C# - ATTRIBUTES

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program. You can add declarative information to a program by using an attribute. A declarative tag is depicted by square [] brackets placed above the element it is used for.

Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. The .Net Framework provides two types of attributes: *the pre-defined* attributes and *custom built* attributes.

Specifying an Attribute

Syntax for specifying an attribute is as follows –

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

Name of the attribute and its values are specified within the square brackets, before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

Predefined Attributes

The .Net Framework provides three pre-defined attributes –

- AttributeUsage
- Conditional
- Obsolete

AttributeUsage

The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

Syntax for specifying this attribute is as follows –

```
[AttributeUsage (
   validon,
   AllowMultiple = allowmultiple,
   Inherited = inherited
)]
```

Where,

- The parameter validon specifies the language elements on which the attribute can be placed. It is a combination of the value of an enumerator *AttributeTargets*. The default value is *AttributeTargets.All*.

- The parameter *allowmultiple optional* provides value for the *AllowMultiple* property of this attribute, a Boolean value. If this is true, the attribute is multiuse. The default is false $single - use$.
- The parameter inherited *optional* provides value for the *Inherited* property of this attribute, a Boolean value. If it is true, the attribute is inherited by derived classes. The default value is false $notinherited$.

For example,

```
[AttributeUsage(
   AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]
```

Conditional

This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

It causes conditional compilation of method calls, depending on the specified value such as **Debug** or **Trace**. For example, it displays the values of the variables while debugging a code.

Syntax for specifying this attribute is as follows –

```
[Conditional(
   conditionalSymbol
)]
```

For example,

```
[Conditional("DEBUG")]
```

The following example demonstrates the attribute –

Live Demo

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

public class Myclass {
   [Conditional("DEBUG")]

   public static void Message(string msg) {
      Console.WriteLine(msg);
   }
}
class Test {
   static void function1() {
      Myclass.Message("In Function 1.");
      function2();
   }
   static void function2() {
```

```
      Myclass.Message("In Function 2.");
   }
   public static void Main() {
      Myclass.Message("In Main function.");
      function1();
      Console.ReadKey();
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
In Main function
In Function 1
In Function 2
```

Obsolete

This predefined attribute marks a program entity that should not be used. It enables you to inform the compiler to discard a particular target element. For example, when a new method is being used in a class and if you still want to retain the old method in the class, you may mark it as obsolete by displaying a message the new method should be used, instead of the old method.

Syntax for specifying this attribute is as follows –

```
[Obsolete (
   message
)]

[Obsolete (
   message,
   iserror
)]
```

Where,

- The parameter *message*, is a string describing the reason why the item is obsolete and what to use instead.
- The parameter *iserror*, is a Boolean value. If its value is true, the compiler should treat the use of the item as an error. Default value is false *compilergeneratesawarning*.

The following program demonstrates this –

```
using System;

public class MyClass {
   [Obsolete("Don't use OldMethod, use NewMethod instead", true)]

   static void OldMethod() {
      Console.WriteLine("It is the old method");
   }
   static void NewMethod() {
      Console.WriteLine("It is the new method");
   }
   public static void Main() {
```

```
      OldMethod();
   }
}
```

When you try to compile the program, the compiler gives an error message stating –

```
Don't use OldMethod, use NewMethod instead
```

Creating Custom Attributes

The .Net Framework allows creation of custom attributes that can be used to store declarative information and can be retrieved at run-time. This information can be related to any target element depending upon the design criteria and application need.

Creating and using custom attributes involve four steps –

- Declaring a custom attribute
- Constructing the custom attribute
- Apply the custom attribute on a target program element
- Accessing Attributes Through Reflection

The Last step involves writing a simple program to read through the metadata to find various notations. Metadata is data about data or information used for describing other data. This program should use reflections for accessing attributes at runtime. This we will discuss in the next chapter.

Declaring a Custom Attribute

A new custom attribute should is derived from the **System.Attribute** class. For example,

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(
   AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

In the preceding code, we have declared a custom attribute named *DeBugInfo*.

Constructing the Custom Attribute

Let us construct a custom attribute named *DeBugInfo*, which stores the information obtained by debugging any program. Let it store the following information –

- The code number for the bug
- Name of the developer who identified the bug
- Date of last review of the code

- A string message for storing the developer's remarks

The *DeBugInfo* class has three private properties for storing the first three information and a public property for storing the message. Hence the bug number, developer's name, and date of review are the positional parameters of the DeBugInfo class and the message is an optional or named parameter.

Each attribute must have at least one constructor. The positional parameters should be passed through the constructor. The following code shows the *DeBugInfo* class –

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(
   AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]

public class DeBugInfo : System.Attribute {
   private int bugNo;
   private string developer;
   private string lastReview;
   public string message;

   public DeBugInfo(int bg, string dev, string d) {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
   }
   public int BugNo {
      get {
         return bugNo;
      }
   }
   public string Developer {
      get {
         return developer;
      }
   }
   public string LastReview {
      get {
         return lastReview;
      }
   }
   public string Message {
      get {
         return message;
      }
      set {
         message = value;
      }
   }
}
```

Applying the Custom Attribute

The attribute is applied by placing it immediately before its target –

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle {
   //member variables
   protected double length;
   protected double width;
   public Rectangle(double l, double w) {
      length = l;
      width = w;
   }
   [DeBugInfo(55, "Zara Ali", "19/10/2012", Message = "Return type mismatch")]

   public double GetArea() {
      return length * width;
   }
   [DeBugInfo(56, "Zara Ali", "19/10/2012")]

   public void Display() {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
   }
}
```

In the next chapter, we retrieve attribute information using a Reflection class object.