CHAPTER 1

The Big Picture

The Web has now existed for roughly two decades. In that time, the way websites look and work has changed dramatically. The way people *create* websites has also evolved. Today web pages can be written by hand (perhaps with the help of a design tool such as Adobe Dreamweaver), or they can be *programmed* using any one of a number of powerful platforms.

ASP.NET is Microsoft's web programming toolkit. It's a part of .NET, a cluster of technologies that are designed to help developers build a variety of applications. Developers can use the .NET Framework to build rich Windows applications, services that run quietly in the background, and even command-line tools. Developers write the code in one of several core .NET languages, such as C#, which is the language you'll use in this book.

In this chapter, you'll examine the technologies that underlie .NET. First you'll take a quick look at the history of web development and learn why the .NET Framework was created. Next you'll get a high-level overview of the parts of .NET and see how ASP.NET 4.5 fits into the picture.

The Evolution of Web Development

The Internet began in the late 1960s as an experiment. Its goal was to create a truly resilient information network—one that could withstand the loss of several computers without preventing the others from communicating. Driven by potential disaster scenarios (such as a nuclear attack), the US Department of Defense provided the initial funding.

The early Internet was mostly limited to educational institutions and defense contractors. It flourished as a tool for academic collaboration, allowing researchers across the globe to share information. In the early 1990s, modems were created that could work over existing phone lines, and the Internet began to open up to commercial users. In 1993, the first HTML browser was created, and the Internet revolution began.

Basic HTML

It would be difficult to describe early websites as web *applications*. Instead, the first generation of websites often looked more like brochures, consisting mostly of fixed HTML pages that needed to be updated by hand.

A basic HTML page is a little like a word-processing document—it contains formatted content that can be displayed on your computer, but it doesn't actually *do* anything. The following example shows HTML at its simplest, with a document that contains a heading and a single line of text:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Sample Web Page</title>
    </head>
    <body>
```

```
        <h1>Sample Web Page Heading</h1>
        <p>This is a sample web page.</p>
    </body>
</html>
```

Every respectable HTML document should begin with a *doctype*, a special code that indicates what flavor of HTML follows. Today the best choice is the following all-purpose doctype, which was introduced with HTML5 but works with even the oldest browsers around:

```
<!DOCTYPE html>
```

The rest of the HTML document contains the actual content. An HTML document has two types of content: the text and the elements (or tags) that tell the browser how to format it. The elements are easily recognizable, because they are designated with angle brackets (< >). HTML defines elements for different levels of headings, paragraphs, hyperlinks, italic and bold formatting, horizontal lines, and so on. For example, `<h1>Some Text</h1>` uses the <h1> element. This element tells the browser to display *Some Text* in the Heading 1 style, which uses a large, bold font. Similarly, `<p>This is a sample web page.</p>` creates a paragraph with one line of text. The <head> element groups the header information together and includes the <title> element with the text that appears in the browser window, while the <body> element groups together the actual document content that's displayed in the browser window.

Figure 1-1 shows this simple HTML page in a browser. Right now, this is just a fixed file (named SampleWebPage.htm) that contains HTML content. It has no interactivity, doesn't require a web server, and certainly can't be considered a web application.

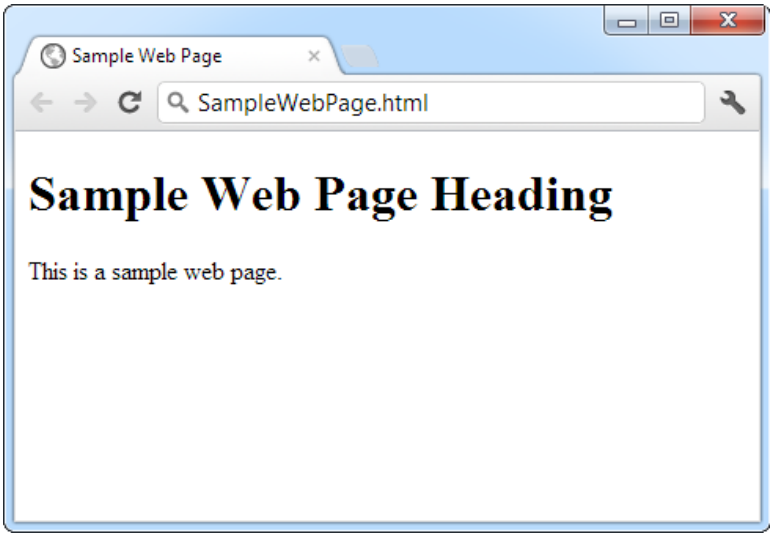

Figure 1-1. *Ordinary HTML*

■ **Tip** You don't need to master HTML to program ASP.NET web pages, although it's certainly a good start. For a quick introduction to HTML, refer to one of the excellent HTML tutorials on the Internet, such as `www.w3schools.com/html`. You'll also get a mini-introduction to HTML elements in Chapter 4.

HTML Forms

HTML 2.0 introduced the first seed of web programming with a technology called *HTML forms*. HTML forms expand HTML so that it includes not only formatting tags but also tags for graphical widgets, or *controls*. These controls include common ingredients such as drop-down lists, text boxes, and buttons. Here's a sample web page created with HTML form controls:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Sample Web Page</title>
    </head>
    <body>
        <form>
            <input type="checkbox" />
             This is choice #1<br />
            <input type="checkbox" />
             This is choice #2<br /><br />
            <input type="submit" value="Submit" />
        </form>
    </body>
</html>
```

In an HTML form, all controls are placed between the <form> and </form> tags. The preceding example includes two check boxes (represented by the ☐ element) and a button (represented by the <input type="submit"/> element). The
 element adds a line break between lines. In a browser, this page looks like Figure 1-2.

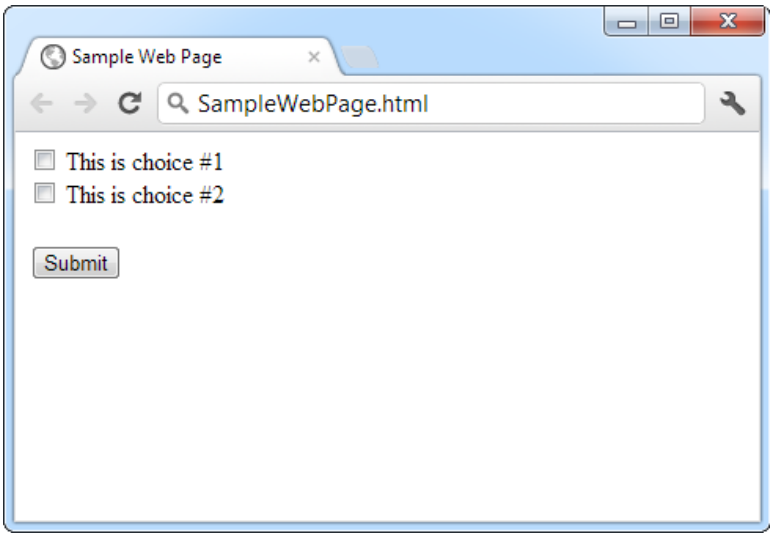

Figure 1-2. *An HTML form*

HTML forms allow web developers to design standard input pages. When the user clicks the Submit button on the page shown in Figure 1-2, all the data in the input controls (in this case, the two check boxes) is patched together into one long string of text and sent to the web server. On the server side, a custom application receives and processes the data. In other words, if the user selects a check box or enters some text, the application finds out about it after the form is submitted.

Amazingly enough, the controls that were created for HTML forms more than ten years ago are still the basic foundation that you'll use to build dynamic ASP.NET pages! The difference is the type of application that runs on the server side. In the past, when the user clicked a button on a form page, the information might have been e-mailed to a set account or sent to an application on the server that used the challenging Common Gateway Interface (CGI) standard. Today you'll work with the much more capable and elegant ASP.NET platform.

■ **Note** The latest version of the HTML language, HTML5, introduced a few new form controls for the first time in the history of the language. For the most part, ASP.NET doesn't use these, because they aren't supported in all browsers (and even the browsers that support them aren't always consistent). However, ASP.NET will use optional HTML5 frills, such as validation attributes (see Chapter 9), when they're appropriate. That's because browsers that don't support these features can ignore them, and the page will still work.

ASP.NET

Early web development platforms had two key problems. First, they didn't always scale well. As a result, popular websites would struggle to keep up with the demand of too many simultaneous users, eventually crashing or slowing to a crawl. Second, they provided little more than a bare-bones programming environment. If you wanted higher-level features, such as the ability to authenticate users or read a database, you needed to write pages of code from scratch. Building a web application this way was tedious and error-prone.

To counter these problems, Microsoft created higher-level development platforms—first ASP and then ASP.NET. These technologies allow developers to program dynamic web pages without worrying about the low-level implementation details. Even better, ASP.NET is stuffed full of sophisticated features, including tools for implementing security, managing data, storing user-specific information, and much more. And amazingly enough, it's even possible to program an ASP.NET page without knowing anything about HTML (although a little bit of HTML smarts will help you build your pages more quickly and effectively).

Server-Side and Client-Side Programming

ASP.NET is designed first and foremost as a *server-side* programming platform. That means that all ASP.NET code runs on the web server. When the ASP.NET code finishes running, the web server sends the user the final result—an ordinary HTML page that can be viewed in any browser.

Server-side programming isn't the only way to make an interactive web page. Another option is *client-side* programming, which asks the browser to download the code and execute it locally, on the client's computer. Just as there are a variety of server-side programming platforms, there are also various ways to perform client-side programming, from snippets of JavaScript code that can be embedded right inside the HTML of a web page, to plug-ins such as Adobe Flash and Microsoft Silverlight. Figure 1-3 shows the difference between the server-side and client-side models.

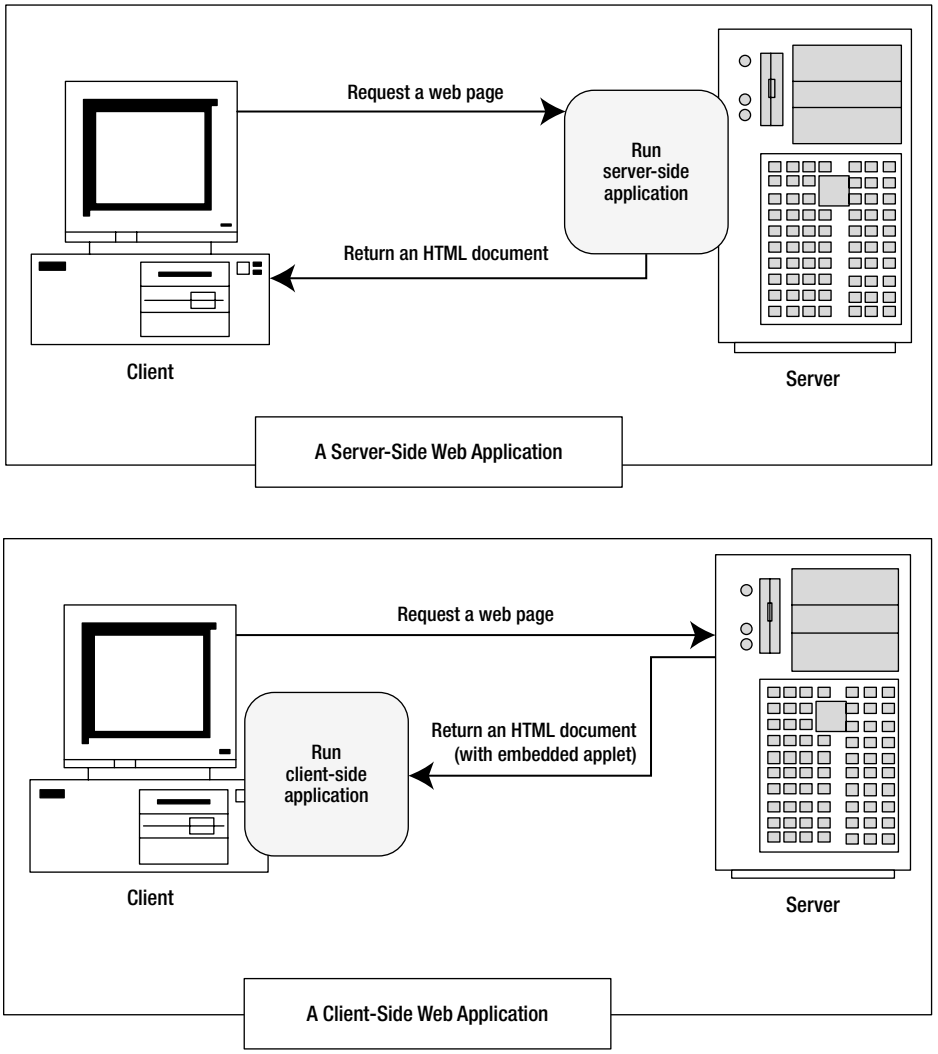

Figure 1-3. *Server-side and client-side web applications*

ASP.NET uses server-side programming to avoid several problems:

Isolation: Client-side code can't access server-side resources. For example, a client-side application has no easy way to read a file or interact with a database on the server (at least not without running into problems with security and browser compatibility).

Security: End users can view client-side code. And once malicious users understand how an application works, they can often tamper with it.

Thin clients: In today's world, web-enabled devices such as tablets and smartphones are everywhere. These devices usually have some sort of built-in web browsing ability, but they may not support client-side programming platforms such as Flash or Silverlight.

In recent years, there's been a renaissance in client programming, particularly with JavaScript. Nowadays developers create client-side applications that communicate with a web server to fetch information and perform tasks that wouldn't be possible if the applications were limited to the local computer. Fortunately, ASP.NET takes advantage of this change in two ways:

JavaScript frills: In some cases, ASP.NET allows you to combine the best of client-side programming with server-side programming. For example, the best ASP.NET controls can "intelligently" detect the features of the client browser. If the browser supports JavaScript, these controls will return a web page that incorporates JavaScript for a richer, more responsive user interface. You'll see a good example of this technique with validation in Chapter 9.

ASP.NET's Ajax features: Ajax is a set of JavaScript techniques used to create fast, responsive pages with dynamic content. In Chapter 25, you'll learn how ASP.NET lets you benefit from many of the advantages of Ajax with none of the complexity.

However, it's important to understand one fundamental fact. No matter what the capabilities of the browser, the C# code that you write is always executed on the server. The client-side frills are just the icing on the cake.

■ **Tip** It's worth noting that ASP.NET is not the best platform for writing complex, app-like client-side programs—at least not on its own. For example, ASP.NET isn't much help to developers who want to build a real-time browser-based game or the next Google Maps. If this is what you want, it's largely up to you to add the huge amounts of complex JavaScript that you need to your ASP.NET web forms. However, if you'd prefer to create an e-commerce hub or a business site, or a site that displays and manages large amounts of data, ASP.NET is the perfect fit.

The .NET Framework

As you've already learned, the .NET Framework is really a cluster of several technologies:

The .NET languages: These include Visual Basic, C#, F#, and C++, although third-party developers have created hundreds more.

The Common Language Runtime (CLR): This is the engine that executes all .NET programs and provides automatic services for these applications, such as security checking, memory management, and optimization.

The .NET Framework class library: The class library collects thousands of pieces of prebuilt functionality that you can "snap in" to your applications. These features are sometimes organized into technology sets, such as ADO.NET (the technology for creating database applications) and Windows Presentation Foundation (WPF, the technology for creating desktop user interfaces).

ASP.NET: This is the engine that hosts the web applications you create with .NET, and supports almost any feature from the .NET Framework class library. ASP.NET also includes a set of web-specific services, such as secure authentication and data storage.

Visual Studio: This optional development tool contains a rich set of productivity and debugging features. Visual Studio includes the complete .NET Framework, so you won't need to download it separately.

Sometimes the division between these components isn't clear. For example, the term *ASP.NET* is sometimes used in a narrow sense to refer to the portion of the .NET class library used to design web pages. On the other hand, ASP.NET also refers to the whole topic of .NET web applications, which includes .NET languages and many fundamental pieces of the class library that aren't web-specific. (That's generally the way we use the term in this book. Our exhaustive examination of ASP.NET includes .NET basics, the C# language, and topics that any .NET developer could use, such as component-based programming and database access.)

Figure 1-4 shows the .NET class library and CLR—the two fundamental parts of .NET.

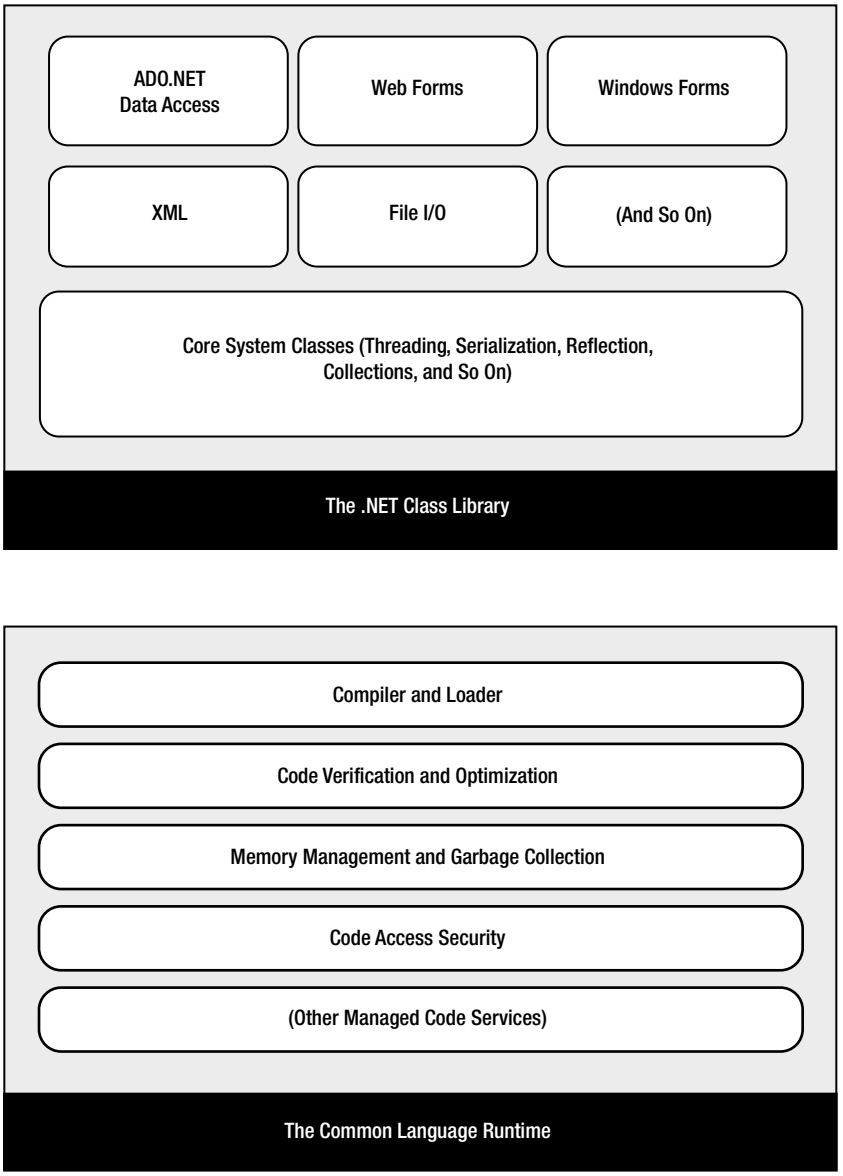

Figure 1-4. *The .NET Framework*

In the remainder of this chapter, you'll take a quick look at the ingredients that make up the .NET Framework.

C#, VB, and the .NET Languages

This book uses the Visual Basic language, which enables you to create readable, modern code. The .NET version of VB is similar in syntax to older flavors of VB that you may have encountered, including "classic" VB 6 and the Visual Basic for Applications (VBA) language often used to write macros in Microsoft Office programs such as Word and Excel. However, you cannot convert classic VB into the .NET flavor of Visual Basic, just as you cannot convert C++ into C#.

This book uses C#, Microsoft's .NET language of preference. C# resembles Java, JavaScript, and C++ in syntax, so programmers who have coded in one of these languages will quickly feel at home.

Interestingly, VB and C# are quite similar. Though the syntax is different, both VB and C# use the .NET class library and are supported by the CLR. In fact, almost any block of C# code can be translated, line by line, into an equivalent block of VB code (and vice versa). An occasional language difference pops up, but for the most part, a developer who has learned one .NET language can move quickly and efficiently to another. There are even software tools that translate C# and VB code automatically (see `http://converter.telerik.com` or `http://tangiblesoftwaresolutions.com` for examples).

In short, both VB and C# are elegant, modern languages that are ideal for creating the next generation of web applications.

■ **Note** .NET 1.0 introduced completely new languages. However, the changes in subsequent versions of .NET have been more subtle. Although the version of C# in .NET 4.5 adds a few new features, most parts of the language remain unchanged. In Chapter 2 and Chapter 3, you'll sort through the syntax of C# and learn the basics of object-oriented programming.

Intermediate Language

All the .NET languages are compiled into another lower-level language before the code is executed. This lower-level language is the *Common Intermediate Language* (CIL, or just IL). The CLR, the engine of .NET, uses only IL code. Because all .NET languages are based on IL, they all have profound similarities. This is the reason that the VB and C# languages provide essentially the same features and performance. In fact, the languages are so compatible that a web page written with C# can use a VB component in the same way it uses a C# component, and vice versa.

The .NET Framework formalizes this compatibility with something called the *Common Language Specification* (CLS). Essentially, the CLS is a contract that, if respected, guarantees that a component written in one .NET language can be used in all the others. One part of the CLS is the *common type system* (CTS), which defines the rules for data types such as strings, numbers, and arrays that are shared in all .NET languages. The CLS also defines object-oriented ingredients such as classes, methods, events, and quite a bit more. For the most part, .NET developers don't need to think about how the CLS works, even though they rely on it every day.

Figure 1-5 shows how the .NET languages are compiled to IL. Every EXE or DLL file that you build with a .NET language contains IL code. This is the file you deploy to other computers. In the case of a web application, you deploy your compiled code to a live web server.

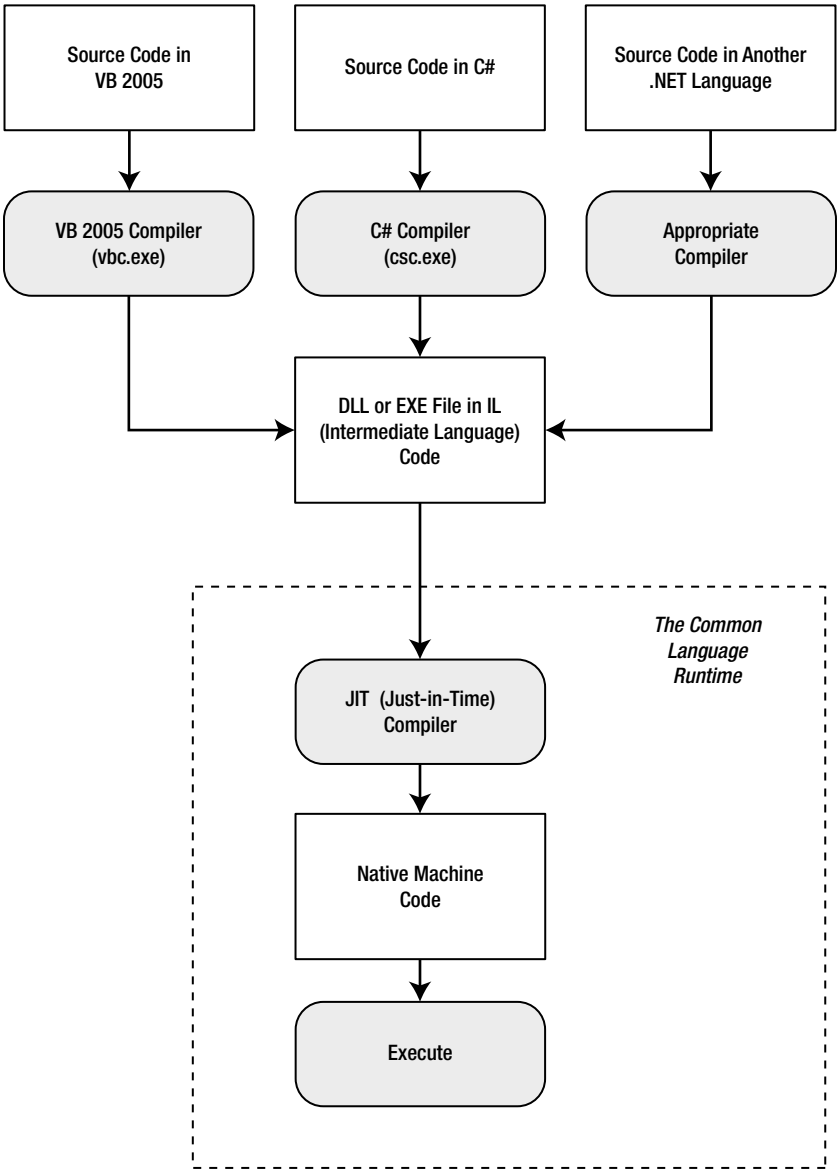

Figure 1-5. *Language compilation in .NET*

The CLR runs only IL code, which means it has no idea which .NET language you originally used. Notice, however, that the CLR performs another compilation step—it takes the IL code and transforms it to native machine language code that's appropriate for the current platform. This step occurs when the application is launched, just before the code is executed. In an ASP.NET application, these machine-specific files are cached while the web application is running so they can be reused, ensuring optimum performance.

■ **Note** You might wonder why .NET compilers don't compile straight to machine code. The reason is that the machine code depends on several factors, including the CPU. If you compile an application to machine code on one computer, there's no guarantee that it will work on another computer with a different processor.

The Common Language Runtime

The CLR is the engine that supports all the .NET languages. All .NET code runs inside the CLR. This is true whether you're running a Windows application or a web service. For example, when a client requests an ASP.NET web page, the ASP.NET service runs inside the CLR environment, executes your code, and creates a final HTML page to send to the client.

Not only does the CLR execute code, but it also provides a whole set of related services such as code verification, optimization, and object management. The implications of the CLR are wide-ranging:

Deep language integration: VB and C#, like all .NET languages, compile to IL. In other words, the CLR makes no distinction between different languages—in fact, it has no way of knowing what language was used to create an executable. This is far more than mere language compatibility; it's language *integration*.

Side-by-side execution: The CLR also has the ability to load more than one version of a component at a time. In other words, you can update a component many times, and the correct version will be loaded and used for each application. As a side effect, multiple versions of the .NET Framework can be installed, meaning that you're able to upgrade to new versions of ASP.NET without replacing the current version or needing to rewrite your applications.

Fewer errors: Whole categories of errors are impossible with the CLR. For example, the CLR prevents many memory mistakes that are possible with lower-level languages such as C++.

Along with these truly revolutionary benefits, the CLR has some potential drawbacks. Here are two issues that are sometimes raised by new developers but aren't always answered:

Performance: A typical ASP.NET application is extremely fast, because ASP.NET code is compiled to machine code before it's executed. However, processor-crunching algorithms still can't match the blinding speed of well-written C++ code, because the CLR imposes some additional overhead. Generally, this is a factor only in a few performance-critical high-workload applications (such as real-time games). With high-volume web applications, the potential bottlenecks are rarely processor-related but are usually tied to the speed of an external resource such as a database or the web server's file system. With ASP.NET caching and some well-written database code, you can ensure excellent performance for any web application.

Code transparency: IL is much easier to disassemble, meaning that if you distribute a compiled application or component, other programmers may have an easier time determining how your code works. This isn't much of an issue for ASP.NET applications, which aren't distributed but are hosted on a secure web server.

The .NET Class Library

The .NET class library is a giant repository of classes that provide prefabricated functionality for everything from reading an XML file to sending an e-mail message. If you've had any exposure to Java, you may already be familiar with the idea of a class library. However, the .NET class library is more ambitious and comprehensive than just about any other programming framework. Any .NET language can use the .NET class library's features by interacting with the right objects. This helps encourage consistency among different .NET languages and removes the need to install numerous components on your computer or web server.

Some parts of the class library include features you'll never need to use in web applications (such as the classes used to create desktop applications with Windows interfaces). Other parts of the class library are targeted directly at web development. Still more classes can be used in various programming scenarios and aren't specific to web or Windows development. These include the base set of classes that define common variable types and the classes for data access, to name just a few. You'll explore the .NET Framework throughout this book.

You can think of the class library as a well-stocked programmer's toolkit. Microsoft's philosophy is that it will provide the tedious infrastructure so that application developers need only to write business-specific code. For example, the .NET Framework deals with thorny issues such as database transactions and concurrency, making sure that hundreds or thousands of simultaneous users can request the same web page at once. You just add the logic needed for your specific application.

Visual Studio

The last part of .NET is the Visual Studio development tool, which provides a rich environment where you can rapidly create advanced applications. Although in theory you could create an ASP.NET application without Visual Studio (for example, by writing all the source code in a text editor and compiling it with .NET's command-line compilers), this task would be tedious, painful, and prone to error. For that reason, all professional ASP.NET developers use a design tool such as Visual Studio.

Some of the features of Visual Studio include the following:

Page design: You can create an attractive page with drag-and-drop ease by using Visual Studio's integrated web form designer. You don't need to understand HTML.

Automatic error detection: You could save hours of work when Visual Studio detects and reports an error before you run your application. Potential problems are underlined, just like the "spell-as-you-go" feature found in many word processors.

Debugging tools: Visual Studio retains its legendary debugging tools, which allow you to watch your code in action and track the contents of variables. And you can test web applications just as easily as any other application type, because Visual Studio has a built-in web server that works just for debugging.

IntelliSense: Visual Studio provides statement completion for recognized objects and automatically lists information such as function parameters in helpful tooltips.

You'll learn about all these features in Chapter 4, when you consider the latest version of Visual Studio. It's also important to note that Visual Studio is available in several editions:

Visual Studio Express for Web: This is a completely free version of Visual Studio that's surprising capable. Its main limitation is that it allows you to build web applications and components only, not other types of .NET programs (for example, Windows applications).

■ **Tip** To download Visual Studio Express for Web, go to www.microsoft.com/express/downloads. To compare the differences between Visual Studio versions, check out www.microsoft.com/visualstudio/11/en-us/products/compare.

Visual Studio Professional: This is the leanest full version of Visual Studio. It has all the features you need to build any type of .NET application (Windows or web).

Visual Studio Premium or Ultimate: These versions increase the cost and pile on more tools and frills (which aren't discussed in this book). For example, they incorporate features for automated testing and version control, which helps team members coordinate their work on large projects.

■ **Note** You'll be able to run all the examples in this book by using any version of Visual Studio, including the free Visual Studio Express for Web.

The Last Word

This chapter presented a high-level overview that gave you your first taste of ASP.NET and the .NET Framework. You also looked at how web development has evolved, from the basic HTML forms standard to the modern ASP.NET platform.

In the next chapter, you'll get a comprehensive overview of the C# language.

CHAPTER 2

The C# Language

Before you can create an ASP.NET application, you need to choose a .NET language in which to program it. Both VB and C# are powerful, modern languages, and you won't go wrong using either of them to code your web pages. Often the choice is simply a matter of personal preference or your work environment. For example, if you've already programmed in a language that uses C-like syntax (for example, Java), you'll probably be most comfortable with C#. Or if you've spent a few hours writing Microsoft Excel macros in VBA, you might prefer the natural style of Visual Basic. Many developers become fluent in both.

This chapter presents an overview of the C# language. You'll learn about the data types you can use, the operations you can perform, and the code you'll need to define functions, loops, and conditional logic. This chapter assumes that you have programmed before and are already familiar with most of these concepts—you just need to see how they're implemented in C#.

If you've programmed with a similar language such as Java, you might find that the most beneficial way to use this chapter is to browse through it without reading every section. This approach will give you a general overview of C#. You can then return to this chapter later as a reference when needed. But remember, though you can program an ASP.NET application without mastering all the language details, this deep knowledge is often what separates the casual programmer from the true programming guru.

Note The examples in this chapter show individual lines and code snippets. You won't be able to use these code snippets in an application until you've learned about objects and .NET types. But don't despair—the next chapter builds on this information, fills in the gaps, and presents an ASP.NET example for you to try.

The .NET Languages

The .NET Framework ships with two core languages that are commonly used for building ASP.NET applications: C# and VB. These languages are, to a large degree, functionally equivalent. Microsoft has worked hard to eliminate language conflicts in the .NET Framework. These battles slow down adoption, distract from the core framework features, and make it difficult for the developer community to solve problems together and share solutions. According to Microsoft, choosing to program in C# instead of VB is just a lifestyle choice and won't affect the performance, interoperability, feature set, or development time of your applications. Surprisingly, this ambitious claim is essentially true.

.NET also allows other third-party developers to release languages that are just as feature-rich as C# or VB. These languages (which include Eiffel, Pascal, and even COBOL) "snap in" to the .NET Framework effortlessly. In fact, if you want to install another .NET language, all you need to do is copy the compiler to your computer and add a line to register it in a configuration file. Typically, a setup program would perform these steps for you automatically. Once installed, the new compiler can transform your code creations into a sequence of Intermediate Language (IL) instructions, just as the VB and C# compilers do with VB and C# code.

IL is the only language that the Common Language Runtime (CLR) recognizes. When you create the code for an ASP.NET web form, it's changed into IL using the C# compiler (`csc.exe`) or the VB compiler (). Although you can perform the compilation manually, you're more likely to let ASP.NET handle it automatically when a web page is requested.

C# Language Basics

New C# programmers are sometimes intimidated by the quirky syntax of the language, which includes special characters such as semicolons (`;`), curly braces (`{}`), and backward slashes (`\`). Fortunately, once you get accustomed to C#, these details will quickly melt into the background. In the following sections, you'll learn about four general principles you need to know about C# before you learn any other concepts.

Case Sensitivity

Some languages are *case-sensitive*, while others are not. Java, C, C++, and C# are all examples of case-sensitive languages. VB is not. This difference can frustrate former VB programmers who don't realize that keywords, variables, and functions must be entered with the proper case. For example, if you try to create a conditional statement in C# by entering *If* instead of *if*, your code will not be recognized, and the compiler will flag it with an error when you try to build your application.

C# also has a definite preference for lowercase words. Keywords—such as `if`, `for`, `foreach`, `while`, `typeof`, and so on—are always written in lowercase letters. When you define your own variables, it makes sense to follow the conventions used by other C# programmers and the .NET Framework class library. That means you should give private variables names that start with a lowercase letter and give public variables names that start with an initial capital letter. For example, you might name a private variable MyNumber in VB and myNumber in C#. Of course, you don't need to follow this style as long as you make sure you use the same capitalization consistently.

■ **Note** If you're designing code that other developers might see (for example, you're creating components that you want to sell to other companies), coding standards are particularly important. But even if you aren't, clear and consistent coding is a good habit that will make it easier for you to understand the code you've written months (or even years!) later. You can find a good summary of best practices in the "IDesign C# Coding Standard" white paper by Juval Lowy, which is available at `www.idesign.net`.

Commenting

Comments are lines of descriptive text that are ignored by the compiler. C# provides two basic types of comments.

The first type is the single-line comment. In this case, the comment starts with two forward slashes and continues for the entire current line:

```
// A single-line C# comment.
```

Optionally, C# programmers can use `/*` and `*/` comment brackets to indicate multiple-line comments:

```
/* A multiple-line
   C# comment. */
```

Multiple-line comments are often used to quickly disable an entire block of code. This trick is called *commenting out* your code:

```
/*
    ...Any code here is ignored...
*/
```

This way, the code won't be executed, but it will still remain in your source code file if you need to refer to it or use it later.

■ **Tip** It's easy to lose track of the `/*` and `*/` comment brackets in your source code file. However, you won't forget that you've disabled a portion of your code, because Visual Studio displays all comments and commented-out code in green text.

C# also includes an XML-based commenting syntax that you can use to describe your code in a standardized way. With XML comments, you use special tags that indicate the portion of code that the comment applies to. Here's an example of a comment that provides a summary for an entire application:

```
/// <summary>
/// This application provides web pages
/// for my e-commerce site.
/// </summary>
```

XML comments always start with three slashes. The benefit of XML-based comments is that automated tools (including Visual Studio) can extract the comments from your code and use them to build help references and other types of documentation. For more information about XML comments, you can refer to an excellent MSDN article at http://msdn.microsoft.com/magazine/cc302121.aspx. And if you're new to XML syntax in general, you'll learn about it in detail in Chapter 18.

Statement Termination

C# uses a semicolon (`;`) as a *statement-termination character*. Every statement in C# code must end with this semicolon, except when you're defining a block structure. (Examples of such statements include methods, conditional statements, and loops, which are three types of code ingredients that you'll learn about later in this chapter.) By omitting the semicolon, you can easily split a statement of code over multiple physical lines. You just need to remember to put the semicolon at the end of the last line to end the statement.

The following code snippet demonstrates four equivalent ways to perform the same operation (adding three numbers together):

```
// A code statement on a single line.
myValue = myValue1 + myValue2 + myValue3;

// A code statement split over two lines.
myValue = myValue1 + myValue2 +
          myValue3;

// A code statement split over three lines.
myValue = myValue1 +
          myValue2 +
          myValue3;
```

```
// Two code statements in a row.
myValue = myValue1 + myValue2;
myValue = myValue + myValue3;
```

As you can see in this example, C# gives you a wide range of freedom to split your statement in whatever way you want. The general rule of thumb is to make your code as readable as possible. Thus, if you have a long statement, spread the statement over several lines so it's easier to read. On the other hand, if you have a complex code statement that performs several operations at once, you can spread the statement over several lines or separate your logic into multiple code statements to make it clearer.

Blocks

The C#, Java, and C languages all rely heavily on curly braces—parentheses with a little more attitude: {}. You can find the curly braces to the right of most keyboards (next to the P key); they share a key with the square brackets: [].

Curly braces group multiple code statements together. Typically, you'll group code statements because you want them to be repeated in a loop, executed conditionally, or grouped into a function. These are all *block structures*, and you'll see all these techniques in this chapter. But in each case, the curly braces play the same role, which makes C# simpler and more concise than other languages that need a different syntax for each type of block structure.

```
{
    // Code statements go here.
}
```

Variables and Data Types

As with all programming languages, you keep track of data in C# by using variables. *Variables* can store numbers, text, dates, and times, and they can even point to full-fledged objects.

When you declare a variable, you give it a name and specify the type of data it will store. To declare a local variable, you start the line with the data type, followed by the name you want to use. A final semicolon ends the statement.

```
// Declare an integer variable named errorCode.
int errorCode;

// Declare a string variable named myName.
string myName;
```

■ **Note** Remember, in C# the variables *name* and *Name* aren't equivalent! To confuse matters even more, C# programmers sometimes use this fact to their advantage—by using multiple variables that have the same name but with different capitalization. This technique is sometimes useful when distinguishing between private and public variables in a class (as demonstrated in Chapter 3), but you should avoid it if there's any possibility for confusion.

Every .NET language uses the same variable data types. Different languages may provide slightly different names (for example, a VB Integer is the same as a C# int), but the CLR makes no distinction—in fact, they are just two different names for the same base data type (in this case, it's System.Int32). This design allows for deep language integration. Because languages share the same core data types, you can easily use objects written in one .NET language in an application written in another .NET language. No data type conversions are required.

■ **Note** All .NET languages have the same data types because they all adhere to the common type system (CTS), a Microsoft-designed ECMA standard that sets the ground rules that all .NET languages must follow when dealing with data.

To create this common data type system, Microsoft cooked up a set of basic data types, which are provided in the .NET class library. Table 2-1 lists the most important core data types.

Table 2-1. *Common Data Types*

C# Name	VB Name	.NET Type Name	Contains
byte	Byte	Byte	An integer from 0 to 255.
short	Short	Int16	An integer from –32,768 to 32,767.
int	Integer	Int32	An integer from –2,147,483,648 to 2,147,483,647.
long	Long	Int64	An integer from about –9.2e18 to 9.2e18.
float	Single	Single	A single-precision floating-point number from approximately –3.4e38 to 3.4e38 (for big numbers) or –1.5e-45 to 1.5e-45 (for small fractional numbers).
double	Double	Double	A double-precision floating-point number from approximately –1.8e308 to 1.8e308 (for big numbers) or –5.0e-324 to 5.0e-324 (for small fractional numbers).
decimal	Decimal	Decimal	A 128-bit fixed-point fractional number that supports up to 28 significant digits.
char	Char	Char	A single Unicode character.
string	String	String	A variable-length series of Unicode characters.
bool	Boolean	Boolean	A true or false value.
*	Date	DateTime	Represents any date and time from 12:00:00 AM, January 1 of the year 1 in the Gregorian calendar, to 11:59:59 PM, December 31 of the year 9999. Time values can resolve values to 100 nanosecond increments. Internally, this data type is stored as a 64-bit integer.
*	*	TimeSpan	Represents a period of time, as in ten seconds or three days. The smallest possible interval is 1 *tick* (100 nanoseconds).
object	Object	Object	The ultimate base class of all .NET types. Can contain any data type or object. (You'll take a much closer look at objects in Chapter 3.)

** If the language does not provide an alias for a given type, you must use the .NET type name.*

You can also declare a variable by using the type name from the .NET class library. This approach produces identical variables. It's also a requirement when the data type doesn't have an alias built into the language. For example, you can rewrite the earlier example that used C# data type names with this code snippet that uses the class library names:

```
System.Int32 errorCode;
System.String myName;
```

This code snippet uses fully qualified type names that indicate that the Int32 data type and the String data type are found in the System namespace (along with all the most fundamental types). In Chapter 3, you'll learn about types and namespaces in more detail.

WHAT'S IN A NAME? NOT THE DATA TYPE!

If you have some programming experience, you'll notice that the preceding examples don't use variable prefixes. Many longtime C/C++ and VB programmers are in the habit of adding a few characters to the start of a variable name to indicate its data type. In .NET, this practice is discouraged, because data types can be used in a much more flexible range of ways without any problem, and most variables hold references to full objects anyway. In this book, variable prefixes aren't used, except for web controls, where it helps to distinguish among lists, text boxes, buttons, and other common user interface elements. In your own programs, you should follow a consistent (typically companywide) standard that may or may not adopt a system of variable prefixes.

Assignment and Initializers

After you've declared your variables, you can freely assign values to them, as long as these values have the correct data type. Here's the code that shows this two-step process:

```
// Declare variables.
int errorCode;
string myName;

// Assign values.
errorCode = 10;
myName = "Matthew";
```

You can also assign a value to a variable in the same line that you declare it. This example compresses the preceding four lines of code into two:

```
int errorCode = 10;
string myName = "Matthew";
```

C# safeguards you from errors by restricting you from using uninitialized variables. For example, the following code causes an error when you attempt to compile it:

```
int number;             // Number is uninitialized.
number = number + 1;    // This causes a compile error.
```

The proper way to write this code is to explicitly initialize the number variable to an appropriate value, such as 0, before using it:

```
int number = 0;         // Number now contains 0.
number = number + 1;    // Number now contains 1.
```

C# also deals strictly with data types. For example, the following code statement won't work as written:

```
decimal myDecimal = 14.5;
```

The problem is that the literal 14.5 is automatically interpreted as a double, and you can't convert a double to a decimal without using casting syntax, which is described later in this chapter. To get around this problem, C# defines a few special characters that you can append to literal values to indicate their data type so that no conversion will be required. These characters are as follows:

- `M` (decimal)
- `D` (double)
- `F` (float)
- `L` (long)

For example, you can rewrite the earlier example by using the decimal indicator as follows:

```
decimal myDecimal = 14.5M;
```

■ **Note** In this example, an uppercase `M` is used, but you can substitute a lowercase `m` in its place. Data type indicators are one of the few details that aren't case-sensitive in C#.

Interestingly, if you're using code like this to declare and initialize your variable in one step, and if the C# compiler can determine the right data type based on the value you're using, you don't need to specify the data type. Instead, you can use the all-purpose `var` keyword in place of the data type. That means the previous line of code is equivalent to this:

```
var myDecimal = 14.5M;
```

Here, the compiler realizes that a decimal data type is the most appropriate choice for the myDecimal variable and uses that data type automatically. There is no performance difference. The myDecimal variable that you create using an inferred data type behaves in exactly the same way as a myDecimal variable created with an explicit data type. In fact, the low-level code that the compiler generates is identical. The only difference is that the `var` keyword saves some typing.

Many C# programmers feel uneasy with the `var` keyword because it makes code less clear. However, the `var` keyword is a more useful shortcut when creating objects, as you'll see in the next chapter.

Strings and Escaped Characters

C# treats text a little differently than other languages such as VB. It interprets any embedded backslash (`\`) as the start of a special character sequence. For example, `\n` means add a new line (carriage return). The most useful character literals are as follows:

- `\"` (double quote)
- `\n` (new line)
- `\t` (horizontal tab)
- `\\` (backward slash)

You can also insert a special character based on its hex code by using the syntax `\x250`. This inserts a single character with hex value 250 (which is a character that looks like an upside-down letter *a*).

Note that in order to specify the backslash character (for example, in a directory name), you require two slashes. Here's an example:

```
// A C# variable holding the
// c:\MyApp\MyFiles path.
string path = "c:\\MyApp\\MyFiles";
```

Alternatively, you can turn off C# escaping by preceding a string with an @ symbol, as shown here:

```
string path = @"c:\MyApp\MyFiles";
```

Arrays

Arrays allow you to store a series of values that have the same data type. Each individual value in the array is accessed by using one or more index numbers. It's often convenient to picture arrays as lists of data (if the array has one dimension) or grids of data (if the array has two dimensions). Typically, arrays are laid out contiguously in memory.

All arrays start at a fixed lower bound of 0. This rule has no exceptions. When you create an array in C#, you specify the number of elements. Because counting starts at 0, the highest index is actually one less than the number of elements. (In other words, if you have three elements, the highest index is 2.)

```
// Create an array with four strings (from index 0 to index 3).
// You need to initialize the array with the
// new keyword in order to use it.
string[] stringArray = new string[4];

// Create a 2x4 grid array (with a total of eight integers).
int[,] intArray = new int[2, 4];
```

By default, if your array includes simple data types, they are all initialized to default values (0 or false), depending on whether you are using some type of number or a Boolean variable. But if your array consists of strings or another object type, it's initialized with null references. (For a more comprehensive discussion that outlines the difference between simple value types and reference types, see Chapter 3.)

You can also fill an array with data at the same time that you create it. In this case, you don't need to explicitly specify the number of elements, because .NET can determine it automatically:

```
// Create an array with four strings, one for each number from 1 to 4.
string[] stringArray = {"1", "2", "3", "4"};
```

The same technique works for multidimensional arrays, except that two sets of curly braces are required:

```
// Create a 4x2 array (a grid with four rows and two columns).
int[,] intArray = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
```

Figure 2-1 shows what this array looks like in memory.

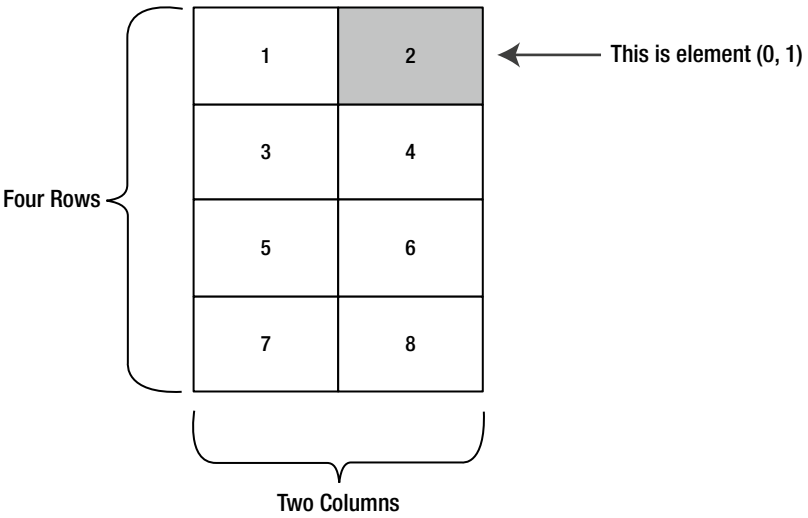

Figure 2-1. *A sample array of integers*

To access an element in an array, you specify the corresponding index number in square brackets: [ ]. Array indices are always zero-based. That means myArray[0] accesses the first cell in a one-dimensional array, myArray[1] accesses the second cell, and so on.

```
int[] intArray = {1, 2, 3, 4};
int element = intArray[2];    // element is now set to 3.
```

In a two-dimensional array, you need two index numbers:

```
int[,] intArray = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};

// Access the value in row 0 (first row), column 1 (second column).
int element = intArray[0, 1];    // element is now set to 2.
```

The ArrayList

C# arrays do not support redimensioning. This means that after you create an array, you can't change its size. Instead, you would need to create a new array with the new size and copy values from the old array to the new, which would be a tedious process. However, if you need a dynamic array-like list, you can use one of the collection classes provided to all .NET languages through the .NET class library. One of the simplest collection classes that .NET offers is the ArrayList, which supports any type of object and always allows dynamic resizing. Here's a snippet of C# code that uses an ArrayList:

```
// Create an ArrayList object. It's a collection, not an array,
// so the syntax is slightly different.
ArrayList dynamicList = new ArrayList();

// Add several strings to the list.
// The ArrayList is not strongly typed, so you can add any data type
// although it's simplest if you store just one type of object
// in any given collection.
```

```
dynamicList.Add("one");
dynamicList.Add("two");
dynamicList.Add("three");

// Retrieve the first string. Notice that the object must be converted to a
// string, because there's no way for .NET to be certain what it is.
string item = Convert.ToString(dynamicList[0]);
```

You'll learn more about the ArrayList and other collections in Chapter 3.

■ **Tip** In many cases, it's easier to dodge counting issues and use a full-fledged collection rather than an array. Collections are generally better suited to modern object-oriented programming and are used extensively in ASP.NET. The .NET class library provides many types of collection classes, including simple collections, sorted lists, key-indexed lists (dictionaries), and queues. You'll see examples of collections throughout this book.

Enumerations

An *enumeration* is a group of related constants, each of which is given a descriptive name. Each value in an enumeration corresponds to a preset integer. In your code, however, you can refer to an enumerated value by name, which makes your code clearer and helps prevent errors. For example, it's much more straightforward to set the border of a label to the enumerated value BorderStyle.Dashed rather than the obscure numeric constant 3. In this case, Dashed is a value in the BorderStyle enumeration, and it represents the number 3.

■ **Note** Just to keep life interesting, the word *enumeration* has more than one meaning. As described in this section, enumerations are sets of constant values. However, programmers often talk about the process of *enumerating*, which means to loop, or *iterate*, over a collection. For example, it's common to talk about enumerating over all the characters of a string (which means looping through the string and examining each character in a separate pass).

Here's an example of an enumeration that defines different types of users:

```
// Define an enumeration type named UserType with three possible values.
enum UserType
{
    Admin,
    Guest,
    Other
}
```

Now you can use the UserType enumeration as a special data type that is restricted to one of three possible values. You assign or compare the enumerated value by using the dot notation shown in the following example:

```
// Create a new value and set it equal to the UserType.Admin constant.
UserType newUserType = UserType.Admin;
```

Internally, enumerations are maintained as numbers. In the preceding example, 0 is automatically assigned to Admin, 1 to Guest, and 2 to Other. You can set a number directly in an enumeration variable, although this can lead to an undetected error if you use a number that doesn't correspond to one of the defined values.

Clearly, enumerations create more-readable code. They also simplify coding, because after you type in the enumeration type name (ErrorCode) and add the dot (`.`), Visual Studio will pop up a list of possible values by using IntelliSense.

■ **Tip** Enumerations are used widely in .NET. You won't need to create your own enumerations to use in ASP.NET applications, unless you're designing your own components. However, the concept of enumerated values is extremely important, because the .NET class library uses it extensively. For example, you set colors, border styles, alignment, and various other web control styles by using enumerations provided in the .NET class library.

Variable Operations

You can use all the standard types of variable operations in C#. When working with numbers, you can use various math symbols, as listed in Table 2-2. C# follows the conventional order of operations, performing exponentiation first, followed by multiplication and division and then addition and subtraction. You can also control order by grouping subexpressions with parentheses:

```
int number;

number = 4 + 2 * 3;
// number will be 10.

number = (4 + 2) * 3;
// number will be 18.
```

Table 2-2. *Arithmetic Operations*

Operator	Description	Example
+	Addition	`1 + 1 = 2`
-	Subtraction	`5 - 2 = 3`
*	Multiplication	`2 * 5 = 10`
/	Division	`5.0 / 2 = 2.5`
%	Gets the remainder left after integer division	`7 % 3 = 1`

Division can sometimes cause confusion in C#. If you divide one integer by another integer, C# performs integer division. That means it automatically discards the fractional part of the answer and returns the whole part as an integer. For example, if you divide 5 by 2, you'll end up with 2 instead of 2.5.

The solution is to explicitly indicate that one of your numbers is a fractional value. For example, if you replace 5 with *5 M*, C# will treat the 5 as a decimal. If you replace 5 with *5.0*, C# will treat it as a double. Either way, the division will return the expected value of 2.5. Of course, this problem doesn't occur very often in real-world code, because then you're usually dividing one *variable* by another. As long as your variables aren't integers, it doesn't matter what number they contain.

The operators in Table 2-2 are designed for manipulating numbers. However, C# also allows you to use the addition operator (+) to join two strings:

```
// Join three strings together.
myName = firstName + " " + lastName;
```

In addition, C# provides special shorthand assignment operators. Here are a few examples:

```
// Add 10 to myValue. This is the same as myValue = myValue + 10;
myValue += 10;

// Multiple myValue by 3. This is the same as myValue = myValue * 3;
myValue *= 3;

// Divide myValue by 12. This is the same as myValue = myValue / 12;
myValue /= 12;
```

Advanced Math

In the past, every language has had its own set of keywords for common math operations such as rounding and trigonometry. In .NET languages, many of these keywords remain. However, you can also use a centralized Math class that's part of the .NET Framework. This has the pleasant side effect of ensuring that the code you use to perform mathematical operations can easily be translated into equivalent statements in any .NET language with minimal fuss.

To use the math operations, you invoke the methods of the System.Math class. These methods are *static*, which means they are always available and ready to use. (The next chapter explores the difference between static and instance members in more detail.)

The following code snippet shows some sample calculations that you can perform with the Math class:

```
double myValue;
myValue = Math.Sqrt(81);          // myValue = 9.0
myValue = Math.Round(42.889, 2);  // myValue = 42.89
myValue = Math.Abs(-10);          // myValue = 10.0
myValue = Math.Log(24.212);       // myValue = 3.18.. (and so on)
myValue = Math.PI;                // myValue = 3.14.. (and so on)
```

The features of the Math class are too numerous to list here in their entirety. The preceding examples show some common numeric operations. For more information about the trigonometric and logarithmic functions that are available, refer to the reference information for the Math class on Microsoft's MSDN website (http://msdn.microsoft.com/library/system.math.aspx).

Type Conversions

Converting information from one data type to another is a fairly common programming task. For example, you might retrieve a user's text input that contains the number you want to use for a calculation. Or, you might need to take a calculated value and transform it into text you can display in a web page.

Conversions are of two types: widening and narrowing. *Widening* conversions always succeed. For example, you can always convert a 32-bit integer into a 64-bit integer. You won't need any special code:

```
int mySmallValue;
long myLargeValue;
```

```
// Get the largest possible value that can be stored as a 32-bit integer.
// .NET provides a constant named Int32.MaxValue that provides this number.
mySmallValue = Int32.MaxValue;

// This always succeeds. No matter how large mySmallValue is,
// it can be contained in myLargeValue.
myLargeValue = mySmallValue;
```

On the other hand, *narrowing* conversions may or may not succeed, depending on the data. If you're converting a 32-bit integer to a 16-bit integer, you could encounter an error if the 32-bit number is larger than the maximum value that can be stored in the 16-bit data type. All narrowing conversions must be performed explicitly. C# uses an elegant method for explicit type conversion. To convert a variable, you simply need to specify the type in parentheses before the expression you're converting.

The following code shows how to change a 32-bit integer to a 16-bit integer:

```
int count32 = 1000;
short count16;

// Convert the 32-bit integer to a 16-bit integer.
// If count32 is too large to fit, .NET will discard some of the
// information you need, and the resulting number will be incorrect.
count16 = (short)count32;
```

This process is called *casting*. If you don't use an explicit cast when you attempt to perform a narrowing conversion, you'll receive an error when you try to compile your code. However, even if you perform an explicit conversion, you could still end up with a problem. For example, consider the code shown here, which causes an overflow:

```
int mySmallValue;
long myLargeValue;

myLargeValue = Int32.MaxValue;
myLargeValue++;

// This will appear to succeed (there won't be an error at runtime),
// but your data will be incorrect because mySmallValue cannot
// hold a value this large.
mySmallValue = (int)myLargeValue;
```

The .NET languages differ in how they handle this problem. In VB, you'll always receive a runtime error that you must intercept and respond to. In C#, however, you'll simply wind up with incorrect data in mySmallValue. To avoid this problem, you should either check that your data is not too large before you attempt a narrowing conversion (which is always a good idea) or use a `checked` block. The `checked` block enables overflow checking for a portion of code. If an overflow occurs, you'll automatically receive an error, just as you would in VB:

```
checked
{
    // This will cause an exception to be thrown.
    mySmallValue = (int)myLargeValue;
}
```

■ **Tip** Usually, you won't use the `checked` block, because it's inefficient. The `checked` block catches the problem (preventing a data error), but it throws an exception, which you need to handle by using error-handling code, as explained in Chapter 7. Overall, it's easier just to perform your own checks with any potentially invalid numbers before you attempt an operation. However, the `checked` block *is* handy in one situation— debugging. That way, you can catch unexpected errors while you're still testing your application and resolve them immediately.

In C#, you can't use casting to convert numbers to strings, or vice versa. In this case, the data isn't just being moved from one variable to another—it needs to be translated to a completely different format. Thankfully, .NET has a number of solutions for performing advanced conversions. One option is to use the static methods of the Convert class, which support many common data types such as strings, dates, and numbers.

```
string countString = "10";

// Convert the string "10" to the numeric value 10.
int count = Convert.ToInt32(countString);

// Convert the numeric value 10 into the string "10".
countString = Convert.ToString(count);
```

The second step (turning a number into a string) will always work. The first step (turning a string into a number) won't work if the string contains letters or other non-numeric characters, in which case an error will occur. Chapter 7 describes how you can use error handling to detect and neutralize this sort of problem.

The Convert class is a good all-purpose solution, but you'll also find other static methods that can do the work, if you dig around in the .NET class library. The following code uses the static Int32.Parse() method to perform the same task:

```
int count;
string countString = "10";

// Convert the string "10" to the numeric value 10.
count = Int32.Parse(countString);
```

You'll also find that you can use object methods to perform some conversions a little more elegantly. The next section demonstrates this approach with the ToString() method.

Object-Based Manipulation

.NET is object-oriented to the core. In fact, even ordinary variables are really full-fledged objects in disguise. This means that common data types have the built-in smarts to handle basic operations (such as counting the number of characters in a string). Even better, it means you can manipulate strings, dates, and numbers in the same way in C# and in VB.

You'll learn far more about objects in Chapter 3. But even now it's worth taking a peek at the object underpinnings in seemingly ordinary data types. For example, every type in the .NET class library includes a ToString() method. The default implementation of this method returns the class name. In simple variables, a more useful result is returned: the string representation of the given variable. The following code snippet demonstrates how to use the ToString() method with an integer:

```
string myString;
int myInteger = 100;
```

```
// Convert a number to a string. myString will have the contents "100".
myString = myInteger.ToString();
```

To understand this example, you need to remember that all int variables are based on the Int32 type in the .NET class library. The ToString() method is built into the Int32 class, so it's available when you use an integer in any language.

The next few sections explore the object-oriented underpinnings of the .NET data types in more detail.

The String Type

One of the best examples of how class members can replace built-in functions is found with strings. In the past, every language has defined its own specialized functions for string manipulation. In .NET, however, you use the methods of the String class, which ensures consistency between all .NET languages.

The following code snippet shows several ways to manipulate a string by using its object nature:

```
string myString = "This is a test string       ";
myString = myString.Trim();               // = "This is a test string"
myString = myString.Substring(0, 4);      // = "This"
myString = myString.ToUpper();            // = "THIS"
myString = myString.Replace("IS", "AT");  // = "THAT"

int length = myString.Length;             // = 4
```

The first few statements use built-in methods, such as Trim(), Substring(), ToUpper(), and Replace(). These methods generate new strings, and each of these statements replaces the current myString with the new string object. The final statement uses a built-in Length property, which returns an integer that represents the number of characters in the string.

■ **Tip** A *method* is just a procedure that's hardwired into an object. A *property* is similar to a variable—it's a way to access a piece of data that's associated with an object. You'll learn more about methods and properties in the next chapter.

Note that the Substring() method requires a starting offset and a character length. Strings use zero-based counting. This means that the first letter is in position 0, the second letter is in position 1, and so on. You'll find this standard of zero-based counting throughout the .NET Framework for the sake of consistency. You've already seen it at work with arrays.

You can even use the string methods in succession in a single (rather ugly) line:

```
myString = myString.Trim().Substring(0, 4).ToUpper().Replace("IS", "AT");
```

Or, to make life more interesting, you can use the string methods on string literals just as easily as string variables:

```
myString = "hello".ToUpper();     // Sets myString to "HELLO"
```

Table 2-3 lists some useful members of the System.String class.

Table 2-3. *Useful String Members*

Member	Description
Length	Returns the number of characters in the string (as an integer).
ToUpper() and ToLower()	Returns a copy of the string with all the characters changed to uppercase or lowercase characters.
Trim(), TrimEnd(), and TrimStart()	Removes spaces (or the characters you specify) from either end (or both ends) of a string.
PadLeft() and PadRight()	Adds the specified character to the appropriate side of a string as many times as necessary to make the total length of the string equal to the number you specify. For example, `"Hi".PadLeft(5, '@')` returns the string *@@@Hi*.
Insert()	Puts another string inside a string at a specified (zero-based) index position. For example, `Insert(1, "pre")` adds the string *pre* after the first character of the current string.
Remove()	Removes a specified number of characters from a specified position. For example, `Remove(0, 1)` removes the first character.
Replace()	Replaces a specified substring with another string. For example, `Replace("a", "b")` changes all *a* characters in a string into *b* characters.
Substring()	Extracts a portion of a string of the specified length at the specified location (as a new string). For example, `Substring(0, 2)` retrieves the first two characters.
StartsWith() and EndsWith()	Determines whether a string starts or ends with a specified substring. For example, `StartsWith("pre")` will return either true or false, depending on whether the string begins with the letters *pre* in lowercase.
IndexOf() and LastIndexOf()	Finds the zero-based position of a substring in a string. This returns only the first match and can start at the end or beginning. You can also use overloaded versions of these methods that accept a parameter that specifies the position to start the search.
Split()	Divides a string into an array of substrings delimited by a specific substring. For example, with `Split(".")` you could chop a paragraph into an array of sentence strings.
Join()	Fuses an array of strings into a new string. You must also specify the separator that will be inserted between each element (or use an empty string if you don't want any separator).

The DateTime and TimeSpan Types

The DateTime and TimeSpan data types also have built-in methods and properties. These class members allow you to perform three useful tasks:

- Extract a part of a DateTime (for example, just the year) or convert a TimeSpan to a specific representation (such as the total number of days or total number of minutes)
- Easily perform date calculations
- Determine the current date and time and other information (such as the day of the week or whether the date occurs in a leap year)

For example, the following block of code creates a DateTime object, sets it to the current date and time, and adds a number of days. It then creates a string that indicates the year that the new date falls in (for example, 2012).

```
DateTime myDate = DateTime.Now;
myDate = myDate.AddDays(100);
string dateString = myDate.Year.ToString();
```

The next example shows how you can use a TimeSpan object to find the total number of minutes between two DateTime objects:

```
DateTime myDate1 = DateTime.Now;
DateTime myDate2 = DateTime.Now.AddHours(3000);

TimeSpan difference;
difference = myDate2.Subtract(myDate1);

double numberOfMinutes;
numberOfMinutes = difference.TotalMinutes;
```

The DateTime and TimeSpan classes also support the + and – arithmetic operators, which do the same work as the built-in methods. That means you can rewrite the example shown earlier like this:

```
// Adding a TimeSpan to a DateTime creates a new DateTime.
DateTime myDate1 = DateTime.Now;
TimeSpan interval = TimeSpan.FromHours(3000);
DateTime myDate2 = myDate1 + interval;

// Subtracting one DateTime object from another produces a TimeSpan.
TimeSpan difference;
difference = myDate2 - myDate1;
```

These examples give you an idea of the flexibility .NET provides for manipulating date and time data. Tables 2-4 and 2-5 list some of the more useful built-in features of the DateTime and TimeSpan objects.

Table 2-4. *Useful DateTime Members*

Member	Description
Now	Gets the current date and time. You can also use the UtcNow property to change the computer's local time (which is relative to the local time zone) to *Coordinated Universal Time* (UTC). Assuming your computer is correctly configured, this corresponds to the current time in the Western European (UTC + 0) time zone.
Today	Gets the current date and leaves time set to 00:00:00.
Year, Date, Month, Hour, Minute, Second, and Millisecond	Returns one part of the DateTime object as an integer. For example, Month will return 12 for any day in December.
DayOfWeek	Returns an enumerated value that indicates the day of the week for this DateTime, using the DayOfWeek enumeration. For example, if the date falls on Sunday, this will return DayOfWeek.Sunday.
Add() and Subtract()	Adds or subtracts a TimeSpan from the DateTime. For convenience, these operations are mapped to the + and – operators, so you can use them instead when performing calculations with dates.
AddYears(), AddMonths(), AddDays(), AddHours(), AddMinutes(), AddSeconds(), AddMilliseconds()	Adds an integer that represents a number of years, months, and so on, and returns a new DateTime. You can use a negative integer to perform a date subtraction.
DaysInMonth()	Returns the number of days in the specified month in the specified year.
IsLeapYear()	Returns true or false depending on whether the specified year is a leap year.
ToString()	Returns a string representation of the current DateTime object. You can also use an overloaded version of this method that allows you to specify a parameter with a format string.

Table 2-5. *Useful TimeSpan Members*

Member	Description
Days, Hours, Minutes, Seconds, Milliseconds	Returns one component of the current TimeSpan. For example, the Hours property can return an integer from –23 to 23.
TotalDays, TotalHours, TotalMinutes, TotalSeconds, TotalMilliseconds	Returns the total value of the current TimeSpan as a number of days, hours, minutes, and so on. The value is returned as a double, which may include a fractional value. For example, the TotalDays property might return a number such as 234.342.
Add() and Subtract()	Combines TimeSpan objects together. For convenience, these operations are mapped to the + and – operators, so you can use them instead when performing calculations with times.
FromDays(), FromHours(), FromMinutes(), FromSeconds(), FromMilliseconds()	Allows you to quickly create a new TimeSpan. For example, you can use TimeSpan.FromHours(24) to create a TimeSpan object exactly 24 hours long.
ToString()	Returns a string representation of the current TimeSpan object. You can also use an overloaded version of this method that allows you to specify a parameter with a format string.

The Array Type

Arrays also behave like objects in the world of .NET. (Technically, every array is an instance of the System.Array type.) For example, if you want to find out the size of a one-dimensional array, you can use the Length property or the GetLength() method, both of which return the total number of elements in an array:

```
int[] myArray = {1, 2, 3, 4, 5};
int numberOfElements;

numberOfElements = myArray.Length;       // numberOfElements = 5
```

You can also use the GetUpperBound() method to find the highest index number in an array. When calling GetUpperBound(), you supply a number that indicates what dimension you want to check. In the case of a one-dimensional array, you must always specify 0 to get the index number from the first dimension. In a two-dimensional array, you can also use 1 for the second bound; in a three-dimensional array, you can also use 2 for the third bound; and so on.

The following code snippet shows GetUpperBound() in action:

```
int[] myArray = {1, 2, 3, 4, 5};
int bound;

// Zero represents the first dimension of an array.
bound = myArray.GetUpperBound(0);        // bound = 4
```

On a one-dimensional array, GetUpperBound() always returns a number that's one less than the length. That's because the first index number is 0, and the last index number is always one less than the total number of items. However, in a two-dimensional array, you can find the highest index number for a specific dimension in

that array. For example, the following code snippet uses GetUpperBound() to find the total number of rows and the total number of columns in a two-dimensional array:

```
// Create a 4x2 array (a grid with four rows and two columns).
int[,] intArray = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};

int rows = intArray.GetUpperBound(0) + 1;    // rows = 4
int columns = intArray.GetUpperBound(1) + 1; // columns = 2
```

Having these values—the array length and indexes—is handy when looping through the contents of an array, as you'll see later in this chapter, in the "Loops" section.

Arrays also provide a few other useful methods, which allow you to sort them, reverse them, and search them for a specified element. Table 2-6 lists some useful members of the System.Array class.

Table 2-6. *Useful Array Members*

Member	Description
Length	Returns an integer that represents the total number of elements in all dimensions of an array. For example, a 3×3 array has a length of 9.
GetLowerBound() and GetUpperBound()	Determines the dimensions of an array. As with just about everything in .NET, you start counting at zero (which represents the first dimension).
Clear()	Empties part or all of an array's contents, depending on the index values that you supply. The elements revert to their initial empty values (such as 0 for numbers).
IndexOf() and LastIndexOf()	Searches a one-dimensional array for a specified value and returns the index number. You cannot use this with multidimensional arrays.
Sort()	Sorts a one-dimensional array made up of comparable data such as strings or numbers.
Reverse()	Reverses a one-dimensional array so that its elements are backward, from last to first.

Conditional Logic

In many ways, *conditional logic*—deciding which action to take based on user input, external conditions, or other information—is the heart of programming.

All conditional logic starts with a *condition*: a simple expression that can be evaluated to true or false. Your code can then make a decision to execute different logic depending on the outcome of the condition. To build a condition, you can use any combination of literal values or variables along with *logical operators*. Table 2-7 lists the basic logical operators.

Table 2-7. *Logical Operators*

Operator	Description
==	Equal to.
!=	Not equal to.
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
&&	Logical and (evaluates to true only if both expressions are true). If the first expression is false, the second expression is not evaluated.
||	Logical or (evaluates to true if either expression is true). If the first expression is true, the second expression is not evaluated.

You can use all the comparison operators with any numeric types. With string data types, you can use only the equality operators (== and !=). C# doesn't support other types of string comparison operators—instead, you need to use the String.Compare() method. The String.Compare() method deems that a string is "less than" another string if it occurs earlier in an alphabetic sort. Thus, *apple* is less than *attach*. The return value from String.Compare is 0 if the strings match, 1 if the first supplied string is greater than the second, and –1 if the first string is less than the second. Here's an example:

```
int result;
result = String.Compare("apple", "attach");  // result = -1
result = String.Compare("apple", "all");     // result = 1
result = String.Compare("apple", "apple");   // result = 0

// Another way to perform string comparisons.
string word = "apple";
result = word.CompareTo("attach");           // result = -1
```

The if Statement

The `if` statement is the powerhouse of conditional logic, able to evaluate any combination of conditions and deal with multiple and different pieces of data. Here's an example with an `if` statement that features two `else` conditions:

```
if (myNumber > 10)
{
    // Do something.
}
else if (myString == "hello")
{
    // Do something.
}
else
{
    // Do something.
}
```

An `if` block can have any number of conditions. If you test only a single condition, you don't need to include any `else` blocks.

■ **Note** In this example, each block is clearly identified with the `{ }` characters. This is a requirement if you want to write multiple lines of code in a conditional block. If your conditional block requires just a single statement, you can omit the curly braces. However, it's never a bad idea to keep them, because it makes your code clear and unambiguous.

Keep in mind that the `if` construct matches one condition at most. For example, if myNumber is greater than 10, the first condition will be met. That means the code in the first conditional block will run, and no other conditions will be evaluated. Whether myString contains the text *hello* becomes irrelevant, because that condition will not be evaluated. If you want to check both conditions, don't use an `else` block—instead, you need two `if` blocks back-to-back, as shown here:

```
if (myNumber > 10)
{
    // Do something.
}
if (myString == "hello")
{
    // Do something.
}
```

The switch Statement

C# also provides a `switch` statement that you can use to evaluate a single variable or expression for multiple possible values. The only limitation is that the variable you're evaluating must be an integer-based data type, a bool, a char, a string, or a value from an enumeration. Other data types aren't supported.

In the following code, each case examines the myNumber variable and tests whether it's equal to a specific integer:

```
switch (myNumber)
{
    case 1:
        // Do something.
        break;
    case 2:
        // Do something.
        break;
    default:
        // Do something.
        break;
}
```

You'll notice that the C# syntax inherits the convention of C/C++ programming, which requires that every branch in a `switch` statement be ended by a special `break` keyword. If you omit this keyword, the compiler will alert you and refuse to build your application. The only exception is if you choose to stack multiple `case`

statements directly on top of each other with no intervening code. This allows you to write one segment of code that handles more than one case. Here's an example:

```
switch (myNumber)
{
    case 1:
    case 2:
        // This code executes if myNumber is 1 or 2.
        break;
    default:
        // Do something.
        break;
}
```

Unlike the `if` statement, the `switch` statement is limited to evaluating a single piece of information at a time. However, it provides a leaner, clearer syntax than the `if` statement when you need to test a single variable.

Loops

Loops allow you to repeat a segment of code multiple times. C# has three basic types of loops. You choose the type of loop based on the type of task you need to perform. Your choices are as follows:

- You can loop a set number of times with a `for` loop.
- You can loop through all the items in a collection of data by using a `foreach` loop.
- You can loop while a certain condition holds true with a `while` or `do...while` loop.

The `for` and `foreach` loops are ideal for chewing through sets of data that have known, fixed sizes. The `while` loop is a more flexible construct that allows you to continue processing until a complex condition is met. The `while` loop is often used with repetitive tasks or calculations that don't have a set number of iterations.

The for Loop

The `for` loop is a basic ingredient in many programs. It allows you to repeat a block of code a set number of times, using a built-in counter. To create a `for` loop, you need to specify a starting value, an ending value, and the amount to increment with each pass. Here's one example:

```
for (int i = 0; i < 10; i++)
{
    // This code executes ten times.
    System.Diagnostics.Debug.Write(i);
}
```

You'll notice that the `for` loop starts with parentheses that indicate three important pieces of information. The first portion (`int i = 0`) creates the counter variable (`i`) and sets its initial value (0). The third portion (`i++`) increments the counter variable. In this example, the counter is incremented by 1 after each pass. That means `i` will be equal to 0 for the first pass, equal to 1 for the second pass, and so on. However, you could adjust this statement so that it decrements the counter (or performs any other operation you want). The middle portion (`i < 10`) specifies the condition that must be met for the loop to continue. This condition is tested at the start of every pass through the block. If `i` is greater than or equal to 10, the condition will evaluate to false, and the loop will end.

If you run this code by using a tool such as Visual Studio, it will write the following numbers in the Debug window:

```
0 1 2 3 4 5 6 7 8 9
```

It often makes sense to set the counter variable based on the number of items you're processing. For example, you can use a `for` loop to step through the elements in an array by checking the size of the array before you begin. Here's the code you would use:

```
string[] stringArray = {"one", "two", "three"};

for (int i = 0; i < stringArray.Length; i++)
{
    System.Diagnostics.Debug.Write(stringArray[i] + " ");
}
```

This code produces the following output:

```
one two three
```

BLOCK-LEVEL SCOPE

If you define a variable inside some sort of block structure (such as a loop or a conditional block), the variable is automatically released when your code exits the block. That means you will no longer be able to access it. The following code demonstrates this behavior:

```
int tempVariableA;
for (int i = 0; i < 10; i++)
{
    int tempVariableB;
    tempVariableA = 1;
    tempVariableB = 1;
}
// You cannot access tempVariableB here.
// However, you can still access tempVariableA.
```

This change won't affect many programs. It's really designed to catch a few more accidental errors. If you do need to access a variable inside and outside of some type of block structure, just define the variable *before* the block starts.

The foreach Loop

C# also provides a `foreach` loop that allows you to loop through the items in a set of data. With a `foreach` loop, you don't need to create an explicit counter variable. Instead, you create a variable that represents the type of data for which you're looking. Your code will then loop until you've had a chance to process each piece of data in the set.

The `foreach` loop is particularly useful for traversing the data in collections and arrays. For example, the next code segment loops through the items in an array by using `foreach`. This code has exactly the same effect as the example in the previous section, but it's a little simpler:

```
string[] stringArray = {"one", "two", "three"};

foreach (string element in stringArray)
{
    // This code loops three times, with the element variable set to
    // "one", then "two", and then "three".
    System.Diagnostics.Debug.Write(element + " ");
}
```

In this case, the `foreach` loop examines each item in the array and tries to convert it to a string. Thus, the `foreach` loop defines a string variable named element. If you used a different data type, you'd receive an error.

The `foreach` loop has one key limitation: it's read-only. For example, if you wanted to loop through an array and change the values in that array at the same time, `foreach` code wouldn't work. Here's an example of some flawed code:

```
int[] intArray = {1,2,3};
foreach (int num in intArray)
{
    num += 1;
}
```

In this case, you would need to fall back on a basic `for` loop with a counter.

The while loop

Finally, C# supports a `while` loop that tests a specific condition before or after each pass through the loop. When this condition evaluates to false, the loop is exited.

Here's an example that loops ten times. At the beginning of each pass, the code evaluates whether the counter (i) is less than some upper limit (in this case, 10). If it is, the loop performs another iteration.

```
int i = 0;
while (i < 10)
{
    i += 1;
    // This code executes ten times.
}
```

You can also place the condition at the end of the loop by using the `do...while` syntax. In this case, the condition is tested at the end of each pass through the loop:

```
int i = 0;
do
{
    i += 1;
    // This code executes ten times.
}
while (i < 10);
```

Both of these examples are equivalent, unless the condition you're testing is false to start. In that case, the `while` loop will skip the code entirely. The `do...while` loop, on the other hand, will always execute the code at least once, because it doesn't test the condition until the end.

■ **Tip** Sometimes you need to exit a loop in a hurry. In C#, you can use the `break` statement to exit any type of loop. You can also use the `continue` statement to skip the rest of the current pass, evaluate the condition, and (if it returns true) start the next pass.

Methods

Methods are the most basic building block you can use to organize your code. Essentially, a method is a named grouping of one or more lines of code. Ideally, each method will perform a distinct, logical task. By breaking down your code into methods, you not only simplify your life, but also make it easier to organize your code into classes and step into the world of object-oriented programming.

The first decision you need to make when declaring a method is whether you want to return any information. For example, a method named GetStartTime() might return a DateTime object that represents the time an application was first started. A method can return, at most, one piece of data.

When you declare a method in C#, the first part of the declaration specifies the data type of the return value, and the second part indicates the method name. If your method doesn't return any information, you should use the `void` keyword instead of a data type at the beginning of the declaration.

Here are two examples—one method that doesn't return anything and one that does:

```
// This method doesn't return any information.
void MyMethodNoReturnedData()
{
    // Code goes here.
}

// This method returns an integer.
int MyMethodReturnsData()
{
    // As an example, return the number 10.
    return 10;
}
```

Notice that the method name is always followed by parentheses. This allows the compiler to recognize that it's a method.

In this example, the methods don't specify their accessibility. This is just a common C# convention. You're free to add an accessibility keyword (such as `public` or `private`), as shown here:

```
private void MyMethodNoReturnedData()
{
    // Code goes here.
}
```

The accessibility determines how different classes in your code can interact. Private methods are hidden from view and are available only locally, whereas public methods can be called by all the other classes in your application. To really understand what this means, you'll need to read the next chapter, which discusses accessibility in more detail.

■ **Tip** If you don't specify accessibility, the method is always private. The examples in this book always include accessibility keywords, because they improve clarity. Most programmers agree that it's a good approach to explicitly spell out the accessibility of your code.

Invoking your methods is straightforward—you simply type the name of the method, followed by parentheses. If your method returns data, you have the option of using the data it returns or just ignoring it:

```
// This call is allowed.
MyMethodNoReturnedData();

// This call is allowed.
MyMethodReturnsData();

// This call is allowed.
int myNumber;
myNumber = MyMethodReturnsData();

// This call isn't allowed.
// MyMethodNoReturnedData() does not return any information.
myNumber = MyMethodNoReturnedData();
```

Parameters

Methods can also accept information through *parameters*. Parameters are declared in a similar way to variables. By convention, parameter names always begin with a lowercase letter in any language.

Here's how you might create a function that accepts two parameters and returns their sum:

```
private int AddNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

When calling a method, you specify any required parameters in parentheses or use an empty set of parentheses if no parameters are required:

```
// Call a method with no parameters.
MyMethodNoReturnedData();

// Call a method that requires two integer parameters.
MyMethodNoReturnedData2(10, 20);

// Call a method with two integer parameters and an integer return value.
int returnValue = AddNumbers(10, 10);
```

Method Overloading

C# supports method *overloading*, which allows you to create more than one method with the same name but with a different set of parameters. When you call the method, the CLR automatically chooses the correct version by examining the parameters you supply.

This technique allows you to collect different versions of several methods together. For example, you might allow a database search that returns an array of Product objects representing records in the database. Rather than create three methods with different names depending on the criteria, such as GetAllProducts(), GetProductsInCategory(), and GetActiveProducts(), you could create three versions of the GetProducts() method. Each method would have the same name but a different *signature*, meaning it would require different parameters. This example provides two overloaded versions for the GetProductPrice() method:

```
private decimal GetProductPrice(int ID)
{
    // Code here.
}

private decimal GetProductPrice(string name)
{
    // Code here.
}

// And so on...
```

Now you can look up product prices based on the unique product ID or the full product name, depending on whether you supply an integer or string argument:

```
decimal price;

// Get price by product ID (the first version).
price = GetProductPrice(1001);

// Get price by product name (the second version).
price = GetProductPrice("DVD Player");
```

You cannot overload a method with versions that have the same signature—that is, the same number of parameters and parameter data types—because the CLR will not be able to distinguish them from each other. When you call an overloaded method, the version that matches the parameter list you supply is used. If no version matches, an error occurs.

■ **Note** .NET uses overloaded methods in most of its classes. This approach allows you to use a flexible range of parameters while centralizing functionality under common names. Even the methods you've seen so far (such as the String methods for padding or replacing text) have multiple versions that provide similar features with various options.

Optional and Named Parameters

Method overloading is a time-honored technique for making methods more flexible, so you can call them in a variety of ways. C# also has another feature that supports the same goal: optional parameters.

An *optional parameter* is any parameter that has a default value. If your method has normal parameters and optional parameters, the optional parameters must be placed at the end of the parameter list. Here's an example of a method that has a single optional parameter:

```
private string GetUserName(int ID, bool useShortForm = false)
{
    // Code here.
}
```

Here, the useShortForm parameter is optional, which gives you two ways to call the GetUserName() method:

```
// Explicitly set the useShortForm parameter.
name = GetUserName(401, true);

// Don't set the useShortForm parameter, and use the default value (false).
name = GetUserName(401);
```

Sometimes you'll have a method with multiple optional parameters, like this one:

```
private decimal GetSalesTotalForRegion(int regionID, decimal minSale = 0,
  decimal maxSale = Decimal.MaxValue, bool includeTax = false)
{
    // Code here.
}
```

In this situation, the easiest option is to pick out the parameters you want to set by name. This feature is called *named parameters*, and to use it you simply add the parameter name followed by a colon (:), followed by the value, as shown here:

```
total = GetSalesTotalForRegion(523, maxSale: 5000);
```

Although you can accomplish many of the same things with optional parameters and method overloading, classes are more likely to use method overloading for two reasons. First, most of the classes in .NET were created in previous versions, when C# did not support optional parameters. Second, not all .NET languages support optional parameters (although C# and VB do).

It's also worth noting that method overloading allows you to deal with either/or situations, while optional parameters do not. For example, the GetProductPrice() method shown in the previous section required a string *or* an integer. It's not acceptable to make both of these into optional parameters, because at least one is required, and supplying the two of them at once makes no sense. Thus, this is a situation where method overloading fits more naturally.

Delegates

Delegates allow you to create a variable that "points" to a method. You can then use this variable at any time to invoke the method. Delegates help you write flexible code that can be reused in many situations. They're also the basis for *events*, an important .NET concept that you'll consider in the next chapter.

The first step when using a delegate is to define its signature. The *signature* is a combination of several pieces of information about a method: its return type, the number of parameters it has, and the data type of each parameter.

A delegate variable can point only to a method that matches its specific signature. In other words, the method must have the same return type, the same number of parameters, and the same data type for each parameter as the delegate. For example, if you have a method that accepts a single string parameter and another method that accepts two string parameters, you'll need to use a separate delegate type for each method.

To consider how this works in practice, assume that your program has the following method:

```
private string TranslateEnglishToFrench(string english)
{
    // Code goes here.
}
```

This method accepts a single string argument and returns a string. With those two details in mind, you can define a delegate that matches this signature. Here's how you would do it:

```
private delegate string StringFunction(string inputString);
```

Notice that the name you choose for the parameters and the name of the delegate don't matter. The only requirement is that the data types for the return value and parameters match exactly.

Once you've defined a type of delegate, you can create and assign a delegate variable at any time. Using the StringFunction delegate type, you could create a delegate variable like this:

```
StringFunction functionReference;
```

Once you have a delegate variable, the fun begins. Using your delegate variable, you can point to any method that has the matching signature. In this example, the StringFunction delegate type requires one string parameter and returns a string. Thus, you can use the functionReference variable to store a reference to the TranslateEnglishToFrench() method you saw earlier. Here's how to do it:

```
functionReference = TranslateEnglishToFrench;
```

■ **Note** When you assign a method to a delegate variable in C#, you don't use brackets after the method name. This indicates that you are *referring* to the method, not attempting to execute it. If you added the parentheses, the CLR would attempt to run your method and convert the return value to the delegate type, which wouldn't work (and therefore would generate a compile-time error).

Now that you have a delegate variable that references a method, you can invoke the method *through* the delegate. To do this, you just use the delegate name as though it were the method name:

```
string frenchString;
frenchString = functionReference("Hello");
```

In the previous code example, the method that the functionReference delegate points to will be invoked with the parameter value "Hello", and the return value will be stored in the frenchString variable.

The following code shows all these steps—creating a delegate variable, assigning a method, and calling the method—from start to finish:

```
// Create a delegate variable.
StringFunction functionReference;

// Store a reference to a matching method in the delegate.
functionReference = TranslateEnglishToFrench;

// Run the method that functionReference points to.
// In this case, it will be TranslateEnglishToFrench().
string frenchString = functionReference("Hello");
```

The value of delegates is in the extra layer of flexibility they add. It's not apparent in this example, because the same piece of code creates the delegate variable and uses it. However, in a more complex application, one method would create the delegate variable, and another method would use it. The benefit in this scenario is that the second method doesn't need to know where the delegate points. Instead, it's flexible enough to use any method that has the right signature. In the current example, imagine a translation library that could translate between English and a variety of different languages, depending on whether the delegate it uses points to TranslateEnglishToFrench(), TranslateEnglishToSpanish(), TranslateEnglishToGerman(), and so on.

DELEGATES ARE THE BASIS OF EVENTS

Wouldn't it be nice to have a delegate that could refer to more than one function at once and invoke them simultaneously? This would allow the client application to have multiple "listeners" and notify the listeners all at once when something happens.

In fact, delegates do have this functionality, but you're more likely to see it in use with .NET events. Events, which are described in the next chapter, are based on delegates but work at a slightly higher level. In a typical ASP.NET program, you'll use events extensively, but you'll probably never work directly with delegates.

The Last Word

It's impossible to do justice to an entire language in a single chapter. However, the concepts introduced in this chapter—variables, conditional logic, loops, and methods—are common to virtually all languages, and you're sure to have seen them before.

Don't worry if the C# details are a bit too much to remember all at once. Instead, use this chapter as a reference as you work through the full ASP.NET examples that are featured in the following chapters. That way, you can quickly clear up any language issues or syntax questions you face.

In the next chapter, you'll learn more language concepts and consider the object-oriented nature of .NET.

CHAPTER 3

Types, Objects, and Namespaces

.NET is all about objects. Not only does .NET allow you to use them, it *demands* that you do. Almost every ingredient you'll use to create a web application is, on some level, really a kind of object. In this chapter, you'll learn how objects are defined and how you manipulate them in your code. Taken together, these concepts are the basics of what's commonly called *object-oriented programming*.

So, how much do you need to know about object-oriented programming to write ASP.NET web pages? It depends on whether you want to follow existing examples and cut and paste code samples or have a deeper understanding of the way .NET works and gain more control. This book assumes that if you're willing to pick up a thousand-page book, then you're the type of programmer who excels by understanding how and why things work the way they do. It also assumes you're interested in some of the advanced ASP.NET programming tasks that *will* require class-based design, such as creating your own database component (see Chapter 22).

This chapter explains objects from the point of view of the .NET Framework. It doesn't rehash the typical object-oriented theory, because countless excellent programming books cover the subject. Instead, you'll see the types of objects .NET allows, how they're constructed, and how they fit into the larger framework of namespaces and assemblies.

The Basics About Classes

When you're starting out as a developer, *classes* are one of the first concepts you come across. Technically, classes are the code definitions for objects. The nice thing about a class is that you can use it to create as many objects as you need. For example, you might have a class that represents an XML file, which can be used to read some data. If you want to access multiple XML files at once, you can create several instances of your class, as shown in Figure 3-1. These instances are called *objects*.

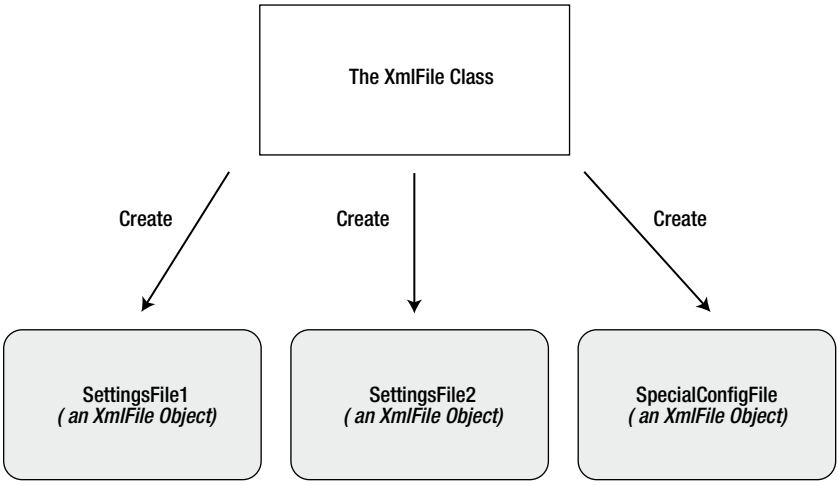

Figure 3-1. *Classes are used to create objects*

■ **Note** At its simplest, *object-oriented programming* is the idea that your code should be organized into separate classes. If followed carefully, this approach leads to code that's easier to alter, enhance, debug, and reuse.

Classes interact with each other with the help of three key ingredients:

Properties: Properties allow you to access an object's data. Some properties are read-only, so they cannot be modified, while others can be changed. For example, the previous chapter demonstrated how you can use the read-only Length property of a String object to find out the number of letters in a string.

Methods: Methods allow you to perform an action on an object. Unlike properties, methods are used for actions that perform a distinct task or may change the object's state significantly. For example, to open a connection to a database, you might call an Open() method in a Connection object.

Events: Events provide notification that something has happened. If you've ever programmed a modern Windows application, you know how controls can fire events to trigger your code. For example, if a user clicks a button, the Button object fires a Click event, which your code can react to. The same pattern works with web controls in an ASP.NET web page, although there are some limitations (as you'll learn in Chapter 5).

In addition, classes contain their own code and internal set of private data. Classes behave like "black boxes," which means that when you use an object, you shouldn't waste any time wondering how it works or what low-level information it's using. Instead, you need to worry only about the *public interface* of a class, which is the set of properties, methods, and events that are available for you to use. Together, these elements are called class *members*.

In ASP.NET, you'll create your own custom classes to represent individual web pages. In addition, you'll create custom classes if you design separate components. For the most part, however, you'll be using prebuilt classes from the .NET class library, rather than programming your own.

Static Members

One of the tricks about .NET classes is that you really use them in two ways. You can use some class members without creating an object first. These are called *static* members, and they're accessed by class name. For example, the DateTime type provides a static property named Now. You can access this property at any time by using the full member name DateTime.Now. You don't need to create a DateTime object first.

On the other hand, the majority of the DateTime members require a valid instance. For example, you can't use the AddDays() method or the Hour property without a valid object. These *instance* members have no meaning without a live object and some valid data to draw on.

The following code snippet uses static and instance members:

```
// Get the current date using a static property.
// Note that you need to use the class name DateTime.
DateTime myDate = DateTime.Now;

// Use an instance method to add a day.
// Note that you need to use the object name myDate.
myDate = myDate.AddDays(1);

// The following code makes no sense.
// It tries to use the instance method AddDays() with the class name DateTime!
myDate = DateTime.AddDays(1);
```

Both properties and methods can be designated as static. Static properties and methods are a major part of the .NET Framework, and you will use them frequently in this book. Some classes may consist entirely of static members (such as the Math class shown in the previous chapter), and some may use only instance members. Other classes, such as DateTime, provide a combination of the two.

The next example, which introduces a basic class, will use only instance members. This is the most common design and a good starting point.

A Simple Class

To create a class, you must define it by using a special block structure:

```
public class MyClass
{
    // Class code goes here.
}
```

You can define as many classes as you need in the same file. However, good coding practices suggest that in most cases you use a single file for each class.

Classes exist in many forms. They may represent an actual thing in the real world (as they do in most programming textbooks), they may represent some programming abstraction (such as a rectangle or color structure), or they may just be a convenient way to group related functionality (as with the Math class). Deciding what a class should represent and breaking down your code into a group of interrelated classes are part of the art of programming.

Building a Basic Class

In the next example, you'll see how to construct a .NET class piece by piece. This class will represent a product from the catalog of an e-commerce company. The Product class will store product data, and it will include the built-in functionality needed to generate a block of HTML that displays the product on a web page. When this class is complete, you'll be able to put it to work with a sample ASP.NET test page.

After you've defined a class, the first step is to add some basic data. The next example defines three member variables that store information about the product—namely, its name, its price, and a URL that points to an image file:

```
public class Product
{
    private string name;
    private decimal price;
    private string imageUrl;
}
```

A local variable exists only until the current method ends. On the other hand, a *member variable* (or *field*) is declared as part of a class. It's available to all the methods in the class, and it lives as long as the containing object lives.

When you create a member variable, you set its *accessibility*. The accessibility determines whether other parts of your code will be able to read and alter this variable. For example, if ClassA contains a private variable, the code in ClassB will not be able to read or modify it. Only the code in ClassA will have that ability. On the other hand, if ObjectA has a public variable, any other object in your application is free to read and alter the information it contains. Local variables don't support any accessibility keywords, because they can never be made available to any code beyond the current procedure. Generally, in a simple ASP.NET application, most of your variables will be private because the majority of your code will be self-contained in a single web page class. As you start creating separate components to reuse functionality, however, accessibility becomes much more important. Table 3-1 explains the access levels you can use.

Table 3-1. *Accessibility Keywords*

Keyword	Accessibility
`public`	Can be accessed by any class
`private`	Can be accessed only by members inside the current class
`internal`	Can be accessed by members in any of the classes in the current assembly (the file with the compiled code)
`protected`	Can be accessed by members in the current class or in any class that inherits from this class
`protected internal`	Can be accessed by members in the current application (as with internal) *and* by the members in any class that inherits from this class

The accessibility keywords don't apply only to variables. They also apply to methods, properties, and events, all of which will be explored in this chapter.

■ **Tip** By convention, all the public pieces of your class (the class name, public events, properties and procedures, and so on) should use *Pascal case*. This means the name starts with an initial capital. (The function name DoSomething() is one example of Pascal case.) On the other hand, private members can use any case you want. Usually, private members will adopt *camel case*. This means the name starts with an initial lowercase letter. (The variable name myInformation is one example of camel case.) Some developers begin all private member names with _ or m_ (for *member*), although this is purely a matter of convention.

Creating an Object

When creating an object, you need to specify the new keyword. The new keyword *instantiates* the object, which means it grabs on to a piece of memory and creates the object there. If you declare a variable for your object but don't use the new keyword to actually instantiate it, you'll receive the infamous "null reference" error when you try to use your object. That's because the object you're attempting to use doesn't exist, and your variable doesn't point to anything at all.

The following code snippet creates an object based on the Product class and then releases it:

```
Product saleProduct = new Product();

// Optionally you could do this in two steps:
// Product saleProduct;
// saleProduct = new Product();

// Now release the object from memory.
saleProduct = null;
```

In .NET, you almost never need to use the last line, which releases the object. That's because objects are automatically released when the appropriate variable goes out of scope. (Remember, a variable goes out of scope when it's no longer accessible to your code. This happens, for example, when you define a variable in a method and the method ends. It also happens when you define a variable inside another block structure—say, a conditional if block or a loop—and that block ends.)

Objects are also released when your application ends. In an ASP.NET web page, your application is given only a few seconds to live. After the web page is rendered to HTML, the application ends, and all objects are automatically released.

■ **Tip** Just because an object is released doesn't mean the memory it uses is immediately reclaimed. The CLR uses a long-running service (called *garbage collection*) that periodically scans for released objects and reclaims the memory they hold.

In some cases, you will want to declare an object variable without using the new keyword to create it. For example, you might want to assign an instance that already exists to your object variable. Or you might receive a live object as a return value from a function. The following code shows one such example:

```
// Declare but don't create the product.
Product saleProduct;

// Call a function that accepts a numeric product ID parameter,
// and returns a product object.
saleProduct = FetchProduct(23);
```

Once you understand the concept, you can compress this code into one statement:

```
Product saleProduct = FetchProduct(23);
```

In these cases, when you aren't actually creating an object, you shouldn't use the new keyword.

Adding Properties

The simple Product class is essentially useless because your code cannot manipulate it. All its information is private and unreachable. Other classes won't be able to set or read this information.

To overcome this limitation, you could make the member variables public. Unfortunately, that approach could lead to problems because it would give other objects free access to change everything, even allowing them to apply invalid or inconsistent data. Instead, you need to add a "control panel" through which your code can manipulate Product objects in a safe way. You do this by adding *property accessors*.

Accessors usually have two parts. The `get` accessor allows your code to retrieve data from the object. The `set` accessor allows your code to set the object's data. In some cases, you might omit one of these parts, such as when you want to create a property that can be examined but not modified.

Accessors are similar to any other type of method in that you can write as much code as you need. For example, your property `set` accessor could raise an error to alert the client code of invalid data and prevent the change from being applied. Or, your property `set` accessor could change multiple private variables at once, thereby making sure the object's internal state remains consistent. In the Product class example, this sophistication isn't required. Instead, the property accessors just provide straightforward access to the private variables. For example, the Name property simply gets or sets the value of the private member variable called *name*.

Property accessors, like any other public piece of a class, should start with an initial capital. This allows you to give the same name to the property accessor and the underlying private variable, because they will have different capitalization, and C# is a case-sensitive language. (This is one of the rare cases where it's acceptable to differentiate between two elements based on capitalization.) Another option would be to precede the private variable name with an underscore or the characters *m_* (for member variable).

```
public class Product
{
    private string name;
    private decimal price;
    private string imageUrl;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    public decimal Price
    {
        get
        {
            return price;
        }
        set
        {
            price = value;
        }
    }

    public string ImageUrl
    {
```

```
        get
        {
            return imageUrl;
        }
        set
        {
            imageUrl = value;
        }
    }
}
```

■ **Note** In your property setter, you access the value that's being applied by using the `value` keyword. Essentially, `value` is a parameter that's passed to your property-setting code automatically.

The client can now create and configure the object by using its properties and the familiar dot syntax. For example, if the object variable is named saleProduct, you can set the product name by using the saleProduct. Name property. Here's an example:

```
Product saleProduct = new Product();
saleProduct.Name = "Kitchen Garbage";
saleProduct.Price = 49.99M;
saleProduct.ImageUrl = "http://mysite/garbage.png";
```

You'll notice that this example uses an *M* to indicate that the literal number 49.99 should be interpreted as a decimal value, not a double.

Usually, property accessors come in pairs—that is, every property has both a `get` accessor and a `set` accessor. But this isn't always the case. You can create properties that can be read but not set (which are called *read-only* properties) and, less commonly, you can create properties that can be set but not retrieved (called *write-only* properties). All you need to do is leave out the accessor that you don't need. Here's an example of a read-only property:

```
public decimal Price
{
    get
    {
        return price;
    }
}
```

This technique is particularly handy if you want to create properties that don't correspond directly to a private member variable. For example, you might want to use properties that represent calculated values or use properties that are based on other properties.

Using Automatic Properties

If you have really simple properties—properties that do nothing except set or get the value of a private member variable—you can simplify your code by using a C# language feature called *automatic properties*.

Automatic properties are properties without any code. When you use an automatic property, you declare it, but you don't supply the code for the `get` and `set` accessors, and you don't declare the matching private variable. Instead, the C# compiler adds these details for you.

Because the properties in the Product class simply get and set member variables, you can replace any of them (or all of them) with automatic properties. Here's an example:

```
public decimal Price
{
    get;
    set;
}
```

You don't know what name the C# compiler will choose when it creates the corresponding private member variable. However, it doesn't matter, because you'll never need to access the private member variable directly. Instead, you'll always use the public Price property.

For even more space savings, you can compress the declaration of an automatic property to a single line. Here's a complete, condensed Product class that uses this technique:

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string ImageUrl { get; set; }
}
```

The only disadvantage to automatic properties is that you'll need to switch them back to normal properties if you want to add some more specialized code after the fact. For example, you might want to add code that performs validation or raises an event when a property is set.

Adding a Method

The current Product class consists entirely of data, which is exposed by a small set of properties. This type of class is often useful in an application. For example, you might use it to send information about a product from one function to another. However, it's more common to add functionality to your classes along with the data. This functionality takes the form of methods.

Methods are simply named procedures that are built into your class. When you call a method on an object, the method does something useful, such as return some calculated data. In this example, we'll add a GetHtml() method to the Product class. This method will return a string representing a formatted block of HTML based on the current data in the Product object. This HTML includes a heading with the product name, the product price, and an <img> element that shows the associated product picture. (You'll explore HTML more closely in Chapter 4.)

```
public class Product
{
    // (Additional class code omitted for clarity.)

    public string GetHtml()
    {
        string htmlString;
        htmlString = "<h1>" + Name + "</h1><br />";
        htmlString += "<h3>Costs: " + Price.ToString() + "</h3><br />";
        htmlString += "<img src='" + ImageUrl + "' />";
        return htmlString;
    }
}
```

All the GetHtml() method does is read the private data and format it in some attractive way. You can take this block of HTML and place it on a web page to represent the product. This really targets the class as a user interface class rather than as a pure data class or "business object."

Adding a Constructor

Currently, the Product class has a problem. Ideally, classes should ensure that they are always in a valid state. However, unless you explicitly set all the appropriate properties, the Product object won't correspond to a valid product. This could cause an error if you try to use a method that relies on some of the data that hasn't been supplied. To solve this problem, you need to equip your class with one or more constructors.

A *constructor* is a method that automatically runs when the class is first created. In C#, the constructor always has the same name as the name of the class. Unlike a normal method, the constructor doesn't define any return type, not even void.

The next code example shows a new version of the Product class. It adds a constructor that requires the product price and name as arguments:

```
public class Product
{
    // (Additional class code omitted for clarity.)

    public Product(string name, decimal price)
    {
        // Set two properties in the class.
        Name = name;
        Price = price;
    }
}
```

Here's an example of the code you need to create an object based on the new Product class, using its constructor:

```
Product saleProduct = new Product("Kitchen Garbage", 49.99M);
```

The preceding code is much leaner than the code that was required to create and initialize the previous version of the Product class. With the help of the constructor, you can create a Product object and configure it with the basic data it needs in a single line.

If you don't create a constructor, .NET supplies a default public constructor that does nothing. If you create at least one constructor, .NET will not supply a default constructor. Thus, in the preceding example, the Product class has exactly one constructor, which is the one that is explicitly defined in code. To create a Product object, you *must* use this constructor. This restriction prevents a client from creating an object without specifying the bare minimum amount of data that's required:

```
// This will not be allowed, because there is
// no zero-argument constructor.
Product saleProduct = new Product();
```

■ **Note** To create an instance of a class, you need to use a constructor. The preceding code fails because it attempts to use a zero-argument constructor, which doesn't exist in the Product class.

Most of the classes you use will have constructors that require parameters. As with ordinary methods, constructors can be overloaded with multiple versions, each providing a different set of parameters.

```
public class Product
{
    // (Additional class code omitted for clarity.)

    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }

    public Product(string name, decimal price, string imageUrl)
    {
        Name = name;
        Price = price;
        ImageUrl = imageUrl;
    }
}
```

When creating an object, you can choose the constructor that suits you best based on the information that you have available. The .NET Framework classes use overloaded constructors extensively.

Adding an Event

Classes can also use events to notify your code. To define an event in C#, you must first create a delegate that defines the signature for the event you're going to use. Here's an example that creates an event with no parameters and no return value:

```
public delegate void PriceChangedEventHandler();
```

(If you've forgotten exactly what a delegate is, flip back to the ending of Chapter 2 to refresh your memory before continuing.)

Once you have a delegate, you can use the `event` keyword to define an event based on that delegate. As with properties and methods, events can be declared with different accessibilities, although public events are the default. Usually, this is what you want, because you'll use the events to allow one object to notify another object that's an instance of a different class.

As an illustration, the Product class example has been enhanced with a PriceChanged event that occurs whenever the price is modified through the Price property procedure. This event won't fire if code inside the class changes the underlying private price variable without going through the property procedure:

```
// Define the delegate that represents the event.
public delegate void PriceChangedEventHandler();

public class Product
{
    // (Additional class code omitted for clarity.)

    // Define the event using the delegate.
    public event PriceChangedEventHandler PriceChanged;
```

```
    public decimal Price
    {
        get
        {
            return price;
        }
        set
        {
            price = value;

            // Fire the event, provided there is at least one listener.
            if (PriceChanged != null)
            {
                PriceChanged();
            }
        }
    }
}
```

To fire an event, you just call it by name. However, before firing an event, you must check that at least one subscriber exists by testing whether the event reference is null. If it isn't null, it's safe to fire the event. If you attempt to fire the event when there are no listeners, you'll derail your code with a runtime error.

It's quite possible that you'll create dozens of ASP.NET applications without once defining a custom event. However, you'll be hard-pressed to write a single ASP.NET web page without *handling* an event. To handle an event, you first create a method called an *event handler*. The event handler contains the code that should be executed when the event occurs. Then you connect the event handler to the event.

To handle the Product.PriceChanged event, you need to begin by creating an event handler, which you'll usually place in another class. The event handler needs to have the same signature as the event it's handling. In the Product example, the PriceChanged event has no parameters, so the event handler would look like the simple method shown here:

```
public void ChangeDetected()
{
    // This code executes in response to the PriceChanged event.
}
```

The next step is to hook up the event handler to the event. To do this, you use a simple assignment statement that sets the event (PriceChanged) to the event-handling method (ChangeDetected) by using the += operator:

```
Product saleProduct = new Product("Kitchen Garbage", 49.99M);

// This connects the saleProduct.PriceChanged event to an event-handling
// procedure called ChangeDetected.
// Note that ChangedDetected needs to match the PriceChangedEventHandler
// delegate.
saleProduct.PriceChanged += ChangeDetected;

// Now the event will occur in response to this code:
saleProduct.Price = saleProduct.Price * 2;
```

This code attaches an event handler to a method named ChangeDetected. This method is in the same class as the event hookup code shown here, and for that reason you don't need to specify the object name when you attach the event handler. If you want to connect an event to a different object, you'd need to use the dot syntax when referring the event handler method, as in myObject.ChangeDetected.

It's worth noting that if you're using Visual Studio, you won't need to manually hook up event handlers for web controls at all. Instead, Visual Studio can add the code you need to connect all the event handlers you create.

ASP.NET uses an *event-driven* programming model, so you'll soon become used to writing code that reacts to events. But unless you're creating your own components, you won't need to fire your own custom events. For an example where custom events make sense, refer to Chapter 11, which discusses how you can add an event to a user control you've created.

■ **Tip** You can also detach an event handler by using the `-=` operator instead of `+=`.

Testing the Product Class

To learn a little more about how the Product class works, it helps to create a simple web page. This web page will create a Product object, get its HTML representation, and then display it in the web page. To try this example, you'll need to use the three files that are provided with the online samples in the Chapter03\Website folder:

Product.cs: This file contains the code for the Product class. It's in the Chapter03\Website\App_Code subfolder, which allows ASP.NET to compile it automatically.

Garbage.jpg: This is the image that the Product class will use.

Default.aspx: This file contains the web page code that uses the Product class.

The easiest way to test this example is to use Visual Studio. Here are the steps you need to perform the test:

1. Start Visual Studio.
2. Choose File ➤ Open ➤ Web Site from the menu.
3. In the Open Web Site dialog box, browse to the Chapter03 folder, double-click it, select the Website folder inside, and click Open.
4. Choose Debug ➤ Start Without Debugging to launch the website. Visual Studio will open a new window with your default browser and navigate to the Default.aspx page.

When the Default.aspx page executes, it creates a new Product object, configures it, and uses the GetHtml() method. The HTML is written to the web page by using the Response.Write() method. Here's the code:

```
<%@ Page Language="C#" %>
<script runat="server">
    private void Page_Load(object sender, EventArgs e)
    {
        Product saleProduct = new Product("Kitchen Garbage", 49.99M, "garbage.jpg");
        Response.Write(saleProduct.GetHtml());
    }
</script>

<html>
    <head>
        <title>Product Test</title>
    </head>
    <body></body>
</html>
```

The <script> block uses the `runat="server"` attribute setting to run the code on the web server, rather than in the user's web browser. The <script> block holds a subroutine named Page_Load. This subroutine is triggered when the page is first created. After this code is finished, the HTML is sent to the client. Figure 3-2 shows the web page you'll see.

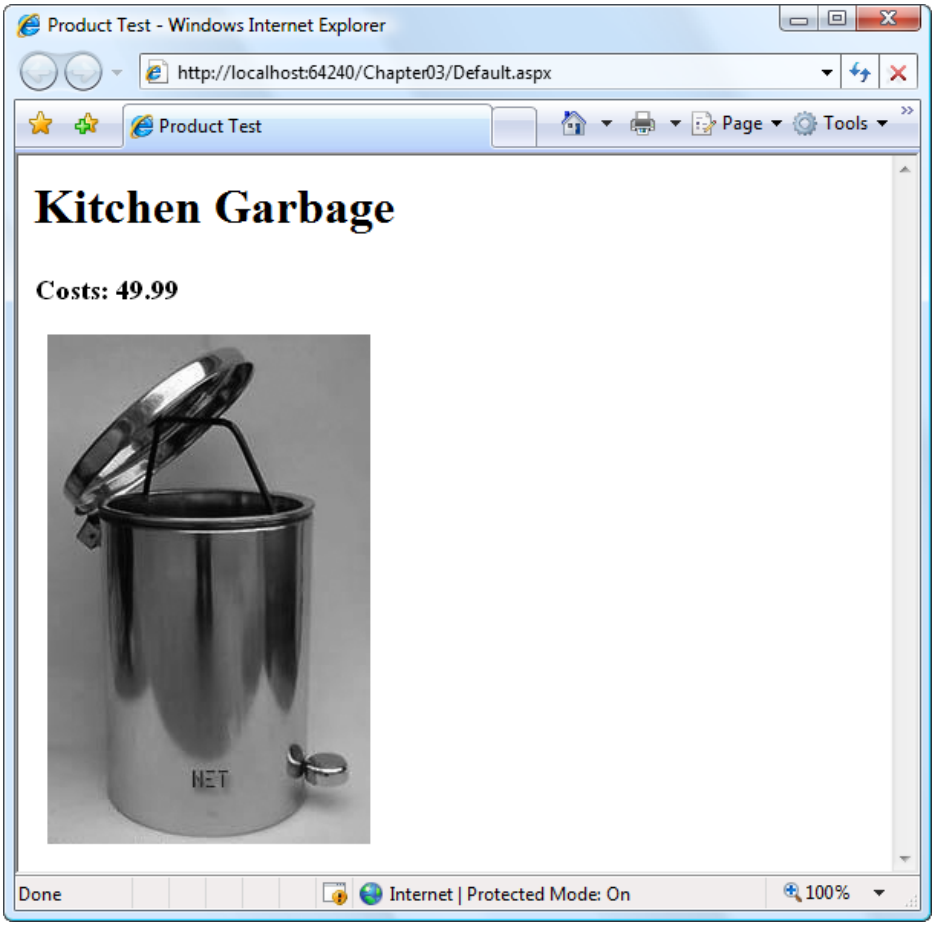

Figure 3-2. *Output generated by a Product object*

Interestingly, the GetHtml() method is similar to how an ASP.NET web control works, but on a much cruder level. To use an ASP.NET control, you create an object (explicitly or implicitly) and configure some properties. Then ASP.NET automatically creates a web page by examining all these objects and requesting their associated HTML (by calling a hidden GetHtml() method or by doing something conceptually similar[1]). It then sends the completed page to the user. The end result is that you work with objects, instead of dealing directly with raw HTML code.

[1] Actually, the ASP.NET engine calls a method named Render() in every web control.

When using a web control, you see only the public interface made up of properties, methods, and events. However, understanding how class code actually works will help you master advanced development.

Now that you've seen the basics of classes and a demonstration of how you can use a class, it's time to introduce a little more theory about .NET objects and revisit the basic data types introduced in the previous chapter.

Value Types and Reference Types

In Chapter 2, you learned how simple data types such as strings and integers are actually objects created from the class library. This allows some impressive tricks, such as built-in string handling and date calculation. However, simple data types differ from more complex objects in one important way: simple data types are *value types,* while classes are *reference types.*

This means that a variable for a simple data type contains the actual information you put in it (such as the number 7). On the other hand, object variables store a reference that points to a location in memory where the full object is stored. In most cases, .NET masks you from this underlying reality, and in many programming tasks you won't notice the difference. However, in three cases you will notice that object variables act a little differently than ordinary data types: in assignment operations, in comparison operations, and when passing parameters.

Assignment Operations

When you assign a simple data variable to another simple data variable, the contents of the variable are copied:

```
integerA = integerB;  // integerA now has a copy of the contents of integerB.
                      // There are two duplicate integers in memory.
```

Reference types work a little differently. Reference types tend to deal with larger amounts of data. Copying the entire contents of a reference type object could slow down an application, particularly if you are performing multiple assignments. For that reason, when you assign a reference type, you copy the reference that *points* to the object, not the full object content:

```
// Create a new Product object.
Product productVariable1 = new Product("Kitchen Garbage", 49.99M);

// Declare a second variable.
Product productVariable2;
productVariable2 = productVariable1;

// productVariable1 and productVariable2 now both point to the same thing.
// There is one object and two ways to access it.
```

The consequences of this behavior are far ranging. This example modifies the Product object by using productVariable2:

```
productVariable2.Price = 25.99M;
```

You'll find that productVariable1.Price is also set to 25.99. Of course, this only makes sense because productVariable1 and productVariable2 are two variables that point to the same in-memory object.

If you really do want to copy an object (not a reference), you need to create a new object and then initialize its information to match the first object. Some objects provide a Clone() method that allows you to easily copy the object. One example is the DataSet, which is used to store information from a database.

Equality Testing

A similar distinction between reference types and value types appears when you compare two variables. When you compare value types (such as integers), you're comparing the contents:

```
if (integerA == integerB)
{
    // This is true as long as the integers have the same content.
}
```

When you compare reference type variables, you're actually testing whether they're the same instance. In other words, you're testing whether the references are pointing to the same object in memory, not whether their contents match:

```
if (productVariable1 == productVariable2)
{
    // This is true if both productVariable1 and productVariable2
    // point to the same thing.
    // This is false if they are separate, yet identical, objects.
}
```

■ **Note** This rule has a special exception. When classes override the == operator, they can change what type of comparison it performs. The only significant example of this technique in .NET is the String class. For more information, read the section "Reviewing .NET Types" later in this chapter.

Passing Parameters by Reference and by Value

You can create three types of method parameters. The standard type is *pass-by-value*. When you use pass-by-value parameters, the method receives a copy of the parameter data. That means if the method modifies the parameter, this change won't affect the code that called the method. By default, all parameters are pass-by-value.

The second type of parameter is *pass-by-reference*. With pass-by-reference, the method accesses the parameter value directly. If a method changes the value of a pass-by-reference parameter, the original object is also modified.

To get a better understanding of the difference, consider the following code, which shows a method that uses a parameter named *number*. This code uses the `ref` keyword to indicate that number should be passed by reference. When the method modifies this parameter (multiplying it by 2), the calling code is also affected:

```
private void ProcessNumber(ref int number)
{
    number *= 2;
}
```

The following code snippet shows the effect of calling the ProcessNumber() method. Note that you need to specify the `ref` keyword when you define the parameter in the method and when you call the method. This indicates that you are aware that the parameter value may change:

```
int num = 10;
ProcessNumber(ref num);        // Once this call completes, Num will be 20.
```

If you don't include the `ref` keyword, you'll get an error when you attempt to compile the code.

The way that pass-by-value and pass-by-reference work when you're using value types (such as integers) is straightforward. However, if you use reference types, such as a Product object or an array, you won't see this

behavior. The reason is that the entire object isn't passed in the parameter. Instead, it's just the *reference* that's transmitted. This is much more efficient for large objects (it saves having to copy a large block of memory), but it doesn't always lead to the behavior you expect.

One notable quirk occurs when you use the standard pass-by-value mechanism. In this case, pass-by-value doesn't create a copy of the object, but a copy of the *reference*. This reference still points to the same in-memory object. This means that if you pass a Product object to a method, for example, the method will be able to alter your Product object, regardless of whether you use pass-by-value or pass-by-reference. (The only limitation is that if you use pass-by-value, you won't be able to *change* the reference—for example, replace the object with a completely new one that you create.)

C# also supports a third type of parameter: the output parameter. To use an output parameter, precede the parameter declaration with the keyword `out`. Output parameters are commonly used as a way to return multiple pieces of information from a single method.

When you use output parameters, the calling code can submit an uninitialized variable as a parameter, which is otherwise forbidden. This approach wouldn't be appropriate for the ProcessNumber() method, because it reads the submitted parameter value (and then doubles it). If, on the other hand, the method used the parameter just to return information, you could use the `out` keyword, as shown here:

```
private void ProcessNumber(int number, out int doubled, out int tripled)
{
    doubled = number * 2;
    tripled = number * 3;
}
```

Remember, output parameters are designed solely for the method to return information to your calling code. In fact, the method won't be allowed to retrieve the value of an `out` parameter, because it may be uninitialized. The only action the method can take is to set the output parameter.

Here's an example of how you can call the revamped ProcessNumber() method:

```
int num = 10;
int doubled, tripled;
ProcessNumber(num, out doubled, out tripled);
```

Reviewing .NET Types

So far, the discussion has focused on simple data types and classes. The .NET class library is actually composed of *types,* which is a catchall term that includes several object-like relatives:

Classes: This is the most common type in the .NET Framework. Strings and arrays are two examples of .NET classes, although you can easily create your own.

Structures: Structures, like classes, can include fields, properties, methods, and events. Unlike classes, they are value types, which alters the way they behave with assignment and comparison operations. Structures also lack some of the more advanced class features (such as inheritance) and are generally simpler and smaller. Integers, dates, and chars are all structures.

Enumerations: An enumeration defines a set of integer constants with descriptive names. Enumerations were introduced in the previous chapter.

Delegates: A delegate is a function pointer that allows you to invoke a procedure indirectly. Delegates are the foundation for .NET event handling and were introduced in the previous chapter.

Interfaces: They define contracts to which a class must adhere. Interfaces are an advanced technique of object-oriented programming, and they're useful when standardizing how objects interact. Interfaces aren't discussed in this book.

Occasionally, a class can override its behavior to act more like a value type. For example, the String type is a full-featured class, not a simple value type. (This is required to make strings efficient, because they can contain a variable amount of data.) However, the String type overrides its equality and assignment operations so that these operations work like those of a simple value type. This makes the String type work in the way that programmers intuitively expect. Arrays, on the other hand, are reference types through and through. If you assign one array variable to another, you copy the reference, not the array (although the Array class also provides a Clone() method that returns a duplicate array to allow true copying).

Table 3-2 sets the record straight and explains a few common types.

Table 3-2. *Common Reference and Value Types*

Data Type	Nature	Behavior
Int32, Decimal, Single, Double, and all other basic numeric types	Value type	Equality and assignment operations work with the variable contents, not a reference.
DateTime, TimeSpan	Value type	Equality and assignment operations work with the variable contents, not a reference.
Char, Byte, and Boolean	Value type	Equality and assignment operations work with the variable contents, not a reference.
String	Reference type	Equality and assignment operations appear to work with the variable contents, not a reference.
Array	Reference type	Equality and assignment operations work with the reference, not the contents.

Understanding Namespaces and Assemblies

Whether you realize it at first, every piece of code in .NET exists inside a .NET type (typically a class). In turn, every type exists inside a namespace. Figure 3-3 shows this arrangement for your own code and the DateTime class. Keep in mind that this is an extreme simplification—the System namespace alone is stocked with several hundred classes. This diagram is designed only to show you the layers of organization.

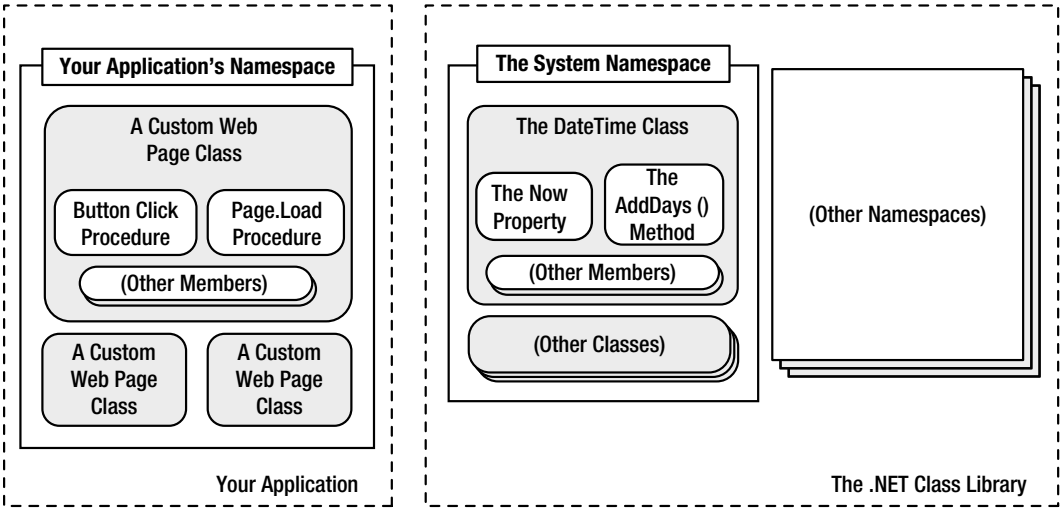

Figure 3-3. *A look at two namespaces*

Namespaces can organize all the different types in the class library. Without namespaces, these types would all be grouped into a single long and messy list. This sort of organization is practical for a small set of information, but it would be impractical for the thousands of types included with .NET.

Many of the chapters in this book introduce you to new .NET classes and namespaces. For example, in the chapters on web controls, you'll learn how to use the objects in the System.Web.UI namespace. In the chapters about web services, you'll study the types in the System.Web.Services namespace. For databases, you'll turn to the System.Data namespace. In fact, you've already learned a little about one namespace: the basic System namespace that contains all the simple data types explained in the previous chapter.

To continue your exploration after you've finished the book, you'll need to turn to the class library reference on Microsoft's MSDN website (`http://msdn.microsoft.com/library/ms229335.aspx`. It painstakingly documents the properties, methods, and events of every class in every namespace.

To browse the class library reference, scroll down the list of namespaces in the Namespaces section on the page. When you find a namespace you want to investigate, click its name. When you do, a new page appears, which features a list of all the classes in that namespace. Figure 3-4 shows what you'll see in the System.Web.UI namespace.

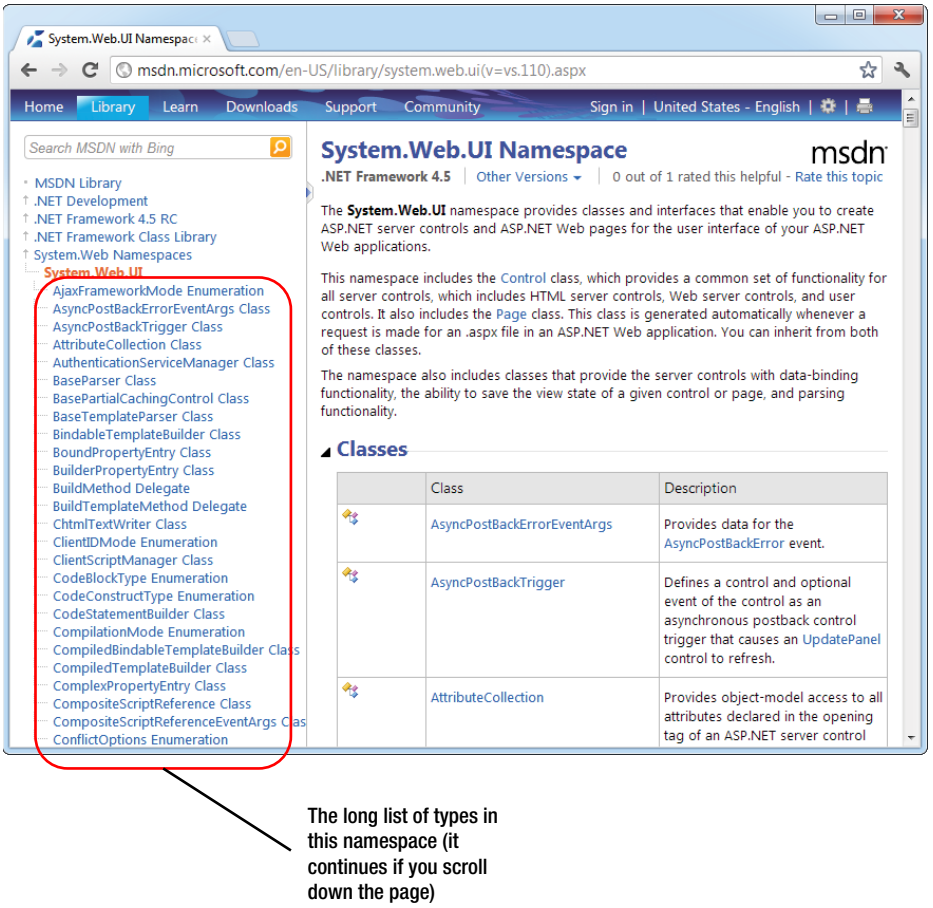

Figure 3-4. *The online class library reference*

Next, click a class name (for example, Page). Now the page shows an overview of the Page class and additional links that let you view the different types of class members (Figure 3-5). For example, click Page Properties to review all the properties of the Page class, or Page Events to explore its events.

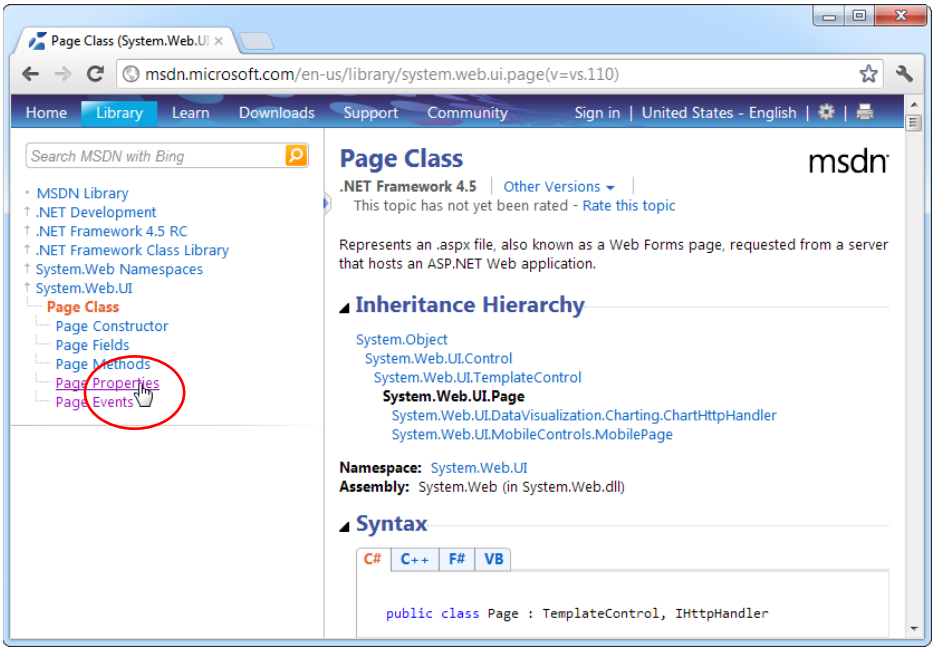

Figure 3-5. *Exploring the members of a specific class*

Using Namespaces

Often when you write ASP.NET code, you'll just use the namespace that Visual Studio creates automatically. If, however, you want to organize your code into multiple namespaces, you can define the namespace by using a simple block structure, as shown here:

```
namespace MyCompany
{
    namespace MyApp
    {
        public class Product
        {
            // Code goes here.
        }
    }
}
```

In the preceding example, the Product class is in the namespace MyCompany.MyApp. Code inside this namespace can access the Product class by name. Code outside it needs to use the fully qualified name, as in MyCompany.MyApp.Product. This ensures that you can use the components from various third-party developers without worrying about a name collision. If those developers follow the recommended naming standards, their classes will always be in a namespace that uses the name of their company and software product. The fully qualified name of a class will then almost certainly be unique.

Namespaces don't take an accessibility keyword and can be nested as many layers deep as you need. Nesting is purely cosmetic—for example, in the previous example, no special relationship exists between the MyCompany namespace and the MyApp namespace. In fact, you could create the namespace MyCompany.MyApp without using nesting at all, by using this syntax:

```
namespace MyCompany.MyApp
{
    public class Product
    {
        // Code goes here.
    }
}
```

You can declare the same namespace in various code files. In fact, more than one project can even use the same namespace. Namespaces are really nothing more than a convenient, logical container that helps you organize your classes.

Importing Namespaces

Having to type long, fully qualified names is certain to tire your fingers and create overly verbose code. To tighten code up, it's standard practice to import the namespaces you want to use. After you have imported a namespace, you don't need to type the fully qualified name for any of the types it contains. Instead, you can use the types in that namespace as though they were defined locally.

To import a namespace, you use the `using` statement. These statements must appear as the first lines in your code file, outside of any namespaces or block structures:

```
using MyCompany.MyApp;
```

Consider the situation without importing a namespace:

```
MyCompany.MyApp.Product salesProduct = new MyCompany.MyApp.Product(...);
```

It's much more manageable when you import the MyCompany.MyApp namespace. When you do, you can use this syntax instead:

```
Product salesProduct = new Product(...);
```

Importing namespaces is really just a convenience. It has no effect on the performance of your application. In fact, whether or not you use namespace imports, the compiled IL code will look the same. That's because the language compiler will translate your relative class references into fully qualified class names when it generates an EXE or a DLL file.

STREAMLINED OBJECT CREATION

Even if you choose not to import a namespace, you can compress any statement that declares an object variable and instantiates it. The trick is to use the `var` keyword you learned about in Chapter 2.

For example, you can replace this statement:

```
MyCompany.MyApp.Product salesProduct = new MyCompany.MyApp.Product();
```

with this:

```
var salesProduct = new MyCompany.MyApp.Product();
```

This works because the compiler can determine the correct data type for the salesProduct variable based on the object you're creating with the `new` keyword. Best of all, this statement is just as readable as the non-`var` approach, because it's still clear what type of object you're creating.

Of course, this technique won't work if the compiler can't determine the type of object you want. For that reason, neither of these statements is allowed:

```
var salesProductInvalid1;
var salesProductInvalid2 = null;
```

Furthermore, the `var` trick is limited to local variables. You can't use it when declaring the member variables of a class.

Using Assemblies

You might wonder what gives you the ability to use the class library namespaces in a .NET program. Are they hardwired directly into the language? The truth is that all .NET classes are contained in *assemblies*. Assemblies are the physical files that contain compiled code. Typically, assembly files have the extension .exe if they are stand-alone applications, or .dll if they're reusable components.

■ **Tip** The .dll extension is also used for code that needs to be executed (or *hosted*) by another type of program. When your web application is compiled, it's turned into a DLL file, because your code doesn't represent a stand-alone application. Instead, the ASP.NET engine executes it when a web request is received.

You can find the core assemblies for the .NET Framework in the folder C:\Windows\Assembly. (Technically, the assemblies are in *subdirectories* of the C:\Windows\Assembly folder, which allows .NET to manage versioning. However, Windows Explorer hides this fact.)

A strict relationship doesn't exist between assemblies and namespaces. An assembly can contain multiple namespaces. Conversely, more than one assembly file can contain classes in the same namespace. Technically, namespaces are a *logical* way to group classes. Assemblies, however, are a *physical* package for distributing code.

The .NET classes are actually contained in a number of assemblies. For example, the basic types in the System namespace come from the mscorlib.dll assembly. Many ASP.NET types are found in the System.Web.dll assembly. In addition, you might want to use other, third-party assemblies. Often, assemblies and namespaces have the same names. For example, you'll find the namespace System.Web in the assembly file System.Web.dll. However, this is a convenience, not a requirement.

When compiling an application, you need to tell the language compiler what assemblies the application uses. By default, a wide range of .NET assemblies are automatically made available to ASP.NET applications. If you need to use additional assemblies, you need to define them in a configuration file for your website. Visual Studio makes this process seamless, letting you add assembly references to the configuration file by using the Website ➤ Add Reference command. You'll use the Add Reference command in Chapter 22.

Advanced Class Programming

Part of the art of object-oriented programming is determining object relations. For example, you could create a Product object that contains a ProductFamily object or a Car object that contains four Wheel objects. To create this sort of object relationship, all you need to do is define the appropriate variable or properties in the class. This type of relationship is called *containment* (or *aggregation*).

For example, the following code shows a ProductCatalog class, which holds an array of Product objects:

```
public class ProductCatalog
{
    private Product[] products;

    // (Other class code goes here.)
}
```

In ASP.NET programming, you'll find special classes called *collections* that have no purpose other than to group various objects. Some collections also allow you to sort and retrieve objects by using a unique name. In the previous chapter, you saw an example with the ArrayList from the System.Collections namespace, which provides a dynamically resizable array. Here's how you might use the ArrayList to modify the ProductCatalog class:

```
public class ProductCatalog
{
    private ArrayList products = new ArrayList();

    // (Other class code goes here.)
}
```

This approach has benefits and disadvantages. It makes it easier to add and remove items from the list, but it also removes a useful level of error checking, because the ArrayList supports any type of object. You'll learn more about this issue later in this chapter (in the "Generics" section).

In addition, classes can have a different type of relationship known as *inheritance*.

Inheritance

Inheritance is a form of code reuse. It allows one class to acquire and extend the functionality of another class. For example, you could create a class called TaxableProduct that inherits (or *derives*) from Product. The TaxableProduct class would gain all the same fields, methods, properties, and events of the Product class. (However, it wouldn't inherit the constructors.) You could then add additional members that relate to taxation.

Here's an example that adds a read-only property named TotalPrice:

```
public class TaxableProduct : Product
{
    private decimal taxRate = 1.15M;

    public decimal TotalPrice
    {
        get
        {
            // The code can access the Price property because it's
            // a public part of the base class Product.
            // The code cannot access the private price variable, however.
            return (Price * taxRate);
        }
    }

    // Create a three-argument constructor that calls the three-argument constructor
    // from the Product class.
```

```
    public TaxableProduct(string name, decimal price, string imageUrl) :
      base(name, price, imageUrl)
    { }
}
```

There's an interesting wrinkle in this example. With inheritance, constructors are never inherited. However, you still need a way to initialize the inherited details (in this case, that's the Name, Price, and ImageUrl properties).

The only way to handle this problem is to add a constructor in your derived class (TaxableProduct) that calls the right constructor in the base class (Product) by using the `base` keyword. In the previous example, the TaxableProduct class uses this technique. It includes a single constructor that requires the familiar three arguments and calls the corresponding three-argument constructor from the Product class to initialize the Name, Price, and ImageUrl properties. The TaxableProduct constructor doesn't contain any additional code, but it could; for example, you could use it to initialize other details that are specific to the derived class.

Inheritance is less useful than you might expect. In an ordinary application, most classes use containment and other relationships instead of inheritance, because inheritance can complicate life needlessly without delivering many benefits. Dan Appleman, a renowned .NET programmer, once described inheritance as "the coolest feature you'll almost never use."

However, you'll see inheritance at work in ASP.NET in at least one place. Inheritance allows you to create a custom class that inherits the features of a class in the .NET class library. For example, when you create a custom web form, you inherit from a basic Page class to gain the standard set of features.

There are many more subtleties of class-based programming with inheritance. For example, you can override parts of a base class, prevent classes from being inherited, or create a class that must be used for inheritance and can't be directly created. However, these topics aren't covered in this book, and they aren't required to build ASP.NET applications. For more information about these language features, consult a more detailed book that covers the C# language, such as Andrew Troelsen's *Pro C# and the .NET 4.5 Framework* (Apress, 2012).

Static Members

The beginning of this chapter introduced the idea of static properties and methods, which can be used without a live object. Static members are often used to provide useful functionality related to an object. The .NET class library uses this technique heavily (as with the System.Math class explored in the previous chapter).

Static members have a wide variety of possible uses. Sometimes they provide basic conversions and utility functions that support a class. To create a static property or method, you just need to use the `static` keyword right after the accessibility keyword.

The following example shows a TaxableProduct class that contains a static TaxRate property and private variable. This means there is one copy of the tax rate information, and it applies to all TaxableProduct objects:

```
public class TaxableProduct : Product
{
    // (Additional class code omitted for clarity.)

    private static decimal taxRate = 1.15M;

    // Now you can call TaxableProduct.TaxRate, even without an object.
    public static decimal TaxRate
    {
        get
        { return taxRate; }
        set
        { taxRate = value; }
    }
}
```

■ **Note** This version of the TaxableProduct class still includes the details added in the previous section (such as the TotalPrice property and the three-argument constructor). They're just left out of the listing to focus on the newly added details.

You can now retrieve the tax rate information directly from the class, without needing to create an object first:

```
// Change the TaxRate. This will affect all TotalPrice calculations for any
// TaxableProduct object.
TaxableProduct.TaxRate = 1.24M;
```

Static data isn't tied to the lifetime of an object. In fact, it's available throughout the life of the entire application. This means static members are the closest thing .NET programmers have to global data.

A static member can't access an instance member. To access a nonstatic member, it needs an actual instance of your object.

■ **Tip** You can create a class that's entirely composed of static members. Just add the `static` keyword to the declaration, as in the following:

```
public static class TaxableUtil
```

When you declare a class with the `static` keyword, you ensure that it can't be instantiated. However, you still need to use the `static` keyword when declaring static members in your static class.

Casting Objects

Object variables can be converted with the same syntax that's used for simple data types. This process is called *casting*. When you perform casting, you don't actually change anything about an object; in fact, it remains the exact same blob of binary data floating somewhere in memory. What you change is the variable that points to the object—in other words, the way your code "sees" the object. This is important, because the way your code sees an object determines what you can do with that object.

An object variable can be cast into one of three things: itself, an interface that it supports, or a base class from which it inherits. You can't cast an object variable into a string or an integer. Instead, you need to call a conversion method, if it's available, such as ToString() or Parse().

As you've already seen, the TaxableProduct class derives from Product. That means you cast a TaxableProduct reference to a Product reference, as shown here:

```
// Create a TaxableProduct.
TaxableProduct theTaxableProduct =
  new TaxableProduct("Kitchen Garbage", 49.99M, "garbage.jpg");

// Cast the TaxableProduct reference to a Product reference.
Product theProduct = theTaxableProduct;
```

You don't lose any information when you perform this casting. There is still just one object in memory (with two variables pointing to it), and this object really *is* a TaxableProduct. However, when you use the variable theProduct to access your TaxableProduct object, you'll be limited to the properties and methods that are defined in the Product class. That means code like this won't work:

```
// This code generates a compile-time error.
decimal TotalPrice = theProduct.TotalPrice;
```

Even though theProduct actually holds a reference that points to a TaxableProduct and even though the TaxableProduct has a TotalPrice property, you can't access it through theProduct. That's because theProduct treats the object it refers to as an ordinary Product.

You can also cast in the reverse direction—for example, cast a Product reference to a TaxableProduct reference. The trick here is that this works only if the object that's in memory really is a TaxableProduct. This code is correct:

```
Product theProduct = new TaxableProduct(...);
TaxableProduct theTaxableProduct = (TaxableProduct)theProduct;
```

But this code generates a runtime error:

```
Product theProduct = new Product(...);
TaxableProduct theTaxableProduct = (TaxableProduct)theProduct;
```

■ **Note** When casting an object from a base class to a derived class, as in this example, you must use the explicit casting syntax that you learned about in Chapter 2. That means you place the data type in parentheses before the variable that you want to cast. This is a safeguard designed to highlight the fact that casting is taking place. It's required because this casting operation might fail.

Incidentally, you can check whether you have the right type of object before you attempt to cast with the help of the `is` keyword:

```
if (theProduct is TaxableProduct)
{
    // It's safe to cast the reference.
    TaxableProduct theTaxableProduct = (TaxableProduct)theProduct;
}
```

Another option is the `as` keyword, which attempts to cast an object to the type you request but returns a null reference if it can't (rather than causing a runtime error). Here's how it works:

```
Product theProduct = new TaxableProduct(...);
TaxableProduct theTaxableProduct = theProduct as TaxableProduct;
if (theTaxableProduct != null)
{
    // (It's safe to use the object.)
}
else
{
    // (Either the conversion failed or theTaxableProduct was null to begin with.)
}
```

Keep in mind that this approach may simply postpone the problem, by replacing an immediate casting error with a null-reference error later in your code, when you attempt to use the null object. However, you can use this technique in much the same way that you use the `is` keyword—to cast an object if possible or just to keep going if it's not.

■ **Note** One of the reasons casting is used is to facilitate more reusable code. For example, you might design an application that uses the Product object. That application is actually able to handle an instance of any Product-derived class. Your application doesn't need to distinguish between all the different derived classes (TaxableProduct, NonTaxableProduct, PromoProduct, and so on); it can work seamlessly with all of them.

At this point, it might seem that being able to convert objects is a fairly specialized technique that will be required only when you're using inheritance. This isn't always true. Object conversions are also required when you use some particularly flexible classes.

One example is the ArrayList class introduced in the previous chapter. The ArrayList is designed in such a way that it can store any type of object. To have this ability, it treats all objects in the same way—as instances of the root System.Object class. (Remember, all classes in .NET inherit from System.Object at some point, even if this relationship isn't explicitly defined in the class code.) The end result is that when you retrieve an object from an ArrayList collection, you need to cast it from a System.Object to its real type, as shown here:

```
// Create the ArrayList.
ArrayList products = new ArrayList();

// Add several Product objects.
products.Add(product1);
products.Add(product2);
products.Add(product3);

// Retrieve the first item, with casting.
Product retrievedProduct = (Product)products[0];

// This works.
Response.Write(retrievedProduct.GetHtml());

// Retrieve the first item, as an object. This doesn't require casting,
// but you won't be able to use any of the Product methods or properties.
Object retrievedObject = products[0];

// This generates a compile error. There is no Object.GetHtml() method.
Response.Write(retrievedObject.GetHtml());
```

As you can see, if you don't perform the casting, you won't be able to use the methods and properties of the object you retrieve. You'll find many cases like this in .NET code, where your code is handed one of several possible object types and it's up to you to cast the object to the correct type in order to use its full functionality.

Partial Classes

Partial classes give you the ability to split a single class into more than one C# source code (.cs) file. For example, if the Product class became particularly long and intricate, you might decide to break it into two pieces, as shown here:

```
// This part is stored in file Product1.cs.
public partial class Product
{
    public string Name { get; set; }

    public event PriceChangedEventHandler PriceChanged;
    private decimal price;
    public decimal Price
    {
        get
        {
            return price;
        }
        set
        {
            price = value;

            // Fire the event, provided there is at least one listener.
            if (PriceChanged != null)
            {
                PriceChanged();
            }
        }
    }

    public string ImageUrl { get; set;}

    public Product(string name, decimal price, string imageUrl)
    {
        Name = name;
        Price = price;
        ImageUrl = imageUrl;
    }
}

// This part is stored in file Product2.cs.
public partial class Product
{
    public string GetHtml()
    {
        string htmlString;
        htmlString = "<h1>" + Name + "</h1><br />";
        htmlString += "<h3>Costs: " + Price.ToString() + "</h3><br />";
        htmlString += "<img src='" + ImageUrl + "' />";
        return htmlString;
    }
}
```

A partial class behaves the same as a normal class. This means every method, property, and variable you've defined in the class is available everywhere, no matter which source file contains it. When you compile the application, the compiler tracks down each piece of the Product class and assembles it into a complete unit. It doesn't matter what you name the source code files, so long as you keep the class name consistent.

Partial classes don't offer much in the way of solving programming problems, but they can be useful if you have extremely large, unwieldy classes. The real purpose of partial classes in .NET is to hide automatically generated designer code by placing it in a separate file from your code. Visual Studio uses this technique when you create web pages for a web application and forms for a Windows application.

■ **Note** Every fragment of a partial class must use the `partial` keyword in the class declaration.

Generics

Generics are a more subtle and powerful feature than partial classes. Generics allow you to create classes that are parameterized by type. In other words, you create a class template that supports any type. When you instantiate that class, you specify the type you want to use, and from that point on, your object is "locked in" to the type you chose.

To understand how this works, it's easiest to consider some of the .NET classes that support generics. In the previous chapter (and earlier in this chapter), you saw how the ArrayList class allows you to create a dynamically sized collection that expands as you add items and shrinks as you remove them. The ArrayList has one weakness, however—it supports any type of object. This makes it extremely flexible, but it also means you can inadvertently run into an error. For example, imagine you use an ArrayList to track a catalog of products. You intend to use the ArrayList to store Product objects, but there's nothing to stop a piece of misbehaving code from inserting strings, integers, or any arbitrary object in the ArrayList. Here's an example:

```
// Create the ArrayList.
ArrayList products = new ArrayList();

// Add several Product objects.
products.Add(product1);
products.Add(product2);
products.Add(product3);

// Notice how you can still add other types to the ArrayList.
products.Add("This string doesn't belong here.");
```

The solution is a new List collection class. Like the ArrayList, the List class is flexible enough to store different objects in different scenarios. But because it supports generics, you can lock it into a specific type whenever you instantiate a List object. To do this, you specify the class you want to use in angle brackets after the class name, as shown here:

```
// Create the List for storing Product objects.
List<Product> products = new List<Product>();
```

Now you can add only Product objects to the collection:

```
// Add several Product objects.
products.Add(product1);
products.Add(product2);
products.Add(product3);
```

```
// This line fails. In fact, it won't even compile.
products.Add("This string can't be inserted.");
```

To figure out whether a class uses generics, look for the angle brackets. For example, the List class is listed as List<T> in the .NET Framework documentation to emphasize that it takes one type parameter. You can find this class, and many more collections that use generics, in the System.Collections.Generic namespace. (The original ArrayList resides in the System.Collections namespace.)

■ **Note** Now that you've seen the advantage of the List class, you might wonder why .NET includes the ArrayList at all. In truth, the ArrayList is still useful if you really do need to store different types of objects in one place (which isn't terribly common). However, the real answer is that generics weren't implemented in .NET until version 2.0, so many existing classes don't use them because of backward compatibility.

You can also create your own classes that are parameterized by type, like the List collection. Creating classes that use generics is beyond the scope of this book, but you can find a solid overview at http://tinyurl.com/39sa5q3 if you're still curious.

The Last Word

This chapter has presented a quick and compressed overview of object-oriented programming in .NET. You've seen how to define classes and instantiate objects, how value types and reference types work behind the scenes, and how to access the rich set of namespaces that .NET has to offer. Along the way, you also saw a simple example that used a custom object to insert HTML into a web page. Surprisingly enough, this crude example roughly approximates the way ASP.NET uses web controls (a specialized type of .NET object) when it renders a web page. You'll gain much more insight into this process in Chapter 5 and Chapter 6.

Now that you know the basics of object-oriented programming, you're ready for the next step. In Chapter 4, you'll take a tour of Visual Studio, and you'll use it to create an ASP.NET site of your own.

■ **Note** In the previous two chapters, you learned the essentials about C# and object-oriented programming. The C# language continues to evolve, and there are many more advanced language features that you haven't seen in these two chapters. If you want to continue your exploration of C# and become a language guru, you can visit Microsoft's C# Developer Center online at http://msdn.microsoft.com/vstudio/hh388566, or you can refer to a more in-depth book about C#, such as the excellent and very in-depth *Pro C# and the .NET 4.5 Framework* by Andrew Troelsen (Apress, 2012).
