Contents

Attributes (C#)

1/23/2019 • 5 minutes to read • Edit Online

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*. For more information, see Reflection (C#).

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. For more information, see, Creating Custom Attributes (C#).
- 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. For more information, see Accessing Attributes by Using Reflection (C#).

Using attributes

Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is 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:

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

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

```
[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:

```
using System.Runtime.InteropServices;
```

```
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:

```
[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 Framework 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 cannot 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:

```
[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.

Attribute targets

The *target* of an attribute is the entity to which the attribute applies. 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 it precedes. 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:

```
[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

TARGET VALUE	APPLIES TO
`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#).

```
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#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to return value
[return: ValidatedContract]
int Method3() { 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 (C#).

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.

Related sections

For more information, see:

- Creating Custom Attributes (C#)
- Accessing Attributes by Using Reflection (C#)
- How to: Create a C/C++ Union by Using Attributes (C#)
- Common Attributes (C#)
- Caller Information (C#)

See also

- C# Programming Guide
- Reflection (C#)
- Attributes
- Using Attributes in C#

Creating Custom Attributes (C#)

1/23/2019 • 2 minutes to read • Edit Online

You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from Attribute, which makes identifying attribute definitions in metadata fast and easy. Suppose you want to tag types with the name of the programmer who wrote the type. You might define a custom `Author` attribute class:

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class Author : System.Attribute
{
    private string name;
    public double version;

    public Author(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

The class name is the attribute's name, `Author`. It is derived from `System.Attribute`, so it is a custom attribute class. The constructor's parameters are the custom attribute's positional parameters. In this example, `name` is a positional parameter. Any public read-write fields or properties are named parameters. In this case, `version` is the only named parameter. Note the use of the `AttributeUsage` attribute to make the `Author` attribute valid only on class and `struct` declarations.

You could use this new attribute as follows:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` has a named parameter, `AllowMultiple`, with which you can make a custom attribute single-use or multiuse. In the following code example, a multiuse attribute is created.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // multiuse attribute
]
public class Author : System.Attribute
```

In the following code example, multiple attributes of the same type are applied to a class.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

See also

- System.Reflection
- C# Programming Guide
- Writing Custom Attributes
- Reflection (C#)
- Attributes (C#)
- Accessing Attributes by Using Reflection (C#)
- AttributeUsage (C#)

AttributeUsage (C#)

1/23/2019 • 2 minutes to read • Edit Online

Determines how a custom attribute class can be used. AttributeUsageAttribute is an attribute you apply to custom attribute definitions. The `AttributeUsage` attribute enables you to control:

- Which program elements attribute may be applied to. Unless you restrict its usage, an attribute may be applied to any of the following program elements:
  - assembly
  - module
  - field
  - event
  - method
  - param
  - property
  - return
  - type
- Whether an attribute can be applied to a single program element multiple times.
- Whether attributes are inherited by derived classes.

The default settings look like the following example when applied explicitly:

```
[System.AttributeUsage(System.AttributeTargets.All,
                   AllowMultiple = false,
                   Inherited = true)]
class NewAttribute : System.Attribute { }
```

In this example, the `NewAttribute` class can be applied to any supported program element. But it can be applied only once to each entity. The attribute is inherited by derived classes when applied to a base class.

The AllowMultiple and Inherited arguments are optional, so the following code has the same effect:

```
[System.AttributeUsage(System.AttributeTargets.All)]
class NewAttribute : System.Attribute { }
```

The first AttributeUsageAttribute argument must be one or more elements of the AttributeTargets enumeration. Multiple target types can be linked together with the OR operator, like the following example shows:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Beginning in C# 7.3, attributes can be applied to either the property or the backing field for an auto-implemented property. The attribute applies to the property, unless you specify the `field` specifier on the attribute. Both are shown in the following example:

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; }

    // Attribute attached to backing field:
    [field:NewPropertyOrField]
    public string Description { get; set; }
}
```

If the AllowMultiple argument is `true`, then the resulting attribute can be applied more than once to a single entity, as shown in the following example:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

In this case, `MultiUseAttribute` can be applied repeatedly because `AllowMultiple` is set to `true`. Both formats shown for applying multiple attributes are valid.

If Inherited is `false`, then the attribute isn't inherited by classes derived from an attributed class. For example:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

In this case `NonInheritedAttribute` isn't applied to `DClass` via inheritance.

Remarks

The `AttributeUsage` attribute is a single-use attribute--it can't be applied more than once to the same class. `AttributeUsage` is an alias for AttributeUsageAttribute.

For more information, see Accessing Attributes by Using Reflection (C#).

Example

The following example demonstrates the effect of the Inherited and AllowMultiple arguments to the AttributeUsageAttribute attribute, and how the custom attributes applied to a class can be enumerated.

```
using System;

// Create some custom attributes:
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class FirstAttribute : Attribute { }

[AttributeUsage(AttributeTargets.Class)]
class SecondAttribute : Attribute { }

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class ThirdAttribute : Attribute { }

// Apply custom attributes to classes:
[First, Second]
class BaseClass { }

[Third, Third]
class DerivedClass : BaseClass { }

public class TestAttributeUsage
{
    static void Main()
    {
        BaseClass b = new BaseClass();
        DerivedClass d = new DerivedClass();

        // Display custom attributes for each class.
        Console.WriteLine("Attributes on Base Class:");
        object[] attrs = b.GetType().GetCustomAttributes(true);
        foreach (Attribute attr in attrs)
        {
            Console.WriteLine(attr);
        }

        Console.WriteLine("Attributes on Derived Class:");
        attrs = d.GetType().GetCustomAttributes(true);
        foreach (Attribute attr in attrs)
        {
            Console.WriteLine(attr);
        }
    }
}
```

Sample Output

```
Attributes on Base Class:
FirstAttribute
SecondAttribute
Attributes on Derived Class:
ThirdAttribute
ThirdAttribute
SecondAttribute
```

See also

- Attribute
- System.Reflection
- C# Programming Guide
- Attributes
- Reflection (C#)

- Attributes
- Creating Custom Attributes (C#)
- Accessing Attributes by Using Reflection (C#)

Accessing Attributes by Using Reflection (C#)

1/23/2019 • 2 minutes to read • Edit Online

The fact that you can define custom attributes and place them in your source code would be of little value without some way of retrieving that information and acting on it. By using reflection, you can retrieve the information that was defined with custom attributes. The key method is `GetCustomAttributes`, which returns an array of objects that are the run-time equivalents of the source code attributes. This method has several overloaded versions. For more information, see Attribute.

An attribute specification such as:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
```

is conceptually equivalent to this:

```
Author anonymousAuthorObject = new Author("P. Ackerman");
anonymousAuthorObject.version = 1.1;
```

However, the code is not executed until `SampleClass` is queried for attributes. Calling `GetCustomAttributes` on `SampleClass` causes an `Author` object to be constructed and initialized as above. If the class has other attributes, other attribute objects are constructed similarly. `GetCustomAttributes` then returns the `Author` object and any other attribute objects in an array. You can then iterate over this array, determine what attributes were applied based on the type of each array element, and extract information from the attribute objects.

Example

Here is a complete example. A custom attribute is defined, applied to several entities, and retrieved via reflection.

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
        return 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
{
    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 {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("   {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}
/* 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
*/
```

See also

- System.Reflection
- Attribute
- C# Programming Guide
- Retrieving Information Stored in Attributes
- Reflection (C#)
- Attributes (C#)
- Creating Custom Attributes (C#)

How to: Create a C/C++ Union by Using Attributes (C#)

1/23/2019 • 2 minutes to read • Edit Online

By using attributes you can customize how structs are laid out in memory. For example, you can create what is known as a union in C/C++ by using the `StructLayout(LayoutKind.Explicit)` and `FieldOffset` attributes.

Example

In this code segment, all of the fields of `TestUnion` start at the same location in memory.

```
// Add a using directive for System.Runtime.InteropServices.

       [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;
       }
```

Example

The following is another example where fields start at different explicitly set locations.

```
// Add a using directive for System.Runtime.InteropServices.

       [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;
       }
```

The two integer fields, `i1` and `i2`, share the same memory locations as `lg`. This sort of control over struct layout is useful when using platform invocation.

See also

- System.Reflection
- Attribute
- C# Programming Guide
- Attributes
- Reflection (C#)
- Attributes (C#)
- Creating Custom Attributes (C#)
- Accessing Attributes by Using Reflection (C#)

Common Attributes (C#)

1/23/2019 • 6 minutes to read • Edit Online

This topic describes the attributes that are most commonly used in C# programs.

- Global Attributes
- Obsolete Attribute
- Conditional Attribute
- Caller Info Attributes

Global Attributes

Most attributes are applied to specific language elements such as classes or methods; however, some attributes are global—they apply to an entire assembly or module. For example, the AssemblyVersionAttribute attribute can be used to embed version information into an assembly, like this:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Global attributes appear in the source code after any top-level `using` directives and before any type, module, or namespace declarations. Global attributes can appear in multiple source files, but the files must be compiled in a single compilation pass. In C# projects, global attributes are put in the AssemblyInfo.cs file.

Assembly attributes are values that provide information about an assembly. They fall into the following categories:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes

Assembly Identity Attributes

Three attributes (with a strong name, if applicable) determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when you reference it in code. You can set an assembly's version and culture using attributes. However, the name value is set by the compiler, the Visual Studio IDE in the Assembly Information Dialog Box, or the Assembly Linker (Al.exe) when the assembly is created, based on the file that contains the assembly manifest. The AssemblyFlagsAttribute attribute specifies whether multiple copies of the assembly can coexist.

The following table shows the identity attributes.

ATTRIBUTE	PURPOSE
AssemblyName	Fully describes the identity of an assembly.
AssemblyVersionAttribute	Specifies the version of an assembly.
AssemblyCultureAttribute	Specifies which culture the assembly supports.

ATTRIBUTE	PURPOSE
AssemblyFlagsAttribute	Specifies whether an assembly supports side-by-side execution on the same computer, in the same process, or in the same application domain.

Informational Attributes

You can use informational attributes to provide additional company or product information for an assembly. The following table shows the informational attributes defined in the System.Reflection namespace.

ATTRIBUTE	PURPOSE
AssemblyProductAttribute	Defines a custom attribute that specifies a product name for an assembly manifest.
AssemblyTrademarkAttribute	Defines a custom attribute that specifies a trademark for an assembly manifest.
AssemblyInformationalVersionAttribute	Defines a custom attribute that specifies an informational version for an assembly manifest.
AssemblyCompanyAttribute	Defines a custom attribute that specifies a company name for an assembly manifest.
AssemblyCopyrightAttribute	Defines a custom attribute that specifies a copyright for an assembly manifest.
AssemblyFileVersionAttribute	Instructs the compiler to use a specific version number for the Win32 file version resource.
CLSCompliantAttribute	Indicates whether the assembly is compliant with the Common Language Specification (CLS).

Assembly Manifest Attributes

You can use assembly manifest attributes to provide information in the assembly manifest. This includes title, description, default alias, and configuration. The following table shows the assembly manifest attributes defined in the System.Reflection namespace.

ATTRIBUTE	PURPOSE
AssemblyTitleAttribute	Defines a custom attribute that specifies an assembly title for an assembly manifest.
AssemblyDescriptionAttribute	Defines a custom attribute that specifies an assembly description for an assembly manifest.
AssemblyConfigurationAttribute	Defines a custom attribute that specifies an assembly configuration (such as retail or debug) for an assembly manifest.
AssemblyDefaultAliasAttribute	Defines a friendly default alias for an assembly manifest

Obsolete Attribute

The `Obsolete` attribute marks a program entity as one that is no longer recommended for use. Each use of an entity marked obsolete will subsequently generate a warning or an error, depending on how the attribute is configured. For example:

```
[System.Obsolete("use class B")]
class A
{
    public void Method() { }
}
class B
{
    [System.Obsolete("use NewMethod", true)]
    public void OldMethod() { }
    public void NewMethod() { }
}
```

In this example the `Obsolete` attribute is applied to class `A` and to method `B.OldMethod`. Because the second argument of the attribute constructor applied to `B.OldMethod` is set to `true`, this method will cause a compiler error, whereas using class `A` will just produce a warning. Calling `B.NewMethod`, however, produces no warning or error.

The string provided as the first argument to attribute constructor will be displayed as part of the warning or error. For example, when you use it with the previous definitions, the following code generates two warnings and one error:

```
// Generates 2 warnings:
// A a = new A();

// Generate no errors or warnings:
B b = new B();
b.NewMethod();

// Generates an error, terminating compilation:
// b.OldMethod();
```

Two warnings for class `A` are generated: one for the declaration of the class reference, and one for the class constructor.

The `Obsolete` attribute can be used without arguments, but including an explanation of why the item is obsolete and what to use instead is recommended.

The `Obsolete` attribute is a single-use attribute and can be applied to any entity that allows attributes. `Obsolete` is an alias for ObsoleteAttribute.

Conditional Attribute

The `Conditional` attribute makes the execution of a method dependent on a preprocessing identifier. The `Conditional` attribute is an alias for ConditionalAttribute, and can be applied to a method or an attribute class.

In this example, `Conditional` is applied to a method to enable or disable the display of program-specific diagnostic information:

```
#define TRACE_ON
using System;
using System.Diagnostics;

public class Trace
{
    [Conditional("TRACE_ON")]
    public static void Msg(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class ProgramClass
{
    static void Main()
    {
        Trace.Msg("Now in Main...");
        Console.WriteLine("Done.");
    }
}
```

If the `TRACE_ON` identifier is not defined, no trace output will be displayed.

The `Conditional` attribute is often used with the `DEBUG` identifier to enable trace and logging features for debug builds but not in release builds, like this:

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

When a method marked as conditional is called, the presence or absence of the specified preprocessing symbol determines whether the call is included or omitted. If the symbol is defined, the call is included; otherwise, the call is omitted. Using `Conditional` is a cleaner, more elegant, and less error-prone alternative to enclosing methods inside `#if…#endif` blocks, like this:

```
#if DEBUG
    void ConditionalMethod()
    {
    }
#endif
```

A conditional method must be a method in a class or struct declaration and must not have a return value.

Using Multiple Identifiers

If a method has multiple `Conditional` attributes, a call to the method is included if at least one of the conditional symbols is defined (in other words, the symbols are logically linked together by using the OR operator). In this example, the presence of either `A` or `B` will result in a method call:

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

To achieve the effect of logically linking symbols by using the AND operator, you can define serial conditional methods. For example, the second method below will execute only if both `A` and `B` are defined:

```
[Conditional("A")]
static void DoIfA()
{
    DoIfAandB();
}

[Conditional("B")]
static void DoIfAandB()
{
    // Code to execute when both A and B are defined...
}
```

Using Conditional with Attribute Classes

The `Conditional` attribute can also be applied to an attribute class definition. In this example, the custom attribute `Documentation` will only add information to the metadata if DEBUG is defined.

```
[Conditional("DEBUG")]
public class Documentation : System.Attribute
{
    string text;

    public Documentation(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Caller Info Attributes

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain the file path of the source code, the line number in the source code, and the member name of the caller.

To obtain member caller information, you use attributes that are applied to optional parameters. Each optional parameter specifies a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePathAttribute	Full path of the source file that contains the caller. This is the path at compile time.	`String`
CallerLineNumberAttribute	Line number in the source file from which the method is called.	`Integer`
CallerMemberNameAttribute	Method name or property name of the caller. For more information, see Caller Information (C#).	`String`

For more information about the Caller Info attributes, see Caller Information (C#).

See also

- System.Reflection
- Attribute
- C# Programming Guide
- Attributes
- Reflection (C#)
- Accessing Attributes by Using Reflection (C#)