Chapter 23. Assembly Loading and Reflection

In this chapter:

This chapter is all about discovering information about types, creating instances of them, and accessing their members when you didn't know anything about them at compile time. The information in this chapter is typically used to create a dynamically extensible application. This is the kind of application for which one company builds a host application and other companies create add-ins to extend the host application. The host can't be built or tested against the add-ins because the add-ins are created by different companies and are likely to be created after the host application has already shipped. This is why the host needs to discover the add-ins at run time.

A dynamically extensible application could take advantage of common language runtime (CLR) hosting and AppDomains as discussed in Chapter 22. The host could run the add-in code in an AppDomain with its own security and configuration settings. The host could also unload the add-in code by unloading the AppDomain. At the end of this chapter, I'll talk a little about how to put all of this stuff together—CLR hosting, AppDomains, assembly loading, type discovery, type instance construction, and reflection—in order to build a robust, secure, and dynamically extensible application.

IMPORTANT

For version 4.5 of the .NET Framework, Microsoft has introduced a new reflection API. The old API had many shortcomings. For example, it did not support LINQ well, it had policies embedded in it that were not correct for some languages, it would sometimes force the loading of assemblies unnecessarily, and it was an overly complex API that offered solutions for problems rarely encountered. The new API fixes all of these issues. However, as of .NET 4.5, the new reflection API is not as complete as the old API. With the new API and some extension methods (in the `System.Reflection.RuntimeReflectionExtensions` class), you can accomplish all you need. Expect additional methods to be added to the new API in future versions of the .NET Framework.

Of course, for desktop applications, the old API still exists so that it doesn't break existing code when re-compiling it. However, the new API is the recommended API going forward, and that is why I explain the new API exclusively in this chapter. For Windows Store Apps (where backward compatibility is not an issue), using the new API is mandatory.

Assembly Loading

As you know, when the just-in-time (JIT) compiler compiles the Intermediate Language (IL) for a method, it sees what types are referenced in the IL code. Then at run time, the JIT compiler uses the assembly's TypeRef and AssemblyRef metadata tables to determine what assembly defines the type being referenced. The AssemblyRef metadata table entry contains all of the parts that make up the strong name of the assembly. The JIT compiler grabs all of these parts—name (without extension or path), version, culture, and public key token—concatenates them into a string, and then attempts to load an assembly matching this identity into the AppDomain (assuming that it's not already loaded). If the assembly being loaded is weakly named, the identity is just the name of the assembly (no version, culture, or public key token information).

Internally, the CLR attempts to load this assembly by using the `System.Reflection.Assembly` class's static `Load` method. This method is publicly documented, and you can call it to explicitly load an assembly into your AppDomain. This method is the CLR equivalent of Win32's `LoadLibrary` function. There are actually several overloaded versions of `Assembly`'s `Load` method. Here are the prototypes of the more commonly used overloads.

```
public class Assembly {

   public static Assembly Load(AssemblyName assemblyRef);

   public static Assembly Load(String assemblyString);

   // Less commonly used overloads of Load are not shown

}
```

Internally, `Load` causes the CLR to apply a version-binding redirection policy to the assembly and looks for the assembly in the global assembly cache (GAC), followed by the application's base directory, private path subdirectories, and codebase locations. If you call `Load` by passing a weakly named assembly, `Load` doesn't apply a version-binding redirection policy to the assembly, and the CLR won't look in the GAC for the assembly. If `Load` finds the specified assembly, it returns a reference to an `Assembly` object that represents the loaded assembly. If `Load` fails to find the specified assembly, it throws a `System.IO.FileNotFoundException`.

NOTE

In some extremely rare situations, you may want to load an assembly that was built for a specific CPU architecture. In this case, when specifying an assembly's identity, you can also include a process architecture part. For example, if my GAC happened to have an IL-neutral and an x86-specific version of an assembly, the CLR would favor the CPU-specific version of the assembly (as discussed in Chapter 3). However, I can force the CLR to load the IL-neutral version by passing the following string to `Assembly`'s `Load` method.

```
"SomeAssembly, Version=2.0.0.0, Culture=neutral,
   PublicKeyToken=01234567890abcde, ProcessorArchitecture=MSIL"
```

Today, the CLR supports five possible values for ProcessorArchitecture: MSIL (Microsoft IL), x86, IA64, AMD64, and Arm.

IMPORTANT

Some developers notice that `System.AppDomain` offers a `Load` method. Unlike `Assembly`'s static `Load` method, AppDomain's `Load` method is an instance method that allows you to load an assembly into the specified AppDomain. This method was designed to be called by unmanaged code, and it allows a host to inject an assembly into a specific AppDomain. Managed code developers generally shouldn't call this method because when AppDomain's `Load` method is called, you pass it a string that identifies an assembly. The method then applies policy and searches the normal places looking for the assembly. Recall that an AppDomain has settings associated with it that tell the CLR how to look for assemblies. To load this assembly, the CLR will use the settings associated with the specified AppDomain, not the calling AppDomain.

However, AppDomain's `Load` method returns a reference to an assembly. Because the `System.Assembly` class isn't derived from `System.MarshalByRefObject`, the assembly object must be marshaled by value back to the calling AppDomain. But the CLR will now use the calling AppDomain's settings to locate the assembly and load it. If the assembly can't be found by using the calling AppDomain's policy and search locations, a `FileNotFoundException` is thrown. This behavior is usually undesirable and is the reason that you should avoid AppDomain's `Load` method.

In most dynamically extensible applications, `Assembly`'s `Load` method is the preferred way of loading an assembly into an AppDomain. However, it does require that you have all of the pieces that make up an assembly's identity. Frequently, developers write tools or utilities (such as ILDasm.exe, PEVerify.exe, CorFlags.exe, GACUtil.exe, SGen.exe, SN.exe, XSD.exe) that perform some kind of processing on an assembly. All of these tools take a command-line argument that refers to the path name of an assembly file (including file extension).

To load an assembly specifying a path name, you call `Assembly`'s `LoadFrom` method.

```
public class Assembly {
   public static Assembly LoadFrom(String path);
   // Less commonly used overloads of LoadFrom are not shown
}
```

Internally, `LoadFrom` first calls `System.Reflection.AssemblyName`'s static `GetAssemblyName` method, which opens the specified file, finds the AssemblyDef metadata table's entry, and extracts the assembly identity information and returns it in a `System.Reflection.AssemblyName` object (the file is also closed). Then, `LoadFrom` internally calls `Assembly`'s `Load` method, passing it the `AssemblyName` object. At this point, the CLR applies a version-binding redirection policy and searches the various locations looking for a matching assembly. If `Load` finds the assembly, it will load it, and an `Assembly` object that represents the loaded assembly will be returned; `LoadFrom` returns this value. If `Load` fails to find an assembly, `LoadFrom` loads the assembly at the path name specified in `LoadFrom`'s argument. Of course, if an assembly with the same identity is already loaded, `LoadFrom` simply returns an `Assembly` object that represents the already loaded assembly.

By the way, the `LoadFrom` method allows you to pass a URL as the argument. Here is an example.

```
Assembly a = Assembly.LoadFrom(@"http://Wintellect.com/SomeAssembly.dll");
```

When you pass an Internet location, the CLR downloads the file, installs it into the user's download cache, and loads the file from there. Note that you must be online or an exception will be thrown. However, if the file has been downloaded previously, and if Windows Internet Explorer has been set to work offline (see Internet Explorer's Work Offline menu item in its File menu), the previously downloaded file will be used, and no exception will be thrown. You can also call `UnsafeLoadFrom`, which can load a web-downloaded assembly, bypassing some security checks.

IMPORTANT

It is possible to have different assemblies on a single machine all with the same identity. Because `LoadFrom` calls `Load` internally, it is possible that the CLR will not load the specified file and instead will load a different file, giving you unexpected behavior. It is highly recommended that each build of your assembly change the version number; this ensures that each version has its own identity, and because of this, `LoadFrom` will now work as expected.

Microsoft Visual Studio's UI designers and other tools typically use `Assembly`'s `LoadFile` method. This method can load an assembly from any path and can be used to load an assembly with the same identity multiple times into a single AppDomain. This can happen as changes to an application's UI are made in the designer/tool and the user rebuilds the assembly. When loading an assembly via `LoadFile`, the CLR will not resolve any dependencies automatically; your code must register with `AppDomain`'s `AssemblyResolve` event and have your event callback method explicitly load any dependent assemblies.

If you are building a tool that simply analyzes an assembly's metadata via reflection (as discussed later in this chapter), and you want to ensure that none of the code contained inside the assembly executes, the best way for you to load an assembly is to use `Assembly`'s `ReflectionOnlyLoadFrom` method, or in some rarer cases, `Assembly`'s `ReflectionOnlyLoad` method. Here are the prototypes of both methods.

```
public class Assembly {
   public static Assembly ReflectionOnlyLoadFrom(String assemblyFile);
   public static Assembly ReflectionOnlyLoad(String assemblyString);
   // Less commonly used overload of ReflectionOnlyLoad is not shown
}
```

The `ReflectionOnlyLoadFrom` method will load the file specified by the path; the strong-name identity of the file is not obtained, and the file is not searched for in the GAC or elsewhere. The `ReflectionOnlyLoad` method will search for the specified assembly looking in the GAC, application base directory, private paths, and codebases. However, unlike the `Load` method, the `ReflectionOnlyLoad` method does not apply versioning policies, so you will get the exact version that you specify. If you want to apply versioning policy yourself to an assembly identity, you can pass the string into `AppDomain`'s `ApplyPolicy` method.

When an assembly is loaded with `ReflectionOnlyLoadFrom` or `ReflectionOnlyLoad`, the CLR forbids any code in the assembly from executing; any attempt to execute code in an assembly loaded with either of these methods causes the CLR to throw an `InvalidOperationException`. These methods allow a tool to load an assembly that was delay-signed, would normally require security permissions that prevent it from loading, or was created for a different CPU architecture.

Frequently, when using reflection to analyze an assembly loaded with one of these two methods, the code will have to register a callback method with `AppDomain`'s `ReflectionOnlyAssemblyResolve` event to manually load any referenced assemblies (calling `AppDomain`'s `ApplyPolicy` method, if desired); the CLR doesn't do it automatically for you. When the callback method is invoked, it must call `Assembly`'s `ReflectionOnlyLoadFrom` or `ReflectionOnlyLoad` method to explicitly load a referenced assembly and return a reference to this assembly.

NOTE

People often ask about assembly unloading. Unfortunately, the CLR doesn't support the ability to unload individual assemblies. If the CLR allowed it, your application would crash if a thread returned back from a method to code in the unloaded assembly. The CLR is all about robustness, security, and allowing an application to crash in this way would be counterproductive to its goals. If you want to unload an assembly, you must unload the entire AppDomain that contains it. This was discussed in great detail in Chapter 22.

It would seem that assemblies loaded with either the `ReflectionOnlyLoadFrom` or the `ReflectionOnlyLoad` method could be unloaded. After all, code in these assemblies is not allowed to execute. However, the CLR also doesn't allow assemblies loaded via either of these two methods to be unloaded. The reason is that after an assembly is loaded this way, you can still use reflection to create objects that refer to the metadata defined inside these assemblies. Unloading the assembly would require the objects to be invalidated somehow. Keeping track of this would be too expensive in terms of implementation and execution speed.

Many applications consist of an EXE file that depends on many DLL files. When deploying this application, all the files must be deployed. However, there is a technique that you can use to deploy just a single EXE file. First, identify all the DLL files that your EXE file depends on that do not ship as part of the Microsoft .NET Framework itself. Then add these DLLs to your Visual Studio project. For each DLL file you add, display its properties and change its Build Action to Embedded Resource. This causes the C# compiler to embed the DLL file(s) into your EXE file, and you can deploy this one EXE file.

At run time, the CLR won't be able to find the dependent DLL assemblies, which is a problem. To fix this, when your application initializes, register a callback method with the `AppDomain`'s `ResolveAssembly` event. The callback method's code should look something like the following.

```
private static Assembly ResolveEventHandler(Object sender, ResolveEventArgs args) {
   String dllName = new AssemblyName(args.Name).Name + ".dll";

   var assem = Assembly.GetExecutingAssembly();
   String resourceName = assem.GetManifestResourceNames().FirstOrDefault(rn =>
rn.EndsWith(dllName));
   if (resourceName == null) return null; // Not found, maybe another handler will find it
   using (var stream = assem.GetManifestResourceStream(resourceName)) {
      Byte[] assemblyData = new Byte[stream.Length];
       stream.Read(assemblyData, 0, assemblyData.Length);
       return Assembly.Load(assemblyData);
```

```
        }
    }
```

Now, the first time a thread calls a method that references a type in a dependent DLL file, the `AssemblyResolve` event will be raised and the preceding callback code will find the embedded DLL resource desired and load it by calling an overload of `Assembly`'s `Load` method that takes a `Byte[]` as an argument. Although I love the technique of embedding dependent DLLs inside another assembly, you should be aware that this does increase the memory used by your application at run time.

Using Reflection to Build a Dynamically Extensible Application

As you know, metadata is stored in a bunch of tables. When you build an assembly or a module, the compiler that you're using creates a type definition table, a field definition table, a method definition table, and so on. The `System.Reflection` namespace contains several types that allow you to write code that reflects over (or parses) these metadata tables. In effect, the types in this namespace offer an object model over the metadata contained in an assembly or a module.

Using these object model types, you can easily enumerate all of the types in a type definition metadata table. Then for each type, you can obtain its base type, the interfaces it implements, and the flags that are associated with the type. Additional types in the `System.Reflection` namespace allow you to query the type's fields, methods, properties, and events by parsing the corresponding metadata tables. You can also discover any custom attributes (covered in Chapter 18) that have been applied to any of the metadata entities. There are even classes that let you determine referenced assemblies and methods that return the IL byte stream for a method. With all of this information, you could easily build a tool very similar to Microsoft's ILDasm.exe.

> **NOTE**
>
> You should be aware that some of the reflection types and some of the members defined by these types are designed specifically for use by developers who are producing compilers for the CLR. Application developers don't typically use these types and members. The Framework Class Library (FCL) documentation doesn't explicitly point out which of these types and members are for compiler developers rather than application developers, but if you realize that not all reflection types and their members are for everyone, the documentation can be less confusing.

In reality, very few applications will have the need to use the reflection types. Reflection is typically used by class libraries that need to understand a type's definition in order to provide some rich functionality. For example, the FCL's serialization mechanism (discussed in Chapter 24) uses reflection to determine what fields a type defines. The serialization formatter can then obtain the values of these fields and write them into a byte stream that is used for sending across the Internet, saving to a file, or copying to the clipboard. Similarly, Visual Studio's designers use reflection to determine which properties should be shown to developers when laying out controls on their Web Forms or Windows Forms at design time.

Reflection is also used when an application needs to load a specific type from a specific assembly at run time to accomplish some task. For example, an application might ask the user to provide the name of an assembly and a type. The application could then explicitly load the assembly, construct an instance of the type, and call methods defined in the type. This usage is conceptually similar to calling Win32's `LoadLibrary` and `GetProcAddress` functions. Binding to types and calling methods in this way is frequently referred to as *late binding*. (*Early binding* is when the types and methods used by an application are determined at compile time.)

Reflection Performance

Reflection is an extremely powerful mechanism because it allows you to discover and use types and members at run time that you did not know about at compile time. This power does come with two main drawbacks:

- Reflection prevents type safety at compile time. Because reflection uses strings heavily, you lose type safety at compile time. For example, if you call `Type.GetType("int")`; to ask reflection to find a type called "`int`", the code compiles but returns `null` at run time because the CLR knows the "`int`" type as "`System.Int32`".
- Reflection is slow. When using reflection, the names of types and their members are not known at compile time; you discover them at run time by using a string name to identify each type and member. This means that reflection is constantly performing string searches as the types in the `System.Reflection` namespace scan through an assembly's metadata. Often, the string searches are case-insensitive comparisons, which can slow this down even more.

Invoking a member by using reflection will also hurt performance. When using reflection to invoke a method, you must first package the arguments into an array; internally, reflection must unpack these on to the thread's stack. Also, the CLR must check that the arguments are of the correct data type before invoking a method. Finally, the CLR ensures that the caller has the proper security permission to access the member being invoked.

For all of these reasons, it's best to avoid using reflection to access a field or invoke a method/property. If you're writing an application that will dynamically discover and construct type instances, you should take one of the following approaches:

- Have the types derive from a base type that is known at compile time. At run time, construct an instance of the derived type, place the reference in a variable that is of the base type (by way of a cast), and call virtual methods defined by the base type.
- Have the type implement an interface that is known at compile time. At run time, construct an instance of the type, place the reference in a variable that is of the interface type (by way of a cast), and call the methods defined by the interface.

I tend to prefer using the interface technique over the base type technique because the base type technique doesn't allow the developer to choose the base type that works best in a particular situation. Although the base type technique works better in versioning scenarios, because you could always add a member to the base type and the derived types just inherit it; you can't add a member to an interface without forcing all types that implement the interface to modify their code and recompile it.

When you use either of these two techniques, I strongly suggest that the interface or base type be defined in its own assembly. This will reduce versioning issues.

For more information about how to do this, see the section titled Designing an Application That Supports Add-Ins later in this chapter.

Discovering Types Defined in an Assembly

Reflection is frequently used to determine what types an assembly defines. The FCL offers many APIs to get this information. By far, the most commonly used API is `Assembly`'s `ExportedTypes`. property. Here is an example of code that loads an assembly and shows the names of all of the publicly exported types defined in it.

```
using System;
using System.Reflection;

public static class Program {
   public static void Main() {
      String dataAssembly = "System.Data, version=4.0.0.0, " +
         "culture=neutral, PublicKeyToken=b77a5c561934e089";
      LoadAssemAndShowPublicTypes(dataAssembly);
   }

   private static void LoadAssemAndShowPublicTypes(String assemId) {
      // Explicitly load an assembly in to this AppDomain
      Assembly a = Assembly.Load(assemId);

      // Execute this loop once for each Type
      // publicly-exported from the loaded assembly
      foreach (Type t in a.ExportedTypes) {
         // Display the full name of the type
         Console.WriteLine(t.FullName);
      }
   }
}
```

What Exactly Is a Type Object?

Notice that the previous code iterates over a sequence of `System.Type` objects. The `System.Type` type is your starting point for doing type and object manipulations. A `System.Type` object represents a type reference (as opposed to a type definition).

Recall that `System.Object` defines a public, nonvirtual instance method named `GetType`. When you call this method, the CLR determines the specified object's type and returns a reference to its `Type` object. Because there is only one `Type` object per type in an AppDomain, you can use equality and inequality operators to see whether two objects are of the same type.

```
private static Boolean AreObjectsTheSameType(Object o1, Object o2) {
   return o1.GetType() == o2.GetType();
}
```

In addition to calling `Object`'s `GetType` method, the FCL offers several more ways to obtain a `Type` object:

- The `System.Type` type offers several overloaded versions of the static `GetType` method. All versions of this method take a `String`. The string must specify the full name of the type (including its namespace). Note that the primitive type names supported by the compiler (such as C#'s `int, string, bool`, and so on) aren't allowed because these names mean nothing to the CLR. If the string is simply the name of a type, the method checks the calling assembly to see whether it defines a type of the specified name. If it does, a reference to the appropriate `Type` object is returned.

  If the calling assembly doesn't define the specified type, the types defined by MSCorLib.dll are checked. If a type with a matching name still can't be found, `null` is returned or a `System.TypeLoadException` is thrown, depending on which overload of the `GetType` method you called and what parameters you passed to it. The FCL documentation fully explains this method.

  You can pass an assembly-qualified type string, such as "System.Int32, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089", to `GetType`. In this case, `GetType` will look for the type in the specified assembly (loading the assembly if necessary).

- The `System.Type` type offers a static `ReflectionOnlyGetType` method. This method behaves similarly to the `GetType` method mentioned in the

previous bullet, except that the type is loaded so that it can be reflected over but cannot be executed.

- The `System.TypeInfo` type offers the following instance members: `DeclaredNestedTypes` and `GetDeclaredNestedType`.
- The `System.Reflection.Assembly` type offers the following instance members: `GetType, DefinedTypes`, and `ExportedTypes`.

NOTE

Microsoft has defined a Backus-Naur Form grammar for type names and assembly-qualified type names that is used for constructing strings that will be passed to reflection methods. Knowledge of the grammar can come in quite handy when you are using reflection, specifically if you are working with nested types, generic types, generic methods, reference parameters, or arrays. For the complete grammar, see the FCL documentation or do a web search for "Backus-Naur Form Grammar for Type Names." You can also look at `Type`'s and `TypeInfo`'s `MakeArrayType, MakeByRefType, MakeGenericType`, and `MakePointerType` methods.

Many programming languages also offer an operator that allows you to obtain a `Type` object from a type name that is known at compile time. When possible, you should use this operator to obtain a reference to a `Type` instead of using any of the methods in the preceding list, because the operator generally produces faster code. In C#, the operator is called `typeof`, and you use this operator typically to compare late-bound type information with early-bound (known at compile time) type information. The following code demonstrates an example of its use.

```
private static void SomeMethod(Object o) {
   // GetType returns the type of the object at runtime (late-bound)
   // typeof returns the type of the specified class (early-bound)
   if (o.GetType() == typeof(FileInfo))      { ... }
   if (o.GetType() == typeof(DirectoryInfo)) { ... }
}
```

NOTE

The first `if` statement in the code checks whether the variable `o` refers to an object of the `FileInfo` type; it does not check whether `o` refers to an object that is derived from the `FileInfo` type. In other words, the preceding code tests for an exact match, not a compatible match, which is what you would get if you use a cast or C#'s `is` or `as` operators.

As mentioned earlier, a `Type` object represents a type reference that is a lightweight object. If you want to learn more about the type itself, then you must acquire a `TypeInfo` object, which represents a type definition. You can convert a `Type` object to a `TypeInfo` object by calling `System.Reflection.IntrospectionExtensions`' `GetTypeInfo` extension method.

```
Type typeReference = ...;  // For example: o.GetType() or typeof(Object)
TypeInfo typeDefinition = typeReference.GetTypeInfo();
```

And, although less useful, you can convert a `TypeInfo` object to a `Type` object by calling `TypeInfo`'s `AsType method`.

```
TypeInfo typeDefinition = ...;
Type typeReference = typeDefinition.AsType();
```

Obtaining a `TypeInfo` object forces the CLR to resolve the type by ensuring that the assembly that defines the type is loaded. This can be an expensive operation that can be avoided if all you need are type references (`Type` objects). However, after you have a `TypeInfo` object, you can query many of the type's properties to learn more about it. Most of the properties, such as `IsPublic, IsSealed, IsAbstract, IsClass, IsValueType`, and so on, indicate flags associated with the type. Other properties, such as `Assembly, AssemblyQualifiedName, FullName, Module`, and so on, return the name of the type's defining assembly or module and the full name of the type. You can also query the `BaseType` property to obtain a reference to the type's base type, and a slew of members will give you even more information about the type. The FCL documentation describes all of the methods and properties that `TypeInfo` exposes.

Building a Hierarchy of Exception-Derived Types

The following code uses many of the concepts discussed already in this chapter to load a bunch of assemblies into the AppDomain and display all of the classes that are ultimately derived from `System.Exception`. By the way, this is the program I wrote to build the exception hierarchy displayed in the FCL-Defined Exception Classes section in Chapter 20.

```
public static void Go() {
   // Explicitly load the assemblies that we want to reflect over
```

```
   LoadAssemblies();

   // Filter & sort all the types
   var allTypes =
      (from a in AppDomain.CurrentDomain.GetAssemblies()
       from t in a.ExportedTypes
       where typeof(Exception).GetTypeInfo().IsAssignableFrom(t.GetTypeInfo())
       orderby t.Name
       select t).ToArray();

   // Build the inheritance hierarchy tree and show it
   Console.WriteLine(WalkInheritanceHierarchy(new StringBuilder(), 0, typeof(Exception),
allTypes));
}

private static StringBuilder WalkInheritanceHierarchy(
   StringBuilder sb, Int32 indent, Type baseType, IEnumerable<Type> allTypes) {
   String spaces = new String(' ', indent * 3);
   sb.AppendLine(spaces + baseType.FullName);
   foreach (var t in allTypes) {
      if (t.GetTypeInfo().BaseType != baseType) continue;
      WalkInheritanceHierarchy(sb, indent + 1, t, allTypes);
   }
   return sb;
}


private static void LoadAssemblies() {
   String[] assemblies = {
      "System,                    PublicKeyToken={0}",
      "System.Core,               PublicKeyToken={0}",
      "System.Data,               PublicKeyToken={0}",
      "System.Design,             PublicKeyToken={1}",
      "System.DirectoryServices,  PublicKeyToken={1}",
      "System.Drawing,            PublicKeyToken={1}",
      "System.Drawing.Design,     PublicKeyToken={1}",
      "System.Management,         PublicKeyToken={1}",
      "System.Messaging,          PublicKeyToken={1}",
      "System.Runtime.Remoting,   PublicKeyToken={0}",
      "System.Security,           PublicKeyToken={1}",
      "System.ServiceProcess,     PublicKeyToken={1}",
      "System.Web,                PublicKeyToken={1}",
      "System.Web.RegularExpressions, PublicKeyToken={1}",
      "System.Web.Services,       PublicKeyToken={1}",
      "System.Xml,                PublicKeyToken={0}",
   };

   String EcmaPublicKeyToken = "b77a5c561934e089";
   String MSPublicKeyToken = "b03f5f7f11d50a3a";
```

```
      // Get the version of the assembly containing System.Object
      // We'll assume the same version for all the other assemblies
      Version version = typeof(System.Object).Assembly.GetName().Version;

      // Explicitly load the assemblies that we want to reflect over
      foreach (String a in assemblies) {
         String AssemblyIdentity =
            String.Format(a, EcmaPublicKeyToken, MSPublicKeyToken) +
               ", Culture=neutral, Version=" + version;
         Assembly.Load(AssemblyIdentity);
      }
   }
```

Constructing an Instance of a Type

After you have a reference to a `Type`-derived object, you might want to construct an instance of this type. The FCL offers several mechanisms to accomplish this:

- **`System.Activator`'s `CreateInstance` methods** The `Activator` class offers several overloads of its static `CreateInstance` method. When you call this method, you can pass either a reference to a `Type` object or a `String` that identifies the type of object you want to create. The versions that take a type are simpler. You get to pass a set of arguments for the type's constructor, and the method returns a reference to the new object.

  The versions of this method in which you specify the desired type by using a string are a bit more complex. First, you must also specify a string identifying the assembly that defines the type. Second, these methods allow you to construct a remote object if you have remoting options configured properly. Third, these versions don't return a reference to the new object. Instead, they return a `System.Runtime.Remoting.ObjectHandle` (which is derived from `System.MarshalByRefObject`).

  An `ObjectHandle` is a type that allows an object created in one AppDomain to be passed around to other AppDomains without forcing the object to materialize. When you're ready to materialize the object, you call `ObjectHandle`'s `Unwrap` method. This method loads the assembly that defines the type being materialized in the AppDomain where `Unwrap` is called. If the object is being marshaled by reference, the proxy type and object are created. If the object is being marshaled by value, the copy is deserialized.

- **`System.Activator`'s `CreateInstanceFrom` methods** The `Activator` class also offers a set of static `CreateInstanceFrom` methods. These methods behave just as the `CreateInstance` method, except that you must always specify the type and its assembly via string parameters. The assembly is loaded into the calling AppDomain by using `Assembly`'s `LoadFrom` method (instead of `Load`). Because none of these methods takes a `Type` parameter, all of the `CreateInstanceFrom` methods return a reference to an `ObjectHandle`, which must be unwrapped.

- **`System.AppDomain`'s methods** The `AppDomain` type offers four instance methods (each with several overloads) that construct an instance of a type: `CreateInstance, CreateInstanceAndUnwrap, CreateInstanceFrom`, and `CreateInstanceFromAndUnwrap`. These methods work just as `Activator`'s methods except that these methods are instance methods, allowing you to specify which AppDomain the object should be constructed in. The methods that end with `Unwrap` exist for convenience so that you don't have to make an additional method call.

- **`System.Reflection.ConstructorInfo`'s `Invoke` instance method** Using a reference to a `TypeInfo` object, you can bind to a particular constructor and obtain a reference to the constructor's `ConstructorInfo` object. Then you can use the reference to the `ConstructorInfo` object to call its `Invoke` method. The type is always created in the calling AppDomain, and a reference to the new object is returned. I'll also discuss this method in more detail later in this chapter.

> **NOTE**
>
> The CLR doesn't require that value types define any constructors. However, this is a problem because all of the mechanisms in the preceding list construct an object by calling its constructor. However, `Activator`'s `CreateInstance` methods will allow you to create an instance of a value type without calling a constructor. If you want to create an instance of a value type without calling a constructor, you must call the version of the `CreateInstance` method that takes a single `Type` parameter or the version that takes `Type` and `Boolean` parameters.

The mechanisms just listed allow you to create an object for all types except for arrays (`System.Array`-derived types) and delegates (`System.MulticastDelegate`-derived types). To create an array, you should call `Array`'s static `CreateInstance` method (several overloaded versions exist). The first parameter to all versions of `CreateInstance` is a reference to the `Type` of elements you want in the array. `CreateInstance`'s other parameters allow you to specify various combinations of dimensions and bounds. To create a delegate, you should call `MethodInfo`'s `CreateDelegate` method. The first

parameter to all versions of `CreateDelegate` is a reference to the `Type` of delegate you want to create. `CreateDelegate`'s other parameter allows you to specify which object should be passed as the `this` parameter when calling an instance method.

To construct an instance of a generic type, first get a reference to the open type, and then call `Type`'s `MakeGenericType` method, passing in an array of types that you want to use as the type arguments. Then, take the returned `Type` object and pass it into one of the various methods previously listed. Here is an example.

```
using System;
using System.Reflection;


internal sealed class Dictionary<TKey, TValue> { }


public static class Program {
   public static void Main() {
      // Get a reference to the generic type's type object
      Type openType = typeof(Dictionary<,>);


      // Close the generic type by using TKey=String, TValue=Int32
      Type closedType = openType.MakeGenericType(typeof(String), typeof(Int32));


      // Construct an instance of the closed type
      Object o = Activator.CreateInstance(closedType);


      // Prove it worked
      Console.WriteLine(o.GetType());
   }
}
```

If you compile the preceding code and run it, you get the following output.

```
Dictionary`2[System.String,System.Int32]
```

Designing an Application That Supports Add-Ins

When you're building extensible applications, interfaces should be the centerpiece. You could use a base class instead of an interface, but in general, an interface is preferred because it allows add-in developers to choose their own base class. Suppose, for example, that you're writing an application and you want others to be able to create types that your application can load and use seamlessly.

Here's the way to design this application:

- Create a Host SDK assembly that defines an interface whose methods are used as the communication mechanism between the host application and the add-in components. When defining the parameters and return types for the interface methods, try to use other interfaces or types defined in MSCorLib.dll. If you want to pass and return your own data types, define them in this Host SDK assembly, too. After you settle on your interface definitions, give this assembly a strong name (discussed in Chapter 3), and then package and deploy it to your partners and users. Once published, you should really avoid making any kind of breaking changes to the types in this assembly. For example, do not change the interface in any way. However, if you define any data types, it is OK to add new members. If you make any modifications to the assembly, you'll probably want to deploy it with a publisher policy file (also discussed in Chapter 3).

  > **NOTE**
  >
  > You can use types defined in MSCorLib.dll because the CLR always loads the version of MSCorLib.dll that matches the version of the CLR itself. Also, only a single version of MSCorLib.dll is ever loaded into a CLR instance. In other words, different versions of MSCorLib.dll never load side by side (as described in Chapter 3). As a result, you won't have any type version mismatches, and your application will require less memory.

- The add-in developers will, of course, define their own types in their own Add-In assembly. Their Add-In assembly will reference the types in your Host SDK assembly. The add-in developers are able to put out a new version of their assembly as often as they'd like, and the host application will be able to consume the add-in types without any problem whatsoever.
- Create a separate Host Application assembly containing your application's types. This assembly will obviously reference the Host SDK assembly and use the types defined in it. Feel free to modify the code in the Host Application assembly to your heart's desire. Because the add-in developers don't reference the Host Application assembly, you can put out a new version of it every hour if you want to and not affect any of the add-in developers.

This section contains some very important information. When using types across assemblies, you need to be concerned with assembly-versioning issues. Take your time to architect this cleanly by isolating the types that you use for communication across assembly boundaries into their own assembly. Avoid mutating or changing these type definitions. However, if you really need to modify the type definitions, make sure that you change the assembly's version number and create a

publisher policy file for the new version.

I'll now walk through a very simple scenario that puts all of this together. First, here is the code for the HostSDK.dll assembly.

```
using System;


namespace Wintellect.HostSDK {
   public interface IAddIn {
      String DoSomething(Int32 x);
   }
}
```

Second, here is the code for an AddInTypes.dll assembly defining two public types that implement the HostSDK's interface. To build this assembly, the HostSDK.dll assembly must be referenced.

```
using System;
using Wintellect.HostSDK;


public sealed class AddIn_A : IAddIn {
   public AddIn_A() {
   }
   public String DoSomething(Int32 x) {
      return "AddIn_A: " + x.ToString();
   }
}


public sealed class AddIn_B : IAddIn {
   public AddIn_B() {
   }
   public String DoSomething(Int32 x) {
      return "AddIn_B: " + (x * 2).ToString();
   }
}
```

Third, here is the code for a simple Host.exe assembly (a console application). To build this assembly, the HostSDK.dll assembly must be referenced. To discover usable add-in types, this host code assumes that the types are defined in assemblies ending with a .dll file extension and that these assemblies are deployed into the same directory as the host's EXE file. Microsoft's Managed Extensibility Framework (MEF) is built on top of the various mechanisms that I show here, and it also offers add-in registration and discovery mechanisms. I urge you to check MEF out if you are building a dynamically extensible application, because it can simplify some of the material in this chapter.

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;
using Wintellect.HostSDK;


public static class Program {
   public static void Main() {
      // Find the directory that contains the Host exe
      String AddInDir = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);


      // Assume AddIn assemblies are in same directory as host's EXE file
      var AddInAssemblies = Directory.EnumerateFiles(AddInDir, "*.dll");


      // Create a collection of Add-In Types usable by the host
      var AddInTypes =
```

```
            from file in AddInAssemblies
            let assembly = Assembly.Load(file)
            from t in assembly.ExportedTypes // Publicly-exported types
            // Type is usable if it is a class that implements IAddIn
            where t.IsClass && typeof(IAddIn).GetTypeInfo().IsAssignableFrom(t.GetTyp
eInfo())
            select t;
         // Initialization complete: the host has discovered the usable Add-Ins


         // Here's how the host can construct Add-In objects and use them
         foreach (Type t in AddInTypes) {
            IAddIn ai = (IAddIn) Activator.CreateInstance(t);
            Console.WriteLine(ai.DoSomething(5));
         }
      }
   }
```

The simple host/add-in scenario just shown doesn't use AppDomains. However, in a real-life scenario, you will likely create each add-in in its own AppDomain with its own security and configuration settings. And of course, each AppDomain could be unloaded if you wanted to remove an add-in from memory. To communicate across the AppDomain boundary, you'd either tell the add-in developers to derive their add-in types from `MarshalByRefObject` or, more likely, have the host application define its own internal type that is derived from `MarshalByRefObject`. As each AppDomain is created, the host would create an instance of its own `MarshalByRefObject`-derived type in the new AppDomain. The host's code (in the default AppDomain) would communicate with its own type (in the other AppDomains) to have it load add-in assemblies and create and use instances of the add-in types.

Using Reflection to Discover a Type's Members

So far, this chapter has focused on the parts of reflection—assembly loading, type discovery, and object construction—necessary to build a dynamically extensible application. In order to have good performance and compile-time type safety, you want to avoid using reflection as much as possible. In the dynamically extensible application scenario, after an object is constructed, the host code typically casts the object to an interface type or a base class that is known at compile time; this allows the object's members to be accessed in a high-performance and compile-time type-safe way.

In the remainder of this chapter, I'm going to focus on some other aspects of reflection that you can use to discover and then invoke a type's members. The ability to discover and invoke a type's members is typically used to create developer tools and utilities that analyze an assembly by looking for certain programming patterns or uses of certain members. Examples of tools/utilities that do this are ILDasm.exe, FxCopCmd.exe, and Visual Studio's Windows Forms, Windows Presentation Foundation, and Web Forms designers. In addition, some class libraries use the ability to discover and invoke a type's members in order to offer rich functionality as a convenience to developers. Examples of class libraries that do so are serialization/deserialization and simple data binding.

Discovering a Type's Members

Fields, constructors, methods, properties, events, and nested types can all be defined as members within a type. The FCL contains a type called `System.Reflection.MemberInfo`. This class is an abstract base class that encapsulates a bunch of properties common to all type members. Derived from `MemberInfo` are a bunch of classes; each class encapsulates some more properties related to a specific type member. Figure 23-1 shows the hierarchy of these types.

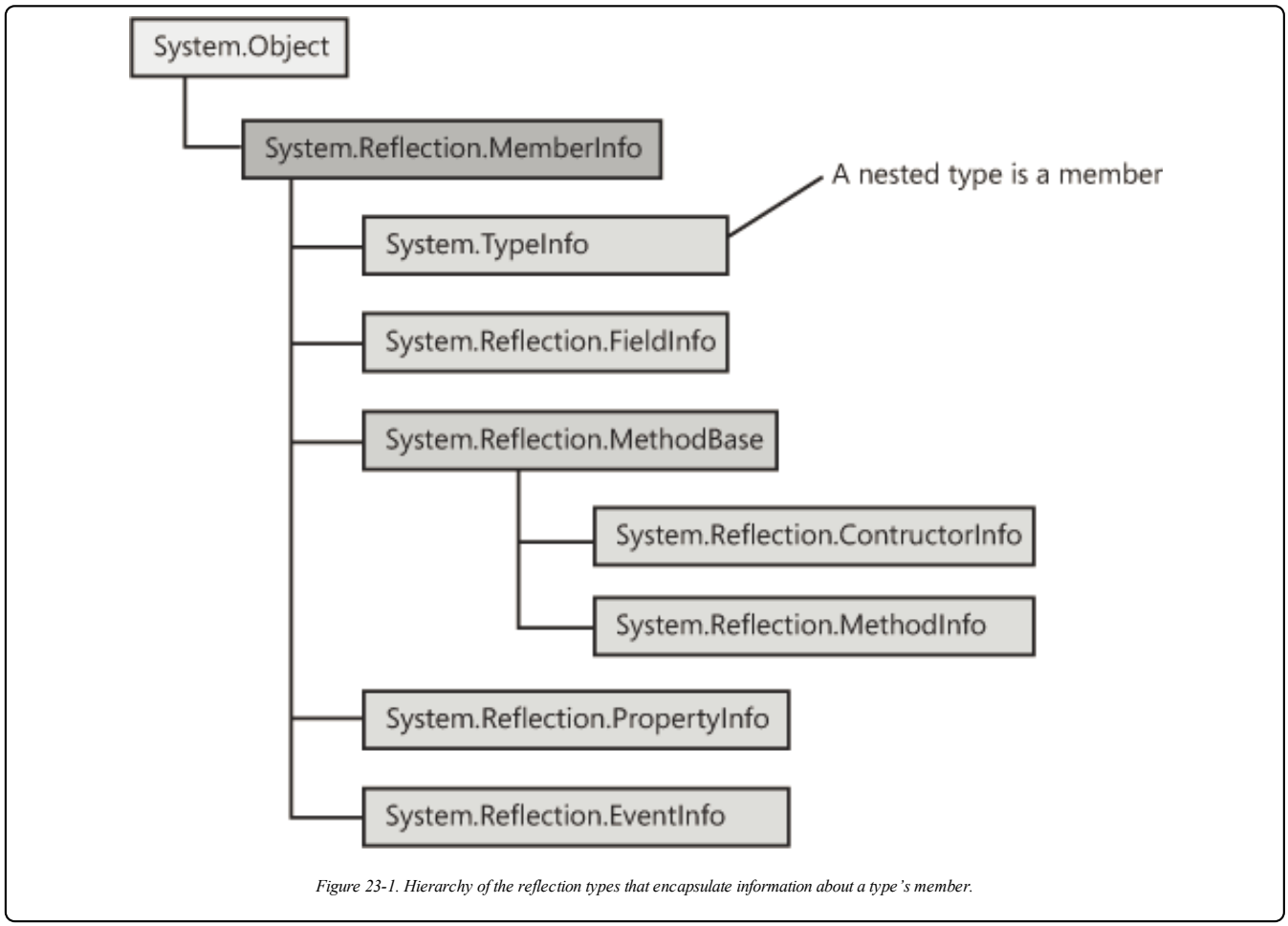

Figure 23-1. Hierarchy of the reflection types that encapsulate information about a type's member.

The following program demonstrates how to query a type's members and display some information about them. This code processes all of the public types defined in all assemblies loaded in the calling AppDomain. For each type, the `DeclaredMembers` property is called and returns a collection of `MemberInfo`-derived objects; each object refers to a single member defined within the type. Then, for each member, its kind (field, constructor, method, property, etc.) and its string value (obtained by calling `ToString`) is shown.

```
using System;
using System.Reflection;

public static class Program {
   public static void Main() {
      // Loop through all assemblies loaded in this AppDomain
      Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
      foreach (Assembly a in assemblies) {
         Show(0, "Assembly: {0}", a);

         // Find Types in the assembly
         foreach (Type t in a.ExportedTypes) {
            Show(1, "Type: {0}", t);

            // Discover the type's members
            foreach (MemberInfo mi in t.GetTypeInfo().DeclaredMembers) {
               String typeName = String.Empty;
               if (mi is Type) typeName = "(Nested) Type";
               if (mi is FieldInfo) typeName = "FieldInfo";
               if (mi is MethodInfo) typeName = "MethodInfo";
               if (mi is ConstructorInfo) typeName = "ConstructoInfo";
```

```
                if (mi is PropertyInfo) typeName = "PropertyInfo";
                if (mi is EventInfo) typeName = "EventInfo";
                Show(2, "{0}: {1}", typeName, mi);
             }
          }      }
      }
      private static void Show(Int32 indent, String format, params Object[] args) {
         Console.WriteLine(new String(' ', 3 * indent) + format, args);
      }
   }
```

When you compile and run this code, a ton of output is produced. Here is a small sampling of what it looks like.

```
Assembly: mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
   Type: System.Object
      MethodInfo: System.String ToString()
      MethodInfo: Boolean Equals(System.Object)
      MethodInfo: Boolean Equals(System.Object, System.Object)
      MethodInfo: Boolean ReferenceEquals(System.Object, System.Object)
      MethodInfo: Int32 GetHashCode()
      MethodInfo: System.Type GetType()
      MethodInfo: Void Finalize()
      MethodInfo: System.Object MemberwiseClone()
      MethodInfo: Void FieldSetter(System.String, System.String, System.Object)
      MethodInfo: Void FieldGetter(System.String, System.String, System.Object ByRef)
      MethodInfo: System.Reflection.FieldInfo GetFieldInfo(System.String, System.String)
      ConstructoInfo: Void .ctor()
   Type: System.Collections.Generic.IComparer`1[T]
      MethodInfo: Int32 Compare(T, T)
   Type: System.Collections.IEnumerator
      MethodInfo: Boolean MoveNext()
      MethodInfo: System.Object get_Current()
      MethodInfo: Void Reset()
      PropertyInfo: System.Object Current
   Type: System.IDisposable
      MethodInfo: Void Dispose()
   Type: System.Collections.Generic.IEnumerator`1[T]
      MethodInfo: T get_Current()
      PropertyInfo: T Current
   Type: System.ArraySegment`1[T]
      MethodInfo: T[] get_Array()
      MethodInfo: Int32 get_Offset()
      MethodInfo: Int32 get_Count()
      MethodInfo: Int32 GetHashCode()
      MethodInfo: Boolean Equals(System.Object)
      MethodInfo: Boolean Equals(System.ArraySegment`1[T])
      MethodInfo: Boolean op_Equality(System.ArraySegment`1[T], System.ArraySegment`1[T])
      MethodInfo: Boolean op_Inequality(System.ArraySegment`1[T], System.ArraySegment`1[T]
)
      ConstructoInfo: Void .ctor(T[])
      ConstructoInfo: Void .ctor(T[], Int32, Int32)
      PropertyInfo: T[] Array
```

```
PropertyInfo: Int32 Offset
PropertyInfo: Int32 Count
FieldInfo: T[] _array
FieldInfo: Int32 _offset
```

Because `MemberInfo` is the root of the member hierarchy, it makes sense for us to discuss it a bit more. Table 23-1 shows several read-only properties and methods offered by the `MemberInfo` class. These properties and methods are common to all members of a type. Don't forget that `System.TypeInfo` is derived from `MemberInfo`, and therefore, `TypeInfo` also offers all of the properties shown in Table 23-1.

Table 23-1. Properties and Methods Common to All MemberInfo-Derived Types

Member Name	Member Type	Description
`Name`	`String` property	Returns the name of the member.
`DeclaringType`	`Type` property	Returns the `Type` that declares the member.
`Module`	`Module` property	Returns the `Module` that declares the member.
`CustomAttributes`	Property returning `IEnumerable<CustomAttributeData>`	Returns a collection in which each element identifies an instance of a custom attribute applied to this member. Custom attributes can be applied to any member. Even though `Assembly` does not derive from `MemberInfo`, it provides the same property that can be used with assemblies.

Each element of the collection returned by querying `DeclaredMembers` is a reference to one of the concrete types in the hierarchy. Although `TypeInfo`'s `DeclaredMembers` property returns all of the type's members, `TypeInfo` also offers methods that return specific member types for a specified string name. For example, `TypeInfo` offers `GetDeclaredNestedType, GetDeclaredField, GetDeclaredMethod, GetDeclaredProperty`, and `GetDeclaredEvent`. These methods all return a reference to a `TypeInfo` object, `FieldInfo` object, `MethodInfo` object, `PropertyInfo` object, or `EventInfo` object, respectively. There is also a `GetDeclaredMethods` method that returns a collection of `MethodInfo` objects describing the methods matching the specified string name.

Figure 23-2 summarizes the types used by an application to walk reflection's object model. From an AppDomain, you can discover the assemblies loaded into it. From an assembly, you can discover the modules that make it up. From an assembly or a module, you can discover the types that it defines. From a type, you can discover its nested types, fields, constructors, methods, properties, and events. Namespaces are not part of this hierarchy because they are simply syntactical gatherings of types. If you want to list all of the namespaces defined in an assembly, you need to enumerate all of the types in this assembly and take a look at their `Namespace` property.

From a type, it is also possible to discover the interfaces it implements. And from a constructor, method, property accessor method, or event add/remove method, you can call the `GetParameters` method to obtain an array of `ParameterInfo` objects, which tells you the types of the member's parameters. You can also query the read-only `ReturnParameter` property to get a `ParameterInfo` object for detailed information about a member's return type. For a generic type or method, you can call the `GetGenericArguments` method to get the set of type parameters. Finally, for any of these items, you can query the `CustomAttributes` property to obtain the set of custom attributes applied to them.

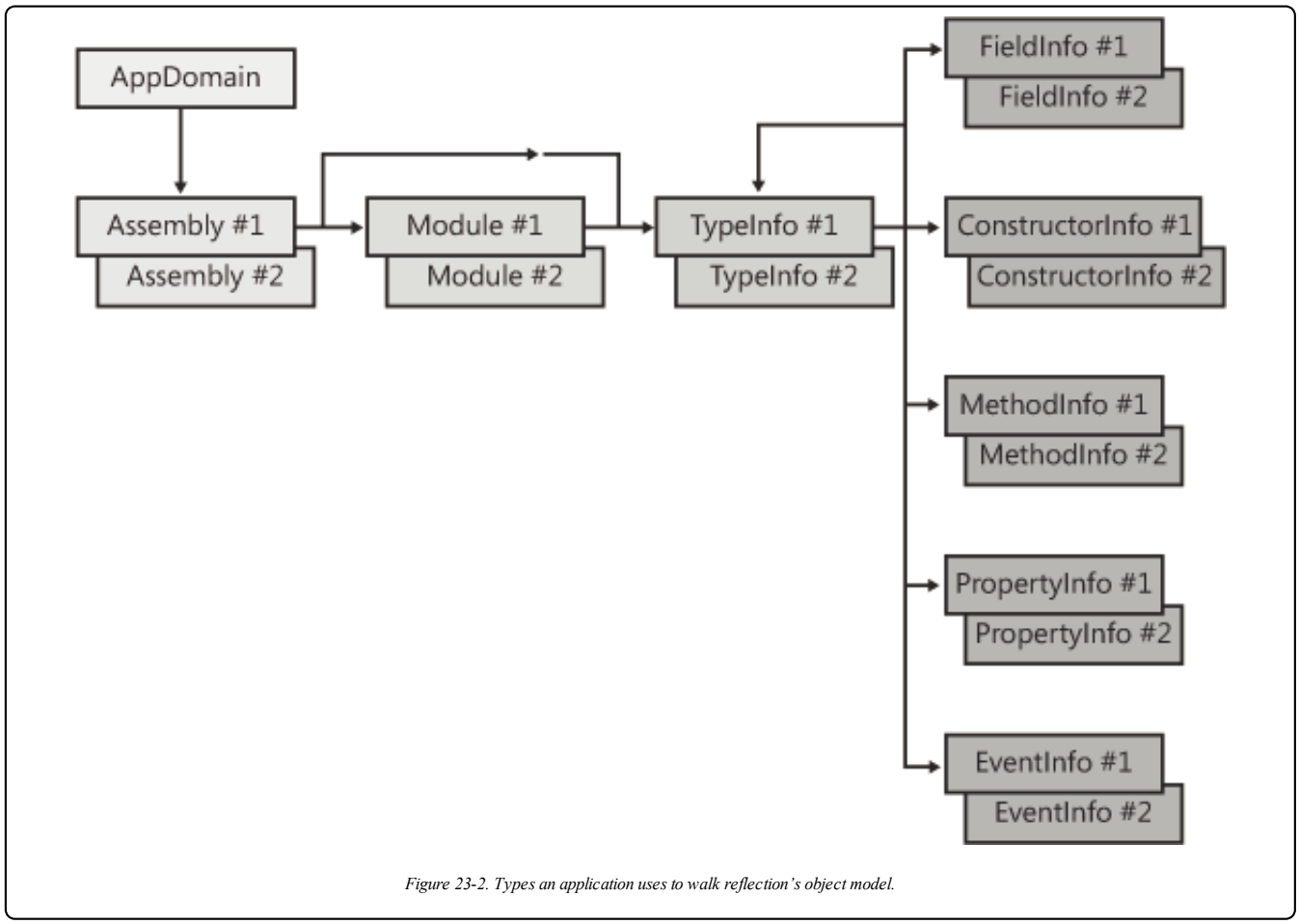

Figure 23-2. Types an application uses to walk reflection's object model.

Invoking a Type's Members

Now that you know how to discover the members defined by a type, you may want to invoke one of these members. What invoke means depends on the kind of member being invoked. Table 23-2 shows which method to call for each kind of member to invoke that member.

Table 23-2. How to Invoke a Member

Type of Member	Method to Invoke Member
`FieldInfo`	Call `GetValue` to get a field's value.
	Call `SetValue` to set a field's value.
`ConstructorInfo`	Call `Invoke` to construct an instance of the type and call a constructor.
`MethodInfo`	Call `Invoke` to call a method of the type.
`PropertyInfo`	Call `GetValue` to call a property's get accessor method.
	Call `SetValue` to call a property's set accessor method.
`EventInfo`	Call `AddEventHandler` to call an event's add accessor method.
	Call `RemoveEventHandler` to call an event's remove accessor method.

The `PropertyInfo` type represents metadata information about a property (as discussed in Chapter 10); that is, `PropertyInfo` offers `CanRead, CanWrite`, and `PropertyType` read-only properties. These properties indicate whether a property is readable or writeable and what data type the property is. `PropertyInfo` also has read-only `GetMethod` and `SetMethod` properties, which return `MethodInfo` objects representing the methods that get and set a property's value. `PropertyInfo`'s `GetValue` and `SetValue` methods exist for convenience; internally, they invoke the appropriate `MethodInfo` object. To support parameterful properties (C# indexers), the `GetValue` and `SetValue` methods offer an `index` parameter of `Object[]` type.

The `EventInfo` type represents metadata information about an event (as discussed in Chapter 11). The `EventInfo` type offers a read-only `EventHandlerType` property that returns the `Type` of the event's underlying delegate. The `EventInfo` type also has read-only `AddMethod` and `RemoveMethod` properties, which return the `MethodInfo` objects corresponding to the methods that add or remove a delegate to/from the event. To add or remove a delegate, you can invoke these `MethodInfo` objects, or you can call `EventInfo`'s more convenient `AddEventHandler` and `RemoveEventHandler` methods.

The following sample application demonstrates the various ways to use reflection to access a type's members. The `SomeType` class represents a type that has various members: a private field (`m_someField`), a public constructor (`SomeType`) that takes an `Int32` argument passed by reference, a public method (`ToString`), a public property (`SomeProp`), and a public event (`SomeEvent`). Having defined the `SomeType` type, I offer three different methods that use reflection to access `SomeType`'s members. Each method uses reflection in a different way to accomplish the same thing.

- The `BindToMemberThenInvokeTheMember` method demonstrates how to bind to a member and invoke it later.
- The `BindToMemberCreateDelegateToMemberThenInvokeTheMember` method demonstrates how to bind to an object or member, and then it creates a delegate that refers to that object or member. Calling through the delegate is very fast, and this technique yields faster performance if you intend to invoke the same member on the same object multiple times.
- The `UseDynamicToBindAndInvokeTheMember` method demonstrates how to use C# `dynamic` primitive type (discussed at the end of Chapter 5) to simplify the syntax for accessing members. In addition, this technique can give reasonably good performance if you intend to invoke the same member on different objects that are all of the same type because the binding will happen once per type and be cached so that it can be invoked multiple times quickly. You can also use this technique to invoke a member on objects of different types.

```
using System;
using System.Reflection;
using Microsoft.CSharp.RuntimeBinder;
using System.Linq;



// This class is used to demonstrate reflection
// It has a field, constructor, method, property, and an event
internal sealed class SomeType {
   private Int32 m_someField;
   public SomeType(ref Int32 x) { x *= 2; }
   public override String ToString() { return m_someField.ToString(); }
   public Int32 SomeProp {
      get { return m_someField; }
      set {
         if (value < 1)
            throw new ArgumentOutOfRangeException("value");
         m_someField = value;
      }
   public event EventHandler SomeEvent;
   private void NoCompilerWarnings() { SomeEvent.ToString();}
}


public static class Program {
   public static void Main() {
      Type t = typeof(SomeType);
      BindToMemberThenInvokeTheMember(t);
      Console.WriteLine();


      BindToMemberCreateDelegateToMemberThenInvokeTheMember(t);
      Console.WriteLine();


      UseDynamicToBindAndInvokeTheMember(t);
      Console.WriteLine();
```

```
   }


   private static void BindToMemberThenInvokeTheMember(Type t) {
      Console.WriteLine("BindToMemberThenInvokeTheMember");

      // Construct an instance
      Type ctorArgument = Type.GetType("System.Int32&"); // or typeof(Int32).MakeByRef
Type();
      ConstructorInfo ctor = t.GetTypeInfo().DeclaredConstructors.First(
         c => c.GetParameters()[0].ParameterType == ctorArgument);
      Object[] args = new Object[] { 12 };  // Constructor arguments
      Console.WriteLine("x before constructor called: " + args[0]);
      Object obj = ctor.Invoke(args);
      Console.WriteLine("Type: " + obj.GetType());
      Console.WriteLine("x after constructor returns: " + args[0]);

      // Read and write to a field
      FieldInfo fi = obj.GetType().GetTypeInfo().GetDeclaredField("m_someField");
      fi.SetValue(obj, 33);
      Console.WriteLine("someField: " + fi.GetValue(obj));

      // Call a method
      MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("ToString");
      String s = (String)mi.Invoke(obj, null);
      Console.WriteLine("ToString: " + s);

      // Read and write a property
      PropertyInfo pi = obj.GetType().GetTypeInfo().GetDeclaredProperty("SomeProp");
      try {
         pi.SetValue(obj, 0, null);
      }
      catch (TargetInvocationException e) {
         if (e.InnerException.GetType() != typeof(ArgumentOutOfRangeException)) throw;

         Console.WriteLine("Property set catch.");
      }
      pi.SetValue(obj, 2, null);
      Console.WriteLine("SomeProp: " + pi.GetValue(obj, null));

      // Add and remove a delegate from the event
      EventInfo ei = obj.GetType().GetTypeInfo().GetDeclaredEvent("SomeEvent");
      EventHandler eh = new EventHandler(EventCallback); // See ei.EventHandlerType
      ei.AddEventHandler(obj, eh);
      ei.RemoveEventHandler(obj, eh);
   }

   // Callback method added to the event
   private static void EventCallback(Object sender, EventArgs e) { }
```

```
   private static void BindToMemberCreateDelegateToMemberThenInvokeTheMember(Type t) {
      Console.WriteLine("BindToMemberCreateDelegateToMemberThenInvokeTheMember");

      // Construct an instance (You can't create a delegate to a constructor)
      Object[] args = new Object[] { 12 };  // Constructor arguments
      Console.WriteLine("x before constructor called: " + args[0]);
      Object obj = Activator.CreateInstance(t, args);
      Console.WriteLine("Type: " + obj.GetType().ToString());
      Console.WriteLine("x after constructor returns: " + args[0]);

      // NOTE: You can't create a delegate to a field

      // Call a method
      MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("ToString");
      var toString = mi.CreateDelegate<Func<String>>(obj);
      String s = toString();
      Console.WriteLine("ToString: " + s);

      // Read and write a property
      PropertyInfo pi = obj.GetType().GetTypeInfo().GetDeclaredProperty("SomeProp");
      var setSomeProp = pi.SetMethod.CreateDelegate<Action<Int32>>(obj);
      try {
         setSomeProp(0);
      }
      catch (ArgumentOutOfRangeException) {
         Console.WriteLine("Property set catch.");
      }
      setSomeProp(2);
      var getSomeProp = pi.GetMethod.CreateDelegate<Func<Int32>>(obj);
      Console.WriteLine("SomeProp: " + getSomeProp());

      // Add and remove a delegate from the event
      EventInfo ei = obj.GetType().GetTypeInfo().GetDeclaredEvent("SomeEvent");
      var addSomeEvent = ei.AddMethod.CreateDelegate<Action<EventHandler>>(obj
);
      addSomeEvent(EventCallback);
      var removeSomeEvent = ei.RemoveMethod.CreateDelegate<Action<EventHandler>
>(obj);
      removeSomeEvent(EventCallback);
   }

   private static void UseDynamicToBindAndInvokeTheMember(Type t) {
      Console.WriteLine("UseDynamicToBindAndInvokeTheMember");

      // Construct an instance (You can't use dynamic to call a constructor)
      Object[] args = new Object[] { 12 };  // Constructor arguments
      Console.WriteLine("x before constructor called: " + args[0]);
      dynamic obj = Activator.CreateInstance(t, args);
      Console.WriteLine("Type: " + obj.GetType().ToString());
```

```
      Console.WriteLine("x after constructor returns: " + args[0]);

      // Read and write to a field
      try {
         obj.m_someField = 5;
         Int32 v = (Int32)obj.m_someField;
         Console.WriteLine("someField: " + v);
      }
      catch (RuntimeBinderException e) {
         // We get here because the field is private
         Console.WriteLine("Failed to access field: " + e.Message);
      }

      // Call a method
      String s = (String)obj.ToString();
      Console.WriteLine("ToString: " + s);

      // Read and write a property
      try {
         obj.SomeProp = 0;
      }
      catch (ArgumentOutOfRangeException) {
         Console.WriteLine("Property set catch.");
      }
      obj.SomeProp = 2;
      Int32 val = (Int32)obj.SomeProp;
      Console.WriteLine("SomeProp: " + val);

      // Add and remove a delegate from the event
      obj.SomeEvent += new EventHandler(EventCallback);
      obj.SomeEvent -= new EventHandler(EventCallback);
   }
}



internal static class ReflectionExtensions {
   // Helper extension method to simplify syntax to create a delegate
   public static TDelegate CreateDelegate<TDelegate>(this MethodInfo mi, Object targ
et = null) {
      return (TDelegate)(Object)mi.CreateDelegate(typeof(TDelegate), target);
   }
}
```

If you build and run this code, you'll see the following output.

```
BindToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 33
ToString: 33
Property set catch.
```

```
SomeProp: 2

BindToMemberCreateDelegateToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
ToString: 0
Property set catch.
SomeProp: 2

UseDynamicToBindAndInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
Failed to access field: 'SomeType.m_someField' is inaccessible due to its protection level
ToString: 0
Property set catch.
SomeProp: 2
```

Notice that `SomeType`'s constructor takes an `Int32` by reference as its only parameter. The previous code shows how to call this constructor and how to examine the modified `Int32` value after the constructor returns. Near the top of the `BindToMemberThenInvokeTheMember` method, I show how to accomplish this by calling `Type`'s `GetType` method passing in a string of **"`System.Int32&`"**. The ampersand (**&**) in the string allows me to identify a parameter passed by reference. This ampersand is part of the Backus-Naur Form grammar for type names, which you can look up in the FCL documentation. The code also shows how to accomplish the same thing using `Type`'s `MakeByRefType` method.

Using Binding Handles to Reduce Your Process's Memory Consumption

Many applications bind to a bunch of types (`Type` objects) or type members (`MemberInfo`-derived objects) and save these objects in a collection of some sort. Then later, the application searches the collection for a particular object and then invokes this object. This is a fine way of doing things except for one small issue: `Type` and `MemberInfo`-derived objects require a lot of memory. So if an application holds on to too many of these objects and invokes them occasionally, the application's memory consumption increases dramatically, having an adverse effect on the application's performance.

Internally, the CLR has a more compact way of representing this information. The CLR creates these objects for our applications only to make things easier for developers. The CLR doesn't need these big objects itself in order to run. Developers who are saving/caching a lot of `Type` and `MemberInfo`-derived objects can reduce their working set by using run-time handles instead of objects. The FCL defines three runtime handle types (all defined in the System namespace): `RuntimeTypeHandle, RuntimeFieldHandle`, and `RuntimeMethodHandle`. All of these types are value types that contain just one field, an `IntPtr`; this makes instances of these types cheap (memory-wise). The `IntPtr` field is a handle that refers to a type, field, or method in an AppDomain's loader heap. So what you need now is an easy and efficient way to convert a heavyweight `Type/MemberInfo` object to a lightweight run-time handle instance and vice versa. Fortunately, this is easy using the following conversion methods and properties:

- To convert a `Type` object to a `RuntimeTypeHandle`, call `Type`'s static `GetTypeHandle` method passing in the reference to the `Type` object.
- To convert a `RuntimeTypeHandle` to a `Type` object, call `Type`'s static `GetTypeFromHandle` method passing in the `RuntimeTypeHandle`.
- To convert a `FieldInfo` object to a `RuntimeFieldHandle`, query `FieldInfo`'s instance read-only `FieldHandle` property.
- To convert a `RuntimeFieldHandle` to a `FieldInfo` object, call `FieldInfo`'s static `GetFieldFromHandle` method.
- To convert a `MethodInfo` object to a `RuntimeMethodHandle`, query `MethodInfo`'s instance read-only `MethodHandle` property.
- To convert a `RuntimeMethodHandle` to a `MethodInfo` object, call `MethodInfo`'s static `GetMethodFromHandle` method.

The following program sample acquires a lot of `MethodInfo` objects, converts them to `RuntimeMethodHandle` instances, and shows the working set difference.

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

```
public sealed class Program {
   private const BindingFlags c_bf = BindingFlags.FlattenHierarchy | BindingFlags.Instance
 |
      BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic;

   public static void Main() {
      // Show size of heap before doing any reflection stuff
      Show("Before doing anything");

      // Build cache of MethodInfo objects for all methods in MSCorlib.dll
      List<MethodBase> methodInfos = new List<MethodBase>();
      foreach (Type t in typeof(Object).Assembly.GetExportedTypes()) {
         // Skip over any generic types
         if (t.IsGenericTypeDefinition) continue;

         MethodBase[] mb = t.GetMethods(c_bf);
         methodInfos.AddRange(mb);
      }

      // Show number of methods and size of heap after binding to all methods
      Console.WriteLine("# of methods={0:N0}", methodInfos.Count);
      Show("After building cache of MethodInfo objects");

      // Build cache of RuntimeMethodHandles for all MethodInfo objects
      List<RuntimeMethodHandle> methodHandles =
         methodInfos.ConvertAll<RuntimeMethodHandle>(mb => mb.MethodHandle);

      Show("Holding MethodInfo and RuntimeMethodHandle cache");
      GC.KeepAlive(methodInfos); // Prevent cache from being GC'd early

      methodInfos = null;        // Allow cache to be GC'd now
      Show("After freeing MethodInfo objects");

      methodInfos = methodHandles.ConvertAll<MethodBase>(
         rmh=> MethodBase.GetMethodFromHandle(rmh));
      Show("Size of heap after re-creating MethodInfo objects");
      GC.KeepAlive(methodHandles);  // Prevent cache from being GC'd early
      GC.KeepAlive(methodInfos);    // Prevent cache from being GC'd early

      methodHandles = null;         // Allow cache to be GC'd now
      methodInfos = null;           // Allow cache to be GC'd now
      Show("After freeing MethodInfos and RuntimeMethodHandles");
   }
}
```

When I compiled and executed this program, I got the following output.

```
Heap size=     85,000 - Before doing anything
# of methods=48,467
Heap size=   7,065,632 - After building cache of MethodInfo objects
Heap size=   7,453,496 - Holding MethodInfo and RuntimeMethodHandle cache
Heap size=   6,732,704 - After freeing MethodInfo objects
```

```
Heap size=   7,372,704 - Size of heap after re-creating MethodInfo objects
Heap size=     192,232 - After freeing MethodInfos and RuntimeMethodHandles
```