CHAPTER 18

Custom Attributes

In this chapter:

In this chapter, I'll discuss one of the most innovative features the Microsoft .NET Framework has to offer: *custom attributes*. Custom attributes allow you to declaratively annotate your code constructs, thereby enabling special features. Custom attributes allow information to be defined and applied to almost any metadata table entry. This extensible metadata information can be queried at run time to dynamically alter the way code executes. As you use the various .NET Framework technologies (Windows Forms, WPF,WCF, , and so on), you'll see that they all take advantage of custom attributes, allowing developers to express their intentions within code very easily. A solid understanding of custom attributes is necessary for any .NET Framework developer.

Using Custom Attributes

Attributes, such as `public`, `private`, `static`, and so on, can be applied to types and members. I think we'd all agree on the usefulness of applying attributes, but wouldn't it be even more useful if we could define our own attributes? For example, what if I could define a type and somehow indicate that the type can be remoted via serialization? Or maybe I could apply an attribute to a method to indicate that certain security permissions must be granted before the method can execute.

Of course, creating and applying user-defined attributes to types and methods would be great and convenient, but it would require the compiler to be aware of these attributes so it would emit the attribute information into the resulting metadata. Because compiler vendors usually prefer not to release the source code for their compiler, Microsoft came up with another way to allow user-defined attributes. This mechanism, called *custom attributes*, is an incredibly powerful mechanism that's useful at both application design time and run time. Anyone can define and use custom attributes, and all compilers that target the common language runtime (CLR) must be designed to recognize custom attributes and emit them into the resulting metadata.

The first thing you should realize about custom attributes is that they're just a way to associate additional information with a target. The compiler emits this additional information into the managed module's metadata. Most attributes have no meaning for the compiler; the compiler simply detects the attributes in the source code and emits the corresponding metadata.

The .NET Framework Class Library (FCL) defines literally hundreds of custom attributes that can be applied to items in your own source code. Here are some examples:

- Applying the `DllImport` attribute to a method informs the CLR that the implementation of the method is actually in unmanaged code contained in the specified DLL.
- Applying the `Serializable` attribute to a type informs the serialization formatters that an instance's fields may be serialized and deserialized.
- Applying the `AssemblyVersion` attribute to an assembly sets the version number of the assembly.
- Applying the `Flags` attribute to an enumerated type causes the enumerated type to act as a set of bit flags.

Following is some C# code with many attributes applied to it. In C#, you apply a custom attribute to a target by placing the attribute in square brackets immediately before the target. It's not important to understand what this code does. I just want you to see what attributes look like.

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

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
internal sealed class OSVERSIONINFO {
   public OSVERSIONINFO() {
      OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
   }

   public UInt32 OSVersionInfoSize = 0;
   public UInt32 MajorVersion      = 0;
   public UInt32 MinorVersion      = 0;
   public UInt32 BuildNumber       = 0;
   public UInt32 PlatformId        = 0;

   [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
   public String CSDVersion        = null;
}

internal sealed class MyClass {
   [DllImport("Kernel32", CharSet = CharSet.Auto, SetLastError = true)]
   public static extern Boolean GetVersionEx([In, Out] OSVERSIONINFO ver);
}
```

In this case, the `StructLayout` attribute is applied to the `OSVERSIONINFO` class, the `MarshalAs` attribute is applied to the `CSDVersion` field, the `DllImport` attribute is applied to the `GetVersionEx` method, and the `In` and `Out` attributes are applied to `GetVersionEx`'s `ver` parameter. Every programming language defines the syntax a developer must use in order to apply a custom attribute to a target. Microsoft Visual Basic .NET, for example, requires angle brackets (<, >) instead of square brackets.

The CLR allows attributes to be applied to just about anything that can be represented in a file's metadata. Most commonly, attributes are applied to entries in the following definition tables: TypeDef (classes, structures, enumerations, interfaces, and delegates), MethodDef (including constructors), ParamDef, FieldDef, PropertyDef, EventDef, AssemblyDef, and ModuleDef. Specifically, C# allows you to apply an attribute only to source code that defines any of the following targets: assembly, module, type (class, struct, enum, interface, delegate), field, method (including constructors), method parameter, method return value, property, event, and generic type parameter.

When you're applying an attribute, C# allows you to specify a prefix specifically indicating the target the attribute applies to. The following code shows all of the possible prefixes. In many cases, if you leave out the prefix, the compiler can still determine the target an attribute applies to, as shown in the previous example. In some cases, the prefix must be specified to make your intentions clear to the compiler. The prefixes shown in italics in the following code are mandatory.

```
using System;

[assembly: SomeAttr]            // Applied to assembly
[module:   SomeAttr]            // Applied to module

[type:     SomeAttr]            // Applied to type
internal sealed class SomeType<[typevar: SomeAttr] T> {   // Applied to generic type variable

   [field: SomeAttr]            // Applied to field
   public Int32 SomeField = 0;

   [return: SomeAttr]           // Applied to return value
   [method: SomeAttr]           // Applied to method
   public Int32 SomeMethod(
      [param: SomeAttr]         // Applied to parameter
      Int32 SomeParam) { return SomeParam; }

   [property: SomeAttr]         // Applied to property
   public String SomeProp {
      [method: SomeAttr]        // Applied to get accessor method
      get { return null; }
   }

   [event:  SomeAttr]           // Applied to event
   [field:  SomeAttr]           // Applied to compiler-generated field
   [method: SomeAttr]           // Applied to compiler-generated add & remove methods
   public event EventHandler SomeEvent;
}
```

Now that you know how to apply a custom attribute, let's find out what an attribute really is. A custom attribute is simply an instance of a type. For Common Language Specification (CLS) compliance, custom attribute classes must be derived, directly or indirectly, from the public abstract `System.Attribute` class. C# allows only CLS-compliant attributes. By examining the .NET Framework SDK documentation, you'll see that the following classes (from the earlier example) are defined: `StructLayoutAttribute`, `MarshalAsAttribute`, `DllImportAttribute`, `InAttribute`, and `OutAttribute`. All of these classes happen to be defined in the `System.Runtime.InteropServices` namespace, but attribute classes can be defined in any namespace. Upon further examination, you'll notice that all of these classes are derived from `System.Attribute`, as all CLS-compliant attribute classes must be.

Note When applying an attribute to a target in source code, the C# compiler allows you to omit the `Attribute` suffix to reduce programming typing and to improve the readability of the source code. My code examples in this chapter take advantage of this C# convenience. For example, my source code contains `[DllImport(...)]` instead of `[DllImportAttribute(...)]`.

As I mentioned earlier, an attribute is an instance of a class. The class must have a public constructor so that instances of it can be created. So when you apply an attribute to a target, the syntax is similar to that for calling one of the class's instance constructors. In addition, a language might permit some special syntax to allow you to set any public fields or properties associated with the attribute class. Let's look at an example. Recall the application of the `DllImport` attribute as it was applied to the `GetVersionEx` method earlier.

```
[DllImport("Kernel32", CharSet = CharSet.Auto, SetLastError = true)]
```

The syntax of this line should look pretty strange to you because you could never use syntax like this when calling a constructor. If you examine the `DllImportAttribute` class in the documentation, you'll see that its constructor requires a single `String` parameter. In this example, `"Kernel32"` is being passed for this parameter. A constructor's parameters are called *positional parameters* and are mandatory; the parameter must be specified when the attribute is applied.

What are the other two "parameters"? This special syntax allows you to set any public fields or properties of the `DllImportAttribute` object after the object is constructed. In this example, when the `DllImportAttribute` object is constructed and `"Kernel32"` is passed to the constructor, the object's public instance fields, `CharSet` and `SetLastError`, are set to `CharSet.Auto` and `true`, respectively. The "parameters" that set fields or properties are called *named parameters* and are optional because the parameters don't have to be specified when you're applying an instance of the attribute. A little later on, I'll explain what causes an instance of the `DllImportAttribute` class to actually be constructed.

Also note that it's possible to apply multiple attributes to a single target. For example, in this chapter's first program listing, the `GetVersionEx` method's `ver` parameter has both the `In` and `Out` attributes applied to it. When applying multiple attributes to a single target, be aware that the order of attributes has no significance. Also, in C#, each attribute can be enclosed in square brackets, or multiple attributes can be comma-separated within a single set of square brackets. If the attribute class's constructor takes no parameters, the parentheses are optional. Finally, as mentioned earlier, the `Attribute` suffix is also optional. The following lines behave identically and demonstrate all of the possible ways of applying multiple attributes.

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

Defining Your Own Attribute Class

You know that an attribute is an instance of a class derived from `System.Attribute`, and you also know how to apply an attribute. Let's now look at how to define your own custom attribute classes. Say you're the Microsoft employee responsible for adding the bit flag support to enumerated types. To accomplish this, the first thing you have to do is define a `FlagsAttribute` class.

```
namespace System {
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

Notice that the `FlagsAttribute` class inherits from `Attribute`; this is what makes the `Flags-Attribute` class a CLS-compliant custom attribute. In addition, the class's name has a suffix of `Attribute`; this follows the standard convention but is not mandatory. Finally, all non-abstract attributes must contain at least one public constructor. The simple `FlagsAttribute` constructor takes no parameters and does absolutely nothing.

> **Important** You should think of an attribute as a logical state container. That is, while an attribute type is a class, the class should be simple. The class should offer just one public constructor that accepts the attribute's mandatory (or positional) state information, and the class can offer public fields/properties that accept the attribute's optional (or named) state information. The class should not offer any public methods, events, or other members.
>
> In general, I always discourage the use of public fields, and I still discourage them for attributes. It is much better to use properties because this allows more flexibility if you ever decide to change how the attribute class is implemented.

So far, instances of the `FlagsAttribute` class can be applied to any target, but this attribute should really be applied to enumerated types only. It doesn't make sense to apply the attribute to a property or a method. To tell the compiler where this attribute can legally be applied, you apply an instance of the System.AttributeUsageAttribute class to the attribute class. Here's the new code.

```
namespace System {
   [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

In this new version, I've applied an instance of AttributeUsageAttribute to the attribute. After all, the attribute type is just a class, and a class can have attributes applied to it. The Attribute-Usage attribute is a simple class that allows you to specify to a compiler where your custom attribute can legally be applied. All compilers have built-in support for this attribute and generate errors when a user-defined custom attribute is applied to an invalid target. In this example, the AttributeUsage attribute specifies that instances of the Flags attribute can be applied only to enumerated type targets.

Because all attributes are just types, you can easily understand the AttributeUsageAttribute class. Here's what the FCL source code for the class looks like.

```
[Serializable]
[AttributeUsage(AttributeTargets.Class, Inherited=true)]
public sealed class AttributeUsageAttribute : Attribute {
   internal static AttributeUsageAttribute Default =
      new AttributeUsageAttribute(AttributeTargets.All);

   internal Boolean m_allowMultiple = false;
   internal AttributeTargets m_attributeTarget = AttributeTargets.All;
   internal Boolean m_inherited = true;

   // This is the one public constructor
   public AttributeUsageAttribute(AttributeTargets validOn) {
      m_attributeTarget = validOn;
   }

   internal AttributeUsageAttribute(AttributeTargets validOn,
      Boolean allowMultiple, Boolean inherited) {
      m_attributeTarget = validOn;
      m_allowMultiple = allowMultiple;
      m_inherited = inherited;
   }

   public Boolean AllowMultiple {
      get { return m_allowMultiple; }
      set { m_allowMultiple = value; }
   }

   public Boolean Inherited {
      get { return m_inherited; }
      set { m_inherited = value; }
   }
   public AttributeTargets ValidOn {
      get { return m_attributeTarget; }
   }
}
```

As you can see, the `AttributeUsageAttribute` class has a public constructor that allows you to pass bit flags that indicate where your attribute can legally be applied. The `System.AttributeTargets` enumerated type is defined in the FCL as follows.

```
[Flags, Serializable]
public enum AttributeTargets {
   Assembly         = 0x0001,
   Module           = 0x0002,
   Class            = 0x0004,
   Struct           = 0x0008,
   Enum             = 0x0010,
   Constructor      = 0x0020,
   Method           = 0x0040,
   Property         = 0x0080,
   Field            = 0x0100,
   Event            = 0x0200,
   Interface        = 0x0400,
   Parameter        = 0x0800,
   Delegate         = 0x1000,
   ReturnValue      = 0x2000,
   GenericParameter = 0x4000,
   All           = Assembly    | Module    | Class    | Struct | Enum  |
                    Constructor | Method    | Property | Field  | Event |
                    Interface   | Parameter | Delegate | ReturnValue |
                    GenericParameter
}
```

The `AttributeUsageAttribute` class offers two additional public properties that can optionally be set when the attribute is applied to an attribute class: `AllowMultiple` and `Inherited`.

For most attributes, it makes no sense to apply them to a single target more than once. For example, nothing is gained by applying the `Flags` or `Serializable` attributes more than once to a single target. In fact, if you tried to compile the following code, the compiler would report the following message: `error CS0579: Duplicate 'Flags' attribute.`

```
[Flags][Flags]
internal enum Color {
   Red
}
```

For a few attributes, however, it does make sense to apply the attribute multiple times to a single target. In the FCL, the `ConditionalAttribute` attribute class allows multiple instances of itself to be applied to a single target. If you don't explicitly set `AllowMultiple` to `true`, your attribute can be applied no more than once to a selected target.

AttributeUsageAttribute's other property, Inherited, indicates if the attribute should be applied to derived classes and overriding methods when applied on the base class. The following code demonstrates what it means for an attribute to be inherited.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited=true)]
internal class TastyAttribute : Attribute {
}

[Tasty][Serializable]
internal class BaseType {

   [Tasty] protected virtual void DoSomething() { }
}

internal class DerivedType : BaseType {
   protected override void DoSomething() { }
}
```

In this code, DerivedType and its DoSomething method are both considered Tasty because the TastyAttribute class is marked as inherited. However, DerivedType is not serializable because the FCL's SerializableAttribute class is marked as a noninherited attribute.

Be aware that the .NET Framework considers targets only of classes, methods, properties, events, fields, method return values, and parameters to be inheritable. So when you're defining an attribute type, you should set Inherited to true only if your targets include any of these targets. Note that inherited attributes do not cause additional metadata to be emitted for the derived types into the managed module. I'll say more about this a little later in the "Detecting the Use of a Custom Attribute" section.

Note If you define your own attribute class and forget to apply an AttributeUsage attribute to your class, the compiler and the CLR will assume that your attribute can be applied to all targets, can be applied only once to a single target, and is inherited. These assumptions mimic the default field values in the AttributeUsageAttribute class.

Attribute Constructor and Field/Property Data Types

When defining your own custom attribute class, you can define its constructor to take parameters that must be specified by developers when they apply an instance of your attribute type. In addition, you can define nonstatic public fields and properties in your type that identify settings that a developer can optionally choose for an instance of your attribute class.

When defining an attribute class's instance constructor, fields, and properties, you must restrict yourself to a small subset of data types. Specifically, the legal set of data types is limited to the following: Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, String, Type, Object, or an enumerated type. In addition, you can use a single-dimensional, zero-based array of any of these types. However, you should avoid using arrays because a custom attribute class whose constructor takes an array is not CLS-compliant.

When applying an attribute, you must pass a compile-time constant expression that matches the type defined by the attribute class. Wherever the attribute class defines a `Type` parameter, `Type` field, or `Type` property, you must use C#'s `typeof` operator, as shown in the following code. Wherever the attribute class defines an `Object` parameter, `Object` field, or `Object` property, you can pass an `Int32`, `String`, or any other constant expression (including `null`). If the constant expression represents a value type, the value type will be boxed at run time when an instance of the attribute is constructed.

Here's an example of an attribute and its usage.

```
using System;

internal enum Color { Red }

[AttributeUsage(AttributeTargets.All)]
internal sealed class SomeAttribute : Attribute {
   public SomeAttribute(String name, Object o, Type[] types) {
      // 'name' refers to a String
      // 'o' refers to one of the legal types (boxing if necessary)
      // 'types' refers to a 1-dimension, 0-based array of Types
   }
}

[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
internal sealed class SomeType {
}
```

Logically, when a compiler detects a custom attribute applied to a target, the compiler constructs an instance of the attribute class by calling its constructor, passing it any specified parameters. Then the compiler initializes any public fields and properties by using the values specified via the enhanced constructor syntax. Now that the custom attribute object is initialized, the compiler serializes the attribute object's state out to the target's metadata table entry.

Important I've found this to be the best way for developers to think of custom attributes: instances of classes that have been serialized to a byte stream that resides in metadata. Later, at run time, an instance of the class can be constructed by deserializing the bytes contained in the metadata. In reality, what actually happens is that the compiler emits the information necessary to create an instance of the attribute class into metadata. Each constructor parameter is written out with a 1-byte type ID followed by the value. After "serializing" the constructor's parameters, the compiler emits each of the specified field and property values by writing out the field/property name followed by a 1-byte type ID and then the value. For arrays, the count of elements is saved first, followed by each individual element.

Detecting the Use of a Custom Attribute

Defining an attribute class is useless by itself. Sure, you could define attribute classes all you want and apply instances of them all you want, but this would just cause additional metadata to be written out to the assembly—the behavior of your application code wouldn't change.

In Chapter 15, "Enumerated Types and Bit Flags," you saw that applying the Flags attribute to an enumerated type altered the behavior of System.Enum's ToString and Format methods. The reason that these methods behave differently is that they check at run time if the enumerated type that they're operating on has the Flags attribute metadata associated with it. Code can look for the presence of attributes by using a technology called *reflection*. I'll give some brief demonstrations of reflection here, but I'll discuss it fully in Chapter 23, "Assembly Loading and Reflection."

If you were the Microsoft employee responsible for implementing Enum's Format method, you would implement it like the following.

```
public override String ToString() {

   // Does the enumerated type have an instance of
   // the FlagsAttribute type applied to it?
   if (this.GetType().IsDefined(typeof(FlagsAttribute), false)) {
      // Yes; execute code treating value as a bit flag enumerated type.
      ...
   } else {
      // No; execute code treating value as a normal enumerated type.
      ...
   }
   ...
}
```

This code calls Type's IsDefined method, effectively asking the system to look up the metadata for the enumerated type and see whether an instance of the FlagsAttribute class is associated with it. If IsDefined returns true, an instance of FlagsAttribute is associated with the enumerated type, and the Format method knows to treat the value as though it contained a set of bit flags. If IsDefined returns false, Format treats the value as a normal enumerated type.

So if you define your own attribute classes, you must also implement some code that checks for the existence of an instance of your attribute class (on some target) and then execute some alternate code path. This is what makes custom attributes so useful!

The FCL offers many ways to check for the existence of an attribute. If you're checking for the existence of an attribute via a System.Type object, you can use the IsDefined method as shown earlier. However, sometimes you want to check for an attribute on a target other than a type, such as an assembly, a module, or a method. For this discussion, let's concentrate on the extension methods defined by the System.Reflection.CustomAttributeExtensions class. This class defines three static methods for retrieving the attributes associated with a target: IsDefined, GetCustomAttributes, and GetCustomAttribute. Each of these functions has several overloaded versions. For example, each method has a version that works on type members (classes, structs, enums, interfaces, delegates, constructors, methods, properties, fields, events, and return types), parameters, and assemblies. There are also versions that allow you to tell the system to walk up the derivation hierarchy to include inherited attributes in the results. Table 18-1 briefly describes what each method does.

TABLE 18-1 `System.Reflection.CustomAttributeExtensions'` Methods That Reflect over Metadata Looking for Instances of CLS-Compliant Custom Attributes

Method	Description
`IsDefined`	Returns `true` if there is at least one instance of the specified `Attribute`-derived class associated with the target. This method is efficient because it doesn't construct (deserialize) any instances of the attribute class.
`GetCustomAttributes`	Returns a collection of the specified attribute objects that have been applied to the target. Each instance is constructed (deserialized) by using the parameters, fields, and properties specified during compilation. If the target has no instances of the specified attribute class, an empty collection is returned. This method is typically used with attributes that have `AllowMultiple` set to `true` or to list all applied attributes.
`GetCustomAttribute`	Returns an instance of the specified attribute class that was applied to the target. The instance is constructed (deserialized) by using the parameters, fields, and properties specified during compilation. If the target has no instances of the specified attribute class, `null` is returned. If the target has multiple instances of the specified attribute applied to it, a `System.Reflection.AmbiguousMatchException` exception is thrown. This method is typically used with attributes that have `AllowMultiple` set to `false`.

If you just want to see if an attribute has been applied to a target, you should call `IsDefined` because it's more efficient than the other two methods. However, you know that when an attribute is applied to a target, you can specify parameters to the attribute's constructor and optionally set fields and properties. Using `IsDefined` won't construct an attribute object, call its constructor, or set its fields and properties.

If you want to construct an attribute object, you must call either `GetCustomAttributes` or `GetCustomAttribute`. Every time one of these methods is called, it constructs new instances of the specified attribute type and sets each of the instance's fields and properties based on the values specified in the source code. These methods return references to fully constructed instances of the applied attribute classes.

When you call any of these methods, internally, they must scan the managed module's metadata, performing string comparisons to locate the specified custom attribute class. Obviously, these operations take time. If you're performance conscious, you should consider caching the result of calling these methods rather than calling them repeatedly asking for the same information.

The `System.Reflection` namespace defines several classes that allow you to examine the contents of a module's metadata: `Assembly`, `Module`, `ParameterInfo`, `MemberInfo`, `Type`, `MethodInfo`, `ConstructorInfo`, `FieldInfo`, `EventInfo`, `PropertyInfo`, and their respective `*Builder` classes. All of these classes also offer `IsDefined` and `GetCustomAttributes` methods.

The version of `GetCustomAttributes` defined by the reflection classes returns an array of `Object` instances (`Object[]`) instead of an array of `Attribute` instances (`Attribute[]`). This is because the reflection classes are able to return objects of non–CLS-compliant attribute classes. You shouldn't be concerned about this inconsistency because non–CLS-compliant attributes are incredibly rare. In fact, in all of the time I've been working with the .NET Framework, I've never even seen one.

> **Note** Be aware that only `Attribute`, `Type`, and `MethodInfo` classes implement reflection methods that honor the Boolean `inherit` parameter. All other reflection methods that look up attributes ignore the `inherit` parameter and do not check the inheritance hierarchy. If you need to check the presence of an inherited attribute for events, properties, fields, constructors, or parameters, you must call one of `Attribute`'s methods.

There's one more thing you should be aware of: When you pass a class to `IsDefined`, `GetCustomAttribute`, or `GetCustomAttributes`, these methods search for the application of the attribute class you specify or any attribute class derived from the specified class. If your code is looking for a specific attribute class, you should perform an additional check on the returned value to ensure that what these methods returned is the exact class you're looking for. You might also want to consider defining your attribute class to be `sealed` to reduce potential confusion and eliminate this extra check.

Here's some sample code that lists all of the methods defined within a type and displays the attributes applied to each method. The code is for demonstration purposes; normally, you wouldn't apply these particular custom attributes to these targets as I've done here.

```
using System;
using System.Diagnostics;
using System.Reflection;


[assembly: CLSCompliant(true)]


[Serializable]
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name = "Jeff", Target = typeof(Program))]
public sealed class Program {
   [Conditional("Debug")]
   [Conditional("Release")]
   public void DoSomething() { }

   public Program() {
   }

   [CLSCompliant(true)]
   [STAThread]
   public static void Main() {
      // Show the set of attributes applied to this type
      ShowAttributes(typeof(Program));
      // Get the set of methods associated with the type
      var members =
            from m in typeof(Program).GetTypeInfo().DeclaredMembers.OfType<MethodBase>()
            where m.IsPublic
            select m;

      foreach (MemberInfo member in members) {
         // Show the set of attributes applied to this member
         ShowAttributes(member);
      }
   }

   private static void ShowAttributes(MemberInfo attributeTarget) {
      var attributes = attributeTarget.GetCustomAttributes<Attribute>();

      Console.WriteLine("Attributes applied to {0}: {1}",
         attributeTarget.Name, (attributes.Count() == 0 ? "None" : String.Empty));

      foreach (Attribute attribute in attributes) {
         // Display the type of each applied attribute
         Console.WriteLine("  {0}", attribute.GetType().ToString());

         if (attribute is DefaultMemberAttribute)
            Console.WriteLine("    MemberName={0}",
               ((DefaultMemberAttribute) attribute).MemberName);

         if (attribute is ConditionalAttribute)
            Console.WriteLine("    ConditionString={0}",
               ((ConditionalAttribute) attribute).ConditionString);

         if (attribute is CLSCompliantAttribute)
            Console.WriteLine("    IsCompliant={0}",
               ((CLSCompliantAttribute) attribute).IsCompliant);

         DebuggerDisplayAttribute dda = attribute as DebuggerDisplayAttribute;
         if (dda != null) {
            Console.WriteLine("    Value={0}, Name={1}, Target={2}",
               dda.Value, dda.Name, dda.Target);
         }
      }
      Console.WriteLine();
   }
}
```

Building and running this application yields the following output.

```
Attributes applied to Program:
  System.SerializableAttribute
  System.Diagnostics.DebuggerDisplayAttribute
    Value=Richter, Name=Jeff, Target=Program
  System.Reflection.DefaultMemberAttribute
    MemberName=Main
```

```
Attributes applied to DoSomething:
  System.Diagnostics.ConditionalAttribute
    ConditionString=Release
  System.Diagnostics.ConditionalAttribute
    ConditionString=Debug

Attributes applied to Main:
  System.CLSCompliantAttribute
    IsCompliant=True
  System.STAThreadAttribute

Attributes applied to .ctor: None
```

Matching Two Attribute Instances Against Each Other

Now that your code knows how to check whether an instance of an attribute is applied to a target, it might want to check the fields of the attribute to see what values they have. One way to do this is to write code that explicitly checks the values of the attribute class's fields. However, `System.Attribute` overrides `Object`'s `Equals` method, and internally, this method compares the types of the two objects. If they are not identical, `Equals` returns `false`. If the types are identical, then `Equals` uses reflection to compare the values of the two attribute objects' fields (by calling `Equals` for each field). If all the fields match, then `true` is returned; otherwise, `false` is returned. You might override `Equals` in your own attribute class to remove the use of reflection, improving performance.

`System.Attribute` also exposes a virtual `Match` method that you can override to provide richer semantics. The default implementation of `Match` simply calls `Equals` and returns its result. The following code demonstrates how to override `Equals` and `Match` (which returns `true` if one attribute represents a subset of the other) and then shows how `Match` is used.

```
using System;


[Flags]
internal enum Accounts {
   Savings   = 0x0001,
   Checking  = 0x0002,
   Brokerage = 0x0004
}


[AttributeUsage(AttributeTargets.Class)]
internal sealed class AccountsAttribute : Attribute {
   private Accounts m_accounts;

   public AccountsAttribute(Accounts accounts) {
      m_accounts = accounts;
   }


   public override Boolean Match(Object obj) {
      // If the base class implements Match and the base class
      // is not Attribute, then uncomment the following line.
      // if (!base.Match(obj)) return false;

      // Since 'this' isn't null, if obj is null,
      // then the objects can't match
      // NOTE: This line may be deleted if you trust
      // that the base type implemented Match correctly.
      if (obj == null) return false;

      // If the objects are of different types, they can't match
      // NOTE: This line may be deleted if you trust
      // that the base type implemented Match correctly.
      if (this.GetType() != obj.GetType()) return false;

      // Cast obj to our type to access fields. NOTE: This cast
      // can't fail since we know objects are of the same type
      AccountsAttribute other = (AccountsAttribute) obj;

      // Compare the fields as you see fit
      // This example checks if 'this' accounts is a subset
      // of others' accounts
      if ((other.m_accounts & m_accounts) != m_accounts)
         return false;

      return true;   // Objects match
   }


   public override Boolean Equals(Object obj) {
      // If the base class implements Equals, and the base class
      // is not Object, then uncomment the following line.
      // if (!base.Equals(obj)) return false;

      // Since 'this' isn't null, if obj is null,
      // then the objects can't be equal
      // NOTE: This line may be deleted if you trust
      // that the base type implemented Equals correctly.
      if (obj == null) return false;

      // If the objects are of different types, they can't be equal
      // NOTE: This line may be deleted if you trust
      // that the base type implemented Equals correctly.
      if (this.GetType() != obj.GetType()) return false;

      // Cast obj to our type to access fields. NOTE: This cast
      // can't fail since we know objects are of the same type
      AccountsAttribute other = (AccountsAttribute) obj;

      // Compare the fields to see if they have the same value
      // This example checks if 'this' accounts is the same
      // as other's accounts
      if (other.m_accounts != m_accounts)
         return false;
      return true;   // Objects are equal
   }


   // Override GetHashCode since we override Equals
   public override Int32 GetHashCode() {
      return (Int32) m_accounts;
   }
}


[Accounts(Accounts.Savings)]
internal sealed class ChildAccount { }


[Accounts(Accounts.Savings | Accounts.Checking | Accounts.Brokerage)]
internal sealed class AdultAccount { }


public sealed class Program {
   public static void Main() {
      CanWriteCheck(new ChildAccount());
      CanWriteCheck(new AdultAccount());

      // This just demonstrates that the method works correctly on a
      // type that doesn't have the AccountsAttribute applied to it.
      CanWriteCheck(new Program());
   }

   private static void CanWriteCheck(Object obj) {
      // Construct an instance of the attribute type and initialize it
      // to what we are explicitly looking for.
      Attribute checking = new AccountsAttribute(Accounts.Checking);

      // Construct the attribute instance that was applied to the type
      Attribute validAccounts =
         obj.GetType().GetCustomAttribute<AccountsAttribute>(false);

      // If the attribute was applied to the type AND the
      // attribute specifies the "Checking" account, then the
      // type can write a check
      if ((validAccounts != null) && checking.Match(validAccounts)) {
         Console.WriteLine("{0} types can write checks.", obj.GetType());
      } else {
         Console.WriteLine("{0} types can NOT write checks.", obj.GetType());
      }
   }
}
```

Building and running this application yields the following output.

```
ChildAccount types can NOT write checks.
AdultAccount types can write checks.
Program types can NOT write checks.
```

Detecting the Use of a Custom Attribute Without Creating Attribute-Derived Objects

In this section, I discuss an alternate technique for detecting custom attributes applied to a metadata entry. In some security-conscious scenarios, this alternate technique ensures that no code in an `Attribute`-derived class will execute. After all, when you call `Attribute`'s `GetCustomAttribute(s)` methods, internally, these methods call the attribute class's constructor and can also call property set accessor methods. In addition, the first access to a type causes the CLR to invoke the type's type constructor (if it exists). The constructor, set accessor, and type constructor methods could contain code that will execute whenever code is just looking for an attribute. This allows unknown code to run in the AppDomain, and this is a potential security vulnerability.

To discover attributes without allowing attribute class code to execute, you use the `System.Reflection.CustomAttributeData` class. This class defines one static method for retrieving the attributes associated with a target: `GetCustomAttributes`. This method has four overloads: one that takes an `Assembly`, one that takes a `Module`, one that takes a `ParameterInfo`, and one that takes a `MemberInfo`. This class is defined in the `System.Reflection` namespace, which is discussed in Chapter 23. Typically, you'll use the `CustomAttributeData` class to analyze attributes in metadata for an assembly that is loaded via `Assembly`'s static `ReflectionOnlyLoad` method (also discussed in Chapter 23). Briefly, `ReflectionOnlyLoad` loads an assembly in such a way that prevents the CLR from executing any code in it; this includes type constructors.

`CustomAttributeData`'s `GetCustomAttributes` method acts as a factory. That is, when you call it, it returns a collection of `CustomAttributeData` objects in an object of type `IList<CustomAttributeData>`. The collection contains one element per custom attribute applied to the specified target. For each `CustomAttributeData` object, you can query some read-only properties to determine how the attribute object would be constructed and initialized. Specifically, the `Constructor` property indicates which constructor method *would be* called, the `ConstructorArguments` property returns the arguments that *would be* passed to this constructor as an instance of `IList<CustomAttributeTypedArgument>`, and the `NamedArguments` property returns the fields/properties that *would be* set as an instance of `IList<CustomAttributeNamedArgument>`. Notice that I say "would be" in the previous sentences because the constructor and set accessor methods will not actually be called—we get the added security by preventing any attribute class methods from executing.

Here's a modified version of a previous code sample that uses the `CustomAttributeData` class to securely obtain the attributes applied to various targets.

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Collections.Generic;


[assembly: CLSCompliant(true)]


[Serializable]
```

```
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name="Jeff", Target=typeof(Program))]
public sealed class Program {
   [Conditional("Debug")]
   [Conditional("Release")]
   public void DoSomething() { }

   public Program() {
   }

   [CLSCompliant(true)]
   [STAThread]
   public static void Main() {
      // Show the set of attributes applied to this type
      ShowAttributes(typeof(Program));

      // Get the set of methods associated with the type
      MemberInfo[] members = typeof(Program).FindMembers(
         MemberTypes.Constructor | MemberTypes.Method,
         BindingFlags.DeclaredOnly | BindingFlags.Instance |
         BindingFlags.Public | BindingFlags.Static,
         Type.FilterName, "*");

      foreach (MemberInfo member in members) {
         // Show the set of attributes applied to this member
         ShowAttributes(member);
      }
   }

   private static void ShowAttributes(MemberInfo attributeTarget) {
      IList<CustomAttributeData> attributes =
         CustomAttributeData.GetCustomAttributes(attributeTarget);

      Console.WriteLine("Attributes applied to {0}: {1}",
         attributeTarget.Name, (attributes.Count == 0 ? "None" : String.Empty));

      foreach (CustomAttributeData attribute in attributes) {
         // Display the type of each applied attribute
         Type t = attribute.Constructor.DeclaringType;
         Console.WriteLine("  {0}", t.ToString());
         Console.WriteLine("    Constructor called={0}", attribute.Constructor);

         IList<CustomAttributeTypedArgument> posArgs = attribute.ConstructorArguments;
         Console.WriteLine("    Positional arguments passed to constructor:" +
            ((posArgs.Count == 0) ? " None" : String.Empty));
         foreach (CustomAttributeTypedArgument pa in posArgs) {
             Console.WriteLine("      Type={0}, Value={1}", pa.ArgumentType, pa.Value);
         }


         IList<CustomAttributeNamedArgument> namedArgs = attribute.NamedArguments;
         Console.WriteLine("    Named arguments set after construction:" +
            ((namedArgs.Count == 0) ? " None" : String.Empty));
         foreach(CustomAttributeNamedArgument na in namedArgs) {
            Console.WriteLine("     Name={0}, Type={1}, Value={2}",
               na.MemberInfo.Name, na.TypedValue.ArgumentType, na.TypedValue.Value);
         }
         Console.WriteLine();
      }
      Console.WriteLine();
   }
}
```

Building and running this application yields the following output.

```
Attributes applied to Program:
  System.SerializableAttribute
    Constructor called=Void .ctor()
    Positional arguments passed to constructor: None
    Named arguments set after construction: None

  System.Diagnostics.DebuggerDisplayAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
      Type=System.String, Value=Richter
    Named arguments set after construction:
     Name=Name, Type=System.String, Value=Jeff
     Name=Target, Type=System.Type, Value=Program

  System.Reflection.DefaultMemberAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
      Type=System.String, Value=Main
    Named arguments set after construction: None


Attributes applied to DoSomething:
  System.Diagnostics.ConditionalAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
      Type=System.String, Value=Release
    Named arguments set after construction: None

  System.Diagnostics.ConditionalAttribute
    Constructor called=Void .ctor(System.String)
    Positional arguments passed to constructor:
      Type=System.String, Value=Debug
    Named arguments set after construction: None


Attributes applied to Main:
  System.CLSCompliantAttribute
    Constructor called=Void .ctor(Boolean)
    Positional arguments passed to constructor:
      Type=System.Boolean, Value=True
    Named arguments set after construction: None

  System.STAThreadAttribute
    Constructor called=Void .ctor()
    Positional arguments passed to constructor: None
    Named arguments set after construction: None


Attributes applied to .ctor: None
```

Conditional Attribute Classes

Over time, the ease of defining, applying, and reflecting over attributes has caused developers to use them more and more. Using attributes is also a very easy way to annotate your code while simultaneously implementing rich features. Lately, developers have been using attributes to assist them with design time and debugging. For example, the Microsoft Visual Studio code analysis tool (FxCopCmd.exe) offers a `System.Diagnostics.CodeAnalysis.SuppressMessageAttribute` that you can apply to types and members in order to suppress the reporting of a specific static analysis tool rule violation. This attribute is only looked for by the code analysis utility; the attribute is never looked for when the program is running normally. When not using code analysis, having `SuppressMessage` attributes sitting in the metadata just bloats the metadata, which makes your file bigger, increases your process's working set, and hurts your application's performance. It would be great if there were an easy way to have the compiler emit the `SuppressMessage` attributes only when you intend to use the code analysis tool. Fortunately, there is a way to do this by using conditional attribute classes.

An attribute class that has the `System.Diagnostics.ConditionalAttribute` applied to it is called a *conditional attribute class*. Here is an example.

```
//#define TEST
#define VERIFY

using System;
using System.Diagnostics;


[Conditional("TEST")][Conditional("VERIFY")]
public sealed class CondAttribute : Attribute {
}

[Cond]
public sealed class Program {
   public static void Main() {
      Console.WriteLine("CondAttribute is {0}applied to Program type.",
         Attribute.IsDefined(typeof(Program),
            typeof(CondAttribute)) ? "" : "not ");
   }
}
```

When a compiler sees an instance of the `CondAttribute` being applied to a target, the compiler will emit the attribute information into the metadata only if the `TEST` or `VERIFY` symbol is defined when the code containing the target is compiled. However, the attribute class definition metadata and implementation is still present in the assembly.