MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
NOV 20, 2015

How to work with attributes in C#

Take advantage of attributes to embed metadata information to your assemblies and decorate the business objects in your application

Attributes are a powerful feature in the C# programming language that can add metadata information to your assemblies.

An attribute is actually an object that is associated with any of these elements: Assembly, Class, Method, Delegate, Enum, Event, Field, Interface, Property and Struct. They can be used to associate declarative information -- you can retrieve such information at runtime at a later point of time if need be using reflection. In other words, you can use attributes to inject additional information to the assemblies that can be queried at runtime if needed using reflection. An attribute comprises of its name and optionally, a list of parameters. The attribute name corresponds to the attribute class.

You can take advantage of attributes to validate the business objects in your application. There are two types of attributes -- intrinsic attributes and custom attributes. While the former is available as part of the .Net framework, the latter can be implemented by deriving a class from the System.Attribute class. The MSDN states: "An attribute is a piece of additional declarative information that is specified for a declaration."

[Discover the Bossie Award winners: 2018's best open source software for enterprise for software development, machine learning, cloud computing, and data storage and analytics.]

Let's now get into some code. The Obsolete attribute can be used to denote a method as obsolete -- one that shouldn't be used anymore as it is no longer needed or may have some other alternative. The following code snippet illustrates how you can use the Obsolete attribute on top of a method declaration.

```
[Obsolete("This method is obsolete...")]
        public static void DoSomeWork()
        {
            //Some code
        }
```


If you use this method in your code and compile your program, you would see a warning displayed in the output window of the Visual Studio IDE. So, you can ignore this warning if you want to. Now, what if you want your developers no to use this method at all? Well, you can then use the second parameter (it's optional though) while declaring the Obsolete attribute. Here's the modified version of the DoSomeWork() method. Notice the usage of the Boolean parameter this time.

```
[Obsolete("This method is obsolete...", true)]
        public static void DoSomeWork()
        {
                    //Some code
}
```

When you pass "true" as the second optional parameter this time and compile your program, the code wouldn't compile at all. That's what you wanted to do, isn't it?

Custom attributes

In this section we will explore how we can implement custom attributes. Custom attributes inherit the System.Attribute class. So, to implement a custom attribute class, create a new class and derive it from System.Attribute class as shown below.

```
using System;

public class CustomAttribute : Attribute

{

}
```


To control the usage of custom attributes, you can take advantage of the AttributeUsage class. This class contains properties like, ValidOn, AllowMultiple and Inherited which can be used to control the usage of your custom attribute.

The following snippet of code illustrates a modified version of our custom attribute class. Note the usage of a constructor that accepts a string as an argument and assigns it to the private string member of the class. This is just for illustration purposes only.

```
[AttributeUsage(AttributeTargets.All)]

    public class CustomAttribute : Attribute

    {

        private string text;

        public CustomAttribute(string text)

        {

            this.Text = text;

        }

        public string Text

        {
```

```
            get
            {
                return this.text;
            }
            set
            {
                this.text = value;
            }
        }
    }
```


You can also specify the attribute targets that your custom attribute should be applied to. Here's how you can do it.

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
  public class CustomAttribute : Attribute
    {
        private string text;
        public CustomAttribute(string text)
```

```
        {
text;
        }
        public string Text
        {
            get
            {
                return this.text;
            }
            set
            {
                this.text = value;
            }
        }
    }
```


You can now use reflection to display all the attributes that are applied to a particular object using the following code snippet.

```
MemberInfo memberInfo = typeof(CustomAttribute);

object[] attributes = memberInfo.GetCustomAttributes(true);

for (int i = 0, j = attributes.Length; i < j; i++)

  {

     Console.WriteLine(attributes[i]);
```

class on which we would apply our custom attribute.

```
[CustomAttribute("Hello World...")]
public class SomeClass
{

}
```


Note how the custom attribute has been used and a text passed as argument to it. The following code snippet illustrates how you can print the content of the Text property.

```
MemberInfo memberInfo = typeof(SomeClass);
object[] attributes = memberInfo.GetCustomAttributes(true);

foreach (object attribute in attributes)
{
CustomAttribute customAttribute = attribute as CustomAttribute;

if (customAttribute != null)
Console.WriteLine("Text = {0}", customAttribute.Text);
else
Console.WriteLine();
}
```

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles with more than 16 years of experience in IT.

Follow

Want more InfoWorld?