Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the GetType() method - inherited by all types from the `Object` base class - to obtain the type of a variable:

> **Note**
>
> Make sure you add `using System;` and `using System.Reflection;` at the top of your *.cs* file.

C#

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

> **Note**
>
> The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the **IsAssembly** property. To identify a `protected internal` method, use the **IsFamilyOrAssembly**.

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute DllImportAttribute is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **Note**
>
> By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the Attributes section of the C# language specification

Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

C#

```
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an auto-implemented property.

The following example shows how to apply attributes to assemblies and modules. For more information, see Common Attributes (C#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in 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; }
```

> **Note**
>
> Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see **AttributeUsage**.

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the article Dynamically Loading and Using Types.

Related sections

For more information:

- Common Attributes (C#)
- Caller Information (C#)
- Attributes
- Reflection
- Viewing Type Information
- Reflection and Generic Types
- System.Reflection.Emit
- Retrieving Information Stored in Attributes