Chapter 18

Reflection

In this chapter:

Reflection is a set of classes that allow you to access and manipulate assemblies and modules and the types and the metadata that they contain. For example, you can use reflection to enumerate loaded assemblies, modules, and classes and the methods, properties, fields, and events that each type exposes. Reflection plays a fundamental role in the Microsoft .NET Framework and works as a building block for other important portions of the runtime. The runtime uses reflection in many circumstances, such as to enumerate fields when a type is being serialized or is being marshaled to another process or another machine. Microsoft Visual Basic transparently uses reflection whenever you access an object's method through late binding.

Reflection code typically uses the types in the System.Reflection namespace; the only class used by reflection outside this namespace is System.Type, which represents a type in a managed module. The .NET Framework also contains the System.Reflection.Emit namespace, which contains classes that let you create an assembly dynamically in memory. For example, the .NET Framework uses the classes in this namespace to compile a regular expression into IL code when the RegexOptions.Compiled option is specified. Because of its narrow scope, I won't cover the System.Reflection.Emit namespace in this book.

Note To avoid long lines, code samples in this chapter assume that the following Imports statements are used at the file or project level:

```
Imports System.CodeDom.Compiler
Imports System.Collections.Generic
Imports System.ComponentModel
Imports System.Diagnostics
Imports System.IO
Imports System.Reflection
Imports System.Runtime.InteropServices
```

Working with Assemblies and Modules

The types in the System.Reflection namespace form a logical hierarchy, at the top of which you find the Assembly class, as you can see in Figure 18-1. All the classes in the hierarchy shown in the figure belong to the System.Reflection namespace, except System.Type. FieldInfo, PropertyInfo, and EventInfo inherit from the MemberInfo abstract class, whereas MethodInfo and Constructor-Info inherit from the MethodBase abstract class (which in turn derives from MemberInfo).

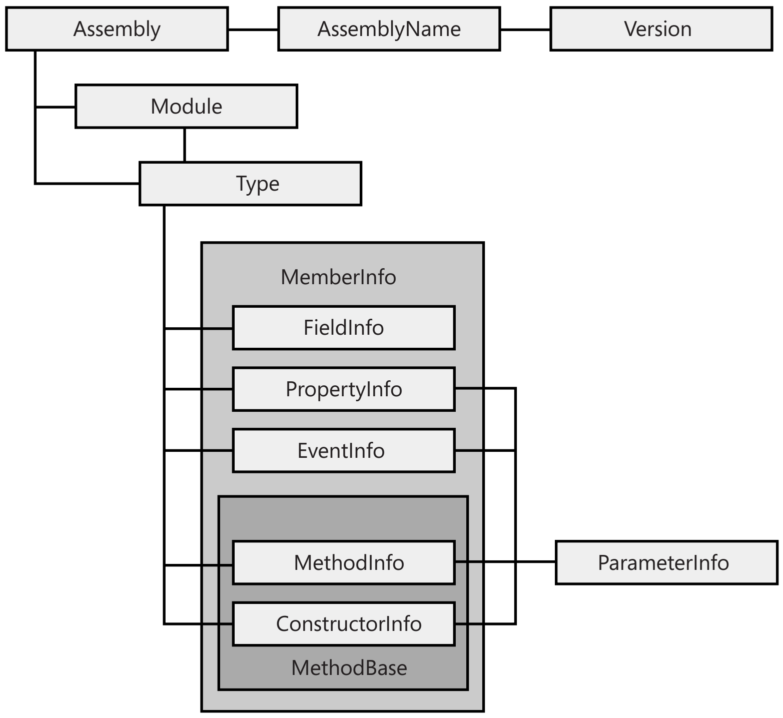


Figure 18-1 The Reflection logical hierarchy

In this section, I describe the Assembly, AssemblyName, and Module classes.

The Assembly Type

As its name implies, the Assembly type represents a .NET assembly. This type offers no constructor method because you never actually create an assembly, but simply get a reference to an existing assembly. There are many ways to perform this operation, as described in the following section.

Loading an Assembly

The Assembly type exposes several static methods that return a reference to an assembly, either running or not (that is, stored on disk but currently not running):

```
' Get a reference to the assembly this code is running in.
Dim asm As Assembly = Assembly.GetExecutingAssembly()
```

```
' Get a reference to the assembly a type belongs to.
asm = Assembly.GetAssembly(GetType(System.Data.DataSet))
' Another way to reach the same result.
asm = GetType(System.Data.DataSet).Assembly

' Get a reference to an assembly given its display name.
' (The argument can be the assembly's full name, which includes
'  version, culture, and public key.)
asm = Assembly.Load("mscorlib")

' Get a reference to an assembly given its filename or its full name.
asm = Assembly.LoadFrom("c:\myapp\mylib.dll")

' Another way to get a reference to an assembly given its path. (See text for notes.)
asm = Assembly.LoadFile("c:\myapp\mylib.dll")
```

Microsoft Visual Basic .NET 2003 requires that Assembly be enclosed in a pair of brackets to distinguish it from the language keyword, but this restriction has been lifted in the current version of the language. Also notice that Microsoft .NET Framework version 1.1 supports the LoadWithPartialName method, which is now obsolete and causes a compilation warning. Along the same lines, version 1.1 of these Load*Xxxx* methods ignores any unknown or incorrect attribute in the display name; in the same circumstances, the Microsoft .NET Framework version 2.0 runtime throws an exception.

A few subtle differences exist among the Load, LoadFrom, and LoadFile methods, and also a few minor changes from .NET Framework 1.1 might impact the way existing applications behave, as explained in the following list. These differences have to do with how the assembly is located and the binding context in which the assembly is loaded. The *binding context* works like a cache for loaded assemblies so that the .NET runtime doesn't have to locate the same assembly again and again each time the application asks for it. (See the section titled "Previously Loaded Assemblies and GAC Searches" in Chapter 17, "Assemblies and Resources.")

- The Load method takes the assembly name, either the short name or the fully qualified name (that includes version, culture, and public key token). If a fully qualified name is provided, this method searches the GAC first and, in general, it follows the same probing sequence that the .NET runtime applies when loading an assembly because your code references one of its types. (See Chapter 17 for details about the probing process.) Assemblies loaded with the Load method become part of the execution context; the main advantage of assemblies loaded in this context is that the .NET runtime is able to resolve their dependencies. You can enumerate assemblies in the execution context with this code:

  ```
  For Each refAsm As Assembly In AppDomain.CurrentDomain.GetAssemblies()
     Console.WriteLine(refAsm.FullName)
  Next
  ```

- The LoadFrom method takes either a relative or absolute file path; if relative, the assembly is searched in the application's base directory. An assembly loaded with this method becomes part of the LoadFrom context. If an NGen image for the assembly exists, it won't be used, but if an assembly with the same identity can be found by probing or is already loaded in the LoadFrom context, the method returns that assembly instead, a behavior that can be quite confusing. When an assembly in the LoadFrom context is executed, the .NET runtime is able to locate its dependencies correctly only if they are under the application's base directory or are already loaded in the LoadFrom context.

- The LoadFile method also takes a file path. It works similarly to LoadFrom, but the assembly is loaded in a different context, and the .NET runtime is unable to find its dependencies, unless they are already loaded in the Load context or you handle the AssemblyResolve event of the AppDomain object. (See the next section for more details about this event.)

To make things more complicated, the behavior of these methods has changed slightly in .NET Framework 2.0. First, both LoadFrom and LoadFile apply the probing policy (which they ignored in .NET Framework 1.1). Second, these methods check the identity of the assembly and load the assembly from the GAC if possible. There is a small probability that these minor changes may break your existing code, so pay attention when you are migrating your reflection-intensive applications to Microsoft Visual Basic 2005.

Another minor difference from version 1.1 is that if loading of an assembly fails once, it will continue to fail even if you remedy the error, until that AppDomain exists. In other words, you can't just trap the exception and ask the user to install the requested assembly in the GAC or in the application's base directory. Instead, you'll have to restart the application or at least load the assembly in a different AppDomain.

.NET Framework 2.0 has the ability to load an assembly for inspection purposes only, using either the ReflectionOnlyLoad or the ReflectionOnlyLoadFrom method:

```
' Load the System.Data.dll for reflection purposes.
asm = Assembly.ReflectionOnlyLoad( _
   "System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")

' Load a file for reflection purposes, given its path.
asm = Assembly.ReflectionOnlyLoadFrom("c:\myapp\mylib.dll")
```

You can enumerate members and perform most other reflection-related operations when you load an assembly in this way, but you can't instantiate a type in these assemblies and therefore you can't execute any code inside them. Unlike the LoadFrom and LoadFile methods, you are allowed to load the assembly even though Code Access Security (CAS) settings would otherwise prevent you from doing so.

Another difference from other load methods is that the ReflectionOnlyLoad and Reflection-OnlyLoadFrom methods ignore the binding policy. Thus, you can load exactly the assemblies you are pointing to—you can even inspect assemblies compiled for a different process architecture—except if you load an assembly with the same identity as one that is already loaded in the inspection context, the latter assembly is returned.

Assemblies loaded with these two methods become part of yet another context, known as the inspection context. There is exactly one inspection context in each AppDomain, and you can enumerate the assemblies it contains with this code:

```
For Each refAsm As Assembly In AppDomain.CurrentDomain.ReflectionOnlyGetAssemblies()
   Console.WriteLine(refAsm.FullName)
Next
```

The Assembly object has a new read-only property, ReflectionOnly, which returns True if the assembly is loaded in the inspection context.

AppDomain Events

When the .NET runtime successfully loads an assembly, either as the result of a JIT-compilation action or while executing an Assembly.Load*Xxxx* method, the AppDomain instance that represents the current application domain fires an AssemblyLoad event. You can use this event to determine exactly when an assembly is loaded, for example, for diagnostics purposes:

```
Sub TestAppDomainLoadAssemblyEvent()
   ' Subscribe to the AppDomain.AssemblyLoad event.
   Dim appDom As AppDomain = AppDomain.CurrentDomain
   AddHandler appDom.AssemblyLoad, AddressOf AppDomain_AssemblyLoad
   ' This statement causes the JIT compilation of DoSomethingWithXml method,
   ' which in turn loads the System.Xml.dll assembly.
   DoSomethingWithXml()
   ' Unsubscribe from the event.
   RemoveHandler appDom.AssemblyLoad, AddressOf AppDomain_AssemblyLoad
End Sub

Private Sub DoSomethingWithXml()
   ' This statement causes the loading of the System.Xml.dll assembly
   ' (assuming that no other XML-related type has been used already and that
   ' the program has been compiled in Release mode).
   Dim doc As New System.Xml.XmlDocument()
End Sub

Sub AppDomain_AssemblyLoad(ByVal sender As Object, ByVal e As AssemblyLoadEventArgs)
   Console.WriteLine("Assembly {0} is being loaded", e.LoadedAssembly.Location)
End Sub
```

Notice that methods in applications compiled in Debug mode are JIT-compiled earlier; therefore, the previous code snippet delivers the expected results only if compiled in Release mode.

When the .NET runtime isn't able to load an assembly, the current AppDomain instance fires an AssemblyResolve event. By handling this event, you can tell the CLR where the assembly is located. For example, let's suppose that you want to force an application to search for dependent assemblies—either private or shared—in a given folder. As you might recall from Chapter 17, by default weakly typed assemblies must be located in the application's folder or one of its subfolders, but the AssemblyResolve event enables you effectively to override the .NET standard binding policy. The handler for this event is peculiar: it is implemented as a

Function that returns an Assembly instance (the assembly that our code loaded manually) or Nothing if the load operation must fail:

```
Sub AppDomainAssemblyResolveEvent()
   ' Subscribe to the AppDomain.AssemblyResolve event.
   Dim appDom As AppDomain = AppDomain.CurrentDomain
   AddHandler appDom.AssemblyResolve, AddressOf AppDomain_AssemblyResolve
   ' Attempt to load an assembly that isn't in the private path.
   Dim asm As Assembly = Assembly.Load("EvaluatorLibrary")
   Console.WriteLine("Found {0} assembly at {1}", asm.FullName, asm.Location)
   ' Unsubscribe from the event.
   RemoveHandler appDom.AssemblyResolve, AddressOf AppDomain_AssemblyResolve
End Sub

Private Function AppDomain_AssemblyResolve(ByVal sender As Object, _
      ByVal e As ResolveEventArgs) As Assembly
   ' Search the assembly in a different directory.
   Dim searchDir As String = "c:\myassemblies"
   For Each dllFile As String In Directory.GetFiles(searchDir, "*.dll")
      Try
         Dim asm As Assembly = Assembly.LoadFile(dllFile)
         ' If the DLL is an assembly and its name matches, we've found it.
         If asm.GetName().Name = e.Name Then Return asm
      Catch ex As Exception
         ' Ignore DLLs that aren't valid assemblies.
      End Try
   Next
   ' If we get here, return Nothing to signal that the search failed.
   Return Nothing
End Function
```

The AssemblyResolve event lets you do wonders, if used appropriately. For example, you might load an assembly from a network share; or you might store all your assemblies in a database binary field and load them when needed, leveraging the Assembly.Load overload that takes a Byte array as an argument.

The AppDomain type also exposes the ReflectionOnlyAssemblyResolve event. As its name suggests, this event is similar to AssemblyResolve, except it fires when the resolution of an assembly fails in the reflection-only context, that is, when the ReflectionOnlyLoad or the ReflectionOnlyLoadFrom method fails. The ReflectionOnlyAssemblyResolve event also fires when the .NET runtime successfully locates the assembly you're loading for reflection-only purposes but fails to load one of the assemblies that the target assembly depends on.

Properties and Methods

Once you have a valid reference to an Assembly object, you can query its properties to learn additional information about it. For example, the FullName property returns a string that holds information about the version and the public key token (this data is the same as the string returned by the ToString method).

```
' This is the ADO.NET assembly.
asm = GetType(System.Data.DataSet).Assembly
Console.WriteLine(asm.FullName)
   ' => System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

The Location and CodeBase read-only properties both return the actual location of the assembly's file, so you can learn where assemblies in the GAC are actually stored, for example:

```
Console.WriteLine(asm.Location)
   ' => C:\WINDOWS\assembly\GAC_32\System.Data\2.0.0.0__ b77a5c561934e089\System.Data.dll
```

When you're not working with assemblies downloaded from the Internet, the information these properties return differs only in format:

```
' …(Continuing previous example)…
Console.WriteLine(asm.CodeBase)
   ' => file:///C:/WINDOWS/assembly/GAC_32/System.Data/2.0.0.0__
        b77a5c561934e089/System.Data.dll
```

The GlobalAssemblyCache property returns a Boolean value that tells you whether the assembly was loaded from the GAC. The ImageRuntimeVersion returns a string that describes the version of the CLR stored in the assembly's manifest (for example, v.2.0.50727). The EntryPoint property returns a MethodInfo object that describes the entry point method for the assembly, or it returns Nothing if the assembly has no entry point (for example, if it's a DLL class library). MethodInfo objects are described in the section titled "Enumerating Members" later in this chapter.

The Assembly class exposes many instance methods, the majority of which enable you to enumerate all the modules, files, and types in the assembly. For example, the GetTypes method returns an array with all the types (classes, interfaces, and so on) defined in an assembly:

```
' Enumerate all the types defined in an assembly.
For Each ty As Type In asm.GetTypes()
   Console.WriteLine(ty.FullName)
Next
```

You can also list only the public types that an assembly exports by using the GetExportedTypes method.

The Assembly class overloads the GetType method inherited from System.Object so that it can take a type name and return the specified Type object.

```
' Next statement assumes that the asm variable is pointing to System.Data.dll.
Dim ty2 As Type = asm.GetType("System.Data.DataTable")
```

If the assembly doesn't contain the specified type, the GetType method returns Nothing. By passing True as its second argument, you can have this method throw a TypeLoadException if the specified type isn't found, and you can have the type name compared in a case-insensitive way by passing True as a third argument:

```
' This statement doesn't raise any exception because type name
' is compared in a case-insensitive way.
Dim ty3 As Type = asm.GetType("system.data.datatable", True, True)
```

Finally, two methods of the Assembly class return an AssemblyName object, which is described in the next section.

The AssemblyName Type

The AssemblyName class represents the object that .NET uses to hold the identity and to retrieve information about an assembly. A fully specified AssemblyName object has a name, a culture, and a version number, but the runtime can also use partially filled AssemblyName objects when searching for an assembly to be bound to caller code. Most often, you get a reference to an existing AssemblyName object by using the GetName property of the Assembly object:

```
' Get a reference to an assembly and its AssemblyName.
Dim asm As Assembly = Assembly.Load("mscorlib")
Dim an As AssemblyName = asm.GetName()
```

You can also get an array of AssemblyName objects using the GetReferencedAssemblies method:

```
' Get information on all the assemblies referenced by the current assembly.
Dim anArr() As AssemblyName
anArr = Assembly.GetExecutingAssembly.GetReferencedAssemblies()
```

Most of the properties of the AssemblyName type are self-explanatory, and some of them are also properties of the Assembly type (as is the case of the FullName and CodeBase properties):

```
Console.WriteLine(an.FullName)
   ' => mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

' The ProcessorArchitecture property is new in .NET Framework 2.0.
' It can be MSIL, X86, IA64, Amd64, or None.
Console.WriteLine(an.ProcessorArchitecture.ToString())   ' => X86

' These properties come from the version object.
Console.WriteLine(an.Version.Major)                      ' => 2
Console.WriteLine(an.Version.Minor)                      ' => 0
Console.WriteLine(an.Version.Build)                      ' => 0
Console.WriteLine(an.Version.Revision)                   ' => 0
' You can also get the version as a single number.
Console.WriteLine(an.Version)                            ' => 2.0.0.0
```

A few methods of the AssemblyName object return a Byte array. For example, you can get the public key and the public key token by using the GetPublicKey and GetPublicKeyToken methods:

```
' Display the public key token of the assembly.
For Each b As Byte In an.GetPublicKeyToken()
   Console.Write("{0} ", b)
Next
```

The CultureInfo property gets or sets the culture supported by the assembly, or returns Nothing if the assembly is culture-neutral.

Unlike most other reflection types, the AssemblyName type has a constructor, which lets you create an AssemblyName instance from the display name of an assembly:

```
Dim an2 As New AssemblyName("mscorlib, Version=2.0.0.0, Culture=neutral, " _
   & "PublicKeyToken=b77a5c561934e089, ProcessorArchitecture=x86")
```

The Module Type

The Module class represents one of the modules in an assembly; don't confuse it with a Visual Basic Module block, which corresponds to a Type object. You can enumerate all the elements in an assembly by using the Assembly.GetModules method:

```
' Enumerate all the modules in the mscorlib assembly.
Dim asm As Assembly = Assembly.Load("mscorlib")
' (Note that Module is a reserved word in Visual Basic.)
For Each mo As [Module] In asm.GetModules()
   Console.WriteLine("{0} - {1}", mo.Name, mo.ScopeName)
Next
```

The preceding code produces only one output line:

```
mscorlib.dll - CommonLanguageRuntimeLibrary
```

The Name property returns the name of the actual DLL or EXE, whereas the ScopeName property is a readable string that represents the module. The vast majority of .NET assemblies (and all the assemblies you can build with Microsoft Visual Studio 2005 without using the Assembly Linker tool) contain only one module. This module is the one that contains the assembly manifest, and you can get a reference to it by means of the Assembly.Manifest-Module property:

```
Dim manModule As [Module] = asm.ManifestModule
```

In general, you rarely need to work with the Module type, and I won't cover it in more detail in this book.

Working with Types

The System.Type class is central to all reflection actions. It represents a managed type, a concept that encompasses classes, structures, modules, interfaces, and enums. The Type class provides all the means to enumerate a type's fields, properties, methods, and events, as well as set properties and fields and invoke methods dynamically.

An interesting detail: a Type object that represents a managed type is unique in a given App-Domain. This means that when you retrieve the Type object corresponding to a given type (for example, System.String) you always get the same instance, regardless of how you retrieve the Type object. This feature allows for the automatic synchronization of multiple static method invocations, among other benefits.

Retrieving a Type Object

The Type class itself doesn't expose any constructors because you never really create a Type object; rather, you get a reference to an existing one. You can choose from many ways to retrieve a reference to a Type object. In previous sections, you saw that you can enumerate all the types in an Assembly or a Module:

```
For Each t As Type In asm.GetTypes()
   Console.WriteLine(t.FullName)
Next
```

More often, you get a Type object using the Visual Basic GetType operator, which takes the unquoted name of a class:

```
Dim ty As Type = GetType(String)
Console.WriteLine(ty.FullName)            ' => System.String
```

If you already have an instance of the class in question, you can use the GetType method that all objects inherit from System.Object:

```
Dim d As Double = 123.45
ty = d.GetType()
Console.WriteLine(ty.FullName)            ' => System.Double
```

The Type.GetType static method takes a quoted class name, so you can build the name of the class dynamically (something you can't do with the GetType function):

```
' Note that you can't pass Type.GetType a Visual Basic synonym,
' such as Short, Integer, Long, or Date.
ty = Type.GetType("System.Int64")
Console.WriteLine(ty.FullName)            ' => System.Int64
```

The GetType method looks for the specified type in the current assembly and then in the system assembly (mscorlib.dll). Like the Assembly.GetType instance method, the Type.GetType static method returns Nothing if the specified type doesn't exist, but you can also pass True as its second argument to force a TypeLoadException in this case, and you can pass True as its third argument if you want the type name to be compared in a case-insensitive way. If the type you want to reference is neither in the caller's assembly nor in mscorlib.dll, you must append a comma and the name of the assembly in which the type resides. For example, the following code snippet shows how you get a reference to the System.Data.DataSet class, which resides in the assembly named System.Data. Because the GAC might contain many assemblies with this friendly name, you must pass the complete identity of the assembly after the first comma:

```
Dim typeName As String = "System.Data.DataSet, System.Data, " _
    & "Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
ty = Type.GetType(typeName)
```

.NET Framework 2.0 adds a variant of the previous method, which loads a type for inspection purposes only (similar to the Assembly.ReflectionOnlyLoad method):

```
' Second argument tells whether an exception is thrown if the type isn't found.
' Third argument tells whether case should be ignored in the search.
ty = Type.ReflectionOnlyGetType(typeName, False, True)
```

The TypeResolve Event

When the .NET runtime isn't able to load a type successfully, it fires the TypeResolve event of the AppDomain object that represents the current application domain. As it happens with the AssemblyResolve event, the TypeResolve event gives you the ability to override the .NET Framework's default binding policy. The following example shows how you can use this event:

```
Sub TestTypeResolveEvent()
   ' Subscribe to the AppDomain.TypeResolve event.
   Dim appDom As AppDomain = AppDomain.CurrentDomain
   AddHandler appDom.TypeResolve, AddressOf AppDomain_TypeResolve
   ' Get a reference to the Form type.
   ' (It should fail, but it doesn't because we are handling the TypeResolve event.)
   Dim ty As Type = Type.GetType("System.Windows.Forms.Form")
   ' Create a form and show it.
   Dim obj As Object = ty.InvokeMember("", BindingFlags.CreateInstance, Nothing, _
      Nothing, Nothing, Nothing)
   ty.InvokeMember("Show", BindingFlags.InvokeMethod, Nothing, obj, Nothing)
   ' Unsubscribe from the event.
   RemoveHandler appDom.TypeResolve, AddressOf AppDomain_TypeResolve
End Sub

Private Function AppDomain_TypeResolve(ByVal sender As Object, _
      ByVal e As ResolveEventArgs) As Assembly
   If e.Name = "System.Windows.Forms.Form" Then
      Dim asmFile As String = Path.Combine( _
         RuntimeEnvironment.GetRuntimeDirectory, "System.Windows.Forms.dll")
      Return Assembly.LoadFile(asmFile)
   End If
   ' Return Nothing if unable to provide an alternative.
   Return Nothing
End Function
```

The TypeResolve event fires when you fail to load a type through reflection, but it doesn't when the .NET runtime has located the assembly and the assembly doesn't contain the searched type. (If the CLR isn't able to locate the assembly, an AppDomain.AssemblyResolve event fires.) For example, the following statement doesn't cause the TypeResolve event to be fired because the CLR can locate the mscorlib.dll even though that assembly doesn't contain the definition of the Form type:

```
Dim ty2 As Type = GetType(Object).Assembly.GetType("System.Windows.Forms.Form")
```

Exploring Type Properties

All the properties of the Type object are read-only for one obvious reason: you can't change an attribute (such as name or scope) of a type defined in a compiled assembly. The names of most properties are self-explanatory, such as the Name (the type's name), FullName (the complete name, which includes the namespace), and Assembly (the Assembly object that contains the type). The IsClass, IsInterface, IsEnum, and IsValueType properties let you classify a given Type object. For example, the following code lists all the types exported by mscorlib.dll, specifying whether each is a class, an enum, a value type, or an interface:

```
Dim asm As Assembly = Assembly.Load("mscorlib")
For Each t As Type In asm.GetExportedTypes()
   If t.IsClass Then
      Console.WriteLine(t.Name & " (Class)")
   ElseIf t.IsEnum Then
      ' An enum is also a value type, so we must test IsEnum before IsValueType.
      Console.WriteLine(t.Name & " (Enum)")
   ElseIf t.IsValueType Then
      Console.WriteLine(t.Name & " (Structure)")
   ElseIf t.IsInterface Then
      Console.WriteLine(t.Name & " (Interface)")
   Else
      ' This statement is never reached because a type
      ' can't be anything other than one of the above.
   End If
Next
```

The IsPublic and IsNotPublic properties return information about the type's visibility. You should use these properties only with types that aren't nested in other types: the IsPublic property of a nested type is always False.

If the type is nested inside another type, you must use the following IsNested*Xxxx* properties to deduce the scope used to declare the type: IsNestedPublic (Public), IsNestedAssembly (Friend), IsNestedFamily (Protected), IsNestedFamORAssem (Protected Friend), IsNested-Private (Private), and IsNestedFamANDAssem (Protected and visible only from inside the assembly, a scope you can't define with Visual Basic). You can also use the DeclaringType property to get the enclosing type of a nested type; this property returns Nothing if the type isn't nested.

While we are on this subject, notice that the FullName property of a nested type includes a plus sign (+) to separate the name of the class and the name of its enclosing type, as in:

```
MyNamespace.MyEnclosingType+MyNestedType
```

A couple of properties are new in .NET Framework 2.0: IsNested returns True if the type is nested in another type (regardless of its scope), whereas IsVisible lets you determine whether the type can be accessed from outside the assembly; it returns True if the type is a Public top-level type or is a Public type nested inside a Public type.

You can get information about inheritance relationships by means of the BaseType (the base class for a type), IsAbstract (True for MustInherit classes), and IsSealed (True for NotInheritable classes) properties:

```
' (The asm variable is pointing to mscorlib…)
For Each t As Type In asm.GetExportedTypes()
   Dim text As String = t.FullName & " "
   If t.IsAbstract Then text &= "MustInherit "
   If t.IsSealed Then text &= "NotInheritable "
   ' We need this test because System.Object has no base class.
   If t.BaseType IsNot Nothing Then
      text &= "(base: " & t.BaseType.FullName & ") "
   End If
   Console.WriteLine(text)
Next
```

You can get additional information on a given type by querying a few methods, such as IsSubclassOfType (returns True if the current type is derived from the type passed as an argument), IsAssignableFrom (returns True if the type passed as an argument can be assigned to the current type), and IsInstanceOfType (returns True if the object passed as an argument is an instance of the current type). Let's recap a few of the many ways you have to test an object's type:

```
If TypeOf obj Is Person Then
   ' obj can be assigned to a Person variable (the Visual Basic way).
End If

If GetType(Person).IsAssignableFrom(obj.GetType()) Then
   ' obj can be assigned to a Person variable (the reflection way).
End If

If GetType(Person).IsInstanceOfType(obj) Then
   ' obj is a Person object.
End If

If GetType(Person) Is obj.GetType() Then
   ' obj is a Person object (but fails if obj is Nothing).
End If

If obj.GetType().IsSubclassOf(GetType(Person)) Then
   ' obj is an object that inherits from Person.
End If
```

Enumerating Members

The Type class exposes an intimidatingly large number of methods. The following methods let you enumerate type members: GetMembers, GetFields, GetProperties, GetMethods, GetEvents, GetConstructors, GetInterfaces, GetNestedTypes, and GetDefaultMembers. All these methods (note the plural names) return an array of elements that describe the members of the type represented by the current Type object. The most generic method in this group is

GetMembers, which returns an array with all the fields, properties, methods, and events that the type exposes. For example, the following code lists all the members of the System.String type:

```
Dim minfos() As MemberInfo = GetType(String).GetMembers()
For Each mi As MemberInfo In minfos
   Console.WriteLine("{0} ({1})", mi.Name, mi.MemberType)
Next
```

The GetMembers function returns an array of MemberInfo elements, where each MemberInfo represents a field, a property, a method, a constructor, an event, or a nested type (including delegates defined inside the class). MemberInfo is an abstract type from which more specific types derive—for example, FieldInfo for field members and MethodInfo for method members. The MemberInfo.MemberType enumerated property lets you discern between methods, properties, fields, and so on.

The GetMembers method returns two or more MemberInfo objects with the same name if the class exposes overloaded properties and methods. So, for example, the output from the preceding code snippet includes multiple occurrences of the Format and Concat methods. You also find multiple occurrences of the constructor method, which is always named .ctor. In the next section, I show how you can explore the argument signature of these overloaded members. Also note that the GetMembers method returns public, instance, and static members, as well as methods inherited by other objects, such as the GetHashCode method inherited from System.Object.

The GetMembers method supports an optional BindingFlags enumerated argument. This bit-coded value lets you narrow the enumeration—for example, by listing only public or instance members. The BindingFlags type is used in many reflection methods and includes many enumerated values, but in this case only a few are useful:

- The Public and NonPublic enumerated values restrict the enumeration according to the scope of the elements. (You must specify at least one of these flags to get a nonempty result.)
- The Instance and Static enumerated values restrict the enumeration to instance members and static members, respectively. (You must specify at least one of these flags to get a nonempty result.)
- The DeclaredOnly enumerated value restricts the enumeration to members declared in the current type (as opposed to members inherited from its base class).
- The FlattenHierarchy enumerated value is used to include static members up the hierarchy.

This code lists only the public, nonstatic, and noninherited members of the String class:

```
' Get all public, instance, noninherited members of String type.
Dim minfo() As MemberInfo = GetType(String).GetMembers( _
    BindingFlags.Public Or BindingFlags.Instance Or BindingFlags.DeclaredOnly)
```

The preceding code snippet produces an array that includes the ToString method, which at first glance shouldn't be in the result because it's inherited from System.Object. It's included because the String class adds an overloaded version of this method, and this overloaded method is the one that appears in the result array.

To narrow the enumeration to a given member type, you can use a more specific Get*Xxxx*s method. When you're using a Get*Xxxx*s method other than GetMembers, you can assign the result to an array of a more specific type, namely, PropertyInfo, MethodInfo, ConstructorInfo, FieldInfo, or EventInfo. (All these specific types derive from MemberInfo.) For example, this code lists only the methods of the String type:

```
For Each mi As MethodInfo In GetType(String).GetMethods()
   Console.WriteLine(mi.Name)
Next
```

The GetInterfaces or GetNestedTypes method return an array of Type elements, rather than a MemberInfo array, so the code in the loop is slightly different:

```
For Each itf As Type In GetType(String).GetInterfaces()
   Console.WriteLine(itf.FullName)
Next
```

All the Get*Xxxx*s methods—with the exception of GetDefaultMembers and GetInterfaces—can take an optional BindingFlags argument to restrict the enumeration to public or nonpublic, static or instance, and declared or inherited members. For more sophisticated searches, you can use the FindMembers method, which takes a delegate pointing to a function that filters individual members. (See MSDN documentation for additional information.)

In many cases, you don't need to enumerate a type's members because you have other ways to find out the name of the field, property, methods, or event you want to get information about. You can use the GetMember or other Get*Xxxx* methods (where *Xxxx* is a singular word) of the Type class—namely, GetMember, GetField, GetProperty, GetMethod, GetEvent, GetInterface, GetConstructor, and GetNestedType—to get the corresponding MemberInfo (or a more specific object):

```
' Get information about the String.Chars property.
Dim pi2 As PropertyInfo = GetType(String).GetProperty("Chars")
```

If you're querying for an overloaded property or method, you need to ask for a specific version of the member by using GetProperty or GetMethod and specifying the exact argument signature by passing an array of Type objects as its second argument:

```
' Get the MethodInfo object for the IndexOf string method with the
' following signature: IndexOf(char, startIndex, endIndex).

' Prepare the signature as an array of Type objects.
Dim argTypes() As Type = {GetType(Char), GetType(Integer), GetType(Integer)}
' Ask for the method with given name and signature.
Dim mi2 As MethodInfo = GetType(String).GetMethod("IndexOf", argTypes)
```

The method signature you pass to GetMethod must include information about whether the argument is passed by reference or is an array. Two new methods of the Type class make this task simpler than it is in .NET Framework 1.1:

```
' This code shows how you can build a reference to the following method
'      Sub TestMethod(ByRef x As Integer, ByVal arr(,) As String).
Dim argType1 As Type = GetType(Integer).MakeByRefType()
Dim argType2 As Type = GetType(String).MakeArrayType(2)
Dim arrTypes() As Type = {argType1, argType2}
Dim mi3 As MethodInfo = GetType(TestClass).GetMethod("TestMethod", arrTypes)
```

Speaking of arrays, notice that the name of array types ends with a pair of brackets:

```
Dim arrTy As Type = GetType(Integer())
Dim arrTy2 As Type = GetType(Integer(,))
Console.WriteLine(arrTy.FullName)           ' => System.Int32[]
Console.WriteLine(arrTy2.FullName)          ' => System.Int32[,]
```

Also, the name of a type that represents a ByRef argument has a trailing ampersand (&) character; therefore, you need to process the value returned by the FullName property if you want to display a type name using Visual Basic syntax:

```
Dim vbTypeName As String = _
   argType1.FullName.Replace("[", "(").Replace("]", ")").Replace("&", "")
```

Finally, your code can easily get a reference to the MethodBase that describes the method being executed by means of a static member of the MethodBase type:

```
Dim currMethod As MethodBase = MethodBase.GetCurrentMethod()
```

Exploring Type Members

After you get a reference to a MemberInfo object—or a more specific object, such as FieldInfo or PropertyInfo—you can retrieve information about the corresponding member. Because all these specific *Xxxx*Info objects derive from MemberInfo, they have some properties in common, including Name, MemberType, ReflectedType (the type used to retrieve this MemberInfo instance), and DeclaringType (the type where this member is declared). The values returned by the last two properties differ only if the member has been inherited.

The following loop displays the name of all the members exposed by the String type, together with a description of the member type. To make things more interesting, I'm suppressing constructor methods, multiple definitions for overloaded methods, and methods inherited from the base Object class:

```
' We use this ArrayList to keep track of items already displayed.
Dim al As New ArrayList()
For Each mi As MemberInfo In GetType(String).GetMembers()
   If mi.MemberType = MemberTypes.Constructor Then
      ' Ignore constructor methods.
   ElseIf Not mi.DeclaringType Is mi.ReflectedType Then
      ' Ignore inherited members.
   ElseIf Not al.Contains(mi.Name) Then
      ' If this element hasn't been listed yet, do it now.
      Console.WriteLine("{0}  ({1})", mi.Name, mi.MemberType)
      ' Add this element to the list of processed items.
      al.Add(mi.Name)
   End If
Next
```

Exploring Fields

Except for the members inherited from MemberInfo, a FieldInfo object exposes only a few properties, including FieldType (the type of the field), IsLiteral (True if the field is actually a constant), IsInitOnly (True if the field is marked as ReadOnly), IsStatic (True if the field is marked as Shared), and other Boolean properties that reflect the scope of the field, such as IsPublic, IsAssembly (Friend), IsFamily (Protected), IsFamilyOrAssembly (Protected Friend), IsFamilyAndAssembly (Protected but visible only from inside the same assembly, a scope not supported by Visual Basic), and IsPrivate:

```
' List all the nonconstant fields with Public or Friend scope in the TestClass type.
For Each fi As FieldInfo In GetType(TestClass).GetFields( _
      BindingFlags.Public Or BindingFlags.NonPublic Or BindingFlags.Instance)
   If (fi.IsPublic OrElse fi.IsAssembly) AndAlso Not fi.IsLiteral Then
      Console.WriteLine("{0} As {1}", fi.Name, fi.FieldType.Name)
   End If
Next
```

A new method in .NET Framework 2.0 allows you to extract the value of a constant:

```
' List all the public constants in the TestClass type.
For Each fi As FieldInfo In GetType(TestClass).GetFields()
   If fi.IsLiteral Then
      Console.WriteLine("{0} = {1}", fi.Name, fi.GetRawConstantValue())
   End If
Next
```

Exploring Methods

Like FieldInfo, the MethodInfo type exposes the IsStatic property and all the other scope-related properties you've just seen, plus a few additional Boolean properties: IsVirtual (the method is marked with the Overridable keyword), IsAbstract (MustOverride), and IsFinal (NotOverridable). The IsSpecialName property returns True if the method has been created by the compiler and should be dealt with in a special way, as is the case of the methods generated by properties and operators. If the method returns a value (a Function, in Visual Basic parlance), the ReturnType property returns the type of the return value; otherwise, it returns a special type whose name is System.Void. This snippet uses these properties to display information on all the methods in a class, exposed in a Visual Basic–like syntax:

```
For Each mi As MethodInfo In GetType(Array).GetMethods()
   ' Ignore special methods, such as property getters and setters.
   If mi.IsSpecialName Then Continue For
   If mi.IsFinal Then
      Console.Write("NotOverridable ")
   ElseIf mi.IsVirtual Then
      Console.Write("Overridable ")
   ElseIf mi.IsAbstract Then
      Console.Write("MustOverride ")
   End If
   Dim retTypeName As String = mi.ReturnType.FullName
   If retTypeName = "System.Void" Then
      Console.WriteLine("Sub {0}", mi.Name)
   Else
      Console.WriteLine("Function {0} As {1}", mi.Name, retTypeName)
   End If
Next
```

The ConstructorInfo type exposes the same members as the MethodInfo type (not surprisingly because both these types inherit from the MethodBase abstract class, which in turn derives from MemberInfo), with the exception of ReturnType (constructors don't have a return type).

Exploring Properties

The PropertyInfo type exposes only three interesting properties besides those inherited from MemberInfo: PropertyType (the type returned by the property), CanRead (False for write-only properties), and CanWrite (False for read-only properties). Oddly, the PropertyInfo type doesn't expose members that indicate the scope of the property or whether it's a static property. You can access this information only indirectly by means of one of the following methods: GetGetMethod (which returns the MethodInfo object corresponding to the Get method), GetSetMethod (the MethodInfo object corresponding to the Set method), or GetAccessors (an array of one or two MethodInfo objects, corresponding to the Get and/or Set accessor methods):

```
Sub ExploringProperties()
   ' Display instance and static Public properties.
   For Each pi As PropertyInfo In GetType(TestClass).GetProperties( _
         BindingFlags.Public Or BindingFlags.Instance Or BindingFlags.Static)
      ' Get either the Get or the Set accessor methods.
      Dim modifier As String = ""
      Dim mi As MethodInfo
      If pi.CanRead Then
         mi = pi.GetGetMethod()
         If Not pi.CanWrite Then modifier = "ReadOnly "
      Else
         mi = pi.GetSetMethod()
         modifier = "WriteOnly "
      End If
      ' Add the Shared qualifier if necessary.
      If mi.IsStatic Then modifier = "Shared " & modifier
      ' Display only Public and Protected properties.
      If mi.IsPublic Or mi.IsFamily Then
         Console.WriteLine("Public {0}Property {1} As {2}", modifier, pi.Name, _
            pi.PropertyType.FullName)
      End If
   Next
End Sub
```

If you need to retrieve a property accessor only to determine its scope or whether the property is static, you can use the GetAccessors method as follows:

```
' Get the first property accessor, even if it's private.
mi = pi.GetAccessors(True)(0)
```

By default the GetGetMethod, GetSetMethod, and GetAccessors methods return only public accessor methods; if the accessor method doesn't exist or isn't public, the return value is Nothing. However, these methods are overloaded to take a Boolean argument: if you pass True, they return the accessor method even if it doesn't have a public scope.

Exploring Events

Getting information about an event is complicated by the fact that the EventInfo type has no property that lets you determine the scope of the event or whether it's static. Instead, you must use GetAddMethod to return the MethodInfo object corresponding to the method that adds a new subscriber to the list of listeners for this event. (This is the method that the AddHandler keyword calls for you behind the scenes.) Typically, this method is named add_*Eventname* and is paired with the remove_*Eventname* hidden method (the method called by RemoveHandler and whose MethodInfo is returned by the GetRemoveMethod). The Visual Basic compiler creates these methods for you by default, unless you define a custom event.

You can query the MethodInfo object returned by either GetAddMethod or GetRemoveMethod to discover the event's scope, its arguments, and whether it's static:

```
' Get information on the SampleEvent event of the TestClass object.
Dim ei As EventInfo = GetType(TestClass).GetEvent("SampleEvent")
' Get a reference to the hidden add_SampleEvent method.
Dim mi2 As MethodInfo = ei.GetAddMethod()
' Test the method scope and check whether it's static.
…
```

Exploring Parameters

The one thing left to do is enumerate the parameters that a property or a method expects. Both the GetIndexParameters (of ParameterInfo) and the GetParameters (of MethodInfo) methods return an array of ParameterInfo objects, where each element describes the attributes of the arguments passed to and from the member.

A ParameterInfo object has properties with names that are self-explanatory: Name (the name of the parameter), ParameterType (the type of the parameter), Member (the MemberInfo the parameter belongs to), Position (an integer that describes where the parameter appears in the method signature), IsOptional (True for optional parameters), and DefaultValue (the default value of an optional parameter). The following code shows how to display the calling syntax for a given method:

```
Dim mi As MethodInfo = GetType(TestClass).GetMethod("MethodWithOptionalArgs")
Console.Write(mi.Name & "(")
For Each pi As ParameterInfo In mi.GetParameters()
   ' Display a comma if it isn't the first parameter.
   If pi.Position > 0 Then Console.Write(", ")
   If pi.IsOptional Then Console.Write("Optional ")
   ' Notice how you can discern between ByVal and ByRef parameters.
   Dim direction As String = "ByVal"
   If pi.ParameterType.IsByRef Then direction = "ByRef"
   ' Process the parameter type.
   Dim tyName As String = pi.ParameterType.FullName
   ' Convert [] into () and drop the & character (included if parameter is ByRef).
   tyName = tyName.Replace("[", "(").Replace("]", ")").Replace("&", "")
   Console.Write("{0} {1} As {2}", direction, pi.Name, tyName)
   ' Append the default value for optional parameters.
   If pi.IsOptional Then Console.Write(" = " & GetObjectValue(pi.DefaultValue))
Next
Console.WriteLine(")")
```

The previous code snippet uses the GetObjectValue auxiliary method, which returns the value of an object in Visual Basic syntax:

```
Function GetObjectValue(ByVal obj As Object) As String
   If obj Is Nothing Then
      Return "Nothing"
   ElseIf obj.GetType() Is GetType(String) Then
      Return """" & obj.ToString() & """"
   ElseIf obj.GetType() Is GetType(Date) Then
      Return "#" & obj.ToString() & "#"
   ElseIf obj.GetType().IsEnum Then
      ' It's an enum type.
      Return obj.GetType().Name & "." & [Enum].GetName(obj.GetType(), obj)
   Else
      ' It's something else, including a number.
      Return obj.ToString()
   End If
End Function
```

Getting the syntax for an event is more complicated because the EventInfo object doesn't expose the GetParameters method. Instead, you must use the EventHandlerType property to retrieve the Type object corresponding to the delegate that defines the event. The Invoke method of this delegate, in turn, has the same signature as the event:

```
Dim ei As EventInfo = GetType(TestClass).GetEvent("SampleEvent")
Dim delegType As Type = ei.EventHandlerType
Dim mi2 As MethodInfo = delegType.GetMethod("Invoke")
For Each pi As ParameterInfo In mi2.GetParameters()
   …
Next
```

Exploring the Method Body

Version 2.0 of the .NET Framework introduces a new feature that, although not completely implemented, surely goes in a very promising direction: the ability to peek at the IL code compiled for a given method. The entry point for this capability is the new MethodBase .GetMethodBody method, which returns a MethodBody object. In turn, a MethodBody object exposes properties that let you list the local variables, evaluate the size of the stack that the method uses, and explore the Try...Catch exception handlers defined in the inspected method.

```
' Get a reference to the method in a type.
Dim mi As MethodInfo = GetType(TestClass).GetMethod("TestMethod")
Dim mb As MethodBody = mi.GetMethodBody()
' Display the number of used stack elements.
Console.WriteLine("Stack Size = {0}", mb.MaxStackSize)

' Display index and type of local variables.
Console.WriteLine("Local variables:")
For Each lvi As LocalVariableInfo In mb.LocalVariables
   Console.WriteLine("  var[{0}] As {1}", lvi.LocalIndex, lvi.LocalType.FullName)
Next

' Display information about exception handlers.
Console.WriteLine("Exception handlers:")
For Each ehc As ExceptionHandlingClause In mb.ExceptionHandlingClauses
   Console.Write("  Type={0}, ", ehc.Flags.ToString())
   If ehc.Flags = ExceptionHandlingClauseOptions.Clause Then
      Console.Write("ex As {0}, ", ehc.CatchType.Name)
   End If
   Console.Write("Try off/len={0}/{1}, ", ehc.TryOffset, ehc.TryLength)
   Console.WriteLine("Handler off/len={0}/{1}", ehc.HandlerOffset, ehc.HandlerLength)
Next
```

The list of exception handlers doesn't differentiate between Catch and Finally clauses belonging to distinct Try blocks, but you can group them correctly by looking at elements with identical TryOffset properties. The Flags property of the ExceptionHandlingClause object helps you understand whether the clause is a filter (When block), a clause (Catch block), or a Finally block. The type of the Exception object is exposed by the CatchType property. For example, given the following method:

```
Sub TestMethod(ByRef x As Integer, ByVal arr(,) As String)
   Try
      Console.WriteLine("First Try block")
   Catch ex As NullReferenceException
      …
   Catch ex As Exception
      …
   Finally
      …
   End Try

   Try
      Console.WriteLine("Second Try block")
   Catch ex As NullReferenceException
      …
   Catch ex As OverflowException
      …
   Catch ex As Exception
      …
   End Try
End Sub
```

Here's the information that might be displayed in the console window. (The actual offset and length information varies depending on the actual executable statements in the method.)

```
Stack Size = 2
Local variables:
  var[0] As System.NullReferenceException
  var[1] As System.Exception
  var[2] As System.NullReferenceException
  var[3] As System.OverflowException
  var[4] As System.Exception
Exception handlers:
  Type=Clause, ex As NullReferenceException, Try off/len=2/14, Handler off/len=16/26
  Type=Clause, ex As Exception, Try off/len=2/14, Handler off/len =42/26
  Type=Finally, Try off/len=2/66, Handler off/len=68/12
  Type=Clause, ex As NullReferenceException, Try off/len=82/13, Handler off/len=95/26
  Type=Clause, ex As OverflowException, Try off/len=82/13, Handler off/len=121/26
  Type=Clause, ex As Exception, Try off/len=82/13, Handler off/len=147/27
```

Notice that the list of local variables is likely to include variables that you haven't declared explicitly but that are created for you by the compiler to store intermediate results, such as the Exception variables in Catch clauses or the upper limit of a For loop.

The only method of the MethodBody object of interest is GetILAsByteArray, which returns an array containing the raw IL opcodes. These opcodes are fully documented, so you might use this method to disassemble a .NET executable. As you can guess, this isn't exactly a trivial task, however.

Reflecting on Generics

Reflection techniques in .NET Framework 2.0 fully support generic types, and you must account for them when exploring the types that an assembly exposes.

Exploring Generic Types

You can distinguish generic type definitions from regular types when enumerating all the types in an assembly by checking their IsGenericTypeDefinition method. The full name of a generic type definition contains a inverse quote character followed by the number of type arguments in the definition. Therefore, given the following code:

```
' List all the generic types in mscorlib.
Dim asm As Assembly = GetType(Object).Assembly
```

```
For Each ty As Type In asm.GetTypes()
   If ty.IsGenericTypeDefinition() Then
      Console.WriteLine(ty.FullName)
   End If
Next
```

This is the kind of results you'll see in the console window:

```
System.Collections.Generic.List`1
System.Collections.Generic.Dictionary`2
System.Action`1
…
```

The names of the generic parameters in the type definition don't appear in the type name because they aren't meaningful in the composition of a unique type name: as you might recall from Chapter 11, "Generics," you can have two generic type definitions with the same name only if the number of their generic parameters differs. The syntax based on the inverse quote character becomes important if you want to retrieve a reference to the generic type definition, as in this code:

```
Dim genType As Type = asm.GetType("System.Collections.Generic.Dictionary`2")
```

There is no built-in method or property that returns the signature of the generic type as it appears in source code, and thus you have to manually strip the inverse quote character from the name and use the GetGenericArguments method to retrieve the name of type parameters:

```
' (Continuing previous code example)
Dim typeName As String = genType.FullName
' Strip the inverse quote character.
typeName = typeName.Remove(typeName.IndexOf("`"c)) & "(Of "
' Append the name of each type parameter.
For Each tyArg As Type In genType.GetGenericArguments()
   ' The GenericParameterPosition property reflects the position where
   ' this argument appears in the signature.
   If tyArg.GenericParameterPosition > 0 Then typeName &= ","
   typeName &= tyArg.Name
Next
typeName &= ")"     ' => System.Collections.Generic.Dictionary(Of TKey,TValue)
```

Exploring Generic Methods

You must adopt a similar approach when exploring the generics methods of a type. (Remember that a method with generic arguments can appear in both a regular and a generic type.) You can check whether a method has a generic definition by means of the MethodInfo .IsGenericMethodDefinition method and explore its generic parameters by means of the MethodInfo.GetGenericArguments method. For example, the following loop displays the name of all the methods in a type using the Of clause for generic methods:

```
' List all the generic methods of the System.Array type.
For Each mi As MethodInfo In GetType(Array).GetMethods()
   If mi.IsGenericMethodDefinition Then
      Dim methodName As String = mi.Name & "(Of "
      For Each tyArg2 As Type In mi.GetGenericArguments()
         If tyArg2.GenericParameterPosition > 0 Then methodName &= ","
         methodName &= tyArg2.Name
      Next
      methodName &= ")"
      Console.WriteLine(methodName)     ' => IndexOf(Of T),…
   End If
Next
```

When you explore the parameters of a method, you must discern between regular types (e.g., System.String) and types passed as an argument to a generic type or method (e.g., T or K). This is possible because the Type class exposes a new IsGenericParameter property, which returns False in the former case and True in the latter. It is essential that you test this method before doing anything else with a Type value because some properties of a type used as a parameter in a generic class return meaningless values or might throw an exception. For example, this is the most correct way to assemble the signature of a method:

```
' (The mi variable points to a MethodInfo object.)
Dim signature As String = mi.Name & "("
For Each par As ParameterInfo In mi.GetParameters()
   If par.Position > 0 Then signature &= ", "
   signature &= par.Name & " As " & GetTypeName(par.ParameterType)
Next
signature &= ")"
Dim retType As Type = mi.ReturnType
If retType.FullName <> "System.Void" Then
   signature &= " As " & GetTypeName(retType)
End If
Console.WriteLine(signature)
   ' => TestMethod(key As K, values As V(), count As System.Int32) As T
```

where the GetTypeName function is defined as follows:

```
Function GetTypeName(ByVal type As Type) As String
   If type.IsGenericParameter Then
      Return type.Name
   Else
      Return type.FullName.Replace("[", "(").Replace("]", ")").Replace("&", "")
   End If
End Function
```

Exploring Members That Use Generic Types

A slightly different problem occurs when you are dealing with a member of a type (either a regular or generic type) and the member uses or returns a generic type that has already been bound with nongeneric arguments, as in the following case:

```
Function Convert(x As List(Of Integer)) As Dictionary(Of String, Double)
   …
End Function
```

When you reflect on the argument and the return type of the previous method, you get the following types:

```
System.Collections.Generic.List`1[[System.Int32, mscorlib, Version=2.0.0.0,
   Culture=neutral, PublicKeyToken=b77a5c561934e089]]

System.Collections.Generic.Dictionary`2[[System.String, mscorlib, Version=2.0.0.0,
   Culture=neutral, PublicKeyToken=b77a5c561934e089],
   [System.Double, mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089]]
```

Three details are worth noticing:

- The type name uses the inverse quote character syntax and is followed by the names of all the types that are bound to the generic type.
- Each argument consists of the type's full name followed by the display name of the assembly where the type is defined, all enclosed in a pair of square brackets.
- The entire list of argument types is enclosed in an additional pair of square brackets.

You can easily extract the name of a generic type and its argument types by parsing this full name, for example, by using a regular expression. Alternatively, you can use the IsGenericType method to check whether the type is the bound version of a generic type, and, if this is the case, you can use the GetGenericTypeDefinition method to extract the name of the original generic type and the GetGenericArguments method to extract the type of individual type arguments:

```
' (The mi variable points to a MethodInfo object.)
Dim retType As Type = mi.ReturnType
Dim typeName As String = retType.GetGenericTypeDefinition.FullName
typeName = typeName.Remove(typeName.IndexOf("`"c)) & "(Of "
Dim sep As String = ""
For Each argType As Type In retType.GetGenericArguments
   typeName &= sep & GetTypeName(argType)
   sep = ", "
Next
typeName &= ")"
Console.WriteLine(typeName)
   ' => System.Collections.Generic.Dictionary(Of System.String, System.Double)
```

In practice, you can gather all the cases that I've illustrated so far in an expanded version of the GetTypeName function (which I introduced in the previous section):

```
Function GetTypeName(ByVal type As Type) As String
   Dim typeName As String = Nothing
   If type.IsGenericTypeDefinition Then
      ' It's the type definition of an "open" generic type.
      typeName = type.FullName
      typeName = typeName.Remove(typeName.IndexOf("`"c)) & "(Of "
      For Each targ As Type In type.GetGenericArguments()
         If targ.GenericParameterPosition > 0 Then typeName &= ","
         typeName &= targ.Name
      Next
      typeName &= ")"
   ElseIf type.IsGenericParameter Then
      ' It's a parameter in an Of clause.
      typeName = type.Name
   ElseIf type.IsGenericType Then
      ' This is a generic type that has been bound to specific types.
      typeName = type.GetGenericTypeDefinition.FullName
      typeName = typeName.Remove(typeName.IndexOf("`"c)) & "(Of "
      Dim sep As String = ""
      For Each argType As Type In type.GetGenericArguments
         typeName &= sep & GetTypeName(argType)
         sep = ", "
      Next
      typeName &= ")"
   Else
      ' This is a regular type.
      typeName = type.FullName
   End If
   ' Account for array types and byref types.
   typeName = typeName.Replace("[", "(").Replace("]", ")").Replace("&", "")
   Return typeName
End Function
```

Thanks to its being recursive, this function is able to deal correctly even with contorted cases such as these:

```
Public MyList As List(Of Dictionary(Of String, Double))
Public MyDictionary As Dictionary(Of String, Dictionary(Of String, List(Of Integer)))
```

Binding a Generic Type

Sometimes you might need to bind a generic type with a set of one or more specific type arguments. This is necessary, for example, when you want to retrieve the Type object that corresponds to List(Of String) or Dictionary(Of String, Integer). The key for this operation is the Type.MakeGenericType method:

```
' Retrieve the type that corresponds to MyGenericTable(Of String, Double).
Dim typeName As String = "MyApp.MyGenericTable`2"
' Get a reference to the "open" generic type.
Dim genType As Type = Assembly.GetExecutingAssembly().GetType(typeName)
' Bind the "open" generic type to a set of arguments, and retrieve
' a reference to the MyGenericTable(Of String, Double).
Dim type As Type = genType.MakeGenericType(GetType(String), GetType(Double))
```

A bound generic type can be useful on at least a couple of occasions. First, you can use it when you need to create an instance of a specific type. (I cover object instantiation through reflection later in this chapter.) Second, you can use it when you are looking for a method with a signature that contains an argument of a specific type. Say you have the following class:

```
Public Class TestClass
   Sub TestSub(ByVal list As List(Of Integer), ByVal x As Integer)
      …
   End Sub
   Sub TestSub(ByVal list As List(Of String), ByVal x As String)
      …
   End Sub
End Class
```

How can you build a MethodInfo object that points to the first TestMethod rather than the second one? Here's the solution:

```
' First, get a reference to the List "open" generic type.
Dim typeName As String = "System.Collections.Generic.List`1"
Dim openType As Type = GetType(Object).Assembly.GetType(typeName)
' Bind the open List type to the Integer type.
Dim boundType As Type = openType.MakeGenericType(GetType(Integer))
' Prepare the signature of the method you're interested in.
Dim argTypes() As Type = {boundType, GetType(Integer)}
' Get the reference to that specific method.
Dim method As MethodInfo = GetType(TestClass).GetMethod("TestSub", argTypes)
```

When you bind an open generic type to a set of argument types, you should ensure that generic constraints are fulfilled. Reflection enables you to extract the constraints associated with each argument by means of the GetGenericParameterConstraints method, whereas the GenericParameterAttributes property returns an enum value that provides information about the New, Class, and Structure constraints:

```
Dim genType As Type = Assembly.GetExecutingAssembly().GetType("MyApp.GenericList`1")
For Each argType As Type In genType.GetGenericArguments()
   ' Get the class and interface constraints for this argument.
   For Each constraint As Type In argType.GetGenericParameterConstraints()
      Console.WriteLine(constraint.FullName)
   Next
   ' Get the New, Class, or Structure constraints.
   Dim attrs As GenericParameterAttributes = argType.GenericParameterAttributes
   If CBool(attrs And GenericParameterAttributes.DefaultConstructorConstraint) Then
      Console.WriteLine("  New (default constructor)")
   End If
   If CBool(attrs And GenericParameterAttributes.ReferenceTypeConstraint) Then
      Console.WriteLine("  Class (reference type)")
   End If
   If CBool(attrs And GenericParameterAttributes.NotNullableValueTypeConstraint) Then
      Console.WriteLine("  Structure (nonnullable value type)")
   End If
Next
```

Reflecting on Attributes

As you might recall from Chapter 2, "Basic Language Concepts," .NET attributes provide a standard way to extend the metadata at the assembly, type, and member levels and can include additional information with a format defined by the programmer. Not surprisingly, the .NET Framework also provides the means to read this attribute-based metadata from an assembly. Because the attributes you define in your code are perfectly identical to the attributes that the .NET Framework defines for its own purposes, the mechanism for extracting either kind of attributes is the same. Therefore, even though in this section I show you how to extract .NET attributes, keep in mind that you can apply the same techniques for extracting your custom attributes. (You'll find many examples of the latter ones in Chapter 19, "Custom Attributes.")

Exploring Attributes

You can use several techniques to extract the custom attribute associated with a specific element.

First, you can use the IsDefined method exposed by the Assembly, Module, Type, ParameterInfo, and MemberInfo classes (and all the classes that inherit from MemberInfo, such as FieldInfo and PropertyInfo). This method returns True if the attribute is defined for the specified element, but doesn't let you read the attribute's fields and properties. The last argument passed to the method is a Boolean value that specifies whether attributes inherited from the base class should be returned:

```
' The second argument specifies whether you also want to test
' attributes inherited from the base class.
If GetType(Person).IsDefined(GetType(SerializableAttribute), False) Then
   Console.WriteLine("The Person class is serializable")
End If
```

(See the companion code for the complete listing of the Person type.) Second, you can use the GetCustomAttributes method (note the plural) exposed by the Assembly, Module, Type, ParameterInfo, and MemberInfo classes (and all the classes that inherit from MemberInfo). This method returns an array containing all the attributes of the specified type that are associated with the specified element so that you can read their fields and properties, and you can specify whether attributes inherited from the base class should be included in the result:

```
' Display all the Conditional attributes associated with the Person.SendEmail method.
Dim mi As MethodInfo = GetType(Person).GetMethod("SendEmail")
' GetCustomAttributes returns an array of Object elements, so you need to cast.
Dim miAttrs() As ConditionalAttribute = DirectCast( _
   mi.GetCustomAttributes(GetType(ConditionalAttribute), False), ConditionalAttribute())
' Check whether the result contains at least one element.
If miAttrs.Length > 0 Then
   Console.WriteLine("SendEmail is marked with the following Conditional attribute(s):")
   ' Read the properties of individual attributes.
   For Each attr As ConditionalAttribute In miAttrs
      Console.WriteLine("   <Conditional(""{0}"")>", attr.ConditionString)
   Next
End If
```

Third, you can use an overload of the GetCustomAttributes method exposed by the Assembly, Module, Type, ParameterInfo, and MemberInfo classes (and all the classes that inherit from it) that doesn't take an attribute type as an argument. When you use this overload, the method returns an array containing all the custom attributes associated with the element:

```
' Display all the attributes associated with the Person.FirstName field.
Dim fi As FieldInfo = GetType(Person).GetField("FirstName")
Dim fiAttrs As Array = fi.GetCustomAttributes(False)
' Check whether the result contains at least one element.
```

```
If fiAttrs.Length > 0 Then
   Console.WriteLine("FirstName is marked with the following attribute(s):")
   ' Display the name of all attributes (but not their properties).
   For Each attr As Attribute In fiAttrs
      Console.WriteLine(attr.GetType().FullName)
   Next
End If
```

To further complicate your decision, you can achieve the same results shown previously by means of static methods of the System.Attribute type:

```
' Check whether the Person class is marked as serializable.
If Attribute.IsDefined(GetType(Person), GetType(SerializableAttribute)) Then
   Console.WriteLine("The Person class is serializable")
End If

' Retrieve the Conditional attributes associated with the Person.SendEmail method,
' including those inherited from the base class.
Dim mi As MethodInfo = GetType(Person).GetMethod("SendEmail")
Dim miAttrs() As ConditionalAttribute = DirectCast( _
   Attribute.GetCustomAttributes(mi, GetType(ConditionalAttribute), True),
ConditionalAttribute())
' Check whether the result contains at least one element.
If miAttrs.Length > 0 Then
   …
End If

' Display all the attributes associated with the Person.FirstName field.
Dim fi As FieldInfo = GetType(Person).GetField("FirstName")
Dim fiAttrs As Array = Attribute.GetCustomAttributes(fi, False)
' Check whether the result contains at least one element.
If fiAttrs.Length > 0 Then
   …
End If
```

The System.Attribute class also exposes the GetCustomAttribute static method (note the singular), which returns the only attribute of the specified type:

```
' Read the Obsolete attribute associated with the Person class, if any.
Dim tyAttr As ObsoleteAttribute = DirectCast(Attribute.GetCustomAttribute( _
   GetType(Person), GetType(ObsoleteAttribute)), ObsoleteAttribute)
If tyAttr IsNot Nothing Then
   Console.WriteLine("The Person class is marked as obsolete.")
   Console.WriteLine("  IsError={0}, Message={1}", tyAttr.IsError, tyAttr.Message)
End If
```

An important note: you should never use the Attribute.GetCustomAttribute method with attributes that might appear multiple times—such as the Conditional attribute—because in that case the method might throw an AmbiguousMatchException object.

All these alternatives are quite confusing, so let me recap when each of them should be used:

- If you just need to check whether an attribute is associated with an element, use the Attribute.IsDefined static method or the IsDefined instance method exposed by the Assembly, Module, Type, ParameterInfo, and MemberInfo classes. This technique doesn't actually instantiate the attribute object in memory and is therefore the fastest of the group.

- If you are checking whether a single-instance attribute is associated with an element and you want to read the attribute's fields and properties, use the Attribute.GetCustomAttribute static method. (Don't use this technique with attributes that might appear multiple times—such as the Conditional attribute—because in that case the method might throw an AmbiguousMatchException object.)
- If you are checking whether a multiple-instance attribute is associated with an element and you want to read the attribute's fields and properties, use the Attribute.GetCustomAttributes static method or the GetCustomAttributes method exposed by the Assembly, Module, Type, ParameterInfo, and MemberInfo classes. You must use this technique when reading all the attributes associated with an element, regardless of the attribute type.

Although all the techniques discussed in this section are available in .NET Framework 1.1 as well, there is a new important change in how you use them to query some special CLR attributes, such as Serializable, NonSerialized, DllImport, StructLayout, and FieldOffset. To improve performance and to save space in metadata tables, previous versions of the .NET Framework stored these special attributes using a format different from all other attributes. Consequently, you couldn't reflect on these attributes by using one of the techniques I just illustrated. Instead, you had to use special properties exposed by other reflection objects, for example, the IsSerialized and IsLayoutSequential properties of the Type class or the IsNonSerialized property of the FieldInfo class. A welcome addition in .NET Framework 2.0 is that you don't need to use any of these properties any longer because all the special .NET attributes can be queried by means of the IsDefined, GetCustomAttribute, and GetCustomAttributes methods described in this section. (However, properties such as IsSerializable and IsLayoutSequential continue to be supported for backward compatibility.)

The CustomAttributeData Type

Version 1.1 of the .NET Framework has a serious limitation related to custom attributes: you could search attributes buried in metadata, instantiate them, and read their properties, but you have no documented means for extracting the exact syntax used in code to define the attribute. For example, you can't determine whether an attribute field or property is assigned in the attribute's constructor using a standard (mandatory) argument or a named (optional) argument; if the field or property is equal to its default value (Nothing or zero), you can't determine whether it happened because the property was omitted in the attribute's constructor. For example, these limitations prevent a .NET developer from building a full-featured object browser.

In addition to this limitation inherited from .NET Framework 1.1, you run into another problem under .NET Framework 2.0 when you want to extract custom attributes from assemblies that have been loaded for reflection-only purposes. In fact, both the GetCustomAttribute and the GetCustomAttributes methods instantiate the custom attribute and therefore would run some code inside the assembly, which is prohibited.

Both issues have been resolved by means of the new CustomAttributeData type and the auxiliary CustomAttributeTypedArgument class (which represents a positional argument in the attribute's constructor) and CustomAttributeNamedArgument class (which represents a named argument).

You create an instance of the CustomAttributeData type by means of the GetCustomAttributes static method that the type itself exposes. Each CustomAttributeData object has three properties: Constructor (the ConstructorInfo object that represents the attribute's constructor being used), ConstructorArguments (a list of CustomAttributeTypedArgument objects), and NamedArguments (a list of CustomAttributeNamedArgument objects):

```
' Retrieve the syntax used in custom attributes for the TestClass type.
Dim attrList As IList(Of CustomAttributeData) = _
   CustomAttributeData.GetCustomAttributes(GetType(TestClass))

' Iterate over all the attributes.
For Each attrData As CustomAttributeData In attrList
   ' Retrieve the attribute's type, by means of the ConstructorInfo object.
   Dim attrType As Type = attrData.Constructor.DeclaringType
   ' Start building the Visual Basic code.
   Dim attrString As String = "<" & attrType.FullName & "("
   Dim sep As String = ""

   ' Include all mandatory arguments for this constructor.
   For Each typedArg As CustomAttributeTypedArgument In attrData.ConstructorArguments
      attrString &= sep & FormatTypedArgument(typedArg)
      ' A comma is used as the separator for all elements after the first one.
      sep = ", "
   Next

   ' Include all optional arguments for this constructor.
   For Each namedArg As CustomAttributeNamedArgument In attrData.NamedArguments
      ' The TypedValue property returns a CustomAttributeTypedArgument object.
      Dim typedArg As CustomAttributeTypedArgument = namedArg.TypedValue
      ' Use the MemberInfo property to retrieve the field or property name.
      attrString &= sep & namedArg.MemberInfo.Name & ":=" & FormatTypedArgument(typedArg)
      ' A comma is used as the separator for all elements after the first one.
      sep = ", "
   Next
   ' Complete the attribute syntax and display it.
   attrString &= ")>"
   Console.WriteLine(attrString)
Next
```

The FormatTypedArgument method takes a CustomAttributeTypedArgument object and returns the corresponding Visual Basic code that can initialize it:

```
' Return a textual representation of a string, date, or numeric value.
Function FormatTypedArgument(ByVal typedArg As CustomAttributeTypedArgument) As String
   If typedArg.ArgumentType Is GetType(String) Then
      ' It's a quoted string.
      Return """" & typedArg.Value.ToString() & """"
   ElseIf typedArg.ArgumentType Is GetType(Date) Then
      ' It's a Date constant.
      Return "#" & typedArg.Value.ToString() & "#"
   ElseIf typedArg.ArgumentType.IsEnum Then
      ' It's an enum value.
      Return typedArg.ArgumentType.Name & "." & _
         [Enum].GetName(typedArg.ArgumentType, typedArg.Value)
   Else
      ' It's something else (presumably a number).
      Return typedArg.Value.ToString()
   End If
End Function
```

Creating a Custom Object Browser

All the reflection properties shown enable you to create a custom object browser that can solve problems that are out of reach for the object browser included in Visual Studio. Creating a custom object browser isn't a trivial task, though, especially if you want to implement a sophisticated user interface. For this reason, in this section I focus on a simple but useful object browser implemented as a Console application.

The sample application I discuss here is able to display all the types and members in an assembly that are marked with the Obsolete attribute. I implemented this utility to keep an updated list of members that are in beta versions of .NET Framework 2.0 but would have been removed before the release version, as well as members that were present in .NET Framework 1.1 but have been deprecated in the current version. You can pass it the path of an assembly or launch it without passing anything on the command line: in the latter case, the utility will analyze all the assemblies in the .NET Framework main directory.

The program displays its results in the console window and takes several minutes to explore all the assemblies in the .NET Framework, but you can redirect its output to a file and then load the file in a text editor to quickly search for a type or a method. Here are the core routines:

```
Imports System.IO
Imports System.Reflection
Imports System.Runtime.InteropServices

Module MainModule
   Sub Main(ByVal args() As String)
      If args.Length = 0 Then
         ShowObsoleteMembers()
      Else
         ShowObsoleteMembers(args(0))
      End If
   End Sub

   ' Process all the assemblies in the .NET Framework directory.
   Sub ShowObsoleteMembers()
      Dim path As String = RuntimeEnvironment.GetRuntimeDirectory()
      For Each asmFile As String In Directory.GetFiles(path, "*.dll")
         ShowObsoleteMembers(asmFile)
      Next
   End Sub

   ' Process an assembly at the specified file path.
   Sub ShowObsoleteMembers(ByVal asmFile As String)
      Try
         Dim asm As Assembly = Assembly.LoadFrom(asmFile)
         ShowObsoleteMembers(asm)
      Catch ex As Exception
         ' The file isn't a valid assembly.
      End Try
   End Sub

   ' Process all the types and members in an assembly.
   Sub ShowObsoleteMembers(ByVal asm As Assembly)
      Dim attrType As Type = GetType(ObsoleteAttribute)

      ' This header is displayed only if this assembly contains obsolete members.
      Dim asmHeader As String = String.Format("ASSEMBLY {0}{1}", _
         asm.GetName().Name, ControlChars.CrLf)

      For Each type As Type In asm.GetTypes()
         ' This header will be displayed only if the type is obsolete or
         ' contains obsolete members.
         Dim typeHeader As String = String.Format("   TYPE {0}{1}", _
            GetTypeName(type), ControlChars.CrLf)

         ' Search the Obsolete attribute at the type level.
         Dim attr As ObsoleteAttribute = DirectCast( _
            Attribute.GetCustomAttribute(type, attrType), ObsoleteAttribute)
         If attr IsNot Nothing Then
            ' This type is obsolete.
            Console.Write(asmHeader & typeHeader)
            ' Display the message attached to the attribute.
            Dim message As String = "WARNING"
            If attr.IsError Then message = "ERROR"
            Console.WriteLine("      {0}: {1}", message, attr.Message)
            ' Don't display the assembly header again.
            asmHeader = ""
         Else
            ' The type isn't obsolete; let's search for obsolete members.
            For Each mi As MemberInfo In type.GetMembers()
               attr = DirectCast(Attribute.GetCustomAttribute(mi, _
                  attrType), ObsoleteAttribute)
               If attr IsNot Nothing Then
                  ' This member is obsolete.
                  Dim memberHeader As String = String.Format("        {0} {1}", _
                     mi.MemberType.ToString().ToUpper(), GetMemberSyntax(mi))
                  Console.WriteLine(asmHeader & typeHeader & memberHeader)
                  ' Display the message attached to the attribute.
                  Dim message As String = "WARNING"
                  If attr.IsError Then message = "ERROR"
                  Console.WriteLine("            {0}: {1}", message, attr.Message)
                  ' Don't display the assembly and the type header again.
                  asmHeader = ""
                  typeHeader = ""
               End If
            Next
         End If
      Next
   End Sub
End Module
```

The main program uses a few helper routines, for example, to assemble the name of a type or the signature of a method using Visual Basic syntax. I have explained how this code works in previous sections, so I won't do it again here. I have gathered these methods in a separate module so that you can reuse them easily in other reflection-intensive projects:

```
Module ReflectionHelpers

   ' Returns the name of a type. (Supports generics and array types.)
   Function GetTypeName(ByVal type As Type) As String
      Dim typeName As String = Nothing
      Dim suffix As String = ""

      ' Account for array types.
      If type.IsArray Then
         suffix = "()"
         type = type.GetElementType()
      End If
      ' Account for byref types.
      If type.IsByRef Then type = type.GetElementType()

      If type.IsGenericTypeDefinition Then
         ' It's the type definition of an "open" generic type.
         typeName = type.FullName
         typeName = typeName.Remove(typeName.IndexOf("`"c)) & "(Of "
         For Each targ As Type In type.GetGenericArguments()
            If targ.GenericParameterPosition > 0 Then typeName &= ","
            typeName &= targ.Name
         Next
         typeName &= ")"
      ElseIf type.IsGenericParameter Then
         ' It's a parameter in an Of clause.
         typeName = type.Name
      ElseIf type.IsGenericType Then
         ' This is a generic type that has been bound to specific types.
         typeName = type.GetGenericTypeDefinition.FullName
         typeName = typeName.Remove(typeName.IndexOf("`"c)) & "(Of "
         Dim sep As String = ""
         For Each argType As Type In type.GetGenericArguments
            typeName &= sep & GetTypeName(argType)
            sep = ", "
         Next
         typeName &= ")"
      Else
         ' This is a regular type.
         typeName = type.FullName
      End If
      Return typeName & suffix
   End Function
```

```
' Return the name of a member. (Recognizes constructors and generic methods.)
Public Function GetMemberName(ByVal mi As MemberInfo) As String
   Dim memberName As String = mi.Name

   Select Case mi.MemberType
      Case MemberTypes.Constructor
         memberName = "New"
      Case MemberTypes.Method
         ' Account for generic methods.
         Dim method As MethodInfo = DirectCast(mi, MethodInfo)
         If method.IsGenericMethodDefinition() Then
            ' Include all type arguments.
            memberName &= "(Of "
            For Each ty As Type In method.GetGenericArguments()
               If ty.GenericParameterPosition > 0 Then memberName &= ","
               memberName &= ty.Name
            Next
            memberName &= ")"
         End If
   End Select
   Return memberName
End Function

' Returns the syntax of a member
Public Function GetMemberSyntax(ByVal member As MemberInfo) As String
   Dim memberSyntax As String = GetMemberName(member)

   Select Case member.MemberType
      Case MemberTypes.Property
         Dim pi As PropertyInfo = DirectCast(member, PropertyInfo)
         memberSyntax &= GetParametersSyntax(pi.GetGetMethod(True).GetParameters()) _
            & " As " & GetTypeName(pi.PropertyType)
      Case MemberTypes.Method
         Dim mi As MethodInfo = DirectCast(member, MethodInfo)
         memberSyntax = memberSyntax & GetParametersSyntax(mi.GetParameters())
         If mi.ReturnType.FullName <> "System.Void" Then
            memberSyntax &= " As " & GetTypeName(mi.ReturnType)
         End If
      Case MemberTypes.Constructor
         Dim ci As ConstructorInfo = DirectCast(member, ConstructorInfo)
         memberSyntax &= memberSyntax & GetParametersSyntax(ci.GetParameters)
      Case MemberTypes.Event
         Dim ei As EventInfo = DirectCast(member, EventInfo)
         Dim mi As MethodInfo = ei.EventHandlerType.GetMethod("Invoke")
         memberSyntax &= GetParametersSyntax(mi.GetParameters())
   End Select
   Return memberSyntax
End Function

' Returns the syntax of an array of parameters.
Private Function GetParametersSyntax(ByVal parInfos() As ParameterInfo) As String
   Dim paramSyntax As String = "("
   Dim sep As String = ""
   For Each pi As ParameterInfo In parInfos
      paramSyntax &= sep & GetTypeName(pi.ParameterType)
      sep = ", "
      Next
      Return paramSyntax & ")"
   End Function
End Module
```

As provided, the utility displays output in a purely textual format. It is easy, however, to change the argument of String.Format methods so that it outputs XML or HTML text, which would greatly improve the appearance of the result. (The complete demo program contains modified versions of this code that outputs HTML and XML text.)

Reflection at Run Time

So far, I've shown how to use reflection to enumerate all the types and members in an assembly, an activity that is central to applications such as object browsers or code generators. If you write mostly business applications, you might object that reflection doesn't have much to offer you, but this isn't correct. In fact, reflection also allows you to actually create objects and invoke methods in a sort of "late-bound" mode, that is, without you having to burn the type name and the method name in code. In this section, I show a series of techniques based on this capability.

Creating an Object Dynamically

Let's start by seeing how you can instantiate an object given its type name. You can choose from three ways to create a .NET object using reflection: by using the CreateInstance method of the System.Activator class, by using the InvokeMember method of the Type class, or by invoking one of the type's constructor methods.

If the type has a parameterless constructor, creating an instance is simple:

```
' Next statement assumes that the Person class is defined in
' an assembly named "MyApp".
Dim type As Type = Assembly.GetExecutingAssembly().GetType("MyApp.Person")
Dim o As Object = Activator.CreateInstance(type)
' Prove that we created a Person.
Console.WriteLine("A {0} object has been created", o.GetType().Name)
```

To call a constructor that takes one or more parameters, you must prepare an array of values:

```
' (We reuse the type variable from previous code…)
' Use the constructor that takes two arguments.
Dim args2() As Object = {"Joe", "Evans"}
' Call the constructor that matches the parameter signature.
Dim o2 As Object = Activator.CreateInstance(type, args2)
```

You can use InvokeMember to create an instance of the class and even pass arguments to its constructor, as in the following code:

```
' Prepare the array of parameters.
Dim args3() As Object = {"Joe", "Evans"}
```

```
' Constructor methods have no name and take Nothing in the second to last argument.
Dim o3 As Object = type.InvokeMember("", BindingFlags.CreateInstance, _
    Nothing, Nothing, args3)
```

Creating an object through its constructor method is a bit more convoluted, but I'll demonstrate the technique here for the sake of completeness:

```
' Prepare the argument signature as an array of types (two strings).
Dim argTypes() As Type = {GetType(String), GetType(String)}
' Get a reference to the correct constructor.
Dim ci As ConstructorInfo = type.GetConstructor(argTypes)
' Prepare the parameters.
Dim args4() As Object = {"Joe", "Evans"}
' Invoke the constructor and assign the result to a variable.
Dim o4 As Object = ci.Invoke(args4)
```

Regardless of the technique you used to create an instance of the type, you usually assign the instance you've created to an Object variable, as opposed to a strongly typed variable. (If you knew the name of the type at compile time, you wouldn't need to use reflection in the first place.) There is only one relevant exception to this rule: when you know in advance that the type being instantiated derives from a specific base class (or implements a given interface), you can cast the Object variable to a variable typed after that base class (or interface) and access all the members that the object inherits from the base class (or interface).

The new MakeArrayType method of the Type class makes it very simple to instantiate arrays using reflection, as you can see in this code:

```
' Create an array of Double. (You can pass an integer argument to the MakeArrayType
' method to specify the rank of the array, for multidimensional arrays.
Dim arrType As Type = GetType(Double).MakeArrayType()
' The new array has 10 elements.
Dim arr As Array = DirectCast(Activator.CreateInstance(arrType, 10), Array)
' Prove that an array of 10 elements has been created.
Console.WriteLine("{0} {1} elements", arr.Length, arr.GetValue(0).GetType.Name)
```

When you work with an array created using reflection, you typically assign its elements with the SetValue method and read them back with the GetValue method:

```
' Assign the first element and read it back.
arr.SetValue(123.45, 0)
Console.WriteLine(arr.GetValue(0))           ' => 123.45
```

Accessing Members

In the most general case, after you've created an instance by using reflection, all you have is an Object variable pointing to a type and no direct way to access one of its members. The easiest operation you can perform is reading or writing a field by means of the GetValue and SetValue methods of the FieldInfo object:

```
' Create a Person object and reflect on it.
Dim type As Type = Assembly.GetExecutingAssembly().GetType("MyApp.Person")
Dim args() As Object = {"Joe", "Evans"}
Dim o As Object = Activator.CreateInstance(type, args)

' Get a reference to its FirstName field.
Dim fi As FieldInfo = type.GetField("FirstName")
' Display its current value, and then change it.
Console.WriteLine(fi.GetValue(o))       ' => Joe
fi.SetValue(o, "Robert")

' Prove that it changed, by casting to a strong-type variable.
Dim pers As Person = DirectCast(o, Person)
Console.WriteLine(pers.FirstName)       ' => Robert
```

Like FieldInfo, the PropertyInfo type exposes the GetValue and SetValue methods, but properties can take arguments, and thus these methods take an array of arguments. You must pass Nothing in the second argument if you're calling parameterless properties.

```
' (Continuing previous example…)
' This code assumes that the Person type exposes a 16-bit Age property.
' Get a reference to the PropertyInfo object.
Dim pi As PropertyInfo = type.GetProperty("Age")
' Note that the type of value must match exactly.
' (Integer constants must be converted to Short, in this case.)
pi.SetValue(pers, 35S, Nothing)
' Read it back.
Console.WriteLine(pi.GetValue(pers, Nothing))   ' => 35
```

If the property takes one or more arguments, you must pass an Object array containing one element for each argument:

```
' Get a reference to the PropertyInfo object.
Dim pi2 As PropertyInfo = type.GetProperty("Notes")
' Prepare the array of parameters.
Dim args2() As Object = {1}
' Set the property.
pi2.SetValue(o, "Tell John about the briefing", args2)
' Read it back.
Console.WriteLine(pi2.GetValue(o, args2))
```

A similar thing happens when you're invoking methods, except that you use the Invoke method instead of GetValue or SetValue:

```
' Get the MethodInfo for this method.
Dim mi As MethodInfo = type.GetMethod("SendEmail")
' Prepare an array for expected arguments.
Dim arguments() As Object = {"This is a message", 3}
' Invoke the method.
mi.Invoke(o, arguments)
```

Things are more interesting when optional arguments are involved. In this case, you pass the Type.Missing special value, as in this code:

```
' …(Initial code as above)…
' Don't pass the second argument (optional).
```

```
arguments = New Object() {"This is a message", type.Missing}
mi.Invoke(o, arguments)' Don't pass the second argument.
```

Alternatively, you can query the DefaultValue property of corresponding ParameterInfo to learn the default value for that specific argument:

```
' …(Initial code as above)…
' Retrieve the DefaultValue from the ParameterInfo object.
arguments = New Object() {"This is a message", mi.GetParameters(1).DefaultValue}
mi.Invoke(o, arguments)
```

The Invoke method traps all the exceptions thrown in the called method and converts them into TargetInvocationException; you must check the InnerException property of the caught exception to retrieve the real exception:

```
Try
   mi.Invoke(o, arguments)
Catch ex As TargetInvocationException
   Console.WriteLine(ex.InnerException.Message)
Catch ex As Exception
   Console.WriteLine(ex.Message)
End Try
```

The InvokeMember Method

In some cases, you might find it easier to set properties dynamically and invoke methods by means of the Type object's InvokeMember method. This method takes the name of the member; a flag that says whether it's a field, property, or method; the object for which the member should be invoked; and an array of Objects for the arguments if there are any. Here are a few examples:

```
' Create an instance of the Person type using InvokeMember.
Dim type As Type = Assembly.GetExecutingAssembly().GetType("MyApp.Person")
Dim arguments() As Object = {"John", "Evans"}
Dim obj As Object = type.InvokeMember("", BindingFlags.CreateInstance,_
   Nothing, Nothing, arguments)

' Set the FirstName field.
Dim args() As Object = {"Francesco"}        ' One argument
type.InvokeMember("FirstName", BindingFlags.SetField, Nothing, obj, args)
' Read the FirstName field. (Pass Nothing for the argument array.)
Dim value As Object = type.InvokeMember("FirstName", BindingFlags.GetField, _
   Nothing, obj, Nothing)

' Set the Age property, create the argument array on the fly.
type.InvokeMember("Age", BindingFlags.SetProperty, Nothing, obj, _
   New Object() {35S})

' Call the SendEMail method.
Dim args2() As Object = {"This is a message", 2}
type.InvokeMember("SendEmail", BindingFlags.InvokeMethod, Nothing, obj, args2)
```

It is very important that you pass the correct value for the BindingFlags argument. All the examples shown so far access public instance members, but you must explicitly add the NonPublic and/or Static modifiers if the member is private or static:

```
' Read the m_Age private field.
value = type.InvokeMember("m_Age", BindingFlags.GetField Or BindingFlags.NonPublic _
   Or BindingFlags.Instance, Nothing, obj, Nothing)
```

When you invoke a static member, you must pass Nothing in the second to last argument. The same rule applies when you use InvokeMember to call a constructor method because you don't yet have a valid instance in that case.

The InvokeMember method does a case-sensitive search for the member with the specified name, but it's quite forgiving when it matches the type of the arguments because it will perform any necessary conversion for you if the types don't correspond exactly. You can change this default behavior by means of the BindingFlags.IgnoreCase (for case-insensitive searches) and the BindingFlags.ExactBinding (for exact type matches) values.

InvokeMember works correctly if one or more arguments are passed by reference. For example, if the SendEmail method would take the priority in a ByRef argument, on return from the method call the args2(1) element would contain the new value assigned to that argument.

Even though InvokeMember can make your code more concise—because you don't have to get a reference to a specific FieldInfo, PropertyInfo, or MethodInfo object—it surely doesn't make your code faster. In fact, the InvokeMember method must perform two distinct operations internally: the discovery phase (looking for the member with the specified signature) and the execution phase. If you use InvokeMember to call the same method a hundred times, it will "rediscover" the same method a hundred times, which clearly adds overhead that you can avoid if you reflect on the member once and then access the member through a FieldInfo, PropertyInfo, or MethodInfo object. For this reason, you shouldn't use InvokeMember when repeatedly accessing the same member, especially in time-critical code.

Creating a Universal Comparer

As you might recall from Chapter 10, "Interfaces," you implement the IComparer interface in auxiliary classes that work as comparers for other types, whether they are .NET types or custom types you've defined. The main problem with comparer types is that you must define a distinct comparer type for each possible sort criterion. Clearly, this requirement can soon become a nuisance. Reflection gives you the opportunity to implement a *universal comparer*, a class capable of working with any type of object and any combination of fields and properties and that supports both ascending and descending sorts.

Before discussing how the UniversalComparer type works, let me show you how you can use it. You create a UniversalComparer instance by passing its constructor a string argument that resembles an ORDER BY clause in SQL.

```
Dim persons() As Person = Nothing
' Init the array here.
…
' Sort the array on the LastName and FirstName fields.
Dim comp As New UniversalComparer(Of Person)("LastName, FirstName ")
Array.Sort(Of Person)(persons, comp)
```

You can even sort in descending mode separately on each field:

```
Dim comp As New UniversalComparer(Of Person)("LastName DESC, FirstName DESC")
Array.Sort(Of Person)(persons, comp)
```

Not surprisingly, the UniversalComparer class relies heavily on reflection to perform its magic. Here's its complete source code:

```
Public Class UniversalComparer(Of T)
   Implements IComparer, IComparer(Of T)

   Private sortKeys() As SortKey

   Public Sub New(ByVal sort As String)
      Dim type As Type = GetType(T)
      ' Split the list of properties.
      Dim props() As String = sort.Split(","c)
      ' Prepare the array that holds information on sort criteria.
      ReDim sortKeys(props.Length - 1)

      ' Parse the sort string.
      For i As Integer = 0 To props.Length - 1
         ' Get the Nth member name.
         Dim memberName As String = props(i).Trim()
         If memberName.ToLower().EndsWith(" desc") Then
            ' Discard the DESC qualifier.
            sortKeys(i).Descending = True
            memberName = memberName.Remove(memberName.Length - 5).TrimEnd()
         End If
         ' Search for a field or a property with this name.
         sortKeys(i).FieldInfo = type.GetField(memberName)
         If sortKeys(i).FieldInfo Is Nothing Then
            sortKeys(i).PropertyInfo = type.GetProperty(memberName)
         End If
      Next
   End Sub

   ' Implementation of IComparer.Compare
   Public Function Compare(ByVal o1 As Object, ByVal o2 As Object) As Integer _
         Implements IComparer.Compare
      Return Compare(CType(o1, T), CType(o2, T))
   End Function
```

```
' Implementation of IComparer(Of T).Compare
Public Function Compare(ByVal o1 As T, ByVal o2 As T) As Integer _
      Implements IComparer(Of T).Compare
   ' Deal with simplest cases first.
   If o1 Is Nothing Then
      ' Two null objects are equal.
      If o2 Is Nothing Then Return 0
      ' A null object is less than any non-null object.
      Return -1
   ElseIf o2 Is Nothing Then
      ' Any non-null object is greater than a null object.
      Return 1
   End If

   ' Iterate over all the sort keys.
   For i As Integer = 0 To sortKeys.Length - 1
      Dim value1 As Object, value2 As Object
      Dim sortKey As SortKey = sortKeys(i)
      ' Read either the field or the property.
      If sortKey.FieldInfo IsNot Nothing Then
         value1 = sortKey.FieldInfo.GetValue(o1)
         value2 = sortKey.FieldInfo.GetValue(o2)
      Else
         value1 = sortKey.PropertyInfo.GetValue(o1, Nothing)
         value2 = sortKey.PropertyInfo.GetValue(o2, Nothing)
      End If

      Dim res As Integer
      If value1 Is Nothing And value2 Is Nothing Then
         ' Two null objects are equal.
         res = 0
      ElseIf value1 Is Nothing Then
         ' A null object is always less than a non-null object.
         res = -1
      ElseIf value2 Is Nothing Then
         ' Any object is greater than a null object.
         res = 1
      Else
         ' Compare the two values, assuming that they support IComparable.
         res = DirectCast(value1, IComparable).CompareTo(value2)
      End If

      ' If values are different, return this value to caller.
      If res <> 0 Then
         ' Negate it if sort direction is descending.
         If sortKey.Descending Then res = -res
         Return res
      End If
   Next
   ' If we get here, the two objects are equal.
   Return 0
End Function

' Nested type to store detail on sort keys
Private Structure SortKey
   Public FieldInfo As FieldInfo
```

```
      Public PropertyInfo As PropertyInfo
      ' True if sort is descending.
      Public Descending As Boolean
   End Structure
End Class
```

As the comments in the source code explain, the universal comparer supports comparisons on both fields and properties. Because this class uses reflection intensively, it isn't as fast as a more specific comparer can be, but in most cases the speed difference isn't noticeable.

Dynamic Registration of Event Handlers

Another programming technique you can implement through reflection is the dynamic registration of an event handler. For example, let's say that the Person class exposes a GotEmail event and you have an event handler in the MainModule type:

```
Public Class Person
   Event GotEmail(ByVal sender As Object, ByVal e As EventArgs)
   …
   ' A method that fires the GotEmail event
   Sub SendEmail(ByVal text As String, Optional ByVal priority As Integer = 1)
      …
      Dim e As New GotEmailEventArgs(text, priority)
      RaiseEvent GotEmail(Me, e)
   End Sub
End Class

Module MainModule
   Sub GotEmail_Handler(ByVal sender As Object, ByVal e As GotEmailEventArgs)
      Console.WriteLine("GotEmail event fired")
   End Sub
   …
End Module
```

Here's the code that registers the procedure for this event, using reflection exclusively:

```
' obj and type initialized as in previous examples…
' Get a reference to the GotEmail event.
Dim ei As EventInfo = type.GetEvent("GotEmail")
' Get a reference to the delegate that defines the event.
Dim handlerType As Type = ei.EventHandlerType
' Create a delegate of this type that points to a method in this module.
Dim handler As [Delegate] = [Delegate].CreateDelegate( _
    handlerType, GetType(MainModule), "GotEmail_Handler")
' Register this handler dynamically.
ei.AddEventHandler(obj, handler)
' Call the method that fires the event, using reflection.
Dim args() as Object = {"Hello Joe", 2}
Type.InvokeMember("SendEmail", BindingFlags.InvokeMethod, Nothing, obj, args)
```

A look at the console window proves that the EventHandler procedure in the MainModule type was invoked when the code in the Person.SendEmail method raised the GotEmail event. If the event handler is an instance method, the second argument to the Delegate.CreateDelegate method must be an instance of the class that defines the method; if the event handler is a static method (as in the previous example), this argument must be a Type object corresponding to the class where the method is defined.

The previous code doesn't really add much to what you can do by registering an event by means of the AddHandler operator. But wait, there's more. To show how this technique can be so powerful, I must make a short digression on delegates.

Delegate Covariance and Contravariance in C# 2.0

.NET Framework 2.0 has enhanced delegates with two important features: covariance and contravariance. Unfortunately, however, these features are available only in C#, and therefore I am forced to illustrate these concepts in that language.

Both these features relax the requirement that a delegate object must match exactly the signature of its target method. More specifically, delegate covariance means that you can have a delegate point to a method with a return value that inherits from the return type specified by the delegate. Let's say we have the following delegate:

```
// A delegate that can point to a method that takes a TextBox and returns an object.
delegate object GetControlData(TextBox ctrl);
```

The GetControlData delegate specifies object as the return value; therefore, the covariance property tells that this delegate can point to any method that takes a TextBox control, regardless of the method's return value, because all .NET types inherit from System.Object. The only requirement is that the method actually returns something; therefore, you can't have this delegate point to a C# void method (a Sub method, in Visual Basic parlance). For example, a GetControlData delegate might point to the following method because the String type inherits from System.Object:

```
// A function that takes a TextBox control and returns a String
string GetText(TextBox ctrl)
{ return ctrl.Text; }
```

Delegate contravariance means that a delegate can point to a method with an argument that is a base class of the argument specified in the delegate's signature. For example, a GetControlData delegate might point to a method that takes one argument of the Control or Object type because both these types are base classes for the TextBox argument that appears in the delegate:

```
// A function that takes a Control and returns an Object value.
object GetTag(Control ctrl)
{ return ctrl.Tag; }
```

It's important to realize that covariance and contravariance relax the constraint that a delegate can point only to a method with a signature that doesn't exactly match the delegate's signature, but they don't make the code less robust because no type mismatch exception can occur at run time.

Delegate Covariance and Contravariance in Visual Basic 2005

Don't look for delegate covariance and contravariance in Visual Basic documentation because you won't find any information. As a matter of fact, Visual Basic 2005 doesn't support these features. Period.

Well, not exactly. Granted, Visual Basic doesn't support these features directly, but you can achieve them nevertheless. Covariance and contravariance are supported at the CLR level, and you can create a delegate that leverages both of them through reflection. Let's say you have the following delegate and the following method in a Windows Forms class:

```
Delegate Function GetControlData(ByVal ctrl As TextBox) As Object

Function GetText(ByVal ctrl As Control) As String
   Return ctrl.Text
End Function
```

You know that you can't create a GetControlData delegate that points to the GetText method directly in Visual Basic because the language supports neither covariance nor contravariance. However, you can create a MethodInfo object that points to the GetText method and then pass this object to the Delegate.CreateDelegate static method:

```
' The target method
Dim method As MethodInfo = Me.GetType().GetMethod("GetText")
' Build the delegate through reflection.
Dim deleg As GetControlData = DirectCast([Delegate].CreateDelegate( _
   GetType(GetControlData), Me, method), GetControlData)
' Show that the delegate works correctly.
Console.WriteLine(deleg(Me.TextBox1))    ' Displays the TextBox1.Text property.
```

This code is only marginally slower than the C# counterpart, but this isn't a serious issue because you typically create a delegate once and use it repeatedly.

The most interesting application of this feature is the ability to have an individual method handle all the events coming from one or more objects, provided that the event has the canonical .NET syntax (sender, e), where the second argument can be any type that derives from EventArgs. Consider the following event handler:

```
' (Inside a Form class)
Sub MyEventHandler(ByVal sender As Object, ByVal e As EventArgs)
   Console.WriteLine("An event has fired")
End Sub
```

The following code can make all the events exposed by an object point to the "universal handler":

```
' (Inside the same Form class…)
' The control we want to trap events from
Dim ctrl As Object = TextBox1
For Each ei As EventInfo In ctrl.GetType().GetEvents()
   Dim handlerType As Type = ei.EventHandlerType
   ' The universal event handler method
   Dim method As MethodInfo = Me.GetType().GetMethod("MyEventHandler")
   ' Leverage contravariance to create a delegate that points to the method.
   Dim handler As [Delegate] = [Delegate].CreateDelegate(handlerType, Me, method)
   ' Use reflection to register the event.
   ei.AddEventHandler(ctrl, handler)
Next
' Prove that it works by causing a TextChanged event.
ctrl.Text &= "*"
```

This code proves that it is technically possible to use reflection to have all the events of an object point to an individual handler, but this technique doesn't look very promising. After all, the MyEventHandler method has no means to understand which event was fired. To get that information, we need to do more.

A Universal Event Handler

What we need is an object that is able to "mediate" between the event source and the object where the event is handled. Writing this object requires some significant code, but the result is well worth the effort. The EventInterceptor class exposes only one event, ObjectEvent, defined by the ObjectEventHandler delegate:

```
Public Delegate Sub ObjectEventHandler(ByVal sender As Object, ByVal e As ObjectEventArgs)

Public Class EventInterceptor
   ' The public event
   Public Event ObjectEvent As ObjectEventHandler

   ' This is invoked from inside the EventInterceptorHandler auxiliary class.
   Protected Sub OnObjectEvent(ByVal e As ObjectEventArgs)
      RaiseEvent ObjectEvent(Me, e)
   End Sub
   …
End Class
```

The EventInterceptor class uses the nested EventInterceptorHandler type to trap events coming from the object source. More precisely, an EventInterceptorHandler instance is created for each event that the event source can raise. The EventInterceptor class supports multiple event sources; therefore, the number of EventInterceptorHandler instances can be quite high: for example, if you trap the events coming from 20 TextBox controls, the EventInterceptor object will create as many as 1,540 EventInterceptorHandler instances because each TextBox control exposes 77 events. For this reason, the AddEventSource method supports a third argument that enables you to specify which events should be intercepted:

```
   Public Sub AddEventSource(ByVal eventSource As Object, _
         ByVal includeChildren As Boolean, ByVal filterPattern As String)
      For Each ei As EventInfo In eventSource.GetType().GetEvents()
         ' Skip this event if its name doesn't match the pattern.
         If Not String.IsNullOrEmpty(filterPattern) AndAlso Not _
            Regex.IsMatch(ei.Name, "^" & filterPattern & "$") Then Continue For

         ' Get the signature of the underlying delegate.
         Dim mi As MethodInfo = ei.EventHandlerType.GetMethod("Invoke")
         Dim pars() As ParameterInfo = mi.GetParameters()
         ' Check that event signature is in the form (sender, e).
         If mi.ReturnType.FullName = "System.Void" AndAlso pars.Length = 2 _
               AndAlso pars(0).ParameterType Is GetType(Object) AndAlso _
               GetType(EventArgs).IsAssignableFrom(pars(1).ParameterType) Then
            ' Create an EventInterceptorHandler that handles this event.
            Dim interceptor As New EventInterceptorHandler(eventSource, ei, Me)
         End If
      Next
      ' Recurse on child controls if so required.
      If TypeOf eventSource Is Control AndAlso includeChildren Then
         For Each ctrl As Control In DirectCast(eventSource, Control).Controls
            AddEventSource(ctrl, includeChildren, filterPattern)
         Next
      End If
   End Sub
   …
End Class     ' End of EventInterceptor class
```

The EventInterceptorHandler nested class does a very simple job: it uses reflection to register its EventHandler method as a listener for the specified event coming from the specified event source. When the event is fired, the EventHandler method calls back the OnObjectEvent method in the parent EventInterceptor object, which in turn fires the ObjectEvent event:

```
Private Class EventInterceptorHandler
   ' The event being intercepted
   Public ReadOnly EventInfo As EventInfo
   ' The parent EventInterceptor
   Public ReadOnly Parent As EventInterceptor

   Public Sub New(ByVal eventSource As Object, ByVal eventInfo As EventInfo, _
         ByVal parent As EventInterceptor)
      Me.EventInfo = eventInfo
      Me.Parent = parent
      ' Create a delegate that points to the EventHandler method.
      Dim method As MethodInfo = Me.GetType().GetMethod("EventHandler")
      Dim handler As [Delegate] = _
         [Delegate].CreateDelegate(eventInfo.EventHandlerType, Me, method)
      ' Register the event.
      eventInfo.AddEventHandler(eventSource, handler)
   End Sub

   Public Sub EventHandler(ByVal sender As Object, ByVal e As EventArgs)
      ' Notify the parent EventInterceptor object.
      Dim objEv As New ObjectEventArgs(sender, EventInfo.Name, e)
      Parent.OnObjectEvent(objEv)
   End Sub
End Class
```

Here's how a Windows Forms application can use the EventInterceptor object to get a notification when any event of any control fires:

```
Dim WithEvents Interceptor As New EventInterceptor

Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs) Handles MyBase.Load
```

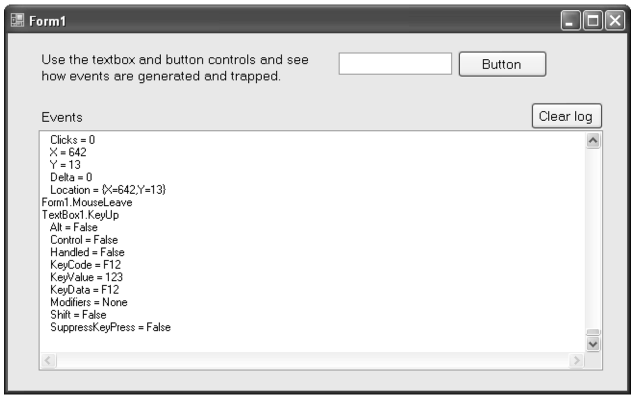
```
   Interceptor.AddEventSource(Me, True, "")
End Sub

Private Sub Interceptor_ObjectEvent(ByVal sender As Object, ByVal e As ObjectEventArgs) _
      Handles Interceptor.ObjectEvent
   Dim msg As String = String.Format("Event {0} from control {1}", _
      DirectCast(e.EventSource, Control).Name, e.EventName)
   Debug.WriteLine(msg)
End Sub
```

You can limit the number of events that you receive by passing a regular expression pattern to the AddEventSource method:

```
   ' Trap only xxxChanged events.
   Interceptor.AddEventSource(Me, True, ".+Changed")
   ' Trap only mouse and keyboard events.
   Interceptor.AddEventSource(Me, True, "(Mouse|Key).+)")
```

For more information, see the source code of the complete demo program. (See Figure 18-2.)


Form1
Use the textbox and button controls and see how events are generated and trapped.
Button
Events
Clear log

```
  Clicks = 0
  X = 642
  Y = 13
  Delta = 0
  Location = {X=642,Y=13}
Form1.MouseLeave
TextBox1.KeyUp
  Alt = False
  Control = False
  Handled = False
  KeyCode = F12
  KeyValue = 123
  KeyData = F12
  Modifiers = None
  Shift = False
  SuppressKeyPress = False
```

Figure 18-2 The EventInterceptor demo application

Scheduling a Sequence of Actions

Reflection allows you to implement techniques that would be very difficult (and sometimes impossible) to implement using a more traditional approach. For example, the ability to invoke methods through MethodInfo objects gives you the ability to deal with call sequences as if they were just another data type that your application processes. To see what I mean, let's say that your application must perform a series of actions—for example, file creation, registry manipulation, variable assignments—as an atomic operation. If any one of the involved actions fails, all the actions performed so far should be undone in an orderly manner: files should be deleted, registry keys should be restored, variables should be assigned their original value, and so forth.

Implementing an undo strategy in the most general case isn't a simple task, especially if some of the actions are performed only conditionally when other conditions are met. What you need is a generic solution to this problem, and you'll see how elegantly you can solve this programming task through reflection. To begin with, let's define an Action class, which represents a method—either a static or an instance method:

```
Public Class Action
   Public ReadOnly Message As String       ' Description of the action
   Public ReadOnly [Object] As Object      ' Instance on which the method is called
   Public ReadOnly Method As MethodInfo    ' The method to be invoked
   Public ReadOnly Arguments() As Object   ' Arguments for the method

   ' Second argument can be an object (for instance methods) or a Type (for static methods).
   Public Sub New(ByVal message As String, ByVal obj As Object, _
         ByVal methodName As String, ByVal ParamArray arguments() As Object)
      Me.Message = message
      Me.Arguments = arguments
      ' Determine the type this method belongs to.
      Dim type As Type = TryCast(obj, Type)
      If type Is Nothing Then
         Me.Object = obj
         type = obj.GetType()
      End If
      ' Prepare the list of argument types, to call GetMethod without any ambiguity.
      Dim argTypes(arguments.Length - 1) As Type
      For index As Integer = 0 To arguments.Length - 1
         If arguments(index) IsNot Nothing Then
            argTypes(index) = arguments(index).GetType()
         End If
      Next
      ' Retrieve the actual MethodInfo object, throw an exception if not found.
      Me.Method = type.GetMethod(methodName, argTypes)
      If Me.Method Is Nothing Then
         Throw New ArgumentException("Missing method")
      End If
   End Sub

   ' Execute this method.
   Public Sub Execute()
      Me.Method.Invoke(Me.Object, Me.Arguments)
   End Sub
End Class
```

Instead of performing a method directly, you can create an Action instance and then invoke its Execute method:

```
' Create c:\backup directory.
Dim act As New Action("Create c:\backup directory", _
   GetType(Directory), "CreateDirectory", "c:\backup")
act.Execute()
```

Of course, executing a method in this way doesn't bring any benefit. You see the power of this technique, however, if you define another type that works as a container for Action instances and that can also remember the "undo" action for each method being executed:

```
Public Class ActionSequence
   ' The parallel lists of actions and undo actions
   Private Actions As New List(Of Action)
   Private UndoActions As New List(Of Action)
   ' This delegate must point to a method that takes a string.
   Private DisplayMethod As Action(Of String)

   ' The constructor takes a delegate to a method that can output a message.
   Public Sub New(ByVal displayMethod As Action(Of String))
      Me.DisplayMethod = displayMethod
   End Sub

   ' Add an action and an undo action to the list.
   Public Sub Add(ByVal action As Action, ByVal undoAction As Action)
      Actions.Add(action)
      UndoActions.Add(undoAction)
   End Sub

   ' Insert an action and an undo action at a specific index in the list.
   Public Sub Insert(ByVal index As Integer, ByVal action As Action, ByVal undoAction As Action)
      Actions.Insert(index, action)
      UndoActions.Insert(index, undoAction)
   End Sub

   ' Execute all pending actions, return true if no exception occurred.
   Public Function Execute(ByVal ignoreExceptions As Boolean) As Boolean
      ' This is the list of undo actions to execute in case of error.
      Dim undoSequence As New ActionSequence(Me.DisplayMethod)

      For index As Integer = 0 To Actions.Count - 1
         Dim act As Action = Actions(index)
         ' Skip over null actions.
         If act Is Nothing Then Continue For

         Try
            ' Display the message and execute the action.
            DisplayMessage(act.Message)
            act.Execute()
            ' If successful, remember the undo action. The undo action is placed
            ' in front of all others so that it will be the first to be executed in case of error.
            undoSequence.Insert(0, UndoActions(index), Nothing)
         Catch ex As TargetInvocationException
            ' Ignore exceptions if so required.
            If ignoreExceptions Then Continue For
            ' Display the error message.
            DisplayMessage("ERROR: " & ex.InnerException.Message)
            ' Perform the undo sequence. (Ignore exceptions while undoing.)
            DisplayMessage("UNDOING OPERATIONS...")
            undoSequence.Execute(True)
            ' Signal that an exception occurred.
            Return False
         End Try
      Next
      ' Signal that no exceptions occurred.
      Return True
   End Function

   ' Report a message through the delegate passed to the constructor.
   Private Sub DisplayMessage(ByVal text As String)
```

```
      If Me.DisplayMethod IsNot Nothing Then
         Me.DisplayMethod(text)
      End If
   End Sub
End Class
```

The constructor of the ActionSequence type takes an Action(Of String) object, which is a delegate to a Sub method that takes a string as an argument. This method will be used to display all the messages that are produced during the action sequence: it can point to a method such as the Console.WriteLine method (to display messages in the console window), the WriteLine method of a StreamWriter object (to write messages to a log file), the AppendText method of a TextBox control (to display messages inside a TextBox control), or a method you define in your application:

```
' Prepare to write diagnostic messages to a log file.
Dim sw As New StreamWriter("c:\logfile.txt")
Dim actionSequence As New ActionSequence(AddressOf sw.WriteLine)
…
' Close the stream when you're done with the ActionSequence object.
sw.Close()
```

The following code shows how you can schedule a sequence of actions and undo them if an exception is thrown during the process:

```
' Schedule the creation of c:\backup directory.
Dim actionSequence As New ActionSequence(AddressOf Console.WriteLine)
Dim act As New Action("Create c:\backup directory", GetType(Directory), _
   "CreateDirectory", "c:\backup")
Dim undoAct As New Action("Delete c:\backup directory", GetType(Directory), _
   "Delete", "c:\backup", True)
actionSequence.Add(act, undoAct)

' Create a readme.txt file in the c:\ root directory.
Dim contents As String = "Instructions for myapp.exe..."
act = New Action("Create the c:\myapp_readme.txt", GetType(File), _
   "WriteAllText", "c:\myapp_readme.txt", contents)
' Notice that this action has no undo action.
actionSequence.Add(act, Nothing)

' Move the readme file to the backup directory.
act = New Action("Move the c:\myapp_readme.txt to c:\backup\readme.txt", GetType(File), _
   "Move", "c:\myapp_readme.txt", "c:\backup\readme.txt")
undoAct = New Action("Move c:\backup\readme.txt to c:\myapp_readme.txt", GetType(File), _
   "Move", "c:\backup\readme.txt", "c:\myapp_readme.txt")
actionSequence.Add(act, undoAct)

' Execute the action sequence. (False means that an exception undoes all actions.)
actionSequence.Execute(False)
```

After running the previous code snippet, you should find a new c:\backup directory containing the files readme.txt and win.ini. To see how the ActionSequence type behaves in case of error, delete the c:\backup directory and intentionally cause an error in the sequence by attempting to copy a file that doesn't exist:

```
' (Insert the lines in bold type before the call to the Execute method.)
…
' Copy the c:\missing.txt file to the c:\backup directory.
act = New Action("Copy c:\missing.txt to c:\backup", GetType(File), _
   "Copy", "c:\missing.text", "c:\backup\missing.txt")
undoAct = New Action("Delete c:\backup\missing.text", GetType(File), _
   "Delete", "c:\backup\missing.text")
actionSequence.Add(act, Nothing)

' Execute the action sequence. (False means that an exception undoes all actions.)
actionSequence.Execute(False)
```

The intentional error causes the ActionSequence object to undo all the actions before the one that caused the exception, and in fact at the end of the process you won't find any c:\backup directory, as confirmed by the text that appears in the console window:

```
Create c:\backup directory
Create the c:\myapp_readme.txt
Move the c:\myapp_readme.txt to c:\backup\readme.txt
Copy c:\missing.txt to c:\backup
ERROR: Could not find file 'c:\missing.text'.
UNDOING OPERATIONS...
Move the c:\backup\readme.txt file back to c:\myapp_readme.txt
Delete c:\backup directory
```

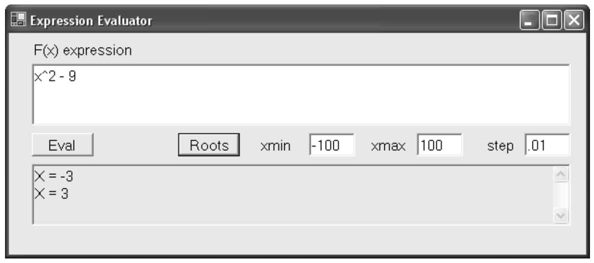
In this example, I used the ActionSequence type to undo a sequence of actions that are hard-coded in the program, but I could have used a similar technique to implement an Undo menu command in your applications. Or I could have read the series of actions from a file instead, to implement undoable scripts.

You can expand the ActionSequence type with new features. For example, you might have the series of actions be performed on a background thread (see Chapter 20, "Threads," for more information) and specify multiple undo methods for a given action. You might add properties to the Action type to specify whether a failed method should abort the entire sequence. Also, you might extend the Action class with the ability to create new instances (that is, to call constructors in addition to regular methods) and to pass instances created in this way as arguments to other methods down in the action sequence. As usual, the only limit is your imagination.

On-the-Fly Compilation

Earlier in this chapter, I mentioned the System.Reflection.Emit namespace, which has classes that let you create an assembly on the fly. The .NET Framework uses these classes internally in a few cases—for example, when you pass the RegexOptions.Compiled option to the constructor of the Regex object (see Chapter 14, "Regular Expressions"). Using reflection emit, however, isn't exactly the easiest .NET programming task, and I'm glad I've never had to use it heavily in a real-world application.

Nevertheless, at times the ability to create an assembly out of thin air can be quite tantalizing because it opens up a number of programming techniques that are otherwise impossible. For example, consider building a routine that takes a math expression entered by the end user (as a string), evaluates it, and returns the result. In the section titled "Parsing and Evaluating Expressions" in Chapter 14, I showed how you can parse and evaluate an expression at run time, but that approach is several orders of magnitude slower than evaluating a compiled expression is and can't be used in time-critical code, such as for doing function plotting or finding the roots of a higher-degree equation (see Figure 18-3). In this case, your best option is to generate the source code of a Visual Basic program, compile it on the fly, and then instantiate one of its classes.



```
x^2 - 9
-100
100
.01
X = -3
X = 3
```

Figure 18-3 The demo application, which uses on-the-fly compilation to evaluate functions and find the roots of any equation that uses the X variable

The types that allow us to compile an assembly at run time are in the Microsoft.VisualBasic namespace (or in the Microsoft.CSharp namespace, if you want to generate and compile C# source code) and in the System.CodeDom.Compiler namespace, so you need to add proper Imports statements to your code to run the code samples that follow.

The first thing to do is generate the source code for the program to be compiled dynamically. In the expression evaluator demo application, such source code is obtained by inserting the expression that the end user enters in the txtExpression field in the middle of the Eval method of an Evaluator public class:

```
Dim source As String = String.Format( _
   "Imports Microsoft.VisualBasic{0}" _
   & "Imports System.Math{0}" _
   & "Public Class Evaluator{0}" _
   & "   Public Function Eval(ByVal x As Double) As Double{0}" _
   & "      Return {1}{0}" _
   & "   End Function{0}" _
   & "End Class{0}", _
   ControlChars.CrLf, txtExpression.Text)
```

Next, you create a CompilerParameters object (in the System.CodeDom.Compiler namespace) and set its properties; this object broadly corresponds to the options you'd pass to the VBC command-line compiler:

```
Dim params As New CompilerParameters
' Generate a DLL, not an EXE executable.
' (Not really necessary because False is the default.)
params.GenerateExecutable = False
```

```
#If DEBUG Then
   ' Include debug information.
   params.IncludeDebugInformation = True
   ' Debugging works if we generate an actual DLL and keep temporary files.
   params.TempFiles.KeepFiles = True
   params.GenerateInMemory = False
#Else
   ' Treat warnings as errors, don't keep temporary source files.
   params.TreatWarningsAsErrors = True
   params.TempFiles.KeepFiles = False
   ' Optimize the code for faster execution.
   params.CompilerOptions = "/Optimize+"
   ' Generate the assembly in memory.
   params.GenerateInMemory = True
#End If
   ' Add a reference to necessary strong-named assemblies.
   params.ReferencedAssemblies.Add("Microsoft.VisualBasic.Dll")
   params.ReferencedAssemblies.Add("System.Dll")
```

The preceding code snippet shows the typical actions you perform to prepare a CompilerParameters object, as well as its most important properties. The statements inside the #If block are especially interesting. You can include debug information in a dynamic assembly and debug it from inside Visual Studio by setting the IncludeDebugInformation property to True. To enable debugging, however, you must generate an actual .dll or .exe file (GenerateInMemory must be False) and must not delete temporary files at the end of the compilation process (the KeepFiles property of the TempFiles collection must be True). If debugging is correctly enabled, you can force a break in the generated assembly by inserting the following statement in the code you generate dynamically:

```
System.Diagnostics.Debugger.Break()
```

You are now ready to compile the assembly:

```
' Create the VB compiler.
Dim provider As New VBCodeProvider
Dim compRes As CompilerResults = provider.CompileAssemblyFromSource(params, source)

' Check whether we have errors.
If compRes.Errors.Count > 0 Then
   ' Gather all error messages and display them.
   Dim msg As String = ""
   For Each compErr As CompilerError In compRes.Errors
      msg &= compErr.ToString & ControlChars.CrLf
   Next
   MessageBox.Show(msg, "Compilation Failed", MessageBoxButtons.OK, MessageBoxIcon.Error)
Else
   ' Compilation was successful.
   …
End If
```

If the compilation was successful, you use the CompilerResults.CompiledAssembly property to get a reference to the created assembly. Once you have this Assembly object, you can create an instance of its Evaluator class and invoke its Eval method by using standard reflection techniques:

```
Dim asm As Assembly = compRes.CompiledAssembly
Dim evaluator As Object = asm.CreateInstance("Evaluator")
Dim evalMethod As MethodInfo = evaluator.GetType.GetMethod("Eval")
Dim args() As Object = {CDbl(123)}    ' Pass x = 123
Dim result As Object = evalMethod.Invoke(evaluator, args)
```

Notice that you can't reference the Evaluator class by a typed variable because this class (and its container assembly) doesn't exist yet when you compile the main application. For this reason, you must use reflection both to create an instance of the class and to invoke its members.

Another tricky thing to do when applying this technique is to have the dynamic assembly call back a method in a class defined in the main application by means of reflection, for example, to let the main application update a progress bar during a lengthy routine. Alternatively, you can define a public interface in a DLL and must have the class in the main application implement the interface; being defined in a DLL, the dynamic assembly can create an interface variable and therefore it can call methods in the main application through that interface.

You must be aware of another detail when you apply on-the-fly compilation in a real application: once you load the dynamically created assembly, that assembly will take memory in your process until the main application ends. In most cases, this problem isn't serious and you can just forget about it. But you can't ignore it if you plan to build many assemblies on the fly. The only solution to this problem is to create a separate AppDomain, load the dynamic assembly in that AppDomain, use the classes in the assembly, and finally unload the AppDomain when you don't need the assembly any longer. On the other hand, loading the assembly in another AppDomain means that you can't use reflection to manage its types (reflection works only with types in the same AppDomain as the caller). Please see the demo application in the companion code for a solution to this complex issue.

Performance Considerations

I have warned about the slow performance of reflection-based techniques often in this chapter. Using reflection to invoke methods is similar to using the late-binding techniques that are available in script languages such as Microsoft Visual Basic Scripting Edition (VBScript), in Visual Basic 6 when you use a Variant variable, or even in Visual Basic 2005 when you invoke a method using an Object variable and Option Strict is Off.

In general, invoking a method by using reflection is many times slower than a direct call is, and therefore you shouldn't use these techniques in time-critical portions of your application. In some scenarios, however, you need to defer the decision about which method to call until run time, and therefore a direct call is out of the question. Even then, reflection should be your last resort and should be used only if you can't solve the problem with another technique based on indirection, for example, an interface or a delegate.

If you decide to use reflection and you must invoke a method more than once or twice, you should use Type.GetMethod to get a reference to a MethodInfo object and then use the MethodInfo.Invoke method to do the actual call, rather than using the Type.InvokeMember method because the former technique requires that you perform the discovery phase only once.

Don't use the BindingFlags.IgnoreCase value with the Get*Xxxx* method (singular form), if you know the exact spelling of the member you're looking for, and specify the BindingFlags .ExactBinding value if possible because it speeds up the search. The latter flag suppresses implicit type conversions; therefore, you must supply the exact type of each argument:

```
' This code doesn't work—the GetMethod method returns Nothing.
' You must either use Integer instead of Short in the argTypes signature
' or drop the BindingFlags.ExactBinding bit in the GetMethod call.
Dim argTypes() As Type = {GetType(Char), GetType(Short)}
Dim mi As MethodInfo = GetType(String).GetMethod("IndexOf", BindingFlags.ExactBinding Or _
   BindingFlags.Public Or BindingFlags.Instance, Nothing, argTypes, Nothing)
```

.NET Framework 2.0 improves performance in many ways. For example, in previous versions of the .NET Framework, a call to the Type.Get*Xxxx* (singular) adds a noticeable overhead because all the type's members are queried anyway, as if Type.GetMembers were called. The results from this first call are cached, so at least you pay this penalty only once, but this approach has a serious issue: if you reflect on many types, all the resulting MemberInfo objects are kept in memory and are never discarded until the application terminates.

In .NET Framework 2.0, a Type.Get*Xxxx* method (singular form) doesn't cause the exploration of the entire Type object, and therefore execution is faster and memory consumption is kept to a minimum. Also, the cache used by reflection is subject to garbage collection; therefore, type and member information is discarded unless you keep it alive by storing a reference in a Type or MemberInfo field at the class level.

An alternative technique for storing information about a large number of types and members without taxing the memory is based on the RuntimeTypeHandle and RuntimeMethodHandle classes that you can use instead of the Type and MemberInfo classes. Handle-based objects use very little memory, yet they allow you to rebuild a reference to the actual Type or MemberInfo-based object very quickly, as this code demonstrates:

```
' Store information about a method in the Person type.
Dim hType As RuntimeTypeHandle = GetType(Person).TypeHandle
Dim hMethod As RuntimeMethodHandle = GetType(Person).GetMethod("SendEmail").MethodHandle
…
' (Later in the application…)
' Rebuild the Type and MethodBase objects.
Dim ty As Type = Type.GetTypeFromHandle(hType)
Dim mb As MethodBase = MethodBase.GetMethodFromHandle(hMethod, hType)
' Use them as needed.
…
```

Security Issues

A warning about using reflection at run time is in order. As you've seen in previous sections, reflection allows you to access any type and any member, regardless of their scope. Therefore, you can use reflection even to instantiate private types, call private methods, or read private variables.

More precisely, your code can perform these operations if it is fully trusted or at least has ReflectionPermission. All the applications that run from the local hard disk have this permission, whereas applications running from the Internet don't. In general, if you don't have ReflectionPermission, you can perform only the following reflection-related techniques:

- Enumerate assemblies and modules.
- Enumerate public types and obtain information about them and their public members.
- Set public fields and properties and invoke public members.
- Access and enumerate family (Protected) members of a base class of the calling code.
- Access and enumerate assembly (Friend) members from inside the assembly in which the calling code runs.

You see that code can access through reflection only those types and members that it can access directly anyway. For example, a piece of code can't access a private field in another type or invoke a private member in another type, even if that type is inside the same assembly as the calling code. In other words, reflection doesn't give code more power than it already has; it just adds flexibility because of the additional level of indirection that it provides.

This discussion on reflection is important for one reason: never rely on the Private scope keyword to hide confidential data from unauthorized eyes because a malicious user might create a simple application that uses reflection to read your data. If the application runs from the local hard disk, it has ReflectionPermission and can therefore access all the members of your assembly, regardless of whether it's an EXE or a DLL. (If you also consider that decompiling a .NET assembly is as easy, you see that the only way to protect confidential data is by means of cryptography.)

The ability to invoke nonpublic members can be important in some scenarios. For example, in the section titled "The ICloneable Interface" in Chapter 10, I show how a type can implement the Clone method by leveraging the MemberwiseClone protected method, but in some cases you'd like to clone an object for which you don't have any source code. Provided that your application runs in full trust mode and has ReflectionPermission, you can clone any object quite easily with reflection. Here's a reusable routine that performs a (shallow) copy of the object passed as an argument:

```
Function CloneObject(Of T)(ByVal obj As T) As T
   If obj Is Nothing Then
      ' Cloning a null object is easy.
      Return Nothing
   ElseIf TypeOf CObj(obj) Is ICloneable Then
      ' Take advantage of the ICloneable interface, if possible.
      Dim iclone As ICloneable = DirectCast(obj, ICloneable)
      Return CType(iclone.Clone(), T)
   Else
      ' Use reflection if everything else failed.
      ' (Throws if application doesn't have ReflectionPermission.)
      Dim mi As MethodInfo = obj.GetType().GetMethod("MemberwiseClone",  _
         BindingFlags.ExactBinding Or BindingFlags.NonPublic Or BindingFlags.Instance)
      Return CType(mi.Invoke(obj, Nothing), T)
   End If
End Function
```

Assuming that you have a Person type—with the usual FirstName, LastName, and Spouse properties—the following code tests that the CloneObject method works correctly:

```
Dim john As New Person("John", "Evans")
Dim ann As New Person("Ann", "Beebe")
john.Spouse = ann
ann.Spouse = john
' We need no CType or DirectCast, thanks to generics.
Dim john2 As Person = CloneObject(john)
' Prove that it worked.
Console.WriteLine("{0} {1}, spouse is {2} {3}", john2.FirstName, john2.LastName, _
   john2.Spouse.FirstName, john2.Spouse.LastName)
   ' => John Evans, spouse is Ann Beebe
```