1

.NET Architecture

You'll find that we emphasize throughout this book that the C# language cannot be viewed in isolation, but must be considered in parallel with the .NET Framework. The C# compiler specifically targets .NET, which means that all code written in C# will always run within the .NET Framework. This has two important consequences for the C# language:

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

Because of this dependence, it is important to gain some understanding of the architecture and methodology of .NET before we begin C# programming. That is the purpose of this chapter.

We will begin by going over what happens when all code (including C#) that targets .NET is compiled and run. Once we have this broad overview, we will take a more detailed look at the *Microsoft Intermediate Language* (MSIL or simply IL), the assembly language which all compiled code ends up in on .NET. In particular, we will see how IL, in partnership with the *Common Type System* (CTS) and *Common Language Specification* (CLS) works to give us interoperability between languages that target .NET. We'll also discuss where common languages (including Visual Basic and C++) fit into .NET.

Once we've done that, we will move on to examine some of the other features of .NET, including assemblies, namespaces, and the .NET base classes. We'll finish the chapter with a brief look at the kinds of applications we can create as C# developers.

The Relationship of C# to .NET

C# is a relatively new programming language, 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 has been able to learn from the experience of all the other similar languages that have been around since object-oriented principles came to prominence some 20 years ago.

One important thing to make clear is that C# is a language in its own right. Although it is designed to generate code that targets the .NET environment, it is not itself part of .NET. There are some features that are supported by .NET but not by C#, and you might be surprised to learn that there are actually features of the C# language that are not supported by .NET (for example, some instances of operator overloading)!

However, since the C# language is intended for use with .NET, it is important for us to have an understanding of this Framework if we want to develop applications in C# effectively. So, in this chapter we're going to take some time to peek underneath the surface of .NET. Let's get started.

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 we develop (in C# or some other language) needs to be compiled. Compilation occurs in two steps in .NET:

1. Compilation of source code to IL
2. Compilation of IL to platform-specific code by the CLR

This two-stage compilation process is very important, because the existence of the IL (managed code) is the key to providing many of the benefits of .NET. Let's see why.

Advantages of Managed Code

Microsoft intermediate language 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 very quickly translated into native machine code. Having this well-defined universal syntax for code has significant advantages.

Platform independence

First, it 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 will run on that particular platform. In other words, by compiling to IL we obtain platform independence for .NET, in much the same way as compiling to Java byte code gives Java platform independence.

You should note that the platform independence of .NET is only theoretical at present because, at the time of writing, a complete implementation of .NET is only available for Windows. However, there is a partial implementation available (see for example the Mono project, an effort to create an open source implementation of .NET, at `www.go-mono.com/`).

Performance improvement

Although we previously made comparisons with 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 of translating from Java byte code to native executable resulted in a loss of performance (with the exception of more recent cases, where Java is JIT compiled on certain platforms).

Instead of compiling the entire application in one go (which could lead to a slow start-up 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 will never be compiled.

This explains why we can expect that execution of managed IL code will be almost as fast as executing native machine code. What it doesn't explain is why Microsoft expects that we will get a performance *improvement*. The reason given for this is that, since the final stage of compilation takes place at runtime, the JIT compiler will know exactly what processor type the program will run 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 will optimize the code, but they can only perform optimizations that are independent of the particular processor that the code will run on. This is because traditional compilers compile to native executable before the software is shipped. This means that the compiler doesn't know what type of processor the code will run on beyond basic generalities, such as that it will be an x86-compatible processor or an Alpha processor. Visual Studio 6, for example, optimizes for a generic Pentium machine, so the code that it generates cannot take advantage of hardware features of Pentium III processors. On the other hand, the JIT compiler can do all the optimizations that Visual Studio 6 can, and in addition it will optimize for the particular processor the code is running on.

Language interoperability

The use of IL not only enables platform independence; 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're probably now wondering which languages aside from C# are interoperable with .NET, so let's briefly discuss how some of the other common languages fit into .NET.

Visual Basic .NET

Visual Basic .NET has undergone a complete revamp from Visual Basic 6 to bring it up-to-date with .NET. The way that Visual Basic has evolved over the last few years means that in its previous version, Visual Basic 6, it was not a suitable language for running .NET programs. For example, it is heavily integrated

into 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 Visual Basic 6 uses are incompatible with .NET.

Visual Basic 6 was upgraded to Visual Basic .NET, and the changes that were made to the language are so extensive you might as well regard Visual Basic .NET as a new language. Existing Visual Basic 6 code does not compile as Visual Basic .NET code. Converting a Visual Basic 6 program to Visual Basic .NET requires extensive changes to the code. However, Visual Studio .NET (the upgrade of VS for use with .NET) can do most of the changes for you. If you attempt to read a Visual Basic 6 project into Visual Studio .NET, it will upgrade the project for you, which means that it will rewrite the Visual Basic 6 source code into Visual Basic .NET source code. Although this means that the work involved for you is heavily cut down, you will need to check through the new Visual Basic .NET code to make sure that the project still works as intended because the conversion might not be perfect.

One side effect of this language upgrade is that it is no longer possible to compile Visual Basic .NET to native executable code. Visual Basic .NET compiles only to IL, just as C# does. If you need to continue coding in Visual Basic 6, you may do so, but the executable code produced will completely ignore the .NET Framework, and you'll need to keep Visual Studio 6 installed if you want to continue to work in this developer environment.

Visual C++ .NET

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, then 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:

```
__gc class MyClass
{
```

will give you a managed class, just as if you'd written the class in C# or Visual Basic .NET. The advantage of using managed C++ over C# code is that we can call unmanaged C++ classes from managed C++ code without having to resort to COM interop.

The compiler raises an error if you attempt to use features that are not supported by .NET on managed types (for example, templates or multiple inheritance of classes). You will also find that you will need to

use nonstandard C++ features (such as the `__gc` keyword shown in the previous code) when using managed classes.

Because of the freedom that C++ allows in terms of low-level pointer manipulation and so on, the C++ compiler is not able to generate code that will pass the CLR's memory type safety tests. If it's important that your code is recognized by the CLR as memory type safe, then you'll need to write your source code in some other language (such as C# or Visual Basic .NET).

Visual J# .NET

The latest language to be added to the mix is Visual J# .NET. Prior to .NET Framework 1.1, users were able to use J# only after making a separate download. Now the J# language is built into the .NET Framework. Because of this, J# users are able to take advantage of all the usual features of Visual Studio .NET. Microsoft expects that most J++ users will find it easiest to use J# if they want to work with .NET. Instead of being targeted at the Java runtime libraries, J# uses the same base class libraries that the rest of the .NET compliant languages use. This means that you can use J# for building ASP.NET Web applications, Windows Forms, XML Web services, and everything else that is possible—just as C# and Visual Basic .NET can.

Scripting languages

Scripting languages are still around, although, in general, their importance is likely to decline with the advent of .NET. JScript, on the other hand, has been upgraded to JScript .NET. We can now write ASP.NET pages in JScript .NET, run JScript .NET as a compiled rather than an interpreted language, and write strongly typed JScript .NET code. With ASP.NET there is no reason to use scripting languages in server-side Web pages. VBA is, however, still used as a language for Microsoft Office and Visual Studio macros.

COM and COM+

Technically speaking, COM and COM+ aren't technologies targeted at .NET, because components based on them cannot be compiled into IL (although it's possible to do so to some degree using managed C++, if the original COM component was written in C++). However, COM+ remains an important tool, because its features are not duplicated in .NET. Also, COM components will still work—and .NET incorporates COM interoperability features that make it possible for managed code to call up COM components and vice versa (this is discussed in Chapter 29). In general, however, 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 as well as the other benefits of running as managed code.

A Closer Look at Intermediate Language

From what we learned in the previous section, Microsoft intermediate language obviously plays a fundamental role in the .NET Framework. As C# developers, we now understand that our C# code will be compiled into IL before it is executed (indeed, the C# compiler *only* compiles to managed code). It makes sense, then, that we should now take a closer look at the main characteristics of IL, since any language that targets .NET would logically need to support the main characteristics of IL too.

Here are the important features of IL:

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

Let's now take a closer look at each of these characteristics.

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 are going to have 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.

> *Those readers unfamiliar with the concepts of object orientation should refer to Appendix A for more information. Appendix A is posted at* `www.wrox.com`.

Besides classic object-oriented programming, IL also brings in the idea of interfaces, which saw their first implementation under Windows with COM. .NET interfaces 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 they do not have associated 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.

Object orientation and language interoperability

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

To start with, we need to consider exactly what we mean by language interoperability. After all, COM allowed 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 allow components to instantiate other components and call methods or properties against them, without worrying about the language the respective components were written in. In order 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 that are hosted as an executable rather than DLL files, separate processes would need to be created in order 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 have to debug components written in different languages independently. It was not possible to step between languages in the debugger. So what we *really* mean by language interoperability is that classes written in one language should be able to 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 we 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 really offered by the Visual Studio .NET IDE rather than by the CLR itself.

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, while *reference types* are those for which a variable simply stores the address at which the corresponding data can be found.

In C++ terms, reference types can be considered to be similar to accessing a variable through a pointer, while 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*, while value types are normally stored on the *stack* (although if value types are declared as fields within reference types, then they will be stored inline on the heap). We will discuss the stack and the heap and how they work in Chapter 3.

Strong Data Typing

One very important aspect of IL is that it is based on exceptionally *strong data typing*. What we mean by that is 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 being able to pass 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. Being able to perform 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 only permitted 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.

You should note that some languages compatible with .NET, such as Visual Basic .NET, still allow some laxity in typing, but that is only possible because the compilers behind the scenes ensure 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:

- Language interoperability
- Garbage collection
- Security
- Application domains

Let's take a closer look at why strong data typing is particularly important for these features of .NET.

The Importance of strong data typing for 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 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 .NET class is defined to return an `Integer`—one of the standard data types available in Visual Basic .NET. C# simply does not have any data type of that name. Clearly, we will only be able to derive from the class, use this method, and use the return type from C# code, if the compiler knows how to map Visual Basic .NET's `Integer` type to some known type that is defined in C#. So how is this problem circumvented in .NET?

Common Type System

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

For the example that we were considering before, Visual Basic .NET's `Integer` is actually a 32-bit signed integer, which maps exactly to the IL type known as `Int32`. This will therefore be 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 will simply treat the Visual Basic .NET method as if it returned an `int`.

The CTS doesn't merely specify 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 Common Type System reflects the single-inheritance object-oriented methodology of IL, and resembles Figure 1-1.

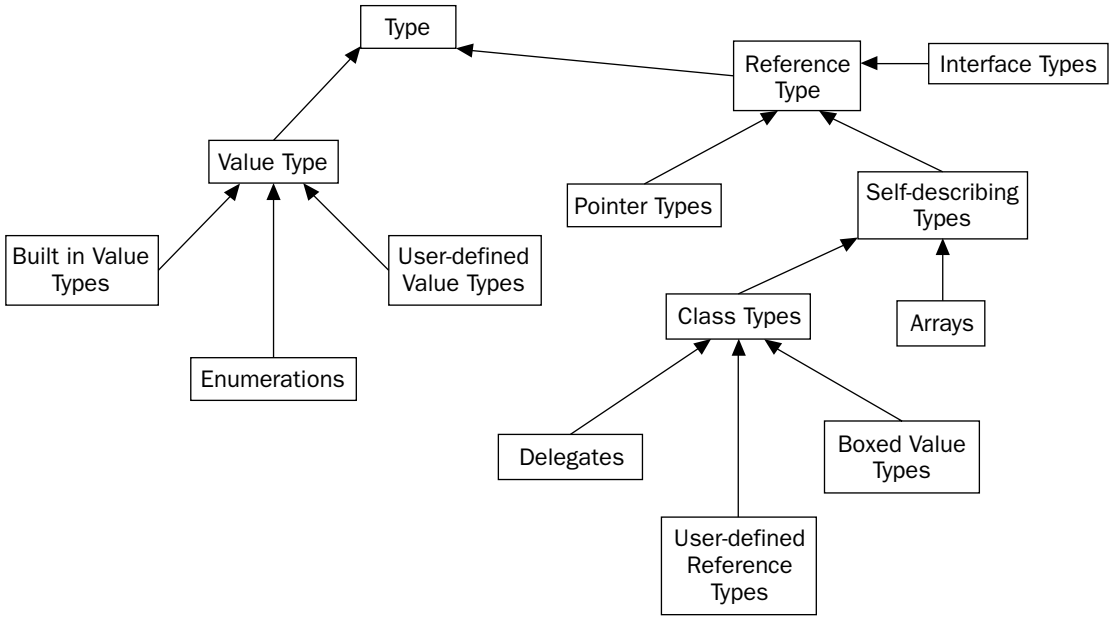

Figure 1-1

The following table explains the types shown in Figure 1-1.

Type	Meaning
`Type`	Base class that represents any type.
`Value Type`	Base class that represents any value type.
`Reference Types`	Any data types that are accessed through a reference and stored on the heap.
`Built-in Value Types`	Includes most of the standard primitive types, which represent numbers, Boolean values, or characters.
`Enumerations`	Sets of enumerated values.
`User-defined Value Types`	Types that have been defined in source code and are stored as value types. In C# terms, this means any struct.
`Interface Types`	Interfaces.
`Pointer Types`	Pointers.
`Self-describing Types`	Data types that provide information about themselves for the benefit of the garbage collector (see the next section).
`Arrays`	Any type that contains an array of objects.
`Class Types`	Types that are self-describing but are not arrays.

Table continued on following page

Type	Meaning
`Delegates`	Types that are designed to hold references to methods.
`User-defined Reference Types`	Types that have been defined in source code and are stored as reference types. In C# terms, this means any class.
`Boxed Value Types`	A value type that is temporarily wrapped in a reference so that it can be stored on the heap.

We won't list all of the built-in value types here, because they are covered in detail in Chapter 2. In C#, each predefined type recognized by the compiler maps onto one of the IL built-in types. The same is true in Visual Basic .NET.

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. Since IL is a very rich language, writers of most compilers will prefer to restrict the capabilities of a given compiler to only support a subset of the facilities offered by IL and the CTS. That is fine, as long as the compiler supports everything that is defined in the CLS.

> **It is perfectly acceptable to write non–CLS-compliant code. However, if you do, the compiled IL code isn't guaranteed to be fully language interoperable.**

For example, let's look at case sensitivity. IL is case-sensitive. Developers who work with case-sensitive languages regularly take advantage of the flexibility this case sensitivity gives them when selecting variable names. Visual Basic .NET, 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 .NET code can work with CLS-compliant code.

This example shows that the CLS works in two ways. First, it means that individual compilers do not have 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. Second, it provides a guarantee that, if you restrict your classes to only exposing CLS-compliant features, then it is guaranteed 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 only applies 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 this chapter) cannot access this part of your code anyway.

We won't go into the details of the CLS specifications here. In general, the CLS won't affect your C# code very much, because there are very few non–CLS-compliant features of C# anyway.

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 there have been two techniques 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 it has the advantage that (in general) resources are never occupied for longer than unnecessary. 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, 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 being 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 only takes 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 will 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. This is a program whose purpose 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). Every so often, 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 to say which objects have references that refer to them. Any objects that are not referred to are deemed to be no longer accessible from your code and can therefore be removed. Java uses a similar system of garbage collection 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, what we mean is that if any reference to an object exists, then there is sufficient information in the reference to exactly determine the type of the object.

It would not be possible to use the garbage collection mechanism with a language such as unmanaged C++, for example, because C++ allows 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 (unless you explicitly call the collector). Though it is also possible to override this process and call up the garbage collector in your code.

Security

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

Role-based security is based on the identity of the account under which the process is running, in other words, who owns and is running the process. Code-based security on the other hand is based on what the code actually does and on how much the code is trusted. Thanks to the strong type safety of IL, the CLR is able to inspect code before running it in order to determine required security permissions. .NET also offers a mechanism by which code can indicate in advance what security permissions it will require 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've downloaded from the Internet). For example, even if code is running under the administrator account, it is possible to use code-based security to indicate that that code should still not be permitted to perform certain types of operation 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 to access the .NET reflection features.

Security issues are covered in more depth in Chapter 14.

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 which also need to be able 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 will, 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 Web site down, or isolating those instances in separate processes, with the associated performance overhead.

Up until now, the only means of isolating code has been through 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 available 4GB of virtual memory 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).

In general, any process is only able to access memory 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 will not be able to damage anything outside its own address space. (Note that on Windows 95/98, these safeguards are not quite as thorough as they are on Windows NT/2000/XP/2003, so the theoretical possibility exists of applications crashing Windows by writing to inappropriate memory.)

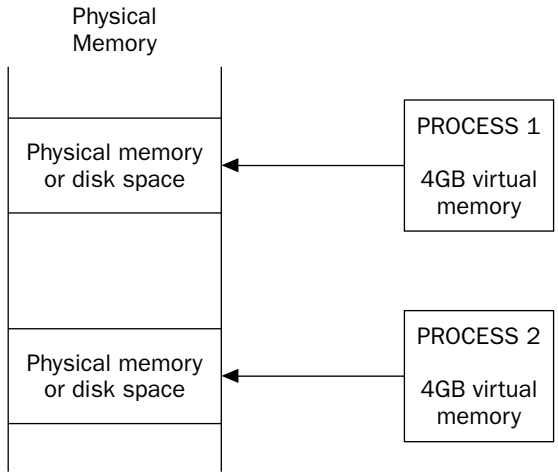

Figure 1-2

Processes don't just serve as a way to isolate instances of running code from each other. On Windows NT/2000/XP/2003 systems, 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.

While processes are great for security reasons, their big disadvantage is in the area of performance. Often a number of processes will actually be working 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. Since processes cannot share any memory, a complex marshaling process has to be used to copy data between the processes. This results in a very significant performance hit. If you need components to work together and don't want that performance hit, then you have to use DLL-based components and have everything running in the same address space—with the associated risk that a badly behaved component will bring everything else down.

Application domains are designed as a way of separating 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 will be running in a particular application domain (see Figure 1-3).

If different executables are running in the same process space, then they are clearly able to 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 its own data areas. This sounds at first sight like an almost impossible trick to pull off—after all how can you tell what the program is going to do without actually running it?

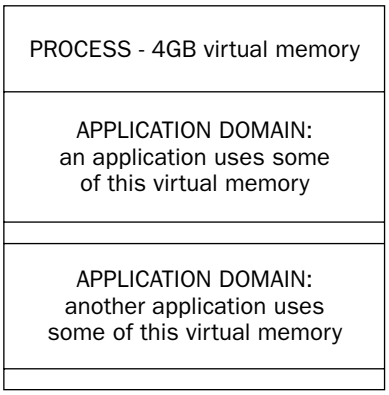

Figure 1-3

In fact, it is usually possible to do this because of the strong type safety of the IL. In most cases, unless code is using unsafe features such as pointers, the data types it is using will ensure 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, then 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 said to be *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 list, is supported by .NET and by C#.

We will cover exceptions in detail in Chapter 11. Briefly, the idea is that certain areas of code are designated as exception handler routines, with each one able to deal 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 wish. 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 .NET, 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 11.)

What .NET does do, however, is provide the infrastructure to allow 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 allow the thrown exception objects to be interpreted by the exception handling code, irrespective 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. The fact that exceptions are handled consistently in different languages is a crucial aspect of facilitating multi-language development.

Use of Attributes

Attributes are a feature that is 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 hence now by C++, C#, and Visual Basic .NET. What is, however, particularly innovative about attributes in .NET is that a mechanism exists whereby 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, where they can be used in conjunction with reflection technology in order to perform programming tasks based on attributes. Also, in common with the .NET philosophy of language independence, attributes can be defined in source code in one language, and read by code that is written in another language.

Attributes are covered in Chapter 10.

Assemblies

An *assembly* is the logical unit that contains compiled code targeted at the .NET Framework. We are not going to cover assemblies in great detail in this chapter, because they are covered in detail in Chapter 13, but we will summarize the main points here.

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 at all). If an assembly is stored in more than one file, then there will be one main file that contains the entry point and describes the other files in the assembly.

Note that the same assembly structure is used for both executable code and library code. The only real difference is that an executable assembly contains a main program entry point, whereas a library assembly doesn't.

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 itself. This assembly metadata, contained in an area known as the *manifest*, allows checks to be made on the version of the assembly, and on its integrity.

ildasm, a Windows-based utility, can be used to inspect the contents of an assembly, including the manifest and metadata. We discuss ildasm in Chapter 13.

The fact that 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, in order to find out how to use that assembly. This is a significant break from the old COM way of doing 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 being able to use the component successfully. With assemblies, there is no risk of this happening, because all the metadata is stored with the program executable instructions. Note that 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 gets 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: *shared* and *private* 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 will ship private assemblies is when you are supplying an application in the form of an executable and a number of libraries, where the libraries contain code that should only be used with that application.

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

Because we would normally expect that commercial software would always be installed in its own directory, this means that there is no risk of one software package overwriting, modifying, or accidentally loading private assemblies intended for another package. As private assemblies can only be used by the software package that they are intended for, this means that you have much more control over what software uses them. There is, therefore, less need to take security precautions, since there is no risk, for example, of some other commercial software overwriting one of your assemblies with some new version of it (apart from the case where software is 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 doesn't matter, because any given application will only be able to see the one set of private assemblies.

Because a private assembly is entirely self-contained, the process of deploying it is simple. You simply place the appropriate file(s) in the appropriate folder in the file system (there are no registry entries that 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 being incompatible with some existing client code.

The solution to these problems involves 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 needs to be specifically installed into the cache. This process can be performed by a number of .NET utilities and involves carrying out certain checks on the assembly, as well as setting up a small folder hierarchy within the assembly cache that is used to ensure assembly integrity.

In order to avoid the risk of name collisions, shared assemblies are given a name that is based on private key cryptography (private assemblies are simply given the same name as their main file name). This name is known as a *strong name*, 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

Since assemblies store metadata, including details of all the types and members of these types that are defined in the assembly, it is possible to access this metadata programmatically. Full details of this can be found in Chapter 10. This technique, known as *reflection*, raises interesting possibilities, since it means that managed code can actually examine other managed code, or 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 of instantiating classes or calling methods, given the names of those classes on methods as strings. In this way you could select classes to instantiate methods to call at runtime, rather than compile time, based on user input (dynamic binding).

.NET Framework Classes

Perhaps one of the biggest benefits of writing managed code, at least from a developer's point of view, is that you get to use the .NET *base class library*.

The .NET base classes are a massive collection of managed code classes that allow you to do almost any of the tasks that were previously available through the Windows API. These classes follow the same object model IL uses, based on single inheritance. This means that you can either instantiate objects of whichever .NET base class is appropriate, or you can derive your own classes from them.

The great thing about the .NET base classes is that they have been designed to be very 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—while 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()`, as well as a whole plethora of functions that required Windows handles to be passed around.

On the other hand, C++ developers always had easy access to the entire Windows API, whereas Visual Basic 6 and Java developers 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. There are still many features of Windows that are not available through the base classes, and for which you will need to call into the API functions, but in general, these are now confined to the more exotic features. For everyday use, you will probably find the base classes adequate. And if you do need to call into an API function, .NET offers a so-called *platform-invoke* which 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 are coding in C#, C++, or Visual Basic .NET.

> *WinCV, a Windows-based utility, can be used to browse the classes, structs, interfaces, and enums in the base class library. We discuss WinCV in Chapter 12.*

Although Chapter 3 is nominally dedicated to the subject of base classes, in reality, once we have completed our 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. That is how comprehensive base classes are. As a rough guide, the areas covered by the .NET base classes include:

- Core features provided by IL (including, the primitive data types in the CTS discussed in Chapter 3)
- Windows GUI support and controls (see Chapter 19)
- Web Forms (ASP.NET, discussed in Chapters 25 through 27)
- Data Access (ADO.NET, see Chapters 21 and 22)
- Directory Access (see Chapter 24)
- File system and registry access (see Chapter 30)
- Networking and Web browsing (see Chapter 31)
- .NET attributes and reflection (see Chapter 10)
- Access to aspects of the Windows OS (environment variables and so on; see Chapter 14)
- COM interoperability (see Chapters 28 and 29)

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 avoid the situation 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—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 automatically get 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, so for example you could place your `Customer` class in a namespace called `YourCompanyName`. This class would have the full name `YourCompanyName.Customer`.

> *If a namespace is not explicitly supplied, then 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 refers to the name of your company, the second one refers to the name of the technology or software package that the class is a member of, 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.

We will look more closely at namespaces in Chapter 2.

Creating .NET Applications Using C#

C# can also be used to create console applications: text-only applications that run in a DOS window. You'll probably use console applications when unit testing class libraries, and for creating Unix or Linux daemon processes. However, more often you'll use C# to create applications that use many of the technologies associated with .NET. In this section, we'll give you an overview of the different types of application that you can write in C#.

Creating ASP.NET Applications

Active Server Pages (ASP) is a Microsoft technology for creating Web pages with dynamic content. An ASP page is basically an HTML file with embedded chunks of server-side VBScript or JavaScript. When a client browser requests an ASP page, the Web server delivers the HTML portions of the page, processing the server-side scripts as it comes to them. Often these scripts query a database for data, and mark up that data in HTML. ASP is an easy way for clients to build browser-based applications.

However, ASP is not without its shortcomings. First, ASP pages sometimes render slowly because the server-side code is interpreted instead of compiled. Second, ASP files can be difficult to maintain because they were unstructured; the server-side ASP code and plain HTML are all jumbled up together. Third, ASP sometimes make development difficult because there is little support for error handling and type-checking.

Specifically, if you are using VBScript and want to implement error handling in your pages, you have to use the `On Error Resume Next` statement, and follow every component call with a check to `Err.Number` to make sure that the call had gone well.

ASP.NET is a complete revision of ASP that fixes many of its problems. It does not replace ASP; rather, ASP.NET pages can live side by side on the same server with legacy ASP applications. Of course, you can also program ASP.NET with C#!

The following section explores the key features of ASP.NET. For more details, refer to Chapters 25 through 27.

Features of ASP.NET

First, and perhaps most importantly, ASP.NET pages are *structured*. That is, each page is effectively a class that inherits from the .NET `System.Web.UI.Page` *class*, and can override a set of methods that are evoked during the `Page` object's lifetime. (You can think of these events as page-specific cousins of the `OnApplication_Start` and `OnSession_Start` events that went in the `global.asa` files of plain old ASP.) Because you can factor a page's functionality into event handlers with explicit meanings, ASP.NET pages are easier to understand.

Another nice thing about ASP.NET pages is that you can create them in Visual Studio .NET, the same environment in which you create the business logic and data access components that those ASP.NET pages use. A Visual Studio .NET project, or *solution*, contains all of the files associated with an application. Moreover, you can debug your classic ASP pages in the editor as well; in the old days of Visual InterDev, it was often a vexing challenge to configure InterDev and the project's Web server to turn debugging on.

For maximum clarity, the ASP.NET `code-behind` feature lets you take the structured approach even further. ASP.NET allows you to isolate the server-side functionality of a page to a class, compile that class into a DLL, and place that DLL into a directory below the HTML portion. A `code-behind` directive at the top of the page associates the file with its DLL. When a browser requests the page, the Web server fires the events in the class in the page's `code-behind` DLL.

Last but not least, ASP.NET is remarkable for its increased performance. Whereas classic ASP pages are interpreted with each page request, the Web server caches ASP.NET pages after compilation. This means that subsequent requests of an ASP.NET page execute more quickly than the first.

ASP.NET also makes it easy to write pages that cause forms to be displayed by the browser, which you might use in an intranet environment. The traditional wisdom is that form-based applications offer a richer user interface, but are harder to maintain because they run on so many different machines. For this reason, people have relied on form-based applications when rich user interfaces were a necessity and extensive support could be provided to the users.

With the advent of Internet Explorer 5 and the lackluster performance of Navigator 6, however, the advantages of form-based applications are clouded. IE 5's consistent and robust support for DHTML allows the programmer to create Web-based applications that are every bit as pretty as their fat client equivalents. Of course, such applications necessitate standardizing on IE and not supporting Navigator. In many industrial situations, this standardization is now common.

Web Forms

To make Web page construction even easier, Visual Studio .NET supplies *Web Forms*. They allow you to build ASP.NET pages graphically in the same way that Visual Basic 6 or C++ Builder windows are created; in other words, by dragging controls from a toolbox onto a form, 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 are creating 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 don't have to use C# to create a Web Form; you can use Visual Basic .NET or another .NET language just as well.

In the past, the difficulty of Web development has discouraged some teams from attempting it. To succeed in Web development, you had to know so many different technologies, such as VBScript, ASP, DHTML, JavaScript, and so on. By applying the Form concepts to Web pages, Web Forms have made Web development considerably easier.

Web 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 is able to render the same server-side control in different ways, producing a transformation that is 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 having to worry about how to ensure that your page will run on any of the available browsers—because Web Forms will take care of that for you.

You can use C# or Visual Basic .NET 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.

XML Web services

Today, HTML pages account for most of the traffic on the World Wide Web. With XML, however, computers have a device-independent format to use for communicating with each other on the Web. In the future, computers may use the Web and XML to communicate information rather than dedicated lines and proprietary formats such as *Electronic Data Interchange* (EDI). XML Web services are designed for a service-oriented Web, in which remote computers provide each other with dynamic information that can be analyzed and re-formatted, before final presentation to a user. An XML Web service is an easy way for a computer to expose information to other computers on the Web in the form of XML.

In technical terms, an XML Web service on .NET is an ASP.NET page that returns XML instead of HTML to requesting clients. Such pages have a `code-behind` DLL containing a class that derives from the `WebService` class. The Visual Studio .NET IDE provides an engine that facilitates Web Service development.

There are two main reasons that an organization might choose to use XML Web services. The first reason is that they rely on HTTP; XML Web services can use existing networks (HTTP) as a medium for conveying information. The other is that because XML Web services use XML, the data format is self-describing, non-proprietary, and platform-independent.

Creating Windows Forms

Although C# and .NET are particularly suited to Web development, they still offer splendid support for so-called *fat-client* or *thick-client* apps, applications that have to be installed on the end-user's machine where most of the processing takes place. This support is from *Windows Forms*.

A Windows Form is the .NET answer to a Visual Basic 6 Form. To design a graphical window interface, you just drag controls from a toolbox onto a Windows Form. To determine the window's behavior, you write event-handling routines for the form's controls. A Windows Form project compiles to an executable that must be installed alongside the .NET runtime on the end user's computer. Like other .NET project types, Windows Form projects are supported by both Visual Basic .NET and C#. We examine Windows Forms more closely in Chapter 19.

Windows Controls

Although Web Forms and Windows Forms are developed in much the same way, you use different kinds of controls to populate them. Web Forms use Web Controls, and Windows Forms use *Windows Controls*.

A Windows Control is a lot like an ActiveX control. After a Windows control is implemented, it compiles to a DLL that must be installed on the client's machine. In fact, the .NET SDK provides a utility that creates a wrapper for ActiveX controls, so that they can be placed on Windows Forms. As is the case with Web Controls, Windows Control creation involves deriving from a particular class, `System.Windows.Forms.Control`.

Windows Services

A Windows Service (originally called an NT Service) is a program that is designed to run in the background in Windows NT/2000/XP/2003 (but not Windows 9x). Services are useful where you want a program to be running continuously and ready to respond to events without having been explicitly started by the user. A good example would be the World Wide Web Service on Web servers, which listens out for Web requests from clients.

It is very easy to write services in C#. There are .NET Framework base classes available in the `System.ServiceProcess` namespace that handle many of the boilerplate tasks associated with services, and in addition, Visual Studio .NET allows you to create a C# Windows Service project, which uses C# source code for a basic Windows service. We'll explore how to write C# Windows Services in Chapter 32.

The Role of C# in the .NET Enterprise Architecture

C# requires the presence of the .NET runtime, and it will probably be a few years before most clients—particularly most home computers—have .NET installed. In the meantime, installing a C# application is likely to mean also installing the .NET redistributable components. Because of that, it is likely that we will see many C# applications first in the enterprise environment. Indeed, C# arguably presents an outstanding opportunity for organizations that are interested in building robust, n-tiered client-server applications.

When combined with ADO.NET, C# has the ability to access quickly and generically data stores like SQL Server and Oracle databases. The returned datasets can easily be manipulated using the ADO.NET object model, and automatically render as XML for transport across an office intranet.

Once 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 inter-component communication, leaving developers free to focus on gluing their data access objects together in methods that accurately enforce their organizations' business rules. Moreover, with attributes, C# business objects can be outfitted for method-level security checks, object pooling, and JIT activation supplied by COM+ Services. Furthermore, .NET ships with utility programs that allows your new .NET business objects to interface with legacy COM components.

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 was a versioning problem, you knew just where to look. Because deploying .NET enterprise components simply involves 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 .NET IDE, they are robust. Because they support full-scale language features like early binding, inheritance, and modularization, ASP.NET pages coded in C# are tidy and easily maintained.

Seasoned developers acquire a healthy skepticism about strongly hyped new technologies and languages and are reluctant to utilize new platforms simply because they are urged to. If you're an enterprise developer in an IT department, though, or if you provide application services across the World Wide Web, let us assure you that C# and .NET offer at least four solid benefits, even if some of the more exotic features like XML Web services and server-side controls don't pan out:

- Component conflicts will become infrequent and deployment is easier, because different versions of the same component can run side by side on the same machine without conflicting.
- Your ASP.NET code won't look like spaghetti code.
- You can leverage a lot of the functionality in the .NET base classes.
- For applications requiring a Windows Forms user interface, C# makes it very easy to write this kind of application.

Windows Forms have to some extent been downplayed in the last year due to the advent of Web Forms and Internet-based applications. However, if you or your colleagues lack expertise in JavaScript, ASP, or

related technologies, then Windows Forms are still a viable option for creating a user interface with speed and ease. Just remember to factor your code so that the user interface logic is separate from the business logic and the data access code. Doing so will allow you to migrate your application to the browser at some point in the future if you need to do so. Also, it is likely that Windows Forms will remain the dominant user interface for applications for use in homes and small businesses for a long time to come.

Summary

We've covered a lot of ground in this chapter, briefly reviewing important aspects of the .NET Framework and C#'s relationship to it. We 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). We 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.

We also discussed the characteristics of IL, particularly its strong data typing and object-orientation. We noted how these characteristics influence the languages that target .NET, including C#. We also noted how the strongly typed nature of IL enables language interoperability, as well as CLR services such as garbage collection and security.

Finally, we talked about how C# can be used as the basis for applications that are built upon several .NET technologies, including ASP.NET.

The following chapter discusses how to write code in C#.

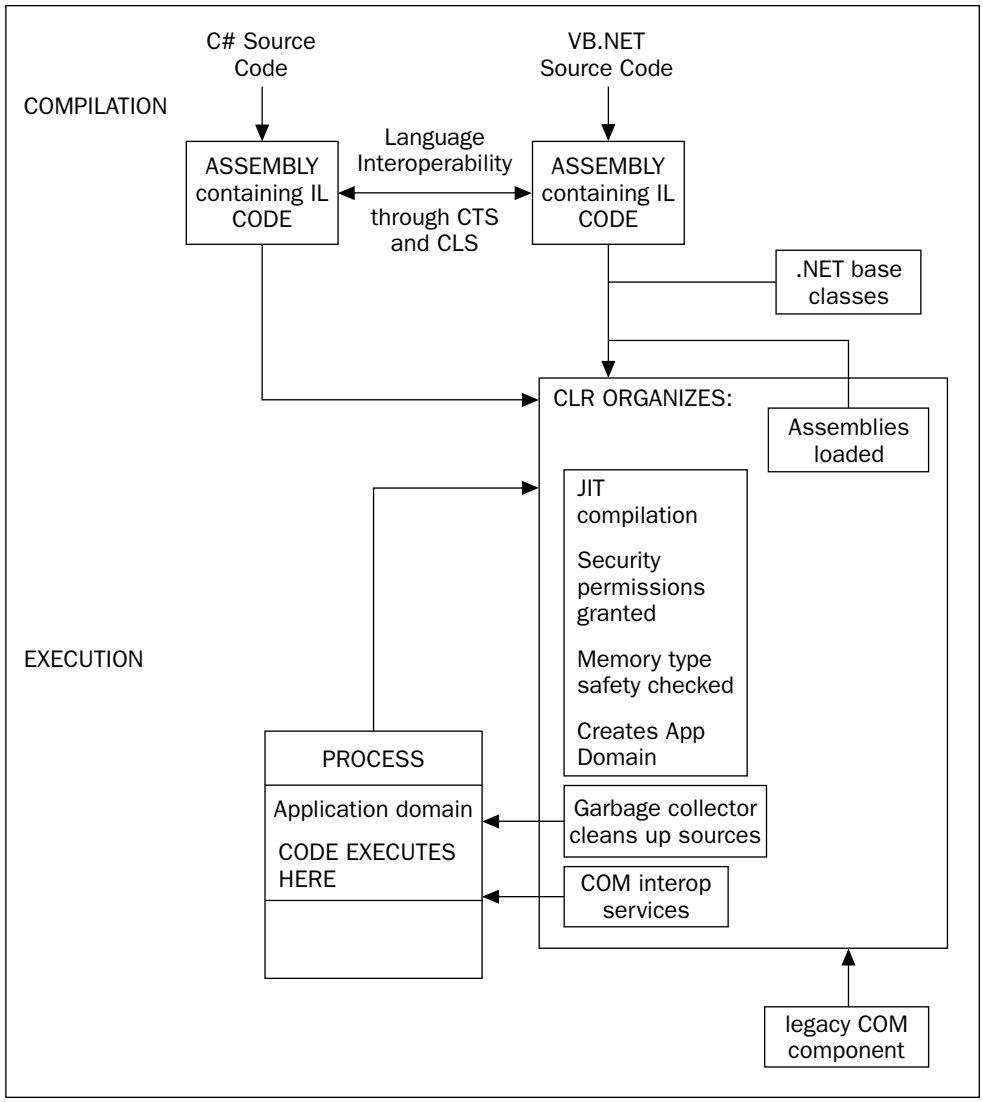
C# Source Code
VB.NET Source Code
COMPILATION
ASSEMBLY containing IL CODE
Language Interoperability
through CTS and CLS
ASSEMBLY containing IL CODE
.NET base classes
CLR ORGANIZES:
Assemblies loaded
JIT compilation
Security permissions granted
Memory type safety checked
Creates App Domain
EXECUTION
PROCESS
Application domain
CODE EXECUTES HERE
Garbage collector cleans up sources
COM interop services
legacy COM component

Figure 1-4