Attributes

The common language runtime (CLR) uses attributes to annotate programming elements such as types, fields, methods, and properties.

Attributes – associate information to types and members:

- Design-time information (such as documentation)
- Run-time information (such as the name of a database column for a field)
- Run-time behavioral characteristics (such as whether a given member is transactionable, or capable of participating in a transaction)

1. Defining Attributes
   - A class derived from the System.Attribute base class
   - Common practice:
     - The attribute class name has an Attribute suffix
     - The suffix is omitted when the attribute is attached to a type or member

```
public class attribute_name_Attribute : System.Attribute
{
}
```

2. Attribute Types
   - Class attributes
   - Method attributes
   - Field attributes

3. Attribute Parameters
   - Positional parameters – each public constructor defines a sequence of positional parameters
   - Named parameters – each non static public field and read/write property
4. Attaching an attribute – an instance of the attribute class is attached to a type or a member and is at the metadata for the type

```
[attribute_name (list_of_positional_parameters,
                 name_of_named_parameter = value, ...)]
```

STAThreadAttribute (STA – Single Threaded Apartment) – specifies that the default threading model for an application is single-threaded apartment.

```
class Class1
{
    [STAThread]
    static void Main (string[] args)
    {
    }
}
```

AttributeUssageAttribute – define how the attribute to be used.

```
[AttributeUsage(AttributeTargets validOn, AllowMultiple = true/false,
                Inherited = true/false)]
```

validOn – valid program element

AttributeTargets.Module | Class | Struct | Enum | Constructor | Method | Property | Field | Event | Interface | Parameter | Delegate | ReturnValue | GenericParameter | All

4. Querying for Attributes

Reflection is a technique to query a type or member about its attachment attributes.

Reflection dynamically determines at run time characteristics of an application using the .NET Framework Reflection APIs to iterate the metadata.

Example:

Custom attribute DeveloperAttribute

- Save the developer name as a positional attribute
- Can be attached for a class or a structure
- Allows multiple instances of the attribute

```
using System;
namespace CustomAttributes
{
   [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                   AllowMultiple = true)]
   public class DeveloperAttribute : Attribute
   {
      private string name;
      public DeveloperAttribute(string name)
      {
         this.name = name;
      }
      public string Developer
      {
         get {   return name;   }
      }
      public override string ToString()
      {
         return "Developer : " + Developer;
      }
   }
}
```

```
using System;
using System.Reflection;

namespace TestAttributes
{
   [CustomAttributes.Developer("Alexander Rusev")]
   [CustomAttributes.Developer("Vladislav Zdravkov")]
   public class TestAttributes
   {
      public string title;
      public void MethodA() { }
      public void MethodB() { }
      public void MethodC() { }
```

Attaching the attribute to the class and omitting the Attribute suffix

```
   static void Main(string[] args)
   {
      // Querying for attributes of the class TestAttributes
      Type t = typeof(TestAttributes);
      object[] attributes = t.GetCustomAttributes(true);
      Console.WriteLine("Attributes for: " + t.Name);
      foreach (object o in attributes)
      {
         Console.WriteLine("\t" + o);
      }
      Console.WriteLine();
   }
}
```

Returns the System.Type object associated with the argument

Returns all attributes (array of Object); true – search this member's inheritance chain to find the attributes

Results:

```
Attributes for: TestAttributes
        Developer: Alexander Rusev
        Developer: Vladislav Zdravkov
```

Example:

Custom attribute IsTestedAttribute

- Save the testing date of a type or a member as non mandatory (named) parameter
- Can be attached for all types and members
- Allows multiple instances of the attribute

```
using System;
namespace CustomAttributes
{
   [AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
   public class IsTestedAttribute : Attribute
   {
      private string date;
      public string Date
      {
         get {   return date;   }
         set {   date = value; }
      }
      public IsTestedAttribute()
      {   date = null;   }
      public override string ToString()
      {   string value = "Tested!";
         if (date != null)
            value += " Date: " + date;
         return value;
      }
   }
}
```

```
using System;
using System.Reflection;

namespace TestAttributes
{
   [CustomAttributes.IsTested]
   public class TestAttributes
   {
      public string title;

      [CustomAttributes.IsTested(Date = "05.12.2008")]
      [CustomAttributes.IsTested(Date = "01.05.2009 ")]
      public void MethodA() { }

      [CustomAttributes.IsTested(Date = "19.10.2009 ")]
      public void MethodB() { }

      public void MethodC() { }
```

```
      static void Main(string[] args)
      {
         // Querying for attributes of the class TestAttributes
         Type t = typeof(TestAttributes);
         object[] attributes = t.GetCustomAttributes(true);
         Console.WriteLine("Attributes for: " + t.Name);
         foreach (object o in attributes)
         {
            Console.WriteLine("\t" + o);
         }
         Console.WriteLine();
```

Returns and array of type MemberInfo for the public methods of the type

Returns an array of type Attribute for a method

```
      // Querying for method attributes of the class TestAttributes
      MemberInfo[] members = t.GetMethods();
      Console.WriteLine("Attributes for methods of the class: " + t.Name);
      foreach (MethodInfo method in members)
      {
         bool flag = false;
         foreach (Attribute attr in method.GetCustomAttributes(true))
         {
            Console.WriteLine("\t{0}: {1}", method.Name, attr);
            if (attr is CustomAttributes.IsTestedAttribute)
               flag = true;
         }
         if (!flag)
            Console.WriteLine("\t{0}: Is not tested!", method.Name);
      }
      Console.WriteLine();
   }
  }
}
```

Results:

```
Attributes for: TestAttributes
      Tested!

Attributes for methods of the class: TestAttributes
      MethodA: Tested! Date: 05.12.2008
      MethodA: Tested! Date: 01.05.2009
      MethodB: Tested! Date: 19.10.2009
      MethodC: Is not tested!
      GetType: Is not tested!
      ToString: Is not tested!
      Equals: Is not tested!
      GetHashCode: Is not tested!
```

Example:

Custom attributeValidLengthAttribute

- Save the minimal and the maximal length of a field as a positional parameter and a message as a non mandatory (named) parameter
- Can be attached for a field or a property

```
using System;
namespace CustomAttributes
{  [AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
   public class ValidLengthAttribute : Attribute
   {  private int min;                    // minimum length
      private int max;                    // maximum length
      private string message;             // message
      public ValidLengthAttribute(int min, int max)
      {  this.min = min;
         this.max = max;
      }
      public string Message
      {  get { return (message); }
         set { message = value; }
      }
      public bool IsValid(string theValue)
      {  int length = theValue.Length;
         if (length >= min && length <= max) return true;
         return false;
      }
   }
}
```

```
using System;
using System.Reflection;

namespace TestAttributes
{
   [CustomAttributes.Developer("Alexander Rusev")]
   [CustomAttributes.Developer("Vladislav Zdravkov")]
   [CustomAttributes.IsTested]
   public class TestAttributes
   {
    [CustomAttributes.ValidLength(4, 30,
    Message = "The length of the title has to be between 4 and 30 symbols")]
      public string title;

      [CustomAttributes.IsTested(Date = "05.12.2008")]
      [CustomAttributes.IsTested(Date = "01.05.2009 ")]
      public void MethodA() { }

      [CustomAttributes.IsTested(Date = "19.10.2009 ")]
      public void MethodB() { }

      public void MethodC() { }
```

```
   static void Main(string[] args)
   {
      // Querying for attributes of the class TestAttributes
      Type t = typeof(TestAttributes);
      object[] attributes = t.GetCustomAttributes(true);
      Console.WriteLine("Attributes for: " + t.Name);
      foreach (object o in attributes)
      {
         Console.WriteLine("\t" + o);
      }
      Console.WriteLine();
```

```
      // Querying for method attributes of the class TestAttributes
      MemberInfo[] members = t.GetMethods();
      Console.WriteLine("Attributes for methods of the class: " + t.Name);
      foreach (MethodInfo method in members)
      {
         bool flag = false;
         foreach (Attribute attr in method.GetCustomAttributes(true))
         {
            Console.WriteLine("\t{0}: {1}", method.Name, attr);
            if (attr is CustomAttributes.IsTestedAttribute)
               flag = true;
         }
         if (!flag)
            Console.WriteLine("\t{0}: Is not tested!", method.Name);
      }
      Console.WriteLine();
```

```
      // Querying for field attributes of the class TestAttributes
      TestAttributes test = new TestAttributes();
      test.title = "Testing of custom attributes";
      FieldInfo[] fields = t.GetFields();
      Console.WriteLine("Attributes for the fields of the class: " + t.Name);
      foreach (FieldInfo field in fields)
         foreach (Attribute attr in field.GetCustomAttributes(true))
         {  Console.WriteLine("\t{0}: {1}", field.Name, attr);
            CustomAttributes.ValidLengthAttribute vla =
                                 attr as CustomAttributes.ValidLengthAttribute;
            if (null != vla)
            {  string theValue = (string)field.GetValue(test);
               Console.WriteLine(vla.IsValid(theValue) ?
                           "\tCorrect length!" : "\tUncorrect length!");
            }
         }
      Console.WriteLine();
   }
  }
}
```

Returns an array of type FieldInfo for public fields of the type

Returns the field value

Results:

```
Attributes for: TestAttributes
      Tested!
      Developer: Alexander Rusev
      Developer: Владислав Здравков

Attributes for methods of the class: TestAttributes
      MethodA: Tested! Date: 05.12.2008
      MethodA: Tested! Date: 01.05.2009
      MethodB: Tested! Date: 19.10.2009
      MethodC: Is not tested!
      GetType: Is not tested!
      ToString: Is not tested!
      Equals: Is not tested!
      GetHashCode: Is not tested!

Attributes for fields of the class: TestAttributes
      title: CustomAttributes.ValidLengthAttribute
      Correct length!
```

5. Rules

- The positional parameters must be specified first and after that the named parameters can exist in any order
- A positional parameter can not be named
- The named parameters can be any publicly accessible field or property including a non static or constant setter method
- The types of positional and named parameters can limited to:
  - bool, byte, char, double, float, int, long, short, string
  - System.Type
  - object
  - enum and any types in which it's nested with public access

- A one-dimensional array of the types just listed
- The attribute constructor doesn't have a class as a parameter (attributes are attached at design time – the class instances are not created at that point)

6. Predefined Attributes

.NET Attribute	Valid Targets	Description
AttributeUsage	class	Specifies the valid usage of another attribute class.
CLSCompliant	all	Indicates whether a program element is compliant with the CLS (Common Language Specification).
Conditional	method	Indicates that the compiler can ignore any calls to this method if the associated strings are defined.
DllImport	method	Specifies the DLL location that contains the implementation of an external method.
MTAThread	method (Main)	Indicates that the default threading model for an application is multithreaded apartment (MTA).
NonSerialized	field	Applies to field of a class flagged as Serializable; specifies that these fields won't be serialized.

.NET Attribute	Valid Targets	Description
Obsolete	all except Assembly, Module, Parameter and Return	Marks an element obsolete – it informs the user that the element will be removed in future versions of the product.
ParamArray	parameter	Allows a single parameter to be implicitly treated as a params (array) parameter.
Serializable	class, struct, enum, delegate	Specifies that all public and private fields of this type can be serialized.
STAThread	method (Main)	Indicates that the default threading model for an application is STA.
StructLayout	class, struct	Specifies the nature of the data layout of a class or struct, such as Auto, Explicit, or Sequential.
ThreadStatic	field (static)	Implements thread-local storage (TLS) – each thread has its own copy of the static field, that isn't shared across multiple threads.

Example: Conditional attribute specifies that the method is compiled into the code only if the preprocessor directive "DEBUG" is defined.

```
using System.Diagnostics;
...
[Conditional ("DEBUG")]
public void SomeDebugFunc ()
{
        Console.WriteLine ("SomeDebugFunc");
}
```

Example: Obsolete attribute has two parameters: the first (string) parameter is a part of the compiler diagnostics; if the second parameter is true, the compiler will produce an error if we attempt to call the method; if the second parameter is false, the code compiles with no warnings or errors.

```
using System;
...
[Obsolete ("Don't use OldFunc, use NewFunc instead", true)]
public void OldFunc()
{
    Console.WriteLine("Old");
}

public void NewFunc()
{
    Console.WriteLine("New");
}
```