Reflection and Attributes

Topics in This Chapter

- *Reflection:* Each .NET assembly contains tables of metadata that describe the code contained within the assembly. A program can read and act upon this metadata—a process know as reflection—by using the members in the `System.Reflection` namespace. Reflection is invaluable to any application or component whose behavior depends upon the runtime capability to identify the types and type members in an assembly's code. Reflection enables a developer to create applications or components that can dynamically adjust their behavior by examining metadata rather than changing program code.
- *Attributes*: Attributes provide a way to associate information with an assembly or just about any element in the assembly. These include a class, class members, delegates, enums, structs, interfaces, parameters, and other attributes. The attribute information is stored in a module as metadata that can be read using the reflection API. The .NET Framework comes with a wide range of predefined attributes that are used for a variety of purposes including object serialization, code access security, and interoperability with Win32 code. More importantly, a developer can create and add custom attributes to extend the metadata generated during compilation.

Chapter 19

In his classic book on program development, *Algorithms + Data Structures = Programs*[1], Niklaus Wirth described programs as "concrete formulations of abstract *algorithms* based on particular representations and structures of *data.*" In other words, programs were regarded as a combination of data and program statements. The data might be represented as a simple variable or complex data structure; the program statements provided the code logic and means to access the data. As developers began to create tools to assist debugging and software development, the division between what was data and what was code began to blur. New applications required that developers answer questions such as what classes a program or component contains, what methods or functions are available, and what parameters are accepted by these functions.

The solution was to treat program code as another form of data, and provide a means of querying that data. This gave rise to compilers—notably Smalltalk and Java—that generated `metadata` as a way to index and describe compiled code. To access this metadata, the languages included an API that enabled them to *reflect* on code. A program could now inspect its own code or code in another program at runtime, and create objects from types in that code.

C# and .NET support this reflection model. During the compilation process, the C# compiler writes metadata into the assembly (actually a module in the assembly) as tables that describe the types and attributes in the code. Special reflection classes in the .NET Framework provide programmatic access to this metadata.

1. Niklaus Wirth, *Algorithms + Data Structures = Programs* (Englewood Cliffs, NJ: Prentice-Hall, 1976).

Section 19.1 of this chapter describes how to use types in the `System.Reflection` namespace to create *reflective programs* that not only query metadata, but also use it to construct objects and invoke methods at runtime. Examples demonstrate how to view all the types in an assembly, enumerate the members of a type, and dynamically identify and access members in a type using *late binding*.

Section 19.2 discusses attributes—classes that annotate an assembly, type, or type members with information that is emitted as metadata, and can be used to affect the calling sequence or other behavior of an application. The discussion includes how to create a custom attribute and how to use reflection to identify an attribute and extract its value(s) at runtime. As you will discover, attributes are like wine—an acquired a taste. They are not needed on all occasions, but when they are, they're indispensable.

19.1 Reflection

In general terms, reflection is the ability of a system to reason and act upon itself. More specifically, .NET supports reflection as a mechanism for reading an assembly's metadata in order to glean information about types within the assembly. It has many uses. For example, VS.NET uses it to implement Intellisense; and .NET object serialization relies on reflection to identify whether a type can be serialized (by looking for the `SerializableAttribute`) and which of its members are serializable.

While reflection is a powerful technique, it's not difficult to understand. Think of the reflection API as a set of methods that can be applied at each level of the .NET coding hierarchy. The `Assembly.GetTypes` method yields all the types in an assembly as `System.Type` objects. The `System.Type` class, in turn, exposes methods that return a specific type's properties, methods, constructors, events, and fields. At an even more granular level, you can then examine the parameters of a method or constructor.

Before a type (class, struct, array, enum, generic types, value types, or interface) can be reflected, a `Type` reference must be acquired for the type. This reference is the key that unlocks the door to the .NET reflection API.

The System.Type class

`System.Type` is an abstract class whose members provide the primary means of accessing the metadata that describes a type.

Syntax:

```
public abstract class Type : MemberInfo, _Type, IReflect
```

The Type class inherits from the MemberInfo class and two interfaces—_Type and IReflect. Of these, MemberInfo is of particular interest. It serves as the base class for some of the most important classes in the reflection namespace, and defines many of the methods and properties that provide information about a type's members and attributes. We'll look at it in detail shortly.

Creating a Type Object

In order to use reflection to examine a type's metadata, it is first necessary to obtain a reference to a Type object that is associated with the type to be reflected. There are several ways to obtain a Type object reference. The choice varies depending on whether the object is being created from a type's definition or an instance of the type. To illustrate the first case, consider the following code segment that lists the methods for the System.Object class.

```
// (1) Get Type for System.Object definition
Type obType = typeof(System.Object);
// Alternative: Type obType= Type.GetType("System.Object");
// (2) Call a method to return list of methods for Object
MethodInfo[] mi = obType.GetMethods();
for(int i=0;i<mi.Length;i++)
{
    MethodInfo myMethodInfo = (MethodInfo)mi[i];
    Console.WriteLine("Method name: {0}({1})",
        myMethodInfo.Name, myMethodInfo.ReturnType);
    /* Output:
           Method Name: GetHashCode (System.Int32)
           Method Name: Equals (System.Boolean)
           (remaining methods listed here)
    */
}
```

This example uses the C# typeof operator to obtain a Type reference. Alternatively, the Type.GetType method, which accepts a type's name as a string value, could be used. Once the Type object is obtained, the Type.GetMethods method is called to return an array of MethodInfo objects that represent the methods for the type. The Name and ReturnType properties of the MethodInfo class expose the name and return type of each method.

To acquire a Type object reference for an instance of a type—rather than its definition—use the GetType method that all types inherit from System.Object:

```
Device myDevice = new Device();   // Create class instance
Type t = myDevice.GetType();      // GetType reference
// Reflection can also be used on value types
int[] ages = {22,43,55};
```

```
Type ageType= ages.GetType();
Console.WriteLine(ageType.Name);  // Output: Int32[]
```

Later in this section, we'll see how to use the `Assembly.GetTypes` and `Module.GetTypes` methods to obtain `Type` object references for types in an assembly and module, respectively.

System.Type Members

Once a `Type` object is created, the fun begins. You can use `Type` properties and methods to inspect the type's constructors, methods, attributes, fields, interfaces, and events. Table 19-1 lists some of the more useful members of the `Type` class. Examples of their use are provided throughout this chapter.

Table 19-1 Selected Members of the `Type` Class

Member	Description
`FindInterfaces( TypeFilter filtdeleg, object criteria)`	Returns a `Type` array containing the interfaces implemented or inherited by the current type. The first parameter is a delegate that specifies a method that is called to compare the interfaces against some criteria.
`GetConstructors()` `GetConstructors( BindingFlags bf)`	Returns a type's constructors as a `ConstructorInfo[]` array.
`GetCustomAttributes( Type attribtype, bool inherit)`	Returns attributes associated with a type as an `object` array. The method has two overloads. The optional first parameter specifies the attribute type that a returned attribute must have. The second parameter indicates whether the type's inheritance chain is to be searched for attributes.
`GetEvents()` `GetEvents( BindingFlags bf)`	Returns a type's events as an `EventInfo[]` array.
`GetInterfaces()`	Returns the interfaces inherited by a type as a `Type[]` array.
`GetMembers()` `GetMembers( BindingFlags bf)`	Returns the members of the current type as a `MemberInfo[]` array.

Table 19-1 Selected Members of the `Type` Class *(continued)*

Member	Description

`GetMethods()`
`GetMethods(`
`  BindingFlags bf)`
	Returns the methods of a type as a `MethodInfo[]` array.

`GetFields()`
`GetFields(`
`  BindingFlags bf)`
	Returns the fields for the class or `struct` as a `FieldInfo[]` array.

`GetGenericArguments()`	Returns the type parameters of a generic type as a `Type[]` array. (2.0 only)

`GetProperties()`
`GetProperties(`
`  BindingFlags bf)`
	Returns the public properties as a `PropertyInfo[]` array.

`GetType(string typename,`
`  bool throwexcept,`
`  bool ignorecase)`
	Gets a reference to a `Type` object for the type whose name is specified as the first parameter. The final two parameters are optional.

`BaseType`	Returns the type which the `Type` inherits from. Interface inheritance is ignored.
`IsSubclassOf(Type t)`	Determines whether the current type derives from a specified type.

`IsArray`
`IsByRef`
`IsClass`
`IsEnum`
`IsGenericType`
`IsInterface`
`IsSealed`
`IsSerializable`
	Boolean property that returns `true` if the type is that specified by the property. Note this is not an exhaustive list.

For each method that returns an array of types or members, there is a corresponding method (same name without the final "s") that searches for a single member or value. For example, to return a `Type` reference to the `IDictionary` interface, you could use:

```
Type hType = Type.GetType("System.Collections.Hashtable");
// Return null if interface no inherited
Type inType = hType.GetInterface("IDictionary");
```

Core Note

Most of the `Type` methods include an overload with a `BindingFlags` parameter. `BindingFlags` is an enumeration that includes values that can be used to filter the reflection search for members and types. Some of its more useful members include `NonPublic`, `Public`, `Static`, and `DeclaredOnly` (not inherited).

The System.Reflection Namespace

From Table 19-1 you can see that the `Type.Getxxx` methods return arrays of objects created from classes defined in the reflection namespace. For example, `GetFields()` returns a `FieldInfo` array representing a type's fields, and `GetMethods()` returns a `MethodInfo` array. All of the reflection classes that identify and provide information about type members are derived from the `System.Reflection.MemberInfo` class. Figure 19-1 shows the class hierarchy.

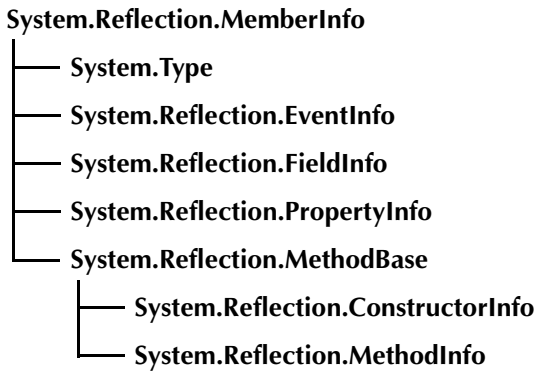

Figure 19-1 Inheritance hierarchy of `MemberInfo` classes

It may seem a bit odd that `System.Type` derives from `MemberInfo`; after all, a type contains members. However, recall that a `Form` inherits from a `Control` although a `Form` contains controls. In both cases the hierarchy is one of inheritance, not containment.

The two base classes in Figure 19-1, `MemberInfo` and `MethodBase`, define members that play a crucial role in reflection for all their derived types. Here are some of the members you will find most useful:

MemberInfo

- `Name`—Returns the name of the member.
- `Module`—Returns the `module` in which the member is declared.
- `GetCustomAttributes`—Method that returns the attributes attached to a member.

MethodBase

- `IsAbstract`, `IsGenericMethod`, `IsStatic`, `IsVirtual`—Properties that describe the method.
- `GetGenericArguments`—Returns the type arguments of a generic method.
- `GetParameters`—Returns the parameters of the method or constructor.
- `Invoke`—Invokes a method or constructor. (See "Reflection and Late Binding" on page 24.)

Examples of Using Reflection

Now that that the key players in reflection have been identified—`System.Type` and the `MemberInfo` derived classes—let's look at them in action. The following code examples illustrate how reflection is used to examine interfaces, constructors, methods, attributes, and generic types.

GetInterfaces and FindInterfaces

The easiest way to retrieve all of the interfaces inherited by a type is to use the `GetInterfaces` method. It takes no parameters and returns the interfaces in a `Type` array. To filter the interfaces that are returned, use the `FindInterfaces` method. It takes two parameters—a `TypeFilter` delegate and an `object`. The delegate specifies a callback method that is called for each interface reflected. The callback method implements logic that returns a `true` value if an interface is to be included in the returned list, and `false` if it's to be excluded. The second parameter to `FindInterfaces` is an object that includes an interface list used by the callback method to filter interfaces. This object is usually a `String` array element.

Here is an example that uses `GetInterfaces` and `FindInterfaces` to list interfaces found in the `System.Hashtable` class. The most interesting part of the code is the callback method, `MyInterfaceFilter`, used by `FindInterfaces`. The code illustrates how to pass it the name of a specific interface to be used as a filter. Note that if the objective is to retrieve only one interface, a better approach is to use the `GetInterface` method.

```
public static void Main() {
// List interfaces for the Hashtable class
Type hashType = typeof(System.Collections.Hashtable);
// ... GetInterfaces()
foreach(Type ti in hashType.GetInterfaces())
   Console.WriteLine(ti.Name);

// ... FindInterfaces
// Define delegate to call method to compare
// interfaces and filter
TypeFilter myFilter = new TypeFilter(MyInterfaceFilter);
// Filter to specify interfaces to search for
String[] myIFaceList = new String[2]{
             "System.Collections.IEnumerable",
             "System.Collections.IDictionary"};
//Type[] interfaces = hashType.FindInterfaces(myFilter, null);
// Second parameter specifies IDictionary interface
Type[] interfaces = hashType.FindInterfaces(myFilter,
                                          myIFaceList[1]);
for (int i=0; i<interfaces.Length;i++)
   Console.WriteLine(interfaces[i].Name);
}

// Callback method to compare interfaces against a filter
// Returns true if the interface is to be included in the list
public static bool MyInterfaceFilter(Type typeObj,
                                     Object criteria)
{
   // typeObj - Type object against which filter is compared
   if(criteria==null) return true;

   // Only accept interface specified in criteria
   if(typeObj.ToString() == criteria.ToString())
      return true;   else  return false;
}
```

GetConstructors

This example collects all of the constructors for a `Hashtable` and then uses the `ConstructorInfo.GetParameters` method to return the constructors' parameters.

```
Type hashType = typeof(System.Collections.Hashtable);
// ... Retrieve Constructors
ConstructorInfo[] ci = hashType.GetConstructors();
foreach (ConstructorInfo construc in ci)
{
   Console.WriteLine(construc.Name);  // Always returns .ctor
```

```
    // Display parameters and their type for the constructor
    ParameterInfo[] pi= construc.GetParameters();
    Console.Write("(");
    // Display the type and name of each parameter
    foreach(ParameterInfo p in pi)
        Console.Write(" {0} {1} ",  p.ParameterType, p.Name);
    Console.Write(")\r\n");
    // Example: ( System.Int32 capacity )
}
```

By default, `GetConstructors` returns only `Public` and `Instance` constructors. To include `Static` constructors as well, use the overloaded form of `GetConstructors` that accepts `BindingFlags` parameters:

```
ConstructorInfo[] ci = hashType.GetConstructors(
     BindingFlags.Public | BindingFlags.NonPublic
     BindingFlags.Static | BindingFlags.Instance);
```

GetMethods

`GetMethods` returns an array of `MethodInfo` types that—like `ConstructorInfo`—derives from the `MethodBase` class. Two of the most useful members it inherits from this class are the `Name` property and the `GetParameters` method. To these, it adds a `ReturnType` property that exposes a method's return type as a `Type`. The following code lists the name, return type, and class in which the method is declared for all `Hashtable` methods:

```
MethodInfo[] mi = hashType.GetMethods();
foreach(MethodInfo meth in mi)
    Console.WriteLine("{0}{1}({2}) IsStatic:{3}",
           meth.ReturnType, meth.Name, meth.DeclaringType,
           meth.IsStatic);
/* Sample Output:
   System.String ToString (System.Object)IsStatic:false
   System.Void Remove (System.Collections.Hashtable)
           IsStatic:false
*/
```

Once you have identified a type's methods, the next step is to examine its parameters. As discussed earlier, `MethodInfo` inherits from `MethodBase` the `GetParameters` method that is used to return a method's parameters as a `ParmaterInfo` array. To illustrate, let's use reflection to list the method and parameters for a sample class. To make it interesting, we'll define a method on the class that accepts both a value and reference parameter.

```
public class TestClass
{
    public static bool GetCapital(string state, out string city)
    {
        // Code to determine capital for state goes here ...
    }
}
```

This code to reflect on `TestClass` creates a `Type` reference for `TestClass`, enumerates its method(s) (`BindingFlags` ensures that only the public static methods are retrieved), and lists the type and name of each parameter for a reflected method.

```
// Retrieve public static method(s) for TestClass
MethodInfo[] methTypes = typeof(TestClass).GetMethods(
    BindingFlags.Public | BindingFlags.Static);
foreach(MethodInfo mi in methTypes)
{
    Console.WriteLine("Method: {0}",mi.Name);
    ParameterInfo[] methParms = mi.GetParameters();
    foreach(ParameterInfo pi in methParms)
        Console.WriteLine("Parameter: {0} {1}",
            pi.ParameterType.ToString(), pi.Name);
}
/*  Output:
       Method: GetCapital
       Parameter: System.String state
       Parameter: System.String& city
*/
```

Observe how an ampersand (`&`) is appended to the parameter type when the parameter is passed by reference. This convention applies to both `ref` and `out` parameters.

Core Note

`Type.GetMethods` returns all public methods defined on a type. This includes the `get` and `set` accessors for properties, which are implemented internally as methods. Use the `PropertyInfo.GetAccessors` method to return the accessor(s) for a property:

```
PropertyInfo pi= myType.GetProperty("age");
MethodInfo[] pAccessors = pi.GetAccessors();
```

GetGenericArguments

.NET 2.0 introduced generics as a way to create type-safe classes, structs, and interfaces. A generic type differs from a non-generic in that its syntax includes one or more *type parameters* that are replaced by an actual type when an object is created. The `GetGenericArguments` method returns an array of `Types` that correspond to each type parameter. Reflection can then be used to further examine the returned parameter `Type`.

To illustrate how reflection is used with generic types let's create a simple generic class that has one type parameter. This class (see Chapter 3, "Class Design in C#" for the entire class) serves as a stack to which items can be added and removed. When the class is instantiated, the type parameter `T` is replaced with a user-specified type. Only items of this type can be placed on the stack.

```
public class GenStack<T>
    where T:struct   // Accept only value type for type parameter
{
    private T[] stackCollection;
    private int count = 0;
    public GenStack(int size)  // Constructor
    {  stackCollection = new T[size];  }

    public void Add( T item)
    {
        stackCollection[count] = item;  // Add item to collection
        count += 1;
    }
    // ... Remaining code to return values
}
```

Listing 19-1 shows how reflection is used to examine the `GenStack` class. The logic is simple, but provides some insight into how generics work. Two `Type` objects are created: one from the `GenStack` definition and one from an instance of `GenStack` (referred to as a *closed constructed* type). Both types are passed to the `ShowGeneric` method, which uses reflection to display information about their type parameter.

Listing 19-1 Using Reflection to Examine a Generic Class

```
using System;
using System.Reflection;
public class Example
{
```

Listing 19-1 Using Reflection to Examine a Generic Class *(continued)*

```
    public static void Main()
    {
        //  (1) To get the generic type definition, omit the type
        Type def = typeof(GenStack<>);
        ShowGeneric(def);

        // (2) Create object and examine it
        GenStack<int> gs = new GenStack<int>(10);
        Type gsType = gs.GetType();
        ShowGeneric(gsType);
        string instr = Console.ReadLine();
    }
}

private static void ShowGeneric(Type genType)
{
    Console.WriteLine("\r\nExamining generic type {0}",
                      genType);
    Console.WriteLine("Is generic definition: {0}",
                      genType.IsGenericTypeDefinition);
    // Get list of Type parameters
    Type[] genTypeargs =genType.GetGenericArguments();
    foreach (Type tp in genTypeargs)
    {
        Console.WriteLine("\r\nType parameter: {0}", tp.Name);
        if(tp.IsGenericParameter)
        {
            //  List any constraints for the type parameters
            Type[] tpConstraints =
                tp.GetGenericParameterConstraints();
            foreach (Type tpc in tpConstraints)
                Console.WriteLine("\tConstraint: {0}", tpc);
        }
        else {Console.WriteLine(
               "\tConstraint unavailable.
               Requires type definition ");
             }
        // List method in generic class
        MethodInfo[] mi = genType.GetMethods();
        Console.WriteLine("Method Name: {0}", mi[0].Name);
    }
}
```

Listing 19-1 Using Reflection to Examine a Generic Class *(continued)*

```
/*  Output:
Examining generic type GenStack`1[T]
Is generic definition: true
Type parameter: T
       Constraint: System.ValueType

Examining generic type: GenStack`1
Is generic definition: false
Type parameter: Int32
       Constraint unavailable. Requires type definition
Method Name: Add
*/
```

The most interesting part of this code is in the `ShowGeneric` method that accepts a `Type` argument and displays information about it using the `GetGenericArguments`, `IsGenericParameter`, and `GetGenericParameterConstraints` `Type` members.

First, `GetGenericArguments` is called to return the type parameter for the `GenStack` class. Next, the name of the parameter is displayed: `T` (the placeholder name) for the `GenStack` definition and `Int32` (the type used in creating the object) for the `GenStack` object. The `IsGenericParameter` property then returns `true` if the type parameter is for a class definition, and `false` if for an object. Finally, the `GetGenericParameterConstraint` method is called to identify any constraint placed on the type parameter. In this example, `GenStack` has a `struct` constraint, which means that type parameter must be a value type (`class` is used to specify reference type). Observe in the output that the constraint value is available only for the definition of a generic type—not an instance of it (called a *closed constructed* type). In fact, an exception occurs if you execute `GetGenericParameterConstraints` on a type parameter for which `IsGenericParameter` is `false`.

GetCustomAttributes

Attributes may be attached to an assembly, types in the assembly, or type members. The `GetCustomAttributes` method defined on the `MethodInfo` class, provides a way to access both .NET predefined and user-defined attributes. To demonstrate, let's first create a simple class whose attributes we wish to examine through reflection. The class shown here includes .NET attributes attached to a class, field, and method.

```
// csc /t:library /out:attribs.dll attribs.cs
using System;
using System.Reflection;
```

```
[Serializable]
public class TestClass
{
    public int age;
    [NonSerialized]
    public string FullName;

    [System.Diagnostics.Conditional("DEBUG")]
    public void Aboutme()
    {  Console.WriteLine("My name is TestClass");  }
}
```

Listing 19-2 shows the code that reflects on `TestClass`. It loads the assembly, invokes `Assembly.GetTypes()` to return all the types in the assembly (a single class in this case), and then examines each type and its members for attributes.

Listing 19-2 Using Reflection to Examine Attributes

```
using System;
using System.Reflection;
using System.Diagnostics;
public class myApp
{
    public static void Main()
    {
        // Load an assembly to reflect on
        Assembly assembly = Assembly.LoadFrom(
               "\\corecsharp\\chapter19\\attribs.dll");
        Type[] types = assembly.GetTypes();
        // (1) Loop through types in assembly (one in this case)
        foreach(Type t in types)
        {
            Object[] myAttributes = t.GetCustomAttributes(true);
            // List attributes for types in assembly
            for(int j = 0; j < myAttributes.Length; j++)
                Console.WriteLine("Attribute for {0}: {1}",
                                  t.Name, myAttributes[j]);
            MemberInfo[] mi = t.GetMembers();
            // List attributes for type members
            foreach(MemberInfo m in mi)
            {
                // (true) indicates to check inheritance chain
                Object[] memAttrib = m.GetCustomAttributes(true);
                foreach(object ma in memAttrib)
                {
```

Listing 19-2 Using Reflection to Examine Attributes *(continued)*

```
                Console.WriteLine("Attribute for {0}: {1}",
                                  m.Name,ma);
                // If this is a conditional attribute,
                // print its value
                if(ma is ConditionalAttribute)
                {
                    ConditionalAttribute ca=
                                     (ConditionalAttribute)ma;
                    Console.WriteLine("\t Value is: {0}",
                                      ca.ConditionString);
                }
            }
        }
    }
    int rd = Console.Read();
}
/* output:
   Attribute for TestClass: System.SerializableAttribute
   Attribute for Aboutme:
       System.Diagnostics.ConditonalAttribute
       Value is: DEBUG
   Attribute for FullName: System.NonSerializedAttribute
*/
}
```

Note how the program checks to determine if an attribute is a `ConditionalAttribute`; if so, it prints the attribute's value exposed by the `ConditionString` property.

Using Reflection with an Assembly

In many cases, the first step for an application that uses reflection is to select or load the assembly whose types it wishes to inspect. This was demonstrated in the preceding example (Listing 19-2), where the `attribs.dll` assembly was loaded using the `Assembly.LoadFrom` method. Alternatively, an `Asembly.Load` method can be used to explicitly load an assembly. In other situations the assembly may already be loaded. For example, you may wish to reflect on the currently executing assembly or all of the currently loaded assemblies. Let's look at some of the techniques used to reference an assembly.

Loading an Assembly

An assembly can be loaded implicitly—when a member of the assembly is referenced—or explicitly using `Assembly` methods designed for that purpose. You have two choices when it comes to obtaining a reference to loaded assemblies: the `Assembly.GetExecutingAssembly` method references the current assembly, and `AppDomain.CurrentDomain.GetAssemblies` returns an array with references to all assemblies loaded in the current `AppDomain`. Application domains are discussed in Chapter 14, "Creating Distributed Applications with Remoting."

The following code segment illustrates the two methods.

```
// (1) Reference current executing assembly
Assembly curr = System.Reflection.Assembly.GetExecutingAssembly();
// (2) Reference assemblies that reside in current AppDomain
Assembly[] myAssemblies = AppDomain.CurrentDomain.GetAssemblies();
    for(int i = 0; i < myAssemblies.Length; i++)
        Console.WriteLine(myAssemblies[i].FullName);
```

In order to explicitly load an assembly whose file path is known, you can use the `Assembly.LoadFrom` method. Its overloads include:

```
public static AssemblyLoadFrom( string file)
public static AssemblyLoadFrom( string file, Evidence ev)
```

file Name or path of file (including file extension) containing the assembly. For a multi-file assembly, it is the file that contains the manifest.

ev An `Evidence` object that provides security information such as a signature or code origin for the calling assembly. (See Chapter 15 "Application Deployment Considerations" for a discussion on code access security.)

`LoadFrom()` is most often used with private assemblies, as the first example in this code segment demonstrates. However, it can also be used to load shared assemblies by supplying the path to the current Common Language Runtime (CLR). This is demonstrated in the second example that loads the `System.Drawing` assembly.

```
// Specify path of assembly
Assembly asmAttr =
    Assembly.LoadFrom(@"c:\components\attribs.dll");
// Get directory where Common Language Runtime stored
// Requires System.Runtime.InteropServices namespace
clrDir = RuntimeEnvironment.GetRuntimeDirectory();
// Loads the System.Drawing assembly
Assembly asmDraw =
    Assembly.LoadFrom(clrDir+"System.Drawing.dll");
```

When `LoadFrom` is called, the CLR locates the file (an exception is thrown if it cannot be found), extracts its identity information—or display name—and then internally calls `Assembly.Load` by passing it the assembly's identity. For a strong named assembly, the CLR first searches the Global Assembly Cache (GAC). If not found, it then searches the application's base directory and private directories. The GAC is not searched for a weakly named assembly.

`Assembly.Load` contains several overloaded forms. Here are the more commonly used ones:

```
public static Assembly Load(string asmString)
public static Assembly Load(string asmString, Evidence ev)
public static Assembly Load(AssemblyName asmName)
public static Assembly Load(AssemblyName asmName, Evidence ev)
```

asmString	The display name of the assembly: simple name, version, culture, and public key token.
asmName	Instance of the `AssemblyName` class that describes the assembly.
ev	An `Evidence` object that provides code access security information.

The following code segment illustrates how to explicitly use `Assembly.Load` to load both weakly and strong named assemblies:

```
// Load weakly named assembly
Assembly a1 = Assembly.Load("attribs");
Assembly a1 = Assembly.Load("attribs, Version=0.0.0.0,
          Culture=neutral, PublicKeyToken=null");

// Load shared assembly (strong named assembly)
Assembly a2 = Assembly.Load("System.Web.Mobile,
          Version=1.0.5000.0, Culture=neutral,
          PublicKeyToken=b03f5f7f11d50a3a");

// Create AssemblyName object
AssemblyName an = new AssemblyName();
an.Name= "attribs"; // Only name needed for weakly named assembly
Assembly a1 = Assembly.Load(an);
```

The primary difficulty with using `Assembly.Load` is providing the display name information, particularly the `Version` and `PublicKeyToken`. Fortunately, you can acquire this information using the Global Assembly Cache tool (`gacutil.exe`). Run `gacutil -l` from the command line to list the identity of assemblies in the GAC. To load one of these assemblies, get its identity information from the utility output, and provide it to the `Assembly.Load` method.

Where possible, it is recommended that `Load` be called directly, rather than indirectly through `LoadFrom`. There are a couple of reasons for this. The most obvious is that `Load` runs faster since it eliminates an extra step. More importantly, `LoadFrom` can produce unexpected results. If assembly A is loaded and you attempt to load an assembly from another path that has the same identity, the second assembly will not be loaded. Instead, assembly A will be used.

Loading an Assembly for Reflection-Only

.NET version 2.0 added a couple of useful static methods to the `Assembly` class, `ReflectionOnlyLoad` and `ReflectionOnlyLoadFrom`, that allow assemblies to be loaded into a *reflection-only* context. This is useful for loading assemblies that were created with a different version of the .NET Framework than is currently active. However, there is a significant restriction on these reflection-only loaded assemblies: the metadata can be read, but the code cannot be used to create executable objects.

The following code segment shows how to load an assembly compiled against .NET 1.1 and an assembly created with .NET 2.0—the active version.

```
// Load earlier .NET version of assembly for reflection-only
Assembly oldAssembly =
    Assembly.ReflectionOnlyLoadFrom(@"c:\v1\attribs.dll");
// Load assembly version created with current .NET
Assembly assembly = Assembly.LoadFrom("c:\v2\attribs.dll");
Console.WriteLine(oldAssembly.GetName()\r\nassembly.GetName());
Console.WriteLine("{0}\r\n{1}",oldAssembly.GetName(),
                  assembly.GetName());
/* output:
   attribs, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
   attribs, Version=2.0.0.0, Culture=neutral, PublicKeyToken=null
*/
```

Note that any assembly loaded for reflection-only does not appear in a list of loaded assemblies for the `AppDomain`.

Inspecting Attributes in a Reflection-Only Assembly

One of the drawbacks of examining code in a reflection-only context, is that you cannot use the `GetCustomAttributes` method described earlier to inspect attributes. The reason is that `GetCustomAttributes` (see Table 19-1 on page 6) returns an array of objects—and objects cannot be created from an `Attribute` class in reflection-only code.

To provide access to custom attributes loaded in a reflection-only context, .NET 2.0 introduced the `CustomAttributeData` class. Its overloaded `GetCustomAttributes` method returns a list of `CustomAttributeData` objects that represent attributes for a specified target. As shown here, the attributes are returned as a `System.Collections.Generic.IList` collection of objects:

```
public static IList <CustomAttributeData> GetCustomAttributes (
    MemberInfo target)
```

Overloaded forms of this method allow the *`target`* parameter to be specified as a `Module`, `ParameterInfo`, or `Assembly` type.

In addition to the `GetCustomAttributes` method, `CustomAttributeData` exposes three important properties that enable a program to inspect an attribute's constructor:

1. `Constructor`—Returns a `ConstructorInfo` object that represents the attribute's constructor.

2. `ConstructorArguments`—Returns a list of `CustomAttributeTypedArgument` structures that represent the attribute's *positional* arguments. Note that positional arguments correspond to parameters defined in the attribute's constructor.

   ```
   public IList<CustomAttributeTypedArgument>
      ConstructorArguments
      { get; }
   ```

3. `NamedArguments`—Returns a list of `CustomAttributeNamedArgument` structures that represent the attribute's *named* arguments. Named arguments are not defined in the attribute's constructor; instead, they are implemented as properties and fields on the attribute.

   ```
   public IList<CustomAttributeNamedArgument> NamedArguments
      { get; }
   ```

Listing 19-3 illustrates how the `CustomAttributeData` members are used to inspect custom attributes. The code loads `attribs.dll` (used in Listing 19-2) as a reflection-only assembly, iterates through its types, and examines the attributes attached to any types or type members.

Listing 19-3 Examine Attributes in Reflection-Only Assembly with Reflection

```
using System;
using System.Reflection;
using System.Diagnostics;
using System.Collections.Generic;
public class myApp
{
    public static void Main()
    {
        // ................. Reflection-only assembly
        Assembly oldAssembly = Assembly.ReflectionOnlyLoadFrom(
            "\\corecsharp\\attribs.dll");
        // Gets list of attributes associated with assembly.
        IList<CustomAttributeData> attributes =
            CustomAttributeData.GetCustomAttributes(oldAssembly);
        foreach( CustomAttributeData cad in attributes )
            Console.WriteLine(" \r\nCustom Attribute  {0}", cad);
        //
        foreach(Type ty in oldAssembly.GetTypes())
        {
            Console.WriteLine("type {0}",ty.Name);
            // Following statement throws exception
            // because this is a reflection-only assembly
            //Object[] att = ty.GetCustomAttributes(true);

            // Iterate through members of a type.
            foreach(MemberInfo mi in ty.GetMembers())
            {
                bool first=false;
                foreach(CustomAttributeData cad in
                    CustomAttributeData.GetCustomAttributes(mi))
                {
                    if(!first) Console.WriteLine("\tMember:
                                                 {0}",mi.Name);
                    Console.WriteLine("\tCustom attrib: {0}", cad);
                    // (1) Get constructor for attribute
                    ConstructorInfo ci = cad.Constructor;
                    Console.WriteLine("\t  Constructor: {0}",
                                      ci.ToString());
                    // (2) List value for attribute's positional
                    // arguments
```

Listing 19-3 Examine Attributes in Reflection-Only Assembly with Reflection *(continued)*

```
                foreach( CustomAttributeTypedArgument cat in
                       cad.ConstructorArguments)
                    Console.WriteLine("\t  Positional Argument
                       Value: {0}", cat.Value.ToString());
                // (3) List any named arguments
                foreach( CustomAttributeNamedArgument can in
                       cad.NamedArguments)
                    Console.WriteLine("\t  Named Argument:
                       {0}", can.ToString());
                first=true;
            }
          } // MemberInfo
       } // Type
    } // static Main
}
```

If you compare Listing 19-2 with this Listing 19-3, you'll find that they diverge at the point where the code iterates through each `MemberInfo` object (`mi`) for a type. `MemberInfo.GetCustomAttributes` cannot be used with reflection-only code. Instead, a list of `CustomAttributeData` objects is created and enumerated using this statement:

```
foreach(CustomAttributeData cad in
          CustomAttributeData.GetCustomAttributes(mi))
```

Inside this iteration, `CustomAttributeData` properties are used to display information about the associated attribute's constructor. Figure 19-2 shows the attribute information generated for members of the `TestClass` type.

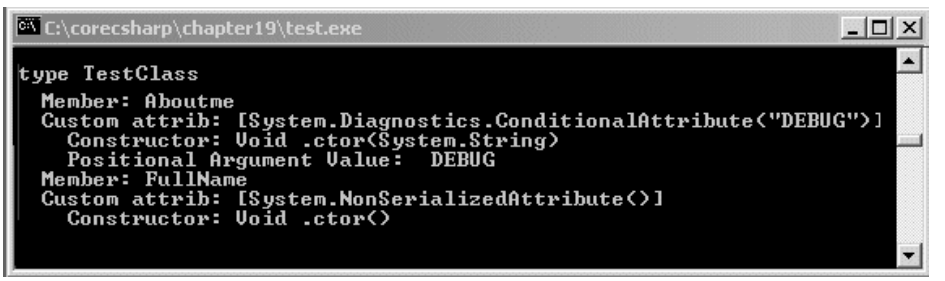

Figure 19-2 Output from inspecting attributes in a reflection-only assembly

Reflection and Late Binding

Imagine an application that permits users to translate words from one language to another. A user selects the language they wish to convert to and enters a word to be translated. The main assembly that implements the user interface calls a separate DLL to perform the conversion. Each language translator is packaged as a separate assembly.

A traditional way to implement this application is to include within the main assembly a series of calls to each language translation assembly. The assembly to be called is determined by which language the user chooses, and code logic—such as a `switch/case` construct—selects the hard-coded method call. This design requires that the main assembly be compiled by including references to the language translation assemblies it supports. As an example, the following would add links to a French and German assembly.

```
csc /r:frenchdict.dll /r:germandict.dll translator.cs
```

Now consider the changes required if you wish to add an assembly to translate into Italian. First, the code must be extended to display Italian as a user option. Then the code and logic must be added to call a method in the new assembly. Finally, `translator.cs` must be recompiled with a reference to the new assembly. A more scalable solution is to design the application to dynamically recognize available assemblies at runtime and invoke the method on them that performs the translation. This technique of binding to objects at runtime is referred to as *late binding*. This is in contrast to *early binding* in which references to objects (DLLs) are established and verified at compile time.

Reflection is the key to implementing late binding in .NET. As we have seen, reflection can be used to view the types in an assembly, and the members within the types. To implement late binding, it is necessary to turn an identified type into an executable object. This is done in .NET using an `Activator` class that includes methods for creating an instance from a type's declaration.

Late Binding Example

To illustrate how reflection and the `Activator` class are used to implement late binding, let's look at how the language translation application can be implemented. Listing 19-4 shows the code for a sample assembly that performs English to German conversion. It accepts an English word and returns the equivalent German version. To simplify the translation, English and matching German words are kept in arrays. In a real application they would, of course, be maintained in a database.

Listing 19-4 Assembly to Perform German Translation

```
using System;
using System.Reflection;
// compile: csc /t:library /out:germandict.dll german.cs
// source code download includes french.cs and Italian.cs
// This attribute describes the role of the assembly
[assembly:AssemblyDescription("English to German")]
public class LookupGerman: ITranslate   // ITranslate interface
{
    public static string GetWord(string inText)
    {
        // translate English word to German
        // use array to simplify demonstration.
        string[] dictEng = {"today", "tomorrow", "yesterday"};
        string[] dictGer = {"heute", "morgen", "gestern"};
        inText= inText.ToLower();
        string deutsch="";
        for(int i=0; i<dictEng.Length;i++)
        {
            if(dictEng[i]==inText)
            {
                deutsch= dictGer[i];
                break;
            }
        }
        return(deutsch);
    }
}
```

Of special interest are the `AssemblyDescription` attribute and the `ITranslate` interface that the conversion class inherits. The `AssemblyDescription` is a built-in attribute available to all assemblies as a way to describe the assembly (see Chapter 15, "Application Deployment Considerations"). In this example it provides a convenient way to describe the conversion performed by the assembly. A menu is created for the user by displaying this information for each assembly.

The `ITranslate` interface serves two purposes: it specifies a name of the method that must be implemented to perform the translation to ensure that each assembly uses the same method name and signature; it also identifies the type (class) that performs the translation. The main assembly only creates instances of types that inherit this interface.

```
public interface ITranslate
{
     string GetWord(string inText);
}
```

Listing 19-5 shows the code for the main assembly. It first searches its directory for possible language conversion assemblies (`*dict.dll`). When one is located, the value of its `AssemblyDescription` attribute is displayed. The user then selects a language from this menu of descriptions and enters an English word to be translated.

Listing 19-5 Application Using Reflection to Provide Late-Bound Invocation

```
using System;
using System.Reflection;
using System.IO;
public class Translator
{
    public static void Main()
    {
        Assembly assembly;
        object[] lookups = new object[10];  // 10 is arbitrary
        MethodInfo[] tMethod = new MethodInfo[10];
        // See Chapter 5 for Directory class
        string[] dlls = Directory.GetFiles(
                  Environment.CurrentDirectory, "*dict.dll");
        Console.WriteLine("Available Translations:");
        int ctr=0;
        // (1) Loop through all matching assemblies
        foreach(string dll in dlls)
        {
            assembly = Assembly.LoadFrom(dll);
            Type[] types= assembly.GetTypes();
            // (2) Loop through types in assembly
            foreach(Type t in types)
            {
                // (3) Search for type implementing ITranslate
                Type intf = t.GetInterface("ITranslate");
                if(intf!=null)
                {
                    // (4) Display description of assembly
                    ctr+=1;
                    Console.WriteLine(ctr+
                        " "+ShowAssembly(assembly));
```

Listing 19-5 Application Using Reflection to Provide Late-Bound Invocation *(continued)*

```
                // (5) Save Translator object
                lookups[ctr] = Activator.CreateInstance(t);
                // GetWord is the method that translates
                MethodInfo mi = t.GetMethod("GetWord");
                tMethod[ctr] = mi;
            }
        }
    }
    Console.WriteLine(" ");
    //
    object[] parms  = new object[1];
    int dict=99;
    while(dict !=0)
    {
        Console.Write("Dictionary (1-{0} 0 to exit)",ctr);
        string sel =  Console.ReadLine();
        dict= int.Parse(sel);
        if(dict>0 && dict <=ctr)
        {
            Console.Write("English word to translate:");
            string myWord;
            myWord = Console.ReadLine();
            // get method to execute on a type object
            MethodInfo mi = tMethod[dict];     // method
            object typeObject= lookups[dict];  // type
            parms[0]= myWord;
            // Execute method to return translated word
            string tWord= (string)mi.Invoke(typeObject, parms);
            if(tWord=="") tWord=
                "Translation not found for:"+myWord;
            Console.WriteLine("--> "+tWord+"\r\n");
        }
    }
}
private static string ShowAssembly(Assembly assembly)
{
    // Get value of AssemblyDescriptionAttribute
    object[] descAttr= assembly.GetCustomAttributes(
           typeof(AssemblyDescriptionAttribute),false);
    AssemblyDescriptionAttribute desc =
           (AssemblyDescriptionAttribute)descAttr[0];
    return(desc.Description);
}
}
```

This example illustrates several features that can be useful in implementing late binding:

- `Directory.GetFiles()` is used to identify DLL files whose name match the pattern (`"*dict.DLL"`) used for naming the language translation assemblies.
- `Assembly.GetTypes()`, which was discussed earlier, returns an array of types in a specified assembly.
- `Type.GetInterface()` is used to detect whether a type inherits from the required `ITranslate` interface. This provides a way to filter only qualifying assemblies. Note that if a type inherits from an `abstract` class instead of an `interface`, the `Type.IsSubclassOf` method can be used to check for inheritance.
- The `Activator.CreateInstance` method accepts a type as input and creates an object using the type's constructor that best matches the specified parameters. In this example, class `LookupGerman` has only a parameterless constructor, so there is no need to use a `CreateInstance` overload that specifies parameters. For a class that does include a constructor having parameters, use an overload that specifies the parameter(s) in an object array:

```
object[] args={"Uppercase"};  // pass string parameter
Activator.CreateInstance(t, args);   // type, parms
```

- A type instance is saved as an `object`, and the type's method that is called to perform the translation (`GetWord` in this example) is stored as a `MethodInfo` type. The `MethodInfo.Invoke` method is executed to return the translation:

```
MethodInfo mi = tMethod[dict];      // method
object typeObject= lookups[dict];   // type
parms[0]= myWord;
// Execute method to return translated word
string tWord= (string)mi.Invoke(typeObject, parms);
```

Figure 19-3 shows the application in action when three language conversion assemblies are available.

Figure 19-3 Application that uses late binding to translate text

19.2 Attributes

Attributes are perhaps the most innovative feature available in .NET and C#. In a nutshell, attributes provide a way to add declarative information to types and members. The information is stored as metadata, and—as seen in Section 19.1—is accessible using reflection. To give you an idea of the power of attributes, consider these examples of attributes built into the .NET Framework:

- `[Serializable]`—Indicates that a class can be serialized.
- `[DllImport("gdi32.dll")]`—Permits managed code to call a function imported from an unmanaged DLL (one that implements native Win32 API calls).
- `[WebMethod(Description="Return stock quote.")]`—Indicates that a method is available in a Web Service, and describes its purpose.
- `[Browsable(true)]`—Makes a property on a custom control displayable in the VS.NET Property Browser at design time.

Although just a small sample of available attributes, this list demonstrates the versatility of attributes: they can be used at design time (`Browsable`) to expose program members to a designer, at runtime (`WebMethod`) to expose program functionality, and at runtime (`Serializable`) to affect the behavior of a program.

Attributes offer another example of the extensibility of the .NET Framework. As we see next, attributes are nothing more than classes that follow certain rules of inheritance and member implementation. By following these rules, any developer can create custom attributes for use in an application or component. Before looking

at how to create a custom attribute, let's look at exactly what an attribute is in programming terms, and how to attach one to a target entity.

What Is an Attribute

An attribute is a type that derives directly or indirectly from the `System.Attribute` class. Its purpose is to associate information with a target entity. This target entity may be an assembly, type, type member, or another attribute. The value of using an attribute lies in the fact that the information it attaches to an entity is stored as metadata, which is, of course, available to a program through reflection.

Attributes are easy to use and easy to build. To illustrate, we'll look at two code examples. The first uses a .NET predefined attribute, `DllImport`, to illustrate the syntax for applying an attribute; in the second example, we'll build a custom attribute and use reflection to inspect its values.

Specifying an Attribute

The syntax for applying an attribute to a declaration should be familiar: a pair of square brackets surrounding one or more comma-separated attributes. Each attribute consists of an attribute name and one or more optional arguments. Multiple attributes can be applied by separating them with commas, or by placing them in separate brackets. By convention, the attribute name ends with the `Attribute` suffix, which can be left off the attribute specification. The arguments may be `positional` or `named`, where any positional arguments must precede named ones.

To illustrate attribute usage, let's look at one of the more interesting attributes provided by .NET—the `DllImport` attribute. If you're not familiar with this attribute, it's found in the `System.Runtime.InteropServices` namespace, and allows managed code to call an unmanaged function in the Win32 API. In C#, the actual call is to a .NET method that serves as a proxy for the unmanaged function. Attached to this proxy is a `DllImport` attribute that identifies and imports the underlying function through which the call to the proxy is routed. Its lone constructor accepts a required positional parameter that specifies the name of the DLL containing the Win32 function to be called:

```
public DllImportAttribute( string dll)
```

The attribute also contains several fields. `EntryPoint` specifies the DLL entry point to be called. `ExactSpelling`, when set to `true`, tells the CLR to search for a function that exactly matches the `EntryPoint` name; otherwise it constructs a special Win32 name based on the `CharSet` parameter. This parameter indicates whether strings passed from the managed code are converted to Unicode or ANSI format for Win32 compatibility. To set the value of an attribute's field, you pass the field name and its value as a *named argument* to the attribute's constructor.

Listing 19-6 shows a class that simulates mouse clicks on a WinForms control. The class contains a static `SendClick` method that has two parameters: the control on which the mouse click is to be simulated, and an `enum` specifying whether a left or right mouse click is to be simulated. `SendClick`, in turn, calls the Win32 `SendMessage` function to pass a message to the selected control indicating it has received a mouse button click. The key to making this work is the `DllImport` attribute that makes `SendMessage` accessible to managed code.

Listing 19-6 Using the `DllImport` Attribute to Simulate Mouse Clicks

```
// using System.Runtime.InteropServices;
// Class to simulate clicking a mouse button
public sealed class ButtonSim
{
    // Win32 Messages that indicate a mouse button event
    private const UInt32 WM_LBUTTONDOWN = 0x201;
    private const UInt32 WM_LBUTTONUP   = 0x202;
    private const UInt32 WM_RBUTTONDOWN = 0x204;
    private const UInt32 WM_RBUTTONUP   = 0x205;

    // SendMessage is used to pass a mouse click message
    [method: DllImport("user32.dll", EntryPoint="SendMessage")]
    private static extern int SendMessage( IntPtr handle,
           UInt32 message, int wParam, int lParam);
    // Simulate left or right mouse button click
    // receiver is target control
    public static void SendClick(Control receiver,
                                 MouseButtons LR)
    {
        // LR specifies left or right mouse button click
        if (receiver != null)
        {
            if(LR==MouseButtons.Left)
            {
                SendMessage(receiver.Handle,WM_LBUTTONDOWN,0,0);
                SendMessage(receiver.Handle,WM_LBUTTONUP,0,0);
            }
            if(LR==MouseButtons.Right)
            {
                SendMessage(receiver.Handle,WM_RBUTTONDOWN,0,0);
                SendMessage(receiver.Handle,WM_RBUTTONUP,0,0);
            }
        }
    }
}
```

An easy way to test this code is to place a button on a form and specify an event handler for its `Click` event. To invoke the `Click` event, call the static `SendClick` method with the button's name and a `MouseButtons enum` value indicting a left or right mouse button click. This causes the event handler to be called as if the mouse button were physically pressed. Here is an example:

```
// Delegate to specify click event handler for button
button1.Click += new System.EventHandler(this.button1_Click);
// Simulate Left mouse click on button1
ButtonSim.SendClick(button1,MouseButtons.Left);
```

Before leaving this example, observe how the attribute specification for `DllImport` includes an *attribute identifier* preceding the attribute name. The purpose of an attribute identifier is to resolve any ambiguity as to what entity the attribute is attached. In this example, `method` specifies that the attribute target is the method and not its return value. (Note that if no identifier is included here, `method` is used as the default.)

```
[method: DllImport("user32.dll", EntryPoint="SendMessage")]
```

To specify an attribute for the method's return value, use the `return` identifier. Other attribute identifiers available are `module`, `type`, `method`, `property`, `event`, `field`, `param`, and `return`. Attribute identifiers are optional, and required only when you need to override the default target entity. As a rule, `return` is never a default identifier, so it must be included when the attribute applies to a return value of a method, delegate, or a property's `get` accessor.

Creating a Custom Attribute

It takes little effort to implement a custom attribute: simply create a class that derives from `System.Attribute` and add some fields or properties to accommodate any desired parameters. As an example, let's create an attribute that can be attached to special device driver classes. The purpose of the attribute is to define the type of device and optionally provide a vendor name. The attribute takes one positional parameter that must be an `enum` of type `DeviceType`, and an optional `Vendor` parameter. Here is the code that defines the attribute:

```
using System;
namespace AttribLib
{
    public enum DeviceType {
        printer,
        reader,
        antenna
    }
```

```
    // Custom attribute with one positional and named parameter
    // Can only be attached to a class
    [AttributeUsage(AttributeTargets.Class)]
    public class DeviceAttribute: Attribute
    {
        private string my_Vendor;
        private DeviceType my_Type;
        public DeviceAttribute(DeviceType dev)
        {  my_Type = dev; }

        public DeviceType Device
        {
            get{ return my_Type; }
        }
        public string Vendor       // serves as a named parameter
        {
            set{my_Vendor = value;}
            get{ return my_Vendor; }
        }
    }
}
```

Note that the custom attribute class has an `AttributeUsage` attribute attached to it. The argument to this attribute specifies which elements the attribute can be used on. In this case, the custom attribute can be used only with a class. Before discussing `AttributeUsage` in detail, let's look at Listing 19-7, which provides a simple code example demonstrating the use of our custom attribute.

Listing 19-7 Using a Custom Attribute to Add Information to a Class

```
using System;
using System.Reflection;
using AttribLib;

[DeviceAttribute(DeviceType.reader, Vendor="Identec")]
public class Identec200
{
    // Other code to implement driver goes here
    public static void ShowMe()
    {
        // Use reflection to describe attribute attached to me
        object[] myAttrib =
               typeof(Identec200).GetCustomAttributes(true);
        DeviceAttribute da = (DeviceAttribute)myAttrib[0];
```

Listing 19-7 Using a Custom Attribute to Add Information to a Class *(continued)*

```
        Console.WriteLine( "Device Type: {0}\r\nVendor: {1}",
               da.Device, da.Vendor);
        /* output:
               Device Type: reader
               Vendor: Identec
        */
    }
}

public class myApp
{
    static void Main()
    {
        Identec200.ShowMe();
    }
}
```

The example is quite straightforward. `DeviceAttribute` is attached to a device driver class. Its two arguments indicate that the class (`Identec200`) supports a reader, and that Identec is the device's vendor. The device driver class contains a `ShowMe` method that describes itself using reflection to display the parameter values of the custom attribute.

Note that `GetCustomAttributes` returns attributes as objects. It is therefore necessary to cast the returned object to the attribute's type in order to access fields or properties that expose an attribute's parameter values. In this case, the values of the positional and named parameter are available through the attribute's `Device` and `Vendor` properties.

Using AttributeUsage to Restrict Where an Attribute Is Applied

In the preceding example, the `AttributeUsage` attribute was used to restrict the custom `DeviceAttribute` attribute from being attached to any entity but a class:

```
[AttributeUsage(AttributeTargets.Class)]
```

An attempt to attach it to any other entity would throw an exception. Let's take a closer look at `AttributeUsage`, beginning with its syntax:

```
[AttributeUsage(ValidOn, AllowMultiple=bool, Inherited=bool )
```

Parameters:

ValidOn	One or more `AttributeTargets` enumeration values that specifies the entity or entities to which an attribute may be applied. Multiple values can be combined using an OR (`|`) operation.
`AllowMultiple`	Indicates whether more than one instance of the attribute can be applied to a single entity.
`Inherited`	If set to `true`, this attribute is inherited by classes derived from the class to which the attribute is applied.

Example:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
                AllowMultiple=false, Inherited=true )
```

The possible `AttributeTargets` names and corresponding values are shown here:

```
Assembly    = 0x0001       Event            = 0x0200
Module      = 0x0002       Interface        = 0x0400
Class       = 0x0004       Parameter        = 0x0800
Struct      = 0x0008       Delegate         = 0x1000
Enum        = 0x0010       ReturnValue      = 0x2000
Constructor = 0x0020       GenericParameter = 0x4000
Method      = 0x0040       All              = 0x7fff
Property    = 0x0080
Field       = 0x0100
```

Note that because an attribute can be assigned to another attribute, you may find it useful to create a custom attribute that represents a combination of `AttributeTargets` values. For example, this code creates a `PropertyorField` attribute that can be used as a replacement for the `AttributeUsage` specification:

```
[AttributeUsage(AttributeTargets.Property |
   AttributeTargets.Field)]
public class PropertyorField: Attribute
{  }
```

19.3 Summary

One of the cornerstones of the .NET Framework is the use of metadata to describe the types and type members in an assembly. The tables that contain this metadata can be accessed using members of the `System.Reflection` namespace along with

the all-important `System.Type` class. As described in Section 19.1, the first step in using reflection is to obtain a `Type` reference to the type being inspected. The `Type` methods can then be used to extract information about methods, properties, constructors, delegates, interfaces, and any other type members. In addition to inspecting metadata, reflection can also be used to dynamically create objects and execute methods on them—a feature known as late binding.

Included in the .NET Framework are a large number of attributes that can be applied to types and type members in order to emit additional information as metadata. This information can be used to affect an application's runtime behavior without modifying code. In addition to the predefined attributes, .NET permits developers to create custom attributes by simply creating a class that inherits from the `System.Attribute` class. Section 19.2 discussed how to create a custom attribute and described how attributes are attached to program entities.

19.4 Test Your Understanding

1. True or False? .NET uses attributes to generate metadata for all of the types and classes in an assembly.

2. Indicate whether each of the following is an advantage of late binding or early binding:

   a. Method calls can be verified.
   b. Intellisense can be used in VS.NET with regard to references.
   c. An executable can be compiled with fewer external references.
   d. The application will run faster.
   e. It is more likely an application can be extended without recompiling it.

3. Which method is used to load an assembly by the long form of its name?

4. Which base class in the reflection namespace defines the `GetCustomAttributes` method?

5. What two methods are used to load an assembly (in execution context)? Which is required for a shared assembly?

6. Explain the difference between *named* and *positional* parameters on an attribute.

7. Which of these `Type` methods can be executed for a reflection-only assembly:

   a. `GetMethods()`
   b. `GetConstructors()`
   c. `GetInterFaces()`
   d. `GetCustomAttributes()`

8. The `DllImport` attribute described in this chapter only works on methods. How do you control which entity (or entities) an attribute can be attached to?