# Managed Extensibility Framework (MEF)

Article • 11/06/2021

This article provides an overview of the Managed Extensibility Framework that was introduced in .NET Framework 4.

## What is MEF?

The Managed Extensibility Framework (MEF) is a library for creating lightweight and extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well.

## The problem of extensibility

Imagine that you are the architect of a large application that must provide support for extensibility. Your application has to include a potentially large number of smaller components, and is responsible for creating and running them.

The simplest approach to the problem is to include the components as source code in your application, and call them directly from your code. This has a number of obvious drawbacks. Most importantly, you cannot add new components without modifying the source code, a restriction that might be acceptable in, for example, a Web application, but is unworkable in a client application. Equally problematic, you may not have access to the source code for the components, because they might be developed by third parties, and for the same reason you cannot allow them to access yours.

A slightly more sophisticated approach would be to provide an extension point or interface, to permit decoupling between the application and its components. Under this model, you might provide an interface that a component can implement, and an API to enable it to interact with your application. This solves the problem of requiring source code access, but it still has its own difficulties.

Because the application lacks any capacity for discovering components on its own, it must still be explicitly told which components are available and should be loaded. This is typically accomplished by explicitly registering the available components in a configuration file. This means that assuring that the components are correct becomes a maintenance issue, particularly if it is the end user and not the developer who is expected to do the updating.

In addition, components are incapable of communicating with one another, except through the rigidly defined channels of the application itself. If the application architect has not anticipated the need for a particular communication, it is usually impossible.

Finally, the component developers must accept a hard dependency on what assembly contains the interface they implement. This makes it difficult for a component to be used in more than one application, and can also create problems when you create a test framework for components.

# What MEF provides

Instead of this explicit registration of available components, MEF provides a way to discover them implicitly, via *composition*. A MEF component, called a *part*, declaratively specifies both its dependencies (known as *imports*) and what capabilities (known as *exports*) it makes available. When a part is created, the MEF composition engine satisfies its imports with what is available from other parts.

This approach solves the problems discussed in the previous section. Because MEF parts declaratively specify their capabilities, they are discoverable at run time, which means an application can make use of parts without either hard-coded references or fragile configuration files. MEF allows applications to discover and examine parts by their metadata, without instantiating them or even loading their assemblies. As a result, there is no need to carefully specify when and how extensions should be loaded.

In addition to its provided exports, a part can specify its imports, which will be filled by other parts. This makes communication among parts not only possible, but easy, and allows for good factoring of code. For example, services common to many components can be factored into a separate part and easily modified or replaced.

Because the MEF model requires no hard dependency on a particular application assembly, it allows extensions to be reused from application to application. This also makes it easy to develop a test harness, independent of the application, to test extension components.

An extensible application written by using MEF declares an import that can be filled by extension components, and may also declare exports in order to expose application services to extensions. Each extension component declares an export, and may also declare imports. In this way, extension components themselves are automatically extensible.

# Where MEF is available

MEF is an integral part of the .NET Framework 4, and is available wherever the .NET Framework is used. You can use MEF in your client applications, whether they use Windows Forms, WPF, or any other technology, or in server applications that use ASP.NET.

## MEF and MAF

Previous versions of the .NET Framework introduced the Managed Add-in Framework (MAF), designed to allow applications to isolate and manage extensions. The focus of MAF is slightly higher-level than MEF, concentrating on extension isolation and assembly loading and unloading, while MEF's focus is on discoverability, extensibility, and portability. The two frameworks interoperate smoothly, and a single application can take advantage of both.

## SimpleCalculator: An example application

The simplest way to see what MEF can do is to build a simple MEF application. In this example, you build a very simple calculator named SimpleCalculator. The goal of SimpleCalculator is to create a console application that accepts basic arithmetic commands, in the form "5+3" or "6-2", and returns the correct answers. Using MEF, you'll be able to add new operators without changing the application code.

To download the complete code for this example, see the SimpleCalculator sample (Visual Basic).

> **Note**
>
> The purpose of SimpleCalculator is to demonstrate the concepts and syntax of MEF, rather than to necessarily provide a realistic scenario for its use. Many of the applications that would benefit most from the power of MEF are more complex than SimpleCalculator. For more extensive examples, see the Managed Extensibility Framework on GitHub.

- To start, in Visual Studio, create a new Console Application project and name it `SimpleCalculator`.
- Add a reference to the `System.ComponentModel.Composition` assembly, where MEF resides.

- Open *Module1.vb* or *Program.cs* and add `Imports` or `using` directives for `System.ComponentModel.Composition` and `System.ComponentModel.Composition.Hosting`. These two namespaces contain MEF types you will need to develop an extensible application.
- If you're using Visual Basic, add the `Public` keyword to the line that declares the `Module1` module.

# Composition container and catalogs

The core of the MEF composition model is the *composition container*, which contains all the parts available and performs composition. Composition is the matching up of imports to exports. The most common type of composition container is CompositionContainer, and you'll use this for SimpleCalculator.

If you're using Visual Basic, add a public class named `Program` in *Module1.vb*.

Add the following line to the `Program` class in *Module1.vb* or *Program.cs*:

C#

```
private CompositionContainer _container;
```

In order to discover the parts available to it, the composition containers makes use of a *catalog*. A catalog is an object that makes available parts discovered from some source. MEF provides catalogs to discover parts from a provided type, an assembly, or a directory. Application developers can easily create new catalogs to discover parts from other sources, such as a Web service.

Add the following constructor to the `Program` class:

C#

```
private Program()
{
    try
    {
        // An aggregate catalog that combines multiple catalogs.
        var catalog = new AggregateCatalog();
        // Adds all the parts found in the same assembly as the Program
class.
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));

        // Create the CompositionContainer with the parts in the catalog.
        _container = new CompositionContainer(catalog);
```

```
            _container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            Console.WriteLine(compositionException.ToString());
        }
    }
```

The call to ComposeParts tells the composition container to compose a specific set of parts, in this case the current instance of `Program`. At this point, however, nothing will happen, since `Program` has no imports to fill.

## Imports and Exports with attributes

First, you have `Program` import a calculator. This allows the separation of user interface concerns, such as the console input and output that will go into `Program`, from the logic of the calculator.

Add the following code to the `Program` class:

C#

```
[Import(typeof(ICalculator))]
public ICalculator calculator;
```

Notice that the declaration of the `calculator` object is not unusual, but that it is decorated with the ImportAttribute attribute. This attribute declares something to be an import; that is, it will be filled by the composition engine when the object is composed.

Every import has a *contract*, which determines what exports it will be matched with. The contract can be an explicitly specified string, or it can be automatically generated by MEF from a given type, in this case the interface `ICalculator`. Any export declared with a matching contract will fulfill this import. Note that while the type of the `calculator` object is in fact `ICalculator`, this is not required. The contract is independent from the type of the importing object. (In this case, you could leave out the `typeof(ICalculator)`. MEF will automatically assume the contract to be based on the type of the import unless you specify it explicitly.)

Add this very simple interface to the module or `SimpleCalculator` namespace:

C#

```
public interface ICalculator
{
```

```
    string Calculate(string input);
}
```

Now that you have defined `ICalculator`, you need a class that implements it. Add the following class to the module or `SimpleCalculator` namespace:

C#

```
[Export(typeof(ICalculator))]
class MySimpleCalculator : ICalculator
{

}
```

Here is the export that will match the import in `Program`. In order for the export to match the import, the export must have the same contract. Exporting under a contract based on `typeof(MySimpleCalculator)` would produce a mismatch, and the import would not be filled; the contract needs to match exactly.

Since the composition container will be populated with all the parts available in this assembly, the `MySimpleCalculator` part will be available. When the constructor for `Program` performs composition on the `Program` object, its import will be filled with a `MySimpleCalculator` object, which will be created for that purpose.

The user interface layer (`Program`) does not need to know anything else. You can therefore fill in the rest of the user interface logic in the `Main` method.

Add the following code to the `Main` method:

C#

```
static void Main(string[] args)
{
    // Composition is performed in the constructor.
    var p = new Program();
    Console.WriteLine("Enter Command:");
    while (true)
    {
        string s = Console.ReadLine();
        Console.WriteLine(p.calculator.Calculate(s));
    }
}
```

This code simply reads a line of input and calls the `Calculate` function of `ICalculator` on the result, which it writes back to the console. That is all the code you need in

`Program`. All the rest of the work will happen in the parts.

# Imports and ImportMany attributes

In order for SimpleCalculator to be extensible, it needs to import a list of operations. An ordinary ImportAttribute attribute is filled by one and only one ExportAttribute. If more than one is available, the composition engine produces an error. To create an import that can be filled by any number of exports, you can use the ImportManyAttribute attribute.

Add the following operations property to the `MySimpleCalculator` class:

C#

```
[ImportMany]
IEnumerable<Lazy<IOperation, IOperationData>> operations;
```

Lazy<T,TMetadata> is a type provided by MEF to hold indirect references to exports. Here, in addition to the exported object itself, you also get *export metadata*, or information that describes the exported object. Each Lazy<T,TMetadata> contains an `IOperation` object, representing an actual operation, and an `IOperationData` object, representing its metadata.

Add the following simple interfaces to the module or `SimpleCalculator` namespace:

C#

```
public interface IOperation
{
     int Operate(int left, int right);
}

public interface IOperationData
{
    char Symbol { get; }
}
```

In this case, the metadata for each operation is the symbol that represents that operation, such as +, -, *, and so on. To make the addition operation available, add the following class to the module or `SimpleCalculator` namespace:

C#

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '+')]
```

```
class Add: IOperation
{
    public int Operate(int left, int right)
    {
        return left + right;
    }
}
```

The ExportAttribute attribute functions as it did before. The ExportMetadataAttribute attribute attaches metadata, in the form of a name-value pair, to that export. While the `Add` class implements `IOperation`, a class that implements `IOperationData` is not explicitly defined. Instead, a class is implicitly created by MEF with properties based on the names of the metadata provided. (This is one of several ways to access metadata in MEF.)

Composition in MEF is *recursive*. You explicitly composed the `Program` object, which imported an `ICalculator` that turned out to be of type `MySimpleCalculator`. `MySimpleCalculator`, in turn, imports a collection of `IOperation` objects, and that import will be filled when `MySimpleCalculator` is created, at the same time as the imports of `Program`. If the `Add` class declared a further import, that too would have to be filled, and so on. Any import left unfilled results in a composition error. (It is possible, however, to declare imports to be optional or to assign them default values.)

# Calculator logic

With these parts in place, all that remains is the calculator logic itself. Add the following code in the `MySimpleCalculator` class to implement the `Calculate` method:

C#

```
public String Calculate(string input)
{
    int left;
    int right;
    char operation;
    // Finds the operator.
    int fn = FindFirstNonDigit(input);
    if (fn < 0) return "Could not parse command.";

    try
    {
        // Separate out the operands.
        left = int.Parse(input.Substring(0, fn));
        right = int.Parse(input.Substring(fn + 1));
    }
    catch
```

```
    {
        return "Could not parse command.";
    }

    operation = input[fn];

    foreach (Lazy<IOperation, IOperationData> i in operations)
    {
        if (i.Metadata.Symbol.Equals(operation))
        {
            return i.Value.Operate(left, right).ToString();
        }
    }
    return "Operation Not Found!";
}
```

The initial steps parse the input string into left and right operands and an operator character. In the `foreach` loop, every member of the `operations` collection is examined. These objects are of type Lazy<T,TMetadata>, and their metadata values and exported object can be accessed with the Metadata property and the Value property respectively. In this case, if the `Symbol` property of the `IOperationData` object is discovered to be a match, the calculator calls the `Operate` method of the `IOperation` object and returns the result.

To complete the calculator, you also need a helper method that returns the position of the first non-digit character in a string. Add the following helper method to the `MySimpleCalculator` class:

C#

```
private int FindFirstNonDigit(string s)
{
    for (int i = 0; i < s.Length; i++)
    {
        if (!char.IsDigit(s[i])) return i;
    }
    return -1;
}
```

You should now be able to compile and run the project. In Visual Basic, make sure that you added the `Public` keyword to `Module1`. In the console window, type an addition operation, such as "5+3", and the calculator returns the results. Any other operator results in the "Operation Not Found!" message.

# Extend SimpleCalculator using a new class

Now that the calculator works, adding a new operation is easy. Add the following class to the module or `SimpleCalculator` namespace:

C#

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '-')]
class Subtract : IOperation
{
    public int Operate(int left, int right)
    {
        return left - right;
    }
}
```

Compile and run the project. Type a subtraction operation, such as "5-3". The calculator now supports subtraction as well as addition.

# Extend SimpleCalculator using a new assembly

Adding classes to the source code is simple enough, but MEF provides the ability to look outside an application's own source for parts. To demonstrate this, you will need to modify SimpleCalculator to search a directory, as well as its own assembly, for parts, by adding a DirectoryCatalog.

Add a new directory named `Extensions` to the SimpleCalculator project. Make sure to add it at the project level, and not at the solution level. Then add a new Class Library project to the solution, named `ExtendedOperations`. The new project will compile into a separate assembly.

Open the Project Properties Designer for the ExtendedOperations project and click the **Compile** or **Build** tab. Change the **Build output path** or **Output path** to point to the Extensions directory in the SimpleCalculator project directory (*..\SimpleCalculator\Extensions*).

In *Module1.vb* or *Program.cs*, add the following line to the `Program` constructor:

C#

```
catalog.Catalogs.Add(
    new DirectoryCatalog(
        "C:\\SimpleCalculator\\SimpleCalculator\\Extensions"));
```

Replace the example path with the path to your Extensions directory. (This absolute path is for debugging purposes only. In a production application, you would use a relative path.) The DirectoryCatalog will now add any parts found in any assemblies in the Extensions directory to the composition container.

In the `ExtendedOperations` project, add references to `SimpleCalculator` and `System.ComponentModel.Composition`. In the `ExtendedOperations` class file, add an `Imports` or a `using` directive for `System.ComponentModel.Composition`. In Visual Basic, also add an `Imports` statement for `SimpleCalculator`. Then add the following class to the `ExtendedOperations` class file:

C#

```
[Export(typeof(SimpleCalculator.IOperation))]
[ExportMetadata("Symbol", '%')]
public class Mod : SimpleCalculator.IOperation
{
    public int Operate(int left, int right)
    {
        return left % right;
    }
}
```

Note that in order for the contract to match, the ExportAttribute attribute must have the same type as the ImportAttribute.

Compile and run the project. Test the new Mod (%) operator.

# Conclusion

This topic covered the basic concepts of MEF.

- Parts, catalogs, and the composition container

  Parts and the composition container are the basic building blocks of a MEF application. A part is any object that imports or exports a value, up to and including itself. A catalog provides a collection of parts from a particular source. The composition container uses the parts provided by a catalog to perform composition, the binding of imports to exports.

- Imports and exports

  Imports and exports are the way by which components communicate. With an import, the component specifies a need for a particular value or object, and with

an export it specifies the availability of a value. Each import is matched with a list of exports by way of its contract.

## Next steps

To download the complete code for this example, see the SimpleCalculator sample (Visual Basic).

For more information and code examples, see Managed Extensibility Framework. For a list of the MEF types, see the System.ComponentModel.Composition namespace.

# Attributed programming model overview (MEF)

Article • 12/08/2021

In the Managed Extensibility Framework (MEF), a *programming model* is a particular method of defining the set of conceptual objects on which MEF operates. These conceptual objects include parts, imports, and exports. MEF uses these objects, but does not specify how they should be represented. Therefore, a wide variety of programming models are possible, including customized programming models.

The default programming model used in MEF is the *attributed programming model*. In the attributed programming model parts, imports, exports, and other objects are defined with attributes that decorate ordinary .NET Framework classes. This topic explains how to use the attributes provided by the attributed programming model to create a MEF application.

## Import and Export Basics

An *export* is a value that a part provides to other parts in the container, and an *import* is a requirement that a part expresses to the container, to be filled from the available exports. In the attributed programming model, imports and exports are declared by decorating classes or members with the `Import` and `Export` attributes. The `Export` attribute can decorate a class, field, property, or method, while the `Import` attribute can decorate a field, property, or constructor parameter.

In order for an import to be matched with an export, the import and export must have the same *contract*. The contract consists of a string, called the *contract name*, and the type of the exported or imported object, called the *contract type*. Only if both the contract name and contract type match is an export considered to fulfill a particular import.

Either or both of the contract parameters can be implicit or explicit. The following code shows a class that declares a basic import.

C#

```
public class MyClass
{
    [Import]
    public IMyAddin MyAddin { get; set; }
}
```

In this import, the `Import` attribute has neither a contract type nor a contract name parameter attached. Therefore, both will be inferred from the decorated property. In this case, the contract type will be `IMyAddin`, and the contract name will be a unique string created from the contract type. (In other words, the contract name will match only exports whose names are also inferred from the type `IMyAddin`.)

The following shows an export that matches the previous import.

C#

```
[Export(typeof(IMyAddin))]
public class MyLogger : IMyAddin { }
```

In this export, the contract type is `IMyAddin` because it is specified as a parameter of the `Export` attribute. The exported type must be either the same as the contract type, derive from the contract type, or implement the contract type if it is an interface. In this export, the actual type `MyLogger` implements the interface `IMyAddin`. The contract name is inferred from the contract type, which means that this export will match the previous import.

> **Note**
>
> Exports and imports should usually be declared on public classes or members. Other declarations are supported, but exporting or importing a private, protected, or internal member breaks the isolation model for the part and is therefore not recommended.

The contract type must match exactly for the export and import to be considered a match. Consider the following export.

C#

```
[Export] //WILL NOT match the previous import!
public class MyLogger : IMyAddin { }
```

In this export, the contract type is `MyLogger` instead of `IMyAddin`. Even though `MyLogger` implements `IMyAddin`, and therefore could be cast to an `IMyAddin` object, this export will not match the previous import because the contract types are not the same.

In general, it is not necessary to specify the contract name, and most contracts should be defined in terms of the contract type and metadata. However, under certain

circumstances, it is important to specify the contract name directly. The most common case is when a class exports several values that share a common type, such as primitives. The contract name can be specified as the first parameter of the `Import` or `Export` attribute. The following code shows an import and an export with a specified contract name of `MajorRevision`.

C#

```
public class MyClass
{
    [Import("MajorRevision")]
    public int MajorRevision { get; set; }
}

public class MyExportClass
{
    [Export("MajorRevision")] //This one will match.
    public int MajorRevision = 4;

    [Export("MinorRevision")]
    public int MinorRevision = 16;
}
```

If the contract type is not specified, it will still be inferred from the type of the import or export. However, even if the contract name is specified explicitly, the contract type must also match exactly for the import and export to be considered a match. For example, if the `MajorRevision` field was a string, the inferred contract types would not match and the export would not match the import, despite having the same contract name.

## Importing and Exporting a Method

The `Export` attribute can also decorate a method, in the same way as a class, property, or function. Method exports must specify a contract type or contract name, as the type cannot be inferred. The specified type can be either a custom delegate or a generic type, such as `Func`. The following class exports a method named `DoSomething`.

C#

```
public class MyAddin
{
    //Explicitly specifying a generic type.
    [Export(typeof(Func<int, string>))]
    public string DoSomething(int TheParam);
}
```

In this class, the `DoSomething` method takes a single `int` parameter and returns a `string`. To match this export, the importing part must declare an appropriate member. The following class imports the `DoSomething` method.

C#

```
public class MyClass
{
    [Import] //Contract name must match!
    public Func<int, string> DoSomething { get; set; }
}
```

For more information about how to use of the `Func<T, T>` object, see Func<T,TResult>.

# Types of Imports

MEF support several import types, including dynamic, lazy, prerequisite, and optional.

## Dynamic Imports

In some cases, the importing class may want to match exports of any type that have a particular contract name. In this scenario, the class can declare a *dynamic import*. The following import matches any export with contract name "TheString".

C#

```
public class MyClass
{
    [Import("TheString")]
    public dynamic MyAddin { get; set; }
}
```

When the contract type is inferred from the `dynamic` keyword, it will match any contract type. In this case, an import should **always** specify a contract name to match on. (If no contract name is specified, the import will be considered to match no exports.) Both of the following exports would match the previous import.

C#

```
[Export("TheString", typeof(IMyAddin))]
public class MyLogger : IMyAddin { }

[Export("TheString")]
public class MyToolbar { }
```

Obviously, the importing class must be prepared to deal with an object of arbitrary type.

## Lazy Imports

In some cases, the importing class may require an indirect reference to the imported object, so that the object is not instantiated immediately. In this scenario, the class can declare a *lazy import* by using a contract type of `Lazy<T>`. The following importing property declares a lazy import.

C#

```
public class MyClass
{
    [Import]
    public Lazy<IMyAddin> MyAddin { get; set; }
}
```

From the point of view of the composition engine, a contract type of `Lazy<T>` is considered identical to contract type of `T`. Therefore, the previous import would match the following export.

C#

```
[Export(typeof(IMyAddin))]
public class MyLogger : IMyAddin { }
```

The contract name and contract type can be specified in the `Import` attribute for a lazy import, as described earlier in the "Basic Imports and Exports" section.

## Prerequisite Imports

Exported MEF parts are typically created by the composition engine, in response to a direct request or the need to fill a matched import. By default, when creating a part, the composition engine uses the parameter-less constructor. To make the engine use a different constructor, you can mark it with the `ImportingConstructor` attribute.

Each part may have only one constructor for use by the composition engine. Providing no parameterless constructor and no `ImportingConstructor` attribute, or providing more than one `ImportingConstructor` attribute, will produce an error.

To fill the parameters of a constructor marked with the `ImportingConstructor` attribute, all of those parameters are automatically declared as imports. This is a convenient way

to declare imports that are used during part initialization. The following class uses `ImportingConstructor` to declare an import.

C#

```
public class MyClass
{
    private IMyAddin _theAddin;

    //Parameterless constructor will NOT be
    //used because the ImportingConstructor
    //attribute is present.
    public MyClass() { }

    //This constructor will be used.
    //An import with contract type IMyAddin is
    //declared automatically.
    [ImportingConstructor]
    public MyClass(IMyAddin MyAddin)
    {
        _theAddin = MyAddin;
    }
}
```

By default, the `ImportingConstructor` attribute uses inferred contract types and contract names for all of the parameter imports. It is possible to override this by decorating the parameters with `Import` attributes, which can then define the contract type and contract name explicitly. The following code demonstrates a constructor that uses this syntax to import a derived class instead of a parent class.

C#

```
[ImportingConstructor]
public MyClass([Import(typeof(IMySubAddin))]IMyAddin MyAddin)
{
    _theAddin = MyAddin;
}
```

In particular, you should be careful with collection parameters. For example, if you specify `ImportingConstructor` on a constructor with a parameter of type `IEnumerable<int>`, the import will match a single export of type `IEnumerable<int>`, instead of a set of exports of type `int`. To match a set of exports of type `int`, you have to decorate the parameter with the `ImportMany` attribute.

Parameters declared as imports by the `ImportingConstructor` attribute are also marked as *prerequisite imports*. MEF normally allows exports and imports to form a *cycle*. For

example, a cycle is where object A imports object B, which in turn imports object A. Under ordinary circumstances, a cycle is not a problem, and the composition container constructs both objects normally.

When an imported value is required by the constructor of a part, that object cannot participate in a cycle. If object A requires that object B be constructed before it can be constructed itself, and object B imports object A, then the cycle will be unable to resolve and a composition error will occur. Imports declared on constructor parameters are therefore prerequisite imports, which must all be filled before any of the exports from the object that requires them can be used.

## Optional Imports

The `Import` attribute specifies a requirement for the part to function. If an import cannot be fulfilled, the composition of that part will fail and the part will not be available.

You can specify that an import is *optional* by using the `AllowDefault` property. In this case, the composition will succeed even if the import does not match any available exports, and the importing property will be set to the default for its property type (`null` for object properties, `false` for Booleans, or zero for numeric properties.) The following class uses an optional import.

C#

```
public class MyClass
{
    [Import(AllowDefault = true)]
    public Plugin thePlugin { get; set; }

    //If no matching export is available,
    //thePlugin will be set to null.
}
```

## Importing Multiple Objects

The `Import` attribute will only be successfully composed when it matches one and only one export. Other cases will produce a composition error. To import more than one export that matches the same contract, use the `ImportMany` attribute. Imports marked with this attribute are always optional. For example, composition will not fail if no matching exports are present. The following class imports any number of exports of type `IMyAddin`.

C#

```
public class MyClass
{
    [ImportMany]
    public IEnumerable<IMyAddin> MyAddin { get; set; }
}
```

The imported array can be accessed by using ordinary `IEnumerable<T>` syntax and methods. It is also possible to use an ordinary array (`IMyAddin[]`) instead.

This pattern can be very important when you use it in combination with the `Lazy<T>` syntax. For example, by using `ImportMany`, `IEnumerable<T>`, and `Lazy<T>`, you can import indirect references to any number of objects and only instantiate the ones that become necessary. The following class shows this pattern.

C#

```
public class MyClass
{
    [ImportMany]
    public IEnumerable<Lazy<IMyAddin>> MyAddin { get; set; }
}
```

# Avoiding Discovery

In some cases, you may want to prevent a part from being discovered as part of a catalog. For example, the part may be a base class intended to be inherited from, but not used. There are two ways to accomplish this. First, you can use the `abstract` keyword on the part class. Abstract classes never provide exports, although they can provide inherited exports to classes that derive from them.

If the class cannot be made abstract, you can decorate it with the `PartNotDiscoverable` attribute. A part decorated with this attribute will not be included in any catalogs. The following example demonstrates these patterns. `DataOne` will be discovered by the catalog. Since `DataTwo` is abstract, it will not be discovered. Since `DataThree` used the `PartNotDiscoverable` attribute, it will not be discovered.

C#

```
[Export]
public class DataOne
{
    //This part will be discovered
    //as normal by the catalog.
}
```

```
[Export]
public abstract class DataTwo
{
    //This part will not be discovered
    //by the catalog.
}

[PartNotDiscoverable]
[Export]
public class DataThree
{
    //This part will also not be discovered
    //by the catalog.
}
```

# Metadata and Metadata Views

Exports can provide additional information about themselves known as *metadata*. Metadata can be used to convey properties of the exported object to the importing part. The importing part can use this data to decide which exports to use, or to gather information about an export without having to construct it. For this reason, an import must be lazy to use metadata.

To use metadata, you typically declare an interface known as a *metadata view*, which declares what metadata will be available. The metadata view interface must have only properties, and those properties must have `get` accessors. The following interface is an example metadata view.

C#

```
public interface IPluginMetadata
{
    string Name { get; }

    [DefaultValue(1)]
    int Version { get; }
}
```

It is also possible to use a generic collection, `IDictionary<string, object>`, as a metadata view, but this forfeits the benefits of type checking and should be avoided.

Ordinarily, all of the properties named in the metadata view are required, and any exports that do not provide them will not be considered a match. The `DefaultValue` attribute specifies that a property is optional. If the property is not included, it will be assigned the default value specified as a parameter of `DefaultValue`. The following are

two different classes decorated with metadata. Both of these classes would match the previous metadata view.

C#

```
[Export(typeof(IPlugin)),
    ExportMetadata("Name", "Logger"),
    ExportMetadata("Version", 4)]
public class Logger : IPlugin
{
}

[Export(typeof(IPlugin)),
    ExportMetadata("Name", "Disk Writer")]
    //Version is not required because of the DefaultValue
public class DWriter : IPlugin
{
}
```

Metadata is expressed after the `Export` attribute by using the `ExportMetadata` attribute. Each piece of metadata is composed of a name/value pair. The name portion of the metadata must match the name of the appropriate property in the metadata view, and the value will be assigned to that property.

It is the importer that specifies what metadata view, if any, will be in use. An import with metadata is declared as a lazy import, with the metadata interface as the second type parameter to `Lazy<T,T>`. The following class imports the previous part with metadata.

C#

```
public class Addin
{
    [Import]
    public Lazy<IPlugin, IPluginMetadata> plugin;
}
```

In many cases, you will want to combine metadata with the `ImportMany` attribute, in order to parse through the available imports and choose and instantiate only one, or filter a collection to match a certain condition. The following class instantiates only `IPlugin` objects that have the `Name` value "Logger".

C#

```
public class User
{
    [ImportMany]
    public IEnumerable<Lazy<IPlugin, IPluginMetadata>> plugins;
```

```
    public IPlugin InstantiateLogger()
    {
        IPlugin logger = null;

        foreach (Lazy<IPlugin, IPluginMetadata> plugin in plugins)
        {
            if (plugin.Metadata.Name == "Logger")
                logger = plugin.Value;
        }
        return logger;
    }
}
```

# Import and Export Inheritance

If a class inherits from a part, that class may also become a part. Imports are always inherited by subclasses. Therefore, a subclass of a part will always be a part, with the same imports as its parent class.

Exports declared by using the `Export` attribute are not inherited by subclasses. However, a part can export itself by using the `InheritedExport` attribute. Subclasses of the part will inherit and provide the same export, including contract name and contract type. Unlike an `Export` attribute, `InheritedExport` can be applied only at the class level, and not at the member level. Therefore, member-level exports can never be inherited.

The following four classes demonstrate the principles of import and export inheritance. `NumTwo` inherits from `NumOne`, so `NumTwo` will import `IMyData`. Ordinary exports are not inherited, so `NumTwo` will not export anything. `NumFour` inherits from `NumThree`. Because `NumThree` used `InheritedExport`, `NumFour` has one export with contract type `NumThree`. Member-level exports are never inherited, so `IMyData` is not exported.

C#

```
[Export]
public class NumOne
{
    [Import]
    public IMyData MyData { get; set; }
}

public class NumTwo : NumOne
{
    //Imports are always inherited, so NumTwo will
    //import IMyData.

    //Ordinary exports are not inherited, so
```

```
    //NumTwo will NOT export anything. As a result it
    //will not be discovered by the catalog!
}

[InheritedExport]
public class NumThree
{
    [Export]
    Public IMyData MyData { get; set; }

    //This part provides two exports, one of
    //contract type NumThree, and one of
    //contract type IMyData.
}

public class NumFour : NumThree
{
    //Because NumThree used InheritedExport,
    //this part has one export with contract
    //type NumThree.

    //Member-level exports are never inherited,
    //so IMyData is not exported.
}
```

If there is metadata associated with an `InheritedExport` attribute, that metadata will also be inherited. (For more information, see the earlier "Metadata and Metadata Views" section.) Inherited metadata cannot be modified by the subclass. However, by re-declaring the `InheritedExport` attribute with the same contract name and contract type, but with new metadata, the subclass can replace the inherited metadata with new metadata. The following class demonstrates this principle. The `MegaLogger` part inherits from `Logger` and includes the `InheritedExport` attribute. Since `MegaLogger` re-declares new metadata named Status, it does not inherit the Name and Version metadata from `Logger`.

C#

```
[InheritedExport(typeof(IPlugin)),
    ExportMetadata("Name", "Logger"),
    ExportMetadata("Version", 4)]
public class Logger : IPlugin
{
    //Exports with contract type IPlugin and
    //metadata "Name" and "Version".
}

public class SuperLogger : Logger
{
    //Exports with contract type IPlugin and
    //metadata "Name" and "Version", exactly the same
```

```
    //as the Logger class.
}

[InheritedExport(typeof(IPlugin)),
    ExportMetadata("Status", "Green")]
public class MegaLogger : Logger        {
    //Exports with contract type IPlugin and
    //metadata "Status" only. Re-declaring
    //the attribute replaces all metadata.
}
```

When re-declaring the `InheritedExport` attribute to override metadata, make sure that the contract types are the same. (In the previous example, `IPlugin` is the contract type.) If they differ, instead of overriding, the second attribute will create a second, independent export from the part. Generally, this means that you will have to explicitly specify the contract type when you override an `InheritedExport` attribute, as shown in the previous example.

Since interfaces cannot be instantiated directly, they generally cannot be decorated with `Export` or `Import` attributes. However, an interface can be decorated with an `InheritedExport` attribute at the interface level, and that export along with any associated metadata will be inherited by any implementing classes. The interface itself will not be available as a part, however.

# Custom Export Attributes

The basic export attributes, `Export` and `InheritedExport`, can be extended to include metadata as attribute properties. This technique is useful for applying similar metadata to many parts, or creating an inheritance tree of metadata attributes.

A custom attribute can specify the contract type, the contract name, or any other metadata. In order to define a custom attribute, a class inheriting from `ExportAttribute` (or `InheritedExportAttribute`) must be decorated with the `MetadataAttribute` attribute. The following class defines a custom attribute.

C#

```
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public class MyAttribute : ExportAttribute
{
    public MyAttribute(string myMetadata)
        : base(typeof(IMyAddin))
    {
        MyMetadata = myMetadata;
```

```
    }

    public string MyMetadata { get; private set; }
}
```

This class defines a custom attribute named `MyAttribute` with contract type `IMyAddin` and some metadata named `MyMetadata`. All properties in a class marked with the `MetadataAttribute` attribute are considered to be metadata defined in the custom attribute. The following two declarations are equivalent.

C#

```
[Export(typeof(IMyAddin)),
    ExportMetadata("MyMetadata", "theData")]
public MyAddin myAddin { get; set; }
```

C#

```
[MyAttribute("theData")]
public MyAddin myAddin { get; set; }
```

In the first declaration, the contract type and metadata are explicitly defined. In the second declaration, the contract type and metadata are implicit in the customized attribute. Particularly in cases where a large amount of identical metadata must be applied to many parts (for example, author or copyright information), using a custom attribute can save a lot of time and duplication. Further, inheritance trees of custom attributes can be created to allow for variations.

To create optional metadata in a custom attribute, you can use the `DefaultValue` attribute. When this attribute is applied to a property in a custom attribute class, it specifies that the decorated property is optional and does not have to be supplied by an exporter. If a value for the property is not supplied, it will be assigned the default value for its property type (usually `null`, `false`, or 0.)

# Creation Policies

When a part specifies an import and composition is performed, the composition container attempts to find a matching export. If it matches the import with an export successfully, the importing member is set to an instance of the exported object. Where this instance comes from is controlled by the exporting part's *creation policy*.

The two possible creation policies are *shared* and *non-shared*. A part with a creation policy of shared will be shared between every import in the container for a part with that contract. When the composition engine finds a match and has to set an importing property, it will instantiate a new copy of the part only if one does not already exist; otherwise, it will supply the existing copy. This means that many objects may have references to the same part. Such parts should not rely on internal state that might be changed from many places. This policy is appropriate for static parts, parts that provide services, and parts that consume a lot of memory or other resources.

A part with the creation policy of non-shared will be created every time a matching import for one of its exports is found. A new copy will therefore be instantiated for every import in the container that matches one of the part's exported contracts. The internal state of these copies will not be shared. This policy is appropriate for parts where each import requires its own internal state.

Both the import and the export can specify the creation policy of a part, from among the values `Shared`, `NonShared`, or `Any`. The default is `Any` for both imports and exports. An export that specifies `Shared` or `NonShared` will only match an import that specifies the same, or that specifies `Any`. Similarly, an import that specifies `Shared` or `NonShared` will only match an export that specifies the same, or that specifies `Any`. Imports and exports with incompatible creation policies are not considered a match, in the same way as an import and export whose contract name or contract type are not a match. If both import and export specify `Any`, or do not specify a creation policy and default to `Any`, the creation policy will default to shared.

The following example shows both imports and exports specifying creation policies. `PartOne` does not specify a creation policy, so the default is `Any`. `PartTwo` does not specify a creation policy, so the default is `Any`. Since both import and export default to `Any`, `PartOne` will be shared. `PartThree` specifies a `Shared` creation policy, so `PartTwo` and `PartThree` will share the same copy of `PartOne`. `PartFour` specifies a `NonShared` creation policy, so `PartFour` will be non-shared in `PartFive`. `PartSix` specifies a `NonShared` creation policy. `PartFive` and `PartSix` will each receive separate copies of `PartFour`. `PartSeven` specifies a `Shared` creation policy. Because there is no exported `PartFour` with a creation policy of `Shared`, the `PartSeven` import does not match anything and will not be filled.

C#

```
[Export]
public class PartOne
{
    //The default creation policy for an export is Any.
```

```
}

public class PartTwo
{
    [Import]
    public PartOne partOne { get; set; }

    //The default creation policy for an import is Any.
    //If both policies are Any, the part will be shared.
}

public class PartThree
{
    [Import(RequiredCreationPolicy = CreationPolicy.Shared)]
    public PartOne partOne { get; set; }

    //The Shared creation policy is explicitly specified.
    //PartTwo and PartThree will receive references to the
    //SAME copy of PartOne.
}

[Export]
[PartCreationPolicy(CreationPolicy.NonShared)]
public class PartFour
{
    //The NonShared creation policy is explicitly specified.
}

public class PartFive
{
    [Import]
    public PartFour partFour { get; set; }

    //The default creation policy for an import is Any.
    //Since the export's creation policy was explicitly
    //defined, the creation policy for this property will
    //be non-shared.
}

public class PartSix
{
    [Import(RequiredCreationPolicy = CreationPolicy.NonShared)]
    public PartFour partFour { get; set; }

    //Both import and export specify matching creation
    //policies.  PartFive and PartSix will each receive
    //SEPARATE copies of PartFour, each with its own
    //internal state.
}

public class PartSeven
{
    [Import(RequiredCreationPolicy = CreationPolicy.Shared)]
    public PartFour partFour { get; set; }
```

```
    //A creation policy mismatch.  Because there is no
    //exported PartFour with a creation policy of Shared,
    //this import does not match anything and will not be
    //filled.
}
```

# Life Cycle and Disposing

Because parts are hosted in the composition container, their life cycle can be more complex than ordinary objects. Parts can implement two important life cycle-related interfaces: `IDisposable` and `IPartImportsSatisfiedNotification`.

Parts that require work to be performed at shut down or that need to release resources should implement `IDisposable`, as usual for .NET Framework objects. However, since the container creates and maintains references to parts, only the container that owns a part should call the `Dispose` method on it. The container itself implements `IDisposable`, and as portion of its cleanup in `Dispose` it will call `Dispose` on all the parts that it owns. For this reason, you should always dispose the composition container when it and any parts it owns are no longer needed.

For long-lived composition containers, memory consumption by parts with a creation policy of non-shared can become a problem. These non-shared parts can be created multiple times and will not be disposed until the container itself is disposed. To deal with this, the container provides the `ReleaseExport` method. Calling this method on a non-shared export removes that export from the composition container and disposes it. Parts that are used only by the removed export, and so on down the tree, are also removed and disposed. In this way, resources can be reclaimed without disposing the composition container itself.

`IPartImportsSatisfiedNotification` contains one method named `OnImportsSatisfied`. This method is called by the composition container on any parts that implement the interface when composition has been completed and the part's imports are ready for use. Parts are created by the composition engine to fill the imports of other parts. Before the imports of a part have been set, you cannot perform any initialization that relies on or manipulates imported values in the part constructor unless those values have been specified as prerequisites by using the `ImportingConstructor` attribute. This is normally the preferred method, but in some cases, constructor injection may not be available. In those cases, initialization can be performed in `OnImportsSatisfied`, and the part should implement `IPartImportsSatisfiedNotification`.