A Programmer's Introduction to C#, Second Edition

ERIC GUNNERSON

Apress™

A Programmer's Introduction to C#, Second Edition

ISBN (pbk): 1-893115-62-3

Printed and bound in the United States of America 345678910

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson, Jason Gilmore

Technical Reviewers: David Staheli, Shawn Vita, Gus Perez, Jerry Higgins, Brenton Webster

Copy Editor: Kim Wimpsett

Managing Editor: Grace Wong

Production Editor: Janet Vail

Page Compositor and Soap Bubble Artist: Susan Glinert

Artists: Karl Miyajima and Tony Jonick

Indexer: Valerie Perry

Cover Designer: Karl Miyajima

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010

and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER; orders@springer-ny.com; http://www.springer-ny.com

Outside the United States, contact orders@springer.de; http://www.springer.de; fax +49 6221 345229

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710

Phone: 510-549-5937; Fax: 510-549-5939; info@apress.com; http://www.apress.com

CHAPTER 21

Attributes

IN MOST PROGRAMMING LANGUAGES, some information is expressed through declaration, and other information is expressed through code. For example, in the following class member declaration

```
public int Test;
```

the compiler and runtime will reserve space for an integer variable and set its protection so that it is visible everywhere. This is an example of declarative information; it's nice because of the economy of expression and because the compiler handles the details for us.

Typically, the types of declarative information are predefined by the language designer and can't be extended by users of the language. A user who wants to associate a specific database field with a field of a class, for example, must invent a way of expressing that relationship in the language, a way of storing the relationship, and a way of accessing the information at runtime. In a language like C++, a macro might be defined that stores the information in a field that is part of the object. Such schemes work, but they're error-prone and not generalized. They're also ugly.

The .NET Runtime supports attributes, which are merely annotations that are placed on elements of source code, such as classes, members, parameters, etc. Attributes can be used to change the behavior of the runtime, provide transaction information about an object, or convey organizational information to a designer. The attribute information is stored with the metadata of the element and can be easily retrieved at runtime through a process known as reflection.

C# uses a conditional attribute to control when member functions are called. A use for the conditional attribute would look like this:

```
using System.Diagnostics;
class Test
{
    [Conditional("DEBUG")]
    public void Validate()
    {
    }
}
```

Most programmers will use predefined attributes much more often than writing an attribute class.

Using Attributes

Suppose that for a project that a group was doing, it was important to keep track of the code reviews that had been performed on the classes so that it could be determined when code reviews were finished. The code review information could be stored in a database, which would allow easy queries about status, or it could be stored in comments, which would make it easy to look at the code and the information at the same time.

Or an attribute could be used, which would enable both kinds of access.

To do that, an attribute class is needed. An attribute class defines the name of an attribute, how it can be created, and the information that will be stored. The gritty details of defining attribute classes will be covered in the section entitled "An Attribute of Your Own."

The attribute class will look like this:

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class CodeReviewAttribute: System.Attribute
{
    public CodeReviewAttribute(string reviewer, string date)
    {
        this.reviewer = reviewer;
        this.date = date;
    }
    public string Comment
    {
        get
        {
            return(comment);
        }
        set
        {
            comment = value;
        }
    }
    public string Date
    {
        get
        {
            return(date);
        }
    }
```

```
    public string Reviewer
    {
        get
        {
            return(reviewer);
        }
    }
    string reviewer;
    string date;
    string comment;
}
[CodeReview("Eric", "01-12-2000", Comment="Bitchin' Code")]
class Complex
{
}
```

The `AttributeUsage` attribute before the class specifies that this attribute can only be placed on classes. When an attribute is used on a program element, the compiler checks to see whether the use of that attribute on that program element is allowed.

The naming convention for attributes is to append `Attribute` to the end of the class name. This makes it easier to tell which classes are attribute classes and which classes are normal classes. All attributes must derive from `System.Attribute`.

The class defines a single constructor that takes a reviewer and a date as parameters, and it also has the public string property `Comment`.

When the compiler comes to the attribute use on class `Complex`, it first looks for a class derived from `Attribute` named `CodeReview`. It doesn't find one, so it next looks for a class named `CodeReviewAttribute`, which it finds.

Next, it checks to see whether the attribute is allowed for this usage (on a class).

Then, it checks to see if there is a constructor that matches the parameters we've specified in the attribute use. If it finds one, an instance of the object is created—the constructor is called with the specified values.

If there are named parameters, it matches the name of the parameter with a field or property in the attribute class, and then it sets the field or property to the specified value.

After this is done, the current state of the attribute class is saved to the metadata for the program element for which it was specified.

At least, that's what happens *logically*. In actuality, it only *looks* like it happens that way; see the "Attribute Pickling" sidebar for a description of how it is implemented.

Attribute Pickling

There are a few reasons why it doesn't really work the way it's described, and they're related to performance. For the compiler to actually create the attribute object, the .NET Runtime environment would have to be running, so every compilation would have to start up the environment, and every compiler would have to run as a managed executable.

Additionally, the object creation isn't really required, since we're just going to store the information away.

The compiler therefore validates that it *could* create the object, call the constructor, and set the values for any named parameters. The attribute parameters are then pickled into a chunk of binary information, which is tucked away with the metadata of the object.

A Few More Details

Some attributes can only be used once on a given element. Others, known as multi-use attributes, can be used more than once. This might be used, for example, to apply several different security attributes to a single class. The documentation on the attribute will describe whether an attribute is single-use or multi-use.

In most cases, it's clear that the attribute applies to a specific program element. However, consider the following case:

```
using System.Runtime.InteropServices;
class Test
{
    [return: MarshalAs(UnmanagedType.LPWStr)]
    public static extern string GetMessage();
}
```

In most cases, an attribute in that position would apply to the member function, but this attribute is really related to the return type. How can the compiler tell the difference?

There are several situations in which this can happen:

- Method vs. return value
- Event vs. field or property
- Delegate vs. return value
- Property vs. accessor vs. return value of getter vs. value parameter of setter

For each of these situations, there is a case that is much more common than the other case, and it becomes the default case. To specify an attribute for the non-default case, the element the attribute applies to must be specified:

```
using System.Runtime.InteropServices;
class Test
{
    [return: MarshalAs(UnmanagedType.LPWStr)]
    public static extern string GetMessage();
}
```

The `return:` indicates that this attribute should be applied to the return value.

The element may be specified even if there is no ambiguity. The identifiers are as follows:

SPECIFIER	DESCRIPTION
`assembly`	Attribute is on the assembly
`module`	Attribute is on the module
`type`	Attribute is on a class or struct
`method`	Attribute is on a method
`property`	Attribute is on a property
`event`	Attribute is on an event
`field`	Attribute is on a field
`param`	Attribute is on a parameter
`returnValue`	Attribute is on the return value

Attributes that are applied to assemblies or modules must occur after any `using` clauses and before any code.

```
using System;
[assembly:CLSCompliant(true)]

class Test
{
    Test() {}
}
```

This example applies the `ClsCompliant` attribute to the entire assembly. All assembly-level attributes declared in any file that is in the assembly are grouped together and attached to the assembly.

To use a predefined attribute, start by finding the constructor that best matches the information to be conveyed. Next, write the attribute, passing parameters to the constructor. Finally, use the named parameter syntax to pass additional information that wasn't part of the constructor parameters.

For more examples of attribute use, look at Chapter 31, "Interop."

An Attribute of Your Own

To define attribute classes and reflect on them at runtime, there are a few more issues to consider. This section will discuss some things to consider when designing an attribute.

There are two major things to determine when writing an attribute. The first is the program elements that the attribute may be applied to, and the second is the information that will be stored by the attribute.

Attribute Usage

Placing the `AttributeUsage` attribute on an attribute class controls where the attribute can be used. The possible values for the attribute are listed in the `AttributeTargets` enumerator and are as follows:

VALUE	MEANING
`Assembly`	The program assembly
`Module`	The current program file
`Class`	A class

(Continued)

VALUE	MEANING
`Struct`	A struct
`Enum`	An enumerator
`Constructor`	A constructor
`Method`	A method (member function)
`Property`	A property
`Field`	A field
`Event`	An event
`Interface`	An interface
`Parameter`	A method parameter
`ReturnValue`	The method return value
`Delegate`	A delegate
`All`	Anywhere
`ClassMembers`	Class, Struct, Enum, Constructor, Method, Property, Field, Event, Delegate, Interface

As part of the `AttributeUsage` attribute, one of these can be specified or a list of them can be ORed together.

The `AttributeUsage` attribute is also used to specify whether an attribute is single-use or multi-use. This is done with the named parameter `AllowMultiple`. Such an attribute would look like this:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Event,
                AllowMultiple = true)]
```

Attribute Parameters

The information the attribute will store should be divided into two groups: the information that is required for every use, and the optional items.

The information that is required for every use should be obtained via the constructor for the attribute class. This forces the user to specify all the parameters when they use the attribute.

Optional items should be implemented as named parameters, which allows the user to specify whichever optional items are appropriate.

If an attribute has several different ways in which it can be created, with different required information, separate constructors can be declared for each usage. Don't use separate constructors as an alternative to optional items.

Attribute Parameter Types

The attribute pickling format only supports a subset of all the .NET Runtime types, and therefore, only some types can be used as attribute parameters. The types allowed are the following:

- `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `string`
- `object`
- `System.Type`
- An `enum` that has public accessibility (not nested inside something non-public)
- A one-dimensional array of one of the above types

Reflecting on Attributes

Once attributes are defined on some code, it's useful to be able to find the attribute values. This is done through reflection.

The following code shows an attribute class, the application of the attribute to a class, and the reflection on the class to retrieve the attribute.

```
using System;
using System.Reflection;
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class CodeReviewAttribute: System.Attribute
{
    public CodeReviewAttribute(string reviewer, string date)
    {
        this.reviewer = reviewer;
        this.date = date;
    }
```

```
    public string Comment
    {
        get
        {
            return(comment);
        }
        set
        {
            comment = value;
        }
    }
    public string Date
    {
        get
        {
            return(date);
        }
    }
    public string Reviewer
    {
        get
        {
            return(reviewer);
        }
    }
    string reviewer;
    string date;
    string comment;
}
[CodeReview("Eric", "01-12-2000", Comment="Bitchin' Code")]
[CodeReview("Gurn", "01-01-2000", Comment="Revisit this section")]
class Complex
{
}
```

```
class Test
{
   public static void Main()
    {
        Type type = typeof(Complex);
        foreach (CodeReviewAttribute att in
                    type.GetCustomAttributes(typeof(CodeReviewAttribute), false))
        {
            CodeReviewAttribute    att = (CodeReviewAttribute) atts[0];
            Console.WriteLine("Reviewer: {0}", att.Reviewer);
            Console.WriteLine("Date: {0}", att.Date);
            Console.WriteLine("Comment: {0}", att.Comment);
        }
    }
}
```

The `Main()` function first gets the type object associated with the type `Complex`. It then iterates over all the `CodeReviewAttribute` attributes attached to the type and writes the values out.

Alternately, the code could get all the attributes by omitting the type in the call to `GetCustomAttributes`:

```
        foreach (object o in type.GetCustomAttributes(false))
        {
            CodeReviewAttribute att = o as CodeReviewAttribute;
            if (att != null)
            {
                // write values here…
            }
        }
```

This example produces the following output:

```
Reviewer: Eric
Date: 01-12-2000
Comment: Bitchin' Code
Reviewer: Gurn
Date: 01-01-2000
Comment: Revisit this section
```

The `false` value in the call to `GetCustomAttributes` tells the runtime to ignore any inherited attributes. In this case, that would ignore any attributes on the base class of `Complex`.

In the example, the type object for the `Complex` type is obtained using `typeof`. It can also be obtained in the following manner:

```
Complex c = new Complex();
Type t = c.GetType();
Type t2 = Type.GetType("Complex");
```

NOTE *The "`CustomAttributes`" in the preceding example refers to attributes stored in the section of metadata reserved for attributes. Some .NET Runtime attributes are not stored as custom attributes on the object, but are instead stored as metadata bits on the object. Runtime reflection does not support viewing these attributes through reflection. This restriction may be addressed in future versions of the runtime.*