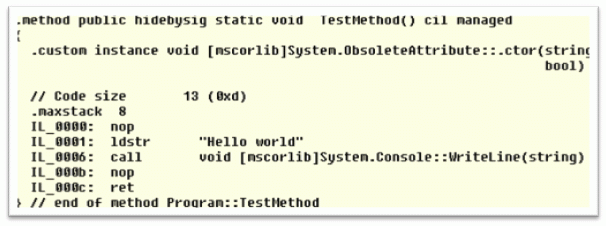
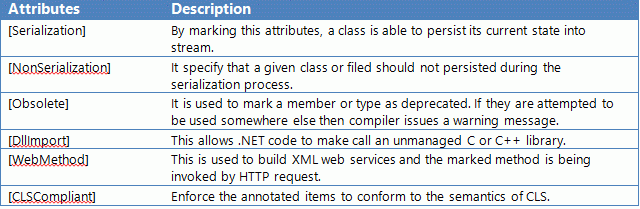
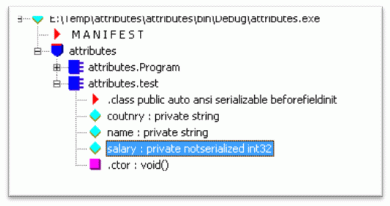
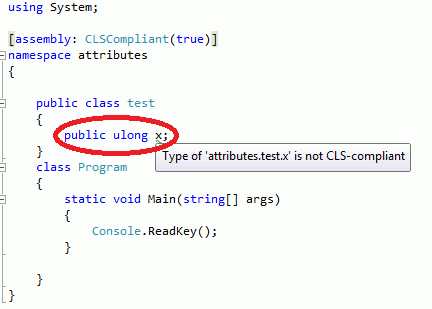
**Attributes**

Attributes allow you to add declarative information to your programs. This information can then be queried at runtime using reflection.

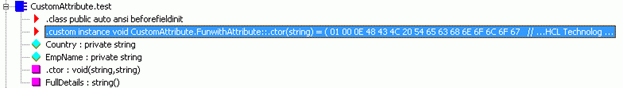
There are several Pre-defined Attributes provided by .NET. It is also possible to create your own Custom Attributes. Creating custom attributes is beyond the scope of this article.  
**A few pre-defined attributes with in the .NET framework.**  
Obsolete - Marks types and type members outdated  
WebMethod - To expose a method as an XML Web service method  
Serializable - Indicates that a class can be serialized  
**Example program using pre defined Obsolete attribute:**  
Obsolete attribute can be used with types or type members that are **obsolete (Outdated).** If a developer uses a type or a type member that is decorated with obsolete attribute, the compiler issues a warning or an error depending on how the attribute is configured.  
In this sample program, **Add(int FirstNumber, int SecondNumber)** method is decorated with [Obsolete] attribute. If you compile this program, in the output window you will see a warning message (Compile complete -- 0 errors, 1 warnings). Also, visual studio, shows a green squiggly line under the  **Add(int FirstNumber, int SecondNumber)** method. If you hover the mouse over the squiggly line, you should see the warning message.  
**Note**: If you don't see the warning message (Compile complete -- 0 errors, 1 warnings), rebuild the soultion.  
using System;  
using System.Collections.Generic;  
public class MainClass  
{  
    private static void Main()  
    {  
        Calculator.Add(10, 15);  
    }  
}  
public class Calculator  
{  
    [Obsolete]  
    public static int Add(int FirstNumber, int SecondNumber)  
    {  
        return FirstNumber + SecondNumber;  
    }  
    public static int Add(List<int> Numbers)  
    {  
        int Sum = 0;  
        foreach (int Number in Numbers)  
        {  
            Sum = Sum + Number;  
        }  
        return Sum;  
    }  
}  
  
  
The warning message says **'Calculator.Add(int, int)' is obsolete**. However, this message is not completely useful, because it says**'Calculator.Add(int, int)' is obsolete**, but not tell us which other method **should we be using instead**. So this is when we can customize, the warning message**using attribute parameters**.  
  
  
The intention of the developer of Calculator class is that, he wanted us to use **Add(List<int> Numbers)**, instead of int **Add(int FirstNumber, int SecondNumber)**. To communicate this message we can customize the warning message using attribute parameters as shown below. With this customization we are not only communicating that **Add(int FirstNumber, int SecondNumber)** method is obsolete, we are also telling to use the alternative method that is available.  
[Obsolete("Use Add(List<int> Numbers) instead")]  
public static int Add(int FirstNumber, int SecondNumber)  
  
  
If you want to generate a compiler error instead of warning, pass true for the bool error parameter of the Obsolete attribute as shown below. Now, we can't even compile the program.

[Obsolete("Use Add(List<int> Numbers) instead", true)]  
public static int Add(int FirstNumber, int SecondNumber)  
  
  
Finally, If you right click on Obsolete attribute and select Go To Definition, you will see that, an attribute is nothing but a class that inherits from System.Attribute base class.

Using Attributes With C# .NET

Attributes are like adjectives, which are used for metadata annotation that can be applied to a given type, assembly, module, method and so on. The .NET framework stipulates two types of attribute implementations, which are Predefined Attributes and Custom Attributes.   
  
Attributes are types derived from the System.Attribute class. This is an abstract class defining the required services of any attribute. The following is the syntax of an attribute;  
  
[**type:** attributeName(parameter1, parameter2,.........n)]  
  
The attribute name is the class name of the attribute. Attributes can have zero or more parameters. The following code sample states the attributes implementation in which we are declaring a method as deprecated using the obsolete attribute:  
  
using System;namespace attributes  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Attributes sample");  
            TestMethod();  
            Console.ReadKey();    
        }  
        [Obsolete("Deprecated Method",false)]  
        public static void TestMethod()  
        {  
            Console.WriteLine("Hello world");    
        }  
    }  
}  
  
The following figure shows the MSIL code of the TestMethod method as displayed in ILDASM. Notice the custom directive that defines the Obsolete attribute.  
  
  
  
**Role of Attributes**  
  
Attributes might be useful for documentation purposes.   
  
**Predefined Attributes**  
  
The predefined attributes have been defined by Microsoft as a part of .NET FCL, and many of them receive special support from the C# compiler. Which implies that for those specific attributes, the compiler could customize the compilation process in a specific way.  
  
The System.Attribute base class library provides a number of attributes in various namespaces. The following table gives a snapshot of some predefined attributes.  
  
  
  
To illustrate the predefined attributes in action, let's create a console based application to apply the implementation of them.  
  
**[Serialization] and [NonSerialization]**  
Here, assume that you have developed a Test class that can be persisted in a binary format using the [Serialization] attribute.  
  
[Serializable]  
public class test  
{  
      public Test() { }      string name;  
      string coutnry;  
      [NonSerialized]  
      int salary;  
}  
  
Once the class has been compiled, you can view the extra metadata using the ildasm.exe utility. Notice the red triangle, where these attributes are recorded using the serializable token and the salary field is tokenized using the nonserilaized attribute as in the following.  
  
  
  
**[WebMethod]**  
  
The following example depicts the implementation of XML web services. Here, the UtilityWebService class is annotated with [WebService] attributes. This class defines two methods marked with [WebMethod] attributes.   
  
[WebService(Namespace = "http://tempuri.org/")]  
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]  
// To allow this Web Service to be called from script, using ASP.NET AJAX, uncomment the following line.   
// [System.Web.Script.Services.ScriptService]  
public class UtilityWebService : System.Web.Services.WebService {    public UtilityWebService () {        //Uncomment the following line if using designed components   
        //InitializeComponent();   
    }    [WebMethod]  
    public string HelloWorld() {  
        return "Hello World";  
    }    [WebMethod]  
    public int addition(int a,int b)  
    {  
        return a+b;  
    }      
}  
  
To refer to the usage of [WebMethod] you may refer to my artile on C# Corner <http://www.c-sharpcorner.com/UploadFile/84c85b/net-web-services/>  
 **[DLLImport]**  
  
The following code plays around with the unmanaged assembly user32.dll to utilize its existing method using the [DLLImport] attributes.   
  
using System;  
using System.Runtime.InteropServices;  
namespace attributes  
{  
    public class Test  
    {  
        [DllImport("user32.dll", EntryPoint = "MessageBox")]  
        public static extern int ShowMessageBox(int hWnd,string text, string caption,uint type);  
    }  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string caption = "Hello World";  
            string text = "Sample Article on DLLImport Attribute";  
            test.ShowMessageBox(0, text, caption, 0);     
            Console.ReadKey();    
        }          
    }  
}  
  
**Note:** a member can assign more than one attribute and can be applied multiple times itself.  
  
Once this code is compiled successfully, it produces the following output as in the following:  
  
  
  
**[CLSCompliant]**  
  
If you annotate this attribute at the assembly or module level and you try to use the following non CLR compliant code, then the compiler issues a warning message.  
  
  
  
To refer to the usage of [CLSCompliant] you may refer to my artile on C# Corner <http://www.c-sharpcorner.com/UploadFile/84c85b/cross-language-interoperability-with-C-Sharp-net/>  
  
**Custom Attributes**  
  
We can create custom attributes for private usage or to be published in a library for others. The following steps are the definitive procedure for creating custom attributes:

1. The custom attribute class should be derived from System.Attribute
2. The Attribute name should  suffixed by "Attribute".
3. Set the probable targets with the AttributeUsage attribute.
4. Implement the class constructor and write-accessible properties.

The first step in building a custom attribute is to create a new class FunWith with Attribute suffix that derives from the System.Attribute class. Then define the class constructor and write an accessible property as Company subsequently.  
  
[AttributeUsage(AttributeTargets.Class)]  
public class FunwithAttribute : Attribute  
{  
      public FunwithAttribute(string s)  
      {  
          this.Company = s;   
      }  
      public string Company { get; set; }   
}  
  
It is now time to apply a custom attribute class on another class. So we are creating another class test which has a FunWith attribute annotation in which we are passing the company name information.  
  
[Funwith("HCL Technology")]  
public class test  
{  
      public test(string name, string country)  
      {  
           this.EmpName = name;  
           this.Country = country;     
      }  
      public string FullDetails()  
      {  
           string str = EmpName + "-" + Country;  
           return str;  
      }      private string EmpName;  
      private string Country;  
}  
class Program  
{  
      static void Main(string[] args)  
      {  
           test obj = new test("Vidya Vrat","India");  
           Console.WriteLine("Result:{0}",obj.FullDetails());  
           Console.ReadKey();    
       }  
}  
  
After compiling this program, it yields the following output.  
  
**Output**  
**Result:** Vidya Vrat - India  
  
Well, as you can notice, the custom attribute does not impose any impact on the final output. But we can reveal the annotation that occurs by attributes at the metadata level using ildasm.exe as in the following;  
  
  
  
Custom defined attributes are sometimes valuable simply as information. However, the real power lies in associating with an attribute. You can read custom attributes with reflection using Attribute.GetCustomAttribute and Type.  
  
The following code accesses the custom attribute values at runtime using reflection. Here, we are storing the custom attribute members in an array of object types, then looping through it to get the property value.  
  
static void Main(string[] args)  
{  
       MemberInfo info = typeof(test);  
       object[] attrib = info.GetCustomAttributes(typeof(FunwithAttribute), false);  
       foreach (Object attribute in attrib)  
       {  
             FunwithAttribute a = (FunwithAttribute)attribute;  
             Console.WriteLine("Company: {0}", a.Company);                  
        }  
}  
  
The following code illustrates the actual utilization of custom attribute values. Here, I am displaying some values based on the custom attribute's Boolean value. The output varies on status True or False values.  
  
using System;  
using System.Reflection;  public class CheckStatus : Attribute  
{  
      private bool Val = false;  
      public bool status  
      {  
          get { return Val; }  
      }  
     public CheckStatus(bool val)  
     {  
          Val = val;  
     }  
}  
  
Now we are annotating the Custom Attribute in the Test class. Here, we need to configure the CheckStatus value to true to false manually. In the FullDetails() method, we are accessing the custom attributes members by foreach loop construct and later we are checking whether the status value is true or false.  
  
[CheckStatus(false)]  
public class test  
{   private string EmpName;  
   private string Country;   public test(string name, string country)  
   {  
        this.EmpName = name;  
        this.Country = country;     
   }  
   public string FullDetails()  
   {  
        string str = null;  
        Type type = this.GetType();        CheckStatus[] attrib =   
           (CheckStatus[])type.GetCustomAttributes(typeof(CheckStatus), false);        if (attrib[0].status == true)  
        {  
             str = EmpName + "-" + Country;  
         }  
         else  
         {  
             str = "Hi " + EmpName;  
         }         return str;  
    }  
}  
class Program  
{  
    static void Main(string[] args)  
    {  
         test obj = new test("Vidya Vrat","India");  
         Console.WriteLine("Result:{0}",obj.FullDetails());  
         Console.ReadKey();    
    }  
}  
  
After successfully compiling this code, the following figure depicts the output from both cases:  
  
***Result : Vidya Vrat - India  
Status = True  
Result: Hi Vidya Vrat  
Starus = False***  
  
Rather than applying attributes on an individual type, it is also possible to apply an attribute on all the types within a given assembly. To do so, simply add the attributes at assembly level in the AssemblyInfo.cs file. This file is a handy place to put attributes to be applied at the assembly level.  
  
using System.Reflection;  
using System.Runtime.CompilerServices;  
using System.Runtime.InteropServices;   
  
[assembly: AssemblyTitle("CustomAttribute")]  
[assembly: AssemblyDescription("")]  
[assembly: AssemblyConfiguration("")]  
[assembly: AssemblyCompany("")]  
[assembly: AssemblyProduct("CustomAttribute")]  
[assembly: AssemblyCopyright("Copyright Â©  2013")]  
[assembly: AssemblyTrademark("")]  
[assembly: AssemblyCulture("")]  
[assembly: ComVisible(false)]  
[assembly: Guid("ce5fc30b-e670-4115-aa64-4be10e7b6ea9")]  
[assembly: AssemblyVersion("1.0.0.0")]  
[assembly: AssemblyFileVersion("1.0.0.0")]