The C# Station Tutorial

by Joe Mayo, 12/15/02, updated 3/12/03

Lesson 16: Using Attributes

This lesson explains how to use C# attributes. Our objectives are as follows:

- Understand what attributes are and why they're used
- Apply various attributes with multiple or no parameters
- Use assembly, type member, and type level attributes

Why Attributes?

Attributes are elements that allow you to add declarative information to your programs. This declarative information is used for various purposes during runtime and can be used at design time by application development tools. For example, there are attributes such as DllImportAttribute that allow a program to communicate with the Win32 libraries. Another attribute, ObsoleteAttribute, causes a compile-time warning to appear, letting the developer know that a method should no longer be used. When building Windows Forms applications, there are several attributes that allow visual components to be drag-n-dropped onto a visual form builder and have their information appear in the properties grid. Attributes are also used extensively in securing .NET assemblies, forcing calling code to be evaluated against pre-defined security constraints. These are just a few descriptions of how attributes are used in C# programs.

The reason attributes are necessary is because many of the services they provide would be very difficult to accomplish with normal code. You see, attributes add what is called metadata to your programs. When your C# program is compiled, it creates a file called an assembly, which is normally an executable or DLL library. Assemblies are self-describing because they have metadata written to them when they are compiled. Via a process known as reflection, a program's attributes can be retrieved from its assembly metadata. Attributes are classes that can be written in C# and used to decorate your code with declarative information. This is a very powerful concept because it means that your can extend your language by creating customized declarative syntax with attributes.

This tutorial will show how to use pre-existing attributes in C# programs. Understanding the concepts and how to use a few attributes, will help in finding the multitude of other pre-existing attributes in the .NET class libraries and use them also.

Attribute Basics

Attributes are generally applied physically in front of type and type member declarations. They're declared with square brackets, "[" and "]", surrounding the attribute such as the following *ObsoleteAttribute* attribute:

```
[ObsoleteAttribute]
```

The "Attribute" part of the attribute name is optional. So the following is equivalent to the attribute above:

```
[Obsolete]
```

You'll notice that the attribute is declared with only the name of the attribute, surrounded by square brackets. Many attributes have parameter lists, that allow inclusion of additional information that customizes a program even further. Listing 16.1 shows various ways of how to use the *ObsoleteAttribute* attribute.

Listing 16-1. How to Use Attributes: BasicAttributeDemo.cs

```
using System;

class BasicAttributeDemo
{
   [Obsolete]
   public void MyFirstDeprecatedMethod()
   {
      Console.WriteLine("Called MyFirstDeprecatedMethod().");
   }

   [ObsoleteAttribute]
   public void MySecondDeprecatedMethod()
   {
      Console.WriteLine("Called MySecondDeprecatedMethod().");
   }

   [Obsolete("You shouldn't use this method anymore.")]
   public void MyThirdDeprecatedMethod()
   {
      Console.WriteLine("Called MyThirdDeprecatedMethod().");
   }

   // make the program thread safe for COM
   [STAThread]
   static void Main(string[] args)
   {
      BasicAttributeDemo attrDemo = new BasicAttributeDemo();

      attrDemo.MyFirstDeprecatedMethod();
      attrDemo.MySecondDeprecatedMethod();
      attrDemo.MyThirdDeprecatedMethod();
   }
}
```

Get Setup Instructions For How to Run this Program

Examining the code in listing 16-1 reveals that the *ObsoleteAttribute* attribute was used a few different ways. The first usage appeared on the *MyFirstDeprecatedMethod()* method and the second usage appeared in the *MySecondDeprecatedMethod()* method as follows:

```
[Obsolete]
public void MyFirstDeprecatedMethod()
...
[ObsoleteAttribute]
public void MySecondDeprecatedMethod()
...
```

The only difference between the two attributes is that *MySecondDeprecatedMethod()* method contains the "Attribute" in the attribute declaration. The results of both attributes are exactly the same. Attributes may also have parameters, as shown in the following declaration:

```
[Obsolete("You shouldn't use this method anymore.")]
public void MyThirdDeprecatedMethod()
...
```

This adds customized behavior to the *ObsoleteAttribute* attribute which produces different results from the other *ObsoleteAttribute* attribute declarations. The results of all three *ObsoleteAttribute* attributes are shown below. These are the warnings that are emitted by the C# compiler when the program is compiled:

```
>csc BasicAttributeDemo.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.2292.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

BasicAttributeDemo.cs(29,3): warning CS0612:
'BasicAttributeDemo.MyFirstDeprecatedMethod()' is obsolete
```

```
BasicAttributeDemo.cs(30,3): warning CS0612:
'BasicAttributeDemo.MySecondDeprecatedMethod()' is obsolete
BasicAttributeDemo.cs(31,3): warning CS0618:
'BasicAttributeDemo.MyThirdDeprecatedMethod()' is obsolete: 'You
shouldn't use this method anymore.'
```

As you can see, the *ObsoleteAttribute* attribute caused the *MyThirdDeprecatedMethod()* method to emit the message that was a parameter to the *ObsoleteAttribute* attribute of that method in the code. The other attributes simply emitted standard warnings.

Listing 16-1 also contains another attribute you're likely to see, the *STAThreadAttribute* attribute. You'll often see this attribute applied to the *Main()* method, indicating that this C# program should communicate with unmanaged COM code using the Single Threading Apartment . It is generally safe to use this attribute all the time because you never know when a 3rd party library you're using is going to be communicating with COM. The following excerpt shows how to use the *STAThreadAttribute* attribute:

```
[STAThread]
static void Main(string[] args)
...
```

Attribute Parameters

Attributes often have parameters that enable customization. There are two types of parameters that can be used on attributes, positional and named. Positional parameters are used when the attribute creator wishes the parameters to be required. However, this is not a hard and fast rule because the *ObsoleteAttribute* attribute has a second positional parameter named error of type *int* that we can omit as demonstrated in Listing 16-1. That attribute could have been written with the second positional parameter to force a compiler error instead of just a warning as follows:

```
[Obsolete("You shouldn't use this method anymore.", true)]
public void MyThirdDeprecatedMethod()
...
```

The difference between positional parameters and named parameters are that named parameters are specified with the name of the parameter and are always optional. The *DllImportAttribute* attribute is one you are likely to see that has both positional and named attributes (Listing 16-2).

Listing 16-2. Using Positional and Named Attribute Parameters: AttributeParamsDemo.cs

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

class AttributeParamsDemo
{
   [DllImport("User32.dll", EntryPoint="MessageBox")]
   static extern int MessageDialog(int hWnd, string msg, string caption, int msgType);

   [STAThread]
   static void Main(string[] args)
   {
      MessageDialog(0, "MessageDialog Called!", "DllImport Demo", 0);
   }
}
```

Get Setup Instructions For How to Run this Program

The *DllImportAttribute* attribute in Listing 16-2 has a one positional parameter, *"User32.dll"*, and one named parameter, *EntryPoint="MessageBox"*. Positional parameters are always specified before any named parameters. When there are named parameters, they may appear in any order. This is because they are marked with the parameter name like in the *DllImportAttribute* attribute *EntryPoint="MessageBox"*. Since the

In the *DllImportAttribute* attribute, *EntryPoint="MessageBox"*. Since the purpose of this lesson is to explain how to use attributes in general, I won't go into the details of the *DllImportAttribute* attribute, which has extra parameters that require knowledge of Win32 and other details that don't pertain to this lesson. Many other attributes can be used with both positional and named parameters.

Attribute Targets

The attributes shown so far have been applied to methods, but there are many other C# language elements that you can use attributes with. Table 16-1 outlines the C# language elements that attributes may be applied to. They are formally called attribute "targets".

Attribute Target	Can be Applied To
all	everything
assembly	entire assembly
class	classes
constructor	constructors
delegate	delegates
enum	enumerations
event	events
field	fields
interface	interfaces
method	methods
module	modules (compiled code that can be part of an assembly)
parameter	parameters
property	properties
returnvalue	return values
struct	structures

Whenever there is ambiguity in how an attribute is applied, you can add a target specification to ensure the right language element is decorated properly. An attribute that helps ensure assemblies adhere to the Common Language Specification (CLS) is the *CLSCompliantAttribute* attribute. The CLS is the set of standards that enable different .NET languages to communicate. Attribute targets are specified by prefixing the attribute name with the target and separating it with a colon (*:*). Listing 16-3 shows how to use the *CLSCompliantAttribute* attribute and apply it to the entire assembly.

Listing 16-3. Using Positional and Named Attribute Parameters: AttributeTargetDemo.cs

```
using System;

[assembly:CLSCompliant(true)]

public class AttributeTargetDemo
{
   public void NonClsCompliantMethod(uint nclsParam)
   {
      Console.WriteLine("Called NonClsCompliantMethod().");
   }

   [STAThread]
   static void Main(string[] args)
   {
      uint myUint = 0;

      AttributeTargetDemo tgtDemo = new AttributeTargetDemo();

      tgtDemo.NonClsCompliantMethod(myUint);
   }
}
```

The code in Listing 16-3 won't compile because the *uint* type parameter

declared on the *NonClsCompliantMethod()* method. If you change the *CLSCompliantAttribute* attribute to false or change the type of the *NonClsCompliantMethod()* method to a CLS compliant type, such as *int*, the program will compile.

The point about Listing 16-3 is that the *CLSCompliantAttribute* attribute is decorated with an attribute target of *"assembly"*. This causes all members of this assembly to be evaluated according to the *CLSCompliantAttribute* attribute setting. To limit the scope of the *CLSCompliantAttribute*, apply it to either the *AttributeTargetDemo* class or *NonClsCompliantMethod()* method directly.

Summary

Attributes are C# language elements that decorate program elements with additional metadata that describes the program. This metadata is then evaluated at different places, such as runtime or design time for various purposes. The examples in this lesson showed how the *ObsoleteAttribute* attribute could be used to generate compile time warnings for deprecated code. Through applying the *DllImportAttribute* attribute, you could see how to apply both positional and named parameters to an attribute. Attributes may also be used to decorate various different types of program elements with a target descriptor. The example applied the *CLSCompliantAttribute* attribute to an entire assembly. However, it could have also been applied to different program elements with applicable target descriptors to limit its scope.

I invite you to return for Lesson 17: Enums.