PART I
The C# Language

1 .NET Architecture

WHAT'S IN THIS CHAPTER?

- Compiling and running code that targets .NET
- Advantages of Microsoft Intermediate Language (MSIL)
- Value and reference types
- Data typing
- Understanding error handling and attributes
- Assemblies, .NET base classes, and namespaces

CODE DOWNLOADS FOR THIS CHAPTER

There are no code downloads for this chapter.

THE RELATIONSHIP OF C# TO .NET

This book emphasizes that the C# language must be considered in parallel with the .NET Framework, rather than viewed in isolation. The C# compiler specifically targets .NET, which means that all code written in C# always runs using the .NET Framework. This has two important consequences for the C# language:

1. The architecture and methodologies of C# reflect the underlying methodologies of .NET.
2. In many cases, specific language features of C# actually depend on features of .NET or of the .NET base classes.

Because of this dependence, you must gain some understanding of the architecture and methodology of .NET before you begin C# programming, which is the purpose of this chapter.

C# is a programming language newly designed for .NET and is significant in two respects:

- It is specifically designed and targeted for use with Microsoft's .NET Framework (a feature-rich platform for the development, deployment, and execution of distributed applications).
- It is a language based on the modern object-oriented design methodology, and when designing it Microsoft learned from the experience of all the other similar languages that have been around since object-oriented principles came to prominence 20 years ago.

C# is a language in its own right. Although it is designed to generate code that targets the .NET environment, it is not part of .NET. Some features are supported by .NET but not by C#, and you might be surprised to learn that some features of the C# language are not supported by .NET or by MSIL (for example, some instances of operator overloading).

However, because the C# language is intended for use with .NET, you must understand this Framework if you want to develop applications in C# effectively. Therefore, this chapter takes some time to peek underneath the surface of .NET.

THE COMMON LANGUAGE RUNTIME

Central to the .NET Framework is its runtime execution environment, known as the *Common Language Runtime* (*CLR*) or the *.NET runtime*. Code running under the control of the CLR is often termed *managed code*.

However, before it can be executed by the CLR, any source code that you develop (in C# or some other language) needs to be compiled. Compilation occurs in two steps in .NET:

1. Compilation of source code to Microsoft Intermediate Language (IL).
2. Compilation of IL to platform-specific code by the CLR.

This two-stage compilation process is important because the existence of the Microsoft Intermediate Language is the key to providing many of the benefits of .NET.

IL shares with Java byte code the idea that it is a low-level language with a simple syntax (based on numeric codes rather than text), which can be quickly translated into native machine code. Having this well-defined universal syntax for code has significant advantages: platform independence, performance improvement, and language interoperability.

Platform Independence

First, platform independence means that the same file containing byte code instructions can be placed on any platform; at runtime, the final stage of compilation can then be easily accomplished so that the code can run on that particular platform. In other words, by compiling to IL you obtain platform independence for .NET in much the same way as compiling to Java byte code gives Java platform independence.

The platform independence of .NET is only theoretical at present because a complete implementation of .NET is available only for Windows. However, a partial, cross-platform implementation is available (see, for example, the Mono project, an effort to create an open source implementation of .NET, at `www.go-mono.com`). You can also use C# on iPhone and Android devices by using tools and libraries from Xamarin (`www.xamarin.com`).

Performance Improvement

Although previously compared to Java, IL is actually a bit more ambitious than Java byte code. IL is always *Just-in-Time* compiled (known as *JIT* compilation), whereas Java byte code was often interpreted. One of the disadvantages of Java was that, on execution, the process to translate from Java byte code to native executable resulted in a loss of performance (with the exception of more recent cases in which Java is JIT compiled on certain platforms).

Instead of compiling the entire application at one time (which could lead to a slow startup time), the JIT compiler simply compiles each portion of code as it is called (just in time). When code has been compiled once, the resultant native executable is stored until the application exits so that it does not need to be recompiled the next time that portion of code is run. Microsoft argues that this process is more efficient than compiling the entire application code at the start because of the likelihood that large portions of any application code will not actually be executed in any given run. Using the JIT compiler, such code can never be compiled.

This explains why you can expect that execution of managed IL code will be almost as fast as executing native machine code. What it does not explain is why Microsoft expects that you get a performance *improvement*. The reason given for this is that because the final stage of compilation takes place at runtime, the JIT compiler knows exactly what processor type the program runs on. This means that it can optimize the final executable code to take advantage of any features or particular machine code instructions offered by that particular processor.

Traditional compilers optimize the code, but they can perform optimizations that are only independent of the particular processor that the code runs on. This is because traditional compilers compile to native executable code before the software is shipped. This means that the compiler does not know what type of processor the code runs on beyond basic generalities, such as that it is an x86-compatible processor or an Alpha processor.

Language Interoperability

The use of IL not only enables platform independence, but it also facilitates *language interoperability*. Simply put, you can compile to IL from one language, and this compiled code should then be interoperable with code that has been compiled to IL from another language.

You are probably now wondering which languages aside from C# are interoperable with .NET. The following sections briefly discuss how some of the other common languages fit into .NET.

Visual Basic 2013

Visual Basic .NET 2002 underwent a complete revamp from Visual Basic 6 to bring it up to date with the first version of the .NET Framework. The Visual Basic language had dramatically evolved from VB6, which meant that VB6 was not a suitable language to run .NET programs. For example, VB6 is heavily integrated into Component Object Model (COM) and works by exposing only event handlers as source code to the developer — most of the background code is not available as source code. Not only that, it does not support implementation inheritance, and the standard data types that Visual Basic 6 uses are incompatible with .NET.

Visual Basic 6 was upgraded to Visual Basic .NET in 2002, and the changes that were made to the language are so extensive you might as well regard Visual Basic as a new language.

Since then, Visual Basic evolved with many language enhancements — much as C# did. Visual Basic and C# are very similar in their features because they are developed from the same product team within Microsoft. As time has passed, Visual Basic acquired features that were once only available with C#, and C# acquired features that were once only available with Visual Basic. Nowadays there are only minor differences between Visual Basic and C#, and mainly it's a matter of taste to use curly brackets or `END` statements.

Visual C++ 2013

Visual C++ 6 already had a large number of Microsoft-specific extensions on Windows. With Visual C++ .NET, extensions have been added to support the .NET Framework. This means that existing C++ source code will continue to compile to native executable code without modification. It also means, however, that it will run independently of the .NET runtime. If you want your C++ code to run within the .NET Framework, you can simply add the following line to the beginning of your code:

```
#using <mscorlib.dll>
```

You can also pass the flag `/clr` to the compiler, which then assumes that you want to compile to managed code and will hence emit IL instead of native machine code. The interesting thing about C++ is that when you compile to managed code, the compiler can emit IL that contains an embedded native executable. This means that you can mix managed types and unmanaged types in your C++ code. Thus, the managed C++ code,

```
class MyClass
{
```

defines a plain C++ class, whereas the code,

```
ref class MyClass
{
```

gives you a managed class, just as if you had written the class in C# or Visual Basic 2013. The advantage to use managed C++ over C# code is that you can call unmanaged C++ classes from managed C++ code without resorting to COM interop.

The compiler raises an error if you attempt to use features not supported by .NET on managed types (for example, templates or multiple inheritances of classes). You can also find that you need to use nonstandard C++ features when using managed classes.

Writing C++ programs that use .NET gives you different variants of interop scenarios. With the compiler setting /clr for Common Language Runtime Support, you can completely mix all native and managed C++ features. Other options such as `/clr:safe` and `/clr:pure` restrict the use of native C++ pointers and thus enable writing safe code like with C# and Visual Basic.

Visual C++ 2013 enables you to create programs for the Windows Runtime (WinRT) with Windows 8.1. This way C++ does not use managed code but instead accesses the WinRT natively.

Visual F#

F# is a strongly typed functional programming language. This language has strong support inside Visual Studio.

Being a functional programming language it looks very different from C#. For example, declaring a type `Person` with the members `FirstName` and `LastName` looks like this:

```
module PersonSample

type Person(firstName : string, lastName : string) =
    member this.FirstName = firstName
    member this.LastName = lastName
```

And using the `Person` type makes use of the `let` keyword. `printfn` writes results to the console:

```
open PersonSample

[<EntryPoint>]
let main argv =
    let p = Person("Sebastian", "Vettel")
    let first = p.firstName
    let last = p.lastName
    printfn "%s %s" first last
    0 // return an integer exit code
```

F# can use all the types you create with C#, and vice versa. The advantage of F# is that it is a functional language instead of object-oriented, and this helps with programming of complex algorithms — for example, for financial and scientific applications.

COM and COM+

Technically speaking, COM and COM+ are not technologies targeted at .NET — components based on them cannot be compiled into IL. (Although you can do so to some degree using managed C++ if the original COM component were written in C++). However, COM+ remains an important tool because its features are not duplicated in .NET. Also, COM components can still work — and .NET incorporates COM interoperability features that make it possible for managed code to call up COM components and vice versa (discussed in Chapter 23, “Interop”). In general, you will probably find it more convenient for most purposes to code new components as .NET components so that you can take advantage of the .NET base classes and the other benefits of running as managed code.

Windows Runtime

Windows 8 offers a new runtime used by Windows Store apps. Some parts of this runtime can be used with desktop applications as well. You can use this runtime from Visual Basic, C#, C++, and JavaScript. When using the runtime with these different environments, it looks different. Using it from C# it looks like classes from the .NET Framework. Using it from JavaScript it looks like what JavaScript developers are used to with JavaScript libraries. And using it from C++, methods looks like the Standard C++ Library. This is done by using language projection. The Windows Runtime and how it looks from C# is discussed in Chapter 31, "Windows Runtime."

A CLOSER LOOK AT INTERMEDIATE LANGUAGE

From what you learned in the previous section, Microsoft Intermediate Language obviously plays a fundamental role in the .NET Framework. It makes sense now to take a closer look at the main features of IL because any language that targets .NET logically needs to support these characteristics.

Here are the important features of IL:

- Object orientation and the use of interfaces
- Strong distinction between value and reference types
- Strong data typing
- Error handling using exceptions
- Use of attributes

The following sections explore each of these features.

Support for Object Orientation and Interfaces

The language independence of .NET does have some practical limitations. IL is inevitably going to implement some particular programming methodology, which means that languages targeting it need to be compatible with that methodology. The particular route that Microsoft has chosen to follow for IL is that of classic object-oriented programming, with single implementation inheritance of classes.

In addition to classic object-oriented programming, IL also brings in the idea of interfaces, which saw their first implementation under Windows with COM. Interfaces built using .NET produce interfaces that are not the same as COM interfaces. They do not need to support any of the COM infrastructure. (For example, they are not derived from `IUnknown` and do not have associated globally unique identifiers, more commonly known as GUIDs.) However, they do share with COM interfaces the idea that they provide a contract, and classes that implement a given interface must provide implementations of the methods and properties specified by that interface.

You have now seen that working with .NET means compiling to IL, and that in turn means that you need to understand traditional object-oriented methodologies. However, that alone is not sufficient to give you language interoperability. After all, C++ and Java both use the same object-oriented paradigms but are still not regarded as interoperable. You need to look a little more closely at the concept of language interoperability.

So what exactly is language interoperability?

After all, COM enabled components written in different languages to work together in the sense of calling each other's methods. What was inadequate about that? COM, by virtue of being a binary standard, did enable components to instantiate other components and call methods or properties against them, without worrying about the language in which the respective components were written. To achieve this, however, each object had to be instantiated through the COM runtime and accessed through an interface. Depending on the threading models of the relative components, there may have been large performance losses associated with marshaling data between apartments or running components or both on different threads. In the extreme case of components hosted as an executable rather than DLL files, separate processes would need to be created to run them. The emphasis was very much that components could talk to each other but only via the COM runtime. In no way with COM did components written in different languages directly communicate with each other, or instantiate instances of each other — it was always done with COM as an intermediary. Not only that, but the COM architecture did not permit implementation inheritance, which meant that it lost many of the advantages of object-oriented programming.

An associated problem was that, when debugging, you would still need to debug components written in different languages independently. It was not possible to step between languages in the debugger. Therefore, what you *actually* mean by language interoperability is that classes written in one language should talk directly to classes written in another language. In particular:

- A class written in one language can inherit from a class written in another language.
- The class can contain an instance of another class, no matter what the languages of the two classes are.
- An object can directly call methods against another object written in another language.
- Objects (or references to objects) can be passed around between methods.
- When calling methods between languages, you can step between the method calls in the debugger, even when this means stepping between source code written in different languages.

This is all quite an ambitious aim, but amazingly .NET and IL have achieved it. In the case of stepping between methods in the debugger, this facility is actually offered by the Visual Studio integrated development environment (IDE) rather than by the CLR.

Distinct Value and Reference Types

As with any programming language, IL provides a number of predefined primitive data types. One characteristic of IL, however, is that it makes a strong distinction between value and reference types. *Value types* are those for which a variable directly stores its data, whereas *reference types* are those for which a variable simply stores the address at which the corresponding data can be found.

In C++ terms, using reference types is similar to accessing a variable through a pointer, whereas for Visual Basic the best analogy for reference types are objects, which in Visual Basic 6 are always accessed through references. IL also lays down specifications about data storage: Instances of reference types are always stored in an area of memory known as the *managed heap*, whereas value types are normally stored on the *stack*. (Although if value types are declared as fields within reference types, they will be stored inline on the heap.) Chapter 2, “Core C#,” discusses the stack and the managed heap and how they work.

Strong Data Typing

One important aspect of IL is that it is based on exceptionally *strong data typing*. That means that all variables are clearly marked as being of a particular, specific data type. (There is no room in IL, for example, for the `Variant` data type recognized by Visual Basic and scripting languages.) In particular, IL does not normally permit any operations that result in ambiguous data types.

For instance, Visual Basic 6 developers are used to passing variables around without worrying too much about their types because Visual Basic 6 automatically performs type conversion. C++ developers are used to routinely casting pointers between different types. Performing this kind of operation can be great for performance, but it breaks type safety. Hence, it is permitted only under certain circumstances in some of the languages that compile to managed code. Indeed, pointers (as opposed to references) are permitted only in marked blocks of code in C#, and not at all in Visual Basic. (Although they are allowed in managed C++.) Using pointers in your code causes it to fail the memory type-safety checks performed by the CLR. Some languages compatible with .NET, such as Visual Basic 2010, still allow some laxity in typing but only because the compilers behind the scenes ensure that the type safety is enforced in the emitted IL.

Although enforcing type safety might initially appear to hurt performance, in many cases the benefits gained from the services provided by .NET that rely on type safety far outweigh this performance loss. Such services include the following:

- Language interoperability
- Garbage collection
- Security
- Application domains

The following sections take a closer look at why strong data typing is particularly important for these features of .NET.

Strong Data Typing as a Key to Language Interoperability

If a class is to derive from or contains instances of other classes, it needs to know about all the data types used by the other classes. This is why strong data typing is so important. Indeed, it is the absence of any agreed-on system for specifying this information in the past that has always been the real barrier to inheritance and interoperability across languages. This kind of information is simply not present in a standard executable file or DLL.

Suppose that one of the methods of a Visual Basic 2013 class is defined to return an `Integer` — one of the standard data types available in Visual Basic 2013. C# simply does not have any data type of that name. Clearly, you can derive from the class, use this method, and use the return type from C# code only if the compiler knows how to map Visual Basic 2013's `Integer` type to some known type defined in C#. So, how is this problem circumvented in .NET?

Common Type System

This data type problem is solved in .NET using the *Common Type System* (*CTS*). The CTS defines the predefined data types available in IL so that all languages that target the .NET Framework can produce compiled code ultimately based on these types.

For the previous example, Visual Basic 2013's `Integer` is actually a 32-bit signed integer, which maps exactly to the IL type known as `Int32`. Therefore, this is the data type specified in the IL code. Because the C# compiler is aware of this type, there is no problem. At source code-level, C# refers to `Int32` with the keyword `int`, so the compiler simply treats the Visual Basic 2013 method as if it returned an `int`.

The CTS does not specify merely primitive data types but a rich hierarchy of types, which includes well-defined points in the hierarchy at which code is permitted to define its own types. The hierarchical structure of the CTS reflects the single-inheritance object-oriented methodology of IL, and resembles Figure 1-1.

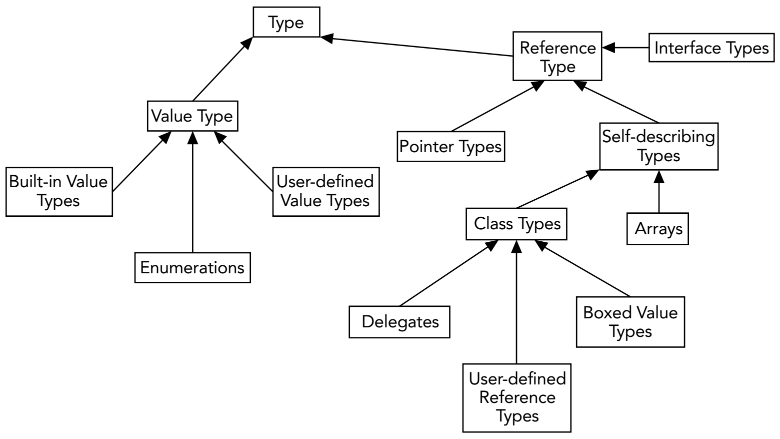

FIGURE 1-1

All of the built-in value types aren't here because they are covered in detail in Chapter 3, "Objects and Types." In C#, each predefined type is recognized by the compiler maps onto one of the IL built-in types. The same is true in Visual Basic 2013.

Common Language Specification

The *Common Language Specification* (*CLS*) works with the CTS to ensure language interoperability. The CLS is a set of minimum standards that all compilers targeting .NET must support. Because IL is a rich language, writers of most compilers prefer to restrict the capabilities of a given compiler to support only a subset of the facilities offered by IL and the CTS. That is fine as long as the compiler supports everything defined in the CLS.

For example, take case sensitivity. IL is case-sensitive. Developers who work with case-sensitive languages regularly take advantage of the flexibility that this case sensitivity gives them when selecting variable names. Visual Basic 2013, however, is not case-sensitive. The CLS works around this by indicating that CLS-compliant code should not expose any two names that differ only in their case. Therefore, Visual Basic 2013 code can work with CLS-compliant code.

This example shows that the CLS works in two ways:

1. Individual compilers do not need to be powerful enough to support the full features of .NET — this should encourage the development of compilers for other programming languages that target .NET.
2. If you restrict your classes to exposing only CLS-compliant features, then it guarantees that code written in any other compliant language can use your classes.

The beauty of this idea is that the restriction to using CLS-compliant features applies only to public and protected members of classes and public classes. Within the private implementations of your classes, you can write whatever non-CLS code you want because code in other assemblies (units of managed code; see later in the section Assemblies) cannot access this part of your code.

Without going into the details of the CLS specifications here, in general, the CLS does not affect your C# code much because of the few non-CLS-compliant features of C#.

NOTE *It is perfectly acceptable to write non-CLS-compliant code. However, if you do, the compiled IL code is not guaranteed to be fully language interoperable. You can mark your assembly or types to be CLS-compliant by applying the* `CLSCompliant` *attribute. Using this attribute, the compiler checks for compliance. If a non-compliant data type is used with the signature of a public method (for example,* `uint`*), you will get a compiler warning that the method is not CLS compliant.*

Garbage Collection

The *garbage collector* is .NET's answer to memory management and in particular to the question of what to do about reclaiming memory that running applications ask for. Up until now, two techniques have been used on the Windows platform for de-allocating memory that processes have dynamically requested from the system:

- Make the application code do it all manually.
- Make objects maintain reference counts.

Having the application code responsible for de-allocating memory is the technique used by lower-level, high-performance languages such as C++. It is efficient and has the advantage that (in general) resources are never occupied for longer than necessary. The big disadvantage, however, is the frequency of bugs. Code that requests memory also should explicitly inform the system when it no longer requires that memory. However, it is easy to overlook this, resulting in memory leaks.

Although modern developer environments do provide tools to assist in detecting memory leaks, they remain difficult bugs to track down. That's because they have no effect until so much memory has been leaked that Windows refuses to grant any more to the process. By this point, the entire computer may have appreciably slowed down due to the memory demands made on it.

Maintaining reference counts is favored in COM. The idea is that each COM component maintains a count of how many clients are currently maintaining references to it. When this count falls to zero, the component can destroy itself and free up associated memory and resources. The problem with this is that it still relies on the good behavior of clients to notify the component that they have finished with it. It takes only one client not to do so, and the object sits in memory. In some ways, this is a potentially more serious problem than a simple C++-style memory leak because the COM object may exist in its own process, which means that it can never be removed by the system. (At least with C++ memory leaks, the system can reclaim all memory when the process terminates.)

The .NET runtime relies on the garbage collector instead. The purpose of this program is to clean up memory. The idea is that all dynamically requested memory is allocated on the heap. (That is true for all languages; although in the case of .NET, the CLR maintains its own managed heap for .NET applications to use.) Sometimes, when .NET detects that the managed heap for a given process is becoming full and therefore needs tidying up, it calls the garbage collector. The garbage collector runs through variables currently in scope in your code, examining references to objects stored on the heap to identify which ones are accessible from your code — that is, which objects have references that refer to them. Any objects not referred to are deemed to be no longer accessible from your code and can therefore be removed. Java uses a system of garbage collection similar to this.

Garbage collection works in .NET because IL has been designed to facilitate the process. The principle requires that you cannot get references to existing objects other than by copying existing references and that IL is type safe. In this context, if any reference to an object exists, there is sufficient information in the reference to exactly determine the type of the object.

The garbage collection mechanism cannot be used with a language such as unmanaged C++, for example, because C++ enables pointers to be freely cast between types.

One important aspect of garbage collection is that it is not deterministic. In other words, you cannot guarantee when the garbage collector will be called. It will be called when the CLR decides that it is needed; though you can override this process and call up the garbage collector in your code. Calling the garbage collector in your code is good for testing purposes, but you shouldn't do this in a normal program.

Look at Chapter 14, "Memory Management and Pointers," for more information on the garbage collection process.

Security

.NET can excel in terms of complementing the security mechanisms provided by Windows because it can offer Code Access Security, whereas Windows offers only role-based security.

Role-based security is based on the identity of the account under which the process runs (that is, who owns and runs the process). *Code Access Security*, by contrast, is based on what the code actually does and on how much the code is trusted. Because of the strong type safety of IL, the CLR can inspect code before running it to determine required security permissions. .NET also offers a mechanism by which code can indicate in advance what security permissions it requires to run.

The importance of code-based security is that it reduces the risks associated with running code of dubious origin (such as code that you have downloaded from the Internet). For example, even if code runs under the administrator account, you can use code-based security to indicate that the code should still not be permitted to perform certain types of operations that the administrator account would normally be allowed to do, such as read or write to environment variables, read or write to the registry, or access the .NET reflection features.

NOTE *Security issues are covered in more depth in Chapter 22, "Security."*

Application Domains

Application domains are an important innovation in .NET and are designed to ease the overhead involved when running applications that need to be isolated from each other, but also need to communicate with each other. The classic example of this is a web server application, which may be simultaneously responding to a number of browser requests. It can, therefore, probably have a number of instances of the component responsible for servicing those requests running simultaneously.

In pre-.NET days, the choice would be between allowing those instances to share a process (with the resultant risk of a problem in one running instance bringing the whole website down) or isolating those instances in separate processes (with the associated performance overhead). Before .NET, isolation of code was only possible by using different processes. When you start a new application, it runs within the context of a process. Windows isolates processes from each other through address spaces. The idea is that each process has 4GB of virtual memory available in which to store its data and executable code (4GB is for 32-bit systems; 64-bit systems use more memory). Windows imposes an extra level of indirection by which this virtual memory maps into a particular area of actual physical memory or disk space. Each process gets a different mapping, with no overlap between the actual physical memories that the blocks of virtual address space map to (see Figure 1-2).

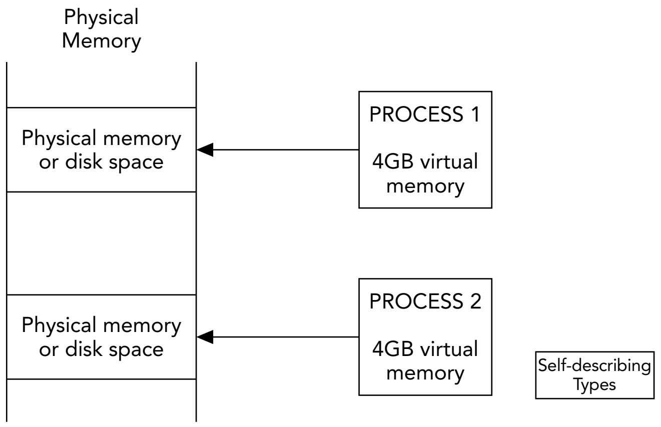

FIGURE 1-2

In general, any process can access memory only by specifying an address in virtual memory — processes do not have direct access to physical memory. Hence, it is simply impossible for one process to access the memory allocated to another process. This provides an excellent guarantee that any badly behaved code cannot damage anything outside of its own address space.

Processes do not just serve as a way to isolate instances of running code from each other; they also form the unit to which security privileges and permissions are assigned. Each process has its own security token, which indicates to Windows precisely what operations that process is permitted to do.

Although processes are great for security reasons, their big disadvantage is in the area of performance. Often, a number of processes can actually work together, and therefore need to communicate with each other. The obvious example of this is where a process calls up a COM component, which is an executable and therefore is required to run in its own process. The same thing happens in COM when surrogates are used. Because processes cannot share any memory, a complex marshaling process must be used to copy data between the processes. This results in a significant performance hit. If you need components to work together and do not want that performance hit, you must use DLL-based components and have everything running in the same address space — with the associated risk that a badly behaved component can bring everything else down.

Application domains are designed as a way to separate components without resulting in the performance problems associated with passing data between processes. The idea is that any one process is divided into a number of application domains. Each application domain roughly corresponds to a single application, and each thread of execution can run in a particular application domain (see Figure 1-3).

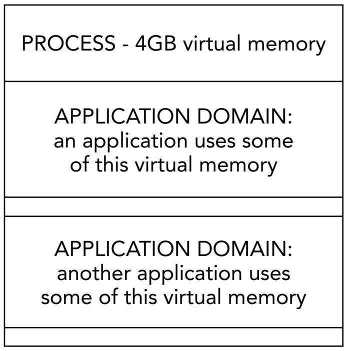
PROCESS - 4GB virtual memory
APPLICATION DOMAIN: an application uses some of this virtual memory
APPLICATION DOMAIN: another application uses some of this virtual memory

FIGURE 1-3

If different executables run in the same process space, then they clearly can easily share data because theoretically they can directly see each other's data. However, although this is possible in principle, the CLR makes sure that this does not happen in practice by inspecting the code for each running application to ensure that the code cannot stray outside of its own data areas. This looks, at first, like an almost impossible task to pull off — after all, how can you tell what the program is going to do without actually running it?

It is usually possible to do this because of the strong type safety of the IL. In most cases, unless code uses unsafe features such as pointers, the data types it uses ensures that memory is not accessed inappropriately. For example, .NET array types perform bounds checking to ensure that no out-of-bounds array operations are permitted. If a running application does need to communicate or share data with other applications running in different application domains, it must do so by calling on .NET's remoting services.

Code that has been verified to check that it cannot access data outside its application domain (other than through the explicit remoting mechanism) is *memory type safe*. Such code can safely be run alongside other type-safe code in different application domains within the same process.

Error Handling with Exceptions

The .NET Framework is designed to facilitate handling of error conditions using the same mechanism based on exceptions that is employed by Java and C++. C++ developers should note that because of IL's stronger typing system, there is no performance penalty associated with the use of exceptions with IL in the way that there is in C++. Also, the `finally` block, which has long been on many C++ developers' wish lists, is supported by .NET and by C#.

Exceptions are covered in detail in Chapter 16, "Errors and Exceptions." Briefly, the idea is that certain areas of code are designated as exception handler routines, with each one dealing with a particular error condition (for example, a file not being found, or being denied permission to perform some operation). These conditions can be defined as narrowly or as widely as you want. The exception architecture ensures that when an error condition occurs, execution can immediately jump to the exception handler routine that is most specifically geared to handle the exception condition in question.

The architecture of exception handling also provides a convenient means to pass an object containing precise details of the exception condition to an exception-handling routine. This object might include an appropriate message for the user and details of exactly where in the code the exception was detected.

Most exception-handling architecture, including the control of program flow when an exception occurs, is handled by the high-level languages (C#, Visual Basic 2013, C++), and is not supported by any special IL commands. C#, for example, handles exceptions using `try{}`, `catch{}`, and `finally{}` blocks of code. (For more details, see Chapter 16.)

What .NET does do, however, is provide the infrastructure to enable compilers that target .NET to support exception handling. In particular, it provides a set of .NET classes that can represent the exceptions and the language interoperability to enable the thrown exception objects to be interpreted by the exception-handling code, regardless of what language the exception-handling code is written in. This language independence is absent from both the C++ and Java implementations of exception handling; although it is present to a limited extent in the COM mechanism for handling errors, which involves returning error codes from methods and passing error objects around. Because exceptions are handled consistently in different languages is a crucial aspect of facilitating multi-language development.

Use of Attributes

Attributes are familiar to developers who use C++ to write COM components (through their use in Microsoft's COM Interface Definition Language [IDL]). The initial idea of an attribute was that it provided extra information concerning some item in the program that could be used by the compiler.

Attributes are supported in .NET — and now by C++, C#, and Visual Basic 2013. What is, however, particularly innovative about attributes in .NET is that you can define your own custom attributes in your source code. These user-defined attributes will be placed with the metadata for the corresponding data types or methods. This can be useful for documentation purposes, in which they can be used with reflection technology to perform programming tasks based on attributes. In addition, in common with the .NET philosophy of language independence, attributes can be defined in source code in one language and read by code written in another language.

> **NOTE** *Chapter 15, "Reflection," covers attributes.*

ASSEMBLIES

An *assembly* is the logical unit that contains compiled code targeted at the .NET Framework. This chapter doesn't cover assemblies in detail because they are covered thoroughly in Chapter 19, "Assemblies," but following are the main points.

An assembly is completely self-describing and is a logical rather than a physical unit, which means that it can be stored across more than one file. (Indeed, dynamic assemblies are stored in memory, not on file.) If an assembly is stored in more than one file, there will be one main file that contains the entry point and describes the other files in the assembly.

The same assembly structure is used for both executable code and library code. The only difference is that an executable assembly contains a main program entry point, whereas a library assembly does not.

An important characteristic of assemblies is that they contain metadata that describes the types and methods defined in the corresponding code. An assembly, however, also contains assembly metadata that describes the assembly. This assembly metadata, contained in an area known as the *manifest*, enables checks to be made on the version of the assembly and on its integrity.

> **NOTE** `ildasm`, *a Windows-based utility, can be used to inspect the contents of an assembly, including the manifest and metadata.* `ildasm` *is discussed in Chapter 19.*

Because an assembly contains program metadata means that applications or other assemblies that call up code in a given assembly do not need to refer to the registry, or to any other data source, to find out how to use that assembly. This is a significant break from the old COM way to do things, in which the GUIDs of the components and interfaces had to be obtained from the registry, and in some cases, the details of the methods and properties exposed would need to be read from a type library.

Having data spread out in up to three different locations meant there was the obvious risk of something getting out of synchronization, which would prevent other software from using the component successfully. With assemblies, there is no risk of this happening because all the metadata is stored with the program executable instructions. Even though assemblies are stored across several files, there are still no problems with data going out of synchronization. This is because the file that contains the assembly entry point also stores details of, and a hash of, the contents of the other files, which means that if one of the files is replaced, or in any way tampered with, this will almost certainly be detected and the assembly will refuse to load.

Assemblies come in two types: *private* and *shared* assemblies.

Private Assemblies

Private assemblies are the simplest type. They normally ship with software and are intended to be used only with that software. The usual scenario in which you ship private assemblies is when you supply an application in the form of an executable and a number of libraries, where the libraries contain code that should be used only with that application.

The system guarantees that private assemblies will not be used by other software because an application may load only private assemblies located in the same folder that the main executable is loaded in, or in a subfolder of it.

Because you would normally expect that commercial software would always be installed in its own directory, there is no risk of one software package overwriting, modifying, or accidentally loading private assemblies intended for another package. And, because private assemblies can be used only by the software package that they are intended for, you have much more control over what software uses them. There is, therefore, less need to take security precautions because there is no risk, for example, of some other commercial software overwriting one of your assemblies with some new version of it (apart from software designed specifically to perform malicious damage). There are also no problems with name collisions. If classes in your private assembly happen to have the same name as classes in someone else's private assembly, that does not matter because any given application can see only the one set of private assemblies.

Because a private assembly is entirely self-contained, the process to deploy it is simple. You simply place the appropriate file(s) in the appropriate folder in the file system. (No registry entries need to be made.) This process is known as *zero impact (xcopy) installation*.

Shared Assemblies

Shared assemblies are intended to be common libraries that any other application can use. Because any other software can access a shared assembly, more precautions need to be taken against the following risks:

- Name collisions, where another company's shared assembly implements types that have the same names as those in your shared assembly. Because client code can theoretically have access to both assemblies simultaneously, this could be a serious problem.
- The risk of an assembly being overwritten by a different version of the same assembly — the new version is incompatible with some existing client code.

The solution to these problems is placing shared assemblies in a special directory subtree in the file system, known as the *global assembly cache* (*GAC*). Unlike with private assemblies, this cannot be done by simply copying the assembly into the appropriate folder; it must be specifically installed into the cache. This process can be performed by a number of .NET utilities and requires certain checks on the assembly, as well as setting up of a small folder hierarchy within the assembly cache used to ensure assembly integrity.

To prevent name collisions, shared assemblies are given a name based on private key cryptography. (Private assemblies are simply given the same name as their main filename.) This name is known as a *strong name*; it is guaranteed to be unique and must be quoted by applications that reference a shared assembly.

Problems associated with the risk of overwriting an assembly are addressed by specifying version information in the assembly manifest and by allowing side-by-side installations.

Reflection

Because assemblies store metadata, including details of all the types and members of these types defined in the assembly, you can access this metadata programmatically. Full details of this are given in Chapter 15. This technique, known as reflection, raises interesting possibilities because it means that managed code can actually examine other managed code, and can even examine itself, to determine information about that code. This is most commonly used to obtain the details of attributes; although you can also use reflection, among other purposes, as an indirect way to instantiate classes or calling methods, given the names of those classes or methods as strings. In this way, you could select classes to instantiate methods to call at runtime, rather than at compile time, based on user input (dynamic binding).

Parallel Programming

The .NET Framework enables you to take advantage of all the multicore processors available today. The parallel computing capabilities provide the means to separate work actions and run these across multiple processors. The parallel programming APIs available now make writing safe multithreaded code simple; though you must realize that you still need to account for race conditions and things such as deadlocks.

The new parallel programming capabilities provide a new Task Parallel Library and a PLINQ Execution Engine. Chapter 21, "Tasks, Threads, and Synchronization," covers parallel programming.

Asynchronous Programming

Based on the `Task` from the Task Parallel Library are the new async features of C# 5. Since .NET 1.0, many classes from the .NET Framework offered asynchronous methods besides the synchronous variant. The user interface thread should not be blocked when doing a task that takes a while. You've probably seen several programs that have become unresponsive, which is annoying. A problem with the asynchronous methods was that they were difficult to use. The synchronous variant was a lot easier to program with, and thus this one was usually used.

Using the mouse the user is — with many years of experience — used to a delay. When moving objects or just using the scrollbar, a delay is normal. With new touch interfaces, if there's a delay the experience for the user can be extremely annoying. This can be solved by calling asynchronous methods. If a method with the WinRT might take more than 50 milliseconds, the WinRT offers only asynchronous method calls.

C# 5 now makes it easy to invoke new asynchronous methods. C# 5 defines two new keywords: `async` and `await`. These keywords and how they are used are discussed in Chapter 13, "Asynchronous Programming."

.NET FRAMEWORK CLASSES

Perhaps one of the biggest benefits to writing managed code, at least from a developer's point of view, is that you can use the .NET *base class library*. The .NET base classes are a massive collection of managed code classes that enable you to do almost any of the tasks that were previously available through the Windows API. These classes follow the same object model that IL uses, based on single inheritance. This means that you can either instantiate objects of whichever .NET base class is appropriate or derive your own classes from them.

The great thing about the .NET base classes is that they have been designed to be intuitive and easy to use. For example, to start a thread, you call the `Start()`method of the `Thread` class. To disable a `TextBox`, you set the `Enabled` property of a `TextBox` object to `false`. This approach — though familiar to Visual Basic and Java developers whose respective libraries are just as easy to use — will be a welcome relief to C++ developers, who for years have had to cope with such API functions as `GetDIBits()`, `RegisterWndClassEx()`, and `IsEqualIID()`, and a plethora of functions that require Windows handles to be passed around.

However, C++ developers always had easy access to the entire Windows API, unlike Visual Basic 6 and Java developers who were more restricted in terms of the basic operating system functionality that they have access to from their respective languages. What is new about the .NET base classes is that they combine the ease of use that was typical of the Visual Basic and Java libraries with the relatively comprehensive coverage of the Windows API functions. Many features of Windows are still not available through the base classes, and for those you need to call into the API functions, but in general, these are now confined to the more exotic features. For everyday use, you can probably find the base classes adequate. Moreover, if you do need to call into an API function, .NET offers a *platform-invoke* that ensures data types are correctly converted, so the task is no harder than calling the function directly from C++ code would have been — regardless of whether you code in C#, C++, or Visual Basic 2013.

Although Chapter 3 is nominally dedicated to the subject of base classes, after you have completed the coverage of the syntax of the C# language, most of the rest of this book shows you how to use various classes within the .NET base class library for the .NET Framework 4.5. That is how comprehensive base classes are. As a rough guide, the areas covered by the .NET 4.5 base classes include the following:

- Core features provided by IL (including the primitive data types in the CTS discussed in Chapter 2)
- Windows UI support and controls (see Chapters 35–39)
- ASP.NET with Web Forms and MVC (see Chapters 30–42)
- Data access with ADO.NET and XML (see Chapters 32–34)
- File system and registry access (see Chapter 24, "Manipulating Files and Registry")
- Networking and web browsing (see Chapter 26, "Networking")
- .NET attributes and reflection (see Chapter 15)
- COM interoperability (see Chapter 23)

Incidentally, according to Microsoft sources, a large proportion of the .NET base classes have actually been written in C#.

NAMESPACES

Namespaces are the way that .NET avoids name clashes between classes. They are designed to prevent situations in which you define a class to represent a customer, name your class `Customer`, and then someone else does the same thing. (A likely scenario in which — the proportion of businesses that have customers seems to be quite high.)

A namespace is no more than a grouping of data types, but it has the effect that the names of all data types within a namespace are automatically prefixed with the name of the namespace. It is also possible to nest namespaces within each other. For example, most of the general-purpose .NET base classes are in a namespace called `System`. The base class `Array` is in this namespace, so its full name is `System.Array`.

.NET requires all types to be defined in a namespace; for example, you could place your `Customer` class in a namespace called `YourCompanyName.ProjectName`. This class would have the full name `YourCompanyName.ProjectName.Customer`.

NOTE *If a namespace is not explicitly supplied, the type will be added to a nameless global namespace.*

Microsoft recommends that for most purposes you supply at least two nested namespace names: the first one represents the name of your company, and the second one represents the name of the technology or software package of which the class is a member, such as `YourCompanyName.SalesServices.Customer`. This protects, in most situations, the classes in your application from possible name clashes with classes written by other organizations.

Chapter 2 looks more closely at namespaces.

CREATING .NET APPLICATIONS USING C#

You can also use C# to create console applications: text-only applications that run in a DOS window. You can probably use console applications when unit testing class libraries and for creating UNIX or Linux daemon processes. More often, however, you can use C# to create applications that use many of the technologies associated with .NET. This section gives you an overview of the different types of applications that you can write in C#.

Creating ASP.NET Applications

The original introduction of ASP.NET 1.0 fundamentally changed the web programming model. ASP.NET 4.5 is a major release of the product and builds upon its earlier achievements. ASP.NET 4.5 follows on a series of major revolutionary steps designed to increase your productivity. The primary goal of ASP.NET is to enable you to build powerful, secure, dynamic applications using the least possible amount of code. As this is a C# book, there are many chapters showing you how to use this language to build the latest in web applications.

The following section explores the key features of ASP.NET. For more details, refer to Chapters 40 to 42.

Features of ASP.NET

With the invention of ASP.NET, there were only ASP.NET Web Forms, which had the goal of easily creating web applications in a way a Windows application developer was used to writing applications. It was the goal not to need to write HTML and JavaScript.

Nowadays this is different again. HTML and JavaScript became important and modern again. And there's a new ASP.NET Framework that makes it easy to do this and gives a separation based on the well-known Model View Controller (MVC) pattern for easier unit testing: ASP.NET MVC.

ASP.NET was refactored to have a foundation available both for ASP.NET Web Forms and ASP.NET MVC, and then the UI frameworks are based on this foundation.

NOTE *Chapter 40, "Core ASP.NET" covers the foundation of ASP.NET.*

ASP.NET Web Forms

To make web page construction easy, Visual Studio 2013 supplies *Web Forms*. Web pages can be built graphically by dragging controls from a toolbox onto a form and then flipping over to the code aspect of that form and writing event handlers for the controls. When you use C# to create a Web Form, you create a C# class that inherits from the `Page` base class and an ASP.NET page that designates that class as its code-behind. Of course, you do not need to use C# to create a Web Form; you can use Visual Basic 2013 or another .NET-compliant language just as well.

ASP.NET Web Forms provide a rich functionality with controls that do not create only simple HTML code, but with controls that do input validation using both JavaScript and server-side validation logic, grids, data sources to access the database, offer Ajax features for dynamically rendering just parts of the page on the client, and much more.

NOTE *Chapter 41, "ASP.NET Web Forms" discusses ASP.NET Web Forms.*

Web Server Controls

The controls used to populate a Web Form are not controls in the same sense as ActiveX controls. Rather, they are XML tags in the ASP.NET namespace that the web browser dynamically transforms into HTML and client-side script when a page is requested. Amazingly, the web server can render the same server-side control in different ways, producing a transformation appropriate to the requestor's particular web browser. This means that it is now easy to write fairly sophisticated user interfaces for web pages, without worrying about how to ensure that your page can run on any of the available browsers — because Web Forms take care of that for you.

You can use C# or Visual Basic 2013 to expand the Web Form toolbox. Creating a new server-side control is simply a matter of implementing .NET's `System.Web.UI.WebControls.WebControl` class.

ASP.NET MVC

Visual Studio comes with ASP.NET MVC 4. This technology is already available in version 4. Contrary to Web Forms where HTML and JavaScript is abstracted away from the developer, with the advent of HTML 5 and jQuery, using these technologies has become more important again. With ASP.NET MVC the focus is on writing server-side code separated within model and controller and using views with just a little bit of server-side code to get information from the controller. This separation makes unit testing a lot easier and gives the full power to use HTML 5 and JavaScript libraries.

NOTE *Chapter 42, "ASP.NET MVC" covers ASP.NET MVC.*

Windows Presentation Foundation (WPF)

For creating Windows desktop applications, two technologies are available: Windows Forms and Windows Presentation Foundation. Windows Forms consists of classes that just wrap native Windows controls and is thus based on pixel graphics. *Windows Presentation Foundation* (*WPF*) is the newer technology based on vector graphics.

WPF makes use of XAML in building applications. XAML stands for eXtensible Application Markup Language. This new way to create applications within a Microsoft environment is something introduced in 2006 and is part of the .NET Framework 3.0. This means that to run any WPF application, you need to make sure that at least the .NET Framework 3.0 is installed on the client machine. Of course, you get new WPF features with newer versions of the framework. With version 4.5, for example, the ribbon control and live shaping are new features among many new controls.

XAML is the XML declaration used to create a form that represents all the visual aspects and behaviors of the WPF application. Though you can work with a WPF application programmatically, WPF is a step in the direction of declarative programming, which the industry is moving to. *Declarative programming* means that instead of creating objects through programming in a compiled language such as C#, VB, or Java, you declare everything through XML-type programming. Chapter 29, "Core XAML," introduces XAML (which is also used with XML Paper Specification, Windows Workflow Foundation, and Windows Communication Foundation).

Chapter 35, "Core WPF," details how to build WPF applications using XAML and C#. Chapter 36, "Business Applications with WPF," goes into more details on data-driven business applications with WPF and XAML. Printing and creating documents is another important aspect of WPF covered in Chapter 37, "Creating Documents with WPF."

Windows Store Apps

Windows 8 started a new paradigm with touch-first Windows Store apps. With desktop applications the user usually gets a menu and a toolbar, and receives a chrome with the application to see what he can do next. Windows Store apps have the focus on the content. Chrome should be minimized to tasks the user can do with the content, and not on different options he has. The focus is on the current task, and not what the user might do next. This way the user remembers the application based on its content. Content and no chrome is a buzz phrase with this technology.

Windows Store apps can be written with C# and XAML, using the Windows Runtime with a subset of the .NET Framework. Windows Store apps offer huge new opportunities. The major disadvantage is that they are only available with Windows 8 and newer Windows operating systems.

> **NOTE** *Chapter 31, "Windows Runtime", Chapter 38, "Windows Store Apps: UI," and Chapter 39, "Windows Store Apps: Contracts and Devices," cover creating Windows Store apps.*

Windows Services

A Windows Service (originally called an NT Service) is a program designed to run in the background in Windows NT kernel based operating systems. Services are useful when you want a program to run continuously and ready to respond to events without having been explicitly started by the user. A good example is the World Wide Web Service on web servers, which listens for web requests from clients.

It is easy to write services in C#. .NET Framework base classes are available in the `System.ServiceProcess` namespace that handles many of the boilerplate tasks associated with services. In addition, Visual Studio .NET enables you to create a C# Windows Service project, which uses C# source code for a basic Windows Service. Chapter 27, "Windows Services," explores how to write C# Windows Services.

Windows Communication Foundation

One communication technology fused between client and server is the ASP.NET Web API. The ASP.NET Web API is easy to use but doesn't offer a lot of features such as offered from the SOAP protocol.

Windows Communication Foundation (*WCF*) is a feature-rich technology to offer a broad set of communication options. With WCF you can use a REST-based communication but also a SOAP-based communication with all the features used by standards-based web services such as security, transactions, duplex and one-way communication, routing, discovery, and so on. WCF provides you with the ability to build your service one time and then expose this service in a multitude of ways (under different protocols even) by just making changes within a configuration file. You can find that WCF is a powerful new way to connect disparate systems. Chapter 43, "Windows Communication Foundation," covers this in detail. You can also find WCF-based technologies such as Message Queuing with WCF in Chapter 47, "Message Queuing."

ASP.NET Web API

A new way for simple communication to occur between the client and the server — a REST-based style — is offered with the *ASP.NET Web API*. This new framework is based on ASP.NET MVC and makes use of controllers and routing. The client can receive JSON or Atom data based on the Open Data specification.

The features of this new API make it easy to consume from web clients using JavaScript and also from Windows Store apps.

> **NOTE** *The ASP.NET Web API is covered in Chapter 44, "ASP.NET Web API."*

Windows Workflow Foundation

The *Windows Workflow Foundation* (*WF*) was introduced with the release of the .NET Framework 3.0 but had a good overhaul that many find more approachable now since .NET 4. There are some smaller improvements with .NET 4.5 as well. You can find that Visual Studio 2013 has greatly improved for working with WF and makes it easier to construct your workflows and write expressions using C# (instead of VB in the previous edition). You can also find a new state machine designer and new activities.

NOTE *WF is covered in Chapter 45, "Windows Workflow Foundation."*

THE ROLE OF C# IN THE .NET ENTERPRISE ARCHITECTURE

New technologies are coming at a fast pace. What should you use for enterprise applications? There are many aspects that influence the decision. For example, what about the existing applications that have been developed with current technology knowledge of the developers. Can you integrate new features with legacy applications? Depending on the maintenance required, maybe it makes sense to rebuild some existing applications for easier use of new features. Usually, legacy and new can coexist for many years to come. What is the requirement for the client systems? Can the .NET Framework be upgraded to version 4.5, or is 2.0 a requirement? Or is .NET not available on the client?

There are many decisions to make, and .NET gives many options. You can use .NET on the client with Windows Forms, WPF, or Windows 8-style apps. You can use .NET on the web server hosted with IIS and the ASP.NET Runtime with ASP.NET Web Forms or ASP.NET MVC. Services can run within IIS, and you can host the services from within Windows Services. C# presents an outstanding opportunity for organizations interested in building robust, n-tiered client-server applications.

When combined with ADO.NET, C# has the capability to quickly and generically access data stores such as SQL Server or other databases with data providers. The ADO.NET Entity Framework can be an easy way to map database relations to object hierarchies. This is not only possible with SQL Server, but also many different databases where an Entity Framework provider is offered. The returned data can be easily manipulated using the ADO.NET object model or LINQ and automatically rendered as XML or JSON for transport across an office intranet.

After a database schema has been established for a new project, C# presents an excellent medium for implementing a layer of data access objects, each of which could provide insertion, updates, and deletion access to a different database table.

Because it's the first component-based C language, C# is a great language for implementing a business object tier, too. It encapsulates the messy plumbing for intercomponent communication, leaving developers free to focus on gluing their data access objects together in methods that accurately enforce their organizations' business rules.

To create an enterprise application with C#, you create a class library project for the data access objects and another for the business objects. While developing, you can use Console projects to test the methods on your classes. Fans of extreme programming can build Console projects that can be executed automatically from batch files to unit test that working code has not been broken.

On a related note, C# and .NET will probably influence the way you physically package your reusable classes. In the past, many developers crammed a multitude of classes into a single physical component because this arrangement made deployment a lot easier; if there were a versioning problem, you knew just where to look. Because deploying .NET components involves simply copying files into directories, developers can now package their classes into more logical, discrete components without encountering "DLL Hell."

Last, but not least, ASP.NET pages coded in C# constitute an excellent medium for user interfaces. Because ASP.NET pages compile, they execute quickly. Because they can be debugged in the Visual Studio 2013 IDE, they are robust. Because they support full-scale language features such as early binding, inheritance, and modularization, ASP.NET pages coded in C# are tidy and easily maintained.

After the hype of SOA and service-based programming, nowadays using services has becoming the norm. The new hype is cloud-based programming, with Windows Azure as Microsoft's offering. You can run .NET applications in a range from ASP.NET Web Forms, ASP.NET Web API, or WCF either on on-premise servers or in the cloud. Clients can make use of HTML 5 for a broad reach or make use of WPF or Windows Store apps for rich functionality. Still with new technologies and options, .NET has a prosperous life.

SUMMARY

This chapter covered a lot of ground, briefly reviewing important aspects of the .NET Framework and C#'s relationship to it. It started by discussing how all languages that target .NET are compiled into Microsoft Intermediate Language (IL) before this is compiled and executed by the Common Language Runtime (CLR). This chapter also discussed the roles of the following features of .NET in the compilation and execution process:

- Assemblies and .NET base classes
- COM components
- JIT compilation
- Application domains
- Garbage collection

Figure 1-4 provides an overview of how these features come into play during compilation and execution.

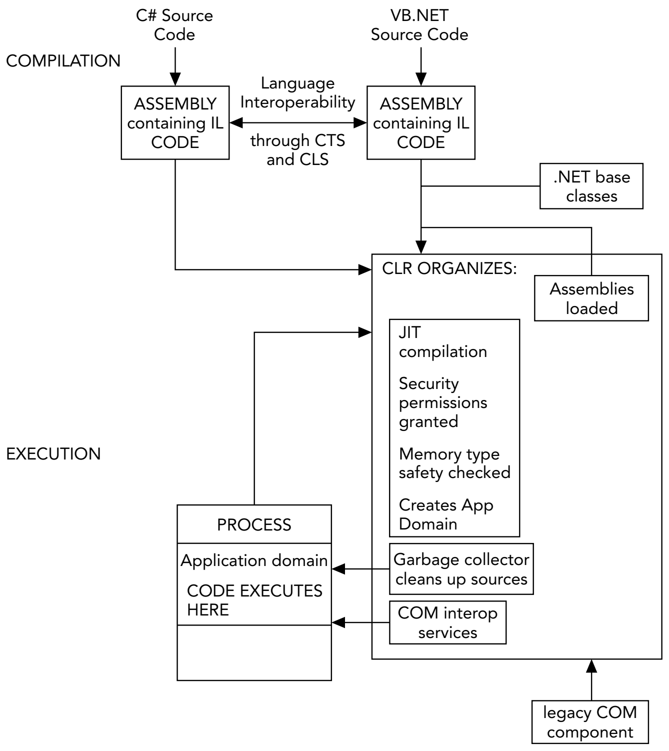

FIGURE 1-4

You learned about the characteristics of IL, particularly its strong data typing and object orientation, and how these characteristics influence the languages that target .NET, including C#. You also learned how the strongly typed nature of IL enables language interoperability, as well as CLR services such as garbage collection and security. There was also a focus on the Common Language Specification (CLS) and the Common Type System (CTS) to help deal with language interoperability.

Finally, you learned how C# can be used as the basis for applications built on several .NET technologies, including ASP.NET and WPF.

Chapter 2 discusses how to write code in C#.