CHAPTER 1

The CLR's Execution Model

In this chapter:

The Microsoft .NET Framework introduces many concepts, technologies, and terms. My goal in this chapter is to give you an overview of how the .NET Framework is designed, introduce you to some of the technologies the framework includes, and define many of the terms you'll be seeing when you start using it. I'll also take you through the process of building your source code into an application or a set of redistributable components (files) that contain types (classes, structures, etc.) and then explain how your application will execute.

Compiling Source Code into Managed Modules

OK, so you've decided to use the .NET Framework as your development platform. Great! Your first step is to determine what type of application or component you intend to build. Let's just assume that you've completed this minor detail; everything is designed, the specifications are written, and you're ready to start development.

Now you must decide which programming language to use. This task is usually difficult because different languages offer different capabilities. For example, in unmanaged C/C++, you have pretty low-level control of the system. You can manage memory exactly the way you want to, create threads easily if you need to, and so on. Microsoft Visual Basic 6.0, on the other hand, allows you to build UI applications very rapidly and makes it easy for you to control COM objects and databases.

The common language runtime (CLR) is just what its name says it is: a runtime that is usable by different and varied programming languages. The core features of the CLR (such as memory management, assembly loading, security, exception handling, and thread synchronization) are available to any and all programming languages that target it—period. For example, the runtime uses exceptions to report errors, so all languages that target the runtime also get errors reported via exceptions. Another example is that the runtime also allows you to create a thread, so any language that targets the runtime can create a thread.

In fact, at runtime, the CLR has no idea which programming language the developer used for the source code. This means that you should choose whatever programming language allows you to express your intentions most easily. You can develop your code in any programming language you desire as long as the compiler you use to compile your code targets the CLR.

So, if what I say is true, what is the advantage of using one programming language over another? Well, I think of compilers as syntax checkers and "correct code" analyzers. They examine your source code, ensure that whatever you've written makes some sense, and then output code that describes your intention. Different programming languages allow you to develop using different syntax. Don't underestimate the value of this choice. For mathematical or financial applications, expressing your intentions by using APL syntax can save many days of development time when compared to expressing the same intention by using Perl syntax, for example.

Microsoft has created several language compilers that target the runtime: C++/CLI, C# (pronounced "C sharp"), Visual Basic, F# (pronounced "F sharp"), Iron Python, Iron Ruby, and an Intermediate Language (IL) Assembler. In addition to Microsoft, several other companies, colleges, and universities have created compilers that produce code to target the CLR. I'm aware of compilers for Ada, APL, Caml, COBOL, Eiffel, Forth, Fortran, Haskell, Lexico, LISP, LOGO, Lua, Mercury, ML, Mondrian, Oberon, Pascal, Perl, PHP, Prolog, RPG, Scheme, Smalltalk, and Tcl/Tk.

Figure 1-1 shows the process of compiling source code files. As the figure shows, you can create source code files written in any programming language that supports the CLR. Then you use the corresponding compiler to check the syntax and analyze the source code. Regardless of which compiler you use, the result is a *managed module*. A managed module is a standard 32-bit Windows portable executable (PE32) file or a standard 64-bit Windows portable executable (PE32+) file that requires the CLR to execute. By the way, managed assemblies always take advantage of Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) in Windows; these two features improve the security of your whole system.

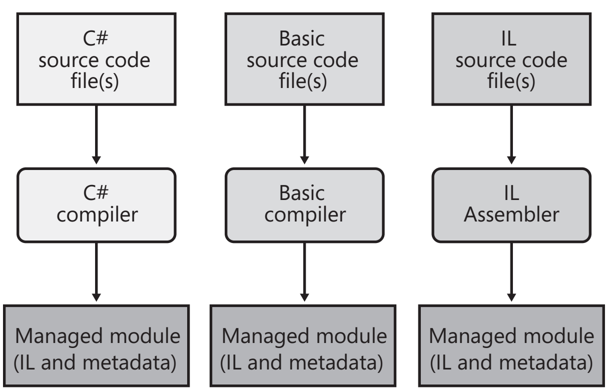

FIGURE 1-1 Compiling source code into managed modules.

Table 1-1 describes the parts of a managed module.

TABLE 1-1 Parts of a Managed Module

Part	Description
PE32 or PE32+ header	The standard Windows PE file header, which is similar to the Common Object File Format (COFF) header. If the header uses the PE32 format, the file can run on a 32-bit or 64-bit version of Windows. If the header uses the PE32+ format, the file requires a 64-bit version of Windows to run. This header also indicates the type of file: GUI, CUI, or DLL, and contains a time stamp indicating when the file was built. For modules that contain only IL code, the bulk of the information in the PE32(+) header is ignored. For modules that contain native CPU code, this header contains information about the native CPU code.
CLR header	Contains the information (interpreted by the CLR and utilities) that makes this a managed module. The header includes the version of the CLR required, some flags, the `MethodDef` metadata token of the managed module's entry point method (`Main` method), and the location/size of the module's metadata, resources, strong name, some flags, and other less interesting stuff.
Metadata	Every managed module contains metadata tables. There are two main types of tables: tables that describe the types and members defined in your source code and tables that describe the types and members referenced by your source code.
IL code	Code the compiler produced as it compiled the source code. At run time, the CLR compiles the IL into native CPU instructions.

Native code compilers produce code targeted to a specific CPU architecture, such as x86, x64, or ARM. All CLR-compliant compilers produce IL code instead. (I'll go into more detail about IL code later in this chapter.) IL code is sometimes referred to as *managed code* because the CLR manages its execution.

In addition to emitting IL, every compiler targeting the CLR is required to emit full *metadata* into every managed module. In brief, metadata is a set of data tables that describe what is defined in the module, such as types and their members. In addition, metadata also has tables indicating what the managed module references, such as imported types and their members. Metadata is a superset of older technologies such as COM's Type Libraries and Interface Definition Language (IDL) files. The important thing to note is that CLR metadata is far more complete. And, unlike Type Libraries and IDL, metadata is always associated with the file that contains the IL code. In fact, the metadata is always embedded in the same EXE/DLL as the code, making it impossible to separate the two. Because the compiler produces the metadata and the code at the same time and binds them into the resulting managed module, the metadata and the IL code it describes are never out of sync with one another.

Metadata has many uses. Here are some of them:

- Metadata removes the need for native C/C++ header and library files when compiling because all the information about the referenced types/members is contained in the file that has the IL that implements the type/members. Compilers can read metadata directly from managed modules.
- Microsoft Visual Studio uses metadata to help you write code. Its IntelliSense feature parses metadata to tell you what methods, properties, events, and fields a type offers, and in the case of a method, what parameters the method expects.

- The CLR's code verification process uses metadata to ensure that your code performs only "type-safe" operations. (I'll discuss verification shortly.)
- Metadata allows an object's fields to be serialized into a memory block, sent to another machine, and then deserialized, re-creating the object's state on the remote machine.
- Metadata allows the garbage collector to track the lifetime of objects. For any object, the garbage collector can determine the type of the object and, from the metadata, know which fields within that object refer to other objects.

In Chapter 2, "Building, Packaging, Deploying, and Administering Applications and Types," I'll describe metadata in much more detail.

Microsoft's C#, Visual Basic, F#, and the IL Assembler always produce modules that contain managed code (IL) and managed data (garbage-collected data types). End users must have the CLR (presently shipping as part of the .NET Framework) installed on their machine in order to execute any modules that contain managed code and/or managed data in the same way that they must have the Microsoft Foundation Class (MFC) library or Visual Basic DLLs installed to run MFC or Visual Basic 6.0 applications.

By default, Microsoft's C++ compiler builds EXE/DLL modules that contain unmanaged (native) code and manipulate unmanaged data (native memory) at run time. These modules don't require the CLR to execute. However, by specifying the /CLR command-line switch, the C++ compiler produces modules that contain managed code, and of course, the CLR must then be installed to execute this code. Of all of the Microsoft compilers mentioned, C++ is unique in that it is the only compiler that allows the developer to write both managed and unmanaged code and have it emitted into a single module. It is also the only Microsoft compiler that allows developers to define both managed and unmanaged data types in their source code. The flexibility provided by Microsoft's C++ compiler is unparalleled by other compilers because it allows developers to use their existing native C/C++ code from managed code and to start integrating the use of managed types as they see fit.

Combining Managed Modules into Assemblies

The CLR doesn't actually work with modules, it works with assemblies. An *assembly* is an abstract concept that can be difficult to grasp initially. First, an assembly is a logical grouping of one or more modules or resource files. Second, an assembly is the smallest unit of reuse, security, and versioning. Depending on the choices you make with your compilers or tools, you can produce a single-file or a multifile assembly. In the CLR world, an assembly is what we would call a *component*.

In Chapter 2, I'll go over assemblies in great detail, so I don't want to spend a lot of time on them here. All I want to do now is make you aware that there is this extra conceptual notion that offers a way to treat a group of files as a single entity.

Figure 1-2 should help explain what assemblies are about. In this figure, some managed modules and resource (or data) files are being processed by a tool. This tool produces a single PE32(+) file that represents the logical grouping of files. What happens is that this PE32(+) file contains a block of data called the *manifest*. The manifest is simply another set of metadata tables. These tables describe the files that make up the assembly, the publicly exported types implemented by the files in the assembly, and the resource or data files that are associated with the assembly.

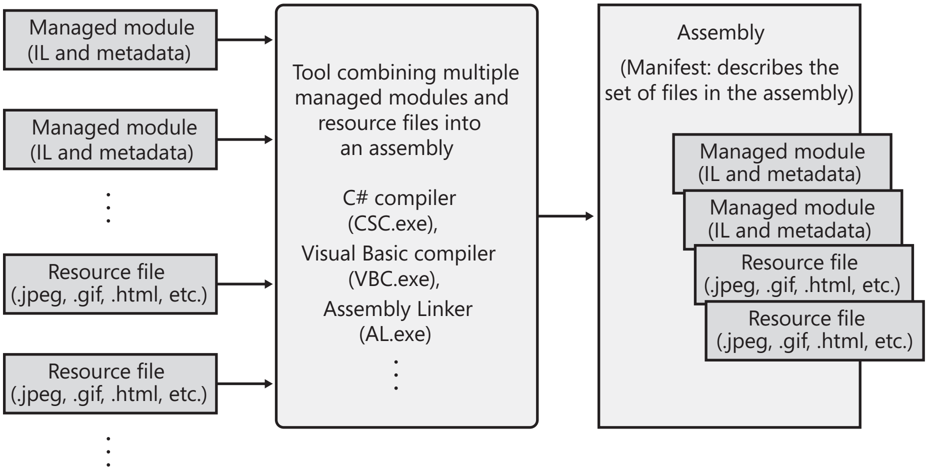

FIGURE 1-2 Combining managed modules into assemblies.

By default, compilers actually do the work of turning the emitted managed module into an assembly; that is, the C# compiler emits a managed module that contains a manifest. The manifest indicates that the assembly consists of just the one file. So, for projects that have just one managed module and no resource (or data) files, the assembly will be the managed module, and you don't have any additional steps to perform during your build process. If you want to group a set of files into an assembly, you'll have to be aware of more tools (such as the assembly linker, AL.exe) and their command-line options. I'll explain these tools and options in Chapter 2.

An assembly allows you to decouple the logical and physical notions of a reusable, securable, versionable component. How you partition your code and resources into different files is completely up to you. For example, you could put rarely used types or resources in separate files that are part of an assembly. The separate files could be downloaded on demand from the web as they are needed at run time. If the files are never needed, they're never downloaded, saving disk space and reducing installation time. Assemblies allow you to break up the deployment of the files while still treating all of the files as a single collection.

An assembly's modules also include information about referenced assemblies (including their version numbers). This information makes an assembly *self-describing*. In other words, the CLR can determine the assembly's immediate dependencies in order for code in the assembly to execute. No additional information is required in the registry or in Active Directory Domain Services (AD DS). Because no additional information is needed, deploying assemblies is much easier than deploying unmanaged components.

Loading the Common Language Runtime

Each assembly you build can be either an executable application or a DLL containing a set of types for use by an executable application. Of course, the CLR is responsible for managing the execution of code contained within these assemblies. This means that the .NET Framework must be installed on the host machine. Microsoft has created a redistribution package that you can freely ship to install the .NET Framework on your customers' machines. Some versions of Windows ship with the .NET Framework already installed.

You can tell if the .NET Framework has been installed by looking for the MSCorEE.dll file in the %SystemRoot%\System32 directory. The existence of this file tells you that the .NET Framework is installed. However, several versions of the .NET Framework can be installed on a single machine simultaneously. If you want to determine exactly which versions of the .NET Framework are installed, examine the subdirectories under the following directories.

```
%SystemRoot%\Microsoft.NET\Framework
%SystemRoot%\Microsoft.NET\Framework64
```

The .NET Framework SDK includes a command-line utility called CLRVer.exe that shows all of the CLR versions installed on a machine. This utility can also show which version of the CLR is being used by processes currently running on the machine by using the `–all` switch or passing the ID of the process you are interested in.

Before we start looking at how the CLR loads, we need to spend a moment discussing 32-bit and 64-bit versions of Windows. If your assembly files contain only type-safe managed code, you are writing code that should work on both 32-bit and 64-bit versions of Windows. No source code changes are required for your code to run on either version of Windows. In fact, the resulting EXE/DLL file produced by the compiler should work correctly when running on x86 and x64 versions of Windows. In addition, Windows Store applications or class libraries will run on Windows RT machines (which use an ARM CPU). In other words, the one file will run on any machine that has the corresponding version of the .NET Framework installed on it.

On extremely rare occasions, developers want to write code that works only on a specific version of Windows. Developers might do this when using unsafe code or when interoperating with unmanaged code that is targeted to a specific CPU architecture. To aid these developers, the C# compiler offers a `/platform` command-line switch. This switch allows you to specify whether the resulting assembly can run on x86 machines running 32-bit Windows versions only, x64 machines running 64-bit Windows only, or ARM machines running 32-bit Windows RT only. If you don't specify a platform, the default is `anycpu`, which indicates that the resulting assembly can run on any version of Windows. Users of Visual Studio can set a project's target platform by displaying the project's property pages, clicking the Build tab, and then selecting an option in the Platform Target list (see Figure 1-3).

In Figure 1-3, you'll notice the Prefer 32-Bit check box. This check box is only enabled when Platform Target is set to Any CPU and if the project type produces an executable. If you select the Prefer 32-Bit check box, then Visual Studio spawns the C# compiler specifying the `/platform: anycpu-32bitpreferred` compiler switch. This option indicates that the executable should run as a 32-bit executable even when running on a 64-bit machine. If your application doesn't require the additional memory afforded to a 64-bit process, then this is typically a good way to go because Visual Studio does not support edit-and-continue of x64 applications. In addition, 32-bit applications can interoperate with 32-bit DLLs and COM components should your application require this.

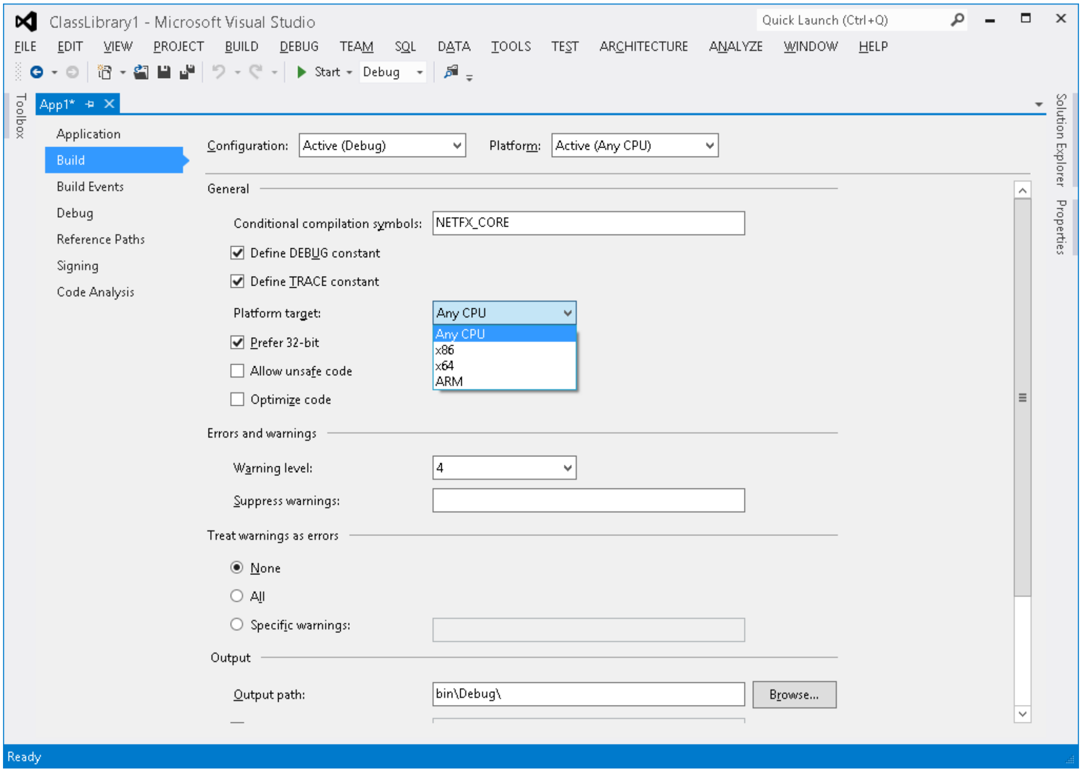

FIGURE 1-3 Setting the platform target by using Visual Studio.

Depending on the platform switch, the C# compiler will emit an assembly that contains either a PE32 or PE32+ header, and the compiler will also emit the desired CPU architecture (or agnostic) into the header as well. Microsoft ships two SDK command-line utilities, DumpBin.exe and CorFlags.exe, that you can use to examine the header information emitted in a managed module by the compiler.

When running an executable file, Windows examines this EXE file's header to determine whether the application requires a 32-bit or 64-bit address space. A file with a PE32 header can run with a 32-bit or 64-bit address space, and a file with a PE32+ header requires a 64-bit address space. Windows also checks the CPU architecture information embedded inside the header to ensure that it matches the CPU type in the computer. Lastly, 64-bit versions of Windows offer a technology that allows 32-bit Windows applications to run. This technology is called *WoW64* (for Windows on Windows 64).

Table 1-2 shows two things. First, it shows what kind of managed module you get when you specify various `/platform` command-line switches to the C# compiler. Second, it shows how that application will run on various versions of Windows.

TABLE 1-2 Effects of `/platform` on Resulting Module and at Run Time

/platform Switch	Resulting Managed Module	x86 Windows	x64 Windows	ARM Windows RT
anycpu (the default)	PE32/agnostic	Runs as a 32-bit application	Runs as a 64-bit application	Runs as a 32-bit application
anycpu32bitpreferred	PE32/agnostic	Runs as a 32-bit application	Runs as a 32-bit application	Runs as a 32-bit application
x86	PE32/x86	Runs as a 32-bit application	Runs as a WoW64 application	Doesn't run
x64	PE32+/x64	Doesn't run	Runs as a 64-bit application	Doesn't run
ARM	PE32/ARM	Doesn't run	Doesn't run	Runs as a 32-bit application

After Windows has examined the EXE file's header to determine whether to create a 32-bit or 64-bit process, Windows loads the x86, x64, or ARM version of MSCorEE.dll into the process's address space. On an x86 or ARM version of Windows, the 32-bit version of MSCorEE.dll can be found in the %SystemRoot%\System32 directory. On an x64 version of Windows, the x86 version of MSCorEE.dll can be found in the %SystemRoot%\SysWow64 directory, whereas the 64-bit version can be found in the %SystemRoot%\System32 directory (for backward compatibility reasons). Then, the process's primary thread calls a method defined inside MSCorEE.dll. This method initializes the CLR, loads the EXE assembly, and then calls its entry point method (`Main`). At this point, the managed application is up and running.[1]

> **Note** Assemblies built by using version 1.0 or 1.1 of Microsoft's C# compiler contain a PE32 header and are CPU-architecture agnostic. However, at load time, the CLR considers these assemblies to be x86 only. For executable files, this improves the likelihood of the application actually working on a 64-bit system because the executable file will load in WoW64, giving the process an environment very similar to what it would have on a 32-bit x86 version of Windows.

If an unmanaged application calls the Win32 `LoadLibrary` function to load a managed assembly, Windows knows to load and initialize the CLR (if not already loaded) in order to process the code contained within the assembly. Of course, in this scenario, the process is already up and running, and this may limit the usability of the assembly. For example, a managed assembly compiled with the `/platform:x86` switch will not be able to load into a 64-bit process at all, whereas an executable file compiled with this same switch would have loaded in WoW64 on a computer running a 64-bit version of Windows.

[1] Your code can query `Environment`'s `Is64BitOperatingSystem` property to determine if it is running on a 64-bit version of Windows. Your code can also query `Environment`'s `Is64BitProcess` property to determine if it is running in a 64-bit address space.

Executing Your Assembly's Code

As mentioned earlier, managed assemblies contain both metadata and IL. IL is a CPU-independent machine language created by Microsoft after consultation with several external commercial and academic language/compiler writers. IL is a much higher-level language than most CPU machine languages. IL can access and manipulate object types and has instructions to create and initialize objects, call virtual methods on objects, and manipulate array elements directly. It even has instructions to throw and catch exceptions for error handling. You can think of IL as an object-oriented machine language.

Usually, developers will program in a high-level language, such as C#, Visual Basic, or F#. The compilers for these high-level languages produce IL. However, as any other machine language, IL can be written in assembly language, and Microsoft does provide an IL Assembler, ILAsm.exe. Microsoft also provides an IL Disassembler, ILDasm.exe.

Keep in mind that any high-level language will most likely expose only a subset of the facilities offered by the CLR. However, the IL assembly language allows a developer to access all of the CLR's facilities. So, should your programming language of choice hide a facility the CLR offers that you really want to take advantage of, you can choose to write that portion of your code in IL assembly or perhaps another programming language that exposes the CLR feature you seek.

The only way for you to know what facilities the CLR offers is to read documentation specific to the CLR itself. In this book, I try to concentrate on CLR features and how they are exposed or not exposed by the C# language. I suspect that most other books and articles will present the CLR via a language perspective, and that most developers will come to believe that the CLR offers only what the developer's chosen language exposes. As long as your language allows you to accomplish what you're trying to get done, this blurred perspective isn't a bad thing.

> **Important** I think this ability to switch programming languages easily with rich integration between languages is an awesome feature of the CLR. Unfortunately, I also believe that developers will often overlook this feature. Programming languages such as C# and Visual Basic are excellent languages for performing I/O operations. APL is a great language for performing advanced engineering or financial calculations. Through the CLR, you can write the I/O portions of your application in C# and then write the engineering calculations part in APL. The CLR offers a level of integration between these languages that is unprecedented and really makes mixed-language programming worthy of consideration for many development projects.

To execute a method, its IL must first be converted to native CPU instructions. This is the job of the CLR's JIT (just-in-time) compiler.

Figure 1-4 shows what happens the first time a method is called.

Just before the `Main` method executes, the CLR detects all of the types that are referenced by `Main`'s code. This causes the CLR to allocate an internal data structure that is used to manage access to the referenced types. In Figure 1-4, the `Main` method refers to a single type, `Console`, causing the CLR to allocate a single internal structure. This internal data structure contains an entry for each method defined by the `Console` type. Each entry holds the address where the method's implementation can be found. When initializing this structure, the CLR sets each entry to an internal, undocumented function contained inside the CLR itself. I call this function `JITCompiler`.

When `Main` makes its first call to `WriteLine`, the `JITCompiler` function is called. The `JITCompiler` function is responsible for compiling a method's IL code into native CPU instructions. Because the IL is being compiled "just in time," this component of the CLR is frequently referred to as a *JITter* or a *JIT compiler*.

> **Note** If the application is running on an x86 version of Windows or in WoW64, the JIT compiler produces x86 instructions. If your application is running as a 64-bit application on an x64 version of Windows, the JIT compiler produces x64 instructions. If the application is running on an ARM version of Windows, the JIT compiler produces ARM instructions.

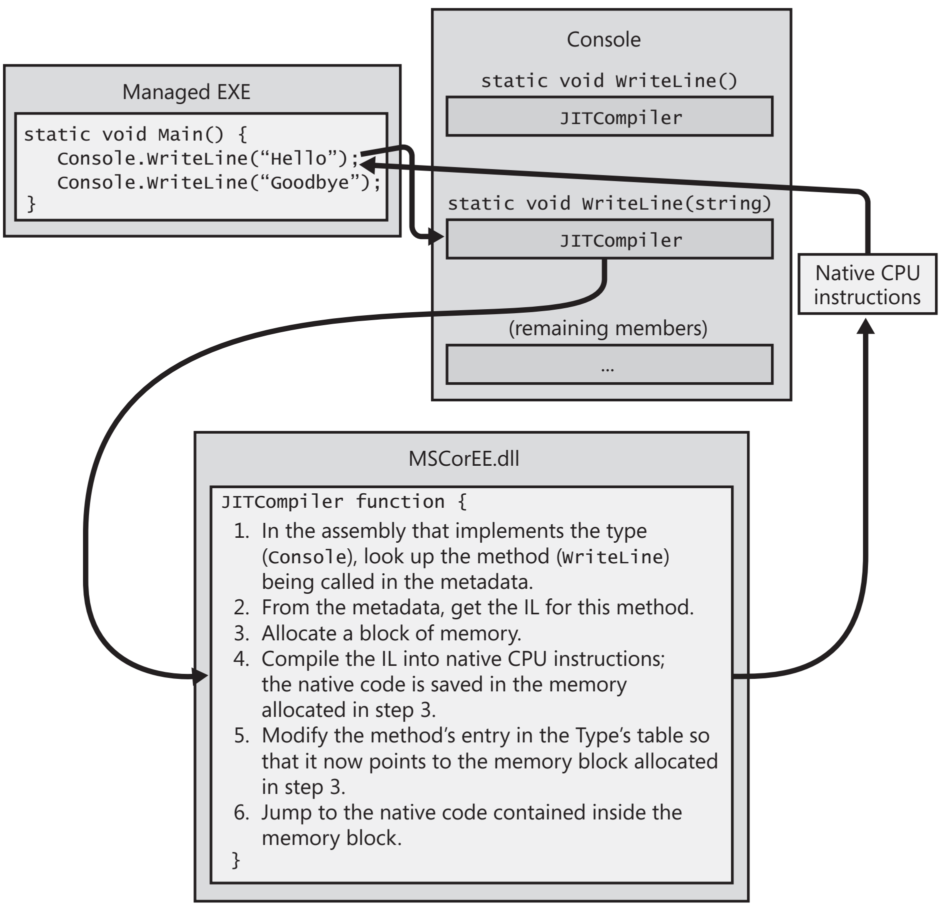

FIGURE 1-4 Calling a method for the first time.

When called, the `JITCompiler` function knows what method is being called and what type defines this method. The `JITCompiler` function then searches the defining assembly's metadata for the called method's IL. `JITCompiler` next verifies and compiles the IL code into native CPU instructions. The native CPU instructions are saved in a dynamically allocated block of memory. Then, `JITCompiler` goes back to the entry for the called method in the type's internal data structure created by the CLR and replaces the reference that called it in the first place with the address of the block of memory containing the native CPU instructions it just compiled. Finally, the `JITCompiler` function jumps to the code in the memory block. This code is the implementation of the `WriteLine` method (the version that takes a `String` parameter). When this code returns, it returns to the code in `Main`, which continues execution as normal.

`Main` now calls `WriteLine` a second time. This time, the code for `WriteLine` has already been verified and compiled. So the call goes directly to the block of memory, skipping the `JITCompiler` function entirely. After the `WriteLine` method executes, it returns to `Main`. Figure 1-5 shows what the process looks like when `WriteLine` is called the second time.

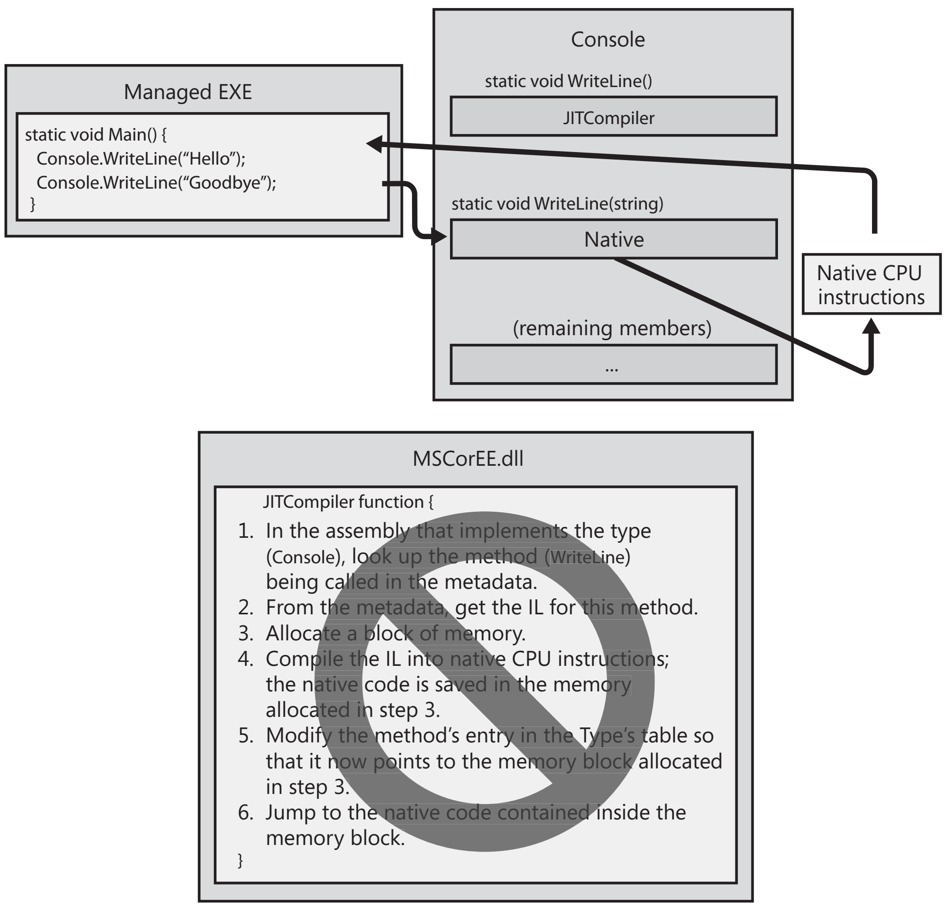

FIGURE 1-5 Calling a method for the second time.

A performance hit is incurred only the first time a method is called. All subsequent calls to the method execute at the full speed of the native code because verification and compilation to native code don't need to be performed again.

The JIT compiler stores the native CPU instructions in dynamic memory. This means that the compiled code is discarded when the application terminates. So if you run the application again in the future or if you run two instances of the application simultaneously (in two different operating system processes), the JIT compiler will have to compile the IL to native instructions again. Depending on the application, this can increase memory consumption significantly compared to a native application whose read-only code pages can be shared by all instances of the running application.

For most applications, the performance hit incurred by JIT compiling isn't significant. Most applications tend to call the same methods over and over again. These methods will take the performance hit only once while the application executes. It's also likely that more time is spent inside the method than calling the method.

You should also be aware that the CLR's JIT compiler optimizes the native code just as the back end of an unmanaged C++ compiler does. Again, it may take more time to produce the optimized code, but the code will execute with much better performance than if it hadn't been optimized.

There are two C# compiler switches that impact code optimization: `/optimize` and `/debug`. The following table shows the impact these switches have on the quality of the IL code generated by the C# compiler and the quality of the native code generated by the JIT compiler.

Compiler Switch Settings	C# IL Code Quality	JIT Native Code Quality
`/optimize- /debug-` (this is the default)	Unoptimized	Optimized
`/optimize- /debug(+/full/pdbonly)`	Unoptimized	Unoptimized
`/optimize+ /debug(-/+/full/pdbonly)`	Optimized	Optimized

With `/optimize-`, the unoptimized IL code produced by the C# compiler contains many no-operation (NOP) instructions and also branches that jump to the next line of code. These instructions are emitted to enable the edit-and-continue feature of Visual Studio while debugging and the extra instructions also make code easier to debug by allowing breakpoints to be set on control flow instructions such as `for`, `while`, `do`, `if`, `else`, `try`, `catch`, and `finally` statement blocks. When producing optimized IL code, the C# compiler will remove these extraneous NOP and branch instructions, making the code harder to single-step through in a debugger because control flow will be optimized. Also, some function evaluations may not work when performed inside the debugger. However, the IL code is smaller, making the resulting EXE/DLL file smaller, and the IL tends to be easier to read for those of you (like me) who enjoy examining the IL to understand what the compiler is producing.

Furthermore, the compiler produces a Program Database (PDB) file only if you specify the `/debug(+/full/pdbonly)` switch. The PDB file helps the debugger find local variables and map the IL instructions to source code. The `/debug:full` switch tells the JIT compiler that you intend to debug the assembly, and the JIT compiler will track what native code came from each IL instruction. This allows you to use the just-in-time debugger feature of Visual Studio to connect a debugger to an already-running process and debug the code easily. Without the `/debug:full` switch, the JIT compiler does not, by default, track the IL to native code information, which makes the JIT compiler run a little faster and also uses a little less memory. If you start a process with the Visual Studio debugger, it forces the JIT compiler to track the IL to native code information (regardless of the `/debug` switch) unless you turn off the Suppress JIT Optimization On Module Load (Managed Only) option in Visual Studio.

When you create a new C# project in Visual Studio, the Debug configuration of the project has `/optimize-` and `/debug:full` switches, and the Release configuration has `/optimize+` and `/debug:pdbonly` switches specified.

For those developers coming from an unmanaged C or C++ background, you're probably thinking about the performance ramifications of all this. After all, unmanaged code is compiled for a specific CPU platform, and, when invoked, the code can simply execute. In this managed environment, compiling the code is accomplished in two phases. First, the compiler passes over the source code, doing as much work as possible in producing IL. But to execute the code, the IL itself must be compiled into native CPU instructions at run time, requiring more non-shareable memory to be allocated and requiring additional CPU time to do the work.

Believe me, because I approached the CLR from a C/C++ background myself, I was quite skeptical and concerned about this additional overhead. The truth is that this second compilation stage that occurs at run time does hurt performance, and it does allocate dynamic memory. However, Microsoft has done a lot of performance work to keep this additional overhead to a minimum.

If you too are skeptical, you should certainly build some applications and test the performance for yourself. In addition, you should run some nontrivial managed applications Microsoft or others have produced, and measure their performance. I think you'll be surprised at how good the performance actually is.

You'll probably find this hard to believe, but many people (including me) think that managed applications could actually outperform unmanaged applications. There are many reasons to believe this. For example, when the JIT compiler compiles the IL code into native code at run time, the compiler knows more about the execution environment than an unmanaged compiler would know. Here are some ways that managed code can outperform unmanaged code:

- A JIT compiler can determine if the application is running on an Intel Pentium 4 CPU and produce native code that takes advantage of any special instructions offered by the Pentium 4. Usually, unmanaged applications are compiled for the lowest-common-denominator CPU and avoid using special instructions that would give the application a performance boost.

- A JIT compiler can determine when a certain test is always false on the machine that it is running on. For example, consider a method that contains the following code.

  ```
  if (numberOfCPUs > 1) {
   ...
  }
  ```

  This code could cause the JIT compiler to not generate any CPU instructions if the host machine has only one CPU. In this case, the native code would be fine-tuned for the host machine; the resulting code is smaller and executes faster.

- The CLR could profile the code's execution and recompile the IL into native code while the application runs. The recompiled code could be reorganized to reduce incorrect branch predictions depending on the observed execution patterns. Current versions of the CLR do not do this, but future versions might.

These are only a few of the reasons why you should expect future managed code to execute better than today's unmanaged code. As I said, the performance is currently quite good for most applications, and it promises to improve as time goes on.

If your experiments show that the CLR's JIT compiler doesn't offer your application the kind of performance it requires, you may want to take advantage of the NGen.exe tool that ships with the .NET Framework SDK. This tool compiles all of an assembly's IL code into native code and saves the resulting native code to a file on disk. At run time, when an assembly is loaded, the CLR automatically checks to see whether a precompiled version of the assembly also exists, and if it does, the CLR loads the precompiled code so that no compilation is required at run time. Note that NGen.exe must be conservative about the assumptions it makes regarding the actual execution environment, and for this reason, the code produced by NGen.exe will not be as highly optimized as the JIT compiler–produced code. I'll discuss NGen.exe in more detail later in this chapter.

In addition, you may want to consider using the `System.Runtime.ProfileOptimization` class. This class causes the CLR to record (to a file) what methods get JIT compiled while your application is running. Then, on a future startup of your application, the JIT compiler will concurrently compile these methods by using other threads if your application is running on a machine with multiple CPUs. The end result is that your application runs faster because multiple methods get compiled concurrently, and during application initialization instead of compiling the methods just in time as the user is interacting with your application.

IL and Verification

IL is stack-based, which means that all of its instructions push operands onto an execution stack and pop results off the stack. Because IL offers no instructions to manipulate registers, it is easy for people to create new languages and compilers that produce code targeting the CLR.

IL instructions are also typeless. For example, IL offers an add instruction that adds the last two operands pushed on the stack. There are no separate 32-bit and 64-bit versions of the add instruction. When the add instruction executes, it determines the types of the operands on the stack and performs the appropriate operation.

In my opinion, the biggest benefit of IL isn't that it abstracts away the underlying CPU. The biggest benefit IL provides is application robustness and security. While compiling IL into native CPU instructions, the CLR performs a process called *verification*. Verification examines the high-level IL code and ensures that everything the code does is safe. For example, verification checks that every method is called with the correct number of parameters, that each parameter passed to every method is of the correct type, that every method's return value is used properly, that every method has a return statement, and so on. The managed module's metadata includes all of the method and type information used by the verification process.

In Windows, each process has its own virtual address space. Separate address spaces are necessary because you can't trust an application's code. It is entirely possible (and unfortunately, all too common) that an application will read from or write to an invalid memory address. By placing each Windows process in a separate address space, you gain robustness and stability; one process can't adversely affect another process.

By verifying the managed code, however, you know that the code doesn't improperly access memory and can't adversely affect another application's code. This means that you can run multiple managed applications in a single Windows virtual address space.

Because Windows processes require a lot of operating system resources, having many of them can hurt performance and limit available resources. Reducing the number of processes by running multiple applications in a single operating system process can improve performance, require fewer resources, and be just as robust as if each application had its own process. This is another benefit of managed code as compared to unmanaged code.

The CLR does, in fact, offer the ability to execute multiple managed applications in a single operating system process. Each managed application executes in an AppDomain. By default, every managed EXE file will run in its own separate address space that has just one AppDomain. However, a process hosting the CLR (such as Internet Information Services [IIS] or Microsoft SQL Server) can decide to run AppDomains in a single operating system process. I'll devote part of Chapter 22, "CLR Hosting and AppDomains," to a discussion of AppDomains.

Unsafe Code

By default, Microsoft's C# compiler produces safe code. Safe code is code that is verifiably safe. However, Microsoft's C# compiler allows developers to write unsafe code. Unsafe code is allowed to work directly with memory addresses and can manipulate bytes at these addresses. This is a very powerful feature and is typically useful when interoperating with unmanaged code or when you want to improve the performance of a time-critical algorithm.

However, using unsafe code introduces a significant risk: unsafe code can corrupt data structures and exploit or even open up security vulnerabilities. For this reason, the C# compiler requires that all methods that contain unsafe code be marked with the `unsafe` keyword. In addition, the C# compiler requires you to compile the source code by using the `/unsafe` compiler switch.

When the JIT compiler attempts to compile an unsafe method, it checks to see if the assembly containing the method has been granted the `System.Security.Permissions.SecurityPermission` with the `System.Security.Permissions.SecurityPermissionFlag`'s `SkipVerification` flag set. If this flag is set, the JIT compiler will compile the unsafe code and allow it to execute. The CLR is trusting this code and is hoping the direct address and byte manipulations do not cause any harm. If the flag is not set, the JIT compiler throws either a `System.InvalidProgramException` or a `System.Security.VerificationException`, preventing the method from executing. In fact, the whole application will probably terminate at this point, but at least no harm can be done.

> **Note** By default, assemblies that load from the local machine or via network shares are granted full trust, meaning that they can do anything, which includes executing unsafe code. However, by default, assemblies executed via the Internet are not granted the permission to execute unsafe code. If they contain unsafe code, one of the aforementioned exceptions is thrown. An administrator/end user can change these defaults; however, the administrator is taking full responsibility for the code's behavior.

Microsoft supplies a utility called PEVerify.exe, which examines all of an assembly's methods and notifies you of any methods that contain unsafe code. You may want to consider running PEVerify.exe on assemblies that you are referencing; this will let you know if there may be problems running your application via the intranet or Internet.

You should be aware that verification requires access to the metadata contained in any dependent assemblies. So when you use PEVerify to check an assembly, it must be able to locate and load all referenced assemblies. Because PEVerify uses the CLR to locate the dependent assemblies, the assemblies are located using the same binding and probing rules that would normally be used when executing the assembly. I'll discuss these binding and probing rules in Chapter 2 and Chapter 3, "Shared Assemblies and Strongly Named Assemblies."

IL and Protecting Your Intellectual Property

Some people are concerned that IL doesn't offer enough intellectual property protection for their algorithms. In other words, they think that you could build a managed module and that someone else could use a tool, such as an IL Disassembler, to easily reverse engineer exactly what your application's code does.

Yes, it's true that IL code is higher-level than most other assembly languages, and, in general, reverse engineering IL code is relatively simple. However, when implementing server-side code (such as a web service, web form, or stored procedure), your assembly resides on your server. Because no one outside of your company can access the assembly, no one outside of your company can use any tool to see the IL—your intellectual property is completely safe.

If you're concerned about any of the assemblies you do distribute, you can obtain an obfuscator utility from a third-party vendor. These utilities scramble the names of all of the private symbols in your assembly's metadata. It will be difficult for someone to unscramble the names and understand the purpose of each method. Note that these obfuscators can provide only a little protection because the IL must be available at some point for the CLR to JIT compile it.

If you don't feel that an obfuscator offers the kind of intellectual property protection you desire, you can consider implementing your more sensitive algorithms in some unmanaged module that will contain native CPU instructions instead of IL and metadata. Then you can use the CLR's interoperability features (assuming that you have ample permissions) to communicate between the managed and unmanaged portions of your application. Of course, this assumes that you're not worried about people reverse engineering the native CPU instructions in your unmanaged code.

The Native Code Generator Tool: NGen.exe

The NGen.exe tool that ships with the .NET Framework can be used to compile IL code to native code when an application is installed on a user's machine. Because the code is compiled at install time, the CLR's JIT compiler does not have to compile the IL code at run time, and this can improve the application's performance. The NGen.exe tool is interesting in two scenarios:

- **Improving an application's startup time** Running NGen.exe can improve startup time because the code will already be compiled into native code so that compilation doesn't have to occur at run time.
- **Reducing an application's working set** If you believe that an assembly will be loaded into multiple processes simultaneously, running NGen.exe on that assembly can reduce the applications' working set. The reason is because the NGen.exe tool compiles the IL to native code and saves the output in a separate file. This file can be memory-mapped into multiple-process address spaces simultaneously, allowing the code to be shared; not every process needs its own copy of the code.

When a setup program invokes NGen.exe on an application or a single assembly, all of the assemblies for that application or the one specified assembly have their IL code compiled into native code. A new assembly file containing only this native code instead of IL code is created by NGen.exe. This new file is placed in a folder under the directory with a name like `%SystemRoot%\Assembly\NativeImages_v4.0.#####_64`. The directory name includes the version of the CLR and information denoting whether the native code is compiled for 32-bit or 64-bit versions of Windows.

Now, whenever the CLR loads an assembly file, the CLR looks to see if a corresponding NGen'd native file exists. If a native file cannot be found, the CLR JIT compiles the IL code as usual. However, if a corresponding native file does exist, the CLR will use the compiled code contained in the native file, and the file's methods will not have to be compiled at run time.

On the surface, this sounds great! It sounds as if you get all of the benefits of managed code (garbage collection, verification, type safety, and so on) without all of the performance problems of managed code (JIT compilation). However, the reality of the situation is not as rosy as it would first seem. There are several potential problems with respect to NGen'd files:

- **No intellectual property protection** Many people believe that it might be possible to ship NGen'd files without shipping the files containing the original IL code, thereby keeping their intellectual property a secret. Unfortunately, this is not possible. At run time, the CLR requires access to the assembly's metadata (for functions such as reflection and serialization); this requires that the assemblies that contain IL and metadata be shipped. In addition, if the CLR can't use the NGen'd file for some reason (described next), the CLR gracefully goes back to JIT compiling the assembly's IL code, which must be available.

- **NGen'd files can get out of sync** When the CLR loads an NGen'd file, it compares a number of characteristics about the previously compiled code and the current execution environment. If any of the characteristics don't match, the NGen'd file cannot be used, and the normal JIT compiler process is used instead. Here is a partial list of characteristics that must match:

    - CLR version: This changes with patches or service packs.

    - CPU type: This changes if you upgrade your processor hardware.

    - Windows operating system version: This changes with a new service pack update.

    - Assembly's identity module version ID (MVID): This changes when recompiling.

    - Referenced assembly's version IDs: This changes when you recompile a referenced assembly.

    - Security: This changes when you revoke permissions (such as declarative inheritance, declarative link-time, `SkipVerification`, or `UnmanagedCode` permissions), that were once granted.

> **Note** It is possible to run NGen.exe in update mode. This tells the tool to run NGen.exe on all of the assemblies that had previously been NGen'd. Whenever an end user installs a new service pack of the .NET Framework, the service pack's installation program will run NGen.exe in update mode automatically so that NGen'd files are kept in sync with the version of the CLR installed.

- **Inferior execution-time performance** When compiling code, NGen can't make as many assumptions about the execution environment as the JIT compiler can. This causes NGen.exe to produce inferior code. For example, NGen won't optimize the use of certain CPU instructions; it adds indirections for static field access because the actual address of the static fields isn't known until run time. NGen inserts code to call class constructors everywhere because it doesn't know the order in which the code will execute and if a class constructor has already been called. (See Chapter 8, "Methods," for more about class constructors.) Some NGen'd applications actually perform about 5 percent slower when compared to their JIT-compiled counterpart. So, if you're considering using NGen.exe to improve the performance of your application, you should compare NGen'd and non-NGen'd versions to be sure that the NGen'd version doesn't actually run slower! For some applications, the reduction in working set size improves performance, so using NGen can be a net win.

Due to all of the issues just listed, you should be very cautious when considering the use of NGen.exe. For server-side applications, NGen.exe makes little or no sense because only the first client request experiences a performance hit; future client requests run at high speed. In addition, for most server applications, only one instance of the code is required, so there is no working set benefit.

For client applications, NGen.exe might make sense to improve startup time or to reduce working set if an assembly is used by multiple applications simultaneously. Even in a case in which an assembly is not used by multiple applications, NGen'ing an assembly could improve working set. Moreover, if NGen.exe is used for all of a client application's assemblies, the CLR will not need to load the JIT compiler at all, reducing working set even further. Of course, if just one assembly isn't NGen'd or if an assembly's NGen'd file can't be used, the JIT compiler will load, and the application's working set increases.

For large client applications that experience very long startup times, Microsoft provides a Managed Profile Guided Optimization tool (MPGO.exe). This tool analyzes the execution of your application to see what it needs at startup. This information is then fed to the NGen.exe tool in order to better optimize the resulting native image. This allows your application to start faster and with a reduced working set. When you're ready to ship your application, launch it via the MPGO tool and then exercise your application's common tasks. Information about the parts of your code that executed is written to a profile, which is embedded within your assembly file. The NGen.exe tool uses this profile data to better optimize the native image it produces.

The Framework Class Library

The .NET Framework includes the *Framework Class Library (FCL)*. The FCL is a set of DLL assemblies that contain several thousand type definitions in which each type exposes some functionality. Microsoft is producing additional libraries such as the Windows Azure SDK and the DirectX SDK. These additional libraries provide even more types, exposing even more functionality for your use. In fact, Microsoft is producing many libraries at a phenomenal rate, making it easier than ever for developers to use various Microsoft technologies.

Here are just some of the kinds of applications developers can create by using these assemblies:

- **Web services** Methods that can process messages sent over the Internet very easily using Microsoft's ASP.NET XML Web Service technology or Microsoft's Windows Communication Foundation (WCF) technology.
- **Web Forms/MVC HTML-based applications (websites**) Typically, ASP.NET applications will make database queries and web service calls, combine and filter the returned information, and then present that information in a browser by using a rich HTML-based user interface.
- **Rich Windows GUI applications** Instead of using a webpage to create your application's UI, you can use the more powerful, higher-performance functionality offered by Windows Store, Windows Presentation Foundation (WPF), or Windows Forms technologies. GUI applications can take advantage of controls, menus, and touch, mouse, stylus, and keyboard events, and they can exchange information directly with the underlying operating system. Rich Windows applications can also make database queries and consume web services.
- **Windows console applications** For applications with very simple UI demands, a console application provides a quick and easy way to build an application. Compilers, utilities, and tools are typically implemented as console applications.
- **Windows services** Yes, it is possible to build service applications that are controllable via the Windows Service Control Manager (SCM) by using the .NET Framework.
- **Database stored procedures** Microsoft's SQL Server, IBM's DB2, and Oracle's database servers allow developers to write their stored procedures by using the .NET Framework.
- **Component library** The .NET Framework allows you to build stand-alone assemblies (components) containing types that can be easily incorporated into any of the previously mentioned application types.

> **Important** Visual Studio allows you to create a Portable Class Library project. This project type lets you create a single class library assembly that works with various application types, including the .NET Framework proper, Silverlight, Windows Phone, Windows Store apps, and Xbox 360.

Because the FCL contains literally thousands of types, a set of related types is presented to the developer within a single namespace. For example, the `System` namespace (which you should become most familiar with) contains the `Object` base type, from which all other types ultimately derive. In addition, the `System` namespace contains types for integers, characters, strings, exception handling, and console I/O as well as a bunch of utility types that convert safely between data types, format data types, generate random numbers, and perform various math functions. All applications will use types from the `System` namespace.

To access any of the framework's features, you need to know which namespace contains the types that expose the facilities you're after. A lot of types allow you to customize their behavior; you do so by simply deriving your own type from the desired FCL type. The object-oriented nature of the platform is how the .NET Framework presents a consistent programming paradigm to software developers. Also, developers can easily create their own namespaces containing their own types. These namespaces and types merge seamlessly into the programming paradigm. Compared to Win32 programming paradigms, this new approach greatly simplifies software development.

Most of the namespaces in the FCL present types that can be used for any kind of application. Table 1-3 lists some of the more general namespaces and briefly describes what the types in that namespace are used for. This is a very small sampling of the namespaces available. Please see the documentation that accompanies the various Microsoft SDKs to gain familiarity with the ever-growing set of namespaces that Microsoft is producing.

TABLE 1-3 Some General FCL Namespaces

Namespace	Description of Contents
`System`	All of the basic types used by every application
`System.Data`	Types for communicating with a database and processing data
`System.IO`	Types for doing stream I/O and walking directories and files
`System.Net`	Types that allow for low-level network communications and working with some common Internet protocols
`System.Runtime.InteropServices`	Types that allow managed code to access unmanaged operating system platform facilities such as COM components and functions in Win32 or custom DLLs
`System.Security`	Types used for protecting data and resources
`System.Text`	Types to work with text in different encodings, such as ASCII and Unicode
`System.Threading`	Types used for asynchronous operations and synchronizing access to resources
`System.Xml`	Types used for processing Extensible Markup Language (XML) schemas and data

This book is about the CLR and about the general types that interact closely with the CLR. So the content of this book is applicable to all programmers writing applications or components that target the CLR. Many other good books exist that cover specific application types such as Web Services, Web Forms/MVC, Windows Presentation Foundation, etc. These other books will give you an excellent start at helping you build your application. I tend to think of these application-specific books as helping you learn from the top down because they concentrate on the application type and not on the development platform. In this book, I'll offer information that will help you learn from the bottom up. After reading this book and an application-specific book, you should be able to easily and proficiently build any kind of application you desire.

The Common Type System

By now, it should be obvious to you that the CLR is all about types. Types expose functionality to your applications and other types. Types are the mechanism by which code written in one programming language can talk to code written in a different programming language. Because types are at the root of the CLR, Microsoft created a formal specification—the Common Type System (CTS)—that describes how types are defined and how they behave.

Note In fact, Microsoft has been submitting the CTS as well as other parts of the .NET Framework, including file formats, metadata, IL, and access to the underlying platform (P/Invoke) to ECMA for the purpose of standardization. The standard is called the Common Language Infrastructure (CLI) and is the ECMA-335 specification. In addition, Microsoft has also submitted portions of the FCL, the C# programming language (ECMA-334), and the C++/CLI programming language. For information about these industry standards, go to the ECMA website that pertains to Technical Committee 39: *http://www.ecma-international.org*. You can also refer to Microsoft's own website: *http://msdn.microsoft.com/en-us/netframework/aa569283.aspx*. In addition, Microsoft has applied their Community Promise to the ECMA-334 and ECMA-335 specifications. For more information about this, see *http://www.microsoft.com/openspecifications/en/us/programs/community-promise/default.aspx*.

The CTS specification states that a type can contain zero or more members. In Part II, "Designing Types," I'll cover all of these members in great detail. For now, I just want to give you a brief introduction to them:

- **Field** A data variable that is part of the object's state. Fields are identified by their name and type.
- **Method** A function that performs an operation on the object, often changing the object's state. Methods have a name, a signature, and modifiers. The signature specifies the number of parameters (and their sequence), the types of the parameters, whether a value is returned by the method, and if so, the type of the value returned by the method.
- **Property** To the caller, this member looks like a field. But to the type implementer, it looks like a method (or two). Properties allow an implementer to validate input parameters and object state before accessing the value and/or calculating a value only when necessary. They also allow a user of the type to have simplified syntax. Finally, properties allow you to create read-only or write-only "fields."
- **Event** An event allows a notification mechanism between an object and other interested objects. For example, a button could offer an event that notifies other objects when the button is clicked.

The CTS also specifies the rules for type visibility and access to the members of a type. For example, marking a type as *public* (called `public`) exports the type, making it visible and accessible to any assembly. On the other hand, marking a type as *assembly* (called `internal` in C#) makes the type visible and accessible to code within the same assembly only. Thus, the CTS establishes the rules by which assemblies form a boundary of visibility for a type, and the CLR enforces the visibility rules.

A type that is visible to a caller can further restrict the ability of the caller to access the type's members. The following list shows the valid options for controlling access to a member:

- **Private** The member is accessible only by other members in the same class type.
- **Family** The member is accessible by derived types, regardless of whether they are within the same assembly. Note that many languages (such as C++ and C#) refer to family as `protected`.
- **Family and assembly** The member is accessible by derived types, but only if the derived type is defined in the same assembly. Many languages (such as C# and Visual Basic) don't offer this access control. Of course, IL Assembly language makes it available.
- **Assembly** The member is accessible by any code in the same assembly. Many languages refer to *assembly* as `internal`.
- **Family or assembly** The member is accessible by derived types in any assembly. The member is also accessible by any types in the same assembly. C# refers to *family* or *assembly* as `protected internal`.
- **Public** The member is accessible by any code in any assembly.

In addition, the CTS defines the rules governing type inheritance, virtual methods, object lifetime, and so on. These rules have been designed to accommodate the semantics expressible in modern-day programming languages. In fact, you won't even need to learn the CTS rules per se because the language you choose will expose its own language syntax and type rules in the same way that you're familiar with today. And it will map the language-specific syntax into IL, the "language" of the CLR, when it emits the assembly during compilation.

When I first started working with the CLR, I soon realized that it is best to think of the language and the behavior of your code as two separate and distinct things. Using C++/CLI, you can define your own types with their own members. Of course, you could have used C# or Visual Basic to define the same type with the same members. Sure, the syntax you use for defining the type is different depending on the language you choose, but the behavior of the type will be identical regardless of the language because the CLR's CTS defines the behavior of the type.

To help clarify this idea, let me give you an example. The CTS allows a type to derive from only one base class. So, although the C++ language supports types that can inherit from multiple base types, the CTS can't accept and operate on any such type. To help the developer, Microsoft's C++/CLI compiler reports an error if it detects that you're attempting to create managed code that includes a type deriving from multiple base types.

Here's another CTS rule. All types must (ultimately) inherit from a predefined type: `System.Object`. As you can see, `Object` is the name of a type defined in the `System` namespace. This `Object` is the root of all other types and therefore guarantees that every type instance has a minimum set of behaviors. Specifically, the `System.Object` type allows you to do the following:

- Compare two instances for equality.
- Obtain a hash code for the instance.
- Query the true type of an instance.
- Perform a shallow (bitwise) copy of the instance.
- Obtain a string representation of the instance object's current state.

The Common Language Specification

COM allows objects created in different languages to communicate with one another. On the other hand, the CLR now integrates all languages and allows objects created in one language to be treated as equal citizens by code written in a completely different language. This integration is possible because of the CLR's standard set of types, metadata (self-describing type information), and common execution environment.

Although this language integration is a fantastic goal, the truth of the matter is that programming languages are very different from one another. For example, some languages don't treat symbols with case-sensitivity, and some don't offer unsigned integers, operator overloading, or methods to support a variable number of arguments.

If you intend to create types that are easily accessible from other programming languages, you need to use only features of your programming language that are guaranteed to be available in all other languages. To help you with this, Microsoft has defined a Common Language Specification (CLS) that details for compiler vendors the minimum set of features their compilers must support if these compilers are to generate types compatible with other components written by other CLS-compliant languages on top of the CLR.

The CLR/CTS supports a lot more features than the subset defined by the CLS, so if you don't care about interlanguage operability, you can develop very rich types limited only by the language's feature set. Specifically, the CLS defines rules that externally visible types and methods must adhere to if they are to be accessible from any CLS-compliant programming language. Note that the CLS rules don't apply to code that is accessible only within the defining assembly. Figure 1-6 summarizes the ideas expressed in this paragraph.

As Figure 1-6 shows, the CLR/CTS offers a set of features. Some languages expose a large subset of the CLR/CTS. A programmer willing to write in IL assembly language, for example, is able to use all of the features the CLR/CTS offers. Most other languages, such as C#, Visual Basic, and Fortran, expose a subset of the CLR/CTS features to the programmer. The CLS defines the minimum set of features that all languages must support.

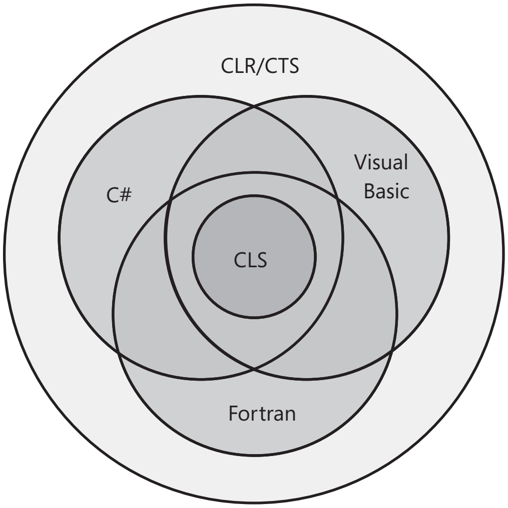

FIGURE 1-6 Languages offer a subset of the CLR/CTS and a superset of the CLS (but not necessarily the same superset).

If you're designing a type in one language, and you expect that type to be used by another language, you shouldn't take advantage of any features that are outside of the CLS in its public and protected members. Doing so would mean that your type's members might not be accessible by programmers writing code in other programming languages.

In the following code, a CLS-compliant type is being defined in C#. However, the type has a few non–CLS-compliant constructs causing the C# compiler to complain about the code.

```
using System;

// Tell compiler to check for CLS compliance
[assembly: CLSCompliant(true)]

namespace SomeLibrary {
   // Warnings appear because the class is public
   public sealed class SomeLibraryType {

      // Warning: Return type of 'SomeLibrary.SomeLibraryType.Abc()'
      // is not CLS-compliant
      public UInt32 Abc() { return 0; }

      // Warning: Identifier 'SomeLibrary.SomeLibraryType.abc()'
      // differing only in case is not CLS-compliant
      public void abc() { }

      // No warning: this method is private
      private UInt32 ABC() { return 0; }
   }
}
```

In this code, the `[assembly:CLSCompliant(true)]` attribute is applied to the assembly. This attribute tells the compiler to ensure that any publicly exposed type doesn't have any construct that would prevent the type from being accessed from any other programming language. When this code is compiled, the C# compiler emits two warnings. The first warning is reported because the method `Abc` returns an unsigned integer; some other programming languages can't manipulate unsigned integer values. The second warning is because this type exposes two public methods that differ only by case and return type: `Abc` and `abc`. Visual Basic and some other languages can't call both of these methods.

Interestingly, if you were to delete `public` from in front of `'sealed class SomeLibraryType'` and recompile, both warnings would go away. The reason is that the `SomeLibraryType` type would default to `internal` and would therefore no longer be exposed outside of the assembly. For a complete list of CLS rules, refer to the "Cross-Language Interoperability" section in the .NET Framework SDK documentation (*http://msdn.microsoft.com/en-us/library/730f1wy3.aspx*).

Let me distill the CLS rules to something very simple. In the CLR, every member of a type is either a field (data) or a method (behavior). This means that every programming language must be able to access fields and call methods. Certain fields and certain methods are used in special and common ways. To ease programming, languages typically offer additional abstractions to make coding these common programming patterns easier. For example, languages expose concepts such as enums, arrays, properties, indexers, delegates, events, constructors, finalizers, operator overloads, conversion operators, and so on. When a compiler comes across any of these things in your source code, it must translate these constructs into fields and methods so that the CLR and any other programming language can access the construct.

Consider the following type definition, which contains a constructor, a finalizer, some overloaded operators, a property, an indexer, and an event. Note that the code shown is there just to make the code compile; it doesn't show the correct way to implement a type.

```
using System;

internal sealed class Test {
   // Constructor
   public Test() {}

   // Finalizer
   ~Test() {}

   // Operator overload
   public static Boolean operator == (Test t1, Test t2) {
      return true;
   }
   public static Boolean operator != (Test t1, Test t2) {
      return false;
   }

   // An operator overload
   public static Test operator + (Test t1, Test t2) { return null; }
```

```
   // A property
   public String AProperty {
      get { return null; }
      set { }
   }

   // An indexer
   public String this[Int32 x] {
      get { return null; }
      set { }
   }

   // An event
   public event EventHandler AnEvent;
}
```

When the compiler compiles this code, the result is a type that has a number of fields and methods defined in it. You can easily see this by using the IL Disassembler tool (ILDasm.exe) provided with the .NET Framework SDK to examine the resulting managed module, which is shown in Figure 1-7.

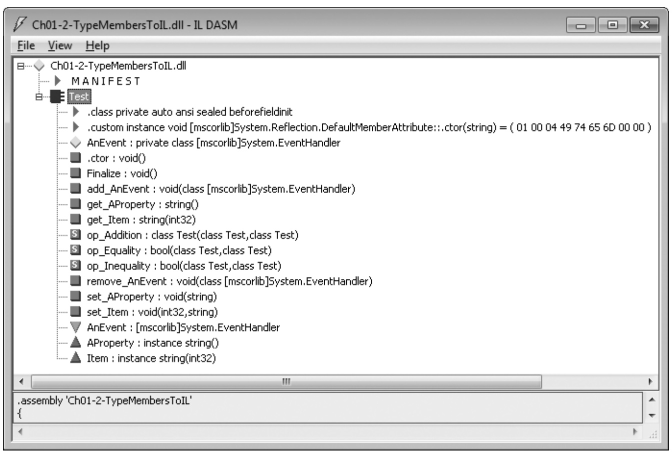

FIGURE 1-7 ILDasm showing Test type's fields and methods (obtained from metadata).

Table 1-4 shows how the programming language constructs got mapped to the equivalent CLR fields and methods.

TABLE 1-4 Test Type's Fields and Methods (Obtained from Metadata)

Type Member	Member Type	Equivalent Programming Language Construct
`AnEvent`	Field	Event; the name of the field is `AnEvent` and its type is System.EventHandler.
`.ctor`	Method	Constructor.
`Finalize`	Method	Finalizer.
`add_AnEvent`	Method	`Event add accessor` method.

Type Member	Member Type	Equivalent Programming Language Construct
get_AProperty	Method	Property get accessor method.
get_Item	Method	Indexer get accessor method.
op_Addition	Method	+ operator.
op_Equality	Method	== operator.
op_Inequality	Method	!= operator.
remove_AnEvent	Method	Event remove accessor method.
set_AProperty	Method	Property set accessor method.
set_Item	Method	Indexer set accessor method.

The additional nodes under the Test type that aren't mentioned in Table 1-4—.class, .custom, AnEvent, AProperty, and Item—identify additional metadata about the type. These nodes don't map to fields or methods; they just offer some additional information about the type that the CLR, programming languages, or tools can get access to. For example, a tool can see that the Test type offers an event, called AnEvent, which is exposed via the two methods (add_AnEvent and remove_AnEvent).

Interoperability with Unmanaged Code

The .NET Framework offers a ton of advantages over other development platforms. However, very few companies can afford to redesign and re-implement all of their existing code. Microsoft realizes this and has constructed the CLR so that it offers mechanisms that allow an application to consist of both managed and unmanaged parts. Specifically, the CLR supports three interoperability scenarios:

- **Managed code can call an unmanaged function in a DLL** Managed code can easily call functions contained in DLLs by using a mechanism called P/Invoke (for Platform Invoke). After all, many of the types defined in the FCL internally call functions exported from Kernel32.dll, User32.dll, and so on. Many programming languages will expose a mechanism that makes it easy for managed code to call out to unmanaged functions contained in DLLs. For example, a C# application can call the CreateSemaphore function exported from Kernel32.dll.

- **Managed code can use an existing COM component (server)** Many companies have already implemented a number of unmanaged COM components. Using the type library from these components, a managed assembly can be created that describes the COM component. Managed code can access the type in the managed assembly just as any other managed type. See the TlbImp.exe tool that ships with the .NET Framework SDK for more information. At times, you might not have a type library or you might want to have more control over what TlbImp.exe produces. In these cases, you can manually build a type in source code that the CLR can use to achieve the proper interoperability. For example, you could use DirectX COM components from a C# application.

- **Unmanaged code can use a managed type (server)** A lot of existing unmanaged code requires that you supply a COM component for the code to work correctly. It's much easier to implement these components by using managed code so that you can avoid all of the code having to do with reference counting and interfaces. For example, you could create an ActiveX control or a shell extension in C#. See the TlbExp.exe and RegAsm.exe tools that ship with the .NET Framework SDK for more information.

> **Note** Microsoft now makes available the source code for the Type Library Importer tool and a P/Invoke Interop Assistant tool to help developers needing to interact with native code. These tools and their source code can be downloaded from *http://CLRInterop.CodePlex.com/*.

With Windows 8, Microsoft has introduced a new Windows API called the Windows Runtime (WinRT). This API is implemented internally via COM components. But, instead of using type library files, the COM components describe their API via the metadata ECMA standard created by the .NET Framework team. The beauty of this is that code written via a .NET language can (for the most part) seamlessly communicate with WinRT APIs. Underneath the covers, the CLR is performing all of the COM interop for you without you having to use any additional tools at all—it just works! Chapter 25, "Interoperating with WinRT Components" goes into all the details.