применяют контравариантность во время определения исходного типа.

Обходные пути

Для работы с прямым назначением можно использовать явное приведение:

C#

```
// Explicit cast to the desired delegate type.
Action<Employee> employeeReadContact = (Action<Person>)((Person p) =>
p.ReadContact());

// Or specify the lambda parameter type that matches the target delegate.
Action<Employee> employeeReadContact2 = (Employee e) => e.ReadContact();
```

Это поведение иллюстрирует разницу между контравариантностью делегатов (которая работает после установки типов) и выводом типа лямбда-выражения (которое происходит во время компиляции).

См. также

- Ковариантность и контравариантность (C#)
- Обобщения

Итераторы (C#)

13.06.2025

Итератор можно использовать для пошагового выполнения коллекций, таких как списки и массивы.

Метод итератора или `get` аксессор выполняет настраиваемую итерацию по коллекции. Метод итератора использует оператор yield return для возврата каждого элемента по одному. После достижения инструкции запоминается текущее `yield return` расположение в коде. Выполнение перезапускается из этого расположения при следующем вызове функции итератора.

Вы используете итератор из клиентского кода с помощью инструкции foreach или с помощью запроса LINQ.

В следующем примере первая итерация `foreach` цикла приводит к тому, что выполнение продолжается в методе итератора `SomeNumbers` до того, как будет достигнута первая `yield return` инструкция. Эта итерация возвращает значение 3, а текущее расположение в методе итератора сохраняется. В следующей итерации цикла выполнение в методе итератора продолжается с того места, где он остался, снова остановившись при достижении инструкции `yield return` . Эта итерация возвращает значение 5, а текущее расположение в методе итератора снова сохраняется. Цикл завершается после достижения конца метода итератора.

C#

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

Возвращаемый тип метода итератора или аксессора `get` может быть IEnumerable, IEnumerable<T>, IEnumerator или IEnumerator<T>.

Вы можете использовать выражение `yield break` для завершения итерации.

> ⓘ **Примечание**
>
> Для всех примеров в этом разделе, кроме примера Простого итератора, включите директивы **using** для пространств имен `System.Collections` и `System.Collections.Generic`.

Простой итератор

В следующем примере имеется один оператор `yield return`, который находится внутри цикла for. В `Main` каждом итерации текста инструкции `foreach` создается вызов функции итератора, которая переходит к следующей `yield return` инструкции.

C#

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Создание класса коллекции

В следующем примере `DaysOfTheWeek` класс реализует IEnumerable интерфейс, для которого требуется GetEnumerator метод. Компилятор неявно вызывает `GetEnumerator` метод, который возвращает IEnumerator.

Метод `GetEnumerator` возвращает каждую строку по одному за раз с помощью инструкции `yield return` .

C#

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

В следующем примере создается `Zoo` класс, содержащий коллекцию животных.

Оператор `foreach`, ссылающийся на экземпляр класса (`theZoo`), неявно вызывает метод `GetEnumerator`. Инструкции `foreach`, ссылающиеся на свойства `Birds` и `Mammals`, используют метод именованного итератора `AnimalsForType`.

C#

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
```

```
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }
```

```
    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Использование итераторов с универсальным списком

В следующем примере универсальный Stack<T> класс реализует универсальный IEnumerable<T> интерфейс. Метод Push назначает значения массиву типов `T`. Метод GetEnumerator возвращает значения массива с помощью инструкции `yield return` .

В дополнение к обобщенному методу GetEnumerator, также должен быть реализован не обобщенный метод GetEnumerator. Это связано с тем, что IEnumerable<T> наследует от IEnumerable. Не универсальная реализация откладывает универсальную реализацию.

В примере используются именованные итераторы для поддержки различных способов итерации по одной коллекции данных. Эти именованные итераторы — `TopToBottom` и свойства `BottomToTop`, а также метод `TopN`.

Свойство `BottomToTop` использует итератор в аксессоре `get`.

C#

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
```

```
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of
Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }
```

```
    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Сведения о синтаксисе

Итератор может выполняться как метод или `get` метод доступа. Итератор не может использоваться в событии, конструкторе экземпляра, статическом конструкторе или статическом финализаторе.

Неявное преобразование должно существовать из типа выражения в `yield return` операторе в аргумент типа для `IEnumerable<T>` возвращаемого итератора.

В C# метод итератора не может иметь никаких `in` `ref` или `out` параметров.

В C# `yield` не является зарезервированным словом и имеет особое значение, только если используется перед ключевым словом `return` или `break`.

Техническая реализация

Хотя вы пишете итератор в качестве метода, компилятор преобразует его в вложенный класс, который, в действительности, является компьютером состояния. Этот класс отслеживает положение итератора до тех пор, пока `foreach` цикл в клиентском коде продолжается.

Чтобы узнать, что делает компилятор, можно использовать средство Ildasm.exe для просмотра кода общего промежуточного языка, созданного для метода итератора.

При создании итератора для класса или структуры не требуется реализовать весь IEnumerator интерфейс. Когда компилятор обнаруживает итератор, он автоматически создает методы `Current`, `MoveNext` и `Dispose` интерфейса IEnumerator или IEnumerator<T>.

При каждой последовательной `foreach` итерации цикла (или прямого вызова `IEnumerator.MoveNext`), следующий текст кода итератора возобновляется после предыдущей `yield return` инструкции. Затем он продолжается до следующего `yield return` оператора до конца текста итератора или до тех пор `yield break` , пока не будет обнаружен оператор.

Итераторы не поддерживают метод IEnumerator.Reset. Чтобы повторить с самого начала, необходимо получить новый итератор. Вызов Reset итератора, возвращаемого методом итератора, вызывает NotSupportedExceptionисключение.

Дополнительные сведения см. в спецификации языка C#.

Использование итераторов

Итераторы позволяют поддерживать простоту `foreach` цикла, если необходимо использовать сложный код для заполнения последовательности списков. Это может быть полезно, если вы хотите сделать следующее:

- Измените последовательность списков после первой `foreach` итерации цикла.

- Избегайте полной загрузки большого `foreach` списка перед первой итерацией цикла. Примером является постраничная выборка для пакетной загрузки строк таблицы. Другим примером EnumerateFiles является метод, реализующий итераторы в .NET.

- Инкапсулируйте процесс создания списка в итераторе. В методе итератора можно создать список, а затем получить каждый результат в цикле.

См. также

- System.Collections.Generic
- IEnumerable<T>
- foreach в
- Обобщения

Сравнение на равенство в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** В Java `==` для объектов и в JavaScript `===` для объектов проверяют идентичность, а не содержимое. Классы C# работают одинаково по умолчанию. В Python `==` вызывает `__eq__` и по умолчанию сравнивает содержимое, аналогично тому, как сравниваются records в C#. Структуры C# также сравниваются по значению при вызове `Equals`.

C# отличает два типа равенства. *Равенство значений* означает, что два экземпляра равны при совпадении данных. *Равенство ссылок* означает, что две переменные равны только в том случае, если они указывают на один и тот же объект в памяти. Это условие также называется *тождеством*. Типы значений обычно сравнивают по данным, а ссылочные типы — по идентичности. Авторы типов могут изменить эти значения по умолчанию, но эта умственная модель предотвращает тонкие ошибки, в которых два объекта, которые выглядят идентичными, не считаются равными, или где мутация через одну переменную автоматически изменяет то, что видит другая переменная.

Типы значений, ссылочные типы и поведение равенства по умолчанию

Каждый тип в C# — это *тип значения* или *ссылочный тип*. *Тип значения* содержит данные непосредственно в переменной. *Ссылочный тип* содержит ссылку на объект. При назначении переменной ссылочного типа другой переменной обе переменные ссылаются на один и тот же объект. Дополнительные сведения о типах значений и ссылочных типах см. в обзоре системы типов.

Поведение равенства по умолчанию обычно соответствует типу:

- **Встроенные числовые типы и перечисления** — это типы значений. Две `int` переменные равны, если их числовые значения совпадают.

- Структуры — это типы **значений**. Обычный `struct` использует равенство значений при вызове Equals.
- **Кортежи** — это типы значений. Два кортежа равны, если все значения элементов совпадают.
- **Классы** являются ссылочными типами. Обычный класс использует равенство ссылок, поэтому `==`Equals и проверяет, указывают ли две переменные на один и тот же объект.

Класс использует равенство ссылок. Два отдельных объекта с одинаковыми данными не равны, но две переменные, ссылающиеся на один и тот же объект, равны:

C#

```
var order1 = new Order(42, "Shoes");
var order2 = new Order(42, "Shoes");

Console.WriteLine(order1 == order2);                // => False
Console.WriteLine(order1.Equals(order2));           // => False
Console.WriteLine(ReferenceEquals(order1, order2)); // => False

Order order3 = order1;
Console.WriteLine(order1 == order3);                // => True
```

`struct` показывает равенство значений с помощью Equals. Два экземпляра структуры равны при совпадении полей:

C#

```
var pt1 = new Point(3, 4);
var pt2 = new Point(3, 4);

Console.WriteLine(pt1.Equals(pt2)); // => True
```

Простые структуры не получают предопределенного `==` оператора. Использование `p1 == p2` для обычной структуры компилируется, только если структура объявляет собственный `operator ==`. Если вам нужны сравнения операторов для структуры, определите `==` и `!=` как пару и сохраните их в соответствии с Equals и GetHashCode.

Кортежи также являются типами значений. Два кортежа равны, если каждое значение элемента совпадает. Имена элементов в именованном кортеже нужны лишь для удобства на этапе компиляции и не учитываются при сравнении. Важны только позиции и значения:

C#

```
var t1 = (Name: "Grace", Role: "Engineer");
var t2 = (Name: "Grace", Role: "Engineer");

Console.WriteLine(t1 == t2); // => True
```

Дополнительные сведения о синтаксисе кортежей и деконструкции см. в разделе Кортежи и деконструкция.

Используйте `Object.ReferenceEquals`, чтобы напрямую проверить личность

ReferenceEquals всегда проверяет тождественность независимо от того, как тип переопределяет Equals или перегружает `==`. Используйте это для проверки идентичности, когда нужно подтвердить, указывают ли две переменные на один и тот же объект:

C#

```
var doc1 = new Document("Report");
var doc2 = new Document("Report");
var doc3 = doc1;

Console.WriteLine(ReferenceEquals(doc1, doc2)); // => False
Console.WriteLine(ReferenceEquals(doc1, doc3)); // => True
```

Часто это используется в переопределении `Equals`, чтобы избежать полного сравнения: если оба аргумента ссылаются на один и тот же объект, они всегда равны без проверки отдельных полей.

> ⓘ **Примечание.**
>
> Когда переменные имеют тип **интерфейс**, `==` проверяет, ссылаются ли переменные интерфейсного типа на один и тот же объект. Вызов `Equals` по-прежнему выполняет реализацию базового объекта.

> ⓘ **Примечание.**
>
> ReferenceEquals всегда возвращается `false` при сравнении типов значений, даже если оба аргумента содержат одинаковые значения. Это поведение возникает, потому что каждый аргумент типа значения независимо *упаковывается* в отдельный объект в куче при передаче в `ReferenceEquals`.

Типы могут определять различные семантики равенства

Типы *могут* определять семантику равенства, отличающуюся от поведения по умолчанию. Наиболее распространенной причиной является реализация равенства значений. Если вы создаете тип, представляющий данные, такие как банковский счет, продукт в инвентаризации или пользователь в системе, рассмотрите экземпляры с теми же значениями, что и равные. *Выберите типы записей для реализации равенства значений*, а компилятор создает все необходимые элементы равенства.

> ⓘ **Примечание.**
>
> **Строки** — это классы, но `==` и Equals сравнивают содержимое строк, а не их идентичность.

- `==`: оператор равенства. Большинство типов используют этот оператор в качестве основной проверки равенства. Его поведение зависит от того, имеет ли тип встроенный или определяемый `==` пользователем оператор.
- `!=`: оператор неравенства. Если тип определяет определяемый `==` пользователем оператор, он также должен определять `!=`.
- Equals: виртуальный метод, который наследуется каждым типом. Его можно переопределить, чтобы изменить семантику равенства для типа.
- GetHashCode: виртуальный метод, используемый хэш-коллекциями. Если два значения равны, их хэш-коды также должны быть равными.
- ReferenceEquals: статический метод, который всегда проверяет идентичность.

Использование записей для равенства значений

Используйте модификатор `record`, чтобы обеспечить равенство значений для типа, ориентированного на данные, если тип может быть записью. Компилятор генерирует члены Equals, GetHashCode и `==`/`!=`, которые сравнивают значения всех объявленных свойств.

`record class` по-прежнему является ссылочным типом, но сравнивает значения, а не идентичность:

C#

```
var person1 = new Person("Ada", "Lovelace");
var person2 = new Person("Ada", "Lovelace");
```

```
Console.WriteLine(person1 == person2);                // => True
Console.WriteLine(person1.Equals(person2));           // => True
Console.WriteLine(ReferenceEquals(person1, person2)); // => False
```

ReferenceEquals подтверждает, что `person1` и `person2` — это разные объекты в памяти, тогда как `==` и Equals возвращают `True`, поскольку сгенерированная компилятором операция проверки равенства сравнивает значения свойств.

То же поколение компилятора применяется к `record struct` типам:

C#

```
var dim1 = new Dimension(1920, 1080);
var dim2 = new Dimension(1920, 1080);

Console.WriteLine(dim1 == dim2);      // => True
Console.WriteLine(dim1.Equals(dim2)); // => True
```

Типы записей создают весь набор равенства для собственного типа. Типы `record class` и `record struct` переопределяют Equals и GetHashCode. Они также создают `==` и `!=` операторы, а также типизированный `Equals` метод для типа записи. В отличие от обычного `struct`, поэтому `record struct` поддерживается `==` и `!=` автоматически. Дополнительные сведения о типах записей и их семантике равенства см. в разделе "Записи".

Записи с элементами ссылочного типа

Для равенства записей используется собственная семантика равенства членов. Каждое свойство или поле сравнивается с помощью собственного `Equals` метода. Для большинства скалярных значений, таких как `int` `string`, или `DateTime`, этот подход сравнивает значения элементов записи. Тонкость возникает с обычными изменяемыми коллекциями, такими как `List<T>` или `T[]`: эти типы сравниваются по ссылке, поэтому два экземпляра record-типа, содержащие *разные объекты списков с одинаковым содержимым*, **не** считаются равными при автоматически сгенерированном сравнении на равенство для record-типов.

C#

```
var playlist1 = new Playlist("Chill", new List<string> { "Song A", "Song B" });
var playlist2 = new Playlist("Chill", new List<string> { "Song A", "Song B" });

Console.WriteLine(playlist1.Equals(playlist2));                        // => False
```

```
(different List instances)
Console.WriteLine(playlist1.Tracks.SequenceEqual(playlist2.Tracks));    // => True
```

`playlist1` и `playlist2` являются отдельными `List<string>` экземплярами. Хотя их содержимое совпадает, `Equals` возвращает `false`.

Если вам нужно, чтобы равенство записей учитывало *содержимое* коллекции, у вас есть несколько вариантов:

- Реализуйте **`IEquatable<T>`** в записи и переопределите `Equals`, чтобы использовать Enumerable.SequenceEqual для элементов коллекции.
- **Используйте тип коллекции с равенством значений** , например настраиваемый `IEqualityComparer<T>` или тип, собственный `Equals` который сравнивает элементы.
- **Проектирование с учётом идентичности**: если запись представляет сущность, а не чистое значение, то ссылочное равенство для элементов её коллекции может быть преднамеренным.

> ⓘ **Важно!**
>
> Реализация равенства вручную редко используется сегодня в C#. Записи обрабатывают общий сценарий равенства значений автоматически. Если вам нужно вручную реализовать равенство — например, потому что ваш тип должен наследоваться от базового класса, не являющегося записью, — см. раздел **Самостоятельная реализация равенства, если тип не может быть записью** в справочнике языка.

См. также

- Общие сведения о системе типов
- Классы
- Структуры
- Записи
- Кортежи и деконструкция.
- Операторы равенства (справочник по языку).
- Равенство в иерархиях классов — расширенное руководство по полиморфическому равенству.
- Арифметические операторы, операторы сравнения, логические операторы и операторы присваивания — обзор оператора равенства наряду с арифметическими, логическими операторами и операторами присваивания.

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.08.2026

Сравнение на равенство в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** В Java `==` для объектов и в JavaScript `===` для объектов проверяют идентичность, а не содержимое. Классы C# работают одинаково по умолчанию. В Python `==` вызывает `__eq__` и по умолчанию сравнивает содержимое, аналогично тому, как сравниваются records в C#. Структуры C# также сравниваются по значению при вызове `Equals`.

C# отличает два типа равенства. *Равенство значений* означает, что два экземпляра равны при совпадении данных. *Равенство ссылок* означает, что две переменные равны только в том случае, если они указывают на один и тот же объект в памяти. Это условие также называется *тождеством*. Типы значений обычно сравнивают по данным, а ссылочные типы — по идентичности. Авторы типов могут изменить эти значения по умолчанию, но эта умственная модель предотвращает тонкие ошибки, в которых два объекта, которые выглядят идентичными, не считаются равными, или где мутация через одну переменную автоматически изменяет то, что видит другая переменная.

Типы значений, ссылочные типы и поведение равенства по умолчанию

Каждый тип в C# — это *тип значения* или *ссылочный тип*. *Тип значения* содержит данные непосредственно в переменной. *Ссылочный тип* содержит ссылку на объект. При назначении переменной ссылочного типа другой переменной обе переменные ссылаются на один и тот же объект. Дополнительные сведения о типах значений и ссылочных типах см. в обзоре системы типов.

Поведение равенства по умолчанию обычно соответствует типу:

- **Встроенные числовые типы и перечисления** — это типы значений. Две `int` переменные равны, если их числовые значения совпадают.

- Структуры — это типы **значений**. Обычный `struct` использует равенство значений при вызове Equals.
- **Кортежи** — это типы значений. Два кортежа равны, если все значения элементов совпадают.
- **Классы** являются ссылочными типами. Обычный класс использует равенство ссылок, поэтому `==`Equals и проверяет, указывают ли две переменные на один и тот же объект.

Класс использует равенство ссылок. Два отдельных объекта с одинаковыми данными не равны, но две переменные, ссылающиеся на один и тот же объект, равны:

C#

```
var order1 = new Order(42, "Shoes");
var order2 = new Order(42, "Shoes");

Console.WriteLine(order1 == order2);                // => False
Console.WriteLine(order1.Equals(order2));           // => False
Console.WriteLine(ReferenceEquals(order1, order2)); // => False

Order order3 = order1;
Console.WriteLine(order1 == order3);                // => True
```

`struct` показывает равенство значений с помощью Equals. Два экземпляра структуры равны при совпадении полей:

C#

```
var pt1 = new Point(3, 4);
var pt2 = new Point(3, 4);

Console.WriteLine(pt1.Equals(pt2)); // => True
```

Простые структуры не получают предопределенного `==` оператора. Использование `p1 == p2` для обычной структуры компилируется, только если структура объявляет собственный `operator ==`. Если вам нужны сравнения операторов для структуры, определите `==` и `!=` как пару и сохраните их в соответствии с Equals и GetHashCode.

Кортежи также являются типами значений. Два кортежа равны, если каждое значение элемента совпадает. Имена элементов в именованном кортеже нужны лишь для удобства на этапе компиляции и не учитываются при сравнении. Важны только позиции и значения:

C#

```
var t1 = (Name: "Grace", Role: "Engineer");
var t2 = (Name: "Grace", Role: "Engineer");

Console.WriteLine(t1 == t2); // => True
```

Дополнительные сведения о синтаксисе кортежей и деконструкции см. в разделе Кортежи и деконструкция.

Используйте `Object.ReferenceEquals`, чтобы напрямую проверить личность

ReferenceEquals всегда проверяет тождественность независимо от того, как тип переопределяет Equals или перегружает `==`. Используйте это для проверки идентичности, когда нужно подтвердить, указывают ли две переменные на один и тот же объект:

C#

```
var doc1 = new Document("Report");
var doc2 = new Document("Report");
var doc3 = doc1;

Console.WriteLine(ReferenceEquals(doc1, doc2)); // => False
Console.WriteLine(ReferenceEquals(doc1, doc3)); // => True
```

Часто это используется в переопределении `Equals`, чтобы избежать полного сравнения: если оба аргумента ссылаются на один и тот же объект, они всегда равны без проверки отдельных полей.

> ⓘ **Примечание.**
>
> Когда переменные имеют тип **интерфейс**, `==` проверяет, ссылаются ли переменные интерфейсного типа на один и тот же объект. Вызов `Equals` по-прежнему выполняет реализацию базового объекта.

> ⓘ **Примечание.**
>
> ReferenceEquals всегда возвращается `false` при сравнении типов значений, даже если оба аргумента содержат одинаковые значения. Это поведение возникает, потому что каждый аргумент типа значения независимо *упаковывается* в отдельный объект в куче при передаче в `ReferenceEquals`.

Типы могут определять различные семантики равенства

Типы *могут* определять семантику равенства, отличающуюся от поведения по умолчанию. Наиболее распространенной причиной является реализация равенства значений. Если вы создаете тип, представляющий данные, такие как банковский счет, продукт в инвентаризации или пользователь в системе, рассмотрите экземпляры с теми же значениями, что и равные. *Выберите типы записей для реализации равенства значений*, а компилятор создает все необходимые элементы равенства.

> ⓘ **Примечание.**
>
> **Строки** — это классы, но `==` и Equals сравнивают содержимое строк, а не их идентичность.

- `==`: оператор равенства. Большинство типов используют этот оператор в качестве основной проверки равенства. Его поведение зависит от того, имеет ли тип встроенный или определяемый `==` пользователем оператор.
- `!=`: оператор неравенства. Если тип определяет определяемый `==` пользователем оператор, он также должен определять `!=`.
- Equals: виртуальный метод, который наследуется каждым типом. Его можно переопределить, чтобы изменить семантику равенства для типа.
- GetHashCode: виртуальный метод, используемый хэш-коллекциями. Если два значения равны, их хэш-коды также должны быть равными.
- ReferenceEquals: статический метод, который всегда проверяет идентичность.

Использование записей для равенства значений

Используйте модификатор `record`, чтобы обеспечить равенство значений для типа, ориентированного на данные, если тип может быть записью. Компилятор генерирует члены Equals, GetHashCode и `==`/`!=`, которые сравнивают значения всех объявленных свойств.

`record class` по-прежнему является ссылочным типом, но сравнивает значения, а не идентичность:

C#

```
var person1 = new Person("Ada", "Lovelace");
var person2 = new Person("Ada", "Lovelace");
```

```
Console.WriteLine(person1 == person2);                // => True
Console.WriteLine(person1.Equals(person2));           // => True
Console.WriteLine(ReferenceEquals(person1, person2)); // => False
```

ReferenceEquals подтверждает, что `person1` и `person2` — это разные объекты в памяти, тогда как `==` и Equals возвращают `True`, поскольку сгенерированная компилятором операция проверки равенства сравнивает значения свойств.

То же поколение компилятора применяется к `record struct` типам:

C#

```
var dim1 = new Dimension(1920, 1080);
var dim2 = new Dimension(1920, 1080);

Console.WriteLine(dim1 == dim2);      // => True
Console.WriteLine(dim1.Equals(dim2)); // => True
```

Типы записей создают весь набор равенства для собственного типа. Типы `record class` и `record struct` переопределяют Equals и GetHashCode. Они также создают `==` и `!=` операторы, а также типизированный `Equals` метод для типа записи. В отличие от обычного `struct`, поэтому `record struct` поддерживается `==` и `!=` автоматически. Дополнительные сведения о типах записей и их семантике равенства см. в разделе "Записи".

Записи с элементами ссылочного типа

Для равенства записей используется собственная семантика равенства членов. Каждое свойство или поле сравнивается с помощью собственного `Equals` метода. Для большинства скалярных значений, таких как `int` `string`, или `DateTime`, этот подход сравнивает значения элементов записи. Тонкость возникает с обычными изменяемыми коллекциями, такими как `List<T>` или `T[]`: эти типы сравниваются по ссылке, поэтому два экземпляра record-типа, содержащие *разные объекты списков с одинаковым содержимым*, **не** считаются равными при автоматически сгенерированном сравнении на равенство для record-типов.

C#

```
var playlist1 = new Playlist("Chill", new List<string> { "Song A", "Song B" });
var playlist2 = new Playlist("Chill", new List<string> { "Song A", "Song B" });

Console.WriteLine(playlist1.Equals(playlist2));                        // => False
```

```
(different List instances)
Console.WriteLine(playlist1.Tracks.SequenceEqual(playlist2.Tracks));    // => True
```

`playlist1` и `playlist2` являются отдельными `List<string>` экземплярами. Хотя их содержимое совпадает, `Equals` возвращает `false`.

Если вам нужно, чтобы равенство записей учитывало *содержимое* коллекции, у вас есть несколько вариантов:

- Реализуйте **`IEquatable<T>`** в записи и переопределите `Equals`, чтобы использовать Enumerable.SequenceEqual для элементов коллекции.
- **Используйте тип коллекции с равенством значений** , например настраиваемый `IEqualityComparer<T>` или тип, собственный `Equals` который сравнивает элементы.
- **Проектирование с учётом идентичности**: если запись представляет сущность, а не чистое значение, то ссылочное равенство для элементов её коллекции может быть преднамеренным.

> ⓘ **Важно!**
>
> Реализация равенства вручную редко используется сегодня в C#. Записи обрабатывают общий сценарий равенства значений автоматически. Если вам нужно вручную реализовать равенство — например, потому что ваш тип должен наследоваться от базового класса, не являющегося записью, — см. раздел **Самостоятельная реализация равенства, если тип не может быть записью** в справочнике языка.

См. также

- Общие сведения о системе типов
- Классы
- Структуры
- Записи
- Кортежи и деконструкция.
- Операторы равенства (справочник по языку).
- Равенство в иерархиях классов — расширенное руководство по полиморфическому равенству.
- Арифметические операторы, операторы сравнения, логические операторы и операторы присваивания — обзор оператора равенства наряду с арифметическими, логическими операторами и операторами присваивания.

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.08.2026

Сравнение на равенство в C#

> **Совет**
>
> Эта статья является частью раздела **"Основы" для разработчиков** , которые уже знают хотя бы один язык программирования и учат C#. Если вы не знакомы с программированием, сначала начните с учебных пособий по началу работы.
>
> **Вы пришли из другого языка?** В Java `==` для объектов и в JavaScript `===` для объектов проверяют идентичность, а не содержимое. Классы C# работают одинаково по умолчанию. В Python `==` вызывает `__eq__` и по умолчанию сравнивает содержимое, аналогично тому, как сравниваются records в C#. Структуры C# также сравниваются по значению при вызове `Equals`.

C# отличает два типа равенства. *Равенство значений* означает, что два экземпляра равны при совпадении данных. *Равенство ссылок* означает, что две переменные равны только в том случае, если они указывают на один и тот же объект в памяти. Это условие также называется *тождеством*. Типы значений обычно сравнивают по данным, а ссылочные типы — по идентичности. Авторы типов могут изменить эти значения по умолчанию, но эта умственная модель предотвращает тонкие ошибки, в которых два объекта, которые выглядят идентичными, не считаются равными, или где мутация через одну переменную автоматически изменяет то, что видит другая переменная.

Типы значений, ссылочные типы и поведение равенства по умолчанию

Каждый тип в C# — это *тип значения* или *ссылочный тип*. *Тип значения* содержит данные непосредственно в переменной. *Ссылочный тип* содержит ссылку на объект. При назначении переменной ссылочного типа другой переменной обе переменные ссылаются на один и тот же объект. Дополнительные сведения о типах значений и ссылочных типах см. в обзоре системы типов.

Поведение равенства по умолчанию обычно соответствует типу:

- **Встроенные числовые типы и перечисления** — это типы значений. Две `int` переменные равны, если их числовые значения совпадают.

- Структуры — это типы **значений**. Обычный `struct` использует равенство значений при вызове Equals.
- **Кортежи** — это типы значений. Два кортежа равны, если все значения элементов совпадают.
- **Классы** являются ссылочными типами. Обычный класс использует равенство ссылок, поэтому `==`Equals и проверяет, указывают ли две переменные на один и тот же объект.

Класс использует равенство ссылок. Два отдельных объекта с одинаковыми данными не равны, но две переменные, ссылающиеся на один и тот же объект, равны:

C#

```
var order1 = new Order(42, "Shoes");
var order2 = new Order(42, "Shoes");

Console.WriteLine(order1 == order2);                // => False
Console.WriteLine(order1.Equals(order2));           // => False
Console.WriteLine(ReferenceEquals(order1, order2)); // => False

Order order3 = order1;
Console.WriteLine(order1 == order3);                // => True
```

`struct` показывает равенство значений с помощью Equals. Два экземпляра структуры равны при совпадении полей:

C#

```
var pt1 = new Point(3, 4);
var pt2 = new Point(3, 4);

Console.WriteLine(pt1.Equals(pt2)); // => True
```

Простые структуры не получают предопределенного `==` оператора. Использование `p1 == p2` для обычной структуры компилируется, только если структура объявляет собственный `operator ==`. Если вам нужны сравнения операторов для структуры, определите `==` и `!=` как пару и сохраните их в соответствии с Equals и GetHashCode.

Кортежи также являются типами значений. Два кортежа равны, если каждое значение элемента совпадает. Имена элементов в именованном кортеже нужны лишь для удобства на этапе компиляции и не учитываются при сравнении. Важны только позиции и значения:

C#

```
var t1 = (Name: "Grace", Role: "Engineer");
var t2 = (Name: "Grace", Role: "Engineer");

Console.WriteLine(t1 == t2); // => True
```

Дополнительные сведения о синтаксисе кортежей и деконструкции см. в разделе Кортежи и деконструкция.

Используйте `Object.ReferenceEquals`, чтобы напрямую проверить личность

ReferenceEquals всегда проверяет тождественность независимо от того, как тип переопределяет Equals или перегружает `==`. Используйте это для проверки идентичности, когда нужно подтвердить, указывают ли две переменные на один и тот же объект:

C#

```
var doc1 = new Document("Report");
var doc2 = new Document("Report");
var doc3 = doc1;

Console.WriteLine(ReferenceEquals(doc1, doc2)); // => False
Console.WriteLine(ReferenceEquals(doc1, doc3)); // => True
```

Часто это используется в переопределении `Equals`, чтобы избежать полного сравнения: если оба аргумента ссылаются на один и тот же объект, они всегда равны без проверки отдельных полей.

> ⓘ **Примечание.**
>
> Когда переменные имеют тип **интерфейс**, `==` проверяет, ссылаются ли переменные интерфейсного типа на один и тот же объект. Вызов `Equals` по-прежнему выполняет реализацию базового объекта.

> ⓘ **Примечание.**
>
> ReferenceEquals всегда возвращается `false` при сравнении типов значений, даже если оба аргумента содержат одинаковые значения. Это поведение возникает, потому что каждый аргумент типа значения независимо *упаковывается* в отдельный объект в куче при передаче в `ReferenceEquals`.

Типы могут определять различные семантики равенства

Типы *могут* определять семантику равенства, отличающуюся от поведения по умолчанию. Наиболее распространенной причиной является реализация равенства значений. Если вы создаете тип, представляющий данные, такие как банковский счет, продукт в инвентаризации или пользователь в системе, рассмотрите экземпляры с теми же значениями, что и равные. *Выберите типы записей для реализации равенства значений*, а компилятор создает все необходимые элементы равенства.

> ⓘ **Примечание.**
>
> **Строки** — это классы, но `==` и Equals сравнивают содержимое строк, а не их идентичность.

- `==`: оператор равенства. Большинство типов используют этот оператор в качестве основной проверки равенства. Его поведение зависит от того, имеет ли тип встроенный или определяемый `==` пользователем оператор.
- `!=`: оператор неравенства. Если тип определяет определяемый `==` пользователем оператор, он также должен определять `!=`.
- Equals: виртуальный метод, который наследуется каждым типом. Его можно переопределить, чтобы изменить семантику равенства для типа.
- GetHashCode: виртуальный метод, используемый хэш-коллекциями. Если два значения равны, их хэш-коды также должны быть равными.
- ReferenceEquals: статический метод, который всегда проверяет идентичность.

Использование записей для равенства значений

Используйте модификатор `record`, чтобы обеспечить равенство значений для типа, ориентированного на данные, если тип может быть записью. Компилятор генерирует члены Equals, GetHashCode и `==`/`!=`, которые сравнивают значения всех объявленных свойств.

`record class` по-прежнему является ссылочным типом, но сравнивает значения, а не идентичность:

C#

```
var person1 = new Person("Ada", "Lovelace");
var person2 = new Person("Ada", "Lovelace");
```

```
Console.WriteLine(person1 == person2);                // => True
Console.WriteLine(person1.Equals(person2));           // => True
Console.WriteLine(ReferenceEquals(person1, person2)); // => False
```

ReferenceEquals подтверждает, что `person1` и `person2` — это разные объекты в памяти, тогда как `==` и Equals возвращают `True`, поскольку сгенерированная компилятором операция проверки равенства сравнивает значения свойств.

То же поколение компилятора применяется к `record struct` типам:

C#

```
var dim1 = new Dimension(1920, 1080);
var dim2 = new Dimension(1920, 1080);

Console.WriteLine(dim1 == dim2);      // => True
Console.WriteLine(dim1.Equals(dim2)); // => True
```

Типы записей создают весь набор равенства для собственного типа. Типы `record class` и `record struct` переопределяют Equals и GetHashCode. Они также создают `==` и `!=` операторы, а также типизированный `Equals` метод для типа записи. В отличие от обычного `struct`, поэтому `record struct` поддерживается `==` и `!=` автоматически. Дополнительные сведения о типах записей и их семантике равенства см. в разделе "Записи".

Записи с элементами ссылочного типа

Для равенства записей используется собственная семантика равенства членов. Каждое свойство или поле сравнивается с помощью собственного `Equals` метода. Для большинства скалярных значений, таких как `int` `string`, или `DateTime`, этот подход сравнивает значения элементов записи. Тонкость возникает с обычными изменяемыми коллекциями, такими как `List<T>` или `T[]`: эти типы сравниваются по ссылке, поэтому два экземпляра record-типа, содержащие *разные объекты списков с одинаковым содержимым*, **не** считаются равными при автоматически сгенерированном сравнении на равенство для record-типов.

C#

```
var playlist1 = new Playlist("Chill", new List<string> { "Song A", "Song B" });
var playlist2 = new Playlist("Chill", new List<string> { "Song A", "Song B" });

Console.WriteLine(playlist1.Equals(playlist2));                        // => False
```

```
(different List instances)
Console.WriteLine(playlist1.Tracks.SequenceEqual(playlist2.Tracks));    // => True
```

`playlist1` и `playlist2` являются отдельными `List<string>` экземплярами. Хотя их содержимое совпадает, `Equals` возвращает `false`.

Если вам нужно, чтобы равенство записей учитывало *содержимое* коллекции, у вас есть несколько вариантов:

- Реализуйте **`IEquatable<T>`** в записи и переопределите `Equals`, чтобы использовать Enumerable.SequenceEqual для элементов коллекции.
- **Используйте тип коллекции с равенством значений** , например настраиваемый `IEqualityComparer<T>` или тип, собственный `Equals` который сравнивает элементы.
- **Проектирование с учётом идентичности**: если запись представляет сущность, а не чистое значение, то ссылочное равенство для элементов её коллекции может быть преднамеренным.

> ⓘ **Важно!**
>
> Реализация равенства вручную редко используется сегодня в C#. Записи обрабатывают общий сценарий равенства значений автоматически. Если вам нужно вручную реализовать равенство — например, потому что ваш тип должен наследоваться от базового класса, не являющегося записью, — см. раздел **Самостоятельная реализация равенства, если тип не может быть записью** в справочнике языка.

См. также

- Общие сведения о системе типов
- Классы
- Структуры
- Записи
- Кортежи и деконструкция.
- Операторы равенства (справочник по языку).
- Равенство в иерархиях классов — расширенное руководство по полиморфическому равенству.
- Арифметические операторы, операторы сравнения, логические операторы и операторы присваивания — обзор оператора равенства наряду с арифметическими, логическими операторами и операторами присваивания.

ⓘ**Примечание.** Автор создал эту статью с помощью ИИ. Подробнее

Last updated on 24.08.2026

Приведение типов и их преобразование (руководство по программированию на C#)

17.06.2025

Так как C# статически типируется во время компиляции, после объявления переменной его нельзя объявить снова или назначить значение другого типа, если этот тип неявно преобразуется в тип переменной. Например, `string` невозможно неявно преобразовать в `int`. Таким образом, после объявления `i` как `int`, вы не можете присвоить ему строку "Привет", как показано в следующем коде:

C#

```
int i;

// error CS0029: can't implicitly convert type 'string' to 'int'
i = "Hello";
```

Однако иногда может потребоваться скопировать значение в переменную или параметр метода другого типа. Например, у вас может быть целочисленная переменная, которую необходимо передать методу, параметр которого вводится как `double`. Кроме того, может потребоваться назначить переменную класса переменной типа интерфейса. Эти виды операций называются *преобразованиями типов*. В C#можно выполнить следующие виды преобразований:

- **Неявные преобразования: специальный** синтаксис не требуется, так как преобразование всегда выполняется успешно, и данные не теряются. Примеры включают преобразования из меньших целочисленных типов в большие, преобразования производных классов в базовые классы и преобразования охватов.

- **Явные преобразования (приведения): для явных** преобразований требуется выражение приведения. Приведение требуется, если информация может быть потеряна в преобразовании или когда преобразование может не выполнено по другим причинам. Типичные примеры включают числовое преобразование в тип с меньшей точностью или меньшим диапазоном, а также преобразование экземпляра базового класса в производный класс.

- Определяемые **пользователем преобразования**: определяемые пользователем преобразования используют специальные методы, которые можно определить, чтобы включить явные и неявные преобразования между пользовательскими

типами, у которых нет связи на основе базового класса. Дополнительные сведения см. в разделе Операторы пользовательского преобразования.

- **Преобразования с вспомогательными классами**: для преобразования между несовместимыми типами, такими как целые числа и объекты, или шестнадцатеричные строки и System.DateTime массивы байтов, можно использовать System.BitConverter класс, System.Convert класс и `Parse` методы встроенных числовых типов, например Int32.Parse. Дополнительные сведения см. в следующих статьях:

- Преобразование массива байтов в int
- Преобразование строки в число
- Преобразование между шестнадцатеричными строками и числовыми типами

Неявные преобразования

Для встроенных числовых типов неявное преобразование может быть выполнено, когда значение, которое нужно сохранить, может поместиться в переменную без усечения или округления. Для целочисленных типов это ограничение означает, что диапазон исходного типа является правильным подмножеством диапазона для целевого типа. Например, переменная типа long (64-разрядное целое число) может хранить любое значение, которое может хранить целое число int (32-разрядное целое число). В следующем примере компилятор неявно преобразует значение `num` справа в тип `long` перед назначением `bigNum`.

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Полный список всех неявных числовых преобразований см. в разделе "Неявные числовые преобразования" статьи "Встроенные числовые преобразования".

Для ссылочных типов неявное преобразование всегда существует из класса в любой из его прямых или косвенных базовых классов или интерфейсов. Специальный синтаксис не требуется, так как производный класс всегда содержит все члены базового класса.

C#

```
Derived d = new Derived();
```

```
// Always OK.
Base b = d;
```

Явные преобразования

Однако, если преобразование невозможно сделать без риска потери информации, компилятор требует выполнения явного преобразования, которое называется *приведением*. Приведение — это способ явного преобразования. Это указывает на то, что вы осознаете возможность потери данных или неудачного приведения во время выполнения. Чтобы выполнить приведение, укажите тип назначения в скобках перед выражением, которое нужно преобразовать. Следующая программа приводит двойник к int. Программа не компилируется без приведения.

C#

```
double x = 1234.7;
int a;
// Cast double to int.
a = (int)x;
Console.WriteLine(a);
// Output: 1234
```

Полный список поддерживаемых явных числовых преобразований см. в разделе "Явные числовые преобразования" статьи "Встроенные числовые преобразования".

Для ссылочных типов требуется явное приведение, если необходимо преобразовать из базового типа в производный тип:

Операция приведения между ссылочными типами не изменяет тип времени выполнения исходного объекта; она изменяет только тип значения, используемого в качестве ссылки на этот объект. Дополнительные сведения см. в разделе Полиморфизм.

Исключения преобразования типов во время выполнения

В некоторых преобразованиях ссылочного типа компилятор не может определить, является ли приведение допустимым. Возможно, операция приведения, которая компилируется без ошибок, может завершиться неудачей во время выполнения. Как показано в следующем примере, неудачное приведение типа во время выполнения программы приводит к тому, что выбрасывается исключение InvalidCastException.

C#

```
Animal a = new Mammal();
Reptile r = (Reptile)a; // InvalidCastException at run time
```

Явное приведение аргумента `a` к `Reptile` приводит к опасному предположению. Лучше не делать предположений, а проверять тип. C# предоставляет оператор is, чтобы проверить совместимость перед выполнением приведения. Дополнительные сведения см. в разделе «Безопасное приведение» с использованием сопоставления шаблонов и операторов as и is.

Спецификация языка C#

Дополнительные сведения см. в разделе "Преобразования " спецификации языка C#.

См. также

- Типы
- Выражение приведения
- Определяемые пользователем операторы преобразования
- Обобщенное преобразование типов
- Преобразование строки в число

Упаковка и распаковка (руководство по программированию на C#)

13.10.2025

Упаковка — это процесс преобразования значимого типа в тип `object` или в любой тип интерфейса, реализованный этим значимым типом. Когда общеязыковая среда выполнения (CLR) упаковывает тип значения, она оборачивает значение внутри экземпляра System.Object и сохраняет его в управляемой куче. Процесс распаковки извлекает тип данных из объекта. Бокс неявен; распаковка является явной. Концепции упаковки и распаковки лежат в основе единого представления системы типов C#, в которой любое значение можно считать объектом.

В следующем примере целочисленная переменная `i` *упаковывается* и присваивается объекту `o`.

C#

```
int i = 123;
// The following line boxes i.
object o = i;
```

Затем объект `o` можно распаковать и присвоить целочисленной переменной `i`.

C#

```
o = 123;
i = (int)o;  // unboxing
```

В следующих примерах показано, как бокс используется в C#.

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = [];

// Add a string element to the list.
```

```
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j];

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

Console.WriteLine($"Sum: {sum}");

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
```

```
// 6
// 7
// 8
// 9
// Sum: 30
```

Производительность

В связи с простыми назначениями, боксинг и распаковка являются дорогостоящими вычислительными процессами. Если задан тип значения, необходимо выделить и создать новый объект. В меньшей степени каст, необходимый для распаковки, также является дорогостоящим с вычислительной точки зрения. Дополнительные сведения см. в разделе "Производительность".

Бокс

Бокс используется для хранения типов значений в куче, собранной мусором. Упаковка — это неявное преобразование типа значения в тип `object` или в любой тип интерфейса, реализованный этим типом значения. Поле типа значения выделяет экземпляр объекта в куче и копирует значение в новый объект.

Рассмотрим следующее объявление переменной типа значения:

C#

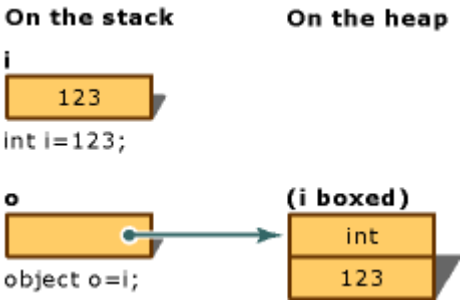
```
int i = 123;
```

Следующая инструкция неявно применяет операцию бокса для переменной `i`:

C#

```
// Boxing copies the value of i into object o.
object o = i;
```

Результатом этого оператора является создание ссылки `o` на объект в стеке, которая ссылается на значение типа `int` в куче. Это значение является копией значения типа значений, назначенных переменной `i`. Разница между двумя переменными `i` и `o` показана на следующем изображении преобразования бокса:



Также можно явно выполнить бокс, как в следующем примере, но явное боксирование никогда не требуется:

C#

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Пример

Этот пример преобразует целочисленную переменную `i` в объект `o` с использованием механизма упаковки. Затем значение, хранящееся в переменной `i` , изменяется на `123` `456`. В примере показано, что исходный тип значения и прямоугольный объект используют отдельные расположения памяти и, следовательно, может хранить разные значения.

C#

```
        // Create an int variable
        int i = 123;

        // Box the value type into an object reference
        object o = i;  // boxing

        // Display the initial values
        Console.WriteLine($"Value of i: {i}");
        Console.WriteLine($"Value of boxed object o: {o}");

        // Modify the original value type
        i = 456;

        // Display the values after modification
        Console.WriteLine("\nAfter changing i to 456:");
        Console.WriteLine($"Value of i: {i}");
        Console.WriteLine($"Value of boxed object o: {o}");

        // Output:
        // Value of i: 123
        // Value of boxed object o: 123
```

```
        // After changing i to 456:
        // Value of i: 456
        // Value of boxed object o: 123
```

Распаковки

Распаковка — это явное преобразование из типа `object` в тип значения или из типа интерфейса в тип значения, который реализует интерфейс. Операция распаковки состоит из следующих элементов:

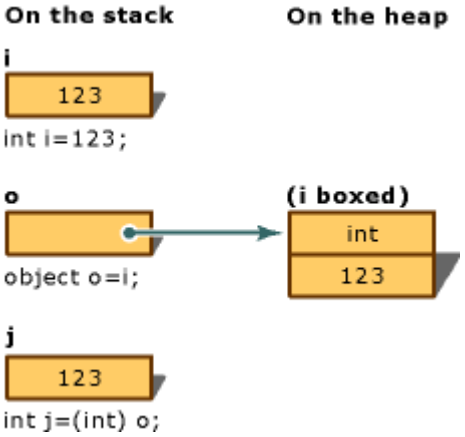
- Проверьте экземпляр объекта, чтобы убедиться, что он является упакованным значением данного типа данных.
- Копирование значения из экземпляра в переменную типа значения.

В следующих инструкциях показаны операции бокса и распаковки:

C#

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

На следующем рисунке показан результат предыдущих утверждений.



Чтобы распаковка типов значений была успешной во время выполнения программы, элемент, который распаковывается, должен быть ссылкой на объект, ранее созданный путем упаковки экземпляра этого типа значения. Попытка распаковки `null` приводит к NullReferenceException. Попытка разблокировать ссылку на несовместимый тип значения приводит InvalidCastExceptionк возникновению ошибки.

Пример

В следующем примере показан случай недопустимой распаковки и результирующего `InvalidCastException`. При использовании `try` и `catch`, отображается сообщение об ошибке, когда происходит ошибка.

```
C#
```

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            Console.WriteLine("Unboxing OK.");
        }
        catch (InvalidCastException e)
        {
            Console.WriteLine($"{e.Message} Error: Incorrect unboxing.");
        }
    }
}
```

Выходные данные этой программы:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

При изменении заявления:

```
C#
```

```
int j = (short)o;
```

Кому:

```
C#
```

```
int j = (int)o;
```

Преобразование выполняется, и вы получите выходные данные:

```
Unboxing OK.
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C# . Спецификация языка является авторитетным источником синтаксиса и использования C#.

См. также

- Ссылочные типы
- типы значений

Преобразование массива байтов в int (руководство по программированию на C#)

Статья • 04.04.2025

В этом примере показано, как использовать BitConverter класс для преобразования массива байтов в int и обратно в массив байтов. Возможно, вам придется преобразовать байты во встроенный тип данных, например, после их получения из сети. В дополнение к методу ToInt32(Byte[], Int32) в примере в следующей таблице перечислены методы в классе, преобразующем байты (из массива байтов) в BitConverter другие встроенные типы.

Развернуть таблицу

Возвращаемый тип	Метод
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Примеры

В этом примере инициализируется массив байтов, переворачивает массив, если архитектура компьютера является малоконечной (то есть младший байт хранится первым), а затем выполняется вызов метода ToInt32(Byte[], Int32) для

преобразования четырех байтов в `int`. Второй аргумент ToInt32(Byte[], Int32) задает начальный индекс массива байтов.

> ⓘ **Примечание**
>
> Выходные данные могут отличаться в зависимости от порядка байтов архитектуры вашего компьютера.

C#

```
byte[] bytes = [0, 0, 0, 25];

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine($"int: {i}");
// Output: int: 25
```

В этом примере метод класса BitConverterGetBytes(Int32) вызывается для преобразования `int` в массив байтов.

> ⓘ **Примечание**
>
> Выходные данные могут отличаться в зависимости от порядка байтов (эндианности) архитектуры вашего компьютера.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

См. также

- BitConverter
- IsLittleEndian
- Типы

Руководство по программированию на C#. Преобразование строки в число

Статья • 28.03.2025

Для преобразования `string` в число используется вызов метода `Parse` или `TryParse`, который можно найти в числовых типах (`int`, `long`, `double` и т. д.), или используются методы в классе System.Convert.

Немного эффективнее и проще вызвать метод `TryParse` (например, int.TryParse("11", out number)) или метод `Parse` (например, var number = int.Parse("11")). Использование метода Convert более удобно для общих объектов, реализующих IConvertible.

Можно использовать методы `Parse` или `TryParse` в числовом типе, который предположительно содержит строка, таком как тип System.Int32. Метод Convert.ToInt32 использует Parse внутри себя. Метод `Parse` возвращает преобразованное число; метод `TryParse` возвращает логическое значение, которое указывает, успешно ли выполнено преобразование, и возвращает преобразованное число в параметр `out`. Если строка имеет недопустимый формат, `Parse` создает исключение, а `TryParse` возвращает значение `false`. В случае сбоя операции синтаксического анализа при вызове метода `Parse` вы всегда должны использовать обработку исключений, чтобы перехватить FormatException.

> **Совет**
>
> С помощью ИИ можно **преобразовать строку в число с помощью GitHub Copilot**.

Вызов метода Parse или TryParse

Методы `Parse` и `TryParse` игнорируют пробелы в начале и в конце строки, но все остальные символы должны быть символами, которые образуют соответствующий числовой тип (`int`, `long`, `ulong`, `float`, `decimal` и т. д.). Любые пробелы в строке, образующие число, приводят к ошибке. Например, можно использовать `decimal.TryParse` для анализа "10", "10.3" или " 10 ", но этот метод нельзя использовать для анализа 10 из "10X", "1 0" (обратите внимание на внедренный

пробел), "10 .3" (обратите внимание на внедренный пробел), "10e1" (здесь работает `float.TryParse`) и т. д. Строку со значением `null` или String.Empty невозможно успешно проанализировать. Вы можете проверить наличие NULL или пустой строки, прежде чем пытаться ее проанализировать, вызвав метод String.IsNullOrEmpty.

В указанном ниже примере демонстрируются успешные и неуспешные вызовы методов `Parse` и `TryParse`.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
```

```
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse
'{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

В следующем примере показан один из подходов к анализу строки, которая, как ожидается, будет включать начальные числовые символы (включая шестнадцатеричные символы) и конечные нечисловые символы. Он назначает допустимые символы в начале новой строки перед вызовом метода TryParse. Поскольку анализируемые строки содержат небольшое количество символов, в примере вызывается метод String.Concat для назначения допустимых символов новой строке. Для большей строки можете использовать класс StringBuilder.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or
trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A'
&& char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
```

```
                break;
            }
        }

        if (int.TryParse(numericString,
System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or
leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Вызов методов Convert

В следующей таблице перечислены некоторые методы класса Convert, которые можно использовать для преобразования строки в число.

Развернуть таблицу

Числовой тип	Способ
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)

Числовой тип	Способ
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

В следующем примере показан прямой вызов метода Convert.ToInt32(String) для преобразования входной строки в int. В примере обрабатываются два наиболее распространенных исключения, выбрасываемых этим методом: FormatException и OverflowException. Если итоговое число можно увеличить, не превышая Int32.MaxValue, пример добавляет 1 к результату и отображает вывод.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between −2,147,483,648 and
+2,147,483,647 (inclusive): ");

            string? input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine($"The new value is {++numVal}");
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond
its current value");
                }
```

```
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of
digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
-1000
//   The new value is -999
//   Go again? Y/N: n
```

Использование GitHub Copilot для преобразования строки в число

Вы можете использовать GitHub Copilot в вашей интегрированной среде разработки, чтобы написать код на C# для преобразования строки в число. Вы можете настроить запрос на использование строки в соответствии с вашими требованиями.

В следующем тексте показан пример запроса для Copilot Chat:

Запрос Copilot

```
Show me how to parse a string as a number, but don't throw an exception if
the input string doesn't represent a number.
```

GitHub Copilot работает на ИИ, поэтому ожидаются сюрпризы и ошибки. Дополнительные сведения см. в часто задаваемых вопросах Copilot .

Дополнительные сведения о GitHub Copilot в Visual Studio и GitHub Copilot в VS Code .

Как преобразовать шестнадцатеричные строки и числовые типы (руководство по программированию на C#)

Статья • 26.03.2025

В этих примерах показано, как выполнить следующие задачи:

- Получите шестнадцатеричное значение каждого символа в строке .
- Получите char, соответствующие каждому значению в шестнадцатеричной строке.
- Преобразуйте шестнадцатеричное `string` в целое число int.
- Преобразовать шестнадцатеричный `string` в float.
- Преобразовать массив в шестнадцатеричное `string`.

Примеры

В этом примере выводится шестнадцатеричное значение каждого символа в `string`. Сначала он анализирует `string` в массив символов. Затем он вызывает ToInt32(Char) на каждом символе, чтобы получить его числовое значение. Наконец, он форматирует число в виде шестнадцатеричного представления в формате `string`.

C#

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
```

```
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

В этом примере анализируется `string` шестнадцатеричных значений и выводится символ, соответствующий каждому шестнадцатеричному значению. Сначала он вызывает метод Split(Char[]), чтобы получить каждое шестнадцатеричное значение в виде отдельного `string` в массиве. Затем он вызывает ToInt32(String, Int32) для преобразования шестнадцатеричного значения в десятичное значение, представленное как int. В нем показаны два разных способа получения символа, соответствующего этому коду символа. Первый метод использует ConvertFromUtf32(Int32), который возвращает символ, соответствующий целочисленному аргументу, в качестве `string`. Второй метод явно приводит `int` к символу.

C#

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
```

```
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

В этом примере показан еще один способ преобразования шестнадцатеричного `string` в целое число путем вызова метода Parse(String, NumberStyles).

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

В следующем примере показано, как преобразовать шестнадцатеричную `string` в float с помощью класса System.BitConverter и метода UInt32.Parse.

C#

```


string hexString = "43480170";
uint num = uint.Parse(hexString,
System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine($"float convert = {f}");

// Output: 200.0056
```

В следующем примере показано, как преобразовать массив байтов в шестнадцатеричную строку с помощью класса System.BitConverter.

C#

```
byte[] vals = [0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD];

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

В следующем примере показано, как преобразовать массив байтов в шестнадцатеричную строку путем вызова метода Convert.ToHexString, введенного в .NET 5.0.

C#

```
byte[] array = [0x64, 0x6f, 0x74, 0x63, 0x65, 0x74];

string hexValue = Convert.ToHexString(array);
Console.WriteLine(hexValue);

/*Output:
  646F74636574
 */
```

См. также

- Стандартные строки в числовом формате
- Типы
- Как определить, представляет ли строка числовое значение

Управление версиями с помощью переопределения и новых ключевых слов (руководство по программированию на C#)

Статья • 09.04.2025

Язык C# разработан таким образом, чтобы версии между базовыми и производными классами в разных библиотеках могли развиваться, сохраняя обратную совместимость. Это означает, например, что введение нового члена в базовый класс с тем же именем, что и член в производном классе, полностью поддерживается C# и не приводит к неожиданному поведению. Это также означает, что класс должен явно указывать, предназначен ли метод для переопределения унаследованного метода или является ли метод новым методом, который скрывает аналогично именованный метод.

В C#производные классы могут содержать методы с тем же именем, что и методы базового класса.

- Если метод в производном классе не предшествует новым или переопределенным ключевым словам, компилятор выдает предупреждение, и метод будет вести себя так, как если `new` бы ключевое слово присутствовало.
- Если метод в производном классе предшествует `new` ключевому слову, метод определяется как независимый от метода в базовом классе.
- Если метод в производном классе предшествует `override` ключевому слову, объекты производного класса вызывают этот метод вместо метода базового класса.
- Чтобы применить ключевое слово `override` к методу в производном классе, метод базового класса должен быть определён как virtual.
- Метод базового класса можно вызвать из производного класса с помощью ключевого `base` слова.
- Ключевые слова `override`, `virtual` и `new` также могут применяться к свойствам, индексаторам и событиям.

По умолчанию методы C# не являются виртуальными. Если метод объявлен как виртуальный, любой класс, наследующий метод, может реализовать собственную версию. Чтобы сделать метод виртуальным, `virtual` модификатор используется в объявлении метода базового класса. Затем производный класс может переопределить базовый виртуальный метод с помощью `override` ключевого слова или скрыть виртуальный метод

в базовом классе с помощью ключевого `new` слова. Если ни ключевое слово `override`, ни ключевое слово `new` не указаны, компилятор выдаст предупреждение, и метод в производном классе скроет метод в базовом классе.

Чтобы продемонстрировать это на практике, предположим, что компания A создала класс с именем `GraphicsClass`, который использует ваша программа. Ниже приведено `GraphicsClass` следующее:

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Ваша компания использует этот класс, и вы используете его для получения собственного класса, добавив новый метод:

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Приложение используется без проблем, пока компания A не выпускает новую версию `GraphicsClass`, которая напоминает следующий код:

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

Новая версия `GraphicsClass` теперь содержит метод с именем `DrawRectangle`. Изначально ничего не происходит. Новая версия по-прежнему бинарно совместима со старой версией. Любая развернутая программа будет работать, даже если новый класс установлен на этих компьютерных системах. Все существующие вызовы метода `DrawRectangle` будут продолжать ссылаться на вашу версию в производном классе.

Однако после повторной компиляции приложения с помощью новой версии `GraphicsClass` вы получите предупреждение от компилятора CS0108. Это предупреждение информирует вас о том, что необходимо обдумать, как вы хотите, чтобы метод `DrawRectangle` вел себя в вашем приложении.

Если вы хотите, чтобы ваш метод переопределял метод нового базового класса, используйте ключевое слово `override`.

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

Ключевое `override` слово гарантирует, что любые объекты, производные от `YourDerivedGraphicsClass`, будут использовать версию класса, полученную от производного `DrawRectangle`. Объекты, производные от `YourDerivedGraphicsClass`, по-прежнему могут получить доступ к версии базового класса с помощью ключевого слова base `DrawRectangle`.

C#

```
base.DrawRectangle();
```

Если вы не хотите, чтобы ваш метод переопределял новый метод базового класса, примените следующие рекомендации. Чтобы избежать путаницы между двумя методами, можно переименовать метод. Это может требовать много времени, нести риск ошибок, а также быть непрактично в некоторых случаях. Однако если проект относительно мал, можно использовать параметры рефакторинга Visual Studio для переименования метода. Дополнительные сведения см. в разделе "Рефакторинг классов и типов" (конструктор классов).

Кроме того, предупреждение можно предотвратить с помощью ключевого слова `new` в определении производного класса:

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

`new` Использование ключевого слова сообщает компилятору, что определение скрывает определение, содержащееся в базовом классе. Это поведение по умолчанию.

Переопределение и выбор метода

При именовавшемся методе в классе компилятор C# выбирает лучший метод для вызова, если с вызовом совместимо несколько методов, например при наличии двух методов с одинаковым именем и параметрами, совместимыми с переданным параметром. Следующие методы будут совместимы:

C#

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

Когда `DoWork` вызывается на экземпляре `Derived`, компилятор C# сначала попытается выполнить вызов, совместимый с версиями `DoWork`, изначально объявленными на `Derived`. Методы переопределения не считаются объявленными в классе, они являются новыми реализациями метода, объявленного в базовом классе. Только если компилятор C# не может сопоставить вызов метода с исходным методом на `Derived`, он попытается сопоставить вызов с переопределенным методом с тем же именем и совместимыми параметрами. Рассмотрим пример.

C#

```
int val = 5;
Derived d = new();
d.DoWork(val);  // Calls DoWork(double).
```

Поскольку переменная `val` может быть неявно преобразована в тип double, компилятор C# вызывает `DoWork(double)` вместо `DoWork(int)`. Это можно избежать двумя способами. Во-первых, не объявляйте новые методы с тем же именем, что и виртуальные методы. Во-вторых, компилятор C# может вызвать виртуальный метод, выполнив поиск в списке методов базового класса путем приведения экземпляра `Derived` в `Base`. Так как метод является виртуальным, будет вызвана реализация `DoWork(int)` на `Derived`. Рассмотрим пример.

C#

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Для получения дополнительных примеров `new` и `override` см. раздел «Когда следует использовать ключевые слова override и new».

См. также

- Система типов C#
- Методы
- Наследование

Знание того, когда следует использовать переопределение и новые ключевые слова (руководство по программированию на C#)

17.06.2025

В C#метод в производном классе может иметь то же имя, что и метод в базовом классе. Вы можете указать, как методы взаимодействуют с помощью новых и переопределения ключевых слов. Модификатор `override` *расширяет* метод базового класса `virtual` , а `new` модификатор *скрывает* доступный метод базового класса. Разница показана в примерах в этом разделе.

В консольном приложении объявите следующие два класса `BaseClass` и `DerivedClass`. `DerivedClass` наследует от `BaseClass`.

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

В методе объявите переменные `Main`, `bc`, `dc` и `bcdc`.

- `bc` имеет тип `BaseClass`, и его значение имеет тип `BaseClass`.
- `dc` имеет тип `DerivedClass`, и его значение имеет тип `DerivedClass`.
- `bcdc` имеет тип `BaseClass`, и его значение имеет тип `DerivedClass`. Это переменная, на которые следует обратить внимание.

Так как `bc` и `bcdc` имеют тип `BaseClass`, они могут обращаться к `Method1` напрямую только в том случае, если вы не используете приведение. Переменная `dc` может получить доступ к обоим `Method1` и `Method2`. Эти связи показаны в следующем коде.

C#

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Затем добавьте следующий `Method2` метод в `BaseClass`. Сигнатура этого метода соответствует сигнатуре `Method2` метода в `DerivedClass`.

C#

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Так как `BaseClass` теперь имеет `Method2` метод, можно добавить вторую вызывающую инструкцию для переменных `BaseClass` `bc` и `bcdc`, как показано в следующем коде.

C#

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

При сборке проекта вы увидите, что добавление метода в `Method2` в `BaseClass` вызывает предупреждение. Предупреждение говорит, что метод `Method2` в `DerivedClass` скрывает метод `Method2` в `BaseClass`. Рекомендуется использовать ключевое `new` слово в `Method2` определении, если вы планируете вызвать этот результат. Кроме того, можно переименовать один из `Method2` методов разрешения предупреждения, но это не всегда практически.

Перед добавлением `new` запустите программу, чтобы просмотреть результаты, созданные дополнительными вызовами. Отображаются следующие результаты.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

Ключевое слово `new` сохраняет связи, которые создают этот результат, но оно подавляет предупреждение. Переменные, имеющие тип `BaseClass`, продолжают получать доступ к членам `BaseClass`, и переменная, которая имеет тип `DerivedClass`, в первую очередь продолжает получать доступ к членам в `DerivedClass`, а затем рассматривает элементы, унаследованные от `BaseClass`.

Чтобы отключить предупреждение, добавьте модификатор `new` в определение `Method2` в `DerivedClass`, как показано в следующем коде. Модификатор можно добавить до или после `public`.

C#

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Запустите программу еще раз, чтобы убедиться, что выходные данные не изменились. Кроме того, убедитесь, что предупреждение больше не отображается. При использовании `new` вы утверждаете, что вы знаете, что элемент, который он изменяет, скрывает элемент, унаследованный от базового класса. Для получения более подробной информации о скрытии имен через наследование см. New Modifier.

Чтобы сравнить это поведение с последствиями использования `override`, добавьте следующий метод в `DerivedClass`. `override` Модификатор можно добавить до или после `public`.

C#

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Добавьте модификатор `virtual` к определению `Method1` в `BaseClass`. `virtual` Модификатор можно добавить до или после `public`.

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Снова запустите проект. Обратите внимание на последние две строки следующих выходных данных.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

Использование `override` модификатора позволяет `bcdc` получить доступ к `Method1` методу, определенному в `DerivedClass`. Как правило, это требуемое поведение в иерархиях наследования. Необходимо, чтобы объекты, имеющие значения, созданные из производного класса, использовали методы, определенные в производном классе. Чтобы достичь этого поведения, используйте `override`, чтобы расширить метод базового класса.

Следующий код содержит полный пример.

C#

```
using System;
using System.Text;
```

```
namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
```

```
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

В следующем примере показано аналогичное поведение в другом контексте. В примере определяются три класса: базовый класс с именем `Car` и двумя классами, производными от него, `ConvertibleCar` и `Minivan`. Базовый `DescribeCar` класс содержит метод. В методе отображается базовое описание автомобиля, а затем вызывается `ShowDetails` для предоставления дополнительных сведений. Каждый из трех классов определяет `ShowDetails` метод. `new` Модификатор используется для определения `ShowDetails` в `ConvertibleCar` классе. `override` Модификатор используется для определения `ShowDetails` в `Minivan` классе.

C#

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}
```

```
// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

В примере проверяется, какая версия `ShowDetails` вызывается. Следующий метод `TestCars1` создаёт экземпляр каждого класса, а затем вызывает `DescribeCar` для каждого экземпляра.

C#

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` создает следующие выходные данные. Особенно обратите внимание на результаты для `car2`, которые, вероятно, не такие, как вы ожидали. Тип объекта — `ConvertibleCar`, но `DescribeCar` не обращается к версии `ShowDetails`, определенной в классе `ConvertibleCar`, так как этот метод объявлен с модификатором `new`, а не с модификатором `override`. В результате объект `ConvertibleCar` отображает то же описание, что и объект `Car`. Сравните результаты для `car3`, который является объектом `Minivan`. В этом случае метод `ShowDetails`, объявленный в классе `Minivan`, переопределяет метод `ShowDetails`, объявленный в классе `Car`, и отображаемое описание минивэна.

C#

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` создает список объектов с типом `Car`. Значения объектов создаются экземплярами классов `Car`, `ConvertibleCar` и `Minivan`. `DescribeCar` вызывается для каждого элемента списка. В следующем коде показано определение `TestCars2`.

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

Отображаются следующие выходные данные. Обратите внимание, что оно совпадает с выходными данными, отображаемыми `TestCars1`. `ShowDetails` метод класса `ConvertibleCar` не вызывается, независимо от того, является ли тип объекта `ConvertibleCar`, как в `TestCars1`, или `Car`, как в `TestCars2`. И наоборот, `car3` вызывает метод `ShowDetails` из класса `Minivan` в обоих случаях, независимо от типа `Minivan` или типа `Car`.

C#

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
```

```
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Методы `TestCars3` и `TestCars4` завершают пример. Эти методы вызывают `ShowDetails` непосредственно из объектов, объявленных типом `ConvertibleCar` и `Minivan` (`TestCars3`), а затем из объектов, объявленных типом `Car` (`TestCars4`). Следующий код определяет эти два метода.

C#

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Методы создают следующие выходные данные, соответствующие результатам первого примера в этом разделе.

C#

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

В следующем коде показан полный проект и его выходные данные.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
```

```
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
```

```
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }
```

```
}
```

См. также

- Система типов C#
- Управление версиями с помощью переопределения и новых ключевых слов
- базовый
- абстрактный

Практическое руководство. Переопределение метода ToString (руководство по программированию на C#)

Статья • 07.04.2023

Каждый класс или структура в языке C# неявно наследует класс Object. Поэтому каждый объект в языке C# получает метод ToString, который возвращает строковое представление данного объекта. Например, все переменные типа `int` имеют метод `ToString`, который позволяет им возвращать их содержимое в виде строки:

```
C#

int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

При создании пользовательского класса или структуры необходимо переопределить метод ToString, чтобы передать информацию о типе клиентскому коду.

Дополнительные сведения об использовании строк форматирования и других типов пользовательского форматирования с методом `ToString` см. в разделе Типы форматирования.

> **ⓘ Важно!**
>
> При принятии решения относительно того, какая информация должна будет предоставляться посредством этого метода, подумайте, будет ли создаваемый класс или структура когда-либо использоваться ненадежным кодом. Постарайтесь не предоставлять информацию, которая может быть использована вредоносным кодом.

Чтобы переопределить метод `ToString` в классе или структуре, выполните указанные ниже действия.

1. Объявите метод `ToString` со следующими модификаторами и типом возвращаемого значения:

```
C#
```

```
public override string ToString(){}
```

2. Реализуйте этот метод таким образом, чтобы он возвращал строку.

   В приведенном ниже примере возвращается не только имя класса, но и специфические данные для конкретного экземпляра класса.

```
C#
```

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

   Метод `ToString` можно проверить с помощью показанного ниже кода.

```
C#
```

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

См. также

- IFormattable
- Руководство по программированию на C#
- Система типов C#
- Строки
- string
- override
- virtual
- Типы форматирования

Участники (руководство по программированию на C#)

13.06.2025

Классы и структуры имеют члены, представляющие свои данные и поведение. Члены класса включают все члены, объявленные в классе, а также все члены (кроме конструкторов и финализаторов), объявленные во всех классах в иерархии наследования. Частные члены в базовых классах наследуются, но недоступны из производных классов.

В следующей таблице перечислены типы элементов класса или структуры:

Развернуть таблицу

Член	Описание
Поля	Поля — это переменные, объявленные в области класса. Поле может быть встроенным числовым типом или экземпляром другого класса. Например, класс календаря может иметь поле, содержащее текущую дату.
Константы	Константы — это поля, значения которых задаются во время компиляции и не могут быть изменены.
Свойства	Свойства — это методы класса, к которому обращается доступ, как если бы они были полями в этом классе. Свойство может обеспечить защиту для поля класса, чтобы он не изменялся без знания объекта.
Методы	Методы определяют действия, которые может выполнить класс. Методы могут принимать параметры, предоставляющие входные данные, и могут возвращать выходные данные с помощью параметров. Методы также могут напрямую возвращать значение без использования параметра.
События	События предоставляют уведомления о вхождениях, таких как нажатия кнопки или успешное завершение метода, другим объектам. События определяются и активируются с помощью делегатов.
Операторы	Перегруженные операторы считаются элементами типа. При перегрузке оператора вы определяете его как публичный метод в типе. Дополнительные сведения см. в разделе " Перегрузка оператора".
Индексаторы	Индексаторы позволяют индексировать объект таким образом, как массивы.
Конструкторы	Конструкторы — это методы, которые вызываются при первом создании объекта. Они часто используются для инициализации данных объекта.
Финализаторы	Финализаторы используются очень редко в C#. Они представляют собой методы, вызываемые подсистемой выполнения среды выполнения, когда объект будет

Член	Описание
	удален из памяти. Они обычно используются, чтобы убедиться, что все ресурсы, которые должны быть освобождены, обрабатываются соответствующим образом.
Вложенные типы	Вложенные типы — это типы, объявленные в другом типе. Вложенные типы часто используются для описания объектов, которые используются только типами, содержащими их.

См. также

- Классы

Абстрактные и запечатанные классы и члены классов (руководство по программированию на C#)

17.06.2025

Абстрактное ключевое слово позволяет создавать классы и члены класса, которые являются неполными и должны быть реализованы в производном классе.

Запечатаемое ключевое слово позволяет предотвратить наследование класса или определенных членов класса, которые ранее были помечены виртуальными.

Абстрактные классы и члены класса

Классы можно объявить как абстрактные, поставив ключевое слово `abstract` перед определением класса. Рассмотрим пример.

C#

```
public abstract class A
{
    // Class members here.
}
```

Создавать экземпляры абстрактного класса нельзя. Цель абстрактного класса — предоставить общее определение базового класса, который могут использовать несколько производных классов. Например, библиотека классов может определить абстрактный класс, используемый в качестве параметра для многих его функций, и требовать, чтобы программисты использовали ее для предоставления собственной реализации класса путем создания производного класса.

Абстрактные классы также могут определять абстрактные методы. Это достигается путем добавления ключевого слова `abstract` до возвращаемого типа метода. Рассмотрим пример.

C#

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Абстрактные методы не имеют реализации, поэтому за определением метода следует точка с запятой вместо обычного блока методов. Производные классы абстрактного класса должны реализовывать все абстрактные методы. Если абстрактный класс наследует виртуальный метод от базового класса, абстрактный класс может переопределить виртуальный метод абстрактным методом. Рассмотрим пример.

C#

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

`virtual` Если метод объявлен `abstract`, он по-прежнему является виртуальным для любого класса, наследуемого от абстрактного класса. Класс, производный от класса с абстрактным методом, `DoWork` не может получить доступ к исходной реализации метода. В предыдущем примере класс F не может вызвать метод `DoWork` в классе D. Таким образом, абстрактный класс может заставить производные классы предоставлять новые реализации для виртуальных методов.

Запечатанные классы и члены класса

Классы можно объявить как запечатанные , поместив ключевое слово `sealed` перед определением класса. Рассмотрим пример.

C#

```
public sealed class D
{
```

```
    // Class members here.
}
```

Запечатанный класс нельзя использовать в качестве базового класса. По этой причине он также не может быть абстрактным классом. Запечатанные классы предотвращают производность. Так как они никогда не могут использоваться в качестве базового класса, некоторые оптимизации во время выполнения могут немного ускорить вызов запечатанных членов класса.

Метод, индексатор, свойство или событие в производном классе, переопределяющий виртуальный член базового класса, может объявить этот элемент как запечатанный. Это отрицает виртуальный аспект члена для любого дальнейшего производного класса. Это достигается путем добавления ключевого слова `sealed` перед ключевым словом переопределение в объявлении члена класса. Рассмотрим пример.

C#

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

См. также

- Система типов C#
- Наследование
- Методы
- Поля
- Определение абстрактных свойств

Статические классы и члены статических классов (руководство по программированию на C#)

13.10.2025

Статический класс аналогичен не статическому классу, за исключением одной разницы: статический класс не может быть инстанцирован. Другими словами, нельзя использовать новый оператор для создания переменной типа класса. Так как переменная экземпляра отсутствует, вы обращаетесь к членам статического класса с помощью самого имени класса. Например, если у вас есть статический класс, который называется `UtilityClass` с открытым статическим методом с именем `MethodA`, вызывается метод, как показано в следующем примере:

C#

```
UtilityClass.MethodA();
```

Статический класс можно использовать в качестве удобного контейнера для наборов методов, которые просто работают с входными параметрами и не должны получать или задавать поля внутреннего экземпляра. Например, в библиотеке классов .NET статический System.Math класс содержит методы, выполняющие математические операции, без каких-либо требований к хранению или извлечению данных, уникальных для определенного экземпляра Math класса. То есть примените элементы класса, указав имя класса и имя метода, как показано в следующем примере.

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Как и в случае со всеми типами классов, среда выполнения .NET загружает сведения о типе статического класса при загрузке программы, ссылающейся на класс. Программа не может точно указать, когда загружается класс. Однако он гарантированно загружает и инициализирует поля и вызывается статическим конструктором до того, как класс

ссылается впервые в вашей программе. Статический конструктор вызывается только один раз, и статический класс остается в памяти в течение всего времени существования домена приложения, в котором находится ваша программа.

> ⓘ **Примечание**
>
> Чтобы создать нестатический класс, где создаётся только один экземпляр, см. реализацию Singleton в C#.

В следующем списке представлены основные функции статического класса:

- Содержит только статические элементы.
- Невозможно создать экземпляр.
- Запечатан.
- Не удается включить конструкторы экземпляров .

Поэтому создание статического класса в основном совпадает с созданием класса, содержащего только статические элементы и закрытый конструктор. Приватный конструктор предотвращает создание экземпляра класса. Преимущество использования статического класса заключается в том, что компилятор может проверить, что элементы экземпляра не добавляются случайно. Компилятор гарантирует, что экземпляры этого класса не могут быть созданы.

Статические классы запечатаны и поэтому не могут быть унаследованы. Они не могут наследоваться от любого класса или интерфейса, кроме Object. Статические классы не могут содержать конструктор экземпляра. Однако они могут содержать статический конструктор. Нестатические классы также должны определять статический конструктор, если класс содержит статические элементы, требующие нетривиальной инициализации. Дополнительные сведения см. в статические конструкторы.

Пример

Ниже приведен пример статического класса, который содержит два метода, которые преобразуют температуру из Цельсия в Fahrenheit и из Fahrenheit в Цельсию:

C#

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius) =>
        double.Parse(temperatureCelsius) * 9 / 5 + 32;
```

```
    public static double FahrenheitToCelsius(string temperatureFahrenheit) =>
        (double.Parse(temperatureFahrenheit) - 32) * 5 / 9;
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                var f =
TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ?? "0");
                Console.WriteLine($"Temperature in Fahrenheit: {f:F2}");
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                var c =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                Console.WriteLine($"Temperature in Celsius: {c:F2}");
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Статические члены

Нестатический класс может содержать статические методы, поля, свойства или события. Статический член может вызываться через класс, даже если экземпляр класса не создан. Статический элемент всегда обращается к имени класса, а не имени экземпляра. Существует только одна копия статического члена независимо от того, сколько экземпляров класса создается. Для универсальных типов каждый закрытый универсальный тип имеет собственную копию статических элементов. Статические поля, помеченные как ThreadStaticAttribute, имеют по одной копии для каждого потока. Статические методы и свойства не могут получить доступ к нестатическому полям и событиям в их содержательном типе, и они не могут получить доступ к переменной экземпляра любого объекта, если он не передается явным образом в параметре метода.

Обычно объявлять нестатический класс с некоторыми статическими элементами, чем объявлять весь класс как статический. Два распространенных использования статических полей — хранить количество объектов, которые были созданы, или хранить значение, общее для всех экземпляров.

Статические методы могут быть перегружены, но не переопределены, так как они принадлежат к классу, а не к любому экземпляру класса.

Хотя поле не может быть объявлено как `static const`, поле констант по сути является статическим в его поведении. Он принадлежит типу, а не экземплярам типа. Таким образом, `const` поля можно получить с помощью той же `ClassName.MemberName` нотации, используемой для статических полей. Экземпляр объекта не требуется.

C# не поддерживает статические локальные переменные (то есть переменные, объявленные в области метода).

Вы объявляете элементы статического класса с помощью ключевого слова `static` перед типом возвращаемого элемента, как показано в следующем примере:

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank => 15;

    public static void Drive() { }

    public static event EventType? RunOutOfGas;
```

```
    // Other non-static fields and properties...
}
```

Статические члены инициализированы перед первым доступом к статическому элементу и до вызова статического конструктора. Чтобы получить доступ к элементу статического класса, используйте имя класса вместо имени переменной, чтобы указать расположение элемента, как показано в следующем примере:

C#

```
Automobile.Drive();
var i = Automobile.NumberOfWheels;
```

Если класс содержит статические поля, укажите статический конструктор, который инициализирует их при загрузке класса.

Вызов статического метода создает инструкцию вызова в общем промежуточном языке (CIL), а вызов метода экземпляра создает инструкцию `callvirt`, которая также проверяет наличие ссылок на объекты NULL. Однако большую часть времени разница между этими двумя производительностью не является значительной.

Спецификация языка C#

Дополнительные сведения см. в статических классах, статических и экземплярных элементах и статических конструкторах в спецификации языка C# . Спецификация языка является авторитетным источником синтаксиса и использования C#.

См. также

- статический
- Классы
- класс
- Статический конструктор
- Конструкторы экземпляров

Модификаторы доступа (руководство по программированию на C#)

Все типы и члены типов имеют уровень доступности. Уровень доступности определяет, может ли другой код в вашей сборке или в других сборках использовать его. Сборка*представляет собой .dll* или *.exe*, созданные путем компиляции одного или нескольких *.cs* файлов в одной компиляции. Следующие модификаторы доступа позволяют указать доступность типа или члена при объявлении:

- public: код в любой сборке может получить доступ к этому типу или члену. Уровень специальных возможностей содержащего типа определяет уровень доступности общедоступных членов типа.
- private: только код, объявленный в том же или `class` `struct` может получить доступ к этому члену.
- защищенный: только код в том же `class` или в производном `class` может получить доступ к этому типу или члену.
- internal: доступ к этому типу или члену может получить только код в одной сборке.
- защищенный внутренний: доступ к этому типу или члену может получить только код в той же сборке *или* в производном классе в другой сборке.
- private protected: только код в одной сборке *и* в том же классе или производном классе может получить доступ к типу или члену.

Используйте модификатор файла для типов верхнего уровня (не вложенных). Модификатор `file` ограничивает доступ к коду в том же исходном файле. Вы не можете объединять модификатор `file` с любым модификатором доступа.

Модификатор record типа приводит компилятору к синтезу дополнительных элементов. Модификатор `record` не влияет на специальные возможности по умолчанию для либо а `record class` `record struct`, либо для .

Сводная таблица

Развернуть таблицу

Расположение вызывающего объекта	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
В классе	✔	✔	✔	✔	✔	✔
Производный класс (та же сборка)	✔	✔	✔	✔	✔	✘

Расположение вызывающего объекта	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Непроизводный класс (та же сборка)	✔	✔	✘	✔	✘	✘
Производный класс (другая сборка)	✔	✔	✔	✘	✘	✘
Непроизводный класс (другая сборка)	✔	✘	✘	✘	✘	✘

В следующих примерах показано, как изменить модификаторы доступа для типа или члена типа:

C#

```
public class Bicycle
{
    public void Pedal() { }
}
```

Не все модификаторы доступа подходят для всех типов или членов во всех контекстах. В некоторых случаях доступность содержащего типа ограничивает доступность его членов.

Несколько объявлений частичного класса или частичного члена должны иметь одинаковые специальные возможности. Если одно объявление не включает модификатор доступа, другие объявления не могут объявлять модификатор доступа. Компилятор создает ошибку, если несколько объявлений для частичного класса или метода объявляют различные специальные возможности.

Доступность классов и структур

Классы и структуры, объявленные непосредственно в пространстве имен (не вложены в другие классы или структуры), могут иметь `public` или `internal` доступ. Если модификатор доступа не указан, используется `internal` значение по умолчанию. Модификатор `file` также можно использовать для ограничения доступа к текущему исходному файлу.

Можно объявить элементы структуры, включая вложенные классы и структуры, как `public` `internal`, или `private`. Можно объявить члены класса, включая вложенные классы и структуры, как `public` `protected internal`, , , `protected` `internal` `private` `protected` или `private`. По умолчанию члены класса и структуры, включая вложенные классы и структуры, имеют `private` доступ.

Производные классы не могут быть более доступны, чем соответствующие базовые типы. Невозможно объявить открытый класс `B`, производный от внутреннего класса `A`. Если бы это было возможно, класс `A` стал бы открытым, так как все члены `protected` или `internal` класса `A` были бы доступны из производного класса.

Доступ к внутренним типам можно предоставить некоторым другим сборкам с помощью `InternalsVisibleToAttribute`. Дополнительные сведения см. в разделе Дружественные сборки.

Другие типы

Интерфейсы можно объявлять непосредственно в пространстве имен как `public` или `internal`. Как классы и структуры, так и интерфейсы по умолчанию имеют `internal` доступ. Члены интерфейса по умолчанию являются `public`, так как интерфейс как раз и создан для того, чтобы обеспечить другим типам доступ к классу или структуре. Объявления членов интерфейса могут включать модификатор доступа. Используйте модификаторы доступа для `interface` членов, чтобы обеспечить общую реализацию, необходимую для всех разработчиков интерфейса.

Тип delegate , объявленный непосредственно в пространстве имен, имеет `internal` доступ по умолчанию.

Дополнительные сведения о модификаторах доступа см. на странице "Уровни специальных возможностей".

Доступность членов

Вы можете объявить члены класса `class` или `struct` (включая вложенные классы и структуры) с любым из шести типов доступа. Нельзя объявлять члены структуры как `protected`, `protected internal` или `private protected`, поскольку структуры не поддерживают наследование.

Как правило, уровень доступности члена не может быть выше уровня доступности типа, в который он входит. Однако можно получить доступ к члену `public` `internal` класса извне сборки, если член реализует методы интерфейса или переопределяет виртуальные методы, определенные в общедоступном базовом классе.

Тип любого поля, свойства или события члена должен иметь по меньшей мере такой же уровень доступности, как у самого члена. Точно так же тип возвращаемого значения и типы параметров любого метода, индексатора или делегата должен иметь по меньшей мере такой же уровень доступности, как у самого члена. Например, невозможно иметь

метод `public M`, возвращающий класс `C`, если только `C` также не является `public`. Аналогичным образом, невозможно иметь свойство `protected` типа `A`, если `A` объявлен как `private`.

Необходимо всегда объявлять определяемые пользователем операторы как `public`. Для получения дополнительной информации см. раздел Перегрузка операторов.

Чтобы настроить уровень доступа для члена `class` или `struct`, добавьте в объявление этого члена соответствующее ключевое слово, как показано в следующем примере.

C#

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int _wheels = 3;

    // protected internal property:
    protected internal int Wheels => _wheels;
}
```

Методы завершения не могут иметь модификаторы доступа. Члены типа `enum` всегда являются `public`, и вы не можете применять модификаторы доступа.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Укажите порядок модификаторов (правило стиля IDE0036)
- Система типов C#
- Интерфейсы
- Уровни доступности
- частный
- общественный
- внутренний
- защищено
- защищенный внутренний

- закрытый защищенный доступ
- запечатанный
- класс
- структура
- интерфейс
- Анонимные типы

Last updated on 15.12.2025

Поля (руководство по программированию на C#)

17.06.2025

Поле представляет собой переменную любого типа, объявленного непосредственно в классе или структуре. Поля являются *элементами* их содержащего типа.

У класса или структуры могут быть поля экземпляров, статические поля или оба. Поля экземпляра характерны для конкретного экземпляра типа. Если у вас есть класс `T` с полем `F` экземпляра, можно создать два объекта типа `T` и изменить значение каждого объекта, не влияя на значение `F` в другом объекте. Напротив, статическое поле принадлежит самому типу и является общим для всех экземпляров этого типа. Доступ к статическому полю можно получить только с помощью имени типа. При доступе к статическому полю по имени экземпляра возникает ошибка времени компиляции CS0176 .

Вообще, следует указать `private` или `protected` доступность для полей. Данные, предоставляемые типом клиентскому коду, должны предоставляться с помощью методов, свойств и индексаторов. Используя эти конструкции для косвенного доступа к внутренним полям, можно защититься от недопустимых входных значений. Частное поле, которое хранит данные, предоставляемые общедоступным свойством, называется *резервным хранилищем* или *резервным полем*. Вы можете объявить `public` поля, но затем нельзя запретить коду, использующего тип, задать это поле в недопустимое значение или изменить данные объекта.

Поля обычно хранят данные, которые должны быть доступны для нескольких методов типа и должны храниться дольше, чем время существования любого одного метода. Например, тип, представляющий дату календаря, может иметь три целых поля: один для месяца, один за день и один в течение года. Переменные, которые не используются вне области одного метода, должны быть *объявлены как локальные переменные* в самом теле метода.

Поля объявляются в блоке класса или структуры, указав уровень доступа, за которым следует тип, а затем имя поля. Рассмотрим пример.

C#

```
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
```

```
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Date");
            }
        }
    }

    // public field (Generally not recommended).
    public string? Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }
}
```

Чтобы получить доступ к полю в экземпляре, добавьте период после имени экземпляра, а затем имя поля, как в `instancename._fieldName`. Рассмотрим пример.

```
C#

CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

Поле можно указать начальное значение с помощью оператора назначения при объявлении поля. Чтобы автоматически назначить поле `Day` к `"Monday"`, например, вы объявляете `Day`, как показано в следующем примере:

```
C#

public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Поля инициализированы непосредственно перед вызовом конструктора для экземпляра объекта. Если конструктор назначает значение поля, он перезаписывает любое значение, заданное во время объявления поля. Дополнительные сведения см. в разделе "Использование конструкторов".

> ⓘ **Примечание**
>
> Инициализатор полей не может ссылаться на другие поля экземпляра.

Поля можно пометить как public, private, , protected, internalprotected internalили private protected. Эти модификаторы доступа определяют, как пользователи типа могут получить доступ к полям. Дополнительные сведения см. в статье Модификаторы доступа.

Поле необязательно можно объявить static. Статические поля доступны вызывающим в любое время, даже если не существует ни одного экземпляра этого типа. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Поле можно объявить readonly. Поле с доступом только для чтения может быть присвоено значение только во время инициализации или в конструкторе. `static readonly` Поле похоже на константу, за исключением того, что компилятор C# не имеет доступа к значению статического поля только для чтения во время компиляции, только во время выполнения. Дополнительные сведения см. в разделе "Константы".

Поле можно объявить required. Обязательное поле должно быть инициализировано конструктором или инициализаторами объектов при создании объекта. Вы добавляете атрибут System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute к любому объявлению конструктора, которое инициализирует все необходимые члены.

`required` Модификатор не может сочетаться с модификатором `readonly` в том же поле. Однако свойство может быть `required` и `init` только.

Начиная с C# 12, параметры первичного конструктора являются альтернативой объявлению полей. Если тип имеет зависимости, которые должны быть предоставлены при инициализации, можно создать первичный конструктор, предоставляющий эти зависимости. Эти параметры могут быть записаны и использованы вместо объявленных полей в типах. В случае типов параметры первичного конструктора record представлены как общедоступные свойства.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

См. также

- Система типов C#
- Использование конструкторов
- Наследование
- Модификаторы доступа
- Абстрактные и закрытые классы и члены классов

Константы (руководство по программированию на C#)

17.06.2025

Константы — это неизменяемые значения, известные во время компиляции и не изменяющиеся для жизни программы. Константы объявляются с модификатором константа . Только встроенные типы C# могут быть объявлены как `const`. Константы ссылочного типа, кроме String, могут быть инициализированы только значением null. Определяемые пользователем типы, включая классы, структуры и массивы, не могут быть `const`. Используйте модификатор только для чтения для создания класса, структуры или массива, который инициализируется один раз в момент выполнения (например, в конструкторе) и после этого не может быть изменен.

C# не поддерживает `const` методы, свойства или события.

Тип перечисления позволяет определять именованные константы для целых встроенных типов (например, `int`, `uint`, `long` и т. д.). Для получения дополнительной информации см. enum.

Константы должны быть инициализированы по мере их объявления. Рассмотрим пример.

C#

```
class Calendar1
{
    public const int Months = 12;
}
```

В этом примере константой `Months` всегда является 12, и она не может быть изменена даже самим классом. На самом деле, когда компилятор обнаруживает постоянный идентификатор в исходном коде C# (например, `Months`), он заменяет литеральное значение непосредственно на код промежуточного языка (IL), который он создает. Так как во время выполнения нет адреса переменной, связанного с константой, `const` поля не могут передаваться по ссылке и не могут отображаться как l-значение в выражении.

> ⓘ **Примечание**
>
> Будьте осторожны, когда вы ссылаетесь на константные значения, определенные в другом коде, таких как DLL. Если новая версия библиотеки DLL определяет новое

значение для константы, программа по-прежнему будет хранить старое литеральное значение, пока оно не будет перекомпилировано в новой версии.

Одновременно можно объявить несколько констант одного типа, например:

C#

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

Выражение, используемое для инициализации константы, может ссылаться на другую константу, если она не создает циклическую ссылку. Рассмотрим пример.

C#

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Константы можно пометить как общедоступные, частные, защищенные, внутренние, защищенные внутренние или закрытые. Эти модификаторы доступа определяют, как пользователи класса могут получить доступ к константе. Дополнительные сведения см. в статье Модификаторы доступа.

Доступ к константам осуществляется так, как если бы они были статическими полями, так как значение константы совпадает со всеми экземплярами типа. Вы не используете ключевое `static` слово для их объявления. Выражения, которые не находятся в классе, определяющем константу, должны использовать имя класса, период и имя константы для доступа к константе. Рассмотрим пример.

C#

```
int birthstones = Calendar.Months;
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

См. также

- Свойства
- Типы
- только для чтения
- Неизменяемость в C#, часть одна: типы неизменяемости

Практическое руководство. Определение абстрактных свойств (руководство по программированию на C#)

Статья • 12.03.2024

В следующем примере показано, как определять абстрактные свойства. В объявлении абстрактного свойства не предоставляется реализация методов доступа к свойству. В нем объявляется, что класс поддерживает свойства, однако реализация методов доступа к ним передается в производные классы. В следующем примере показано, как реализовать абстрактные свойства, наследуемые от базового класса.

Этот пример включает три файла, каждый из которых компилируется отдельно в сборку, на которую задаются ссылки при последующей компиляции:

- abstractshape.cs: класс `Shape`, который содержит абстрактное свойство `Area`.
- shapes.cs: подклассы класса `Shape`.
- shapetest.cs: тестовая программа для отображения областей некоторых объектов, производных от `Shape`.

Чтобы скомпилировать этот пример, используйте следующую команду:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

При этом будет создан исполняемый файл shapetest.exe.

Примеры

В этом файле объявляется класс `Shape`, который содержит свойство `Area` типа `double`.

C#

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;
```

```
    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Модификаторы свойства помещаются в само объявление свойства. Например:

  ```
  C#

  public abstract double Area
  ```

- При объявлении абстрактного свойства, такого как `Area` в этом примере, вы просто указываете используемые методы доступа, но не реализуете их. В этом примере используется только метод доступа get, поэтому свойство будет доступно только для чтения.

В следующем коде представлены три подкласса класса `Shape` и демонстрируется, как они переопределяют свойство `Area` для получения его собственной реализации.

```
C#
```

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
```

```
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

В следующем коде показана тестовая программа, которая создает несколько производных от `Shape` объектов и печатает их области.

C#

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

См. также

- Система типов C#
- Абстрактные и запечатанные классы и члены классов
- Свойства

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Определение констант в C#

Статья • 07.04.2023

Константы — это поля, значения которых устанавливаются во время компиляции и не изменяются. С помощью констант можно присвоить особым значениям значащие имена вместо числовых литералов.

> ⓘ **Примечание**
>
> В C# с помощью директивы препроцессора **#define** нельзя определять константы так, как это было реализовано в C и C++.

Чтобы определить значения константы целочисленного типа (`int`, `byte` и т. д.), используйте перечисляемый тип. Дополнительные сведения см. в разделе Перечисление.

Чтобы определить нецелочисленные константы, можно сгруппировать их в статический класс с именем `Constants`. В этом случае перед любыми ссылками на константы будет необходимо указывать имя класса, как показано в следующем примере.

Пример

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

Используя квалификатор имени класса, вы гарантируете, что вы сами и другие разработчики будете понимать, что имеете дело с константой, которую нельзя изменить.

См. также

- Система типов C#

Свойства (Руководство по программированию в C#)

Свойство — это элемент, предоставляющий гибкий механизм для чтения, записи или вычисления значения поля данных. Свойства отображаются как открытые элементы данных, но они реализуются как специальные методы, называемые *аксессорами*. Эта функция позволяет вызывающим пользователям легко получать доступ к данным и по-прежнему способствовать обеспечению безопасности и гибкости данных. Синтаксис свойств является естественным расширением полей. Поле определяет место хранения:

C#

```
public class Person
{
    public string? FirstName;

    // Omitted for brevity.
}
```

Автоматически реализованные свойства

Определение свойства содержит объявления для методов доступа `get` и `set`, которые получают и устанавливают значение этого свойства:

C#

```
public class Person
{
    public string? FirstName { get; set; }

    // Omitted for brevity.
}
```

В предыдущем примере показано *автоматически реализованное свойство*. Компилятор создает скрытое поле резервного копирования для свойства. Компилятор также реализует тело аксессоров `get` и `set`. Все атрибуты применяются к автоматически реализуемому свойству. Атрибут можно применить к поле резервной копии, созданному компилятором, указав `field:` тег атрибута.

Можно инициализировать свойство до значения, отличного от значения по умолчанию, задав значение после закрывающей фигурной скобки свойства. Вы можете предпочесть, чтобы начальным значением свойства `FirstName` была пустая строка, чем `null`. Как показано в следующем коде, это можно указать следующим образом:

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

Свойства, поддерживаемые полем

В C# 14 вы можете добавить проверку или другую логику в аксессор свойства с помощью ключевого слова field. Ключевое `field` слово обращается к синтезированному полю резервного копирования компилятора для свойства. Он позволяет создавать метод доступа к свойствам без явного объявления отдельного поля резервного копирования.

C#

```
public class Person
{
    public string? FirstName
    {
        get;
        set => field = value.Trim();
    }

    // Omitted for brevity.
}
```

Обязательные свойства

В приведенном выше примере вызывающий объект может создать экземпляр `Person` с использованием конструктора по умолчанию, не устанавливая свойство `FirstName`. Свойство изменило тип на строку, которая может быть null. Для задания свойства можно *требовать, чтобы* вызывающие пользователи могли задать свойство:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }
```

```
    // Omitted for brevity.
}
```

Предыдущий код вносит два изменения в `Person` класс. Во-первых, `FirstName` объявление свойства включает `required` модификатор. Это означает, что любой код, создающий новое `Person` свойство, должен задать это свойство с помощью инициализатора объектов. Во-вторых, конструктор, принимающий `firstName` параметр, имеет System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute атрибут. Этот атрибут сообщает компилятору, что этот конструктор задает *все* `required` элементы. Вызывающим этот конструктор не обязательно использовать инициализатор объектов для задания свойств `required`.

> **ⓘ Важный**
>
> Не путайте `required` с *ненулевой*. Допустимо задать свойству `required` значение `null` или `default`. Если тип не допускает значение NULL, например `string` в этих примерах, компилятор выдает предупреждение.

C#

```
var aPerson = new Person("John");
aPerson = new Person { FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//aPerson2 = new Person();
```

Определения текста выражений

Методы доступа к свойствам часто состоят из однострочных инструкций. Методы доступа присваивают или возвращают результат выражения. Эти свойства можно реализовать как члены, воплощающие выражение. Определения тела выражения состоят из маркера `=>`, за которым следует выражение, которое назначается свойству или извлекается из него.

Свойства, доступные только для чтения, могут реализовать `get` аксессор в виде выражения. В следующем примере реализуется свойство только для `Name` чтения в качестве элемента, наследуемого выражением:

C#

```
public class Person
{
    public Person() { }
```

```
    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    public string Name => $"{FirstName} {LastName}";

    // Omitted for brevity.
}
```

`Name` — вычисляемое свойство. Нет резервного поля для `Name`. Свойство вычисляет его каждый раз.

Управление доступом

В предыдущих примерах показаны свойства чтения и записи. Вы также можете создать свойства только для чтения или задать разные уровни доступа для методов доступа set и get. Предположим, что класс `Person` должен позволять изменять значение свойства `FirstName` только в других методах класса. Вы можете предоставить доступность аксессора set для `private`, вместо `internal` или `public`.

C#

```
public class Person
{
    public string? FirstName { get; private set; }

    // Omitted for brevity.
}
```

Свойство `FirstName` можно считывать из любого кода, но его можно назначить только из кода в `Person` классе.

Вы можете добавить любой ограничивающий модификатор доступа к аксессорам set или get. Модификатор доступа для отдельного метода доступа должен быть более строгим, чем доступ к свойству. Предыдущий код является законным, так как свойство `FirstName` является `public`, но свойство set является `private`. Нельзя объявить `private` свойство с `public` аксессором. Свойство также можно объявить как `protected`, `internal`, `protected internal` или даже `private`.

Существует два специальных модификатора доступа для `set` аксессоров:

- Аксессор `set` может использовать `init` как модификатор доступа. Этот `set` аксессор можно вызывать только из инициализатора объекта или конструкторов типа. Это более строго, чем `private` на `set` аксессоре.
- Автоматически реализованное свойство может объявлять `get` аксессор без `set` аксессора. В этом случае компилятор позволяет вызывать аксессор только из конструкторов типа. Это более ограничительно, чем `init` модификатор доступа на `set`.

Измените `Person` класс следующим образом:

C#

```
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

В предыдущем примере требуется, чтобы вызывающие использовали конструктор, включающий параметр `FirstName`. Вызывающие не могут использовать инициализаторы объектов для назначения значения свойству. Для поддержки инициализаторов можно сделать аксессор `set` аксессором `init`, как показано в следующем коде:

C#

```
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

Эти модификаторы часто используются с модификатором `required` для принудительной инициализации.

Свойства с резервными полями

Вы можете смешать концепцию вычисляемого свойства с частным полем и создать кэшированное *вычисляемое свойство*. Например, обновите `FullName` свойство таким образом, чтобы строковое форматирование происходило при первом доступе:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

Эта реализация работает, потому что свойства `FirstName` и `LastName` доступны только для чтения. Люди могут изменить свое имя. Обновление свойств `FirstName` и `LastName` для разрешения `set` доступа требует аннулирования любого кэшированного значения для `fullName`. Вы изменяете `set` и `FirstName` методы доступа свойства `LastName`, чтобы `fullName` поле вычислялось снова:

C#

```
public class Person
{
    private string? _firstName;
    public string? FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
```

```
        }
    }

    private string? _lastName;
    public string? LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

Эта окончательная версия вычисляет свойство `FullName` только при необходимости. Если это допустимо, используется ранее вычисляемая версия. В противном случае вычисление обновляет кэшированное значение. Разработчикам, использующим этот класс, не нужно знать подробности реализации. Ни одно из этих внутренних изменений не влияет на использование объекта Person.

Начиная с C# 13, можно создавать partial свойства в `partial` классах. Объявление реализации для `partial` свойства не может быть автоматически реализованным свойством. Автоматически реализованное свойство использует тот же синтаксис, что и объявление частичного свойства.

Свойства

Свойства — это своего рода интеллектуальные поля в классе или объекте. Из-за пределов объекта они представляются полями в объекте. Однако для реализации свойства можно использовать полную палитру функциональных возможностей C#. Вы можете предоставлять проверку правильности, различные возможности доступа, отложенное вычисление или любые другие требования, необходимые в ваших сценариях.

- Простые свойства, для которых не требуется пользовательский код доступа, можно реализовать как определения текста выражения или как автоматически

реализованные свойства.

- Свойства позволяют классу предоставлять общий способ получения и задания значений, скрывая при этом код реализации или проверки.
- Метод доступа get используется для возврата значения свойства, а метод доступа set — для присвоения нового значения. Метод доступа к свойству init используется для назначения нового значения только во время построения объекта. Эти акцессоры могут иметь различные уровни доступа. Дополнительные сведения см. в разделе Ограничение доступности аксессоров.
- Ключевое слово value используется для определения значения, которое назначается методом доступа `set` или `init`.
- Свойства могут быть доступны для *чтения и записи* (они имеют оба метода доступа — `get` и `set`), *только для чтения* (они имеют метод доступа `get`, но не имеют метода доступа `set`) или *только для записи* (они имеют метод доступа `set`, но не имеют метода доступа `get`). Свойства, доступные только для записи, редки.

Спецификация языка C#

Дополнительные сведения см. в разделе Свойства в Спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования C#.

См. также

- Индексаторы
- ключевое слово init
- получить ключевое слово
- установить ключевое слово

Last updated on 18.11.2025

Использование свойств (Руководство по программированию в C#)

Свойства сочетают в себе возможности полей и методов. Для пользователя объекта свойство, как представляется, является полем; для доступа к свойству требуется тот же синтаксис. Для разработчика класса свойство состоит из одного или двух блоков кода, представляющих get метод доступа и/или метод доступа set или init. Блок кода для аксессора `get` выполняется при чтении свойства; блок кода для аксессоров `set` или `init` выполняется при назначении свойству значения. Свойство без аксессора `set` считается доступным только для чтения. Свойство без метода доступа `get` считается доступным только для записи. Свойство, у которого имеются оба аксессора, является доступным для чтения и записи. Вы можете использовать `init` метод доступа вместо `set` метода доступа, чтобы свойство было задано как часть инициализации объектов, но в противном случае сделайте его доступным только для чтения.

В отличие от полей, свойства не классифицируются как переменные. Таким образом, нельзя передать свойство в качестве ref или out параметра.

Свойства имеют множество вариантов использования:

- Перед разрешением изменения они могут проверить данные.
- Они могут прозрачно предоставлять данные в классе, где эти данные извлекаются из другого источника, например базы данных.
- Они могут выполнять действия при изменении данных, таких как создание события или изменение значения других полей.

При объявлении свойств в блоке класса сначала указывается уровень доступа, затем тип и имя свойства, после чего следует блок кода, в котором объявляется аксессор `get` и/или аксессор `set`. Например:

C#

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
```

```
        }
    }
}
```

В этом примере `Month` объявляется как свойство, а метод доступа `set` обеспечивает установку значения `Month` в диапазоне от 1 до 12. Для отслеживания фактического значения свойство `Month` использует частное поле. Реальное расположение данных свойства часто называется "резервным хранилищем". Обычно свойства используют частные поля в качестве резервного хранилища. Поле помечается как частное, чтобы гарантировать, что его можно изменить только через вызов свойства. Дополнительные сведения об ограничениях открытого и закрытого доступа см. в разделе Модификаторы доступа. Автоматически реализованные свойства предоставляют упрощенный синтаксис для простых объявлений свойств. Дополнительные сведения см. в разделе "Автоматически реализованные свойства".

Начиная с C# 14, вы можете использовать свойства с резервной функцией полей для введения проверки в `set` аксессор автоматически реализуемого свойства, как показано в следующем примере:

C#

```
public class DateExample
{
    public int Month
    {
        get;
        set
        {
            if ((value > 0) && (value < 13))
            {
                field = value;
            }
        }
    }
}
```

Аксессор get

Тело аксессора `get` похоже на тело метода. Оно должно возвращать значение заданного типа свойства. Компилятор C# и JIT-компилятор обнаруживают распространенные шаблоны для реализации `get` аксессора и оптимизируют эти шаблоны. Например, метод доступа, возвращающий поле без выполнения каких-либо вычислений, скорее всего, оптимизируется для чтения памяти этого поля `get`. Автоматически реализованные свойства соответствуют этому шаблону и получают преимущества от этих оптимизаций.

Однако метод виртуального `get` доступа не может быть встроен, поскольку компилятор не знает во время компиляции, какой метод может вызываться в момент выполнения. В следующем примере показан аксессор `get`, который возвращает значение закрытого поля `_name`.

C#

```
class Employee
{
    private string _name;           // the name field
    public string Name => _name;    // the Name property
}
```

При ссылке на свойство (кроме случаев, когда свойство является целью присваивания) вызывается метод доступа `get`, который считывает значение свойства. Например:

C#

```
var employee = new Employee();
//...

System.Console.Write(employee.Name); // the get accessor is invoked here
```

Метод `get` доступа должен быть членом, реализованным в виде выражения, или заканчиваться оператором return или throw, и выполнение не может выйти за пределы тела метода доступа.

> **⚠ Предупреждение**
>
> Обычно плохой стиль программирования — изменять состояние объекта через `get` акцессор. Одно из исключений этому правилу — *лениво вычисляемое* свойство, когда значение свойства вычисляется только при первом доступе.

Метод доступа `get` можно использовать для получения значения поля либо напрямую, либо после его вычисления. Например:

C#

```
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

В предыдущем примере, если значение свойства не назначено `Name` , возвращается значение `NA`.

Акцессор set

Аксессор `set` напоминает метод с типом возвращаемого значения void. В нем используется неявный параметр `value`, тип которого соответствует типу свойства. Компилятор и компилятор JIT также распознают распространенные шаблоны для `set` или `init` аксессора. Эти распространенные шаблоны оптимизированы для прямой записи в память резервного поля. В следующем примере метод доступа `set` добавляется к свойству `Name`:

C#

```
class Student
{
    private string _name;       // the name field
    public string Name          // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

При присвоении значения свойству вызывается метод доступа `set` с аргументом, содержащим новое значение. Например:

C#

```
var student = new Student();
student.Name = "Joe"; // the set accessor is invoked here

System.Console.Write(student.Name); // the get accessor is invoked here
```

Ошибка заключается в использовании неявного имени параметра `value` для объявления локальной переменной в аксессоре `set`.

Аксессор init

Код для создания метода доступа `init` аналогичен коду для создания метода доступа `set`, за исключением того, что используется ключевое слово `init` вместо `set`. Различие заключается в том, что метод доступа `init` можно использовать только в конструкторе или с помощью инициализатора объекта.

Замечания

Свойства могут быть помечены как `public`, `private`, `protected`, `internal`, `protected internal` или `private protected`. Эти модификаторы доступа определяют, каким образом пользователи класса смогут получать доступ к свойству. Аксессоры `get` и `set` для той же свойства могут иметь разные модификаторы доступа. Например, `get` может быть `public`, чтобы разрешить доступ только для чтения извне типа, и `set` может быть `private` или `protected`. Дополнительные сведения см. в статье Модификаторы доступа.

Свойство можно объявить как статическое свойство с помощью ключевого `static` слова. Статические свойства доступны вызывающим объектам в любое время, даже при отсутствии экземпляра класса. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Свойство можно пометить как виртуальное свойство с помощью виртуального ключевого слова. Виртуальные свойства позволяют производным классам изменять поведение свойства, используя ключевое слово override. Дополнительные сведения об этих параметрах см. в разделе Наследование.

Свойство, переопределяющее виртуальное свойство, также можно запечатывать, указывая, что для производных классов он больше не является виртуальным. Наконец, свойство можно объявить абстрактным (abstract). Абстрактные свойства не определяют реализацию в классе, а производные классы должны писать собственную реализацию. Дополнительные сведения об этих параметрах см. в разделе Абстрактные и запечатанные классы и члены классов.

> ⓘ **Заметка**
>
> Это ошибка использования **виртуального**, **абстрактного** или **переопределения** модификатора для метода доступа **статического** свойства.

Примеры

В этом примере демонстрируются свойства экземпляра, а также статические и доступные только для чтения свойства. Этот метод принимает введенное с клавиатуры имя сотрудника, увеличивает значение `NumberOfEmployees` на 1, после чего отображает имя и номер сотрудника.

```
C#
```

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's
number:
}
```

Пример скрытого свойства

В этом примере показано, как получить доступ к свойству в базовом классе, скрытому другим свойством, которое имеет то же имя в производном классе:

C#

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
```

```
{
    public static void Test()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine($"Name in the derived class is: {m1.Name}");
        System.Console.WriteLine($"Name in the base class is:
{((Employee)m1).Name}");
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

На что следует обратить внимание в предыдущем примере:

- Свойство `Name` в производном классе скрывает свойство `Name` базового класса. В таком случае в объявлении свойства в производном классе используется модификатор `new`:

  C#

  ```
  public new string Name
  ```

- Для доступа к скрытому свойству в базовом классе используется приведение типа `(Employee)`:

  C#

  ```
  ((Employee)m1).Name = "Mary";
  ```

Дополнительную информацию о скрытии элементов см. в разделе Модификатор new.

Пример переопределения свойства

В этом примере два класса (`Cube` и `Square`) реализуют абстрактный класс `Shape` и переопределяют его абстрактное свойство `Area`. Обратите внимание на использование модификатора override в свойствах. Программа принимает введенную длину стороны, на основании которой рассчитывает площади квадрата и куба. Также принимается

введенное значение площади, на основании которой рассчитываются длины сторон квадрата и куба.

C#

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);
```

```
        // Display the results:
        System.Console.WriteLine($"Area of the square = {s.Area:F2}");
        System.Console.WriteLine($"Area of the cube = {c.Area:F2}");
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine($"Side of the square = {s.side:F2}");
        System.Console.WriteLine($"Side of the cube = {c.side:F2}");
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

См. также

- Свойства
- Свойства интерфейса
- Автоматически реализованные свойства
- Частичные свойства

Last updated on 19.11.2025

Свойства интерфейса (Руководство по программированию на C#)

Статья • 26.10.2024

Свойства можно объявлять для интерфейса. Следующий пример объявляет метод доступа к свойству интерфейса:

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Свойства интерфейса обычно не имеют тела. Методы доступа указывают, доступно ли свойство для чтения и записи, только для чтения или только для записи. В отличие от классов и структур, объявление методов доступа без текста не объявляется автоматически реализованным свойством. Интерфейс может определить реализацию по умолчанию для членов, включая свойства. Определение реализации по умолчанию для свойства в интерфейсе редко, так как интерфейсы не могут определять поля данных экземпляра.

Пример

В этом примере интерфейс `IEmployee` содержит доступное для чтения и записи свойство `Name`, а также свойство `Counter`, предназначенное только для чтения. Класс `Employee` реализует интерфейс `IEmployee` и использует эти два свойства. Программа считывает имя нового сотрудника и текущее число сотрудников, после чего отображает имя сотрудника и его рассчитанный номер.

Вы можете использовать полное имя свойства, которое ссылается на интерфейс, где был объявлен член. Например:

C#

```
string IEmployee.Name
{
```

```
    get { return "Employee Name"; }
    set { }
}
```

В предыдущем примере показана явная реализация интерфейса. Например, если класс `Employee` реализует два интерфейса `ICitizen` и `IEmployee` оба интерфейса имеют `Name` свойство, необходимо явное внедрение члена интерфейса. Это значит, что потребуется следующее объявление свойства:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

`Name` Реализует свойство в интерфейсе `IEmployee`, а следующее объявление:

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

`Name` Реализует свойство в интерфейсе `ICitizen`.

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;
```

```
    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

C#

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

Пример полученных результатов

Консоль

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

См. также

- Свойства
- Использование свойств
- Сравнение свойств и индексаторов
- Индексаторы

- Интерфейсы

Ограничение доступности методов доступа (Руководство по программированию на C#)

Части get и set свойства или индексатора называются *методами доступа*. По умолчанию они имеют такие же уровни видимости или доступа, что и свойство или индексатор, которым они принадлежат. Дополнительные сведения см. в разделе Уровни доступа. Однако иногда полезно ограничить доступ к одному из этих методов доступа. Как правило, вы ограничиваете доступность `set` метода доступа, делая метод `get` доступа общедоступным. Например:

C#

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

В этом примере свойство `Name` определяет методы доступа `get` и `set`. Метод доступа `get` получает уровень доступа самого свойства `public` в этом случае, а к методу `set` доступ явно ограничивается установкой модификатора доступа protected.

> ⓘ **Примечание.**
>
> Примеры, приведенные в этой статье, не используют автоматически реализованные свойства. *Автоматически реализованные свойства* предоставляют краткий синтаксис для объявления свойств, если настраиваемое поле резервного копирования не требуется.

Ограничения модификаторов доступа для аксессоров

При использовании модификаторов доступа для свойств или индексаторов необходимо соблюдать следующие ограничения:

- Модификаторы доступа нельзя использовать для интерфейса или явной реализации элемента интерфейса.
- Модификаторы доступа можно использовать только в том случае, если для свойства или индексатора определены одновременно методы доступа `set` и `get`. В этом случае модификатор может применяться только к одному из двух методов доступа.
- Если свойству или индексатору назначен модификатор override, модификатор аксессора должен соответствовать модификатору переопределяемого аксессора, если такой существует.
- Уровень доступности аксессора должен быть более строгим по сравнению с уровнем доступа самого свойства или индексатора.

Модификаторы доступа при переопределении аксессоров

При переопределении свойства или индексатора переопределенные методы доступа должны быть доступны коду переопределения. Кроме того, доступность свойства/индексатора и их аксессоров должна соответствовать доступности соответствующих переопределенных свойства/индексатора и их аксессоров. Например:

C#

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Реализация интерфейсов

Используя аксессор для реализации интерфейса, аксессор может не иметь модификатор доступа. Тем не менее при реализации интерфейса с использованием одного метода доступа (например, `get`) другой метод доступа может иметь модификатор доступа, как показано в следующем примере:

C#

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Домен доступности акцессоров

При использовании модификатора доступа для аксессора этот модификатор определяет область доступности для этого аксессора.

Если для аксессора не использовался модификатор доступа, то его уровень доступа определяется уровнем доступности свойства или индексатора.

Пример

В следующем примере определяются три класса: `BaseClass`, `DerivedClass` и `MainClass`. На обоих классах имеются свойства `BaseClass`, `Name` и `Id`. В этом примере демонстрируется, как свойство `Id` класса `DerivedClass` может быть скрыто свойством `Id` класса `BaseClass`

при использовании ограничивающего модификатора доступа, такого как protected или private. Таким образом, при присвоении значений этому свойству вместо него вызывается свойство класса `BaseClass`. Чтобы сделать это свойство доступным, необходимо заменить модификатор доступа на public.

В этом примере также показано, что ограничивающий модификатор доступа, такой как `private` или `protected`, на методе доступа свойству `Name` в `DerivedClass` предотвращает доступ к этому методу в производном классе. При присваивании ему значения или доступе к свойству базового класса с тем же именем, если оно доступно, генерируется ошибка.

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
```

```
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Комментарии

Обратите внимание, что если заменить объявление `new private string Id` на `new public string Id`, будут получены следующие выходные данные:

`Name and ID in the base class: Name-BaseClass, ID-BaseClass` `Name and ID in the derived class: John, John123`

См. также

- Свойства
- Индексаторы
- Модификаторы доступа
- Свойства только для инициализации
- Обязательные свойства

Last updated on 24.02.2026

Объявление и использование свойств записи чтения (руководство по программированию на C#)

Статья • 08.04.2025

Свойства предоставляют удобство использования общедоступных элементов данных без рисков, связанных с незащищённым, неконтролируемым и непроверенным доступом к данным объекта. Свойства объявляют *методы доступа*: специальные методы, которые назначают и извлекают значения из базового элемента данных. Аксессор set позволяет назначать поля данных, а аксессор get получает значения полей данных.

В этом примере показан класс `Person` с двумя свойствами: `Name` (строка) и `Age` (int). Оба свойства предоставляют `get` и `set` методы доступа, поэтому они считаются свойствами чтения и записи.

Пример

C#

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
```

```
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }

}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine($"Person details - {person}");

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine($"Person details - {person}");

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine($"Person details - {person}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
```

```
    Person details - Name = Joe, Age = 100
*/
```

Надежное программирование

В предыдущем примере свойства `Name` и `Age` являются общедоступными и включают как `get` аксессор, так и `set` аксессор. Общедоступные методы доступа позволяют любому объекту считывать и записывать эти свойства. Однако иногда желательно исключить один из методов доступа. Чтобы сделать свойство доступным только для чтения, можно опустить аксессор `set`.

C#

```
public string Name
{
    get
    {
        return _name;
    }
    private set
    {
        _name = value;
    }
}
```

Кроме того, вы можете предоставить один метод доступа публично, но сделать другой закрытый или защищенный. Дополнительные сведения см. в разделе «Асимметричный доступ аксессора».

После объявления свойств их можно использовать в качестве полей класса. Свойства позволяют использовать естественный синтаксис при получении и установке значения свойства, как в следующих инструкциях:

C#

```
person.Name = "Joe";
person.Age = 99;
```

В методе свойства `set` доступна специальная `value` переменная. Эта переменная содержит значение, указанное пользователем, например:

C#

```
_name = value;
```

Обратите внимание на чистый синтаксис для инкрементации свойства `Age` объекта `Person`.

```
C#

person.Age += 1;
```

Если для моделирования свойств использовались отдельные `set` методы и `get` отдельные методы, эквивалентный код может выглядеть следующим образом:

```
C#

person.SetAge(person.GetAge() + 1);
```

В этом примере метод `ToString` переопределён.

```
C#

public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Обратите внимание, что `ToString` в программе явно не используется. Он вызывается по умолчанию вызовами `WriteLine` .

См. также

- Свойства
- Система типов C#

Автоматически реализованные свойства

Автоматически реализованные свойства делают объявление свойств более кратким, если в методах доступа к свойствам не требуется другая логика. Они также позволяют клиентскому коду создавать объекты. При объявлении свойства, как показано в следующем примере, компилятор создает закрытое анонимное резервное поле, которое может быть доступно только через методы доступа `get` и `set` свойства. `init` Свойства доступа также можно объявить как автоматически реализуемые.

В следующем примере показан простой класс, имеющий некоторые автоматически реализованные свойства:

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

Вы не можете объявлять автоматически реализованные свойства в интерфейсах. Автоматически реализуемые свойства создают закрытое резервное поле экземпляра, а интерфейсы не могут объявлять экземплярные поля. Объявление свойства в интерфейсе без определения тела объявляет свойство с методами доступа. Каждый тип, реализующий этот интерфейс, должен реализовать это свойство.

Вы можете инициализировать автоматически реализованные свойства, аналогичные полям:

C#

```
public string FirstName { get; set; } = "Jane";
```

Класс, который показан в предыдущем примере, является изменяемым. Клиентский код может изменить значения в объектах после создания. В сложных классах, содержащих значительное поведение (методы) и данные, часто необходимо иметь общедоступные свойства. Но для небольших классов или структур, которые просто инкапсулируют набор значений (данных) без какого-либо поведения (или с минимальным поведением), следует использовать один из следующих параметров, чтобы сделать объекты неизменяемыми:

- Объявите только метод доступа `get` (неизменяемое значение везде, за исключением конструктора).
- Объявите акцессор `get` и акцессор `init` (неизменяемые везде, кроме как при создании объекта).
- Объявите метод доступа `set` с атрибутом private (неизменяемое значение для потребителей).

Дополнительные сведения см. в статье "Реализация упрощенного класса с автоматически реализованными свойствами".

Возможно, потребуется добавить проверку в автоматически реализованное свойство. C# 14 добавляет свойства, поддерживаемые полем. Ключевое слово `field` используется для доступа к скрытому полю, синтезированному компилятором, автоматически реализованного свойства. Например, можно убедиться, что свойство `FirstName` в ранее упомянутом примере нельзя установить в `null` или пустую строку.

C#

```
public string FirstName
{
    get;
    set
    {
        field = (string.IsNullOrWhiteSpace(value) is false
```

```
            ? value
            : throw new ArgumentException(nameof(value), "First name can't be
whitespace or null"));
    }
} = "Jane";
```

Эта функция позволяет добавлять логику в методы доступа без необходимости явно объявлять резервное поле. Ключевое `field` слово используется для доступа к резервному полю, созданному компилятором.

См. также

- Использование автоматически реализованных свойств (правило стиля IDE0032)
- Свойства
- Модификаторы

Last updated on 24.02.2026

Реализация упрощенного класса с автоматически реализованными свойствами

Статья • 26.10.2024

В этом примере показано, как создать неизменяемый упрощенный класс, который служит только для инкапсулировать набор автоматически реализованных свойств. Используйте такую конструкцию вместо структуры, когда требуется использовать семантику ссылочного типа.

Неизменяемое свойство можно создать следующими способами:

- Объявить только метод доступа get, который делает свойство неизменяемым везде, кроме конструктора типа.
- Объявить метод доступа init вместо `set`, который делает свойство изменяемым только в конструкторе или с помощью инициализатора объектов.
- Объявить метод доступа set как private. Свойство задано в пределах типа, но оно неизменяемо для потребителей.

Модификатор можно добавить в required объявление свойства, чтобы принудительно вызвать свойство, чтобы задать свойство как часть инициализации нового объекта.

В следующем примере показано, как свойство с методом доступа get, отличается от свойства с методами доступа get и private set.

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;
```

```
    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

Пример

В следующем примере показаны два способа реализации неизменяемого класса, который автоматически реализует свойства. Каждый способ объявляет одно из свойств с закрытым методом доступа `set` и одно из свойств только с методом доступа `get`. Первый класс использует конструктор только для инициализации свойства, а второй класс использует статический фабричный метод, вызывающий конструктор.

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
```

```
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = ["Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"];
        string[] addresses = ["123 Main St.", "345 Cypress Ave.", "678 1st
Ave",
                              "12 108th St.", "89 E. 42nd St."];

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

Компилятор создает резервные поля для каждого автоматически реализованного свойства. Поля недоступны непосредственно из исходного кода.

См. также

- Свойства
- struct
- Инициализаторы объектов и коллекций

Методы (руководство по программированию на C#)

Статья • 04.04.2025

Метод — это блок кода, содержащий ряд инструкций. Программа инициирует выполнение инструкций, вызывая метод и указывая все аргументы, необходимые для этого метода. В C# все инструкции выполняются в контексте метода.

Метод `Main` является точкой входа для каждого приложения C# и вызывается средой CLR при запуске программы. В приложении, использующего инструкции верхнего уровня, `Main` метод создается компилятором и содержит все инструкции верхнего уровня.

> ⓘ **Примечание**
>
> В этой статье рассматриваются именованные методы. Дополнительные сведения об анонимных функциях см. в статье Лямбда-выражения.

Сигнатуры методов

Методы объявляются в классе, структуре или интерфейсе путем указания уровня доступа, например или необязательных модификаторов, таких как `public` `abstract` или `private` `sealed` возвращаемое значение, имя метода и любые параметры метода. Эти части вместе являются сигнатурой метода.

> ⓘ **Важно!**
>
> Тип возврата метода не является частью сигнатуры метода в целях перегрузки метода. Однако он является частью сигнатуры метода при определении совместимости между делегатом и методом, на который он указывает.

Параметры метода заключены в скобки и разделены запятыми. Пустые скобки указывают, что метод не требует параметров. Этот класс содержит четыре метода:

C#

```
abstract class Motorcycle
{
    // Anyone can call this.
```

```
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Доступ к методу

Вызов метода для объекта похож на доступ к полю. После имени объекта добавьте период, имя метода и круглые скобки. Аргументы перечислены в скобках и разделены запятыми. Поэтому методы `Motorcycle` класса можно вызывать следующим образом:

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine($"My top speed is {speed}");
    }
}
```

Параметры метода и аргументы

Определение метода задает имена и типы всех необходимых параметров. Когда вызывающий код вызывает метод, он предоставляет конкретные значения, называемые аргументами, для каждого параметра. Аргументы должны быть совместимы с типом параметра, но имя аргумента (если есть), используемое в

вызывающем коде, не должно совпадать с параметром, указанным в методе. Рассмотрим пример.

C#

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Передача по ссылке и передача по значению

По умолчанию, когда экземпляр типа значения передается методу, его копия передается вместо самого экземпляра. Поэтому изменения аргумента не оказывают никакого влияния на первоначальный экземпляр в методе, в котором он вызывается. Чтобы передать экземпляр типа значения по ссылке, используйте ключевое слово `ref`. Дополнительные сведения см. в разделе "Передача Value-Type параметров".

Когда объект ссылочного типа передается методу, передается ссылка на объект. То есть метод получает не сам объект, а аргумент, указывающий расположение объекта. При изменении элемента объекта с помощью этой ссылки изменение отражается в аргументе в вызывающем методе, даже если объект передается по значению.

Вы создаете ссылочный тип с помощью ключевого `class` слова, как показано в следующем примере:

```
C#

public class SampleRefType
{
    public int value;
}
```

Теперь, если передать объект, основанный на этом типе, методу, передается ссылка на объект. В следующем примере объект типа `SampleRefType` передается методу `ModifyObject`:

```
C#

public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

Пример делает по сути то же самое, что и предыдущий, тем, что передает аргумент по значению методу. Но, поскольку используется ссылочный тип, результат отличается. Изменение, которое выполняется в `ModifyObject`, в поле `value` параметра `obj`, также изменяет поле `value` аргумента `rt` в методе `TestRefType`. Метод `TestRefType` отображает 33 в качестве выходных данных.

Дополнительные сведения о том, как передавать ссылочные типы по ссылке и по значению, см. в разделах Передача Reference-Type параметров и Ссылочные типы.

Возвращаемые значения

Методы могут возвращать значение вызывающему. Если возвращаемый тип (тип, указанный перед именем метода), не равен `void`, метод может вернуть значение, используя оператор return. Оператор с `return` ключевым словом, за которым

следует значение, соответствующее возвращаемому типу, вернет это значение вызывающему методу.

Значение можно вернуть вызывающему объекту по значению или по ссылке. Значения возвращаются вызывающей стороне по ссылке, если ключевое слово `ref` используется в сигнатуре метода и следует за каждым ключевым словом `return`. Например, следующая сигнатура метода и инструкция return указывают, что метод возвращает переменную с именем `estDistance` по ссылке на вызывающий объект.

C#

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

Ключевое слово `return` также останавливает выполнение метода. Если тип возврата — `void`, инструкцию `return` без значения по-прежнему можно использовать для завершения выполнения метода. `return` Без ключевого слова метод перестанет выполняться, когда он достигнет конца блока кода. Методы с типом возвращаемого значения, отличного от void, требуются для использования `return` ключевого слова для возврата значения. Например, в следующих двух методах ключевое слово `return` используется для возврата целочисленных значений.

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Чтобы использовать значение, возвращаемое из метода, вызывающий метод может использовать вызов метода там, где было бы достаточно значения того же типа. Можно также назначить возвращаемое значение переменной. Например, следующие два примера кода выполняют одну и ту же задачу:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Использование локальной переменной в данном случае `result` для хранения значения является необязательным. Это может помочь удобочитаемости кода или может потребоваться, если необходимо сохранить исходное значение аргумента для всей области метода.

Чтобы использовать значение, возвращаемое ссылкой из метода, необходимо объявить локальную переменную ref, если планируется изменить его значение. Например, если метод `Planet.GetEstimatedDistance` возвращает значение по ссылке, его можно определить как локальную переменную ссылочного типа Double с помощью кода, подобного следующему:

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

Возврат многомерного массива из метода, `M` который изменяет содержимое массива, не требуется, если вызывающая функция передает массив в `M`. Можно вернуть полученный массив из `M` для улучшения стиля или функционального потока значений, но это не обязательно, так как C# передает все ссылочные типы по значению, а значение ссылки на массив — это указатель на массив. В методе `M` любые изменения содержимого массива отслеживаются любым кодом, который имеет ссылку на массив, как показано в следующем примере:

C#

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}
```

```
public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Асинхронные методы

С помощью функции async можно вызывать асинхронные методы, не прибегая к использованию явных обратных вызовов или ручному разделению кода между несколькими методами или лямбда-выражениями.

Если пометить метод с помощью модификатора async, можно использовать в этом методе инструкцию await. Когда элемент управления достигает выражения await в асинхронном методе, элемент управления возвращается вызывающему объекту и ход выполнения метода приостанавливается до тех пор, пока ожидающая задача не завершится. После завершения задачи выполнение в методе может быть возобновлено.

> ⓘ **Примечание**
>
> Асинхронный метод возвращает управление вызывающему объекту, когда он встречает первый ожидаемый объект, выполнение которого еще не завершено, или когда выполнение асинхронного метода доходит до конца, в зависимости от того, что произойдет раньше.

Асинхронный метод обычно имеет тип возвращаемого значения Task<TResult>, Task, IAsyncEnumerable<T> или `void`. Тип возвращаемого значения `void` в основном используется для определения обработчиков событий, где требуется возвращать тип `void`. Асинхронный метод, который возвращает тип `void`, не может быть ожидающим. Вызывающий объект метода, возвращающего значение типа void, не может перехватывать исключения, которые выдает этот метод. Асинхронный метод может иметь любой тип возвращаемой задачи.

В следующем примере `DelayAsync` — это асинхронный метод, который имеет тип возвращаемого значения Task<TResult>. У `DelayAsync` есть оператор `return`,

который возвращает целое число. Поэтому объявление `DelayAsync` метода должно иметь возвращаемый тип `Task<int>`. Так как возвращаемый тип - `Task<int>`, вычисление выражения `await` в `DoSomethingAsync` создает целое число, как показано в следующем утверждении: `int result = await delayTask`.

Этот `Main` метод является примером асинхронного метода, имеющего возвращаемый тип Task. Он переходит к методу `DoSomethingAsync`, и поскольку он представлен в одной строке, может опустить ключевые слова `async` и `await`. Поскольку `DoSomethingAsync` является асинхронным методом, выполнение вызова `DoSomethingAsync` должно ожидаться, как показано в следующем выражении: `await DoSomethingAsync();`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Асинхронный метод не может объявлять какие-либо параметры ref или out, но может вызывать методы, имеющие такие параметры.

Дополнительные сведения об асинхронных методах см. в разделах Асинхронное программирование с использованием ключевых слов async и await (C#) и Типы возвращаемых значений асинхронных операций.

Определения текста выражения

Обычно используются определения методов, которые просто сразу возвращают результат выражения или имеют единственное выражение в качестве тела метода. Существует ярлык синтаксиса для определения таких методов с помощью `=>`:

```
C#
```

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Если метод возвращает `void` или является асинхронным методом, текст метода должен быть выражением инструкции (аналогично лямбда-выражениям). Для свойств и индексаторов необходимо, чтобы они были только для чтения, и вы не используете ключевое слово метода доступа `get`.

Итераторы

Итератор выполняет настраиваемую итерацию по коллекции, например по списку или массиву. Итератор использует инструкцию yield return для возврата всех элементов по одному. После достижения инструкции запоминается текущее `yield return` расположение в коде. Выполнение перезапускается с этой позиции, когда итератор вызывается в следующий раз.

Вы вызываете итератор из клиентского кода с помощью инструкции foreach .

Тип возвращаемого итератора может быть IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumeratorили IEnumerator<T>.

Дополнительные сведения см. в разделе Итераторы.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

См. также

- Система типов C#
- модификаторы доступа
- Статические классы и члены статических классов
- Наследование
- абстрактные и неизменяемые классы и их члены
- params
- наружу
- ref;
- Параметры метода

Локальные функции (руководство по программированию на C#)

Статья • 22.11.2024

Локальные функции — это методы типа, вложенного в другой член. Они могут вызываться только из того элемента, в который вложены. Ниже перечислены элементы, в которых можно объявлять и из которых можно вызывать локальные функции:

- Методы, в частности методы итератора и асинхронные методы
- Конструкторы
- Методы доступа к свойствам
- Методы доступа событий
- Анонимные методы
- Лямбда-выражения
- Методы завершения
- Другие локальные функции

Тем не менее локальные функции нельзя объявлять внутри элемента, воплощающего выражение.

> ⓘ **Примечание**
>
> В некоторых случаях для реализации возможностей, поддерживаемых локальными функциями, также можно использовать лямбда-выражения. Дополнительные сведения см. в разделе **Локальные функции или лямбда-выражения**.

Применение локальных функций позволяет сделать предназначение кода более понятным. Любой пользователь, прочитав код, может увидеть, что метод не вызывается, за исключением содержащего метода. В случае с командными проектами это также гарантирует, что другой разработчик не сможет ошибочно вызвать метод напрямую из любого другого места в классе или структуре.

Синтаксис локальной функции

Локальная функция определяется как вложенный метод внутри содержащего ее элемента. Ниже приведен синтаксис определения локальной функции:

```
```

C#

```
<modifiers> <return-type> <method-name> <parameter-list>
```

> ⓘ **Примечание**
>
> Не `<parameter-list>` должен содержать параметры с именем контекстного ключевого слова `value`. Компилятор создает временную переменную "value", которая содержит ссылки на внешние переменные, которые позже вызывают неоднозначность, а также может привести к неожиданному поведению.

С локальной функцией можно использовать следующие модификаторы:

- async
- unsafe
- static Статическую локальную функцию нельзя записывать локальные переменные или состояние экземпляра.
- extern Должна быть `static` внешняя локальная функция.

Все локальные переменные, определенные в содержащем функцию элементе (включая параметры метода), доступны в нестатической локальной функции.

В отличие от определения метода, определение локальной функции не может включать модификатор доступа к члену. Так как все локальные функции являются закрытыми, включая модификатор доступа, например ключевое слово `private`, создает ошибку компилятора CS0106, "Модификатор "private" не является допустимым для этого элемента".

В следующем примере определяется локальная функция `AppendPathSeparator`, которая является частной для метода `GetText`:

C#

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

Атрибуты можно применить к локальной функции, его параметрам и параметрам типа, как показано в следующем примере:

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

В предыдущем примере используется специальный атрибут для помощи компилятору в статическом анализе в контексте, допускающем значение NULL.

Локальные функции и исключения

Полезной особенностью локальных функций является то, что они допускают немедленную обработку исключений. Для методов итератора исключения отображаются только при перечислении возвращаемой последовательности, а не при извлечении итератора. В случае с асинхронными методами любые исключения, возникшие в таком методе, наблюдаются в тот момент, когда возвращаемая задача находится в состоянии ожидания.

В следующем примере определяется метод `OddSequence`, который перечисляет нечетные числа в заданном диапазоне. Поскольку он передает в метод перечислителя `OddSequence` число больше 100, этот метод вызывает исключение ArgumentOutOfRangeException. Как видно из выходных данных этого примера, исключение обрабатывается только в момент перебора чисел, а не при извлечении перечислителя.

C#

```
public class IteratorWithoutLocalExample
{
   public static void Main()
```

```
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11
```

Если поместить логику итератора в локальную функцию, при получении перечислителя вызываются исключения проверки аргументов, как показано в следующем примере:

C#

```
public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
```

```
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        return GetOddSequenceEnumerator();

        IEnumerable<int> GetOddSequenceEnumerator()
        {
            for (int i = start; i <= end; i++)
            {
                if (i % 2 == 1)
                    yield return i;
            }
        }
    }
}
// The example displays the output like this:
//
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

Локальные функции или лямбда-выражения

На первый взгляд локальные функции и лямбда-выражения похожи. Во многих случаях выбор между лямбда-выражениями и локальными функциями является вопросом стиля и личного предпочтения. Однако существуют реальные различия в использовании этих сущностей, о которых нужно знать.

Рассмотрим различия в реализации алгоритма вычисления факториала с использованием локальной функции и лямбда-выражения. В этой версии используется локальная функция:

C#

```
public static int LocalFunctionFactorial(int n)
{
```

```
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

В этой версии используются лямбда-выражения:

C#

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Именование

Локальные функции явно именуются как методы. Лямбда-выражения представляют собой анонимные методы и должны назначаться переменным типа `delegate`, как правило, типа `Action` или `Func`. Процесс объявления локальной функции аналогичен написанию обычного метода: вы объявляете тип возвращаемого значения и сигнатуру функции.

Сигнатуры функций и типы лямбда-выражений

Лямбда-выражения используют тип переменной `Action`/`Func`, которой они назначаются, для определения типов аргументов и возвращаемых значений. Поскольку синтаксис локальных функций во многом аналогичен обычному методу, типы аргументов и возвращаемых значений уже входят в объявление функции.

Некоторые лямбда-выражения имеют *естественного типа*, что позволяет компилятору выводить возвращаемый тип и типы параметров лямбда-выражения.

Определенное назначение

Лямбда-выражения — это объекты, которые объявляются и назначаются во время выполнения. Для того чтобы лямбда-выражение могло быть использовано, оно

должно быть определенно назначено: переменная `Action`/`Func`, которой оно назначается, должна быть объявлена, и лямбда-выражение должно быть назначено этой переменной. Обратите внимание на то, что `LambdaFactorial` должно объявить и инициализировать лямбда-выражение `nthFactorial`, прежде чем его определить. В противном случае возникает ошибка компилятора, связанная со ссылкой на объект `nthFactorial`, который еще не был назначен.

Локальные функции определяются во время компиляции. Так как они не присваиваются переменным, на них можно ссылаться из любой части кода **, где она находится в области видимости**. В первом примере `LocalFunctionFactorial`вы можете объявить локальную функцию до или после инструкции `return`, и это не вызовет ошибок компиляции.

Эти различия означают, что рекурсивные алгоритмы легче создавать, используя локальные функции. Можно объявить и определить локальную функцию, которая вызывает саму себя. Лямбда-выражения должны быть объявлены и им должно быть присвоено значение по умолчанию, прежде чем они могут быть переназначены телу, ссылающемуся на то же лямбда-выражение.

Реализация в виде делегата

Лямбда-выражения преобразуются в делегаты при объявлении. Локальные функции являются более гибкими и могут определяться в виде традиционного метода *или* делегата. Локальные функции преобразуются в делегаты только при ***использовании*** в качестве делегата.

Если вы объявляете локальную функцию и ссылаетесь на нее только путем вызова ее как метода, она не будет преобразована в делегат.

Захват переменной

Правила определенного присваивания также влияют на любые переменные, захваченные локальной функцией или лямбда-выражением. Компилятор может выполнять статический анализ, что позволяет локальным функциям определенно назначать захватываемые переменные во включающей области. Рассмотрим следующий пример:

```
C#

int M()
{
    int y;
    LocalFunction();
```

```
    return y;

    void LocalFunction() => y = 0;
}
```

Компилятор может определить, что `LocalFunction` определенно назначает `y` при вызове. Поскольку `LocalFunction` вызывается перед оператором `return`, `y` определенно назначается в операторе `return`.

Если локальная функция захватывает переменные в окружающей области, она реализуется с помощью замыкания, как и типы делегатов.

Распределение куч

В зависимости от использования при работе с локальными функциями можно избежать распределения куч, которое всегда необходимо выполнять при работе с лямбда-выражениями. Если локальная функция никогда не преобразуется в делегат и ни одна из переменных, захваченных локальной функцией, не захвачена другими лямбда-выражениями или локальными функциями, которые преобразуются в делегаты, компилятор может избежать распределения куч.

Рассмотрим следующий асинхронный пример:

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int
index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    };
```

```
    return await longRunningWorkImplementation();
}
```

Закрытие для этого лямбда-выражения содержит переменные `address`, `index` и `name`. Для локальных функций объект, реализующий закрытие, может быть типом `struct`. Этот тип структуры будет передан в локальную функцию посредством ссылки. Эта разница в реализации позволяет избежать распределения.

Выделение экземпляров для лямбда-выражений ведет к дополнительному расходу памяти, что может влиять на производительность в критически важных участках кода. Локальные функции не влекут за собой эти издержки.

Если известно, что локальная функция не будет преобразована в делегат и ни одна из захватываемых ею переменных не захватывается другими лямбда-выражениями или локальными функциями, которые преобразуются в делегаты, то можно гарантировать, что локальная функция не будет распределяться в куче за счет объявления в качестве локальной функции `static`.

> **Совет**
>
> Включите правило **стиля кода .NET IDE0062** , чтобы гарантировать, что локальные функции всегда помечены `static`.

> **Примечание**
>
> В эквивалентном этому методе на основе локальной функции также используется класс для замыкания. Реализация замыкания для локальной функции в формате `class` или `struct` зависит от особенностей реализации. Локальная функция может использовать `struct`, тогда как в лямбда-выражениях всегда используется `class`.

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
```

```
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    }
}
```

Использование ключевого слова `yield`

Еще одно преимущество локальных функций, которое не показано в этом примере, заключается в том, что они могут быть реализованы в качестве итераторов с использованием синтаксиса `yield return` для создания последовательности значений.

C#

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to
lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

Оператор `yield return` не допускается в лямбда-выражениях. Дополнительные сведения см. в разделе об ошибке компилятора CS1621.

Хотя локальные функции могут показаться избыточными для лямбда-выражений, они на самом деле служат разным целям и имеют разные виды использования.

Локальные функции более эффективны в случаях, когда вам нужно написать функцию, которая будет вызываться только из контекста другого метода.

Спецификация языка C#

Дополнительные сведения см. в разделе "Объявления локальных функций" спецификации языка C#.

См. также

- Используйте локальную функцию вместо лямбда-правила (правило стиля IDE0039)
- Методы

Неявно типизированные локальные переменные (руководство по программированию на C#)

Статья • 14.03.2023

Локальные переменные можно объявлять без указания конкретного типа. Ключевое слово `var` указывает, что компилятор должен вывести тип переменной из выражения справа от оператора инициализации. Выведенный тип может быть встроенным, анонимным, определяемым пользователем либо типом, определяемым в библиотеке классов .NET. Дополнительные сведения об инициализации массивов с `var` см. в разделе Неявно типизированные массивы.

В приведенных ниже примерах показаны различные способы объявления локальных переменных с помощью `var`:

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

Важно понимать, что ключевое слово `var` не означает "variant" и не означает, что переменная является слабо типизированной или имеет позднее связывание. Он указывает только на то, что компилятор определяет и назначает наиболее подходящий тип.

Ключевое слово `var` можно использовать в следующих контекстах:

- С локальными переменными (переменными, объявленными в области метода), как показано в предыдущем примере.
- В операторе инициализации for.

  C#

  ```
  for (var x = 1; x < 10; x++)
  ```

- В операторе инициализации foreach.

  C#

  ```
  foreach (var item in list) {...}
  ```

- В операторе using.

  C#

  ```
  using (var file = new StreamReader("C:\\myfile.txt")) {...}
  ```

Дополнительные сведения см. в разделе Практическое руководство. Использование неявно типизированных локальных переменных и массивов в выражении запроса.

Переменная var и анонимные типы

Во многих случаях переменная `var` не является обязательной и предназначена только для синтаксического удобства. Тем не менее если переменная инициализируется с помощью анонимного типа и вам потребуется доступ к свойствам объекта на более позднем этапе, ее необходимо объявить как `var`. Это распространенный сценарий в выражениях запросов LINQ. Дополнительные сведения см. в статье Анонимные типы.

С точки зрения исходного кода анонимный тип безымянен. Таким образом, если переменная запроса инициализирована с помощью `var`, то единственный способ получить доступ к свойствам в возвращаемой последовательности объектов — это использовать `var` как тип переменной итерации в операторе `foreach`.

C#

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Примечания

В объявлениям неявно типизированных переменных применяются следующие ограничения:

- `var` можно использовать только в том случае, если локальная переменная объявляется и инициализируется в одном и том же операторе; переменная не может инициализироваться в нулевое значение, в группу методов или в анонимную функцию.
- `var` нельзя применять к полям в области видимости класса.
- Переменные, объявленные с помощью `var`, нельзя использовать в выражении инициализации. Другими словами, это выражение является допустимым выражением: `int i = (i = 20);`, но вызывает ошибку времени компиляции: `var i = (i = 20);`
- Инициализировать сразу несколько неявно типизированных переменных в одном и том же операторе нельзя.

- Если тип с именем `var` входит в область видимости, то ключевое слово `var` разрешится в это имя типа и не будет обрабатываться как часть объявления неявно типизированной локальной переменной.

Неявное типизирование с ключевым словом `var` может применяться только к переменным в области локального метода. Неявное типизирование недоступно для полей класса C#, так как при обработке кода компилятор столкнется с логическим парадоксом: компилятор должен знать тип поля, но он не может определить тип, пока не проанализирует выражение присваивания, и не может вычислить выражение, не зная тип. Рассмотрим следующий код.

```
C#

private var bookTitles;
```

`bookTitles` — это поле класса, которому присваивается тип `var`. Так как поле не имеет выражения для оценки, то компилятор не сможет вывести тип `bookTitles`. Кроме того, добавления выражения в поле (так же, как для локальной переменной) тоже недостаточно:

```
C#

private var bookTitles = new List<string>();
```

Когда компилятор обнаруживает поля во время компиляции кода, он записывает тип каждого поля перед обработкой любого выражения, связанного с ним. Компилятор обнаруживает тот же парадокс при попытке анализа `bookTitles`: он должен знать тип поля, но обычно он определяет тип `var` путем анализа выражения, который невозможно определить, если заранее не знать тип.

Переменную `var` можно также использовать в выражениях запросов, где точный сконструированный тип переменной запроса определить непросто. Подобная ситуация может возникнуть в операциях группировки и сортировки.

Кроме того, ключевое слово `var` может пригодиться, если конкретный тип переменной сложно набрать с клавиатуры, а также если он очевиден либо затрудняет чтение кода. Использовать `var` таким образом можно, например, с вложенными универсальными типами — подобные типы применяются в групповых операциях. В следующем запросе переменная запроса имеет тип `IEnumerable<IGrouping<string, Student>>`. Если вы и другие пользователи, которые работают с кодом, это понимаете, использовать неявный ввод для удобства и краткости кода можно без проблем.

C#

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

Ключевое слово `var` позволяет упростить код, однако его следует использовать только там, где оно действительно необходимо или где оно облегчает понимание кода. Дополнительные сведения о том, когда следует использовать ключевое слово `var`, см. в разделе Неявно типизированные локальные переменные в статье с рекомендациями по написанию кода C#.

См. также

- Справочник по C#
- Неявно типизированные массивы
- Практическое руководство. Использование неявно типизированных локальных переменных и массивов в выражении запроса
- Анонимные типы
- Инициализаторы объектов и коллекций
- var
- LINQ в C#
- Встроенный язык запросов LINQ
- Инструкции итерации
- Оператор using

Использование неявно типизированных локальных переменных и массивов в выражении запроса (руководство по программированию на C#)

Неявно типизированные локальные переменные можно использовать всякий раз, когда компилятору требуется определить тип локальной переменной. Для хранения анонимных типов, которые часто используются в выражениях запросов, необходимо использовать неявно типизированные локальные переменные. В следующих примерах показаны как необязательные, так и обязательные использование неявно типизированных локальных переменных в запросах.

Неявно типизированные локальные переменные объявляются с помощью контекстного ключевого слова var . Дополнительные сведения см. в разделе "Неявно типизированные локальные переменные " и неявно типизированные массивы.

Примеры

В следующем примере показан распространенный сценарий, в котором `var` требуется ключевое слово: выражение запроса, которое создает последовательность анонимных типов. В этом сценарии переменная запроса и переменная итерации в `foreach` инструкции должны быть неявно типизированными, `var` так как у вас нет доступа к имени типа для анонимного типа. Дополнительные сведения об анонимных типах см. в разделе "Анонимные типы".

C#

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,
anonType.LastName);
```

```
    }
}
```

В следующем примере используется `var` ключевое слово в аналогичной ситуации, но в которой использование `var` является необязательным. Так как `student.LastName` это строка, выполнение запроса возвращает последовательность строк. Таким образом, тип `queryId` можно объявить как `System.Collections.Generic.IEnumerable<string>` вместо `var`. Ключевое слово `var` используется для удобства. В примере переменная итерации в `foreach` инструкции явно вводится в виде строки, но вместо этого ее можно объявить с помощью `var`. Поскольку тип переменной итерации не является анонимным типом, использование `var` является параметром, а не требованием. Помните, что `var` сам по себе не является типом, а является инструкцией для компилятора вывести и назначить тип.

C#

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine($"Last name: {str}");
}
```

См. также

- Члены расширения
- LINQ (Language-Integrated запрос)
- LINQ в C#

Last updated on 02.12.2025

Практическое руководство. Реализация и вызов пользовательского метода расширения (руководство по программированию на C#)

В этой статье показано, как реализовать собственные методы расширения для любого типа .NET. Клиентский код может использовать ваши методы расширения. Клиентские проекты должны ссылаться на сборку, содержащую их. Клиентские проекты должны добавить директиву using , указывающую пространство имен, в котором определены методы расширения.

Начиная с C# 14, существует два синтаксиса, которые можно использовать для определения методов расширения. C# 14 добавляет блоки расширения, в которых определяются члены расширения для типа. Перед C# 14 добавьте этот модификатор в первый параметр статического метода, чтобы указать, что метод отображается как член экземпляра типа параметра.

Блоки расширения (C# 14 и более поздних версий)

Начиная с C# 14, можно объявить методы расширения с помощью *блоков расширения*. Блок расширения — это блок в невложенном, не универсальном, статичном классе, который содержит члены расширения для типа или экземпляра этого типа.

Определение и вызов метода расширения с помощью блоков расширений:

1. Определите статический класс, который будет содержать метод расширения. Класс не может быть вложен в другой тип и должен быть видимым для клиентского кода. Дополнительные сведения о правилах доступа см. в разделах Модификаторы доступа.
2. Используйте блок расширения для объявления членов расширения для типа.
3. В вызывающем коде добавьте директиву `using`, чтобы задать пространство имен, содержащее класс метода расширения.
4. Вызовите методы в качестве методов экземпляра в типе.

В следующем примере реализуется метод расширения с именем `WordCount` с использованием синтаксиса блока расширения C# 14. Метод работает с классом String . Пространство `CustomExtensions` имен импортируется в пространство имен приложения, и метод вызывается так, как если бы он был методом экземпляра в строке.

C#

```
using CustomExtensions;

string s = "The quick brown fox jumped over the lazy dog.";
// Call the method as if it were an
// instance method on the type. Note that the first
// parameter is not specified by the calling code.
int i = s.WordCount();
System.Console.WriteLine($"Word count of s is {i}");


namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        extension(string str)
        {
            // This is the extension member.
            // The `str` parameter is declared on the extension declaration.
            public int WordCount()
            {
                return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
            }
        }
    }
}
```

Методы расширения (более ранние версии)

Перед C# 14 вы объявляете метод расширения, добавив этот модификатор в первый параметр статического метода.

Чтобы определить и вызвать метод расширения с помощью классического синтаксиса:

1. Определите статический класс, который будет содержать метод расширения. Класс не может быть вложен в другой тип и должен быть видимым для клиентского кода. Дополнительные сведения о правилах доступа см. в разделах Модификаторы доступа.
2. Реализуйте метод расширения как статический метод как минимум с тем же уровнем видимости, что и содержащий класс.
3. Первый параметр метода указывает тип, с которым работает метод. Ему должен предшествовать модификатор this.
4. В вызывающем коде добавьте директиву `using`, чтобы задать пространство имен, содержащее класс метода расширения.
5. Вызовите методы в качестве методов экземпляра в типе.

В следующем примере реализуется метод расширения с именем `WordCount` с помощью классического синтаксиса. Метод работает с классом String , который указывается в качестве первого параметра метода с модификатором `this` . Пространство `CustomExtensions` имен импортируется в пространство имен приложения, и метод вызывается так, как если бы он был методом экземпляра в строке.

C#

```
using CustomExtensions;

string s = "The quick brown fox jumped over the lazy dog.";
// Call the method as if it were an
// instance method on the type. Note that the first
// parameter is not specified by the calling code.
int i = s.WordCount();
System.Console.WriteLine($"Word count of s is {i}");


namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type to be extended.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

Примечание.

Первый параметр не указан путем вызова кода, так как представляет тип, к которому применяется метод, и компилятор уже знает тип объекта. Вам необходимо предоставить аргументы только для параметров со 2 по `n`.

Общая информация

Разрешение перегрузки предпочитает экземплярные или статические методы, определенные самим типом, по сравнению с методами расширения. Методы расширения не могут получить доступ к частным данным в расширенном классе.

См. также

- Методы расширения
- LINQ (Language-Integrated запрос)
- Статические классы и члены статических классов
- защищено
- внутренний
- public
- это
- namespace

Last updated on 28.01.2026

Создание нового метода для перечисления (руководство по программированию на C#)

Методы расширения можно использовать для добавления функциональности, относящейся к конкретному типу перечисления. В следующем примере перечисление `Grades` представляет возможные буквенные оценки, которые студент может получить в классе. Метод расширения `Passing` добавляется к типу `Grades`, чтобы каждый экземпляр этого типа теперь "содержал информацию", представляет ли он проходной балл или нет.

C#

```
public enum Grades
{
    F = 0,
    D = 1,
    C = 2,
    B = 3,
    A = 4
};

// Define an extension method in a non-nested static class.
public static class Extensions
{
    public static bool Passing(this Grades grade, Grades minPassing = Grades.D) =>
        grade >= minPassing;
}
```

Метод расширения можно вызвать так, как будто он был объявлен в типе `enum` :

C#

```
Grades g1 = Grades.D;
Grades g2 = Grades.F;
Console.WriteLine($"First {(g1.Passing() ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing() ? "is" : "is not")} a passing grade.");

Console.WriteLine("\r\nRaising the bar!\r\n");
Console.WriteLine($"First {(g1.Passing(Grades.C) ? "is" : "is not")} a passing
grade.");
Console.WriteLine($"Second {(g2.Passing(Grades.C) ? "is" : "is not")} a passing
grade.");
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!
```

```
    First is not a passing grade.
    Second is not a passing grade.
*/
```

Начиная с C# 14, вы можете объявлять *члены расширения* в блоке расширений. Новый синтаксис позволяет добавлять *свойства расширения*. Вы также можете добавить элементы расширения, которые, как представляется, являются новыми статическими методами или свойствами. Вы больше не ограничены расширениями, которые выглядят как методы экземпляра. В следующем примере показан блок расширения, добавляющий свойство расширения экземпляра для `Passing`, и статическое свойство расширения для `MinimumPassingGrade`.

C#

```
public static class EnumExtensions
{
    private static Grades minimumPassingGrade = Grades.D;

    extension(Grades grade)
    {
        public static Grades MinimumPassingGrade
        {
            get => minimumPassingGrade;
            set => minimumPassingGrade = value;
        }

        public bool Passing => grade >= minimumPassingGrade;
    }
}
```

Вы вызываете эти новые свойства расширения, как если бы они объявлялись в расширенном типе:

C#

```
Grades g1 = Grades.D;
Grades g2 = Grades.F;
Console.WriteLine($"First {(g1.Passing ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing ? "is" : "is not")} a passing grade.");

Grades.MinimumPassingGrade = Grades.C;
Console.WriteLine($"\r\nRaising the bar. Passing grade is now
{Grades.MinimumPassingGrade}!\r\n");
Console.WriteLine($"First {(g1.Passing ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing ? "is" : "is not")} a passing grade.");
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!
```

```
    First is not a passing grade.
    Second is not a passing grade.
*/
```

Дополнительные сведения о новых членах расширения см. в статье о членах расширения и в справочной статье по языку о extension ключевом слове.

См. также

- Методы расширения

Last updated on 25.11.2025

Именованные и необязательные аргументы (Руководство по программированию на C#)

10.10.2025

Именованные аргументы позволяют указать аргумент для параметра, связав этот аргумент с именем параметра, а не с его позицией в списке параметров. *Необязательные аргументы* позволяют опускать аргументы для некоторых параметров. Оба варианта можно использовать с методами, индексаторами, конструкторами и делегатами.

При использовании именованных и необязательных аргументов аргументы оцениваются в том порядке, в котором они отображаются в списке аргументов, а не в списке параметров.

Именованные и дополнительные параметры позволяют указать аргументы для выбранных параметров. Эта возможность значительно упрощает вызов интерфейсов COM, таких как API автоматизации Microsoft Office.

Именованные аргументы

Именованные аргументы освобождают от сопоставления порядка аргументов с порядком параметров в списках параметров вызываемых методов. Аргумент для каждого параметра указывается по имени параметра. Например, функция, которая печатает сведения о заказе (например, имя продавца, номер заказа и имя продукта) вызывается путем отправки аргументов по позиции в порядке, определенном функцией.

C#

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Если вы не помните порядок параметров, но знаете их имена, можете передать аргументы в любом порядке.

C#

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Именованные аргументы также делают код более удобным для чтения, поскольку указывают, чему соответствует каждый аргумент. В следующем примере метода

`sellerName` не может быть пустым или пробелом. Так как и `sellerName`, и `productName` являются строковыми типами, вместо передачи аргументов по позиции имеет смысл использовать именованные аргументы, чтобы устранить данную неоднозначность и сделать код более удобочитаемым.

Именованные аргументы при использовании с позиционными аргументами допустимы при условии, что

- за ними не следуют позиционные аргументы, либо,

  C#

  ```
  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
  ```

- они используются в правильной позиции. В следующем примере параметр `orderNum` находится в правильной позиции, но не имеет явного имени.

  C#

  ```
  PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
  ```

Позиционные аргументы после внеочередных именованных аргументов недопустимы.

C#

```
// This generates CS1738: Named argument specifications must appear after all
fixed arguments have been specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Пример

Следующий код реализует примеры из этого раздела вместе с некоторыми дополнительными.

C#

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional
arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
```

```
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift
Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",
orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or
empty.", paramName: nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product:
{productName}");
    }
}
```

Необязательные аргументы

Определение метода, конструктора, индексатора или делегата может указывать, являются его параметры обязательными или нет. Любой вызов должен содержать аргументы для всех обязательных параметров; аргументы для необязательных параметров можно опустить. Тип ссылок с значением NULL (`T?`) позволяет явным `null` образом использовать аргументы, но не делает параметр необязательным.

Определение каждого необязательного параметра содержит его значение по умолчанию. Если аргумент для параметра не передается, используется значение по умолчанию. Значением по умолчанию должен быть один из следующих типов выражений:

- константное выражение;
- выражение в форме `new ValType()`, где `ValType` — это тип значения, например, enum или struct;
- выражение в форме default(ValType), где `ValType` — это тип значения.

Необязательные параметры определяются в конце списка параметров после всех обязательных параметров. Вызывающий объект должен предоставить аргументы для всех обязательных параметров и любых необязательных параметров, предшествующих указанным. Пробелы, разделенные запятыми в списке аргументов, не поддерживаются. Например, в следующем коде метод `ExampleMethod` экземпляра определяется одним обязательным и двумя необязательными параметрами.

```
C#

public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

Следующий вызов `ExampleMethod` вызывает ошибку компилятора, поскольку аргумент предоставляется для третьего параметра, но не для второго.

```
C#

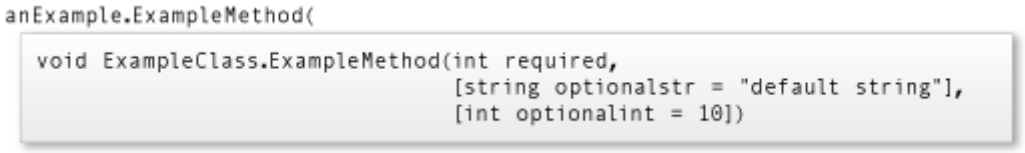
// anExample.ExampleMethod(3, ,4);
```

Если вы знаете имя третьего параметра, можете использовать для выполнения задачи именованный аргумент.

```
C#

anExample.ExampleMethod(3, optionalint: 4);
```

В IntelliSense необязательные параметры заключаются в квадратные скобки, как показано на следующем рисунке:



> ⓘ **Примечание**
>
> Необязательные параметры также можно объявлять с помощью класса .NET OptionalAttribute. `OptionalAttribute` Параметры не требуют значения по умолчанию. Однако если нужное значение по умолчанию, ознакомьтесь с DefaultParameterValueAttribute классом.

Пример

В следующем примере конструктор `ExampleClass` имеет один параметр, который является необязательным. У метода экземпляра `ExampleMethod` есть один обязательный параметр, `required`, и два необязательных параметра, `optionalstr` и `optionalint`. Код в `Main` демонстрирует различные способы, которые можно использовать для вызова конструктора и метода.

C#

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
```

```
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default
string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

Приведенный выше код иллюстрирует несколько случаев, когда необязательные параметры используются правильно и неправильно. Аргументы должны быть предоставлены для всех обязательных параметров. Пробелы в необязательных аргументах должны быть заполнены именованными параметрами.

Атрибуты сведений о вызывающем объекте

Вызывающие информационные атрибуты, такие как CallerFilePathAttribute, CallerLineNumberAttributeCallerMemberNameAttributeи CallerArgumentExpressionAttribute, используются для получения сведений о вызывающем методе. Эти атрибуты особенно полезны при отладке или при необходимости регистрировать сведения о вызовах методов.

Эти атрибуты являются необязательными параметрами со значениями по умолчанию, предоставляемыми компилятором. Вызывающий не должен явно предоставлять значение для этих параметров.

COM-интерфейсы

Именованные и необязательные аргументы, а также поддержка динамических объектов значительно улучшают взаимодействие с API COM, такими как API автоматизации Office.

Например, метод AutoFormatв интерфейсе Microsoft Office Excel Range имеет семь параметров и все они необязательные. Эти параметры показаны на следующем рисунке:

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

Однако можно значительно упростить вызов `AutoFormat` с помощью именованных и необязательных аргументов. Именованные и необязательные аргументы позволяют опускать аргументы для необязательных параметров, если значение параметра по умолчанию менять не нужно. В следующем вызове значение задается только для одного из семи параметров.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

См. сведения и примеры в руководствах по использованию именованных и необязательных аргументов в программировании Office и получению доступа к объектам взаимодействия Office с помощью функций языка C#.

Разрешение перегрузки

Использование именованных и необязательных аргументов влияет на разрешение перегрузки описанным ниже образом.

- Метод, индексатор или конструктор является кандидатом на выполнение, если каждый из его параметров необязателен либо по имени или позиции соответствует одному и тому же аргументу в операторе вызова, и этот аргумент можно преобразовать в тип параметра.
- Если найдено более одного кандидата, правила разрешения перегрузки для предпочтительных преобразований применяются к аргументам, указанным явно. Опущенные аргументы для необязательных параметров игнорируются.

- Если два кандидата определяются как равно подходящие, предпочтение отдается кандидату без необязательных параметров, аргументы которых в вызове были опущены. При разрешении перегрузки обычно используются кандидаты с меньшим числом параметров.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Конструкторы (руководство по программированию на C#)

17.06.2025

Конструктор — это метод, вызываемый средой выполнения при создании экземпляра класса или структуры . Класс или структуру может иметь несколько конструкторов, которые принимают разные аргументы. Конструкторы позволяют обеспечить корректность экземпляров типа, когда они создаются. Дополнительные сведения и примеры см. в разделах Конструкторы экземпляров и Использование конструкторов.

Существует несколько действий, которые являются частью инициализации нового экземпляра. Следующие действия выполняются в следующем порядке:

1. для полей экземпляра *задано значение 0*. Эта инициализация обычно выполняется средой выполнения.
2. *инициализаторы полей запускаются*. Инициализаторы полей в самом производном типе выполняются.
3. *Инициализаторы полей базового типа выполняются*. Инициализаторы полей начинаются с непосредственной базой, проходя через каждый базовый тип к System.Object.
4. *конструкторы базовых экземпляров выполняют*. Все конструкторы экземпляров, начиная с Object.Object и проходя через каждый базовый класс до прямого базового класса.
5. *выполняется конструктор экземпляра*. Конструктор экземпляра типа запускается.
6. *инициализаторы объектов выполняются*. Если выражение включает в себя инициализаторы объектов, они выполняются после запуска конструктора экземпляра. Инициализаторы объектов выполняются в текстовом порядке.

Предыдущие действия выполняются при создании экземпляра с помощью оператора new. Если для нового экземпляра `struct` задано значение `default`, для всех полей экземпляра задано значение 0. Элементы массива имеют значение по умолчанию 0 или `null` при создании массива.

Статический конструктор , если таковой имеется, выполняется перед запуском любого конструктора экземпляра для любого экземпляра этого типа. Статический конструктор выполняется не более одного раза.

Начиная с C# 14, конструкторы экземпляров можно объявлять как частичные члены. Частичные конструкторы должны включать обе декларации: декларативную и реализующую.

Синтаксис конструктора

Конструктор — это метод с тем же именем, что и его тип. Его сигнатура метода может включать необязательный модификатор доступа, имя метода и список параметров; Он не включает тип возвращаемого значения. В следующем примере показан конструктор для класса с именем `Person`.

C#

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

Если конструктор можно реализовать в виде одной инструкции, можно использовать элемент тела выражения. В следующем примере определяется `Location` класс, конструктор которого имеет один строковый параметр. `name` Определение тела выражения присваивает аргумент полю `locationName`.

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Если типу требуется параметр для создания экземпляра, можно использовать *первичный конструктор*, чтобы указать, что для создания экземпляра типа требуется один или несколько параметров, как показано в следующем примере:

C#

```
public class LabelledContainer<T>(string label)
{
   public string Label { get; } = label;
   public required T Contents
   {
      get;
      init;
   }
}
```

Вы можете объявить первичный конструктор для типа `partial`. Допускается только одно объявление первичного конструктора. Другими словами, только одно объявление `partial` типа может включать параметры для основного конструктора.

Статические конструкторы

В предыдущих примерах показаны конструкторы экземпляров, которые инициализируют новый объект. Класс или структуру также может объявлять статический конструктор, который инициализирует статические члены типа. Статические конструкторы не имеют параметров. Если вы не предоставляете статический конструктор для инициализации статических полей, компилятор C# инициализирует статические поля по умолчанию, как указано в значения типов C#, статье.

В следующем примере для инициализации статического поля используется статический конструктор.

C#

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult() => minimumAge = 18;

   // Remaining implementation of Adult class.
}
```

Вы также можете определить статический конструктор с определением текста выражения, как показано в следующем примере.

C#

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

Для получения дополнительной информации и примеров см. Статические конструкторы.

Частичные конструкторы

Начиная с C# 14, можно объявить *частичные конструкторы* в частичном классе или структуре. Любой частичный конструктор должен иметь *определяющее объявление* и *реализующее объявление*. Подписи объявляющих и реализующих частичные конструкторы должны соответствовать правилам частичных членов. Определение объявления частичного конструктора не может использовать инициализаторы конструктора `: base()` или `: this()`. При добавлении любого инициализатора конструктора он должен быть добавлен к объявлению реализации.

См. также

- Система типов C#
- static
- Почему инициализаторы выполняются в порядке, обратном конструкторам? Часть 1

Использование конструкторов (руководство по программированию на C#)

Статья • 18.01.2025

Когда создается экземпляр структуры класса или структуры, среда выполнения вызывает его конструктор. Конструкторы имеют то же имя, что и класс или структура, и обычно инициализируют элементы данных нового объекта.

В следующем примере класс с именем `Taxi` определяется с помощью простого конструктора. Затем этот класс создается с помощью оператора new. Среда выполнения вызывает конструктор `Taxi` сразу после выделения памяти для нового объекта.

C#

```
public class Taxi
{
    private string taxiTag;

    public Taxi(string tag) => taxiTag = tag;

    public override string ToString() => $"Taxi: {taxiTag}";
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi("Tag1345");
        Console.WriteLine(t);
    }
}
```

Конструктор, который не принимает параметров, вызывается *конструктор без параметров*. Среда выполнения вызывает конструктор без параметров, если объект создается с помощью оператора `new`, а аргументы не предоставляются для `new`. В C# 12 появились *первичные конструкторы*. Основной конструктор задает параметры, которые необходимо предоставить для инициализации нового объекта. Дополнительные сведения см. в конструкторах экземпляров.

Если класс не статический, классы без конструкторов получают открытый конструктор без параметров от компилятора C#, для создания экземпляра класса.

Дополнительные сведения см. в Статические классы и статические члены классов.

Вы можете предотвратить создание экземпляра класса, сделав конструктор закрытым, как показано ниже.

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Дополнительные сведения см. в разделе «Частные конструкторы».

Конструкторы для структур типа напоминают конструкторы классов. При создании экземпляра типа структуры с `new` среда выполнения вызывает конструктор. Если для `struct` задано значение `default`, среда выполнения инициализирует всю память в структуре до 0. Если вы объявляете любые инициализаторы полей в типе `struct`, необходимо указать конструктор без параметров. Дополнительные сведения см. в разделе Инициализация структуры и значения по умолчанию статьи типов структуры .

Следующий код использует конструктор без параметров для Int32, чтобы убедиться, что целое число инициализировано:

C#

```
int i = new int();
Console.WriteLine(i);
```

Однако следующий код вызывает ошибку компилятора, так как он не использует `new`, и потому что он пытается использовать объект, который не инициализирован:

C#

```
int i;
Console.WriteLine(i);
```

Кроме того, некоторые `struct` типы (включая все встроенные числовые типы) можно инициализировать или назначать, а затем использовать в следующем примере:

C#

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Классы и структуры могут определять конструкторы, которые принимают параметры, включая первичные конструкторы. Конструкторы, которые принимают параметры, должны вызываться через инструкцию `new` или базовую инструкцию . Классы и структуры также могут определять несколько конструкторов, и ни те, ни другие не обязаны определять конструктор без параметров. Например:

C#

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary) => Salary = annualSalary;

    public Employee(int weeklySalary, int numberOfWeeks) => Salary =
weeklySalary * numberOfWeeks;
}
```

Этот класс можно создать с помощью любой из следующих инструкций:

C#

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Конструктор может использовать ключевое слово `base` для вызова конструктора базового класса. Например:

C#

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

В этом примере конструктор базового класса вызывается перед выполнением блока конструктора. Ключевое слово `base` можно использовать с параметрами или без нее. Любые параметры конструктора можно использовать в качестве параметров для `base` или как часть выражения. Дополнительные сведения см. в базе.

В производном классе, если конструктор базового класса не вызывается явным образом с помощью ключевого слова `base`, конструктор без параметров, если есть один, вызывается неявно. Следующие объявления конструктора фактически одинаковы:

C#

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

C#

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

Если базовый класс не предлагает конструктор без параметров, производный класс должен выполнить явный вызов базового конструктора с помощью `base`.

Конструктор может вызвать другой конструктор в том же объекте с помощью ключевого слова this. Как и `base`, `this` можно использовать с параметрами или без параметров, а любые параметры в конструкторе доступны в качестве параметров для `this` или как часть выражения. Например, второй конструктор в предыдущем примере можно переписать с помощью `this`:

C#

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

Использование ключевого слова `this` в предыдущем примере приводит к вызову следующего конструктора:

C#

```
public Employee(int annualSalary) => Salary = annualSalary;
```

Конструкторы можно пометить как public, private, protected, внутренних, защищенных внутренних или private protected. Эти модификаторы доступа определяют, как пользователи класса могут создавать класс. Дополнительные сведения см. о модификаторах доступа.

Конструктор можно объявить статическим с помощью ключевого слова static. Статические конструкторы вызываются автоматически перед доступом к любым статическим полям и используются для инициализации элементов статического класса. Дополнительные сведения см. в статические конструкторы.

Спецификация языка C#

Дополнительные сведения см. в разделе «Конструкторы экземпляров» и разделе «Статические конструкторы» в спецификации языка C#. Спецификация языка является окончательным источником для синтаксиса И использования C#.

См. также

- система типов C#
- конструкторы
- Финализаторы

Конструкторы экземпляров (руководство по программированию на C#)

Статья • 31.05.2024

Конструктор экземпляра объявляется для указания кода, выполняемого при создании нового экземпляра типа с помощью выражения new. Чтобы инициализировать статический класс или статические переменные в нестатическом классе, можно определить статический конструктор.

Как показано в следующем примере, можно объявить несколько конструкторов экземпляров в одном типе:

C#

```
class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

    public override string ToString() => $"({X},{Y})";
}

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

В предыдущем примере первый конструктор, без параметров, вызывает второй, и оба аргумента равны `0`. Для этого воспользуйтесь ключевым словом `this`.

При объявлении конструктора экземпляра в производном классе можно вызвать конструктор базового класса. Для этого используйте ключевое слово `base`, как показано в следующем примере:

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63
```

```
        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}
```

Конструкторы без параметров

Если у *класса* нет явных конструкторов экземпляров, C# предоставляет конструктор без параметров, который можно использовать для создания экземпляра этого класса, как показано в следующем примере:

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

Этот конструктор инициализирует поля и свойства экземпляров в соответствии с необходимыми инициализаторами. Если поле или свойство не имеет инициализатора, его значение устанавливается в значение по умолчанию для типа поля или свойства. Если в классе объявлен хотя бы один конструктор экземпляра, C# не предоставляет конструктор без параметров.

Тип *структуры* всегда предоставляет конструктор без параметров. Конструктор без параметров — это неявный конструктор без параметров, который создает значение по умолчанию типа или явно объявленный конструктор без параметров. Дополнительные сведения см. в разделе инициализации структуры и значенийпо умолчанию статьи "Типы структур".

Основные конструкторы

Начиная с C# 12, можно объявить основной *конструктор* в классах и конструкциях. Вы помещаете все параметры в скобки по имени типа:

```
C#
```

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

Параметры первичного конструктора находятся в область во всем тексте декларающего типа. Они могут инициализировать свойства или поля. Их можно использовать в качестве переменных в методах или локальных функциях. Их можно передать базовому конструктору.

Основной конструктор указывает, что эти параметры необходимы для любого экземпляра типа. Любой явно написанный конструктор должен использовать `this(...)` синтаксис инициализатора для вызова основного конструктора. Это гарантирует, что основными параметрами конструктора определенно назначены все конструкторы. Для любого `class` типа, включая `record class` типы, неявный конструктор без параметров не создается при наличии основного конструктора. Для любого `struct` типа, включая `record struct` типы, неявный конструктор без параметров всегда создается и всегда инициализирует все поля, включая параметры первичного конструктора, в 0-разрядный шаблон. При написании явного конструктора без параметров он должен вызвать основной конструктор. В этом случае можно указать другое значение для параметров первичного конструктора. В следующем коде показаны примеры основных конструкторов.

```
C#
```

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

Атрибуты можно добавить в синтезированный первичный конструктор, указав целевой `method:` объект для атрибута:

```
C#
```

```
[method: MyAttribute]
public class TaggedWidget(string name)
{
   // details elided
}
```

Если целевой `method` объект не указан, атрибут помещается в класс, а не метод.

В `class` и `struct` типах основные параметры конструктора доступны в любом месте текста типа. Параметр можно реализовать как захваченное частное поле. Если единственными ссылками на параметр являются инициализаторы и вызовы конструктора, этот параметр не фиксируется в частном поле. Используется в других членах типа, что компилятор фиксирует параметр в частном поле.

Если тип включает `record` модификатор, компилятор вместо этого синтезирует общедоступное свойство с тем же именем, что и основной параметр конструктора. Если `record class` для типов основной параметр конструктора использует то же имя, что и базовый конструктор, это свойство является общедоступным свойством базового `record class` типа. Он не дублируется в производном `record class` типе. Эти свойства не создаются для нетипов`record` .

См. также

- Классы, структуры и записи
- Конструкторы
- Методы завершения
- base
- this
- Спецификация основных конструкторов

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Открыть проблему с документацией

Отзыв о продукте

Частные конструкторы (руководство по программированию на C#)

Статья • 02.04.2025

Приватный конструктор — это специальный конструктор экземпляра. Обычно он используется в классах, содержащих только статические элементы. Если класс имеет один или несколько частных конструкторов и нет открытых конструкторов, другие классы (кроме вложенных классов) не могут создавать экземпляры этого класса. Рассмотрим пример.

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Объявление пустого конструктора предотвращает автоматическое создание конструктора без параметров. Обратите внимание, что если с конструктором не используется модификатор доступа, он по-прежнему будет закрытым по умолчанию. Однако частный модификатор обычно используется явно, чтобы ясно, что класс не может быть создан.

Частные конструкторы используются для предотвращения создания экземпляров класса, если нет полей экземпляров или методов, таких как Math класс, или при вызове метода для получения экземпляра класса. Если все методы в классе являются статическими, рекомендуется сделать полный класс статическим. Дополнительные сведения см. в разделе "Статические классы" и "Члены статических классов".

Пример

Ниже приведен пример класса с помощью частного конструктора.

C#

```
public class Counter
{
    private Counter() { }
```

```
    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine($"New count: {Counter.currentCount}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

При раскомментировании следующей инструкции из примера возникнет ошибка, так как конструктор недоступен из-за его уровня доступа.

C#

```
// Counter aCounter = new Counter();   // Error
```

См. также

- Система типов C#
- Конструкторы
- Финализаторы
- частный
- public

Статические конструкторы (Руководство по программированию в C#)

Статья • 02.08.2024

Статический конструктор используется для инициализации любых статических данных или для выполнения определенного действия, которое требуется выполнить только один раз. Он вызывается автоматически до создания первого экземпляра или ссылки на статические элементы. Статический конструктор вызывается по крайней мере один раз.

C#

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Существует несколько действий, которые являются частью статической инициализации. Эти действия выполняются в следующем порядке:

1. *Статические поля имеют значение 0*. Среда выполнения обычно выполняет эту инициализацию.
2. *Запуск* инициализаторов статических полей. Инициализаторы статических полей в большинстве производных типов.
3. *Запуск* инициализаторов статических полей базового типа. Инициализаторы статических полей, начиная с прямой базы с каждым базовым типом System.Object.
4. *Любой статический конструктор выполняется*. Все статические конструкторы из конечного Object.Object базового класса через каждый базовый класс через запуск типа. Порядок выполнения статического конструктора не указан. Однако все статические конструкторы в иерархии выполняются перед созданием экземпляров.

ⓘ Важно!

Существует одно важное исключение из правила, которое выполняется статическим конструктором перед созданием любого экземпляра. Если инициализатор статических полей создает экземпляр типа, этот инициализатор выполняется (включая любой вызов конструктора экземпляра) перед запуском статического конструктора. Это наиболее распространено в однотонном шаблоне*, как показано в* следующем примере:

C#

```
public class Singleton
{
    // Static field initializer calls instance constructor.
    private static Singleton instance = new Singleton();

    private Singleton()
    {
        Console.WriteLine("Executes before static constructor.");
    }

    static Singleton()
    {
        Console.WriteLine("Executes after instance constructor.");
    }

    public static Singleton Instance => instance;
}
```

Инициализатор модуля может быть альтернативой статическому конструктору. Дополнительные сведения см. в спецификации для инициализаторов модулей.

Замечания

Статические конструкторы обладают следующими свойствами.

- Статический конструктор не принимает модификаторы доступа и не имеет параметров.
- Класс или структура могут иметь только один статический конструктор.
- Статические конструкторы не могут быть унаследованы или перегружены.
- Статический конструктор не может вызываться напрямую и должен вызываться только средой CLR. Он вызывается автоматически.
- Пользователь не управляет временем, в течение которого статический конструктор выполняется в программе.

- Статический конструктор вызывается автоматически. Он инициализирует класс перед созданием первого экземпляра или ссылок на какие-либо статические элементы, объявленные в этом классе (не его базовых классах). Статический конструктор выполняется раньше, чем конструктор экземпляра. Если инициализаторы статических полей присутствуют в классе статического конструктора, они выполняются в текстовом порядке, в котором они отображаются в объявлении класса. Инициализаторы выполняются непосредственно перед статическим конструктором.
- Если вы не предоставили статический конструктор для инициализации статических полей, все статические поля инициализируются значениями по умолчанию, которые указаны в разделе Значения по умолчанию типов C#.
- Если статический конструктор создает исключение, среда выполнения не вызывает его во второй раз, а тип остается неинициализируемым в течение времени существования домена приложения. Чаще всего исключение TypeInitializationException вызывается, когда статический конструктор не может создать экземпляр типа или в статическом конструкторе создается необработанное исключение. Для статических конструкторов, не определенных явным образом в исходном коде, может потребоваться проверка кода промежуточного языка (IL).
- Наличие статического конструктора не позволяет добавлять атрибут типа BeforeFieldInit. Это ограничивает возможности оптимизации во время выполнения.
- Поле, объявленное как `static readonly` может быть назначено только в рамках его объявления или в статическом конструкторе. Если явный статический конструктор не требуется, инициализируйте статические поля в объявлении, а не через статический конструктор, чтобы оптимизировать среду выполнения.
- Среда выполнения вызывает статический конструктор не более одного раза для каждого домена приложения. Этот вызов выполняется в заблокированном регионе на основе определенного типа класса. В тексте статического конструктора не требуются дополнительные механизмы блокировки. Чтобы избежать риска взаимоблокировки, не блокируйте текущий поток в статических конструкторах и инициализаторах. Например, не следует ожидать выполнения задач, потоков, обработчиков ожидания или событий, а также создавать блокировки или выполнять блокирующие параллельные операции, как, например, параллельные циклы, `Parallel.Invoke` и запросы Parallel LINQ.

> **Примечание**

Наличие явного статического конструктора следует документировать, несмотря на отсутствие прямого доступа к нему, поскольку это важно для устранения неполадок с исключениями при инициализации.

Использование

- Типичным использованием статических конструкторов является случай, когда класс использует файл журнала и конструктор применяется для добавления записей в этот файл.
- Статические конструкторы также полезны при создании классов-оболочек для неуправляемого кода, когда конструктор может вызвать метод `LoadLibrary`.
- Статические конструкторы также удобны для применения проверок во время выполнения для параметра типа, который не может быть проверен во время компиляции с помощью ограничений типа- параметров.

Пример

В этом примере класс `Bus` имеет статический конструктор. При создании первого экземпляра класса `Bus` (`bus1`) для инициализации класса вызывается статический конструктор. В выходных данных этого примера можно увидеть, что статический конструктор выполняется только один раз, несмотря на то, что создается два экземпляра класса `Bus`. Кроме того, этот конструктор вызывается до выполнения конструктора экземпляра.

C#

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
```

```
        Console.WriteLine("Static constructor sets global start time to
{0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to
simulate
        // actual bus times. Do not do this in your actual bus schedule
program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after
global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
```

```
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

Спецификация языка C#

Дополнительные сведения см. в разделе Статические конструктор в спецификации языка C#.

См. также

- Система типов C#
- Конструкторы
- Статические классы и члены статических классов
- Методы завершения
- Рекомендации по разработке конструкторов
- Предупреждение системы безопасности — CA2121: статические конструкторы должны быть закрытыми
- Инициализаторы модулей

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Практическое руководство. Создание конструктора копий (руководство по программированию на C#)

Статья • 04.07.2024

Для записей C# предоставляет конструктор копирования для объектов, но для классов его необходимо писать самостоятельно.

> **ⓘ Важно!**
>
> Написание конструкторов копирования, работающих для всех производных типов в иерархии классов, может быть сложной задачей. Если класс не `sealed` является, следует настоятельно рассмотреть возможность создания иерархии `record class` типов для использования конструктора копирования, синтезированного компилятором.

Пример

В следующем примере класс определяет конструктор копирования, Person который принимает в качестве аргумента `Person` экземпляр. Значения свойств аргумента присваиваются свойствам нового экземпляра `Person`. Код содержит дополнительный конструктор копии, который отправляет свойства `Name` и `Age` экземпляра, который необходимо скопировать конструктору экземпляра класса. Класс `Person` имеет значение `sealed`, поэтому производные типы не могут быть объявлены, которые могут привести к ошибкам путем копирования только базового класса.

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
```

```
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

См. также

- ICloneable
- Записи
- Система типов C#
- Конструкторы
- Методы завершения

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Члены расширения (руководство по программированию на C#)

Члены расширения позволяют добавлять методы в существующие типы без создания нового производного типа, перекомпиляции или изменения исходного типа.

Начиная с C# 14, существует два синтаксиса, которые используются для определения методов расширения. C# 14 добавляет extension блоки, в которых определяется несколько элементов расширения для типа или экземпляра типа. Перед C# 14 добавьте this модификатор к первому параметру статического метода, чтобы указать, что метод отображается как член экземпляра типа параметра.

Блоки расширений поддерживают несколько типов элементов: методы, свойства и операторы. С помощью блоков расширений можно определить как расширения экземпляров, так и статические расширения. Расширения экземпляров расширяют экземпляр типа; статические расширения расширяют сам тип. Форма методов расширения, объявленных модификатором `this` , поддерживает методы расширения экземпляра.

Методы расширения являются статическими методами, но они вызываются так же, как если бы они были методами экземпляра в расширенном типе. Для клиентского кода, написанного на C#, F# и Visual Basic, нет видимой разницы между вызовом метода расширения и методами, определенными в типе. Обе формы методов расширения компилируются в один и тот же IL (промежуточный язык). Потребители членов расширения не должны знать, какой синтаксис использовался для определения методов расширения.

Наиболее распространенными элементами расширения являются стандартные операторы запросов LINQ, добавляющие функции запросов к существующим System.Collections.IEnumerable и System.Collections.Generic.IEnumerable<T> типам. Чтобы использовать стандартные операторы запросов, сначала подключите их с помощью директивы `using System.Linq`. Затем любой тип, реализующий IEnumerable<T>, по-видимому, имеет такие методы экземпляра, как GroupBy, OrderBy, Average и т. д. Эти дополнительные методы можно увидеть в завершении инструкции IntelliSense, когда вы вводите "точка" после экземпляра типа IEnumerable<T>, например List<T> или Array.

Пример метода OrderBy

В следующем примере показано, как вызвать стандартный метод оператора `OrderBy` запроса в массив целых чисел. Выражение в скобках является лямбда-выражением.

Многие стандартные операторы запросов принимают лямбда-выражения в качестве параметров. Дополнительные сведения см. в разделе Лямбда-выражения.

C#

```
int[] numbers = [10, 45, 15, 39, 21, 26];
IOrderedEnumerable<int> result = numbers.OrderBy(g => g);
foreach (int i in result)
{
    Console.Write(i + " ");
}
//Output: 10 15 21 26 39 45
```

Методы расширения определяются как статические методы, но вызываются с помощью синтаксиса метода экземпляра. Их первый параметр указывает тип, с которым работает метод. Параметр следует этому модификатору. Методы расширения доступны только при явном импорте пространства имен в ваш исходный код с использованием директивы `using`.

Объявление элементов расширения

Начиная с C# 14, можно объявить *блоки расширения*. Блок расширения — это блок в неложенном, негенерическом, статичном классе, который содержит члены расширения для типа или экземпляра этого типа. В следующем примере кода определяется блок расширения для `string` типа. Блок расширения содержит один элемент: метод, который подсчитывает слова в строке:

C#

```
namespace CustomExtensionMembers;

public static class MyExtensions
{
    extension(string str)
    {
        public int WordCount() =>
            str.Split([' ', '.', '?'],
StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Члены расширения в блоке расширения могут использовать открытые или закрытые универсальные шаблоны. Параметры типа могут включать ограничения:

C#

```
public static class MyGenericExtensions
{
    extension<T>(IEnumerable<T> source)
        where T : IEquatable<T>
    {
        public IEnumerable<T> ValuesEqualTo(T threshold)
            => source.Where(x => x.Equals(threshold));
    }
}
```

Перед C# 14 вы объявляете метод расширения путем добавления `this` модификатора в первый параметр:

C#

```
namespace CustomExtensionMethods;

public static class MyExtensions
{
    public static int WordCount(this string str) =>
        str.Split([' ', '.', '?'], StringSplitOptions.RemoveEmptyEntries).Length;
}
```

Обе формы расширений должны быть определены внутри невложенного негенерированного статического класса.

И его можно вызвать из приложения с помощью синтаксиса для доступа к членам экземпляра:

C#

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Хотя члены расширения добавляют новые возможности в существующий тип, члены расширения не нарушают принцип инкапсуляции. Объявления доступа для всех членов расширенного типа применяются к членам расширения.

Класс `MyExtensions` и метод `WordCount` являются `static` и к ним можно получить доступ, как и ко всем остальным членам `static`. Метод `WordCount` можно вызвать так же, как и другие методы `static`, следующим образом:

C#

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

Предыдущий код C# применяется как к блоку расширения, так и к синтаксису `this` для членов расширения. Предыдущий код:

- Объявляет и присваивает новый `string` с именем `s` и значением `"Hello Extension Methods"`.
- Вызывает функцию `MyExtensions.WordCount` с заданным аргументом `s`.

Дополнительные сведения см. в статье о реализации и вызове пользовательского метода расширения.

Как правило, вы вызываете члены расширения функций гораздо чаще, чем реализуете их. Поскольку члены расширения вызываются так, будто они объявлены как члены расширяемого класса, специальных знаний для их использования в клиентском коде не требуется. Чтобы включить члены расширения для определенного типа, достаточно добавить директиву `using` для пространства имен, в котором определены методы. Например, чтобы использовать стандартные операторы запросов, добавьте эту `using` директиву в код:

C#

```
using System.Linq;
```

Привязка членов расширения во время компиляции

Члены расширения можно использовать для расширения класса или интерфейса, но не для переопределения поведения, определенного в классе. Член расширения с тем же именем и подписью, что и у членов интерфейса или класса, никогда не вызывается. Во время компиляции члены расширения всегда имеют более низкий приоритет, чем элементы экземпляра (или статические), определенные в самом типе. Другими словами, если тип имеет метод с именем `Process(int i)`, и у вас есть метод расширения с той же сигнатурой, компилятор всегда привязывается к методу-члену. Когда компилятор обнаруживает вызов элемента, он сначала ищет совпадение в членах типа. Если совпадение не найдено, он ищет какие-либо члены расширения, определенные для типа. Он привязывается к первому элементу расширения, который он находит. В следующем примере демонстрируются правила, которые компилятор C# использует при определении того, следует ли привязать к члену экземпляра данного типа или к члену расширения. Статический класс `Extensions` содержит члены расширения, определенные для любого типа, реализующего `IMyInterface`:

C#

```
public interface IMyInterface
{
    void MethodB();
}

// Define extension methods for IMyInterface.

// The following extension methods can be accessed by instances of any
// class that implements IMyInterface.
public static class Extension
{
    public static void MethodA(this IMyInterface myInterface, int i) =>
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, int
i)");

    public static void MethodA(this IMyInterface myInterface, string s) =>
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, string
s)");

    // This method is never called in ExtensionMethodsDemo1, because each
    // of the three classes A, B, and C implements a method named MethodB
    // that has a matching signature.
    public static void MethodB(this IMyInterface myInterface) =>
        Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
}
```

Эквивалентные расширения можно объявить с помощью синтаксиса члена расширения C# 14:

C#

```
public static class Extension
{
    extension(IMyInterface myInterface)
    {
        public void MethodA(int i) =>
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, int
i)");

        public void MethodA(string s) =>
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface,
string s)");

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public void MethodB() =>
            Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
    }
}
```

Классы `A` и `B` `C` все реализуют интерфейс:

C#

```
// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
class A : IMyInterface
{
    public void MethodB() { Console.WriteLine("A.MethodB()"); }
}

class B : IMyInterface
{
    public void MethodB() { Console.WriteLine("B.MethodB()"); }
    public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
}

class C : IMyInterface
{
    public void MethodB() { Console.WriteLine("C.MethodB()"); }
    public void MethodA(object obj)
    {
        Console.WriteLine("C.MethodA(object obj)");
    }
}
```

`MethodB` Метод расширения никогда не вызывается, так как его имя и подпись точно соответствуют методам, уже реализованным классами. Если компилятор не может найти метод экземпляра с соответствующей сигнатурой, он привязывается к методу соответствующего расширения, если он существует.

C#

```
// Declare an instance of class A, class B, and class C.
A a = new A();
B b = new B();
C c = new C();

// For a, b, and c, call the following methods:
//      -- MethodA with an int argument
//      -- MethodA with a string argument
//      -- MethodB with no argument.

// A contains no MethodA, so each call to MethodA resolves to
// the extension method that has a matching signature.
a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

// A has a method that matches the signature of the following call
// to MethodB.
a.MethodB();            // A.MethodB()
```

```
// B has methods that match the signatures of the following
// method calls.
b.MethodA(1);           // B.MethodA(int)
b.MethodB();            // B.MethodB()

// B has no matching method for the following call, but
// class Extension does.
b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

// C contains an instance method that matches each of the following
// method calls.
c.MethodA(1);           // C.MethodA(object)
c.MethodA("hello");     // C.MethodA(object)
c.MethodB();            // C.MethodB()
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

Распространенные шаблоны использования

Функции коллекции

В прошлом было распространено создание классов коллекции, реализующих System.Collections.Generic.IEnumerable<T> интерфейс для заданного типа и содержащих функциональные возможности, которые действовали в коллекциях этого типа. Хотя при создании этого типа объекта коллекции нет ничего плохого, с помощью расширения на объекте System.Collections.Generic.IEnumerable<T>коллекции можно добиться одной и той же функциональности. Расширения имеют преимущество, позволяя вызывать функциональные возможности из любой коллекции, такой как System.Array или System.Collections.Generic.List<T>, которая реализует System.Collections.Generic.IEnumerable<T> для этого типа. Пример использования массива Int32 можно найти ранее в этой статье.

Layer-Specific Функциональные возможности

При использовании архитектуры onion или другого многоуровневого приложения обычно используется набор сущностей домена или объектов передачи данных, которые можно использовать для обмена данными между границами приложения. Эти объекты

обычно не содержат функциональных возможностей или только минимальных функциональных возможностей, которые применяются ко всем слоям приложения. Методы расширения можно использовать для добавления функциональных возможностей, относящихся к каждому уровню приложений.

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Можно объявить эквивалентное `FullName` свойство в C# 14 и более поздних версиях с помощью нового синтаксиса блока расширения:

C#

```
static class DomainEntityExtensions
{
    extension(DomainEntity value)
    {
        string FullName => $"{value.FirstName} {value.LastName}";
    }
}
```

Расширение предопределенных типов

Вместо создания новых объектов при создании повторно используемых функциональных возможностей часто можно расширить существующий тип, например тип .NET или CLR. Например, если вы не используете методы расширения, можно создать `Engine` или `Query` класс для выполнения запроса на SQL Server, который может вызываться из нескольких мест в нашем коде. Однако вместо этого можно расширить System.Data.SqlClient.SqlConnection класс с помощью методов расширения для выполнения этого запроса в любом месте, где есть подключение к SQL Server. Другие примеры могут быть для добавления общих функций в System.String класс, расширения возможностей System.IO.Stream обработки данных объекта и System.Exception объектов для конкретных функций обработки ошибок. Эти типы вариантов использования ограничены только вашим воображением и хорошим смыслом.

Расширение предопределённых типов может быть затруднительно с типами `struct`, так как они передаются по значению в методы. Это означает, что любые изменения структуры вносятся в копию структуры. Эти изменения не отображаются после выхода метода расширения. Вы можете добавить модификатор `ref` к первому аргументу, что сделает его методом расширения `ref`. Ключевое `ref` слово может отображаться до или после ключевого `this` слова без каких-либо семантических различий. `ref` Добавление модификатора указывает, что первый аргумент передается по ссылке. Этот метод позволяет создавать методы расширения, изменяющие состояние расширенной структуры (обратите внимание, что частные члены недоступны). Только типы значений или универсальные типы, ограниченные структурой (Дополнительные сведения об этих правилах см. в статье struct об ограничении) разрешены в качестве первого параметра `ref` метода расширения или в качестве получателя блока расширения. В следующем примере показано, как использовать `ref` метод расширения для непосредственного изменения встроенного типа без необходимости переназначить результат или передать его через функцию с `ref` ключевым словом:

C#

```
public static class IntExtensions
{
    public static void Increment(this int number)
        => number++;

    // Take note of the extra ref keyword here
    public static void RefIncrement(this ref int number)
        => number++;
}
```

Эквивалентные блоки расширения показаны в следующем коде:

C#

```
public static class IntExtensions
{
    extension(int number)
    {
        public void Increment()
            => number++;
    }

    // Take note of the extra ref keyword here
    extension(ref int number)
    {
        public void RefIncrement()
            => number++;
    }
}
```

Различные блоки расширений требуются для различения режимов параметров по значению и по ссылке для приемника.

Вы можете увидеть разницу, которую применение `ref` к получателю имеет на следующем примере:

C#

```
int x = 1;

// Takes x by value leading to the extension method
// Increment modifying its own copy, leaving x unchanged
x.Increment();
Console.WriteLine($"x is now {x}"); // x is now 1

// Takes x by reference leading to the extension method
// RefIncrement changing the value of x directly
x.RefIncrement();
Console.WriteLine($"x is now {x}"); // x is now 2
```

Вы можете применить тот же метод, добавив `ref` члены расширения в определяемые пользователем типы структур:

C#

```
public struct Account
{
    public uint id;
    public float balance;

    private int secret;
}

public static class AccountExtensions
{
    // ref keyword can also appear before the this keyword
    public static void Deposit(ref this Account account, float amount)
    {
        account.balance += amount;

        // The following line results in an error as an extension
        // method is not allowed to access private members
        // account.secret = 1; // CS0122
    }
}
```

Предыдущий пример также можно создать с помощью блоков расширений в C# 14:

C#

```
public static class AccountExtensions
{
    extension(ref Account account)
    {
        // ref keyword can also appear before the this keyword
        public void Deposit(float amount)
        {
            account.balance += amount;

            // The following line results in an error as an extension
            // method is not allowed to access private members
            // account.secret = 1; // CS0122
        }
    }
}
```

Эти методы расширения можно получить следующим образом:

C#

```
Account account = new()
{
    id = 1,
    balance = 100f
};

Console.WriteLine($"I have ${account.balance}"); // I have $100

account.Deposit(50f);
Console.WriteLine($"I have ${account.balance}"); // I have $150
```

Общие рекомендации

Рекомендуется добавить функциональные возможности, изменив код объекта или создав новый тип всякий раз, когда это разумно и возможно. Методы расширения являются важным вариантом для создания повторно используемых функций во всей экосистеме .NET. Члены расширения предпочтительнее, если исходный источник не находится под вашим контролем, если производный объект является неуместным или невозможным, или если функциональность имеет ограниченную область.

Дополнительные сведения о производных типах см. в разделе "Наследование".

Если вы реализуете методы расширения для данного типа, помните следующие моменты:

- Метод расширения не вызывается, если он имеет ту же сигнатуру, что и метод, определенный в типе.

- Методы расширения попадают в область видимости на уровне пространства имен. Например, если у вас есть несколько статических классов, содержащих методы расширения в одном пространстве имен, `Extensions` все из них попадают в область директивы `using Extensions;` .

Для реализованной библиотеки классов не следует использовать методы расширения, чтобы избежать увеличения числа версий сборки. Если вы хотите добавить значительные функциональные возможности в библиотеку, для которой принадлежит исходный код, следуйте рекомендациям .NET по управлению версиями сборок. Дополнительные сведения см. в разделе "Управление версиями сборок".

См. также

- Примеры параллельного программирования (многие примеры демонстрируют методы расширения)
- Лямбда-выражения
- Обзор операторов стандартных запросов

Last updated on 20.11.2025

Финализаторы (руководство по программированию в C#)

Методы завершения (также называемые **деструкторами**) используются для любой необходимой окончательной очистки, когда сборщик мусора собирает экземпляр класса. В большинстве случаев можно избежать написания финализатора с помощью System.Runtime.InteropServices.SafeHandle или производных классов для упаковки любого неуправляемого дескриптора.

Замечания

- В структурах нельзя определять финализаторы. Они применяются только в классах.
- Каждый класс может иметь только один финализатор.
- Финализаторы не могут быть унаследованы или перегружены.
- Финализаторы невозможно вызвать. Они запускаются автоматически.
- Финализатор не принимает модификаторов и не имеет параметров.

Например, ниже показано объявление метода завершения для класса `Car`.

C#

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

Метод завершения можно также реализовать как определение тела выражения, как показано в следующем примере.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is executing.");
}
```

Финализатор неявно вызывает метод Finalize для базового класса объекта. В связи с этим вызов финализатора неявно преобразуется в следующий код:

C#

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Это означает, что метод `Finalize` вызывается рекурсивно для всех экземпляров цепочки наследования начиная с самого дальнего и заканчивая самым первым.

> ⓘ **Примечание.**
>
> Пустые финализаторы использовать нельзя. Если класс содержит финализатор, то в очереди `Finalize` создается запись. Эта очередь обрабатывается сборщиком мусора. Когда сборщик мусора обрабатывает очередь, он вызывает каждый финализатор. Ненужные финализаторы, включая пустые финализаторы, финализаторы, которые вызывают только финализатор базового класса, или финализаторы, которые вызывают только условно генерируемые методы, приводят к ненужной потере производительности.

Программист не имеет контроля над тем, когда вызывается финализатор; сборщик мусора решает, когда его вызвать. Сборщик мусора проверяет наличие объектов, которые больше не используются приложением. Если он считает, что какой-либо объект требует уничтожения, то вызывает метод завершения (при наличии) и освобождает память, используемую для хранения этого объекта. Принудительная сборка мусора возможна вызовом Collect, но в большинстве случаев этого следует избегать из-за потенциальных проблем с производительностью.

> ⓘ **Примечание.**
>
> От **конкретной реализации .NET** зависит, выполняются ли методы завершения как часть завершения работы приложения. При завершении работы приложения .NET Framework прилагает все возможные усилия для вызова финализаторов объектов, которые еще не были собраны сборщиком мусора, если такая очистка не была подавлена (например, вызовом метода библиотеки `GC.SuppressFinalize`). .NET 5 (включая .NET Core) и более поздние версии не вызывают финализаторы при

завершении приложения. Дополнительные сведения об этой проблеме см. в Проблема dotnet/csharpstandard #291, рассмотренная на сайте GitHub.

Если необходимо надежно выполнить очистку при завершении работы приложения, зарегистрируйте обработчик для события System.AppDomain.ProcessExit. Этот обработчик обеспечит вызов IDisposable.Dispose() (или, IAsyncDisposable.DisposeAsync()) для всех объектов, для которых требуется очистка перед выходом из приложения. Поскольку нельзя вызвать *Finalize* напрямую и нельзя гарантировать, что сборщик мусора вызовет все финализаторы до выхода, необходимо использовать `Dispose` или `DisposeAsync`, чтобы освободить ресурсы.

Использование финализаторов для освобождения ресурсов

В целом язык C# не требует управления памятью в той степени, в какой это требуется в случае разработки кода на языке, не рассчитанном на среду выполнения со сборкой мусора. Это связано с тем, что сборщик мусора платформы .NET неявным образом управляет выделением и высвобождением памяти для объектов. Однако при инкапсуляции приложением неуправляемых ресурсов, например окон, файлов и сетевых подключений, для высвобождения этих ресурсов следует использовать методы завершения. Когда объект подлежит завершению, сборщик мусора инициирует выполнение метода `Finalize` этого объекта.

> **⚠ Предупреждение**
>
> Не обращаться к элементам управляемого объекта из средства завершения. Во время завершения управляемые объекты уже могут быть удалены, что делает их недоступными или в недопустимом состоянии. Доступ только к неуправляемым ресурсам непосредственно из финализаторов.

Явное освобождение ресурсов

В случае, когда приложением используется ценный внешний ресурс, также рекомендуется обеспечить способ высвобождения этого ресурса явным образом, прежде чем сборщик мусора освободит объект. Для высвобождения ресурса реализуется метод `Dispose` интерфейса IDisposable, который выполняет необходимую для объекта очистку. Это может значительно повысить производительность приложения. Даже при наличии

явного управления ресурсами, финализатор становится резервным средством для очистки ресурсов, если вызов метода `Dispose` не удастся выполнить.

Дополнительные сведения об очистке ресурсов см. в следующих статьях:

- Очистка неуправляемых ресурсов
- Реализация метода Dispose
- Реализация метода DisposeAsync
- Заявление using

Пример

В приведенном ниже примере создаются три класса, образующих цепочку наследования. Класс `First` является базовым, класс `Second` является производным от класса `First`, а класс `Third` является производным от класса `Second`. Все три имеют финализаторы. В `Main` создается экземпляр наиболее производного класса. Выходные данные этого кода зависят от того, на какую реализацию .NET нацелено приложение:

- .NET Framework: выходные данные показывают, что методы завершения для трех классов вызываются автоматически при завершении работы приложения, в порядке от наиболее производного до наименее производного.
- .NET 5 (включая .NET Core) или более поздняя версия: выходные данные отсутствуют, так как эта реализация .NET не вызывает финализаторы при завершении работы приложения.

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
```

```
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

Спецификация языка C#

Дополнительные сведения см. в разделе "Методы завершения" спецификации языка C#.

См. также

- IDisposable
- Конструкторы
- Сборка мусора

Last updated on 24.02.2026

Инициализаторы объектов и коллекций (руководство по программированию на C#)

C# позволяет создавать экземпляры объекта или коллекции и выполнять назначения членов в одной инструкции.

Инициализаторы объектов

Инициализаторы объектов позволяют назначать значения любым доступным полям или свойствам объекта при его создании. Вам не нужно вызывать конструктор, а затем использовать инструкции назначения. Синтаксис инициализатора объектов позволяет указать аргументы конструктора или опустить аргументы и скобки. Рекомендации по использованию инициализаторов объектов последовательно см. в разделе Use object initializers (правило стиля IDE0017). В следующем примере показано, как использовать инициализатор объектов с именованным типом `Cat` и как вызвать конструктор без параметров. Обратите внимание на использование автоматически реализованных свойств в `Cat` классе. Дополнительные сведения см. в разделе "Автоматически реализованные свойства".

C#

```
public class Cat
{
    // Automatically implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

Синтаксис инициализатора объектов позволяет создать экземпляр и назначить только что созданный объект со своими назначенными свойствами переменной в назначении.

Начиная с свойств вложенных объектов, можно использовать синтаксис инициализатора объектов без ключевого `new` слова. Этот синтаксис `Property = { ... }` позволяет инициализировать элементы существующих вложенных объектов, которые полезны только для чтения. Для получения дополнительной информации см. Инициализаторы объектов с свойствами типа класса.

Инициализаторы объектов могут задавать индексаторы, а также назначать поля и свойства. Рассмотрим следующий базовый класс `Matrix`:

C#

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

Матрицу удостоверений можно инициализировать с помощью следующего кода:

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Любой доступный индексатор, который содержит доступный метод задания, можно использовать как одно из выражений в инициализаторе объекта независимо от количества или типов аргументов. Аргументы индекса формируют левую часть

присваивания, а значение стоит в его правой части. Например, следующие инициализаторы действительны, если у `IndexersExample` есть соответствующие индексаторы:

C#

```
var thing = new IndexersExample
{
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

Для компиляции приведенного выше кода тип `IndexersExample` должен иметь следующие члены:

C#

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

Инициализаторы объектов с анонимными типами

Хотя вы можете использовать инициализаторы объектов в любом контексте, они особенно полезны в выражениях Language Integrated Query (LINQ). Выражения запросов часто используют анонимные типы, которые можно инициализировать только с помощью инициализатора объектов, как показано в следующем объявлении.

C#

```
var pet = new { Age = 10, Name = "Fluffy" };
```

С помощью анонимных типов `select` предложение в выражении запроса LINQ может преобразовать объекты исходной последовательности в объекты, значения и фигуры которых отличаются от исходного. Может потребоваться сохранить только часть информации из каждого объекта в последовательности. В следующем примере предположим, что объект продукта (`p`) содержит множество полей и методов, и вы

заинтересованы только в создании последовательности объектов, содержащих имя продукта и цену единицы.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

При выполнении этого запроса переменная `productInfos` содержит последовательность объектов, доступ к которым можно получить в операторе `foreach`, как показано в примере.

C#

```
foreach(var p in productInfos){...}
```

Каждый объект нового анонимного типа содержит два общедоступных свойства, имеющих те же имена, что и у свойств или полей исходного объекта. При создании анонимного типа можно также переименовать поле. В следующем примере поле `UnitPrice` переименовывается в `Price`.

C#

```
select new {p.ProductName, Price = p.UnitPrice};
```

Инициализаторы объектов с модификатором `required`

Используйте ключевое `required` слово для принудительного задания значения свойства или поля с помощью инициализатора объектов. Вам не нужно задавать необходимые свойства в качестве параметров конструктора. Компилятор гарантирует, что все вызывающие функции инициализируют эти значения.

C#

```
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
```

```
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object initializer or
attribute constructor.
// var pet = new Pet();
```

Обычно рекомендуется гарантировать правильность инициализации объекта, особенно если у вас есть несколько полей или свойств для управления и не хотят включать их в конструктор.

Инициализаторы объектов с аксессором `init`

Используя аксессор `init`, вы можете убедиться, что объект не изменяется после инициализации. Это помогает ограничить установку значения свойства.

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; init; }
}

// The `LastName` property can be set only during initialization. It CAN'T be
modified afterwards.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852  Init - only property or indexer 'Person.LastName' can only be
assigned in an object initializer,
//               or on 'this' or 'base' in an instance constructor or an 'init'
accessor.
// pet.LastName = "Kowalski";
```

Свойства "только для инициализации" поддерживают неизменяемые структуры, обеспечивая естественный синтаксис для пользователей типа.

Инициализаторы объектов со свойствами типа класса

При инициализации объектов с типизированными свойствами можно использовать два разных синтаксиса:

1. **Инициализатор объектов без `new` ключевого слова**: `Property = { ... }`
2. **Инициализатор объектов с `new` ключевым словом**: `Property = new() { ... }`

Эти синтаксисы ведут себя по-разному. В следующем примере демонстрируется оба подхода:

C#

```
public class HowToClassTypedInitializer
{
    public class EmbeddedClassTypeA
    {
        public int I { get; set; }
        public bool B { get; set; }
        public string S { get; set; }
        public EmbeddedClassTypeB ClassB { get; set; }

        public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

        public EmbeddedClassTypeA()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeA constructor. Values
are: {this}");
            I = 3;
            B = true;
            S = "abc";
            ClassB = new() { BB = true, BI = 43 };
            Console.WriteLine($"Exiting EmbeddedClassTypeA constructor. Values are:
{this})");
        }
    }

    public class EmbeddedClassTypeB
    {
        public int BI { get; set; }
        public bool BB { get; set; }
        public string BS { get; set; }

        public override string ToString() => $"{BI}|{BB}|{BS}";

        public EmbeddedClassTypeB()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeB constructor. Values
are: {this}");
            BI = 23;
            BB = false;
            BS = "BBBabc";
            Console.WriteLine($"Exiting EmbeddedClassTypeB constructor. Values are:
{this})");
```

```
        }
    }

    public static void Main()
    {
        var a = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = { BI = 100003 }
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

        var a2 = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = new() { BI = 100003 } //New instance
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA a2: {a2}");
    }

    // Output:
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //After initializing EmbeddedClassTypeA: 103|False|abc|||100003|True|BBBabc
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //After initializing EmbeddedClassTypeA a2: 103|False|abc|||100003|False|BBBabc
}
```

Основные отличия

- **Без `new` ключевого слова** (`ClassB = { BI = 100003 }`): этот синтаксис изменяет существующий экземпляр свойства, созданного конструктором объектов. Он вызывает инициализаторы членов в существующем объекте.
- **С `new` ключевым словом** (`ClassB = new() { BI = 100003 }`): этот синтаксис создает новый экземпляр и назначает его свойству, заменив любой существующий экземпляр.

Инициализатор без `new` повторно использует текущий экземпляр. В предыдущем примере значения ClassB: `100003` (новое назначенное значение) `true` (сохранено из инициализации EmbeddedClassTypeA) `BBBabc` (без изменений по умолчанию из EmbeddedClassTypeB).

Инициализаторы объектов без `new`, применимые к свойствам только для чтения

Синтаксис без `new` полезен при работе со свойствами только для чтения. Вы не можете создать новый экземпляр, но вы по-прежнему можете инициализировать члены существующего экземпляра:

C#

```
public class ReadOnlyPropertyExample
{
    public class Settings
    {
        public string Theme { get; set; } = "Light";
        public int FontSize { get; set; } = 12;
    }

    public class Application
    {
        public string Name { get; set; } = "";
        // This property is read-only - it can only be set during construction
        public Settings AppSettings { get; } = new();
    }

    public static void Example()
    {
        // You can still initialize the nested object's properties
        // even though AppSettings property has no setter
        var app = new Application
        {
            Name = "MyApp",
            AppSettings = { Theme = "Dark", FontSize = 14 }
        };

        // This would cause a compile error because AppSettings has no setter:
        // app.AppSettings = new Settings { Theme = "Dark", FontSize = 14 };

        Console.WriteLine($"App: {app.Name}, Theme: {app.AppSettings.Theme}, Font
Size: {app.AppSettings.FontSize}");
    }
}
```

Этот подход позволяет инициализировать вложенные объекты, даже если у содержащегося свойства нет метода задания.

Инициализаторы коллекций

Инициализаторы коллекции позволяют указать один или несколько инициализаторов элементов при инициализации типа коллекции, реализующего IEnumerable и имеющий `Add` метод с соответствующим сигнатурой в качестве метода экземпляра или метода расширения. Инициализаторы элементов могут быть значением, выражением или инициализатором объектов. С помощью инициализатора коллекции вам не нужно указывать несколько вызовов; компилятор автоматически добавляет вызовы. Рекомендации по использованию инициализаторов коллекций последовательно см. в статье Use Collection Initializers (style rule IDE0028). Инициализаторы коллекции также полезны в запросах LINQ.

Ниже приведен пример двух простых инициализаторов коллекций.

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

В следующем инициализаторе коллекции используются инициализаторы объектов класса `Cat`, определенного в предыдущем примере. Инициализаторы отдельных объектов заключены в фигурные скобки и разделены запятыми.

C#

```
List<Cat> cats =
[
    new Cat { Name = "Sylvester", Age = 8 },
    new Cat { Name = "Whiskers", Age = 2 },
    new Cat { Name = "Sasha", Age = 14 }
];
```

В качестве элемента инициализатора коллекции можно указать значение null, если метод `Add` коллекции допускает это.

C#

```
List<Cat?> moreCats = new List<Cat?>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
```

```
    null
};
```

С помощью элемента spread можно создать один список, копирующий другой список или списки.

C#

```
List<Cat> allCats = [.. cats, .. moreCats];
```

И включите дополнительные элементы вместе с использованием распределенного элемента.

C#

```
List<Cat> additionalCats = [.. cats, new Cat { Name = "Furrytail", Age = 5 }, ..
moreCats];
```

Можно указать индексированные элементы, если коллекция поддерживает индексирование чтения и записи.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

В предыдущем примере создается код, который вызывает Item[TKey] для задания значений. Вы также можете инициализировать словари и другие ассоциативные контейнеры с помощью следующего синтаксиса. Вместо синтаксиса индексатора с скобками и назначением используется объект с несколькими значениями:

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

В этом примере инициализатор вызывает Add(TKey, TValue) для добавления трех элементов в словарь. Эти два разных способа инициализации ассоциативных коллекций

немного отличаются из-за вызовов методов, которые создает компилятор. Оба варианта могут работать с классом `Dictionary`. Другие типы могут поддерживать только одну или другую на основе общедоступного API.

Аргументы выражения коллекции

Начиная с C# 15, используйте `with(...)` элемент в качестве первого элемента в выражении коллекции для передачи аргументов конструктору коллекции. С помощью этой функции можно указать вместимость, сравниватели или другие параметры конструктора непосредственно в синтаксисе выражения коллекции.

C#

```
public static void CollectionExpressionWithArgumentsExample()
{
    string[] values = ["one", "two", "three"];

    // Use with() to pass capacity to the List<T> constructor
    List<string> names = [with(capacity: values.Length * 2), .. values];

    Console.WriteLine($"Created List<string> with capacity: {names.Capacity}");
    Console.WriteLine($"List contains {names.Count} elements: [{string.Join(", ",
names)}]");

    // Use with() to pass a comparer to the HashSet<T> constructor
    HashSet<string> caseInsensitiveSet = [with(StringComparer.OrdinalIgnoreCase),
"Hello", "HELLO"];
    // caseInsensitiveSet contains only one element because "Hello" and "HELLO" are
equal

    Console.WriteLine($"HashSet with case-insensitive comparer contains
{caseInsensitiveSet.Count} elements: [{string.Join(", ", caseInsensitiveSet)}]");
    Console.WriteLine("Note: 'Hello' and 'HELLO' are treated as the same element
due to OrdinalIgnoreCase comparer");
}
```

Дополнительные сведения о аргументах выражения коллекции, включая поддерживаемые целевые типы и ограничения, см. в аргументах выражения коллекции.

Инициализаторы объектов с инициализацией свойств коллекции, доступных только для чтения

Некоторые классы имеют свойства коллекции, в которых свойство доступно только для чтения, например `Cats` свойство `CatOwner` в следующем случае:

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

Невозможно использовать синтаксис инициализатора коллекции, рассмотренный ранее, так как свойство не может быть назначено новому списку:

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

Однако, вы можете добавить новые записи в `Cats`, используя синтаксис инициализации и пропустив создание списка (`new List<Cat>`), как показано в следующем примере:

C#

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

Набор записей для добавления отображается в виде фигурных скобок. Предыдущий код идентичен написанию:

C#

```
CatOwner owner = new ();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Примеры

В следующем примере объединяются понятия инициализаторов коллекций и объектов.

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Automatically implemented properties.
        public int Age { get; set; }
        public string? Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats =
        [
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        ];

        List<Cat?> moreCats = new List<Cat?>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        List<Cat> allCats = [.. cats, new Cat { Name = "Łapka", Age = 5 }, cat, ..
moreCats];

        // Display results.
        foreach (Cat? c in allCats)
        {
            if (c != null)
            {
                System.Console.WriteLine(c.Name);
            }
            else
            {
```

```
                System.Console.WriteLine("List element has null value.");
            }
        }
    }
    // Output:
    // Sylvester
    // Whiskers
    // Sasha
    // Łapka
    // Fluffy
    // Furrytail
    // Peaches
    // List element has null value.
}
```

В следующем примере показан объект, реализующий IEnumerable и содержащий `Add` метод с несколькими параметрами. Он использует инициализатор коллекции с несколькими элементами в списке, соответствующими сигнатуре метода `Add`.

C#

```
public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new();
        public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
            string street, string city,
            string state, string zipcode) => internalList.Add($"""
            {firstname} {lastname}
            {street}
            {city}, {state} {zipcode}
            """
            );
    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)
```

```
        {
            Console.WriteLine("\r\n" + addressEntry);
        }
    }

    /*
        * Prints:

        Address Entries:

        John Doe
        123 Street
        Topeka, KS 00000

        Jane Smith
        456 Street
        Topeka, KS 00000
        */
}
```

В методах `Add` можно использовать ключевое слово `params`, чтобы принимать переменное число аргументов, как показано в приведенном ниже примере. Здесь также демонстрируется пользовательская реализация индексатора, которая также применяется для инициализации коллекции с помощью индексов. Начиная с C# 13 `params` параметр не ограничен массивом. Это может быть тип коллекции или интерфейс.

C#

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);
```

```
        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue>?
storedValues))
            {
                internalDictionary.Add(key, storedValues = new());
            }
            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a
collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a
collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
```

```
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a
collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
        * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer
using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer
using indexing:
```

```
        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
        */
}
```

Last updated on 11.02.2026

Как инициализировать объекты с помощью инициализатора объектов (руководство по программированию на C#)

03.07.2025

Инициализаторы объектов можно использовать для инициализации объектов типа декларативным образом без явного вызова конструктора для типа.

В следующих примерах показано, как использовать инициализаторы объектов с именованными объектами. Компилятор обрабатывает инициализаторы объектов, сначала обращаюсь к конструктору экземпляра без параметров, а затем обработать инициализации элементов. Таким образом, если конструктор без параметров объявлен как `private` в классе, общедоступные инициализаторы объектов приведут к ошибке.

При определении анонимного типа необходимо использовать инициализатор объектов. Дополнительные сведения см. в разделе "Как возвращать подмножества свойств элемента в запросе".

Пример

В следующем примере показано, как инициализировать новый `StudentName` тип с помощью инициализаторов объектов. В этом примере задаются свойства в типе `StudentName`:

C#

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The parameterless constructor
is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
```

```
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the parameterless constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }
```

```
        public override string ToString() => FirstName + "   " + ID;
    }
}
```

Инициализаторы объектов можно использовать для задания индексаторов в объекте. В следующем примере определяется класс `BaseballTeam`, использующий индексатор для получения и задания игроков на разных позициях. Инициализатор может назначать игроков на основе аббревиатуры позиции или номера, используемого для каждой позиции на бейсбольных карточках счёта.

C#

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

В следующем примере показано порядок выполнения конструктора и инициализации элементов с помощью конструктора и без параметра:

C#

```
public class ObjectInitializersExecutionOrder
{
    public static void Main()
    {
        new Person { FirstName = "Paisley", LastName = "Smith", City = "Dallas" };
        new Dog(2) { Name = "Mike" };
    }

    public class Dog
    {
        private int age;
        private string name;

        public Dog(int age)
        {
            Console.WriteLine("Hello from Dog's non-parameterless constructor");
            this.age = age;
        }

        public required string Name
        {
            get { return name; }

            set
            {
                Console.WriteLine("Hello from setter of Dog's required property
'Name'");
                name = value;
            }
        }
    }

    public class Person
    {
        private string firstName;
        private string lastName;
        private string city;

        public Person()
        {
            Console.WriteLine("Hello from Person's parameterless constructor");
        }

        public required string FirstName
        {
            get { return firstName; }

            set
            {
                Console.WriteLine("Hello from setter of Person's required property
'FirstName'");
                firstName = value;
```

```
            }
        }

        public string LastName
        {
            get { return lastName; }

            init
            {
                Console.WriteLine("Hello from setter of Person's init property
'LastName'");
                lastName = value;
            }
        }

        public string City
        {
            get { return city; }

            set
            {
                Console.WriteLine("Hello from setter of Person's property
'City'");
                city = value;
            }
        }
    }

    // Output:
    // Hello from Person's parameterless constructor
    // Hello from setter of Person's required property 'FirstName'
    // Hello from setter of Person's init property 'LastName'
    // Hello from setter of Person's property 'City'
    // Hello from Dog's non-parameterless constructor
    // Hello from setter of Dog's required property 'Name'
}
```

Инициализаторы объектов без ключевого `new` слова

Можно также использовать синтаксис инициализатора объектов без `new` ключевого слова для инициализации свойств вложенных объектов. Этот синтаксис особенно полезен со свойствами только для чтения:

C#

```
public class ObjectInitializerWithoutNew
{
    public class Address
    {
```

```
        public string Street { get; set; } = "";
        public string City { get; set; } = "";
        public string State { get; set; } = "";
    }

    public class Person
    {
        public string Name { get; set; } = "";
        public Address HomeAddress { get; set; } = new(); // Property with setter
    }

    public static void Examples()
    {
        // Example 1: Using object initializer WITHOUT 'new' keyword
        // This modifies the existing Address instance created in the constructor
        var person1 = new Person
        {
            Name = "Alice",
            HomeAddress = { Street = "123 Main St", City = "Anytown", State = "CA"
}
        };

        // Example 2: Using object initializer WITH 'new' keyword
        // This creates a completely new Address instance
        var person2 = new Person
        {
            Name = "Bob",
            HomeAddress = new Address { Street = "456 Oak Ave", City =
"Somewhere", State = "NY" }
        };

        // Both approaches work, but they behave differently:
        // - person1.HomeAddress is the same instance that was created in Person's
constructor
        // - person2.HomeAddress is a new instance, replacing the one from the
constructor

        Console.WriteLine($"Person 1: {person1.Name} at
{person1.HomeAddress.Street}, {person1.HomeAddress.City},
{person1.HomeAddress.State}");
        Console.WriteLine($"Person 2: {person2.Name} at
{person2.HomeAddress.Street}, {person2.HomeAddress.City},
{person2.HomeAddress.State}");
    }
}
```

Этот подход изменяет существующий экземпляр вложенного объекта, а не создает новый. Дополнительные сведения и примеры см. в разделе "Инициализаторы объектов" с типизированными свойствами класса.

См. также

- Инициализаторы объектов и коллекций

Практическое руководство. Инициализация словаря с помощью инициализатора коллекции (Руководство по программированию на C#)

18.10.2025

Dictionary<TKey,TValue> содержит коллекцию пар «ключ-значение». Ее метод Add принимает два параметра: один для ключа, другой — для значения. Для инициализации Dictionary<TKey,TValue> или любой коллекции, метод `Add` которой принимает несколько параметров, можно заключить каждый набор параметров в скобки, как показано в приведенном ниже примере. Другой вариант — использовать инициализатор индекса, как показано также в следующем примере.

> **Примечание**
>
> Основное различие между этими двумя способами инициализации коллекции заключается в том, как обрабатываются повторяющиеся ключи, например:
>
> C#
>
> ```
> { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
> { 111, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 }
> },
> ```
>
> метод Add вызывает ArgumentException: `'An item with the same key has already been added. Key: 111'`, а вторая часть примера — общедоступный метод индексатора чтения и записи, тихо перезаписывает уже имеющуюся запись с тем же ключом.

Пример

В следующем примере кода Dictionary<TKey,TValue> инициализируется экземплярами типа `StudentName`. Первая инициализация использует метод `Add` с двумя аргументами. Компилятор создает вызов метода `Add` для каждой пары ключа `int` и значения `StudentName`. Во втором случае используется общедоступный метод индексатора для чтения и записи класса `Dictionary`.

C#

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211
} },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova",
ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName}
{students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova",
ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName}
{students2[index].LastName}");
        }
    }
}
```

Обратите внимание на две пары фигурных скобок в каждом элементе коллекции в первом объявлении. Внутренние фигурные скобки заключают инициализатор объекта для `StudentName`, а внешние фигурные скобки заключают инициализатор для пары "ключ-значение", добавляемой в `students` Dictionary<TKey,TValue>. И наконец, целый инициализатор коллекции для словаря заключается в фигурных скобках. Во второй

инициализации левая часть оператора присваивания является ключом, а правая часть — значением, использующим инициализатор объекта для `StudentName`.

Использование искусственного интеллекта для создания тестовых данных для коллекций словарей

Вы можете использовать средства ИИ, такие как GitHub Copilot, для быстрого создания тестовых данных словаря и сценариев проверки в проектах C#.

Ниже приведен пример запроса, который можно использовать в Visual Studio Copilot Chat.

Запрос Copilot

```
Generate data collections for tests to create a separate Dictionary<int, Student>
containing 10 valid Student records and 5 invalid records.
- Valid records should have realistic Name and Grade values.
- Invalid records should include cases such as missing Name, Grade < 0, or Grade >
100.
- This dictionary should be used only for testing purposes and not modify existing
production code.
- Generate test code that utilizes this test data for validation scenarios.
- Call test method to run the test.
```

Ознакомьтесь с предложениями Copilot, прежде чем применять их.

Дополнительные сведения о GitHub Copilot см. в часто задаваемых вопросых о GitHub.

См. также

- Инициализаторы объектов и коллекций

Вложенные типы (руководство по программированию на C#)

Вложенный тип — это тип, который определяется в классе, структуре или интерфейсе. Рассмотрим пример.

C#

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Независимо от того, является ли внешний тип классом, интерфейсом или структурой, вложенные типы по умолчанию закрыты. Доступ к ним можно получить только из типа, который их содержит. В предыдущем примере внешние типы не могут получить доступ к классу `Nested` .

Можно также указать модификатор доступа , чтобы определить специальные возможности вложенного типа, как показано ниже.

- Вложенные типы **класса** могут быть общедоступными, защищенными, внутренними, защищенными внутренними, частными или частными.

  Однако определение `protected`, `protected internal` или `private protected` вложенного класса внутри закрытого класса создает предупреждение компилятора CS0628, "новый защищенный член, объявленный в закрытом классе".

  Также имейте в виду, что делать вложенные типы видимыми нарушает правило качества кода CA1034, "Вложенные типы не должны быть видимы".

- Вложенные типы **структуры** могут быть общедоступными, внутренними или частными.

Следующий пример делает класс `Nested` общедоступным:

C#

```
public class Container
{
    public class Nested
    {
        Nested() { }
```

```
    }
}
```

Вложенный или внутренний тип может получить доступ к содержащему или внешнему типу. Чтобы получить доступ к содержащему типу, передайте его как аргумент конструктору вложенного типа. Рассмотрим пример.

C#

```
public class Container
{
    public class Nested
    {
        private Container? parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

Вложенный тип имеет доступ ко всем элементам, доступным для его содержащего типа. Он может получить доступ к частным и защищенным членам содержащего типа, включая все унаследованные защищенные элементы.

В предыдущем объявлении полное имя класса `Nested` `Container.Nested`. Это имя, используемое для создания нового экземпляра вложенного класса, как показано ниже.

C#

```
Container.Nested nest = new Container.Nested();
```

См. также

- Система типов C#
- Модификаторы доступа
- Конструкторы
- Правило CA1034

Last updated on 17.12.2025

Частичные классы и члены (руководство по программированию на C#)

Можно разделить определение класса, структуры, интерфейса или члена по двум или нескольким исходным файлам. Каждый исходный файл содержит раздел определения типа или члена, а все части объединяются при компиляции приложения.

Частичные классы

Существует несколько ситуаций, когда желательно разделение определения класса.

- Объявление класса по отдельным файлам позволяет нескольким программистам одновременно работать над ним.
- Вы можете добавить код в класс, не воссоздавая исходный файл с автоматически сгенерированным кодом. Visual Studio использует этот подход при создании форм Windows Forms, кода оболочки веб-службы и т. д. Можно создать код, который использует эти классы, без необходимости изменения файла, созданного в Visual Studio.
- Генераторы источников могут создавать дополнительные функциональные возможности в классе.

Чтобы разделить определение класса, используйте модификатор ключевого слова partial. На практике каждый частичный класс обычно определяется в отдельном файле, что упрощает управление и расширение класса с течением времени.

В следующем `Employee` примере показано, как класс может быть разделен на два файла: Employee_Part1.cs и Employee_Part2.cs.

C#

```
// This is in Employee_Part1.cs
public partial class Employee
{
    public void DoWork()
    {
        Console.WriteLine("Employee is working.");
    }
}

// This is in Employee_Part2.cs
public partial class Employee
{
    public void GoToLunch()
    {
        Console.WriteLine("Employee is at lunch.");
```

```
    }
}

//Main program demonstrating the Employee class usage
public class Program
{
    public static void Main()
    {
        Employee emp = new Employee();
        emp.DoWork();
        emp.GoToLunch();
    }
}

// Expected Output:
// Employee is working.
// Employee is at lunch.
```

Ключевое слово `partial` указывает, что другие части класса, структуры или интерфейса могут быть определены в пространстве имен. Все части должны использовать ключевое слово `partial`. Для формирования окончательного типа все части должны быть доступны во время компиляции. Все части должны иметь одинаковую доступность, например `public`, `private` и т. д.

Если какая-либо из частей объявлена абстрактной, то весь тип будет считаться абстрактным. Если какая-либо из частей объявлена запечатанной, то весь тип будет считаться запечатанным. Если какая-либо из частей объявляет базовый тип, то весь тип унаследует данный класс.

Все части, указывающие базовый класс, должны быть согласованы друг с другом, а части, не использующие базовый класс, все равно наследуют базовый тип. Части могут указывать различные базовые интерфейсы, и окончательный тип будет реализовывать все интерфейсы, перечисленные во всех частичных объявлениях. Любые члены класса, структуры или интерфейса, объявленные в частичном определении, доступны для всех остальных частей. Окончательный тип представляет собой комбинацию всех частей, выполненную во время компиляции.

> ⓘ **Примечание.**
>
> Модификатор `partial` недоступен для объявлений делегата или перечисления.

В следующем примере показано, что вложенные типы могут быть частичными, даже если тип, вложенный внутри, не является частичным.

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

Во время компиляции атрибуты определений разделяемого типа объединяются. В качестве примера рассмотрим следующие объявления:

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Они эквивалентны следующим объявлениям:

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

Следующие элементы объединяются из всех определений частичных типов:

- КОММЕНТАРИИ XML. Однако если оба объявления частичного члена включают примечания, будут включены только комментарии от реализующего члена.
- интерфейсы
- атрибуты универсального типа-параметра
- атрибуты классов
- члены

В качестве примера рассмотрим следующие объявления:

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Они эквивалентны следующим объявлениям:

```C#
class Earth : Planet, IRotate, IRevolve { }
```

Ограничения

При работе с определениями частичного класса существует несколько правил:

- Все определения частичного типа, являющиеся частями одного типа, должны быть изменены с помощью `partial`. Например, следующие объявления класса приведут к появлению ошибки:

  ```C#
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- `partial` модификатор может отображаться непосредственно перед ключевым словом `class`, `struct` или `interface`.
- В определениях частичных типов могут присутствовать вложенные частичные типы, что показано в следующем примере:

  ```C#
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- Все определения разделяемого типа, являющиеся частями одного и того же типа, должны быть определены в одной сборке и в одном модуле (EXE-файл или DLL-файл). Частичные определения не могут охватывать несколько модулей.
- Имя класса и параметры универсального типа должны совпадать на всех частичных определениях типов. Универсальные типы могут быть частичными. Все объявления разделяемого типа должны использовать одинаковые имена параметров в одном и том же порядке.

- Следующие ключевые слова для определения частичного типа являются необязательными, но если они присутствуют в одном определении частичного типа, то же самое должно быть указано в другом частичном определении для того же типа:
  - общественный
  - частный
  - защищено
  - внутренний
  - абстрактный
  - запечатанный
  - базовый класс
  - модификатор new (вложенные части)
  - универсальные ограничения

Дополнительные сведения см. в разделе Ограничения параметров типа.

Примеры

В следующем примере поля и конструктор `Coords` класса объявляются в одном определении разделяемого класса (`Coords_Part1.cs`), а `PrintCoords` метод объявляется в другом определении разделяемого класса (`Coords_Part2.cs`). Это разделение показывает, как частичные классы можно разделить на несколько файлов для упрощения обслуживания.

C#

```
// This is in Coords_Part1.cs
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

// This is in Coords_Part2.cs
public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
```

```
 }

// Main program demonstrating the Coords class usage
 class TestCoords
 {
     static void Main()
     {
         Coords myCoords = new Coords(10, 15);
         myCoords.PrintCoords();

         // Keep the console window open in debug mode.
         Console.WriteLine("Press any key to exit.");
         Console.ReadKey();
     }
 }
 // Output: Coords: 10,15
```

В следующем примере показано, что можно также разработать разделяемые структуры и интерфейсы.

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Частичные члены

Частичный класс или структуру может содержать частичный элемент. Одна часть класса содержит сигнатуру члена. Реализацию можно определить в той же части или в другой части.

Реализация не требуется для частичного метода, если подпись подчиняется следующим правилам:

- Объявление не включает модификаторы доступа. Метод имеет private доступ по умолчанию.
- Тип возвращаемого значения — void.
- Ни один из параметров не имеет out модификатора.
- Объявление метода не может содержать ни одного из следующих модификаторов:
  - виртуальных
  - переопределение
  - запечатанный
  - новую
  - наружный

Метод и все вызовы метода удаляются во время компиляции при отсутствии реализации.

Любой член, который не соответствует всем этим ограничениям, включая конструкторы, свойства, индексаторы и события, должен предоставлять реализацию. Эта реализация может быть предоставлена генератором *источника*. Частичные свойства нельзя реализовать с помощью автоматически реализованных свойств. Компилятор не может различать автоматическое свойство и объявление частичного свойства.

Начиная с C# 13, реализация частичного свойства может использовать свойства с поддерживающими полями для определения объявления реализации. Свойство с поддержкой поля предоставляет краткий синтаксис, где ключевое слово `field` обращается к синтезированному компилятором служебному полю для свойства. Например, можно написать следующий код:

C#

```
// in file1.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get; set; }
}

// In file2.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get => field; set; }
}
```

Вы можете использовать `field` как в методе доступа `get`, так и в методе доступа `set`, или в обоих.

Частичные члены позволяют разработчику одной части класса объявить члена. Реализующий другую часть класса может определить этот элемент. Существует два сценария, в которых это разделение полезно: шаблоны, создающие стандартный код и генераторы источников.

- **Код шаблона**: шаблон резервирует имя и подпись метода, чтобы созданный код мог вызвать метод. Эти методы придерживаются ограничений, которые позволяют разработчику решить, следует ли реализовать этот метод. Если метод не реализован, компилятор удаляет сигнатуру метода и все вызовы метода. Вызовы метода, включая любые результаты, которые могли бы произойти от оценки аргументов в вызовах, не имеют эффекта во время выполнения. Поэтому любой код в частичном классе может свободно использовать частичный метод, даже если реализация не предоставлена. Если метод вызывается, но не реализуется, не возникает никаких ошибок во время компиляции или времени выполнения.
- **Генераторы источников: генераторы** источников предоставляют реализацию для членов. Разработчик может добавить объявление члена (часто с атрибутами, считываемыми генератором источника). Разработчик может написать код, вызывающий эти элементы. Генератор источника выполняется во время компиляции и обеспечивает реализацию. В этом сценарии ограничения для частично реализованных участников, которые могут не быть внедрены, часто не соблюдаются.

C#

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
  // method body
}
```

- Объявления частичных элементов должны начинаться с контекстного ключевого слова partial.
- Частичные подписи членов в обеих частях частичного типа должны соответствовать.
- Частичный член может иметь статические и небезопасные модификаторы.
- Частичный элемент может быть обобщённым. Ограничения должны совпадать с определением и реализацией объявления члена. Имена параметров и параметров типов не обязаны совпадать в объявлении реализации и в определяющем.

- Можно создать делегат для частичного метода, который определен и реализован, но не для частичного метода, который не имеет реализации.

Спецификация языка C#

Дополнительные сведения см. в разделе "Частичные типы " и частичные методы в спецификации языка C#. Спецификация языка является окончательным источником синтаксиса и использования языка C#. Новые функции для частичных элементов определяются в спецификациях функций для расширения частичных методов, частичных свойств и индексаторов, а также частичных событий и конструкторов.

См. также

- Классы
- Типы структур
- Интерфейсы
- Частичный (тип)

Last updated on 24.02.2026

Как возвращать подмножества свойств элементов в запросе (руководство по программированию на C#)

Используйте анонимный тип в выражении запроса, если оба из этих условий применяются:

- Вы хотите вернуть только некоторые свойства каждого исходного элемента.
- Результаты запроса не нужно хранить вне области метода, в котором выполняется запрос.

Если вы хотите возвращать только одно свойство или поле из каждого исходного элемента, можно просто использовать оператор dot в предложении `select` . Например, чтобы вернуть только `ID` каждого `student`, напишите `select` следующим образом:

C#

```
select student.ID;
```

Пример

В следующем примере показано, как использовать анонимный тип для возврата только подмножества свойств каждого исходного элемента, соответствующего указанному условию.

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
```

```
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Обратите внимание, что анонимный тип использует имена исходного элемента для его свойств, если имена не указаны. Чтобы предоставить новые имена свойствам в анонимном типе, напишите инструкцию `select` следующим образом:

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

Если вы попробуете это в предыдущем примере, оператор `Console.WriteLine` также должен измениться.

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

Компиляция кода

Чтобы запустить этот код, скопируйте и вставьте класс в консольное приложение C# с директивой `using` System.Linq.

См. также

- Анонимные типы
- LINQ в C#

Last updated on 13.04.2026

Реализация явного интерфейса (руководство по программированию на C#)

17.06.2025

Если класс реализует два интерфейса, которые содержат член с одной и той же сигнатурой, то реализация этого члена в классе приведет к использованию обоих интерфейсов в качестве их реализации. В следующем примере все вызовы `Paint` используют один и тот же метод. Этот первый пример определяет типы:

C#

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

Следующий пример вызывает методы:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Но для обоих интерфейсов может потребоваться не использовать одну и ту же реализацию. Чтобы вызвать другую реализацию в зависимости от используемого интерфейса, можно явно реализовать член интерфейса. Явная реализация интерфейса — это член класса, который вызывается только через указанный интерфейс. Назовите член класса, префиксируя его именем интерфейса и периодом. Рассмотрим пример.

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

Член `IControl.Paint` класса доступен только через `IControl` интерфейс и `ISurface.Paint` доступен только через `ISurface` интерфейс. Обе реализации метода независимы, и ни одна из них не доступна напрямую в классе. Рассмотрим пример.

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint();  // Calls IControl.Paint on SampleClass.
surface.Paint();  // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Явная реализация также используется для устранения случаев, когда два интерфейса объявляют разные члены одного и того же имени, например свойство и метод. Для реализации обоих интерфейсов класс должен использовать явную реализацию для свойства `P` или метода `P` или обоих, чтобы избежать ошибки компилятора. Рассмотрим пример.

C#

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

У явной реализации интерфейса нет модификатора доступа, так как он недоступен в качестве члена типа, в который он определен. Вместо этого он доступен только при вызове через экземпляр интерфейса. При указании модификатора доступа для явной реализации интерфейса возникает ошибка компилятора CS0106. Дополнительные сведения см. в разделе interface (Справочник по C#).

Вы можете определить реализацию для членов, объявленных в интерфейсе. Если класс наследует реализацию метода из интерфейса, этот метод доступен только через ссылку типа интерфейса. Унаследованный элемент не отображается как часть общедоступного интерфейса. В следующем примере определяется реализация по умолчанию для метода интерфейса:

C#

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

Следующий пример вызывает реализацию по умолчанию:

C#

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Любой класс, реализующий `IControl` интерфейс, может переопределить метод по умолчанию либо как открытый метод, либо как явную реализацию `Paint` интерфейса.

См. также

- Объектно-ориентированное программирование
- Интерфейсы
- Наследование

Как явно реализовать члены интерфейса (Руководство по программированию на C#)

Статья • 10.04.2025

В этом примере объявляется интерфейс и `IDimensions` класс, `Box` который явным образом реализует члены `GetLength` интерфейса и `GetWidth`. К членам обращаются через экземпляр интерфейса `dimensions`.

Пример

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float _lengthInches;
    float _widthInches;

    Box(float length, float width)
    {
        _lengthInches = length;
        _widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return _lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return _widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;
```

```
        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //Console.WriteLine($"Length: {box1.GetLength()}");
        //Console.WriteLine($"Width: {box1.GetWidth()}");

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        Console.WriteLine($"Length: {dimensions.GetLength()}");
        Console.WriteLine($"Width: {dimensions.GetWidth()}");
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Надежное программирование

- Обратите внимание, что следующие строки в методе `Main` закомментированы, так как они будут создавать ошибки компиляции. К члену интерфейса, который явно реализован, не удается получить доступ из экземпляра класса :

  C#

  ```
  //Console.WriteLine($"Length: {box1.GetLength()}");
  //Console.WriteLine($"Width: {box1.GetWidth()}");
  ```

- Обратите внимание, что в методе `Main` следующие строки успешно выводят размеры коробки, поскольку методы вызываются из экземпляра интерфейса:

  C#

  ```
  Console.WriteLine($"Length: {dimensions.GetLength()}");
  Console.WriteLine($"Width: {dimensions.GetWidth()}");
  ```

См. также

- Объектно-ориентированное программирование
- Интерфейсы
- Как явно реализовать члены двух интерфейсов

Как явно реализовать элементы двух интерфейсов (руководство по программированию на C#)

Статья • 10.04.2025

Явная реализация интерфейса также позволяет программисту реализовать два интерфейса с одинаковыми именами членов и предоставлять каждому члену интерфейса отдельную реализацию. В этом примере отображаются размеры поля как в метриках, так и в английских единицах. Класс Box реализует два интерфейса: IEnglishDimensions и IMetricDimensions, представляющих разные системы измерения. Оба интерфейса имеют одинаковые имена элементов, длину и ширину.

Пример

C#

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float _lengthInches;
    float _widthInches;

    public Box(float lengthInches, float widthInches)
    {
        _lengthInches = lengthInches;
        _widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => _lengthInches;

    float IEnglishDimensions.Width() => _widthInches;
```

```
    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => _lengthInches * 2.54f;

    float IMetricDimensions.Width() => _widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        Console.WriteLine($"Length(in): {eDimensions.Length()}");
        Console.WriteLine($"Width (in): {eDimensions.Width()}");

        // Print dimensions in metric units:
        Console.WriteLine($"Length(cm): {mDimensions.Length()}");
        Console.WriteLine($"Width (cm): {mDimensions.Width()}");
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Надежное программирование

Если вы хотите сделать измерения по умолчанию в английских единицах, реализуйте методы Length и Width обычно, а также явным образом реализуйте методы Length и Width из интерфейса IMetricDimensions:

C#

```
// Normal implementation:
public float Length() => _lengthInches;
public float Width() => _widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => _lengthInches * 2.54f;
float IMetricDimensions.Width() => _widthInches * 2.54f;
```

В этом случае можно получить доступ к единицам английского языка из экземпляра класса и получить доступ к единицам метрик из экземпляра интерфейса:

C#

```
public static void Test()
{
    Box box1 = new(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    Console.WriteLine($"Length(in): {box1.Length()}");
    Console.WriteLine($"Width (in): {box1.Width()}");
    Console.WriteLine($"Length(cm): {mDimensions.Length()}");
    Console.WriteLine($"Width (cm): {mDimensions.Width()}");
}
```

См. также

- Объектно-ориентированное программирование
- Интерфейсы
- Как правильно реализовывать члены интерфейса

Делегаты (руководство по программированию на C#)

Статья • 28.03.2025

делегат — это тип, который представляет ссылки на методы с определенным списком параметров и типом возврата. При создании экземпляра делегата можно связать экземпляр делегата с любым методом, который имеет совместимую сигнатуру и тип возвращаемого значения. Метод можно вызывать (или обращаться к нему) с помощью экземпляра делегата.

Делегаты используются для передачи методов в качестве аргументов другим методам. Обработчики событий — это, по сути, методы, вызываемые с помощью делегатов. При создании пользовательского метода класс, например элемент управления Windows, может вызывать метод при возникновении определенного события.

В следующем примере показано объявление делегата:

```
C#

public delegate int PerformCalculation(int x, int y);
```

Можно назначить любой метод из любого доступного класса или структуры, которые соответствуют типу делегата, этому делегату. Метод может быть статическим или методом экземпляра. Гибкость позволяет программным способом изменять вызовы методов или подключать новый код к существующим классам.

> ⓘ **Примечание**
>
> В контексте перегрузки метода подпись метода не включает возвращаемое значение. Однако в контексте делегатов подпись включает возвращаемое значение. Другими словами, метод должен иметь совместимый тип возвращаемого значения, как указано делегатом.

Возможность ссылаться на метод как параметр делает делегаты идеальными для определения методов обратного вызова. Вы можете написать метод, который сравнивает два объекта в приложении. Затем метод можно использовать в делегате для алгоритма сортировки. Так как код сравнения отличается от библиотеки, метод сортировки может быть более общим.

указатели функций поддерживают аналогичные сценарии, где требуется больше контроля над соглашением о вызовах. Код, связанный с делегатом, вызывается с помощью виртуального метода, добавленного в тип делегата. При работе с указателями функций можно указать различные соглашения.

Изучение характеристик делегата

Делегаты имеют следующие характеристики:

- Делегаты позволяют передавать методы в качестве параметров.
- Делегаты можно использоваться для определения методов обратного вызова.
- Делегаты можно объединить в цепочку, например вызывать несколько методов в одном событии.
- Методы не должны точно соответствовать типу делегата. Дополнительные сведения см. в разделе Использование вариативности в делегатах.
- Лямбда-выражения — это более краткий способ написания встроенных блоков кода. Лямбда-выражения (в определенных контекстах) компилируются в делегатные типы. Дополнительные сведения о лямбда-выражениях см. в разделе Лямбда-выражения.

Просмотр связанных статей

Дополнительные сведения о делегатах см. в следующих статьях:

- Использование делегатов
- Делегаты с именованными и анонимными методами
- Использование дисперсии в делегатах
- Как объединить делегаты (многоадресные делегаты)
- Как объявить, инициализировать и использовать делегат

Доступ к спецификации языка C#

Спецификация языка является окончательным источником для синтаксиса И использования C#. Для получения дополнительной информации см. раздел Делегаты в спецификации языка C# .

Связанные ссылки

- Delegate
- события

Использование делегатов (руководство по программированию на C#)

17.06.2025

Делегат — это тип, который безопасно инкапсулирует метод, аналогичный указателю функции в C и C++. В отличие от указателей функций C, делегаты являются объектно-ориентированными, типобезопасны и безопасны. В следующем примере объявляется делегат с именем `Callback` , который может инкапсулировать метод, который принимает строку в качестве аргумента и возвращает void:

C#

```
public delegate void Callback(string message);
```

Объект делегата обычно создается путем указания имени метода, который делегат оборачивает, или с использованием лямбда-выражения. Делегат можно вызвать, как только он будет создан. Вызов делегата вызывает метод, подключенный к экземпляру делегата. Параметры, передаваемые делегату вызывающим кодом, передаются методу. Делегат возвращает возвращаемое значение, если таковое имеется, из метода. Рассмотрим пример.

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Типы делегатов являются производными от Delegate класса в .NET. Типы делегатов запечатанные, они не могут быть производными от, и невозможно наследовать пользовательские классы от Delegate. Так как экземпляр делегата является объектом, его можно передать в качестве аргумента или присвоить свойству. Метод может принять делегат в качестве параметра и вызвать его в более позднее время. Это называется

асинхронным обратным вызовом и является общим методом уведомления вызывающего объекта при завершении длительного процесса. Если делегат используется таким образом, код, использующий делегат, не нуждается в знании о реализации используемого метода. Функциональность аналогична той, которую обеспечивают интерфейсы инкапсуляции.

Другое частое использование обратных вызовов — определение пользовательского метода сравнения и передача делегата в метод сортировки. Он позволяет вызывающему коду стать частью алгоритма сортировки. В следующем примере метода в качестве параметра используется `Del` тип:

C#

```
public static void MethodWithCallback(int param1, int param2, Callback callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Затем делегат, созданный в предыдущем примере, можно передать в этот метод:

C#

```
MethodWithCallback(1, 2, handler);
```

И получите следующие выходные данные в консоли:

Консоль

```
The number is: 3
```

`MethodWithCallback` не требуется вызывать консоль напрямую— ее не нужно разрабатывать с учетом консоли. Что делает `MethodWithCallback` — это подготавливает строку и передаёт её другому методу. Делегированный метод может использовать любое количество параметров.

При создании делегата для оболочки метода экземпляра, делегат ссылается как на сам экземпляр, так и на метод. Делегат не имеет знаний о типе экземпляра, кроме метода, который он упаковывает. Делегат может ссылаться на любой тип объекта, если в этом объекте есть метод, соответствующий сигнатуре делегата. Когда делегат создается для упаковки статического метода, он ссылается только на метод. Рассмотрим следующие объявления:

C#

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Наряду со статическим `DelegateMethod`, показанным ранее, теперь у нас есть три метода, которые можно обернуть в экземпляр `Del`.

Делегат может вызывать более одного метода при активации, что называется мультикастингом. Чтобы добавить дополнительный метод в список методов делегата — список вызовов, просто требуется добавить два делегата с помощью операторов добавления или добавления ("+" или "+="). Рассмотрим пример.

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

`allMethodsDelegate` содержит три метода в списке вызовов: `Method1`, `Method2`, и `DelegateMethod`. Исходные три делегата, `d1` `d2`, и `d3`, остаются неизменными. При вызове `allMethodsDelegate` все три метода вызываются в упорядоченном виде. Если делегат использует ссылочные параметры, ссылка передается последовательно каждому из трех методов, и любые изменения по одному методу видны следующему методу. Если любой метод выбрасывает исключение, которое не поймано в методе, это исключение передается вызывающей стороне делегата. В списке вызовов не вызываются последующие методы. Если делегат имеет возвращаемое значение и/или параметры выхода, возвращает возвращаемое значение и параметры последнего вызываемого метода. Чтобы удалить метод из списка вызовов, используйте операторы вычитания или присваивания вычитания (`-` или `-=`). Рассмотрим пример.

C#

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Callback oneMethodDelegate = (allMethodsDelegate - d2)!;
```

Так как типы делегатов являются производными от `System.Delegate`, методы и свойства, определенные тем классом, можно вызвать на делегате. Например, чтобы найти количество методов в списке вызовов делегата, можно написать следующее:

```
C#

int invocationCount = d1.GetInvocationList().GetLength(0);
```

Делегаты с несколькими методами в списке вызовов происходят от MulticastDelegate, который является суперклассом `System.Delegate`. Предыдущий код работает в любом случае, так как оба класса поддерживают `GetInvocationList`.

Делегаты многоадресной рассылки широко используются в обработке событий. Объекты источника событий отправляют уведомления о событиях получателям, зарегистрированным для получения этого события. Чтобы зарегистрировать событие, получатель создает метод, предназначенный для обработки события, а затем создает делегат для этого метода и передает этот делегат источнику данного события. Источник вызывает делегата при возникновении события. Затем делегат вызывает метод обработки событий у получателя, передавая данные события. Источник событий определяет тип делегата для данного события. Дополнительные сведения см. в разделе "События".

Сравнение делегатов двух разных типов, назначенных во время компиляции, приводит к ошибке компиляции. Если экземпляры делегата статически относятся к типу `System.Delegate`, то сравнение допускается, но возвращает значение false во время выполнения. Рассмотрим пример.

```
C#

delegate void Callback1();
delegate void Callback2();

static void method(Callback1 d, Callback2 e, System.Delegate f)
{
    // Compile-time error.
    Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

См. также

- делегаты

- Использование параметров с вариантностью в делегатах
- Изменения среди делегатов
- Использование дисперсии для универсальных делегатов Func и Action
- События

Делегаты с именованными методами и делегаты с анонимными методами (Руководство по программированию в C#)

Делегат может быть связан с именованным методом. При создании экземпляра делегата с использованием именованного метода этот метод передается в качестве параметра, например:

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
WorkCallback d = obj.DoWork;
```

В предыдущем примере используется именованный метод. Делегаты, созданные с именованным методом, могут инкапсулировать как статический метод, так и метод экземпляра. Назначенные методы являются единственным способом инстанцирования делегата в ранних версиях C#. C# позволяет создать экземпляр делегата и сразу же указать блок кода, который делегат обрабатывает при вызове. Блок может содержать лямбда-выражение или анонимный метод, как показано в следующем примере:

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Instantiate the delegate using an anonymous method.
WorkCallback d = (int k) => { /* ... */ };
```

Метод, который вы передаете в качестве параметра делегата, должен иметь такую же сигнатуру, что и объявление делегата. Экземпляр объекта делегата может содержать статический или экземплярный метод.

Примечание.

Несмотря на то, что делегат может использовать параметр **out**, не рекомендуется делать это для делегатов многоадресных событий, поскольку в этом случае невозможно определить, какой делегат будет вызван.

Группы методов с одной перегрузкой имеют *естественный тип*. Компилятор может определить тип возвращаемого значения и типы параметров для типа делегата:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Примеры

В следующем примере показан простой пример объявления и использования делегата. Обратите внимание, что делегат `MultiplyCallback` и связанный с ним метод `MultiplyNumbers` имеют одинаковые сигнатуры.

C#

```
// Declare a delegate
delegate void MultiplyCallback(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        MultiplyCallback d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
```

```
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

В следующем примере делегат, сопоставленный одновременно со статическим методом и методом экземпляра, возвращает информацию из каждого из них.

C#

```
// Declare a delegate
delegate void Callback();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Callback d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

См. также

- Делегаты
- Как объединять делегаты (многоадресные делегаты)
- События

Last updated on 24.02.2026

Как объединять делегаты (многоадресные делегаты) (Руководство по программированию на C#)

В этом примере показано, как создавать делегаты многоадресной рассылки. Полезное свойство объектов делегата заключается в том, что можно назначить несколько методов одному экземпляру делегата `+` с помощью оператора. Делегат многоадресной рассылки содержит список назначенных делегатов. При вызове делегата многоадресной рассылки он вызывает делегатов в списке в порядке. Вы можете объединять только делегаты одного типа. Вы можете использовать оператор `-`, чтобы удалить делегата компонента из мультикастового делегата.

C#

```
using System;


namespace DelegateExamples;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomCallback(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomCallback.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomCallback hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Initialize the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;
```

```
        // Initialize the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = (multiDel - hiDel)!;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

Примечание.

Один и тот же делегат можно добавить в многоадресный делегат несколько раз. При вызове делегата многоадресной рассылки он вызывает всех делегатов в списке, включая повторяющиеся. При удалении делегата из мультикастового делегата удаляется самая правая совпадающая запись, поэтому при наличии нескольких копий удаляется только один экземпляр.

См. также

- MulticastDelegate
- События

Last updated on 11.03.2026

Руководство по программированию на C#. Объявление, создание экземпляра и использование делегата

Статья • 07.04.2023

Делегаты можно объявить с помощью любого из следующих методов:

- Объявите тип делегата и объявите метод с соответствующей сигнатурой:

C#

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

C#

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

- Назначьте группу методов типу делегата:

C#

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

- Объявите анонимный метод:

C#

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

- Используйте лямбда-выражение:

C#

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine($"Notification received for:
{name}"); };
```

Дополнительные сведения см. в разделе Лямбда-выражения.

Следующий пример демонстрирует объявление, создание экземпляра и использование делегата. Класс `BookDB` инкапсулирует базу данных книжного магазина, в которой хранится информация о книгах. Он предоставляет метод `ProcessPaperbackBooks`, который находит в базе данных все книги в мягкой обложке и вызывает делегат для каждой из них. Используется тип `delegate` с именем `ProcessBookCallback`. Класс `Test` использует этот класс для печати заголовков и средней цены книг в мягкой обложке.

Использование делегата помогает правильно разделить функции между базой данных книжного магазина и кодом клиента. Код клиента не имеет сведений о том, как хранятся книги и как код книжного магазина находит книги в мягкой обложке. Код книжного магазина не имеет сведений о том, какая обработка выполняется для найденных книг в мягкой обложке.

Пример

C#

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool
paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }
```

```
    // Declare a delegate type for processing a book:
    public delegate void ProcessBookCallback(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool
paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
```

```
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan
and Dennis M. Ritchie", 19.95m, true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott
Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Отказоустойчивость

- Объявление делегата.

  Следующая инструкция объявляет новый тип делегата.

  ```
  C#

  public delegate void ProcessBookCallback(Book book);
  ```

  Каждый тип делегата описывает количество аргументов и их типы, а также тип возвращаемого значения для всех методов, которые он может инкапсулировать. Каждый раз, когда требуется новый набор типов аргументов или новый тип возвращаемого значения, нужно объявить новый тип делегата.

- Создания экземпляра делегата.

  После объявления типа делегата нужно создать объект этого делегата и связать его с определенным методом. Продолжая предыдущий пример, вы можете передать метод `PrintTitle` в метод `ProcessPaperbackBooks`, как показано в следующем примере:

  ```
  C#

  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  Будет создан новый объект делегата, связанный со статическим методом `Test.PrintTitle`. Аналогичным образом, в следующем примере передается нестатический метод `AddBookToTotal` для объекта `totaller`:

  ```
  C#

  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  В обоих случаях новый объект делегата передается в метод `ProcessPaperbackBooks`.

  После создания делегата невозможно изменить метод, с которым он связан. Объекты делегатов являются неизменяемыми.

- Использование делегатов

Обычно после создания объекта делегата он передается в другой код, который будет использовать этот делегат. Для вызова объекта делегата используется имя этого объекта, за которым следуют параметризованные аргументы, которые нужно передать делегату. Ниже показан пример использования делегата.

```
C#

processBook(b);
```

Делегат можно вызвать синхронно, как показано в этом примере, или асинхронно при помощи методов `BeginInvoke` и `EndInvoke`.

См. также

- Руководство по программированию на C#
- События
- Делегаты

Строки и строковые литералы

Строка — это объект типа String значение которого — текст. Внутреннее хранение текста осуществляется в виде последовательной коллекции объектов Char, которые только для чтения. Свойство Length строки представляет количество объектов, содержащихся в `Char`, а не число символов Юникода. Чтобы получить доступ к отдельным точкам кода Юникода в строке, используйте объект StringInfo.

string vs. System.String

В C#ключевое слово `string` — это псевдоним для String; поэтому `String` и `string` эквивалентны. Используйте указанный псевдоним `string`, так как он работает даже без `using System;`. Класс `String` предоставляет множество методов безопасного создания, управления и сравнения строк. Кроме того, язык C# перегружает некоторые операторы для упрощения общих строковых операций. Для получения дополнительной информации о ключевом слове см. строкув . Дополнительные сведения о типе и его методах см. в String.

Объявление и инициализация строк

Можно объявлять и инициализировать строки различными способами, как показано в следующем примере:

C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string? message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
```

```
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Вы не используете новый оператор для создания строкового объекта, за исключением при инициализации строки с массивом символов.

Инициализирует строку с константным значением Empty, чтобы создать новый объект String, строка которого равна нулевой длине. Строковое литеральное представление строки нулевой длины равно `""`. Инициализировав строки со значением Empty вместо null, можно уменьшить вероятность возникновения NullReferenceException. Используйте статический метод IsNullOrEmpty(String), чтобы проверить значение строки перед попыткой доступа к ней.

Неизменяемость строк

Строковые объекты *неизменяемы*: их нельзя изменить после их создания. Все методы String и операторы C#, которые, как представляется, изменяют строку, фактически возвращают результаты в новом строковом объекте. В следующем примере, когда содержимое `s1` и `s2` объединяются для формирования одной строки, две исходные строки не изменены. Оператор `+=` создает новую строку, содержащую объединенное содержимое. Этот новый объект назначается переменной `s1`, и исходный объект, назначенный переменной `s1`, освобождается для сборки мусора, потому что ни одна другая переменная не содержит ссылку на него.

C#

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Поскольку "изменение строки" на самом деле означает создание новой строки, следует проявлять осторожность при создании ссылок на строки. Если вы создаете ссылку на строку, а затем изменяете исходную строку, ссылка продолжает указывать на исходный объект. Исходный объект не отражает новый объект, созданный при изменении строки. Следующий код иллюстрирует это поведение:

C#

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

Дополнительные сведения о создании новых строк, основанных на изменениях, таких как операции поиска и замены исходной строки, см. в разделе Изменение содержимого строки.

Строковые литералы в кавычках

Цитируемые строковые литералы начинаются и заканчиваются одинарной двойной кавычкой (`"`) на одной строке. Строковые литералы, заключенные в кавычки, лучше всего подходят для строк, которые помещаются на одной строке и не включают управляющие последовательности. Строковый литерал в кавычках должен содержать escape-символы, как показано в следующем примере:

C#

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
    Row 1
    Row 2
    Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Дословные строковые литералы

Дословные строковые литералы удобнее использовать для многострочных строк, строк, содержащих символы обратной косой черты или встроенные двойные кавычки. Строки verbatim сохраняют символы новой строки в качестве части текста строки. Используйте двойные кавычки для внедрения внутренней кавычки в дословную строку. В следующем примере показаны некоторые распространенные способы использования дословных строк.

C#

```
string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge

string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

Необработанные строковые литералы

Необработанные строковые литералы можно использовать для упрощения создания строк, которые являются многостроочными, или использовать любые символы, требующие escape-последовательностей. *Необработанные строковые литералы* устраняют необходимость использования экранирующих последовательностей. Можно написать строку, включая форматирование пробелов, как она будет отображаться в выходных данных. *необработанный строковый литерал*:

- Начинается и заканчивается последовательностью по крайней мере трех двойных символов кавычки (`"""`). Вы можете использовать более трех последовательных символов, чтобы начинать и заканчивать последовательность, поддерживая строковые литералы, содержащие три или более повторяющихся символа кавычек.
- Для однострочных необработанных строковых литералов символы открывающей и закрывающей кавычки должны находиться в одной строке.
- Многострочные сырой строковые литералы требуют как открывающие, так и закрывающие кавычки на отдельной строке.

- В многострочных необработанных строковых литералах все пробелы слева от закрывающих кавычек удаляются во всех строках.
- В многострочных необработанных строковых литералах пробелы после открывающей кавычки в той же строке игнорируются.
- В многострочных необработанных строковых литералах строки, состоящие только из пробелов и следующие за открывающей кавычкой, включаются в строковый литерал.

В следующих примерах демонстрируются следующие правила:

C#

```
string singleLine = """Friends say "hello" as they pass by.""";
string multiLine = """
    "Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
       <element attr = "content">
           <body style="normal">
               Here is the main text
           </body>
           <footer>
               Excerpts from "An amazing story"
           </footer>
       </element >
       """;
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """"
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
    """";
```

В следующих примерах показаны ошибки компилятора, сообщаемые в соответствии с этими правилами:

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
    """;

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
```

```
// of the raw string literal
var noOutdenting = """
    A line of text.
Trying to outdent the second line.
    """;
```

Первые два примера недопустимы, так как многострочные необработанные строковые литералы требуют, чтобы открывающая и закрывающая последовательности кавычек были на собственной строке. Третий пример неправильный, так как текст выходит за пределы последовательности закрывающих кавычек.

Следует учитывать необработанные строковые литералы, если создавать текст, включающий символы, требующие использования escape-последовательностей , при работе с обычными или буквальными строковыми литералами. Необработанные строковые литералы проще для вас и других пользователей для чтения, так как они более похожи на выходной текст. Например, рассмотрим следующий код, содержащий строку форматированного JSON:

C#

```
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"
  ]
}
""";
```

Кончик

Visual Studio и C# Dev Kit предоставляют некоторые проверки и подсветку синтаксиса, когда необработанные строковые литералы содержат данные JSON или регулярные выражения.

Средства синтаксического анализа текста. Если средства уверены, что текст представляет JSON или регулярное выражение, редактор предоставляет цвет синтаксиса.

Вы можете улучшить этот интерфейс, добавив комментарий над объявлением, указывающим формат:

- `// lang=json` указывает, что необработанный строковый литерал представляет данные JSON.
- `// lang=regex` указывает, что необработанный строковый литерал представляет регулярное выражение.

Если необработанный строковый литерал используется в качестве аргумента, где параметр использует формат System.Diagnostics.CodeAnalysis.StringSyntaxAttribute, такие инструменты проверяют необработанный строковый литерал на соответствие некоторым типам форматов. Поддерживаются как JSON, так и regex.

В некоторых форматах комментарий или атрибут позволяют предлагать исправления для строковых литералов на основе формата.

Escape-последовательности строк

Развернуть таблицу

escape-последовательность	Имя персонажа	Кодировка Юникода
`\'`	Одна кавычка	0x0027
`\"`	Двойная кавычка	0x0022
`\\`	Обратная косая черта	0x005C
`\0`	Недействительный	0x0000
`\a`	Тревога	0x0007
`\b`	Бэкспейс	0x0008
`\e`	Побег	0x001B

escape-последовательность	Имя персонажа	Кодировка Юникода
`\f`	Подача формы	0x000C
`\n`	Новая строка	0x000A
`\r`	Возврат каретки	0x000D
`\t`	Горизонтальная вкладка	0x0009
`\v`	Вертикальная вкладка	0x000B
`\u`	Эскейп-последовательность Юникода (UTF-16)	`\uHHHH` (диапазон: 0000 – FFFF; пример: `\u00E7` = "ç")
`\U`	Эскейп-последовательность Юникода (UTF-32)	`\U00HHHHHH` (диапазон: 000000 – 10FFFF; пример: `\U0001F47D` = "👽")
`\x`	Escape-последовательность Юникода похожа на "\u", за исключением переменной длины	`\xH[H][H][H]` (диапазон: 0 – FFFF; пример: `\x00E7` или `\x0E7` или `\xE7` = "ç")

> **⚠ Предупреждение**
>
> При использовании `\x` escape-последовательности и указании менее 4 шестнадцатеричных цифр, если символы, которые немедленно следуют за escape-последовательностью, являются допустимыми шестнадцатеричными цифрами (например, 0-9, A-F и a-f), они интерпретируются как часть escape-последовательности. Например, `\xA1` выводит "¡", что соответствует кодовой точке U+00A1. Однако, если следующий символ — это "A" или "a", то последовательность escape-последовательностей будет интерпретирована как `\xA1A` и создастся "ਚ", что является кодовой точкой U+0A1A. В таких случаях указание всех 4 шестнадцатеричных цифр (например, `\x00A1`) предотвращает любое возможное неправильное толкование.

> **ⓘ Заметка**
>
> Во время компиляции буквальные и сырьевые строки преобразуются в обычные строки с теми же управляющими последовательностями. Поэтому при просмотре дословной или необработанной строки в окне просмотра отладчика отображаются escape-символы, добавленные компилятором, а не дословная или необработанная

версия из вашего исходного кода. Например, буквальная строка `@"C:\files.txt"` отображается в окне отслеживания как `"C:\\files.txt"`.

Форматирование строк

Строка формата — это строка, содержимое которой определяется динамически во время выполнения. Строки форматирования создаются за счёт внедрения *интерполированных выражений* или заполнителей внутрь фигурных скобок, находящихся внутри строки. Все внутри фигурных скобок (`{...}`) преобразуется в значение и отображается в формате строки во время выполнения. Существует два метода создания строк форматирования: интерполяция строк и составное форматирование.

Интерполяция строк

Вы объявляете *интерполированные строки* со специальным символом `$`. Интерполированная строка включает интерполированные выражения в фигурных скобках. Если вы не знакомы с интерполяцией строк, ознакомьтесь с руководством по интерполяции строк — C# для нескольких примеров.

Используйте интерполяцию строк, чтобы улучшить удобочитаемость и удобство обслуживания кода. Интерполяция строк обеспечивает те же результаты, что и метод `String.Format`, но проще использовать и улучшает ясность в коде.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born
in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of
{jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years
old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Интерполяция строк позволяет инициализировать константную строку, если все выражения, используемые для заполнителей, также являются константными строками.

Необработанные литералы строк можно объединить с интерполяциями строк. Вы начинаете и заканчиваете строку формата тремя или более последовательными

двойными кавычками. Если строка вывода должна содержать символ `{` или `}`, можно использовать дополнительные `$` символы, чтобы указать, сколько `{` и `}` символов начинаются и заканчиваются интерполяцией. Любая последовательность меньшего числа `{` или `}` символов включается в выходные данные. В следующем примере показано, как использовать эту функцию для отображения расстояния точки от источника и размещения точки внутри фигурных скобок:

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y * Y)}}
from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

Интерполяция строк дословно

C# также позволяет дословную интерполяцию строк, например, на нескольких строках, с использованием синтаксиса `$@` или `@$`.

Чтобы буквально интерпретировать управляющие последовательности, используйте строковый литерал . Интерполированная дословная строка начинается с символа `$`, за которым следует `@`. Маркеры `$` и `@` можно использовать в любом порядке: `$@"..."` и `@$"..."` являются допустимыми интерполированными строками.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($@"{jh.firstName} {jh.lastName}
    was an African American poet born in {jh.born}.");
Console.WriteLine(@$"He was first published in {jh.published}
at the age of {jh.published - jh.born}.");

// Output:
// Jupiter Hammon
//     was an African American poet born in 1711.
// He was first published in 1761
// at the age of 50.
```

Составное форматирование

String.Format использует плейсхолдеры в фигурных скобках для создания форматированной строки. Этот пример приводит к аналогичным выходным данным метода интерполяции строк, используемому в предыдущем примере.

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName,
pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published,
pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born)
/ 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

Дополнительные сведения см. в статье "Составное форматирование" в .NET.

Подстроки

Подстрока — это любая последовательность символов, содержащихся в строке. Используйте метод Substring для создания новой строки из части исходной строки. Можно выполнить поиск одного или нескольких вхождений подстроки с помощью метода IndexOf. Используйте метод Replace, чтобы заменить все вхождения указанной подстроки новой строкой. Как и метод Substring, Replace фактически возвращает новую строку и не изменяет исходную строку. Дополнительные сведения см. в разделе Поиск строк и Изменение содержимого строки.

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Доступ к отдельным символам

Вы можете использовать нотацию массива со значением индекса для получения доступа только для чтения к отдельным символам, как показано в следующем примере:

C#

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Для более обширной StringBuilder обработки строк можно использовать объект для изменения отдельных символов "на месте". Затем создайте новую строку для хранения результатов с помощью StringBuilder методов. В следующем примере предположим, что необходимо изменить исходную строку определенным образом, а затем сохранить результаты для дальнейшего использования:

C#

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Дополнительные сведения об индексах см. в статье "Изучение индексов и диапазонов ".

Строки NULL и пустые строки

Пустая строка — это экземпляр объекта System.String, содержащего нулевые символы. Пустые строки часто используются в различных сценариях программирования для представления пустого текстового поля. На пустых строках можно вызывать методы, так как они являются допустимыми объектами типа System.String. Пустые строки инициализированы следующим образом:

C#

```
string s = String.Empty;
```

В отличие от этого, строка NULL не ссылается на экземпляр объекта System.String и любая попытка вызвать метод в строке NULL вызывает NullReferenceException. Однако можно использовать null-строки в сцеплении и операциях сравнения с другими строками. В следующих примерах показаны некоторые случаи, в которых ссылка на строку NULL не вызывает исключение:

C#

```
string str = "hello";
string? nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

Использование StringBuilder для быстрого создания строк

Строковые операции в .NET высоко оптимизированы и в большинстве случаев не оказывают значительного влияния на производительность. Однако в некоторых

сценариях, таких как жесткие циклы, выполняющие много сотен или тысяч раз, строковые операции могут повлиять на производительность. Класс StringBuilder создает строковый буфер, который обеспечивает более высокую производительность, если программа выполняет множество строковых манипуляций. Строка StringBuilder также позволяет переназначать отдельные символы, чего встроенный тип данных строки не поддерживает. Например, этот код изменяет содержимое строки без создания новой строки:

C#

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
//Outputs Cat: the ideal pet
```

В этом примере объект StringBuilder используется для создания строки из набора числовых типов:

C#

```
var sb = new StringBuilder();

// Create a string composed of numbers 0 - 9
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}
Console.WriteLine(sb);  // displays 0123456789

// Copy one character of the string (not possible with a System.String)
sb[0] = sb[9];

Console.WriteLine(sb);  // displays 9123456789
```

Строки, методы расширения и LINQ

Поскольку тип String реализует IEnumerable<T>, можно использовать методы расширения, определенные в классе Enumerable для строк. Чтобы избежать визуального загромождений, эти методы исключаются из IntelliSense для типа String, но они доступны тем не менее. Выражения запросов LINQ также можно использовать в строках. Дополнительные сведения см. в разделе LINQ иStrings.

Связанные статьи

- Как изменить содержимое строки: иллюстрирует методы преобразования строк и изменения содержимого строк.
- Сравнение строк. Демонстрирует, как выполнять порядковые и культурно специфичные сравнения строк.
- Как объединить несколько строк: демонстрирует различные способы объединения нескольких строк в одну.
- Как анализировать строки с помощью String.Split: содержит примеры кода, иллюстрирующие использование метода String.Split для анализа строк.
- Как искать строки: объясняется, как использовать поиск определенного текста или шаблонов в строках.
- Как определить, представляет ли строка числовое значение: показывает, как безопасно проанализировать строку, чтобы узнать, имеет ли он допустимое числовое значение.
- Интерполяция строк: Описывает функцию интерполяции строк, которая предоставляет удобный синтаксис для форматирования строк.
- Использование класса StringBuilder в .NET
- синтаксический анализ строк. Описывает преобразование строковых представлений базовых типов .NET в экземпляры соответствующих типов.
- синтаксический анализ строк даты и времени в.NET: показывает, как преобразовать строку, например "01/24/2008" в объект System.DateTime.
- сравнение строк: содержит сведения о том, как сравнивать строки и приведены примеры в C# и Visual Basic.
- Использование класса StringBuilder: описывает создание и изменение динамических строковых объектов с помощью класса StringBuilder.
- LINQ и Strings: содержит сведения о выполнении различных строковых операций с помощью запросов LINQ.

Last updated on 18.11.2025

Как определить, представляет ли строка числовое значение (руководство по программированию на C#)

Статья • 01.04.2025

Чтобы определить, является ли строка допустимым представлением указанного числового типа, используйте статический `TryParse` метод, реализуемый всеми примитивными числовыми типами, а также типами, такими как DateTime и IPAddress. В следующем примере показано, как определить, является ли "108" допустимым int.

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Если строка содержит ненумерные символы или числовое значение слишком велико или слишком мало для указанного типа, `TryParse` возвращает значение false и задает для параметра out значение нулю. В противном случае функция возвращает true и устанавливает параметр out в числовое значение строки.

> ⓘ **Примечание**
>
> Строка может содержать только числовые символы, но всё же быть недействительной для типа, в котором используется метод `TryParse`. Например, "256" не является допустимым значением для `byte`, но допустимо для `int`. "98.6" не является допустимым значением для `int`, но это допустимый `decimal`.

Пример

В следующих примерах показано, как использовать `TryParse` со строковыми представлениями `long`, `byte` и значениями `decimal`.

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
Console.WriteLine($"number1 now = {number1}");
else
Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
Console.WriteLine($"number2 now = {number2}");
else
Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
Console.WriteLine($"number3 now = {number3}");
else
Console.WriteLine("number3 is not a valid decimal");
```

Надежное программирование

Примитивные числовые типы также реализуют статический метод `Parse`, который создает исключение, если строка не является допустимым числом. `TryParse`, как правило, эффективнее, так как он возвращает false, если число некорректно.

Безопасность .NET

Всегда используйте методы `TryParse` или `Parse` для проверки ввода пользователей из элементов управления, таких как текстовые поля и поля со списком.

См. также

- Преобразование массива байтов в int
- Преобразование строки в число
- Преобразование между шестнадцатеричными строками и числовыми типами
- синтаксический анализ числовых строк
- Типы форматирования

Индексаторы

17.06.2025

Индексаторы определяются, когда экземпляры класса или структуры можно индексировать как массив или другую коллекцию. Индексированное значение можно задать или извлечь без явного указания типа или элемента экземпляра. Индексаторы похожи на свойства , за исключением того, что их методы доступа принимают параметры.

В следующем примере определяется универсальный класс с методами доступа get и set для назначения и извлечения значений.

C#

```
namespace Indexers;

public class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}
```

В предыдущем примере показан индексатор чтения и записи. Он содержит как `get` методы доступа, так и `set` методы доступа. Вы можете определить индексаторы только для чтения в качестве элемента выражения, как показано в следующих примерах:

C#

```
namespace Indexers;

public class ReadOnlySampleCollection<T>(params IEnumerable<T> items)
{
   // Declare an array to store the data elements.
   private T[] arr = [.. items];

   public T this[int i] => arr[i];

}
```

Ключевое `get` слово не используется; `=>` вводит текст выражения.

Индексаторы позволяют *индексировать* свойства: свойства, на которые ссылается один или несколько аргументов. Эти аргументы предоставляют индекс в некоторых коллекциях значений.

- Индексаторы позволяют индексировать объекты, аналогичные массивам.
- Метод `get` доступа возвращает значение. Акцессор `set` назначает значение.
- Ключевое this слово определяет индексатор.
- Ключевое value слово — это аргумент `set` для аксессора.
- Индексаторы не требуют целочисленного значения индекса; Это зависит от того, как определить конкретный механизм поиска.
- Индексаторы могут быть перегружены.
- Индексаторы могут иметь один или несколько формальных параметров, например при доступе к двухмерным массивам.
- Индексаторы можно объявить partial в partial типах.

Вы можете применить практически все, что вы узнали о работе со свойствами к индексаторам. Единственное исключение из этого правила — это *автоматически реализуемые свойства*. Компилятор не всегда может создать правильное хранилище для индексатора. Можно определить несколько индексаторов в типе, если списки аргументов для каждого индексатора уникальны.

Использование индексаторов

Вы определяете *индексаторы* в вашем типе, когда API моделирует некоторую коллекцию. Вам не обязательно сопоставлять индексатор непосредственно с типами коллекций, которые являются частью фреймворка .NET Core. Индексаторы позволяют предоставить API, соответствующий абстракции типа, без предоставления внутренних сведений о том, как хранятся или вычисляются значения для этой абстракции.

Массивы и векторы

Тип может моделировать массив или вектор. Преимущество создания собственного индексатора заключается в том, что вы можете определить хранилище для этой коллекции в соответствии с вашими потребностями. Представьте сценарий, в котором исторические данные моделей типов слишком большие для загрузки в память одновременно. Необходимо загружать и разгружать разделы коллекции в зависимости от их использования. В примере ниже приведены модели этого поведения. Он сообщает о том, сколько точек данных существует. Он создает страницы для хранения разделов

данных по запросу. Он удаляет страницы из памяти, чтобы освободить место для страниц, необходимых для более поздних запросов.

C#

```
namespace Indexers;

public record Measurements(double HiTemp, double LoTemp, double AirPressure);

public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new ();
        private readonly int _startingIndex;
        private readonly int _length;

        public Page(int startingIndex, int length)
        {
            _startingIndex = startingIndex;
            _length = length;

            // This stays as random stuff:
            var generator = new Random();
            for (int i = 0; i < length; i++)
            {
                var m = new Measurements(HiTemp: generator.Next(50, 95),
                    LoTemp: generator.Next(12, 49),
                    AirPressure: 28.0 + generator.NextDouble() * 4
                );
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= _startingIndex) &&
            (index < _startingIndex + _length));

        public Measurements this[int index]
        {
            get
            {
                LastAccess = DateTime.Now;
                return pageData[index - _startingIndex];
            }
            set
            {
                pageData[index - _startingIndex] = value;
                Dirty = true;
                LastAccess = DateTime.Now;
            }
        }

        public bool Dirty { get; private set; } = false;
        public DateTime LastAccess { get; set; } = DateTime.Now;
```

```
    }

    private readonly int _totalSize;
    private readonly List<Page> pagesInMemory = new ();

    public DataSamples(int totalSize)
    {
        this._totalSize = totalSize;
    }

    public Measurements this[int index]
    {
        get
        {
            if (index < 0) throw new IndexOutOfRangeException("Cannot index less
than 0");
            if (index >= _totalSize) throw new IndexOutOfRangeException("Cannot
index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0) throw new IndexOutOfRangeException("Cannot index less
than 0");
            if (index >= _totalSize) throw new IndexOutOfRangeException("Cannot
index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
```

```
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
}
```

Вы можете следовать этой идиоме дизайна, чтобы моделировать любую коллекцию, в которой есть хорошие причины не загружать весь набор данных в коллекцию в памяти. Обратите внимание, что `Page` класс является частным вложенным классом, который не является частью общедоступного интерфейса. Эти сведения скрыты от пользователей этого класса.

Словари

Другой распространенный сценарий заключается в том, когда необходимо моделировать словарь или карту. Этот сценарий заключается в том, что тип сохраняет значения на основе ключа, возможно, текстовых ключей. В этом примере создается словарь, который сопоставляет аргументы командной строки с лямбда-выражениями , которые управляют этими параметрами. В следующем примере показаны два класса: `ArgsActions` класс, который сопоставляет параметр командной строки с делегатом System.Action, и класс `ArgsProcessor`, который использует `ArgsActions` для выполнения каждого Action при обнаружении этого параметра.

C#

```
namespace Indexers;
public class ArgsProcessor
{
    private readonly ArgsActions _actions;

    public ArgsProcessor(ArgsActions actions)
    {
        _actions = actions;
    }

    public void Process(string[] args)
    {
        foreach (var arg in args)
        {
            _actions[arg]?.Invoke();
        }
    }
```

```
}
public class ArgsActions
{
    readonly private Dictionary<string, Action> _argsActions = new();

    public Action this[string s]
    {
        get
        {
            Action? action;
            Action defaultAction = () => { };
            return _argsActions.TryGetValue(s, out action) ? action :
defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        _argsActions[s] = a;
    }
}
```

В этом примере коллекция `ArgsAction` в точности сопоставляется с базовой коллекцией. Определяет `get` , настроен ли заданный параметр. Если да, он возвращает связанный с этим параметром Action . Если нет, он возвращает Action, который ничего не делает. Общедоступный аксессор не включает `set` аксессор. Скорее, проект использует общедоступный метод для настройки параметров.

Многомерные карты

Можно создать индексаторы, использующие несколько аргументов. Кроме того, эти аргументы не ограничены тем же типом.

В следующем примере показан класс, который создает значения для набора Mandelbrot. Дополнительные сведения о математике, стоящей за набором, см. в этой статье. Индексатор использует два двойника для определения точки в плоскости X, Y. Аксессор `get` вычисляет количество итераций, пока точка не будет определена как не находящаяся в наборе. Когда достигается максимальное число итераций, точка находится в наборе, а значение maxIterations класса возвращается. (Созданные компьютером изображения, популярные для набора Mandelbrot, определяют цвета для количества итераций, необходимых для определения того, что точка находится за пределами набора.)

C#

```
namespace Indexers;
public class Mandelbrot(int maxIterations)
```

```
{

    public int this[double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x * x + y * y < 4) &&
                (iterations < maxIterations))
            {
                (x, y) = (x * x - y * y + x0, 2 * x * y + y0);
                iterations++;
            }
            return iterations;
        }
    }
}
```

Набор Mandelbrot определяет значения в каждой координате (x,y) для реальных значений чисел. Это определяет словарь, который может содержать бесконечное число значений. Поэтому за установкой нет места для хранения. Вместо этого этот класс вычисляет значение для каждой точки при вызове метода доступа `get`. Базовое хранилище не используется.

Суммирование

Вы создаете индексаторы в любой момент, когда в классе есть элемент, похожий на свойства, где это свойство представляет не одно значение, а скорее набор значений. Один или несколько аргументов определяют каждый отдельный элемент. Эти аргументы могут однозначно определить, на какой элемент в наборе следует ссылаться. Индексаторы расширяют концепцию свойств, где член обрабатывается как элемент данных извне класса, но как метод внутри. Индексаторы позволяют аргументам находить один элемент в свойстве, представляющего набор элементов.

Вы можете получить доступ к образцу папки для индексаторов. Инструкции по скачиванию смотрите в разделах Образцы и руководства.

Спецификация языка C#

Дополнительные сведения см. в разделе "Индексаторы " в спецификации языка C#. Спецификация языка является авторитетным источником синтаксиса и использования языка C#.

Использование индексаторов. Руководство по программированию на C#

Индексаторы — это синтаксическое удобство, позволяющее создавать класс, структуру или интерфейс , доступ к которым клиентские приложения могут получить в виде массива. Компилятор создает `Item` свойство (или альтернативно именованное свойство, если IndexerNameAttribute оно присутствует), а также соответствующие методы доступа. Индексаторы чаще всего реализуются в типах, предназначенных преимущественно для инкапсуляции внутренней коллекции или массива. Допустим, у вас есть класс `TempRecord`, который представляет температуру по шкале Фаренгейта, записанную в 10 различных моментах в течение 24 часов. Этот класс содержит массив `temps` типа `float[]` для хранения значений температуры. Реализация индексатора в этом классе позволит клиентам получать доступ к значениям температуры в экземпляре `TempRecord`, используя `float temp = tempRecord[4]` вместо `float temp = tempRecord.temps[4]`. Использование нотации индексатора не только упрощает синтаксис для клиентских приложений, но и делает класс и его предназначение более понятными для других разработчиков.

Чтобы объявить индексатор для класса или структуры, используйте ключевое слово this, как в следующем примере:

C#

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

> ⓘ **Внимание**
>
> При объявлении индексатора для объекта автоматически создается свойство с именем `Item`. Свойство `Item` не доступно непосредственно из **выражения для доступа к члену** экземпляра. Кроме того, если вы добавите к объекту с индексатором собственное свойство `Item`, возникнет **ошибка компилятора CS0102**. Чтобы избежать этой ошибки, используйте переименование IndexerNameAttribute индексатора, как описано далее в этой статье.

Замечания

Тип индексатора и типы его параметров должны иметь по крайней мере такой же уровень доступности, как и сам индексатор. Дополнительные сведения об уровнях доступа см. в разделе Модификаторы доступа.

Дополнительные сведения об использовании индексаторов с интерфейсом см. в разделе Индексаторы интерфейса.

Сигнатура индексатора определяет число и типы его формальных параметров. В ней не указываются тип индексатора или имена его формальных параметров. Если для одного класса объявляется несколько индексаторов, они должны иметь разные сигнатуры.

Индексатор не классифицируется как переменная; таким образом, значение индексатора не может передаваться по ссылке (как или refout параметру), если его значение не является ссылкой (т. е. возвращается по ссылке).)

Чтобы присвоить индексатору имя, которое можно использовать в других языках, используйте System.Runtime.CompilerServices.IndexerNameAttribute, как показано в этом примере:

C#

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Этот индексатор имеет имя `TheItem`, так как оно переопределяется атрибутом имени индексатора. По умолчанию используется имя индексатора `Item`.

Пример 1

В следующем примере показано, как объявить частное поле массива `temps` и индексатор. Индексатор обеспечивает прямой доступ к экземпляру `tempRecord[i]`. Вместо использования индексатора можно объявить массив как элемент public и осуществлять доступ к его элементам напрямую (`tempRecord.temps[i]`).

C#

```
public class TempRecord
{
    // Array of temperature values
    float[] temps =
    [
```

```
            56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
            61.3F, 65.9F, 62.1F, 59.2F, 57.5F
        ];

        // To enable client code to validate input
        // when accessing your indexer.
        public int Length => temps.Length;

        // Indexer declaration.
        // If index is out of range, the temps array will throw the exception.
        public float this[int index]
        {
            get => temps[index];
            set => temps[index] = value;
        }
    }
```

Обратите внимание, что при определении прав доступа индексатора, например в инструкции `Console.Write`, вызывается метод доступа get. Таким образом, если аксессор `get` отсутствует, возникает ошибка времени компиляции.

C#

```
var tempRecord = new TempRecord();

// Use the indexer's set accessor
tempRecord[3] = 58.3F;
tempRecord[5] = 60.1F;

// Use the indexer's get accessor
for (int i = 0; i < 10; i++)
{
    Console.WriteLine($"Element #{i} = {tempRecord[i]}");
}
```

Индексирование с использованием других значений

В C# тип параметра индексатора не ограничивается целочисленными значениями. Например, можно использовать строку с индексатором. Такой индексатор можно реализовать путем поиска строки в коллекции с возвратом соответствующего значения. Поскольку методы доступа можно перегружать, строковые и целочисленные версии могут сосуществовать.

Пример 2

Этот пример объявляет класс, который хранит названия дней недели. Метод доступа `get` принимает название дня в виде строкового значения и возвращает соответствующее целое число. Например, для Sunday возвращается значение 0, для Monday — 1 и т. д.

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = ["Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat"];

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form \"Sun\",
\"Mon\", etc");
    }
}
```

Пример использования 2

C#

```
var week = new DayCollection();
Console.WriteLine(week["Fri"]);

try
{
    Console.WriteLine(week["Made-up day"]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

Пример 3

В этом примере объявляется класс, в котором хранятся названия дней недели с использованием перечисления System.DayOfWeek. Метод доступа `get` принимает `DayOfWeek`, значение дня, и возвращает соответствующее целое число. Например, для `DayOfWeek.Sunday` возвращается 0, для `DayOfWeek.Monday` — 1 и т. д.

C#

```
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    [
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    ];

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined
System.DayOfWeek value.");
    }
}
```

Пример использования 3

C#

```
var week = new DayOfWeekCollection();
Console.WriteLine(week[DayOfWeek.Friday]);

try
{
    Console.WriteLine(week[(DayOfWeek)43]);
}
catch (ArgumentOutOfRangeException e)
{
```

```
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

надёжное программирование

Повысить безопасность и надежность индексаторов можно двумя способами:

- Реализуйте стратегию обработки ошибок, предусматривающую действия в ситуациях, когда из клиентского кода передается недопустимое значение индекса. В первом примере выше в этой статье класс TempRecord предоставляет свойство Length, позволяющее клиентскому коду проверить входные данные перед передачей в индексатор. Кроме того, код обработки ошибок можно поместить в сам индексатор. Не забудьте задокументировать для пользователей любые исключения, которые вы выбрасываете в индексаторе.

- Настройте доступность методов доступа get и set настолько ограниченной, насколько это разумно. Особенно важно именно для аксессора `set`. Дополнительные сведения см. в разделе Ограничение доступности методов доступа.

См. также

- Индексаторы
- Свойства

Last updated on 24.02.2026

Индексаторы в интерфейсах (Руководство по программированию в C#)

Индексаторы можно объявлять для интерфейса. Методы доступа индексаторов интерфейса отличаются от методов доступа индексаторов класса следующим образом:

- Аксессоры интерфейса не используют модификаторы.
- Аксессор интерфейса обычно не имеет тела.

Модификатор доступа служит для указания, является ли индексатор доступным для чтения и записи, только для чтения или только для записи. Вы можете предоставить реализацию для индексатора, определенного в интерфейсе, но это редко. Индексаторы обычно определяют API для доступа к полям данных, а поля данных не могут быть определены в интерфейсе.

Вот пример аксессора индексатора для интерфейса:

C#

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

Сигнатура индексатора должна отличаться от сигнатур любых других индексаторов, объявленных в том же интерфейсе.

Пример

Следующий пример демонстрирует реализацию индексаторов интерфейса.

C#

```
// Indexer on an interface:
public interface IIndexInterface
{
```

```
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

C#

```
IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

В приведенном выше примере можно использовать явную реализацию члена интерфейса, указав его полное имя. Например.

C#

```
string IIndexInterface.this[int index]
{
}
```

Тем не менее полное имя требуется только в целях устранения неоднозначности, если класс реализует более одного интерфейса с одинаковой сигнатурой индексатора. Например, если класс `Employee` реализует два интерфейса (`ICitizen` и `IEmployee`), оба из которых имеют одинаковую сигнатуру индексатора, требуется явная реализация члена интерфейса. То есть, следующее объявление индексатора:

C#

```
string IEmployee.this[int index]
{
}
```

Реализует индексатор в интерфейсе `IEmployee`, при этом следующее объявление:

C#

```
string ICitizen.this[int index]
{
}
```

Реализует индексатор в интерфейсе `ICitizen` .

См. также

- Индексаторы
- Свойства
- Интерфейсы

Last updated on 24.02.2026

Сравнение свойств и индексаторов (Руководство по программированию в C#)

Индексаторы подобны свойствам. К методам доступа индексаторов применяются те же правила, которые определены для методов доступа к свойствам, за исключением различий, показанных в следующей таблице.

Развернуть таблицу

Свойство	Индексатор
Позволяет вызывать методы так, как если бы они были публичными членами данных.	Обеспечивает доступ к элементам внутренней коллекции объекта с использованием нотации массива для самого объекта.
Доступ по простому имени.	Доступ посредством индекса.
Может быть статическим членом или членом экземпляра.	Должен быть членом экземпляра.
Метод доступа get свойства не имеет параметров.	Метод доступа `get` индексатора имеет тот же список формальных параметров, что и сам индексатор.
Метод доступа set свойства содержит неявный параметр `value`.	Метод доступа `set` индексатора имеет тот же список формальных параметров, что и сам индексатор, а также параметр value.
Поддерживает сокращенный синтаксис с автоматически реализованными свойствами.	Поддерживает элементы в виде выражения для индексаторов только для получения.

См. также

- Индексаторы
- Свойства

Last updated on 24.02.2026

События (руководство по программированию на C#)

Статья • 22.03.2025

События позволяют классу или объекту уведомлять другие классы или объекты, когда возникает что-то интересное. Класс, который отправляет (или *вызывает*) событие, называется *издателем*, а классы, которые получают (или *обрабатывают*) событие, называются *подписчиками*.

В типичном приложении Windows Forms или Web на языке C# вы подписываетесь на события, инициируемые элементами управления, такими как кнопки и поля списка. Интегрированная среда разработки Visual C# (IDE) можно использовать для просмотра событий, публикуемых элементом управления, и выбора тех, которые требуется обрабатывать. Интегрированная среда разработки предоставляет простой способ автоматического добавления пустого метода обработчика событий и кода для подписки на событие. Дополнительные сведения см. в разделе Как подписаться и отменить подписку на события.

Обзор событий

События имеют следующие свойства:

- Издатель определяет, когда возникает событие; подписчики определяют, какие действия принимаются в ответ на событие.
- Событие может иметь несколько подписчиков. Подписчик может обрабатывать несколько событий из нескольких издателей.
- События, не имеющие подписчиков, никогда не вызываются.
- События обычно используются для сигнала о действиях пользователя, таких как нажатия кнопки или выбор меню в графических пользовательских интерфейсах.
- Если событие имеет несколько подписчиков, обработчики событий вызываются синхронно при возникновении события. Для асинхронного вызова событий см. раздел Асинхронный вызов синхронных методов.
- В библиотеке классов .NET события основаны на делегате EventHandler и базовом классе EventArgs.

Связанные разделы

Дополнительные сведения см. в следующем разделе:

- Подписка на события и отмена подписки на события
- Как публиковать события, соответствующие рекомендациям .NET
- Как вызывать события базового класса в производных классах
- Реализация событий интерфейса
- Реализация пользовательских методов доступа к событиям

Спецификация языка C#

Дополнительные сведения см. в разделе событий в спецификации языка C#. Спецификация языка является окончательным источником для синтаксиса И использования C#.

См. также

- EventHandler
- делегаты
- Создание обработчиков событий в Windows Forms

Руководство по программированию на C#. Подписка и отмена подписки на события

Статья • 10.03.2023

Необходимость подписки на событие, опубликованное другим классом, может возникнуть, когда требуется написать пользовательский код, вызываемый при инициировании такого события. Например, можно подписаться на событие кнопки `click`, чтобы приложение выполняло некоторое действие при нажатии пользователем кнопки.

Подписка на события в интегрированной среде разработки Visual Studio

1. Если окно **Свойства** закрыто, в представлении **Конструктор** щелкните правой кнопкой мыши форму или элемент управления, для которого требуется создать обработчик событий, и выберите пункт **Свойства**.

2. Вверху окна **Свойства** щелкните значок **События**.

3. Дважды щелкните событие, которое требуется создать, например событие `Load`.

   Visual C# создаст пустой метод обработчика событий и добавит его в код. Код можно также добавить вручную в представлении **Код**. Например, приведенные ниже строки кода объявляют метод обработчика событий, который будет выполнен при вызове классом `Form` события `Load`.

   C#

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   Строка кода, требуемая для подписки на событие, также создается автоматически в методе `InitializeComponent` в файле Form1.Designer.cs проекта. Она имеет следующий вид:

   C#

```
this.Load += new System.EventHandler(this.Form1_Load);
```

Подписка на события программными средствами

1. Определите метод обработчика событий, сигнатура которого соответствует сигнатуре делегата для события. Например, если событие основано на типе делегата EventHandler, то следующий код представляет заглушку метода:

   C#

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Чтобы присоединить обработчик событий к событию, используйте оператор присваивания сложения (`+=`). В приведенном ниже примере предположим, что объект с именем `publisher` имеет событие с именем `RaiseCustomEvent`. Обратите внимание на то, что классу подписчика требуется ссылка на класс издателя, чтобы подписаться на его события.

   C#

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   Чтобы указать обработчик событий, можно также воспользоваться лямбда-выражением:

   C#

   ```
   public Form1()
   {
       InitializeComponent();
       this.Click += (s,e) =>
           {
               MessageBox.Show(((MouseEventArgs)e).Location.ToString());
           };
   }
   ```

Подписка на события с помощью анонимной функции

Если вам не нужно будет позже отменять подписку на событие, можно использовать оператор присваивания сложения (`+=`) для привязки анонимной

функции как обработчика событий. В следующем примере предположим, что объект с именем `publisher` имеет событие с именем `RaiseCustomEvent` и что класс `CustomEventArgs` также был определен и содержит некие относящиеся к событию сведения. Обратите внимание на то, что классу подписчика требуется ссылка на `publisher`, чтобы подписаться на его события.

```
C#
```

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Отменить подписку на событие не так просто, если для подписки использовалась анонимная функция. Чтобы отменить подписку в этом случае, в коде, где была выполнена подписка на событие, сохраните анонимную функцию в переменной делегата, а затем добавьте делегат к событию. Мы рекомендуем не использовать анонимные функции для подписки на события, если в будущем вам нужно будет отменить подписку. Дополнительные сведения об анонимных функциях см. в статье Лямбда-выражения.

Отмена подписки

Чтобы предотвратить вызов обработчика событий при инициировании события, подписку на событие необходимо отменить. Во избежание утечки ресурсов отменять подписку на события следует до удаления объекта подписчика. До тех пор пока подписка на событие не отменена, делегат многоадресной рассылки, лежащий в основе события в публикующем объекте, будет ссылаться на делегат, инкапсулирующий обработчик событий подписчика. Если ссылка присутствует в публикующем объекте, объект подписчика не будет удален при сборке мусора.

Отмена подписки на событие

- Чтобы отменить подписку на событие, воспользуйтесь оператором присваивания вычитания (`-=`).

  ```
  C#
  ```

  ```
  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

Если подписка на событие отменена для всех подписчиков, экземпляр события в классе издателя получает значение `null`.

См. также

- События
- event
- Практическое руководство. Публикация событий, соответствующих рекомендациям .NET
- Операторы - и -=
- Операторы + и +=

Руководство по программированию на C#. Порождение событий базового класса в производных классах

Статья • 10.05.2023

В следующем простом примере показан стандартный способ объявления событий в базовом классе, позволяющий вызывать их из производных классов. Этот шаблон активно применяется в классах Windows Forms в библиотеке классов .NET.

При создании класса, который можно использовать в качестве базового для других классов, следует учитывать тот факт, что события представляют собой особый тип делегатов, который может вызываться только в том классе, который их объявил. Производные классы не могут создавать события, объявленные в базовом классе, напрямую. Несмотря на то, что в некоторых обстоятельствах вам может потребоваться событие, вызываемое только базовым классом, в большинстве случаев рекомендуется разрешать производному классу вызывать события базового класса. Для этого следует создать в базовом классе защищенный метод, предоставляющий оболочку для события. Вызывая или перезаписывая вызывающий метод, производные классы могут вызывать событие косвенно.

> ⓘ **Примечание**
>
> Не объявляйте виртуальные события в базовом классе и не переопределяйте их в производном классе. Компилятор c# не обрабатывает их корректно, поэтому сложно сказать, будет ли подписчик производного события на самом деле подписываться на событие базового класса.

Пример

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
```

```
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event
type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
```

```
        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
```

```
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

См. также раздел

- Руководство по программированию на C#
- События
- Делегаты
- Модификаторы доступа
- Создание обработчиков событий в Windows Forms

Руководство по программированию на C#. Реализация событий интерфейса

Статья • 12.03.2024

Интерфейс может объявлять события. Следующий пример демонстрирует реализацию событий интерфейса в классе. По сути правила остаются такими же, как при реализации любого метода или свойства интерфейса.

Реализация событий интерфейса в классе

Объявите событие в классе и вызовите его, когда потребуется.

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Пример

Следующий пример демонстрирует обработку менее распространенной ситуации, в которой класс наследует характеристики от двух или более интерфейсов, каждый из которых имеет событие с одним и тем же именем. Вам придется предоставить явную реализацию интерфейса по крайней мере для одного из этих событий. Когда вы создаете для события явную реализацию интерфейса, нужно добавить еще и методы доступа `add` и `remove`. Обычно они предоставляются компилятором, но в этой ситуации компилятор не сможет предоставить их.

Предоставляя собственные методы доступа, вы можете указать одно или разные события вашего класса, которые соответствуют этим двум событиям. Например, если события должны инициироваться в разное время в соответствии со спецификациями интерфейса, можно связать каждое из них с отдельной реализацией в классе. В следующем примере подписчики определяют, какое из событий `OnDraw` они получат, указывая ссылку на фигуру `IShape` или `IDrawingObject`.

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
```

```
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
```

```
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

См. также

- События
- Делегаты

- Явная реализация интерфейса
- Практическое руководство. Создание событий базового класса в производных классах

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Руководство по программированию на C#. Реализация пользовательских методов доступа к событиям

Статья • 12.03.2024

Событие представляет собой особый вид многоадресного делегата, который можно вызвать только из класса, где он объявлен. В клиентском коде создается подписка на событие. Для этого предоставляется ссылка на метод, который должен вызываться при возникновении этого события. Эти методы добавляются в список вызова делегата с помощью методов доступа к событиям, которые схожи с методами доступа к свойствам и отличаются только именами (`add` и `remove`). В большинстве случаев реализовывать настраиваемые методы доступа к событиям не требуется. Если в коде отсутствуют настраиваемые методы доступа к событиям, компилятор добавляет их автоматически. Тем не менее в некоторых случаях требуется реализовать настраиваемое поведение. Один из таких сценариев показан в руководстве по реализации событий интерфейса.

Пример

В следующем примере показано, как реализовать настраиваемые методы доступа add и remove для события. При необходимости вы можете заменять любой код в методах доступа, однако мы рекомендуем заблокировать событие, прежде чем добавлять или удалять новый метод обработчика событий.

C#

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
```

```
    }
}
```

См. также

- События
- event

Совместная работа с нами на GitHub

Источник этого содержимого можно найти на GitHub, где также можно создавать и просматривать проблемы и запросы на вытягивание. Дополнительные сведения см. в нашем руководстве для участников.

.NET

Отзыв о .NET

.NET — это проект с открытым исходным кодом. Выберите ссылку, чтобы оставить отзыв:

Открыть проблему с документацией

Отзыв о продукте

Параметры универсального типа (руководство по программированию на C#)

17.06.2025

В определении универсального типа или метода параметр типа является заменителем для конкретного типа, который клиент указывает при создании экземпляра универсального типа. Универсальный класс, например `GenericList<T>` приведенный в разделе "Введение в универсальные шаблоны", нельзя использовать as-is, так как он не является типом. Это более похоже на схему для типа. Чтобы использовать `GenericList<T>`, клиентский код должен объявить и создать экземпляр созданного типа, указав аргумент типа внутри угловых квадратных скобок. Аргумент типа для этого конкретного класса может быть любым типом, распознаваемым компилятором. Любое количество созданных экземпляров типов можно создать, каждый из которых использует другой аргумент типа, как показано ниже.

C#

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

В каждом из этих случаев `GenericList<T>` каждое вхождение `T` в классе заменяется при выполнении аргументом указанного типа. С помощью этой подстановки мы создали три отдельных типа безопасных и эффективных объектов с помощью определения одного класса. Дополнительные сведения о том, как это замещение выполняется в среде CLR, см. в разделе Обобщения в среде выполнения.

Соглашения об именовании для параметров универсального типа можно узнать в статье о соглашениях об именовании.

См. также

- System.Collections.Generic
- Обобщения
- Различия между шаблонами C++ и универсальными шаблонами C#

Ограничения параметров типа (руководство по программированию на C#)

Ограничения сообщают компилятору о характеристиках, которые должен иметь аргумент типа. Без ограничений аргумент типа может быть любым типом. Компилятор может только предполагать членов System.Object, который является главным базовым классом для всех типов .NET. Дополнительные сведения см. в статье Зачем использовать ограничения. Если клиентский код использует тип, который не удовлетворяет ограничению, компилятор выдает ошибку. Ограничения задаются с помощью контекстного ключевого слова `where`. В следующей таблице описываются различные типы ограничений:

Развернуть таблицу

Ограничение	Описание
`where T : struct`	Аргумент типа должен быть типом значения, который не может принимать значение null и включает `record struct` типы. См. дополнительные сведения о типах значений, допускающих значение NULL. Так как все типы значений имеют конструктор без параметров, объявленный или неявный, `struct` ограничение подразумевает `new()` ограничение и не может сочетаться с ограничением `new()` . Ограничение `struct` нельзя использовать вместе с ограничением `unmanaged`.
`where T : class`	Аргумент типа должен быть ссылочным типом. Это ограничение также применяется к любому типу класса, интерфейса, делегата или массива. В контексте, допускающем значение NULL, `T` должен быть ссылочным типом, не допускающим значения NULL.
`where T : class?`	Тип аргумента должен быть ссылочным и может либо поддерживать значения NULL, либо не поддерживать их. Это ограничение применяется также к любому классу, интерфейсу, делегату или типу массива, включая записи.
`where T : notnull`	Аргумент типа должен быть типом, не допускающим значения NULL. Аргумент может быть ненулевым ссылочным типом или ненулевым типом значения.
`where T : unmanaged`	Аргумент типа должен быть неуправляемым типом, не допускающим значения NULL. Ограничение `unmanaged` подразумевает ограничение `struct` и не может использоваться совместно с ограничением `struct` или `new()`.
`where T : new()`	Аргумент типа должен иметь общий конструктор без параметров. При использовании совместно с другими ограничениями, ограничение `new()` должно указываться последним. Ограничение `new()` не может использоваться с ограничениями `struct` и `unmanaged`.

Ограничение	Описание
`where T :` *<Имя базового класса>*	Аргумент типа должен быть указанным базовым классом или производным от него. В контексте, допускающем значение NULL, `T` должен быть ссылочным типом, который не допускает значение NULL, производным от указанного базового класса.
`where T :` *<имя> базового класса?*	Аргумент типа должен быть указанным базовым классом или наследоваться от него. В контексте, допускающем значение NULL, `T` может быть либо типом, допускаемым null, либо не допускаемым значением NULL, производным от указанного базового класса.
`where T :` *<имя интерфейса>*	Аргумент типа должен являться заданным интерфейсом или реализовывать его. Можно указать несколько ограничений интерфейса. Ограничивающий интерфейс также может быть общим. В контексте, допускающем значение NULL, `T` должен быть ненулевым типом, реализующим указанный интерфейс.
`where T :` *<имя> интерфейса?*	Аргумент типа должен являться заданным интерфейсом или реализовывать его. Можно указать несколько ограничений интерфейса. Ограничительный интерфейс также может быть обобщенным. В nullable-контексте `T` может быть nullable-ссылочным типом, non-nullable-ссылочным типом или типом значения. `T` не может быть типом значений, допускающим значение NULL.
`where T : U`	Аргумент типа, указанный для `T`, должен быть аргументом, указанным для `U`, или производным от него. В контексте, допускающем значение NULL, если `U` это ненулевой ссылочный тип, `T` должен быть ненулевой ссылочный тип. Если `U` это ссылочный тип, допускающий значение NULL, `T` может иметь значение NULL или не допускающее значение NULL.
`where T : default`	Это ограничение устраняет неоднозначность, если необходимо указать неограниченный параметр типа, переопределяя метод или указывая явную реализацию интерфейса. Ограничение `default` подразумевает базовый метод без ограничения `class` или `struct`. Дополнительные сведения см. в предложении спецификации ограничения default.
`where T : allows ref struct`	Это анти-ограничение объявляет, что аргумент типа `T` может быть типом `ref struct`. Универсальный тип или метод должен соответствовать правилам безопасности ссылок для любого экземпляра `T` , так как это может быть `ref struct`.

Некоторые ограничения являются взаимоисключающими, и некоторые ограничения должны быть в указанном порядке:

- Вы можете применить не более одного из ограничений `struct`, `class`, `class?`, `notnull` или `unmanaged`. Если вы предоставляете любой из этих ограничений, это должно быть первое ограничение, указанное для этого параметра типа.
- Ограничение базового класса (`where T : Base` или `where T : Base?`) не может сочетаться с какими-либо из ограничений `struct`, `class`, `class?`, `notnull` или

`unmanaged`.

- Можно применить не более одного ограничения базового класса в любой форме. Если вы хотите поддерживать базовый тип, допускающий значение NULL, используйте `Base?`.
- Вы не можете использовать как ненулевую, так и nullable-форму интерфейса в качестве ограничения.
- Ограничение `new()` нельзя использовать с ограничением `struct` или `unmanaged`. Если указать `new()` ограничение, это должно быть последнее ограничение для этого параметра типа. Анти-ограничения могут, если применимо, следовать ограничению `new()`.
- Ограничение `default` может применяться только к переопределениям или явным реализациям интерфейса. Его нельзя объединить с `struct` ограничениями или `class` ограничениями.
- Анти-ограничение `allows ref struct` не может быть объединено с `class` или `class?` ограничением.
- Анти-ограничение `allows ref struct` должно соответствовать всем ограничениям для этого параметра типа.

Зачем использовать ограничения

Ограничения определяют возможности и ожидания параметра типа. Объявление этих ограничений означает, что можно использовать операции и вызовы методов ограничивающего типа. Ограничения применяются к параметру типа, когда универсальный класс или метод использует любую операцию с универсальными элементами за пределами простого назначения, которая включает вызов любых методов, не поддерживаемых System.Object. Например, ограничение базового класса сообщает компилятору, что только объекты этого типа или производные от этого типа могут заменить этот аргумент типа. Имея такую гарантию, компилятор может вызывать методы указанного типа в универсальном классе. В следующем примере кода показаны функциональные возможности, которые можно добавить в класс `GenericList<T>` (см. раздел Введение в универсальные шаблоны), применив ограничение базового класса.

C#

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
```

```
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;

    public void AddHead(T t)
    {
        Node n = new(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current is not null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;

        while (current is not null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                return current.Data;
            }
            else
            {
                current = current.Next;
            }
        }
        return null;
    }
}
```

Это ограничение позволяет универсальному классу использовать свойство `Employee.Name`. Ограничение указывает, что все элементы типа `T` гарантированно являются либо объектом `Employee`, либо объектом, который наследует от `Employee`.

К одному параметру типа можно применять несколько ограничений, которые сами по себе могут быть универсальными типами, как показано ниже:

C#

```
class EmployeeList<T> where T : notnull, Employee, IComparable<T>, new()
{
    public void AddDefault()
    {
        T t = new();
    }
}
```

При применении `where T : class` ограничения избегайте `==` и `!=` операторов для параметра типа, так как эти операторы проверяют только идентичность ссылок, а не равенство значений. Такое поведение будет наблюдаться даже в том случае, если эти операторы будут перегружены в типе, используемом в качестве аргумента. Эта особенность показана в следующем коде, который будет возвращать значение false даже в том случае, если класс String перегружает оператор `==`.

C#

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new("target");
    string s2 = sb.ToString();
    OpEqualsTest(s1, s2);
}
```

Компилятору известно только то, что `T` является ссылочным типом во время компиляции, и он должен использовать операторы по умолчанию, которые действительны для всех ссылочных типов. Если необходимо проверить равенство значений, примените `where T : IEquatable<T>` или `where T : IComparable<T>` ограничение и реализуйте интерфейс в любом классе, используемом для создания универсального класса.

Ограничение нескольких параметров

Ограничения можно применить к нескольким параметрам. Кроме того, к одному параметру можно применять несколько ограничений, как показано в следующем

примере:

C#

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Несвязанные параметры типа

Не имеющие ограничений параметры типа (например, T в общем классе `SampleClass<T>{}`) называются несвязанными. В отношении несвязанных параметров типа применяются следующие правила:

- Операторы `!=` и `==` нельзя использовать, так как нет гарантии, что аргумент конкретного типа поддерживает их.
- Их можно преобразовывать из и в `System.Object`, а также явно преобразовывать в любой тип интерфейса.
- Можно сравнить их со значением null. Если несвязанный параметр сравнивается `null`, сравнение всегда возвращает значение false, если аргумент типа является типом значения.

Параметры типа в качестве ограничений

Использование параметров универсального типа в качестве ограничений применимо, когда функция-член со своим параметром типа должна ограничивать этот параметр параметром содержащего типа, как показано в следующем примере:

C#

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

В предыдущем примере `T` является ограничением типа в контексте метода `Add` и несвязанным параметром типа в контексте класса `List`.

Параметры типа также можно использовать в определениях универсальных классов. Параметр типа необходимо объявлять в угловых скобках вместе с любыми другими

параметрами типа:

```C#
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

Применение параметров типа в качестве ограничений для универсальных классов ограничено, поскольку в отношении таких параметров типа компилятор может предполагать только то, что они являются производными от `System.Object`. Параметры типа в качестве ограничений следует использовать в универсальных классах в тех случаях, когда необходимо обеспечить отношение наследования между двумя параметрами типа.

Ограничение `notnull`

Ограничение можно использовать `notnull` для указания того, что аргумент типа должен быть типом значения, не допускающим значение NULL, или ссылочным типом, не допускающим значение NULL. В отличие от большинства других ограничений, если аргумент типа нарушает ограничение `notnull`, компилятор генерирует предупреждение вместо ошибки.

Ограничение `notnull` действует только при использовании в контексте, допускающем значения NULL. При добавлении ограничения `notnull` в очевидный контекст, допускающий значения NULL, компилятор не создает никаких предупреждений или ошибок в случае нарушений ограничения.

Ограничение `class`

Ограничение `class` в контексте, допускающем значение NULL, указывает, что аргумент типа должен быть ссылочным типом, не допускающим значение NULL. Если в контексте, допускающем значения NULL, аргумент типа является ссылочным типом, допускающим значения NULL, компилятор выдаст предупреждение.

Ограничение `default`

Добавление ссылочных типов, допускающих значения NULL, усложняет использование `T?` для универсального типа или метода. `T?` можно использовать либо с ограничением `struct`, либо с `class`, но одно из них должно присутствовать. Если используется ограничение `class`, `T?` ссылается на ссылочный тип, допускающий значения NULL, для `T`.

`T?` можно использовать, если ни в коем случае не применяется ограничение. В этом случае `T?` интерпретируется как `T?` для типов значений и ссылочных типов. Но если `T` — экземпляр Nullable<T>, `T?` соответствует `T`. Другими словами, оно не становится `T??`.

Так как `T?` теперь можно использовать без ограничения `class` или `struct`, в переопределениях или явных реализациях интерфейса могут возникать неоднозначности. В обоих этих случаях переопределение не включает ограничения, а унаследует их от базового класса. Если базовый класс не применяет ограничение `class` или `struct`, производные классы должны каким-либо образом указывать переопределение, применяемое к базовому методу без ограничения. Производный `default` метод применяет ограничение. Ограничение `default` *не* уточняет ни ограничение `class`, ни `struct`.

Неуправляемое ограничение

Ограничение можно использовать `unmanaged` для указания того, что параметр типа должен быть неуправляемым типом, не допускающим значение NULL. Ограничение `unmanaged` позволяет создавать многократно используемые подпрограммы для работы с типами, которые могут обрабатываться как блоки памяти, как показано в следующем примере:

C#

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new byte[size];
    byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

В примере выше метод необходимо компилировать в контексте `unsafe`, так как он использует оператор `sizeof` для типа, не известного как встроенный тип. Без ограничения `unmanaged` оператор `sizeof` недоступен.

Ограничение `unmanaged` подразумевает ограничение `struct` и не может использоваться совместно с ним. Поскольку ограничение `struct` подразумевает ограничение `new()`, ограничение `unmanaged` также не может использоваться с ограничением `new()`.

Ограничения делегата

Можно использовать System.Delegate или System.MulticastDelegate в качестве ограничения базового класса. Среда CLR всегда разрешала это ограничение, но язык C# запрещал его. Ограничение `System.Delegate` позволяет написать код, который работает с делегатами типобезопасным образом. Следующий код определяет метод расширения, который объединяет два делегата при условии, что они одного типа:

C#

```
extension<TDelegate>(TDelegate source) where TDelegate : System.Delegate
{
    public TDelegate? TypeSafeCombine(TDelegate target)
        => Delegate.Combine(source, target) as TDelegate;
}
```

Предыдущий метод можно использовать для объединения делегатов, которые являются одинаковыми типами:

C#

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined!();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

Если вы раскомментируете последнюю строку, она не компилируется. `first` и `test` являются типами делегатов, но это разные типы делегатов.

Ограничения перечисления

Можно также указать System.Enum тип в качестве ограничения базового класса. В среде CLR это ограничение всегда было разрешено, но в языке C# оно было запрещено. Универсальные шаблоны с `System.Enum` предоставляют типобезопасное программирование для кэширования результатов использования статических методов в `System.Enum`. В следующем примере выполняется поиск всех допустимых значений для типа перечисления и создается словарь, который сопоставляет эти значения с их строковым представлением.

C#

```
extension<T>(T) where T : System.Enum
{
    public static Dictionary<int, string> EnumNamedValues()
    {
        var result = new Dictionary<int, string>();
        var values = Enum.GetValues(typeof(T));

        foreach (int item in values)
            result.Add(item, Enum.GetName(typeof(T), item)!);
        return result;
    }
}
```

`Enum.GetValues` и `Enum.GetName` используют отражение, которое влияет на производительность. Вы можете не повторять вызовы, требующие отражения, а вызвать `EnumNamedValues` для создания коллекции, которая кэшируется и используется повторно.

Вы можете использовать его, как показано в следующем примере, для создания перечисления и построения словаря значений и имен этого перечисления.

C#

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

C#

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Аргументы типа реализуют объявленный интерфейс

В некоторых сценариях требуется, чтобы аргумент, предоставленный для параметра типа, реализул этот интерфейс. Например:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract T operator +(T left, T right);
    static abstract T operator -(T left, T right);
}
```

Этот шаблон позволяет компилятору C# определить содержащий тип для перегруженных операторов или любого `static virtual` или `static abstract` метода. Он предоставляет синтаксис, чтобы операторы добавления и вычитания могли быть определены для содержащего типа. Без этого ограничения параметры и аргументы должны быть объявлены в качестве интерфейса, а не параметр типа:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract IAdditionSubtraction<T> operator +(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    static abstract IAdditionSubtraction<T> operator -(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

Предыдущий синтаксис требует, чтобы разработчики использовали явную реализацию интерфейса для этих методов. Предоставление дополнительного ограничения позволяет интерфейсу определять операторы с точки зрения параметров типа. Типы, реализующие интерфейс, могут неявно реализовать методы интерфейса.

Разрешает использование ref struct

Анти-ограничение `allows ref struct` объявляет, что соответствующий аргумент типа может быть типом ref struct . Экземпляры этого параметра типа должны соответствовать следующим правилам:

- Он не может быть упакован.
- Он участвует в правилах безопасности ссылок.
- Экземпляры нельзя использовать, если `ref struct` тип не разрешен, например `static` поля.
- Экземпляры можно пометить модификатором `scoped` .

Предложение `allows ref struct` не наследуется. В приведенном ниже коде выполняется следующее:

C#

```
class SomeClass<T, S>
    where T : allows ref struct
    where S : T
{
    // etc
}
```

Аргумент для `S` не может являться `ref struct`, так как `S` не имеет условия `allows ref struct`.

Параметр типа, имеющий `allows ref struct` предложение, нельзя использовать в качестве аргумента типа, если соответствующий параметр типа также не содержит `allows ref struct` предложение. Это правило демонстрируется в следующем примере:

C#

```
public class Allow<T> where T : allows ref struct
{

}

public class Disallow<T>
{
}

public class Example<T> where T : allows ref struct
{
    private Allow<T> fieldOne; // Allowed. T is allowed to be a ref struct

    private Disallow<T> fieldTwo; // Error. T is not allowed to be a ref struct
}
```

В предыдущем примере показано, что аргумент типа, который может быть типом `ref struct` , не может быть заменен параметром типа, который не может быть типом `ref struct` .

См. также

- System.Collections.Generic
- Введение в обобщения
- Универсальные классы

- Новый Constraint

Last updated on 24.02.2026

Универсальные классы (Руководство по программированию на C#)

Статья • 11.04.2023

Универсальные классы инкапсулируют операции, которые не относятся к конкретному типу данных. Универсальные классы чаще всего используются для работы с коллекциями, такими как связанные списки, хэш-таблицы, стеки, очереди, деревья и т. д. Такие операции, как добавление и удаление элементов коллекции, по существу выполняются одинаково, независимо от типа хранимых данных.

В большинстве случаев для этого используются классы коллекций. Рекомендуется выбирать те из них, которые представлены в библиотеке классов платформы .NET. Дополнительные сведения об использовании этих классов см. в разделе Универсальные коллекции в .NET.

Как правило, при создании универсального класса сначала определяется конкретный класс, после чего его типы поочередно заменяются параметрами типов до тех пор, пока не будет достигнут необходимый баланс между степенью обобщения и удобством работы. При создании собственных универсальных классов необходимо учитывать следующие важные моменты:

- Типы, которые требуется обобщить с использованием параметров типа.

  Как правило, чем больше типов параметризовано, тем более гибким и универсальным становится ваш код. Тем не менее слишком высокая степень обобщения может отрицательно сказаться на понятности создаваемого вами кода для других разработчиков.

- Ограничения (если требуются), которые будут применяться к параметрам типа (см. раздел Ограничения параметров типа).

  Рекомендуется применять максимально возможный объем ограничений, при котором вы по-прежнему сможете работать с необходимыми типами. Например, если универсальный класс будет использоваться только для работы со ссылочными типами, примените ограничение класса. Это позволит исключить случайное использование класса с типами значений и позволит использовать оператор `as` в отношении `T`, а также проверять наличие значений null.

- Требуется ли разбивать универсальные функции между базовыми классами и подклассами.

Поскольку универсальные классы могут выступать в качестве базовых классов, здесь необходимо учитывать те же принципы разработки, что и для классов, не являющихся универсальными. См. описание правил наследования от универсальных базовых классов далее в этом разделе.

- Требуется ли реализовывать один или несколько универсальных интерфейсов.

  Например, при разработке класса, который будет использоваться для создания элементов коллекции на основе универсальных шаблонов, может потребоваться реализовать интерфейс IComparable<T>, где `T` — это тип вашего класса.

Пример простого универсального класса можно найти в разделе Введение в универсальные шаблоны.

Правила в отношении параметров типа и ограничений влияют на поведение универсального класса, особенно в контексте наследования и доступа к элементам. Прежде чем продолжить, необходимо ознакомиться с некоторыми терминами и понятиями. Для универсального класса `Node<T>,` клиентский код может ссылаться на класс путем указания аргумента типа для создания закрытого сконструированного типа (`Node<int>`) или путем оставления параметра типа неуказанным, например при указании универсального базового класса, чтобы создать открытый сконструированный тип (`Node<T>`). Универсальные классы могут наследоваться от конкретных, а также закрытых или открытых сконструированных базовых классов:

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Классы, не являющиеся универсальными, то есть конкретные классы, могут наследоваться от закрытых сконструированных базовых классов. Наследование от аналогичных открытых классов или от параметров типа невозможно, поскольку во время выполнения клиентский код не может предоставить аргумент типа, необходимый для создания экземпляра базового класса.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Универсальные классы, наследуемые от открытых сконструированных типов, должны предоставлять аргументы типа для любых параметров типа базового класса, которые не используются совместно с наследующим классом. Это продемонстрировано в следующем коде:

C#

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Универсальные классы, наследуемые от открытых сконструированных типов, должны задавать множество ограничений, которые явно или косвенно включают в себя все ограничения базового типа:

C#

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,
new() { }
```

Универсальные типы могут использовать несколько параметров типа и ограничений, как показано ниже:

C#

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
```

```
    where V : new()
{ }
```

Открытые и закрытые сконструированные типы можно использовать в качестве параметров метода:

```
C#
```

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Если универсальный класс реализует интерфейс, все экземпляры такого класса можно привести к этому интерфейсу.

Универсальные классы инвариантны. Другими словами, если входной параметр задает `List<BaseClass>`, при попытке предоставить `List<DerivedClass>` возникает ошибка времени компиляции.

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Универсальные шаблоны
- Сохранение состояния перечислителей
- Загадка по наследованию, часть 1

Универсальные интерфейсы (руководство по программированию на C#)

Часто рекомендуется определять интерфейсы для универсальных классов коллекций или универсальных классов, представляющих элементы в коллекции. Чтобы избежать операций коробки и распаковки с типами значений, лучше использовать обобщенные интерфейсы, такие как IComparable<T> в обобщенных классах. Библиотека классов .NET определяет несколько универсальных интерфейсов для использования с классами коллекции в System.Collections.Generic пространстве имен. Дополнительные сведения об этих интерфейсах см. в разделе "Универсальные интерфейсы".

Если интерфейс указан в качестве ограничения для параметра типа, можно использовать только типы, реализующие интерфейс. В следующем примере кода показан класс, производный `SortedList<T>` от `GenericList<T>` класса. Дополнительные сведения см. в разделе "Общие сведения". `SortedList<T>` добавляет ограничение `where T : IComparable<T>`. Ограничение позволяет методу `BubbleSort` в `SortedList<T>` использовать универсальный метод CompareTo для элементов списка. В этом примере элементы списка — это простой класс, `Person` реализующий `IComparable<Person>`.

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }
```

```
        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
```

```
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }
```

```
    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names =
        [
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        ];

        int[] ages = [45, 19, 28, 23, 18, 9, 108, 72, 30, 35];

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
```

```
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Несколько интерфейсов можно указать в виде ограничений для одного типа, как показано ниже.

C#

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Интерфейс может определить несколько параметров типа, как показано ниже.

C#

```
interface IDictionary<K, V>
{
}
```

Правила наследования, применяемые к классам, также применяются к интерфейсам:

C#

```
interface IMonth<T> { }

interface IJanuary : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T> : IMonth<T> { }    //No error
                                       //interface IApril<T>  : IMonth<T, U> {}
//Error
```

Универсальные интерфейсы могут наследоваться от не универсальных интерфейсов. В библиотеке классов .NET IEnumerable<T> наследуется от IEnumerable. Если универсальный интерфейс наследует `object` от не универсального интерфейса, параметр типа обычно заменяет элементы переопределенного типа. Например, IEnumerable<T> использует `T` вместо `object` в возвращаемом значении GetEnumerator и в геттере свойства Current. Так как `T` используется только в выходных позициях в этих элементах, IEnumerable<T> его можно пометить как ковариантный. Если `T` он был использован во входной позиции в переопределенном члене, интерфейс не сможет быть ковариантным, и компилятор выдаст ошибку.

Конкретные классы могут реализовать закрытые созданные интерфейсы, как показано ниже.

C#

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Универсальные классы могут реализовать универсальные интерфейсы или закрытые созданные интерфейсы, если список параметров класса предоставляет все аргументы, необходимые интерфейсу, как показано ниже.

C#

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

Правила, управляющие перегрузкой метода, одинаковы для методов в универсальных классах, универсальных структурах или универсальных интерфейсах. Дополнительные сведения см. в разделе "Универсальные методы".

Интерфейсы могут объявлять `static abstract` или `static virtual` члены. Интерфейсы, объявляющие `static abstract` или `static virtual`, почти всегда являются универсальными интерфейсами. Компилятор должен разрешать вызовы `static virtual` и `static abstract` методы во время компиляции. `static virtual` и `static abstract` методы, объявленные в интерфейсах, не имеют механизма диспетчеризации среды выполнения, аналогичного `virtual` методам, `abstract` объявленным в классах. Вместо этого компилятор использует сведения о типе, доступные во время компиляции. Обычно эти члены объявляются в универсальных интерфейсах. Кроме того, большинство интерфейсов, объявляющих или методы, `static virtual` объявляют, что один из параметров типа должен `static abstract`. Затем компилятор использует предоставленные аргументы типа для разрешения типа объявленного члена.

См. также

- Введение в обобщения
- интерфейс
- Обобщения

Last updated on 18.11.2025

Универсальные методы (руководство по программированию на C#)

17.06.2025

Универсальный метод — это метод, объявленный с параметрами типа, как показано ниже.

```C#
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

В следующем примере кода показан один из способов вызова метода с помощью `int` аргумента типа:

```C#
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

Вы также можете опустить аргумент типа, и компилятор выдаст его. Следующий вызов `Swap` эквивалентен предыдущему вызову:

```C#
Swap(ref a, ref b);
```

Те же правила вывода типов применяются к статическим методам и методам экземпляра. Компилятор может определить параметры типа на основе передаваемых аргументов метода; Он не может выводить параметры типа только из ограничения или возвращаемого значения. Поэтому вывод типов не работает с методами, не имеющими параметров. Вывод типов происходит во время компиляции, прежде чем компилятор пытается разрешить перегруженные подписи метода. Компилятор применяет логику

вывода типов ко всем универсальным методам, которые имеют одно и то же имя. На шаге разрешения перегрузки компилятор включает только те универсальные методы, на которых выполнен вывод типов.

В универсальном классе не универсальные методы могут получить доступ к параметрам типа уровня класса следующим образом:

C#

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

Если вы определяете универсальный метод, который принимает те же параметры типа, что и содержащий класс, компилятор создает предупреждение CS0693 , так как в области метода аргумент, предоставленный внутренним `T` , скрывает аргумент, предоставленный для внешнего `T`. Если требуется гибкость вызова метода универсального класса с аргументами типа, отличными от аргументов типа, предоставленных при создании экземпляра класса, рассмотрите возможность предоставления другого идентификатора параметра типа метода, как показано в `GenericList2<T>` следующем примере.

C#

```
class GenericList<T>
{
    // CS0693.
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    // No warning.
    void SampleMethod<U>() { }
}
```

Используйте ограничения для включения более специализированных операций с параметрами типа в методах. Эта версия `Swap<T>`, теперь именованной `SwapIfGreater<T>`, может использоваться только с аргументами типа, реализующими IComparable<T>.

C#

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
```

```
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Универсальные методы могут быть перегружены по нескольким параметрам типа. Например, следующие методы могут находиться в одном классе:

C#

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Вы также можете использовать параметр типа в качестве возвращаемого типа метода. В следующем примере кода показан метод, возвращающий массив типа `T`:

C#

```
T[] Swap<T>(T a, T b)
{
    return [b, a];
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#.

См. также

- System.Collections.Generic
- Введение в обобщения
- Методы

Обобщения и массивы (Руководство по программированию на C#)

17.06.2025

Одномерные массивы, имеющие нижнюю границу нуля, автоматически применяют IList<T>. Это позволяет создавать универсальные методы, которые могут использовать тот же код для итерации через массивы и другие типы коллекций. Этот метод в первую очередь полезен для чтения данных в коллекциях. Интерфейс IList<T> нельзя использовать для добавления или удаления элементов из массива. Будет выброшено исключение, если вы попробуете вызвать метод IList<T> в этом контексте, например, RemoveAt для массива.

В следующем примере кода показано, как один универсальный метод, принимающий входной параметр, IList<T> может выполнять итерацию по списку и массиву, в этом случае массив целых чисел.

C#

```
class Program
{
    static void Main()
    {
        int[] arr = [0, 1, 2, 3, 4];
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item?.ToString() + " ");
```

```
        }
        System.Console.WriteLine();
    }
}
```

См. также

- System.Collections.Generic
- Обобщения
- Массивы
- Обобщения

Универсальные делегаты (руководство по программированию на C#)

17.06.2025

Делегат может определять собственные параметры типа. Код, ссылающийся на универсальный делегат, может указать аргумент типа для создания закрытого созданного типа, как при создании экземпляра универсального класса или вызове универсального метода, как показано в следующем примере:

```
C#
```

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# версии 2.0 имеет новую функцию, называемую преобразованием групп методов, которая применяется к конкретным, а также универсальным типам делегатов и позволяет писать предыдущую строку с помощью этого упрощенного синтаксиса:

```
C#
```

```
Del<int> m2 = Notify;
```

Делегаты, определенные в универсальном классе, могут использовать параметры типа универсального класса таким же образом, как и методы класса.

```
C#
```

```
class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

Код, ссылающийся на делегат, должен указать аргумент типа содержащего класса следующим образом:

```
C#
```

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
```

```
    Stack<float>.StackDelegate d = DoWork;
}
```

Универсальные делегаты особенно полезны при определении событий на основе типичного шаблона проектирования, так как аргумент отправителя может быть строго типизирован и больше не должен приводиться к типу Object и обратно.

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args)
{ }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

См. также

- System.Collections.Generic
- Введение в обобщения
- Универсальные методы
- Универсальные классы
- Универсальные интерфейсы
- делегаты
- Обобщения

Различия между шаблонами C++ и универсальными шаблонами C# (руководство по программированию на C#)

17.06.2025

Универсальные шаблоны C# и C++ — это функции языка, обеспечивающие поддержку параметризованных типов. Однако существует много различий между этими двумя. На уровне синтаксиса универсальные шаблоны C# — это более простой подход к параметризованным типам без сложности шаблонов C++. Кроме того, C# не пытается предоставить все функциональные возможности, предоставляемые шаблонами C++. На уровне реализации основное различие заключается в том, что подстановка универсальных типов C# выполняется во время выполнения, а информация об универсальных типах тем самым сохраняется для созданных объектов. Дополнительные сведения см. в разделе «Универсальные шаблоны в среде выполнения».

Ниже приведены основные различия между универсальными шаблонами C# и шаблонами C++:

- Универсальные шаблоны C# не обеспечивают ту же гибкость, что и шаблоны C++. Например, нельзя вызывать арифметические операторы в универсальном классе C#, хотя можно вызвать определяемые пользователем операторы.
- C# не разрешает параметры шаблона, не относящиеся к типу, например `template C<int i> {}`.
- C# не поддерживает явную специализацию; то есть пользовательская реализация шаблона для определенного типа.
- C# не поддерживает частичную специализацию: пользовательская реализация для подмножества аргументов типа.
- C# не позволяет использовать параметр типа в качестве базового класса для универсального типа.
- C# не позволяет параметрам типа иметь типы по умолчанию.
- В C#параметр универсального типа не может быть универсальным, хотя созданные типы можно использовать в качестве универсальных. C++ разрешает параметры шаблона.

- C++ разрешает код, который может быть недопустимым для всех параметров типа в шаблоне, который затем проверяется для конкретного типа, используемого в качестве параметра типа. C# требует написания кода в классе таким образом, чтобы он работал с любым типом, удовлетворяющим ограничениям. Например, в C++ можно написать функцию, которая использует арифметические операторы `+` и `-` объекты параметра типа, что приведет к ошибке во время создания экземпляра шаблона с типом, который не поддерживает эти операторы. C# запрещает это; Единственными допустимыми конструкциями языка являются те, которые могут быть выведены из ограничений.

См. также

- Введение в обобщения
- Шаблоны

Универсальные шаблоны в среде выполнения (руководство по программированию на C#)

17.06.2025

Когда универсальный тип или метод компилируется на общий промежуточный язык (CIL), он содержит метаданные, которые идентифицируют его как параметры типа. Использование CIL для универсального типа зависит от того, является ли указанный параметр типом значения или ссылочным типом.

Когда универсальный тип сначала создается с типом значения в качестве параметра, среда выполнения создает специализированный универсальный тип с заданным параметром или параметрами, замененными в соответствующих расположениях в CIL. Специализированные универсальные типы создаются один раз для каждого уникального типа значения, используемого в качестве параметра.

Например, предположим, что код программы объявил стек, созданный на основе целых чисел:

```
C#

Stack<int>? stack;
```

На этом этапе среда выполнения создает специализированную версию Stack<T> класса, которая имеет целое число, заменяемое соответствующим образом для его параметра. Теперь, когда код программы использует стек целых чисел, среда выполнения повторно использует созданный специализированный Stack<T> класс. В следующем примере создаются два экземпляра стека целых чисел, и они совместно используют один экземпляр `Stack<int>` кода:

```
C#

Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

Однако предположим, что другой Stack<T> класс с другим типом значения, таким как `long` или определяемая пользователем структура, как его параметр, создается в другой точке кода. В результате среда выполнения создает другую версию универсального типа и подставляет `long` в соответствующих расположениях в CIL. Преобразования больше не

требуются, так как каждый специализированный универсальный класс изначально содержит тип значения.

Универсальные типы работают несколько иначе для ссылочных типов. При первом создании универсального типа с любым ссылочным типом, во время выполнения создается специализированный универсальный тип, в котором ссылки на объекты заменяют параметры в CIL. Затем каждый раз, когда экземпляр созданного типа создается с ссылочным типом в качестве параметра, независимо от конкретного типа, среда выполнения повторно использует ранее созданную специализированную версию универсального типа. Это возможно, так как все ссылки имеют одинаковый размер.

Например, предположим, что у вас есть два ссылочных типа, `Customer` класс и `Order` класс, а также предположим, что вы создали стек `Customer` типов:

C#

```
class Customer { }
class Order { }
```

C#

```
Stack<Customer> customers;
```

На этом этапе среда выполнения создает специализированную версию Stack<T> класса, в которой хранятся ссылки на объекты, которые будут заполнены позже вместо хранения данных. Предположим, что следующая строка кода создает стек другого ссылочного типа, который называется `Order`:

C#

```
Stack<Order> orders = new Stack<Order>();
```

В отличие от типов значений, для типа Stack<T> не создается другая специализированная версия класса `Order`. Вместо этого создается экземпляр специализированной версии Stack<T> класса, и `orders` переменная устанавливается для ссылки на нее. Предположим, что вы столкнулись со строкой кода для создания стека `Customer` типа:

C#

```
customers = new Stack<Customer>();
```

Как и при предыдущем использовании класса, созданного Stack<T> с помощью `Order` типа, создается другой экземпляр специализированного Stack<T> класса. Указатели, содержащиеся в нем, задаются для ссылки на область памяти размером `Customer` типа. Поскольку количество ссылочных типов может отличаться от программы к программе, реализация универсальных шаблонов C# значительно сокращает объем кода, уменьшая до одного числа специализированных классов, созданных компилятором для универсальных классов ссылочных типов.

Кроме того, если универсальный класс C# создается с помощью параметра типа значения или ссылочного типа, отражение может запрашивать его во время выполнения, и его фактический тип и его параметр типа можно определить.

См. также

- System.Collections.Generic
- Введение в обобщения
- Обобщения

Справочник по языку C#

Справочник по языку предоставляет неформальную ссылку на синтаксис C# и идиомы для начинающих и опытных разработчиков C# и .NET.

Справочник по языку C#

ОБЗОР

Стратегия языка C#

ССЫЛКА

Ключевые слова C#

Операторы и выражения C#

Настройка версии языка

Спецификация языка C# — проект C# 8 выполняется

Новые возможности

НОВЫЕ ВОЗМОЖНОСТИ

Новые возможности C# 14

Новые возможности C# 13

Новые возможности C# 12

ССЫЛКА

Критические изменения в компиляторе C#

Совместимость версий

Будьте в курсе

ССЫЛКА

Сообщество разработчиков .NET

YouTube

Twitter

Спецификации

Ознакомьтесь с подробной спецификацией языка C# и подробными спецификациями для последних функций.

Спецификации ECMA и последние функции

ОБЗОР

Процесс спецификации

Подробное содержимое спецификации ECMA

ССЫЛКА

Последние спецификации возможностей

Черновик спецификации ECMA C# — вводный материал

ССЫЛКА

Предисловие

Введение

ССЫЛКА

Область

Нормативные ссылки

Термины и определения

Общее описание

Соответствие

Черновик спецификации ECMA C# — спецификация языка

ССЫЛКА

Лексическая структура

Основные понятия

Типы

Переменные

Преобразования

Шаблоны

Выражения

Выписки

ССЫЛКА

Пространства имен

Классы

Структуры

Массивы

Интерфейсы

Перечисления

Делегаты

Исключения

Атрибуты

Небезопасный код

Черновик спецификации ECMA C# — приложения

ССЫЛКА

грамматики

Проблемы переносимости

Стандартная библиотека

Комментарии для документации

Список литературы