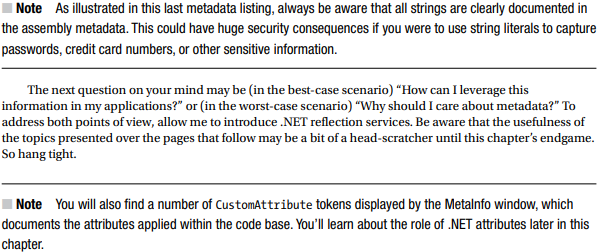
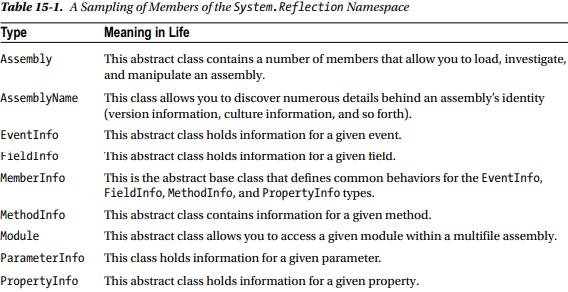
**Type Reflection, Late Binding, and Attribute-Based Programming**

As shown in Chapter 14, assemblies are the basic unit of deployment in the .NET universe. Using the integrated object browsers of Visual Studio (and numerous other IDEs), you are able to examine the types within a project’s referenced set of assemblies. Furthermore, external tools such as ildasm.exe allow you to peek into the underlying CIL code, type metadata, and assembly manifest for a given .NET binary. In addition to this design-time investigation of .NET assemblies, you are also able to programmatically obtain this same information using the System.Reflection namespace. To this end, the first task of this chapter is to define the role of reflection and the necessity of .NET metadata. The remainder of the chapter examines a number of closely related topics, all of which hinge upon reflection services. For example, you’ll learn how a .NET client may employ dynamic loading and late binding to activate types it has no compile-time knowledge of. You’ll also learn how to insert custom metadata into your .NET assemblies through the use of system-supplied and custom attributes. To put all of these (seemingly esoteric) topics into perspective, the chapter closes by demonstrating how to build several “snap-in objects” that you can plug into an extendable desktop GUI application.

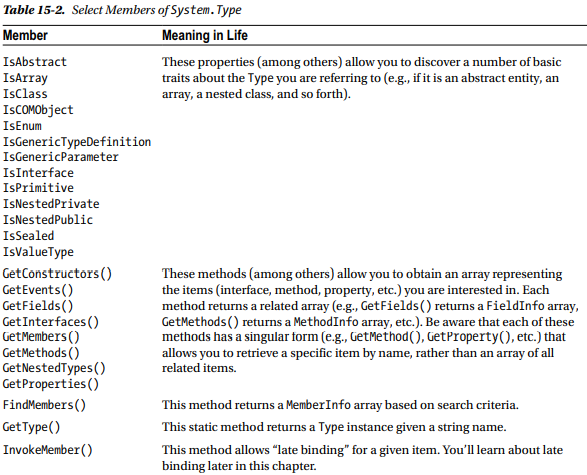
**The Necessity of Type Metadata** The ability to fully describe types (classes, interfaces, structures, enumerations, and delegates) using metadata is a key element of the .NET platform. Numerous .NET technologies, such as Windows Communication Foundation (WCF), and object serialization require the ability to discover the format of types at runtime. Furthermore, cross-language interoperability, numerous compiler services, and an IDE’s IntelliSense capabilities all rely on a concrete description of type.

**Viewing (Partial) Metadata for the EngineState Enumeration** Each type defined within the current assembly is documented using a TypeDef #n token (where TypeDef is short for type definition). If the type being described uses a type defined within a separate .NET assembly, the referenced type is documented using a TypeRef #n token (where TypeRef is short for type reference). A TypeRef token is a pointer (if you will) to the referenced type’s full metadata definition in an external assembly. In a nutshell, .NET metadata is a set of tables that clearly mark all type definitions (TypeDefs) and referenced types (TypeRefs), all of which can be viewed using ildasm.exe’s metadata window. As far as CarLibrary.dll goes, one TypeDef is the metadata description of the CarLibrary. EngineState enumeration (your number may differ; TypeDef numbering is based on the order in which the C# compiler processes the file).

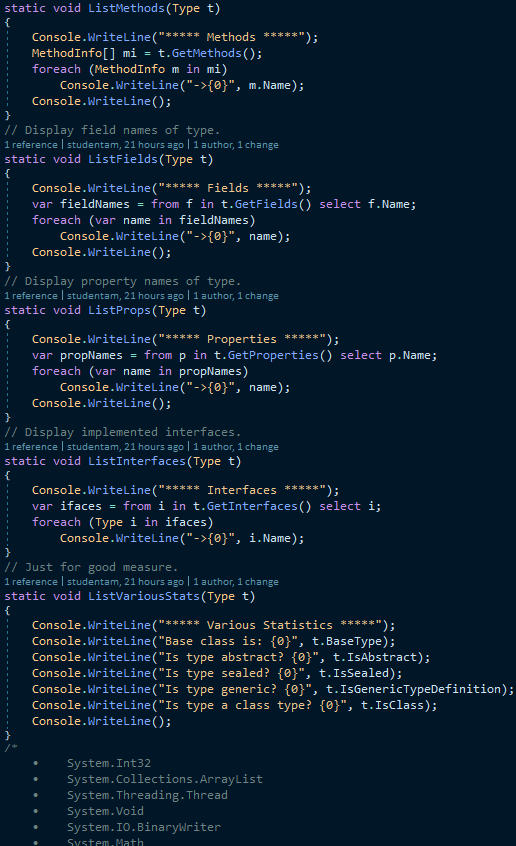
**Understanding Reflection** In the .NET universe, reflection is the process of runtime type discovery. Using reflection services, you are able to programmatically obtain the same metadata information displayed by ildasm.exe using a friendly object model. For example, through reflection, you can obtain a list of all types contained within a given \*.dll or \*.exe assembly, including the methods, fields, properties, and events defined by a given type. You can also dynamically discover the set of interfaces supported by a given type, the parameters

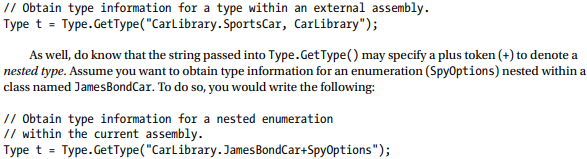
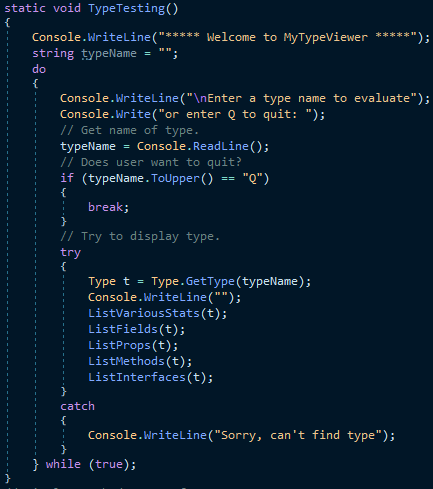
**The System.Type Class**

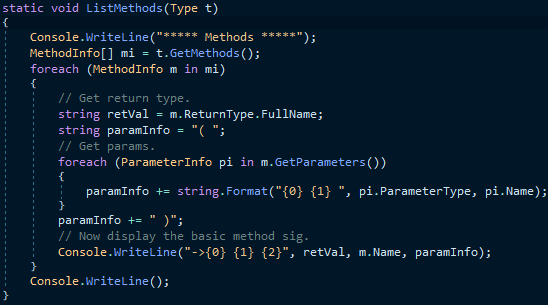
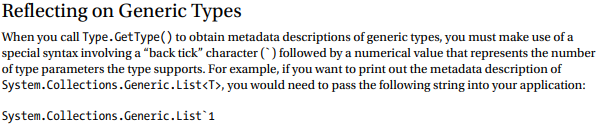
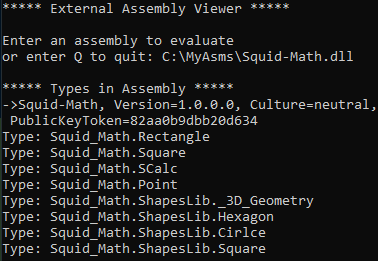
The System.Type class defines a number of members that can be used to examine a type’s metadata, a great number of which return types from the System.Reflection namespace. For example, Type.GetMethods() returns an array of MethodInfo objects, Type.GetFields() returns an array of FieldInfo objects, and so on. The complete set of members exposed by System.Type is quite expansive; however, Table 15-2 offers a partial snapshot of the members supported by System.Type (see the .NET Framework 4.6 SDK documentation for full details).

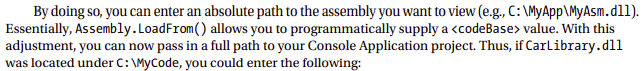
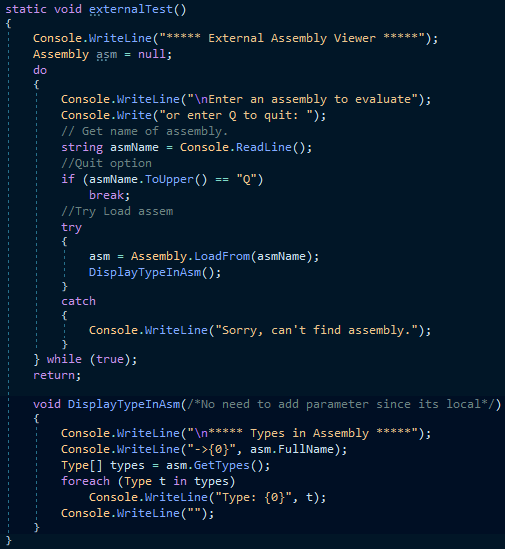
**Obtaining a Type Reference Using System.Object.GetType()** You can obtain an instance of the Type class in a variety of ways. However, the one thing you cannot do is directly create a Type object using the new keyword, as Type is an abstract class. Regarding your first choice, recall that System.Object defines a method named GetType(), which returns an instance of the Type class that represents the metadata for the current object.

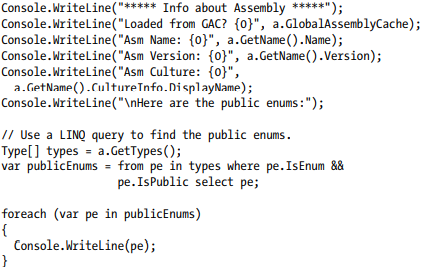
**Building a Custom Metadata Viewer** To illustrate the basic process of reflection (and the usefulness of System.Type), let’s create a Console Application project named MyTypeViewer.







**Dynamically Loading Assemblies** In Chapter 14, you learned all about how the CLR consults the assembly manifest when probing for an externally referenced assembly. However, there will be many times when you need to load assemblies on the fly programmatically, even if there is no record of said assembly in the manifest. Formally speaking, the act of loading external assemblies on demand is known as a dynamic load.

**Reflecting on Shared Assemblies** The Assembly.Load() method has been overloaded a number of times. One variation allows you to specify a culture value (for localized assemblies), as well as a version number and public key token value (for shared assemblies). Collectively speaking, the set of items identifying an assembly is termed the display name.

**Understanding Late Binding** Simply put, late binding is a technique in which you are able to create an instance of a given type and invoke its members at runtime without having hard-coded compile-time knowledge of its existence. When you are building an application that binds late to a type in an external assembly, you have no reason to set a reference to the assembly; therefore, the caller’s manifest has no direct listing of the assembly. At first glance, it is not easy to see the value of late binding. It is true that if you can “bind early” to an object (e.g., add an assembly reference and allocate the type using the C# new keyword), you should opt to do so. For one reason, early binding allows you to determine errors at compile time, rather than at runtime. Nevertheless, late binding does have a critical role in any extendable application you may be building. You will have a chance to build such an “extendable” program at the end of this chapter, in the section “Building an Extendable Application.” Until then, let’s examine the role of the Activator class.

**The System.Activator Class** The System.Activator class (defined in mscorlib.dll) is the key to the .NET late-binding process. For the current example, you are interested only in the Activator.CreateInstance() method, which is used to create an instance of a type à la late binding. This method has been overloaded numerous times to provide a good deal of flexibility. The simplest variation of the CreateInstance() member takes a valid Type object that describes the entity you want to allocate into memory on the fly

**Invoking Methods with No Parameters** Assume you want to invoke the TurboBoost() method of the MiniVan. As you recall, this method will set the state of the engine to “dead” and display an informational message box. The first step is to obtain a MethodInfo object for the TurboBoost() method using Type.GetMethod(). From the resulting MethodInfo, you are then able to call MiniVan.TurboBoost using Invoke(). MethodInfo.Invoke() requires you to send in all parameters that are to be given to the method represented by MethodInfo.