CHAPTER 6

Type and Member Basics

In this chapter:

In Chapters 4 and 5, I focused on types and what operations are guaranteed to exist on all instances of any type. I also explained how all types fall into one of two categories: reference types and value types. In this and the subsequent chapters in this part, I'll show how to design types by using the different kinds of members that can be defined within a type. In Chapters 7 through 11, I'll discuss the various members in detail.

The Different Kinds of Type Members

A type can define zero or more of the following kinds of members:

- **Constants** A constant is a symbol that identifies a never-changing data value. These symbols are typically used to make code more readable and maintainable. Constants are always associated with a type, not an instance of a type. Logically, constants are always static members. Discussed in Chapter 7, "Constants and Fields."
- **Fields** A field represents a read-only or read/write data value. A field can be static, in which case the field is considered part of the type's state. A field can also be instance (nonstatic), in which case it's considered part of an object's state. I strongly encourage you to make fields private so that the state of the type or object can't be corrupted by code outside of the defining type. Discussed in Chapter 7.
- **Instance constructors** An instance constructor is a special method used to initialize a new object's instance fields to a good initial state. Discussed in Chapter 8, "Methods."
- **Type constructors** A type constructor is a special method used to initialize a type's static fields to a good initial state. Discussed in Chapter 8.

- **Methods** A method is a function that performs operations that change or query the state of a type (static method) or an object (instance method). Methods typically read and write to the fields of the type or object. Discussed in Chapter 8.

- **Operator overloads** An operator overload is a method that defines how an object should be manipulated when certain operators are applied to the object. Because not all programming languages support operator overloading, operator overload methods are not part of the Common Language Specification (CLS). Discussed in Chapter 8.

- **Conversion operators** A conversion operator is a method that defines how to implicitly or explicitly cast or convert an object from one type to another type. As with operator overload methods, not all programming languages support conversion operators, so they're not part of the CLS. Discussed in Chapter 8.

- **Properties** A property is a mechanism that allows a simple, field-like syntax for setting or querying part of the logical state of a type (static property) or object (instance property) while ensuring that the state doesn't become corrupt. Properties can be parameterless (very common) or parameterful (fairly uncommon but used frequently with collection classes). Discussed in Chapter 10, "Properties."

- **Events** A static event is a mechanism that allows a type to send a notification to one or more static or instance methods. An instance (nonstatic) event is a mechanism that allows an object to send a notification to one or more static or instance methods. Events are usually raised in response to a state change occurring in the type or object offering the event. An event consists of two methods that allow static or instance methods to register and unregister interest in the event. In addition to the two methods, events typically use a delegate field to maintain the set of registered methods. Discussed in Chapter 11, "Events."

- **Types** A type can define other types nested within it. This approach is typically used to break a large, complex type down into smaller building blocks to simplify the implementation.

Again, the purpose of this chapter isn't to describe these various members in detail but to set the stage and explain what these various members all have in common.

Regardless of the programming language you're using, the corresponding compiler must process your source code and produce metadata and Intermediate Language (IL) code for each kind of member in the preceding list. The format of the metadata is identical regardless of the source programming language you use, and this feature is what makes the CLR a *common language* run time. The metadata is the common information that all languages produce and consume, enabling code in one programming language to seamlessly access code written in a completely different programming language.

This common metadata format is also used by the CLR, which determines how constants, fields, constructors, methods, properties, and events all behave at run time. Simply stated, metadata is the key to the whole Microsoft .NET Framework development platform; it enables the seamless integration of languages, types, and objects.

The following C# code shows a type definition that contains an example of all the possible members. The code shown here will compile (with warnings), but it isn't representative of a type that you'd normally create; most of the methods do nothing of any real value. Right now, I just want to show you how the compiler translates this type and its members into metadata. Once again, I'll discuss the individual members in the next few chapters.

```
using System;

public sealed class SomeType {                              //  1

   // Nested class
   private class SomeNestedType { }                         //  2

   // Constant, read-only, and static read/write field
   private const    Int32  c_SomeConstant = 1;              //  3
   private readonly String m_SomeReadOnlyField = "2";       //  4
   private static   Int32  s_SomeReadWriteField = 3;        //  5

   // Type constructor
   static SomeType() { }                                    //  6

   // Instance constructors
   public SomeType(Int32 x) { }                             //  7
   public SomeType() { }                                    //  8

   // Instance and static methods
   private String InstanceMethod() { return null; }         // 9
   public static void Main() {}                             // 10

   // Instance property
   public Int32 SomeProp {                                  // 11
      get { return 0; }                                     // 12
      set { }                                               // 13
   }

   // Instance parameterful property (indexer)
   public Int32 this[String s] {                            // 14
      get { return 0; }                                     // 15
      set { }                                               // 16
   }

   // Instance event
   public event EventHandler SomeEvent;                     // 17
}
```

If you were to compile the type just defined and examine the metadata in ILDasm.exe, you'd see the output shown in Figure 6-1.

Notice that all the members defined in the source code cause the compiler to emit some metadata. In fact, some of the members cause the compiler to generate additional members as well as additional metadata. For example, the event member (17) causes the compiler to emit a field, two methods, and some additional metadata. I don't expect you to fully understand what you're seeing here now. But as you read the next few chapters, I encourage you to look back to this example to see how the member is defined and what effect it has on the metadata produced by the compiler.

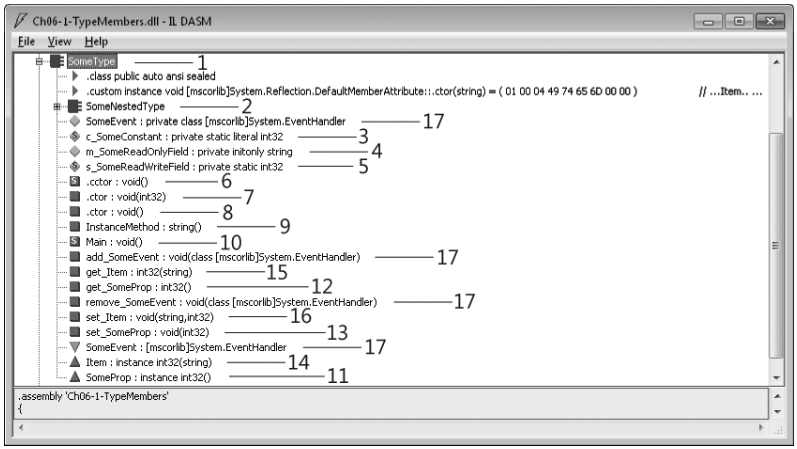

FIGURE 6-1 ILDasm.exe output showing metadata from preceding code.

Type Visibility

When defining a type at file scope (versus defining a type nested within another type), you can specify the type's visibility as being either `public` or `internal`. A `public` type is visible to all code within the defining assembly as well as all code written in other assemblies. An `internal` type is visible to all code within the defining assembly, and the type is not visible to code written in other assemblies. If you do not explicitly specify either of these when you define a type, the C# compiler sets the type's visibility to `internal` (the more restrictive of the two). Here are some examples.

```
using System;

// The type below has public visibility and can be accessed by code
// in this assembly as well as code written in other assemblies.
public class ThisIsAPublicType { ... }

// The type below has internal visibility and can be accessed by code
// in this assembly only.
internal class ThisIsAnInternalType { ... }

// The type below is internal because public/internal
// is not explicitly stated.
class ThisIsAlsoAnInternalType { ... }
```

Friend Assemblies

Imagine the following scenario: a company has one team, TeamA, that is defining a bunch of utility types in one assembly, and they expect these types to be used by members in another team, TeamB. For various reasons such as time schedules or geographical location, or perhaps different cost centers or reporting structures, these two teams cannot build all of their types into a single assembly; instead, each team produces its own assembly file.

In order for TeamB's assembly to use TeamA's types, TeamA must define all of their utility types as `public`. However, this means that their types are publicly visible to any and all assemblies; developers in another company could write code that uses the public utility types, and this is not desirable. Maybe the utility types make certain assumptions that TeamB ensures when they write code that uses TeamA's types. What we'd like to have is a way for TeamA to define their types as `internal` while still allowing TeamB to access the types. The CLR and C# support this via *friend assemblies*. This friend assembly feature is also useful when you want to have one assembly containing code that performs unit tests against the internal types within another assembly.

When an assembly is built, it can indicate other assemblies it considers "friends" by using the `InternalsVisibleTo` attribute defined in the `System.Runtime.CompilerServices` namespace. The attribute has a string parameter that identifies the friend assembly's name and public key (the string you pass to the attribute must not include a version, culture, or processor architecture). Note that friend assemblies can access all of an assembly's `internal` types as well as these type's `internal` members. Here is an example of how an assembly can specify two other strongly named assemblies named "Wintellect" and "Microsoft" as its friend assemblies.

```
using System;
using System.Runtime.CompilerServices; // For InternalsVisibleTo attribute

// This assembly's internal types can be accessed by any code written
// in the following two assemblies (regardless of version or culture):
[assembly:InternalsVisibleTo("Wintellect, PublicKey=12345678...90abcdef")]
[assembly:InternalsVisibleTo("Microsoft, PublicKey=b77a5c56...1934e089")]

internal sealed class SomeInternalType { ... }
internal sealed class AnotherInternalType { ... }
```

Accessing the above assembly's `internal` types from a friend assembly is trivial. For example, here's how a friend assembly called "Wintellect" with a public key of "12345678...90abcdef" can access the internal type `SomeInternalType` in the preceding assembly.

```
using System;

internal sealed class Foo {
   private static Object SomeMethod() {
      // This "Wintellect" assembly accesses the other assembly's
      // internal type as if it were a public type
      SomeInternalType sit = new SomeInternalType();
      return sit;
   }
}
```

Because the `internal` members of the types in an assembly become accessible to friend assemblies, you should think carefully about what accessibility you specify for your type's members and which assemblies you declare as your friends. Note that the C# compiler requires you to use the `/out:<file>` compiler switch when compiling the friend assembly (the assembly that does not contain the `InternalsVisibleTo` attribute). The switch is required because the compiler needs to know the name of the assembly being compiled in order to determine if the resulting assembly should be

considered a friend assembly. You would think that the C# compiler could determine this on its own because it normally determines the output file name on its own; however, the compiler doesn't decide on an output file name until it is finished compiling the code. So requiring the `/out:<file>` compiler switch improves the performance of compiling significantly.

Also, if you are compiling a module (as opposed to an assembly) using C#'s `/t:module` switch, and this module is going to become part of a friend assembly, you need to compile the module by using the C# compiler's `/moduleassemblyname:<string>` switch as well. This tells the compiler what assembly the module will be a part of so the compiler can allow code in the module to access the other assembly's internal types.

Member Accessibility

When defining a type's member (which includes nested types), you can specify the member's accessibility. A member's accessibility indicates which members can be legally accessed from referent code. The CLR defines the set of possible accessibility modifiers, but each programming language chooses the syntax and term it wants developers to use when applying the accessibility to a member. For example, the CLR uses the term Assembly to indicate that a member is accessible to any code within the same assembly, whereas the C# term for this is `internal`.

Table 6-1 shows the six accessibility modifiers that can be applied to a member. The rows of the table are in order from most restrictive (Private) to least restrictive (Public).

TABLE 6-1 Member Accessibility

CLR Term	C# Term	Description
Private	`private`	The member is accessible only by methods in the defining type or any nested type.
Family	`protected`	The member is accessible only by methods in the defining type, any nested type, or one of its derived types without regard to assembly.
Family and Assembly	(not supported)	The member is accessible only by methods in the defining type, any nested type, or by any derived types defined in the same assembly.
Assembly	`internal`	The member is accessible only by methods in the defining assembly.
Family or Assembly	`protected internal`	The member is accessible by any nested type, any derived type (regardless of assembly), or any methods in the defining assembly.
Public	`public`	The member is accessible to all methods in any assembly.

Of course, for any member to be accessible, it must be defined in a type that is visible. For example, if AssemblyA defines an `internal` type with a `public` method, code in AssemblyB cannot call the `public` method because the `internal` type is not visible to AssemblyB.

When compiling code, the language compiler is responsible for checking that the code is referencing types and members correctly. If the code references some type or member incorrectly, the compiler has the responsibility of emitting the appropriate error message. In addition, the just-in-time (JIT) compiler also ensures that references to fields and methods are legal when compiling IL code into native CPU instructions at run time. For example, if the JIT compiler detects code that is improperly attempting to access a private field or method, the JIT compiler throws a `FieldAccessException` or a `MethodAccessException`, respectively.

Verifying the IL code ensures that a referenced member's accessibility is properly honored at run time, even if a language compiler ignored checking the accessibility. Another, more likely, possibility is that the language compiler compiled code that accessed a `public` member in another type (in another assembly); but at run time, a different version of the assembly is loaded, and in this new version, the `public` member has changed and is now `protected` or `private`.

In C#, if you do not explicitly declare a member's accessibility, the compiler usually (but not always) defaults to selecting `private` (the most restrictive of them all). The CLR requires that all members of an interface type be public. The C# compiler knows this and forbids the programmer from explicitly specifying accessibility on interface members; the compiler just makes all the members `public` for you.

More Info See the "Declared Accessibility" section in the C# Language Specification for the complete set of C# rules about what accessibilities can be applied to types and members and what default accessibilities C# selects based on the context in which the declaration takes place.

Furthermore, you'll notice the CLR offers an accessibility called *Family and Assembly*. However, C# doesn't expose this in the language. The C# team felt that this accessibility was for the most part useless and decided not to incorporate it into the C# language.

When a derived type is overriding a member defined in its base type, the C# compiler requires that the original member and the overriding member have the same accessibility. That is, if the member in the base class is `protected`, the overriding member in the derived class must also be `protected`. However, this is a C# restriction, not a CLR restriction. When deriving from a base class, the CLR allows a member's accessibility to become less restrictive but not more restrictive. For example, a class can override a `protected` method defined in its base class and make the overridden method `public` (more accessible). However, a class cannot override a `protected` method defined in its base class and make the overridden method `private` (less accessible). The reason a class cannot make a base class method more restricted is because a user of the derived class could always cast to the base type and gain access to the base class's method. If the CLR allowed the derived type's method to be less accessible, it would be making a claim that was not enforceable.

Static Classes

There are certain classes that are never intended to be instantiated, such as `Console`, `Math`, `Environment`, and `ThreadPool`. These classes have only `static` members and, in fact, the classes exist simply as a way to group a set of related members together. For example, the `Math` class defines a bunch of methods that do math-related operations. C# allows you to define non-instantiable classes by using the C# `static` keyword. This keyword can be applied only to classes, not structures (value types) because the CLR always allows value types to be instantiated and there is no way to stop or prevent this.

The compiler enforces many restrictions on a `static` class:

- The class must be derived directly from `System.Object` because deriving from any other base class makes no sense because inheritance applies only to objects, and you cannot create an instance of a `static` class.
- The class must not implement any interfaces because interface methods are callable only when using an instance of a class.
- The class must define only `static` members (fields, methods, properties, and events). Any instance members cause the compiler to generate an error.
- The class cannot be used as a field, method parameter, or local variable because all of these would indicate a variable that refers to an instance, and this is not allowed. If the compiler detects any of these uses, the compiler issues an error.

Here is an example of a `static` class that defines some `static` members; this code compiles (with a warning) but the class doesn't do anything interesting.

```
using System;

public static class AStaticClass {
   public static void AStaticMethod() { }

   public static String AStaticProperty {
      get { return s_AStaticField; }
      set { s_AStaticField = value; }
   }

   private static String s_AStaticField;

   public static event EventHandler AStaticEvent;
}
```

If you compile the code above into a library (DLL) assembly and look at the result by using ILDasm.exe, you'll see what is shown in Figure 6-2. As you can see in Figure 6-2, defining a class by using the `static` keyword causes the C# compiler to make the class both `abstract` and `sealed`. Furthermore, the compiler will not emit an instance constructor method into the type. Notice that there is no instance constructor (`.ctor`) method shown in Figure 6-2.

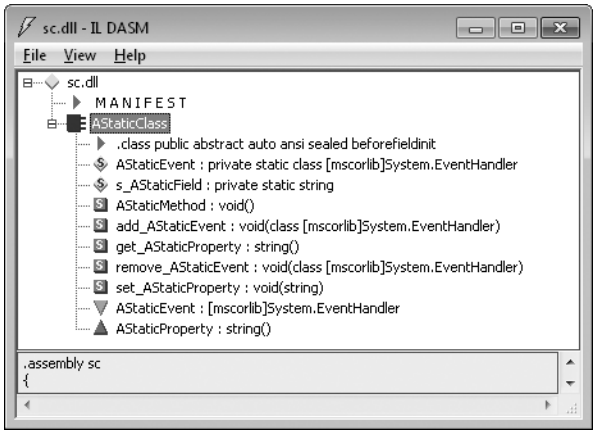

FIGURE 6-2 ILDasm.exe showing the class as abstract sealed in metadata.

Partial Classes, Structures, and Interfaces

In this section, I discuss partial classes, structures, and interfaces. The `partial` keyword tells the C# compiler that the source code for a single class, structure, or interface definition may span one or more source code files. It should be noted that the compiler combines all of a type's partials together at compile time; the CLR always works on complete type definitions. There are three main reasons why you might want to split the source code for a type across multiple files:

- **Source control** Suppose a type's definition consists of a lot of source code, and a programmer checks it out of source control to make changes. No other programmer will be able to modify the type at the same time without doing a merge later. Using the `partial` keyword allows you to split the code for the type across multiple source code files, each of which can be checked out individually so that multiple programmers can edit the type at the same time.

- **Splitting a class, structure, or interface into distinct logical units within a single file** I sometimes create a single type that provides multiple features so that the type can provide a complete solution. To simplify my implementation, I will sometimes declare the same partial type repeatedly within a single source code file. Then, in each part of the partial type, I implement one feature with all its fields, methods, properties, events, and so on. This allows me to easily see all the members that provide a single feature grouped together, which simplifies my coding. Also, I can easily comment out a part of the partial type to remove a whole feature from the type and replace it with another implementation (via a new part of the partial type).

- **Code spitters** In Microsoft Visual Studio, when you create a new project, some source code files are created automatically as part of the project. These source code files contain templates that give you a head start at building these kinds of projects. When you use the Visual Studio designers and drag and drop controls onto the design surface, Visual Studio writes source code for you automatically and spits this code into the source code files. This really improves your productivity. Historically, the generated code was emitted into the same source code file

that you were working on. The problem with this is that you might edit the generated code accidentally and cause the designers to stop functioning correctly. Starting with Visual Studio 2005, when you create a new form, user control, and so on, Visual Studio creates two source code files: one for your code and the other for the code generated by the designer. Because the designer code is in a separate file, you'll be far less likely to accidentally edit it.

The `partial` keyword is applied to the types in all files. When the files are compiled together, the compiler combines the code to produce one type that is in the resulting .exe or .dll assembly file (or .netmodule module file). As I stated in the beginning of this section, the partial types feature is completely implemented by the C# compiler; the CLR knows nothing about partial types at all. This is why all of the source code files for the type must use the same programming language, and they must all be compiled together as a single compilation unit.

Components, Polymorphism, and Versioning

Object-oriented programming (OOP) has been around for many, many years. When it was first used in the late 1970s/early 1980s, applications were much smaller in size and all the code to make the application run was written by one company. Sure, there were operating systems back then and applications did make use of what they could out of those operating systems, but the operating systems offered very few features compared with the operating systems of today.

Today, software is much more complex and users demand that applications offer rich features such as GUIs, menu items, mouse input, tablet input, printer output, networking, and so on. For this reason, our operating systems and development platforms have grown substantially over recent years. Furthermore, it is no longer feasible or even cost effective for application developers to write all of the code necessary for their application to work the way users expect. Today, applications consist of code produced by many different companies. This code is stitched together using an object-oriented paradigm.

Component Software Programming (CSP) is OOP brought to this level. Here are some attributes of a component:

- A component (an assembly in the .NET Framework) has the feeling of being "published."
- A component has an identity (a name, version, culture, and public key).
- A component forever maintains its identity (the code in an assembly is never statically linked into another assembly; .NET always uses dynamic linking).
- A component clearly indicates the components it depends upon (reference metadata tables).
- A component should document its classes and members. C# offers this by allowing in-source Extensible Markup Language (XML) documentation along with the compiler's `/doc` command-line switch.
- A component must specify the security permissions it requires. The CLR's code access security (CAS) facilities enable this.

- A component publishes an interface (object model) that won't change for any servicings. A *servicing* is a new version of a component whose intention is to be backward compatible with the original version of the component. Typically, a servicing version includes bug fixes, security patches, and possibly some small feature enhancements. But a servicing cannot require any new dependencies or any additional security permissions.

As indicated by the last bullet, a big part of CSP has to do with versioning. Components will change over time and components will ship on different time schedules. Versioning introduces a whole new level of complexity for CSP that didn't exist with OOP, with which all code was written, tested, and shipped as a single unit by a single company. In this section, I'm going to focus on component versioning.

In the .NET Framework, a version number consists of four parts: a *major* part, a *minor* part, a *build* part, and a *revision* part. For example, an assembly whose version number is 1.2.3.4 has a major part of 1, a minor part of 2, a build part of 3, and a revision part of 4. The major/minor parts are typically used to represent a consistent and stable feature set for an assembly and the build/revision parts are typically used to represent a servicing of this assembly's feature set.

Let's say that a company ships an assembly with version 2.7.0.0. If the company later wants to fix a bug in this component, they would produce a new assembly in which only the build/revision parts of the version are changed, something like version 2.7.1.34. This indicates that the assembly is a servicing whose intention is to be backward compatible with the original component (version 2.7.0.0).

On the other hand, if the company wants to make a new version of the assembly that has significant changes to it and is therefore not intended to be backward compatible with the original assembly, the company is really creating a new component and the new assembly should be given a version number in which the major/minor parts are different from the existing component (version 3.0.0.0, for example).

Note I have just described how you should think of version numbers. Unfortunately, the CLR doesn't treat version numbers this way. Today, the CLR treats a version number as an opaque value, and if an assembly depends on version 1.2.3.4 of another assembly, the CLR tries to load version 1.2.3.4 only (unless a binding redirection is in place).

Now that we've looked at how we use version numbers to update a component's identity to reflect a new version, let's take a look at some of the features offered by the CLR and programming languages (such as C#) that allow developers to write code that is resilient to changes that may be occurring in components that they are using.

Versioning issues come into play when a type defined in a component (assembly) is used as the base class for a type in another component (assembly). Obviously, if the base class versions (changes) underneath the derived class, the behavior of the derived class changes as well, probably in a way that causes the class to behave improperly. This is particularly true in polymorphism scenarios in which a derived type overrides virtual methods defined by a base type.

C# offers five keywords that you can apply to types and/or type members that impact component versioning. These keywords map directly to features supported in the CLR to support component versioning. Table 6-2 contains the C# keywords related to component versioning and indicates how each keyword affects a type or type member definition.

TABLE 6-2 C# Keywords and How They Affect Component Versioning

C# Keyword	Type	Method/Property/Event	Constant/Field
`abstract`	Indicates that no instances of the type can be constructed	Indicates that the derived type must override and implement this member before instances of the derived type can be constructed	(not allowed)
`virtual`	(not allowed)	Indicates that this member can be overridden by a derived type	(not allowed)
`override`	(not allowed)	Indicates that the derived type is overriding the base type's member	(not allowed)
`sealed`	Indicates that the type cannot be used as a base type	Indicates that the member cannot be overridden by a derived type. This keyword can be applied only to a method that is overriding a virtual method	(not allowed)
`new`	When applied to a nested type, method, property, event, constant, or field, indicates that the member has no relationship to a similar member that may exist in the base class		

I will demonstrate the value and use of all these keywords in the upcoming section titled "Dealing with Virtual Methods When Versioning Types." But before we get to a versioning scenario, let's focus on how the CLR actually calls virtual methods.

How the CLR Calls Virtual Methods, Properties, and Events

In this section, I will be focusing on methods, but this discussion is relevant to virtual properties and virtual events as well. Properties and events are actually implemented as methods; this will be shown in their corresponding chapters.

Methods represent code that performs some operation on the type (static methods) or an instance of the type (nonstatic methods). All methods have a name, a signature, and a return type (that may be `void`). The CLR allows a type to define multiple methods with the same name as long as each method has a different set of parameters or a different return type. So it's possible to define two methods with the same name and same parameters as long as the methods have a different return type. However, except for IL assembly language, I'm not aware of any language that takes advantage of this "feature"; most languages (including C#) require that methods differ by parameters and ignore a method's return type when determining uniqueness. (C# actually relaxes this restriction when defining conversion operator methods; see Chapter 8 for details.)

The Employee class shown below defines three different kinds of methods.

```
internal class Employee {
   // A nonvirtual instance method
   public         Int32    GetYearsEmployed() { ... }

   // A virtual method (virtual implies instance)
   public virtual String   GetProgressReport() { ... }

   // A static method
   public static  Employee Lookup(String name) { ... }
}
```

When the compiler compiles this code, the compiler emits three entries in the resulting assembly's method definition table. Each entry has flags set indicating if the method is instance, virtual, or static.

When code is written to call any of these methods, the compiler emitting the calling code examines the method definition's flags to determine how to emit the proper IL code so that the call is made correctly. The CLR offers two IL instructions for calling a method:

- The `call` IL instruction can be used to call static, instance, and virtual methods. When the `call` instruction is used to call a static method, you must specify the type that defines the method that the CLR should call. When the `call` instruction is used to call an instance or virtual method, you must specify a variable that refers to an object. The `call` instruction assumes that this variable is not `null`. In other words, the type of the variable itself indicates which type defines the method that the CLR should call. If the variable's type doesn't define the method, base types are checked for a matching method. The `call` instruction is frequently used to call a virtual method nonvirtually.

- The `callvirt` IL instruction can be used to call instance and virtual methods, not static methods. When the `callvirt` instruction is used to call an instance or virtual method, you must specify a variable that refers to an object. When the `callvirt` IL instruction is used to call a nonvirtual instance method, the type of the variable indicates which type defines the method that the CLR should call. When the `callvirt` IL instruction is used to call a virtual instance method, the CLR discovers the actual type of the object being used to make the call and then calls the method polymorphically. In order to determine the type, the variable being used to make the call must not be `null`. In other words, when compiling this call, the JIT compiler generates code that verifies that the variable's value is not `null`. If it is `null`, the `callvirt` instruction causes the CLR to throw a `NullReferenceException`. This additional check means that the `callvirt` IL instruction executes slightly more slowly than the `call` instruction. Note that this `null` check is performed even when the `callvirt` instruction is used to call a nonvirtual instance method.

So now, let's put this together to see how C# uses these different IL instructions.

```
using System;

public sealed class Program {
   public static void Main() {
      Console.WriteLine(); // Call a static method

      Object o = new Object();
      o.GetHashCode(); // Call a virtual instance method
      o.GetType();     // Call a nonvirtual instance method
   }
}
```

If you were to compile the code above and look at the resulting IL, you'd see the following.

```
.method public hidebysig static void Main() cil managed {
 .entrypoint
 // Code size 26 (0x1a)
 .maxstack 1
 .locals init (object o)
 IL_0000: call void System.Console::WriteLine()
 IL_0005: newobj instance void System.Object::.ctor()
 IL_000a: stloc.0
 IL_000b: ldloc.0
 IL_000c: callvirt instance int32 System.Object::GetHashCode()
 IL_0011: pop
 IL_0012: ldloc.0
 IL_0013: callvirt instance class System.Type System.Object::GetType()
 IL_0018: pop
 IL_0019: ret
} // end of method Program::Main
```

Notice that the C# compiler uses the `call` IL instruction to call `Console`'s `WriteLine` method. This is expected because `WriteLine` is a static method. Next, notice that the `callvirt` IL instruction is used to call `GetHashCode`. This is also expected, because `GetHashCode` is a virtual method. Finally, notice that the C# compiler also uses the `callvirt` IL instruction to call the `GetType` method. This is surprising because `GetType` is not a virtual method. However, this works because while JIT-compiling this code, the CLR will know that `GetType` is not a virtual method, and so the JIT-compiled code will simply call `GetType` nonvirtually.

Of course, the question is, why didn't the C# compiler simply emit the `call` instruction instead? The answer is because the C# team decided that the JIT compiler should generate code to verify that the object being used to make the call is not `null`. This means that calls to nonvirtual instance methods are a little slower than they could be. It also means that the following C# code will cause a `NullReferenceException` to be thrown. In some other programming languages, the intention of the following code would run just fine.

```
using System;

public sealed class Program {
   public Int32 GetFive() { return 5; }
   public static void Main() {
```

```
      Program p = null;
      Int32 x = p.GetFive(); // In C#, NullReferenceException is thrown
   }
}
```

Theoretically, the preceding code is fine. Sure, the variable p is `null`, but when calling a nonvirtual method (`GetFive`), the CLR needs to know just the data type of p, which is `Program`. If `GetFive` did get called, the value of the `this` argument would be `null`. Because the argument is not used inside the `GetFive` method, no `NullReferenceException` would be thrown. However, because the C# compiler emits a `callvirt` instruction instead of a `call` instruction, the preceding code will end up throwing the `NullReferenceException`.

Important If you define a method as nonvirtual, you should never change the method to virtual in the future. The reason is because some compilers will call the nonvirtual method by using the call instruction instead of the `callvirt` instruction. If the method changes from nonvirtual to virtual and the referencing code is not recompiled, the virtual method will be called nonvirtually, causing the application to produce unpredictable behavior. If the referencing code is written in C#, this is not a problem, because C# calls all instance methods by using `callvirt`. But this could be a problem if the referencing code was written using a different programming language.

Sometimes, the compiler will use a `call` instruction to call a virtual method instead of using a `callvirt` instruction. At first, this may seem surprising, but the code below demonstrates why it is sometimes required.

```
internal class SomeClass {
   // ToString is a virtual method defined in the base class: Object.
   public override String ToString() {

      // Compiler uses the 'call' IL instruction to call
      // Object's ToString method nonvirtually.

      // If the compiler were to use 'callvirt' instead of 'call', this
      // method would call itself recursively until the stack overflowed.
      return base.ToString();
   }
}
```

When calling `base.ToString` (a virtual method), the C# compiler emits a `call` instruction to ensure that the `ToString` method in the base type is called nonvirtually. This is required because if `ToString` were called virtually, the call would execute recursively until the thread's stack overflowed, which obviously is not desired.

Compilers tend to use the `call` instruction when calling methods defined by a value type because value types are sealed. This implies that there can be no polymorphism even for their virtual methods, which causes the performance of the call to be faster. In addition, the nature of a value type instance guarantees it can never be `null`, so a `NullReferenceException` will never be thrown. Finally, if you

were to call a value type's virtual method virtually, the CLR would need to have a reference to the value type's type object in order to refer to the method table within it. This requires boxing the value type. Boxing puts more pressure on the heap, forcing more frequent garbage collections and hurting performance.

Regardless of whether `call` or `callvirt` is used to call an instance or virtual method, these methods always receive a hidden `this` argument as the method's first parameter. The `this` argument refers to the object being operated on.

When designing a type, you should try to minimize the number of virtual methods you define. First, calling a virtual method is slower than calling a nonvirtual method. Second, virtual methods cannot be inlined by the JIT compiler, which further hurts performance. Third, virtual methods make versioning of components more brittle, as described in the next section. Fourth, when defining a base type, it is common to offer a set of convenience overloaded methods. If you want these methods to be polymorphic, the best thing to do is to make the most complex method virtual and leave all of the convenience overloaded methods nonvirtual. By the way, following this guideline will also improve the ability to version a component without adversely affecting the derived types. Here is an example.

```
public class Set {
   private Int32 m_length = 0;

   // This convenience overload is not virtual
   public Int32 Find(Object value) {
      return Find(value, 0, m_length);
   }

   // This convenience overload is not virtual
   public Int32 Find(Object value, Int32 startIndex) {
      return Find(value, startIndex, m_length - startIndex);
   }

   // The most feature-rich method is virtual and can be overridden
   public virtual Int32 Find(Object value, Int32 startIndex, Int32 endIndex) {
      // Actual implementation that can be overridden goes here...
   }

   // Other methods go here
}
```

Using Type Visibility and Member Accessibility Intelligently

With the .NET Framework, applications are composed of types defined in multiple assemblies produced by various companies. This means that the developer has little control over the components he or she is using and the types defined within those components. The developer typically doesn't have access to the source code (and probably doesn't even know what programming language was used to create the component), and components tend to version with different schedules. Furthermore, due to polymorphism and protected members, a base class developer must trust the code written by the derived class developer. And, of course, the developer of a derived class must trust the code that he

is inheriting from a base class. These are just some of the issues that you need to really think about when designing components and types.

In this section, I'd like to say just a few words about how to design a type with these issues in mind. Specifically, I'm going to focus on the proper way to set type visibility and member accessibility so that you'll be most successful.

First, when defining a new type, compilers should make the class sealed by default so that the class cannot be used as a base class. Instead, many compilers, including C#, default to unsealed classes and allow the programmer to explicitly mark a class as sealed by using the `sealed` keyword. Obviously, it is too late now, but I think that today's compilers have chosen the wrong default and it would be nice if this could change with future compilers. There are three reasons why a sealed class is better than an unsealed class:

- **Versioning** When a class is originally sealed, it can change to unsealed in the future without breaking compatibility. However, after a class is unsealed, you can never change it to sealed in the future as this would break all derived classes. In addition, if the unsealed class defines any unsealed virtual methods, ordering of the virtual method calls must be maintained with new versions or there is the potential of breaking derived types in the future.
- **Performance** As discussed in the previous section, calling a virtual method doesn't perform as well as calling a nonvirtual method because the CLR must look up the type of the object at run time in order to determine which type defines the method to call. However, if the JIT compiler sees a call to a virtual method using a sealed type, the JIT compiler can produce more efficient code by calling the method nonvirtually. It can do this because it knows there can't possibly be a derived class if the class is sealed. For example, in the code below, the JIT compiler can call the virtual `ToString` method nonvirtually.

```
using System;
public sealed class Point {
   private Int32 m_x, m_y;

   public Point(Int32 x, Int32 y) { m_x = x; m_y = y; }

   public override String ToString() {
      return String.Format("({0}, {1})", m_x, m_y);
   }

   public static void Main() {
      Point p = new Point(3, 4);

      // The C# compiler emits the callvirt instruction here but the
      // JIT compiler will optimize this call and produce code that
      // calls ToString nonvirtually because p's type is Point,
      // which is a sealed class
      Console.WriteLine(p.ToString());
   }
}
```

- **Security and predictability** A class must protect its own state and not allow itself to ever become corrupted. When a class is unsealed, a derived class can access and manipulate the base class's state if any data fields or methods that internally manipulate fields are accessible and not private. In addition, a virtual method can be overridden by a derived class, and the derived class can decide whether to call the base class's implementation. By making a method, property, or event virtual, the base class is giving up some control over its behavior and its state. Unless carefully thought out, this can cause the object to behave unpredictably, and it opens up potential security holes.

The problem with a sealed class is that it can be a big inconvenience to users of the type. Occasionally, developers want to create a class derived from an existing type in order to attach some additional fields or state information for their application's own use. In fact, they may even want to define some helper or convenience methods on the derived type to manipulate these additional fields. Although the CLR offers no mechanism to extend an already-built type with helper methods or fields, you can simulate helper methods by using C#'s extension methods (discussed in Chapter 8) and you can simulate tacking state onto an object by using the `ConditionalWeakTable` class (discussed in Chapter 21, "The Managed Heap and Garbage Collection."

Here are the guidelines I follow when I define my own classes:

- When defining a class, I always explicitly make it `sealed` unless I truly intend for the class to be a base class that allows specialization by derived classes. As stated earlier, this is the opposite of what C# and many other compilers default to today. I also default to making the class `internal` unless I want the class to be publicly exposed outside of my assembly. Fortunately, if you do not explicitly indicate a type's visibility, the C# compiler defaults to internal. If I really feel that it is important to define a class that others can derive but I do not want to allow specialization, I will simulate creating a closed class by using the above technique of sealing the virtual methods that my class inherits.

- Inside the class, I always define my data fields as `private` and I never waver on this. Fortunately, C# does default to making fields `private`. I'd actually prefer it if C# mandated that all fields be private and that you could not make fields `protected`, `internal`, `public`, and so on. Exposing state is the easiest way to get into problems, have your object behave unpredictably, and open potential security holes. This is true even if you just declare some fields as `internal`. Even within a single assembly, it is too hard to track all code that references a field, especially if several developers are writing code that gets compiled into the same assembly.

- Inside the class, I always define my methods, properties, and events as `private` and nonvirtual. Fortunately, C# defaults to this as well. Certainly, I'll make a method, property, or event `public` to expose some functionality from the type. I try to avoid making any of these members `protected` or `internal`, because this would be exposing my type to some potential vulnerability. However, I would sooner make a member `protected` or `internal` than I would make a member `virtual` because a virtual member gives up a lot of control and really relies on the proper behavior of the derived class.

- There is an old OOP adage that goes like this: when things get too complicated, make more types. When an implementation of some algorithm starts to get complicated, I define helper types that encapsulate discrete pieces of functionality. If I'm defining these helper types for use by a single über-type, I'll define the helper types nested within the über-type. This allows for scoping and also allows the code in the nested, helper type to reference the private members defined in the über-type. However, there is a design guideline rule, enforced by the Code Analysis tool (FxCopCmd.exe) in Visual Studio, which indicates that publicly exposed nested types should be defined at file or assembly scope and not be defined within another type. This rule exists because some developers find the syntax for referencing nested types cumbersome. I appreciate this rule, and I never define public nested types.

Dealing with Virtual Methods When Versioning Types

As was stated earlier, in a Component Software Programming environment, versioning is a very important issue. I talked about some of these versioning issues in Chapter 3, "Shared Assemblies and Strongly Named Assemblies," when I explained strongly named assemblies and discussed how an administrator can ensure that an application binds to the assemblies that it was built and tested with. However, other versioning issues cause source code compatibility problems. For example, you must be very careful when adding or modifying members of a type if that type is used as a base type. Let's look at some examples.

CompanyA has designed the following type, `Phone`.

```
namespace CompanyA {
   public class Phone {
      public void Dial() {
         Console.WriteLine("Phone.Dial");
         // Do work to dial the phone here.
      }
   }
}
```

Now imagine that CompanyB defines another type, `BetterPhone`, which uses CompanyA's `Phone` type as its base.

```
namespace CompanyB {
   public class BetterPhone : CompanyA.Phone {
      public void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      protected virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

When CompanyB attempts to compile its code, the C# compiler issues the following message.

```
warning CS0108: 'CompanyB.BetterPhone.Dial()' hides inherited member 'CompanyA.Phone.Dial()'.
```

Use the new keyword if hiding was intended." This warning is notifying the developer that BetterPhone is defining a Dial method, which will hide the Dial method defined in Phone. This new method could change the semantic meaning of Dial (as defined by CompanyA when it originally created the Dial method).

It's a very nice feature of the compiler to warn you of this potential semantic mismatch. The compiler also tells you how to remove the warning by adding the new keyword before the definition of Dial in the BetterPhone class. Here's the fixed BetterPhone class.

```
namespace CompanyB {
   public class BetterPhone : CompanyA.Phone {

      // This Dial method has nothing to do with Phone's Dial method.
      public new void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      protected virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

At this point, CompanyB can use BetterPhone.Dial in its application. Here's some sample code that CompanyB might write.

```
public sealed class Program {
   public static void Main() {
      CompanyB.BetterPhone phone = new CompanyB.BetterPhone();
      phone.Dial();
   }
}
```

When this code runs, the following output is displayed.

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
```

This output shows that CompanyB is getting the behavior it desires. The call to Dial is calling the new Dial method defined by BetterPhone, which calls the virtual EstablishConnection method and then calls the Phone base type's Dial method.

Now let's imagine that several companies have decided to use CompanyA's Phone type. Let's further imagine that these other companies have decided that the ability to establish a connection in the Dial method is a really useful feature. This feedback is given to CompanyA, which now revises its Phone class.

```
namespace CompanyA {
   public class Phone {
      public void Dial() {
         Console.WriteLine("Phone.Dial");
         EstablishConnection();
         // Do work to dial the phone here.
      }

      protected virtual void EstablishConnection() {
         Console.WriteLine("Phone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

Now when CompanyB compiles its `BetterPhone` type (derived from this new version of CompanyA's `Phone`), the compiler issues this message.

```
warning CS0114: 'CompanyB.BetterPhone.EstablishConnection()' hides inherited member 'CompanyA.
Phone.EstablishConnection()'. To make the current member override that implementation, add the
override keyword. Otherwise, add the new keyword.
```

The compiler is alerting you to the fact that both `Phone` and `BetterPhone` offer an `EstablishConnection` method and that the semantics of both might not be identical; simply recompiling `BetterPhone` can no longer give the same behavior as it did when using the first version of the `Phone` type.

If CompanyB decides that the `EstablishConnection` methods are not semantically identical in both types, CompanyB can tell the compiler that the `Dial` and `EstablishConnection` method defined in `BetterPhone` is the correct method to use and that it has no relationship with the `EstablishConnection` method defined in the `Phone` base type. CompanyB informs the compiler of this by adding the `new` keyword to the `EstablishConnection` method.

```
namespace CompanyB {
   public class BetterPhone : CompanyA.Phone {

      // Keep 'new' to mark this method as having no
      // relationship to the base type's Dial method.
      public new void Dial() {
         Console.WriteLine("BetterPhone.Dial");
         EstablishConnection();
         base.Dial();
      }

      // Add 'new' to mark this method as having no
      // relationship to the base type's EstablishConnection method.
      protected new virtual void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // Do work to establish the connection.
      }
   }
}
```

In this code, the new keyword tells the compiler to emit metadata, making it clear to the CLR that BetterPhone's EstablishConnection method is intended to be treated as a new function that is introduced by the BetterPhone type. The CLR will know that there is no relationship between Phone's and BetterPhone's methods.

When the same application code (in the Main method) executes, the output is as follows.

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

This output shows that Main's call to Dial calls the new Dial method defined by BetterPhone.Dial, which in turn calls the virtual EstablishConnection method that is also defined by BetterPhone. When BetterPhone's EstablishConnection method returns, Phone's Dial method is called. Phone's Dial method calls EstablishConnection, but because BetterPhone's EstablishConnection is marked with new, BetterPhone's EstablishConnection method isn't considered an override of Phone's virtual EstablishConnection method. As a result, Phone's Dial method calls Phone's EstablishConnection method—this is the expected behavior.

Note If the compiler treated methods as overrides by default (as a native C++ compiler does), the developer of BetterPhone couldn't use the method names Dial and EstablishConnection. This would most likely cause a ripple effect of changes throughout the entire source code base, breaking source and binary compatibility. This type of pervasive change is undesirable, especially in any moderate-to-large project. However, if changing the method name causes only moderate updates in the source code, you should change the name of the methods so the two different meanings of Dial and EstablishConnection don't confuse other developers.

Alternatively, CompanyB could have gotten the new version of CompanyA's Phone type and decided that Phone's semantics of Dial and EstablishConnection are exactly what it's been looking for. In this case, CompanyB would modify its BetterPhone type by removing its Dial method entirely. In addition, because CompanyB now wants to tell the compiler that BetterPhone's EstablishConnection method is related to Phone's EstablishConnection method, the new keyword must be removed. Simply removing the new keyword isn't enough, though, because now the compiler can't tell exactly what the intention is of BetterPhone's EstablishConnection method. To express his intent exactly, the CompanyB developer must also change BetterPhone's EstablishConnection method from virtual to override. The following code shows the new version of BetterPhone.

```
namespace CompanyB {
   public class BetterPhone : CompanyA.Phone {

   // Delete the Dial method (inherit Dial from base).

   // Remove 'new' and change 'virtual' to 'override' to
   // mark this method as having a relationship to the base
   // type's EstablishConnection method.
   protected override void EstablishConnection() {
      Console.WriteLine("BetterPhone.EstablishConnection");
      // Do work to establish the connection.
   }
 }
}
```

Now when the same application code (in the `Main` method) executes, the output is as follows.

```
Phone.Dial
BetterPhone.EstablishConnection
```

This output shows that `Main`'s call to `Dial` calls the `Dial` method defined by `Phone` and inherited by `BetterPhone`. Then when `Phone`'s `Dial` method calls the virtual `EstablishConnection` method, `BetterPhone`'s `EstablishConnection` method is called because it overrides the virtual `EstablishConnection` method defined by `Phone`.